PixelLib

Release 0.4.0

Jan 29, 2022

Contents:

PixelLib is a library created for performing image and video segmentation using few lines of code. It is a flexible library created to allow easy integration of image and video segmentation into software solutions.

PixelLib requires python's version 3.5-3.7, Download python

It requires pip's version >= 19.0

Install pip with:

```
pip3 install pip
```

Install PixelLib and its dependencies:

Install the latest version of tensorflow(Tensorflow 2.0+) with:

```
pip3 install tensorflow
```

Install imgaug with:

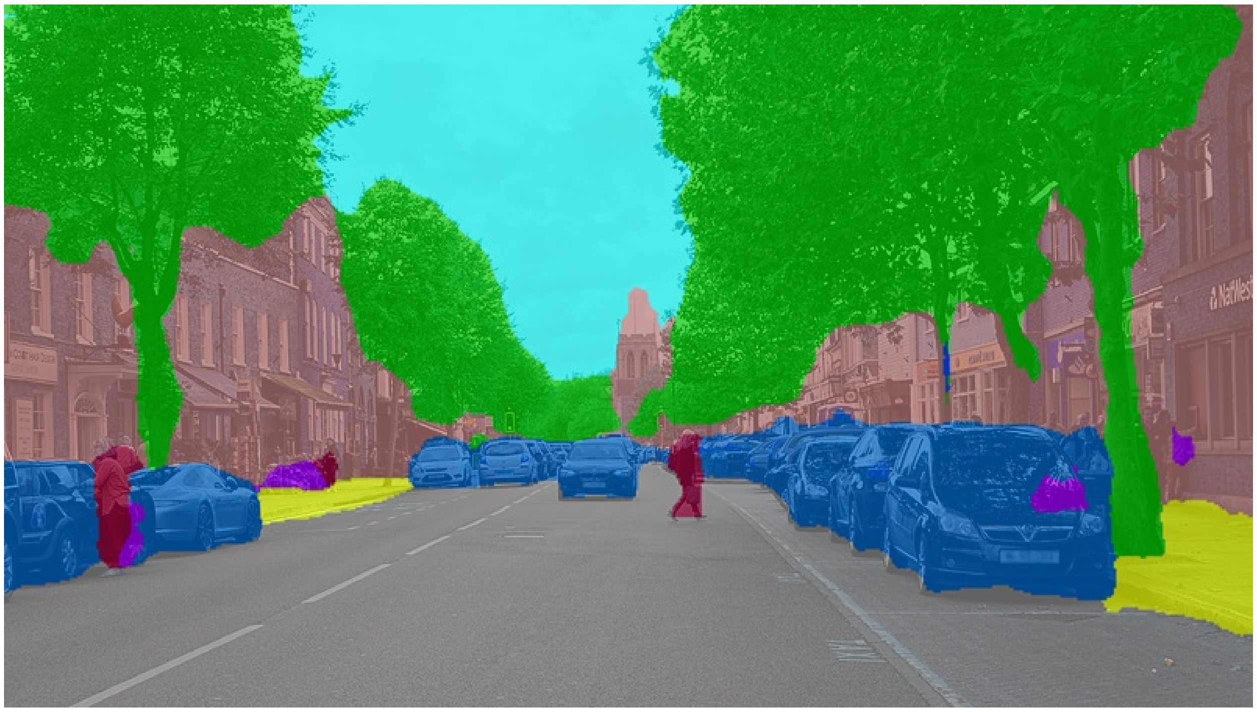
```
pip3 install imgaug
```

Install PixelLib with:

```
pip3 install pixellib --upgrade
```

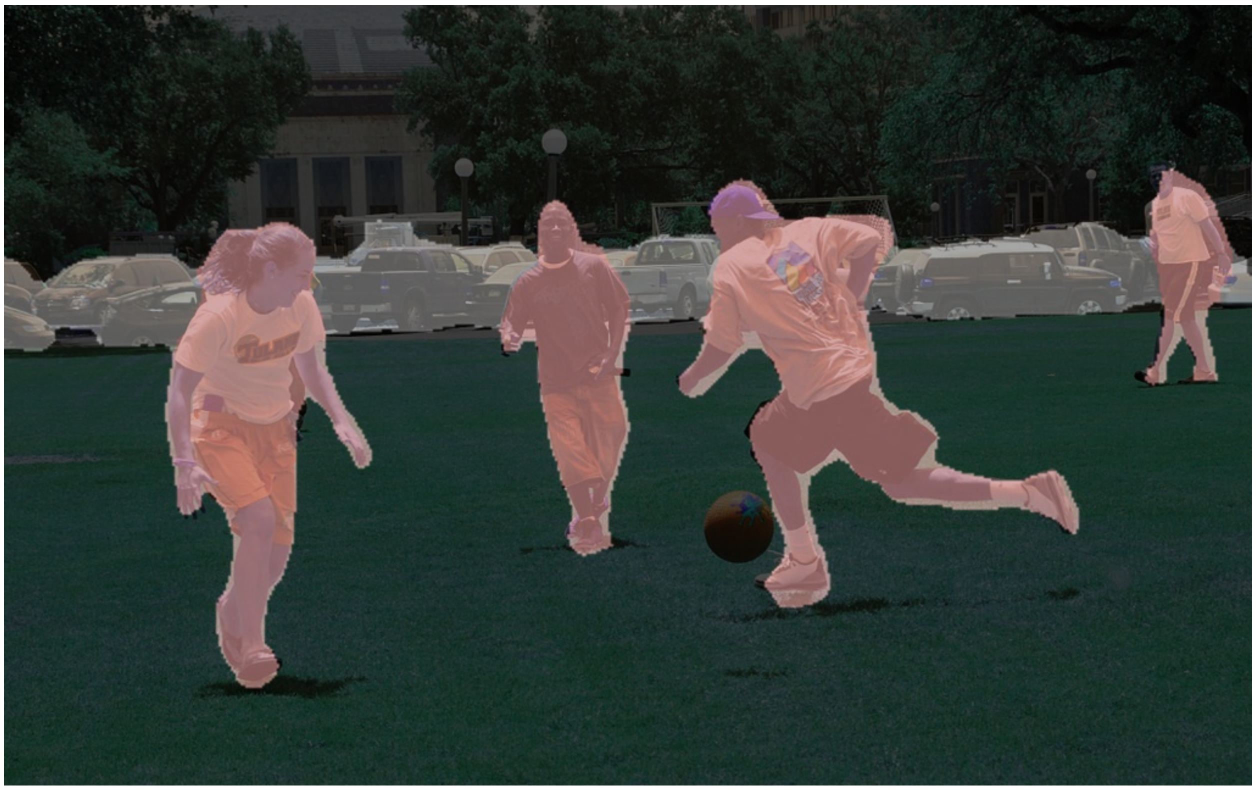
PixelLib supports the two major types of segmentation and you can create a custom model for objects' segmentation by training your dataset with PixelLib:

1 **Semantic segmentation**: Objects in an image with the same pixel values are segmented with the same colormaps.



Semantic segmentation of images with PixelLib using Ade20k model

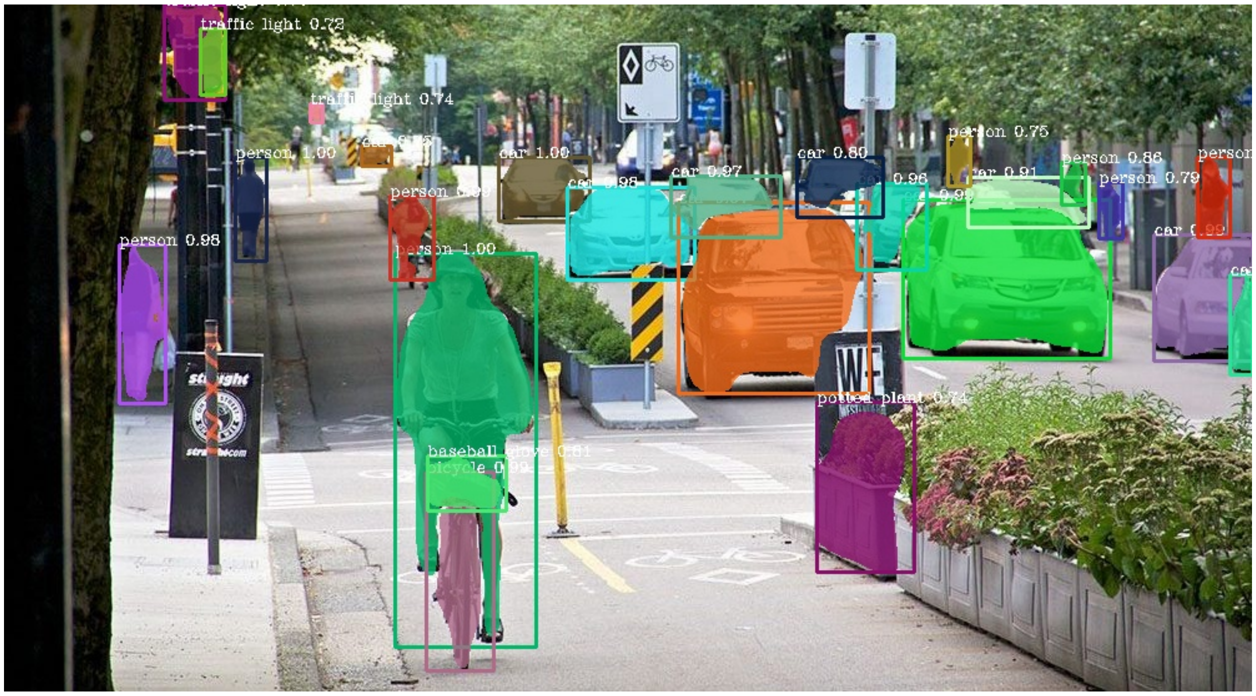
Semantic segmentation of videos with PixelLib using Ade20k model



Semantic segmentation of images with PixelLib using Pascalvoc model

Semantic Segmentation of videos with PixelLib using Pascalvoc model

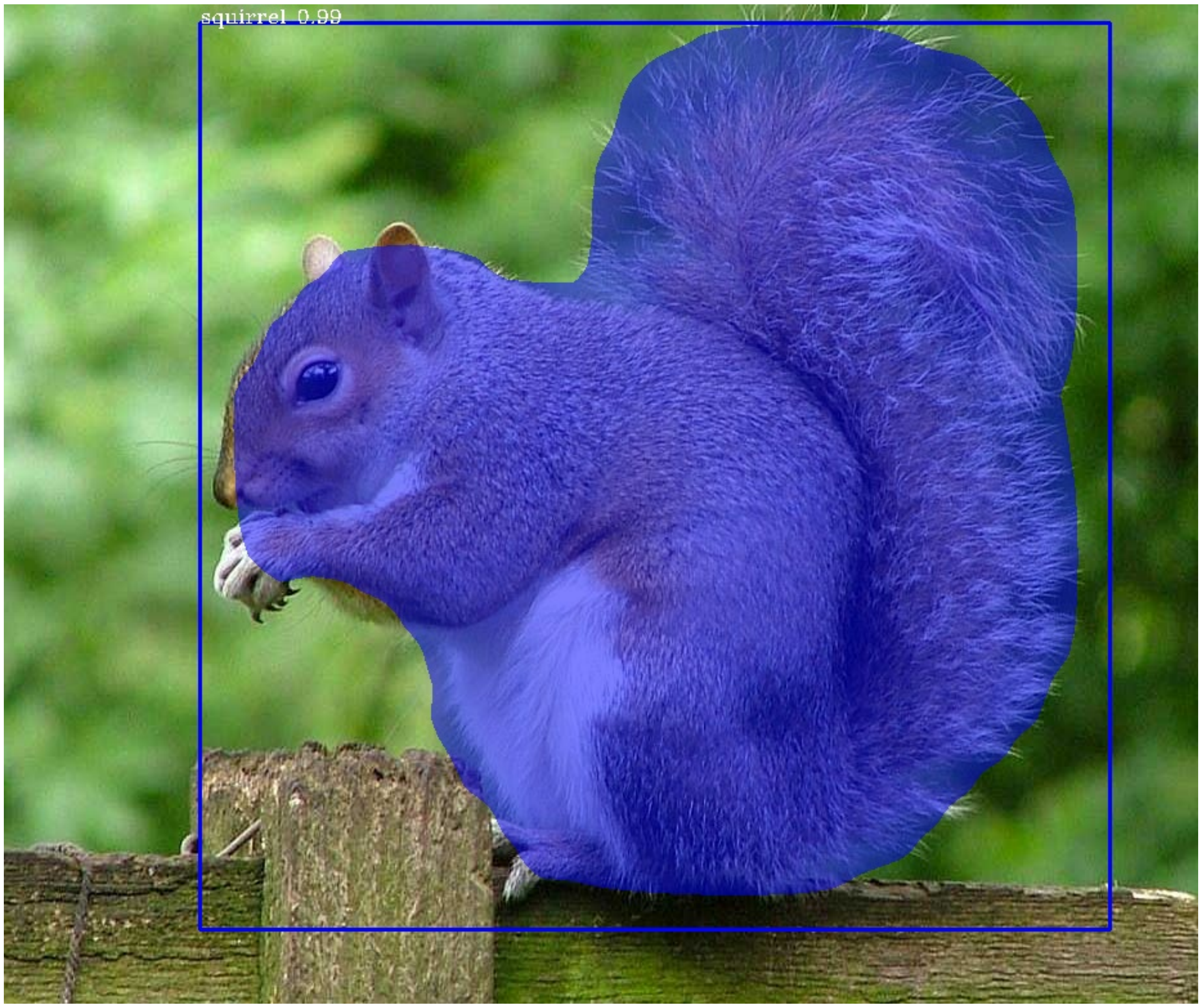
2 **Instance segmentation**: Instances of the same object are segmented with different color maps.



Instance segmentation of images with PixelLib

Instance segmentation of videos with PixelLib

3 **Implement Instance Segmentation And Object Detection On Objects By Training Your Dataset..**



Custom Training With PixelLib

Inference With A Custom Model Trained With PixelLib

Inference With A Custom Model

Implement background editing in images and videos using five lines of code. These are the features supported for background editing.

Change the background of an image with a picture

Assign a distinct color to the background of an image

Grayscale the background of an image

Blur the background of an image

Change the background of an Image

CHAPTER 1

Semantic segmentation of images with PixelLib using Ade20k model

PixelLib is implemented with Deeplabv3+ framework to perform semantic segmentation. Xception model trained on ade20k dataset is used for semantic segmentation.

Download the xception model from here.

Code to implement semantic segmentation:

```
import pixellib
from pixellib.semantic import semantic_segmentation

segment_video = semantic_segmentation()
segment_video.load_ade20k_model("deeplabv3_xception65_ade20k.h5")
segment_image.segmentAsAde20k("path_to_image", output_image_name= "path_to_output_
↪image")
```

We shall take a look into each line of code.

```
import pixellib
from pixellib.semantic import semantic_segmentation

#created an instance of semantic segmentation class
segment_image = semantic_segmentation()
```

The class for performing semantic segmentation is imported from pixellib and we created an instance of the class.

```
segment_video.load_ade20k_model("deeplabv3_xception65_ade20k.h5")
```

We called the function to load the xception model trained on ade20k dataset.

```
segment_image.segmentAsAde20k("path_to_image", output_image_name= "path_to_output_
↪image")
```

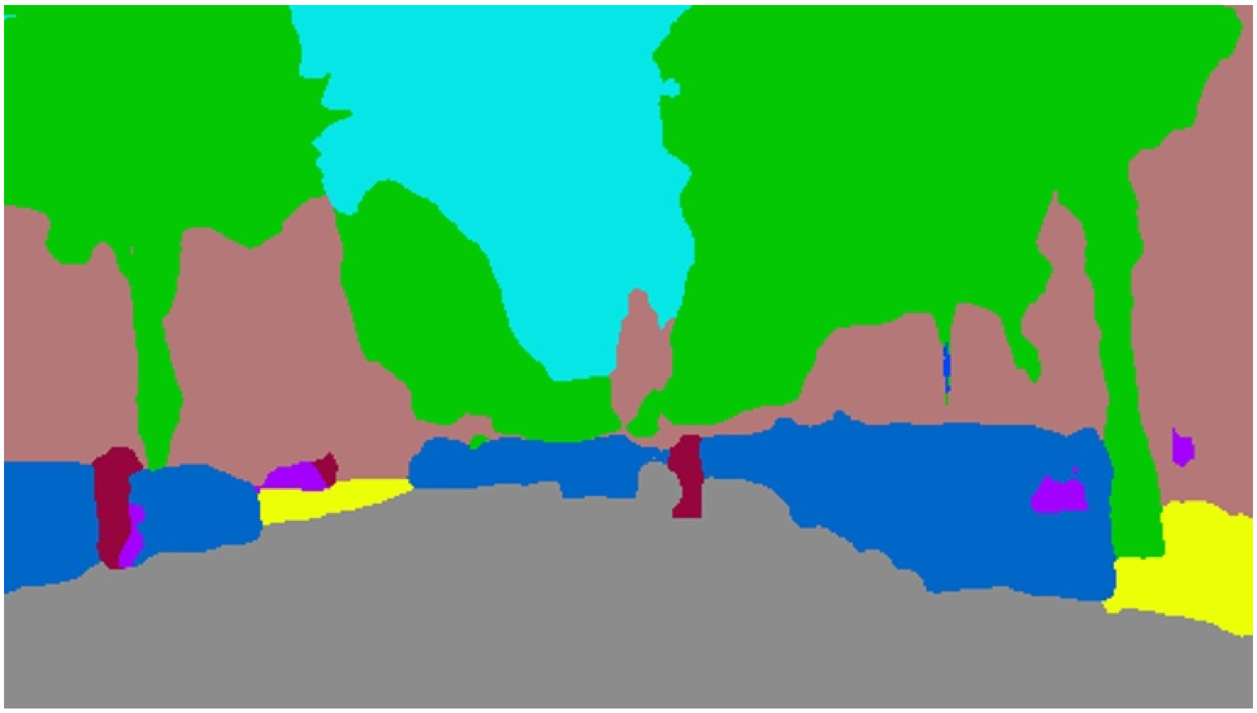
This is the line of code that performs segmentation on an image and the segmentation is done in the pascalvoc's color format. This function takes in two parameters:

path_to_image: the path to the image to be segemented.

path_to_output_image: the path to save the output image. The image will be saved in your current working directory.

Sample1.jpg

```
import pixellib
from pixellib.semantic import semantic_segmentation

segment_video = semantic_segmentation()
segment_video.load_ade20k_model("deeplabv3_xception65_ade20k.h5")
segment_video.segmentAsAde20k("sample1.jpg", output_image_name="image_new.jpg")
```

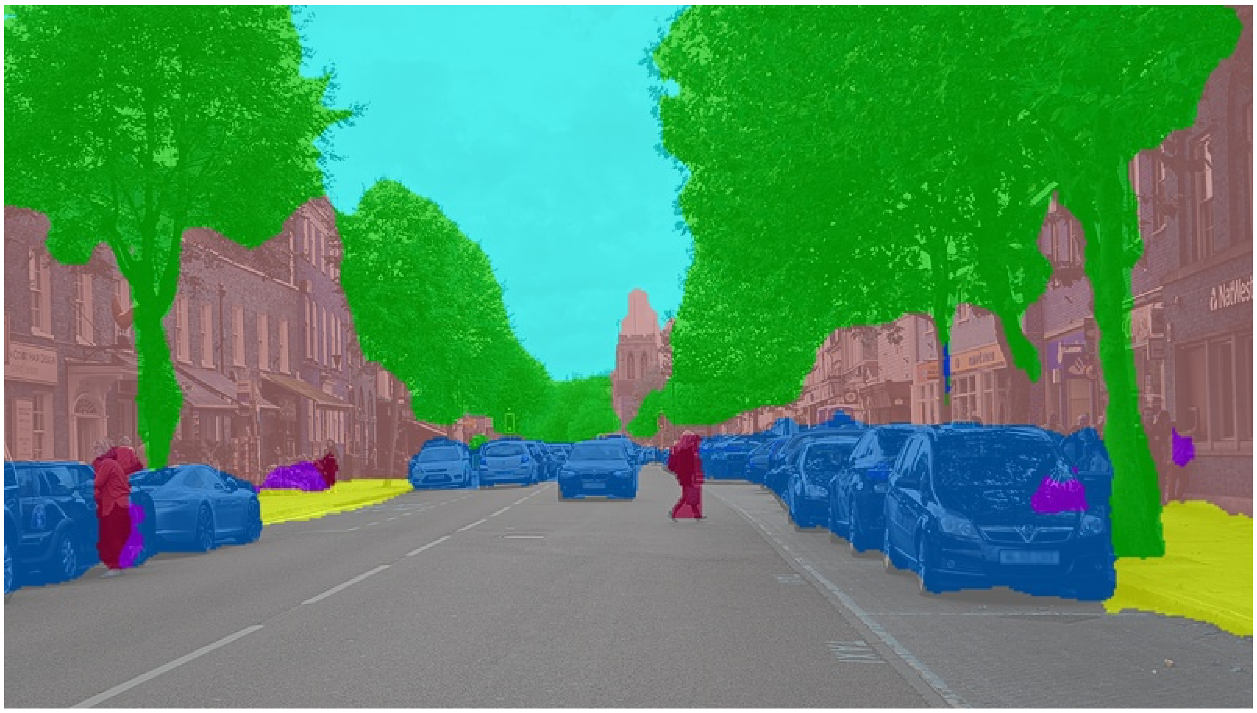


Your saved image with all the objects present segmented.

You can obtain an image with segmentation overlay on the objects with a modified code below.

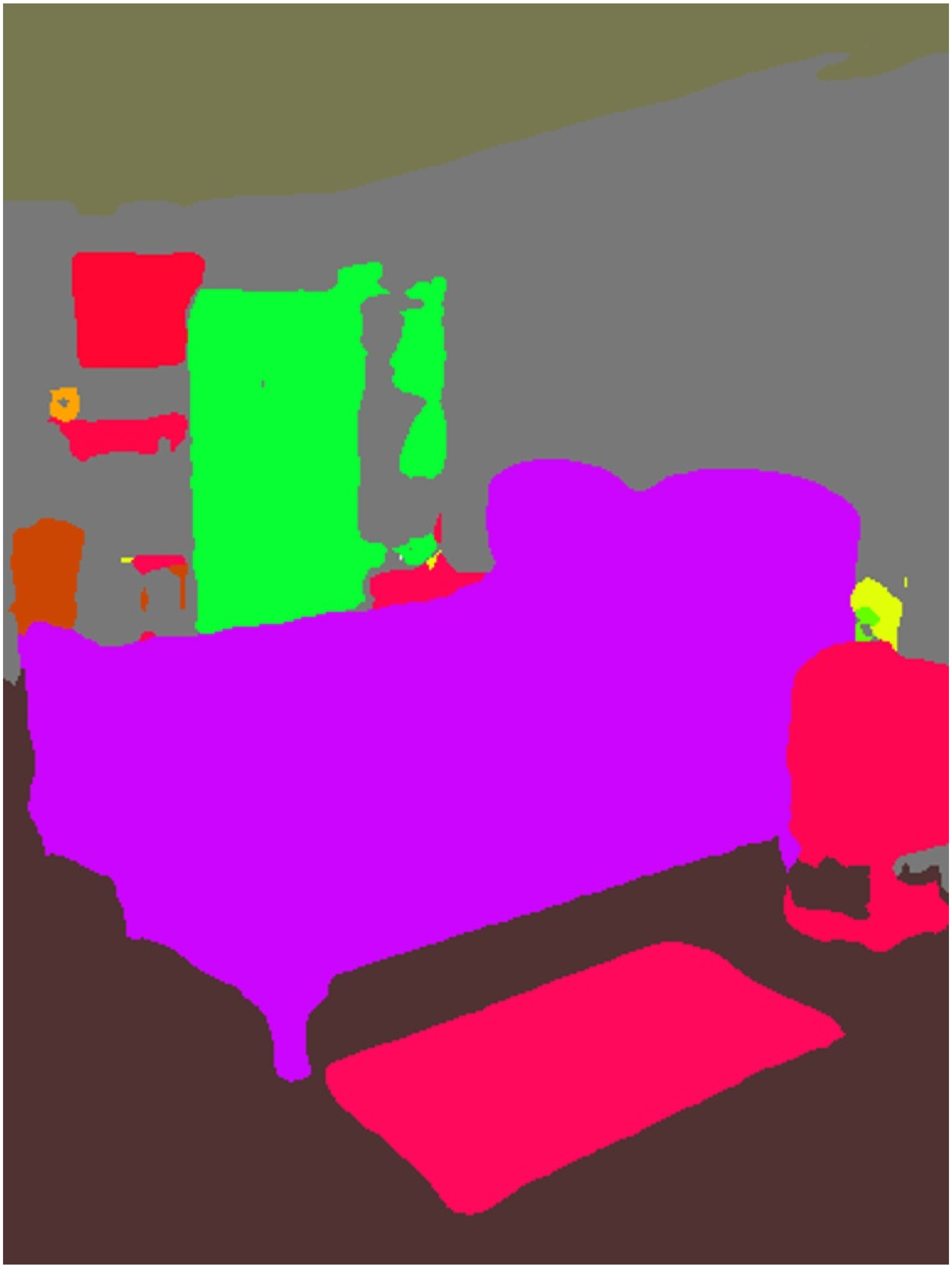
```
segment_image.segmentAsAde20k("sample1.jpg", output_image_name = "image_new.jpg",␣
↪overlay = True)
```

We added an extra parameter **overlay** and set it to **true**, we produced an image with segmentation overlay.



Sample2.jpg

```
segment_video.segmentAsAde20k("sample2.jpg", output_image_name="image_new2.jpg")
```



Specialised uses of PixelLib may require you to return the array of the segmentation's output.

- Obtain the array of the segmentation's output by using this code,

```
segvalues, output = segment_image.segmentAsAde20k()
```

- You can test the code for obtaining arrays and print out the shape of the output by modifying the semantic segmentation code below.

```
import pixellib
from pixellib.semantic import semantic_segmentation
import cv2

segment_image = semantic_segmentation()
segment_image.load_ade20k_model("deeplabv3_xception65_ade20k.h5")
segvalues, output = segment_image.segmentAsAde20k("sample2.jpg")
cv2.imwrite("img.jpg", output)
print(output.shape)
```

Note: Access the *masks* of the objects segmented using **segvalues["masks"]** and their *class ids* using **segvalues["class_ids"]**.

- Obtain both the output and the segmentation overlay's arrays by using this code,

```
segvalues, segoverlay = segment_image.segmentAsAde20k(overlay = True)
```

```
import pixellib
from pixellib.semantic import semantic_segmentation
import cv2

segment_image = semantic_segmentation()
segment_image.load_ade20k_model("deeplabv3_xception65_ade20k.h5")
segvalues, segoverlay = segment_image.segmentAsAde20k("sample2.jpg", overlay= True)
cv2.imwrite("img.jpg", segoverlay)
print(segoverlay.shape)
```

This xception model is trained on ade20k dataset, a dataset with 150 object categories.

Process opencv's frames

```
import pixellib
from pixellib.semantic import semantic_segmentation
import cv2

segment_frame = semantic_segmentation()
segment_frame.load_ade20k_model("deeplabv3_xception65_ade20k.h5")

capture = cv2.VideoCapture(0)
while True:
  ret, frame = capture.read()
  segment_video.segmentFrameAsAde20k(frame)
```

CHAPTER 2

Semantic segmentation of videos with PixelLib using Ade20k model

PixelLib is implemented with Deeplabv3+ framework to perform semantic segmentation. Xception model trained on ade20k dataset is used for semantic segmentation.

Download the xception model from here.

Code to implement semantic segmentation of a video with ade20k model:

```
import pixellib
from pixellib.semantic import semantic_segmentation

segment_video = semantic_segmentation()
segment_video.load_ade20k_model("deeplabv3_xception65_ade20k.h5")
segment_video.process_video_ade20k("video_path", frames_per_second= 15, output_video_
↪name="path_to_output_video")
```

We shall take a look into each line of code.

```
import pixellib
from pixellib.semantic import semantic_segmentation

#created an instance of semantic segmentation class
segment_image = semantic_segmentation()
```

The class for performing semantic segmentation is imported from pixellib and we created an instance of the class.

```
segment_video.load_ade20k_model("deeplabv3_xception65_ade20k.h5")
```

We called the function to load the xception model trained on ade20k.

```
segment_video.process_video_ade20k("video_path", frames_per_second= 15, output_video_
↪name="path_to_output_video")
```

This is the line of code that performs segmentation on an image and the segmentation is done in the ade20k's color format. This function takes in two parameters:

video_path: the path to the video file we want to perform segmentation on.

frames_per_second: this is parameter to set the number of frames per second for the output video file. In this case it is set to 15 i.e the saved video file will have 15 frames per second.

output_video_name: the saved segmented video. The output video will be saved in your current working directory.

sample_video

```
import pixellib
from pixellib.semantic import semantic_segmentation

segment_video = semantic_segmentation()
segment_video.load_ade20k_model("deeplabv3_xception65_ade20k.h5")
segment_video.process_video_ade20k("sample_video.mp4", overlay = True, frames_per_
↪second= 15, output_video_name="output_video.mp4")
```

Output video

CHAPTER 3

Segmentation of live camera with Ade20k model

We can use the same model to perform semantic segmentation on camera. This can be done by few modifications to the code to process video file.

```
import pixellib
from pixellib.semantic import semantic_segmentation
import cv2


capture = cv2.VideoCapture(0)

segment_video = semantic_segmentation()
segment_video.load_ade20k_model("deeplabv3_xception65_ade20k.h5")
segment_video.process_camera_ade20k(capture, overlay=True, frames_per_second= 15,␣
↪output_video_name="output_video.mp4", show_frames= True,
frame_name= "frame")
```

```
import cv2
capture = cv2.VideoCapture(0)
```

We imported cv2 and included the code to capture camera frames.

```
segment_video.process_camera_ade20k(capture,  overlay = True, frames_per_second= 15,␣
↪output_video_name="output_video.mp4", show_frames= True,frame_name= "frame")
```

In the code for performing segmentation, we replaced the video filepath to capture i.e we are going to process a stream camera frames instead of a video file.We added extra parameters for the purpose of showing the camera frames:

show_frames: this parameter handles showing of segmented camera frames and press q to exist. *frame_name:* this is the name given to the shown camera's frame.

A demo showing the output of pixelLib's semantic segmentation of camera's feeds using pascal voc model. *Good work! It was able to successfully segment me.*

CHAPTER 4

Semantic segmentation of images with PixelLib using Pascalvoc model

PixelLib is implemented with Deeplabv3+ framework to perform semantic segmentation. Xception model trained on pascalvoc dataset is used for semantic segmentation.

Download the xception model from here.

Code to implement semantic segmentation:

```
import pixellib
from pixellib.semantic import semantic_segmentation

segment_image = semantic_segmentation()
segment_image.load_pascalvoc_model("deeplabv3_xception_tf_dim_ordering_tf_kernels.h5")
segment_image.segmentAsPascalvoc("path_to_image", output_image_name = "path_to_output_
↪image")
```

We shall take a look into each line of code.

```
import pixellib
from pixellib.semantic import semantic_segmentation

#created an instance of semantic segmentation class
segment_image = semantic_segmentation()
```

The class for performing semantic segmentation is imported from pixellib and we created an instance of the class.

```
segment_image.load_pascalvoc_model("deeplabv3_xception_tf_dim_ordering_tf_kernels.h5")
```

We called the function to load the xception model trained on pascal voc.

```
segment_image.segmentAsPascalvoc("path_to_image", output_image_name = "path_to_output_
↪image")
```

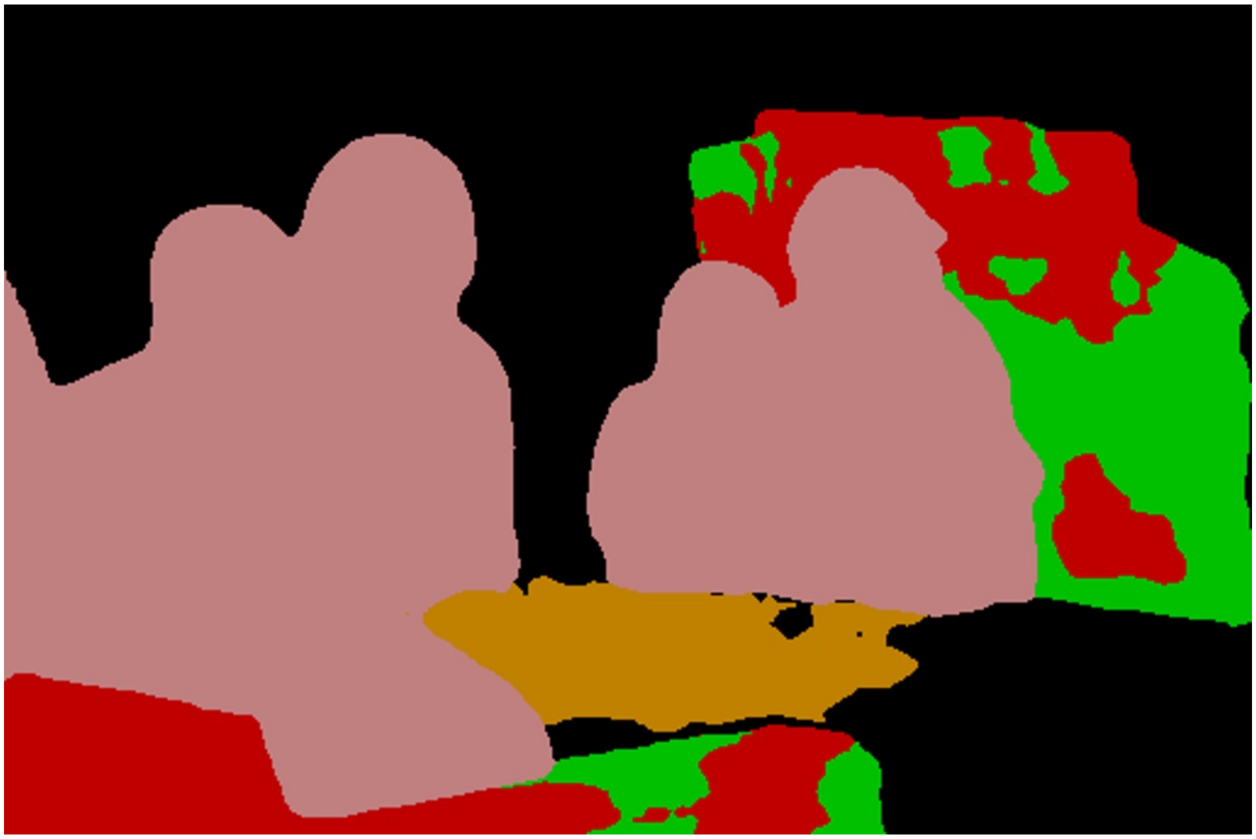
This is the line of code that performs segmentation on an image and the segmentation is done in the pascalvoc's color format. This function takes in two parameters:

path_to_image: the path to the image to be segemented.

path_to_output_image: the path to save the output image. The image will be saved in your current working directory.

Sample1.jpg

```
import pixellib
from pixellib.semantic import semantic_segmentation

segment_image = semantic_segmentation()
segment_image.load_pascalvoc_model("deeplabv3_xception_tf_dim_ordering_tf_kernels.h5")
segment_image.segmentAsPascalvoc("sample1.jpg", output_image_name = "image_new.jpg")
```

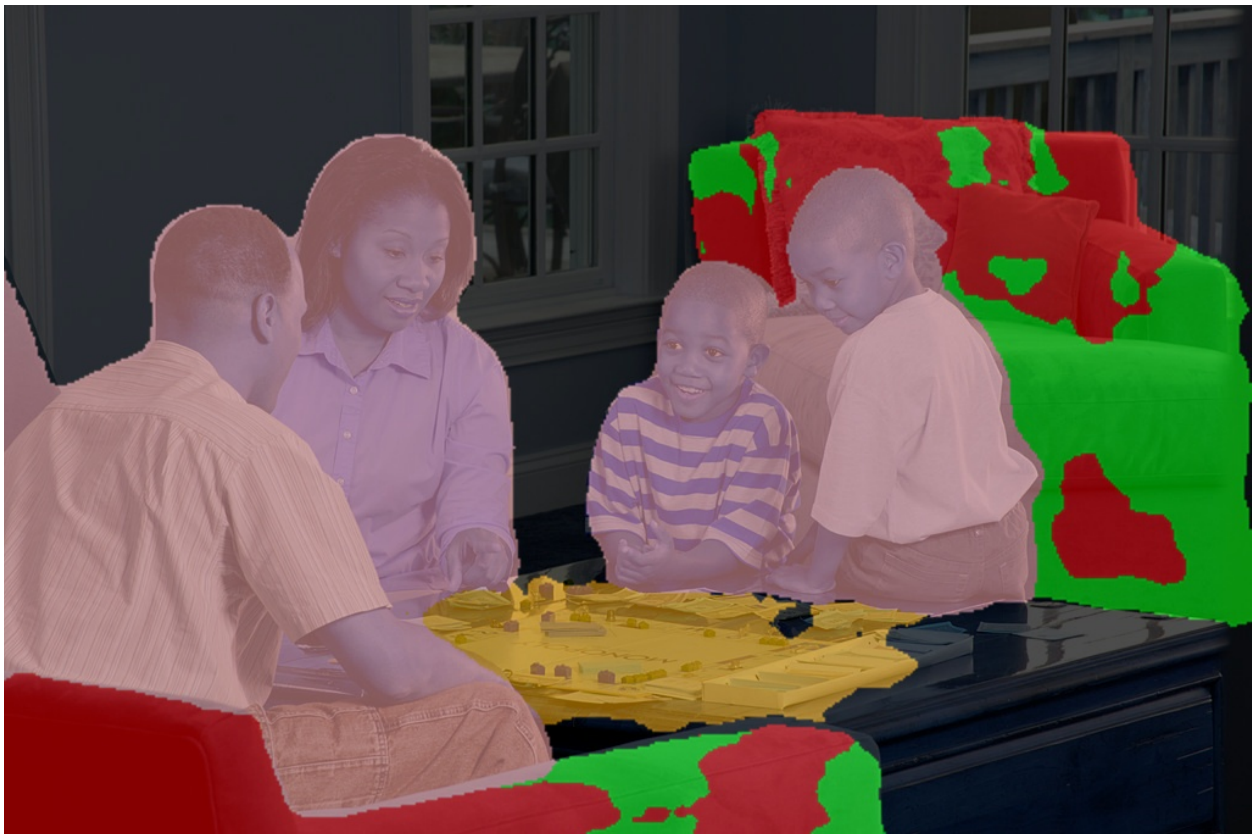


Your saved image with all the objects present segmented.

You can obtain an image with segmentation overlay on the objects with a modified code below.

```
segment_image.segmentAsPascalvoc("sample1.jpg", output_image_name = "image_new.jpg",␣
↪overlay = True)
```

We added an extra parameter **overlay** and set it to **true**, we produced an image with segmentation overlay.



Specialised uses of PixelLib may require you to return the array of the segmentation's output.

- Obtain the array of the segmentation's output by using this code,

```
segvalues, output = segment_image.segmentAsPascalvoc()
```

- You can test the code for obtaining arrays and print out the shape of the output by modifying the semantic segmentation code below.

```
import pixellib
from pixellib.semantic import semantic_segmentation
import cv2

segment_image = semantic_segmentation()
segment_image.load_pascalvoc_model("deeplabv3_xception_tf_dim_ordering_tf_kernels.h5")
segvalues, output = segment_image.segmentAsPascalvoc("sample1.jpg")
cv2.imwrite("img.jpg", output)
print(output.shape)
```

Note: Access the *masks* of the objects segmented using **segvalues["masks"]** and their *class ids* using **segvalues["class_ids"]**.

- Obtain both the output and the segmentation overlay's arrays by using this code,

```
segvalues, segoverlay = segment_image.segmentAsPascalvoc(overlay = True)
```

```
import pixellib
from pixellib.semantic import semantic_segmentation
import cv2
```

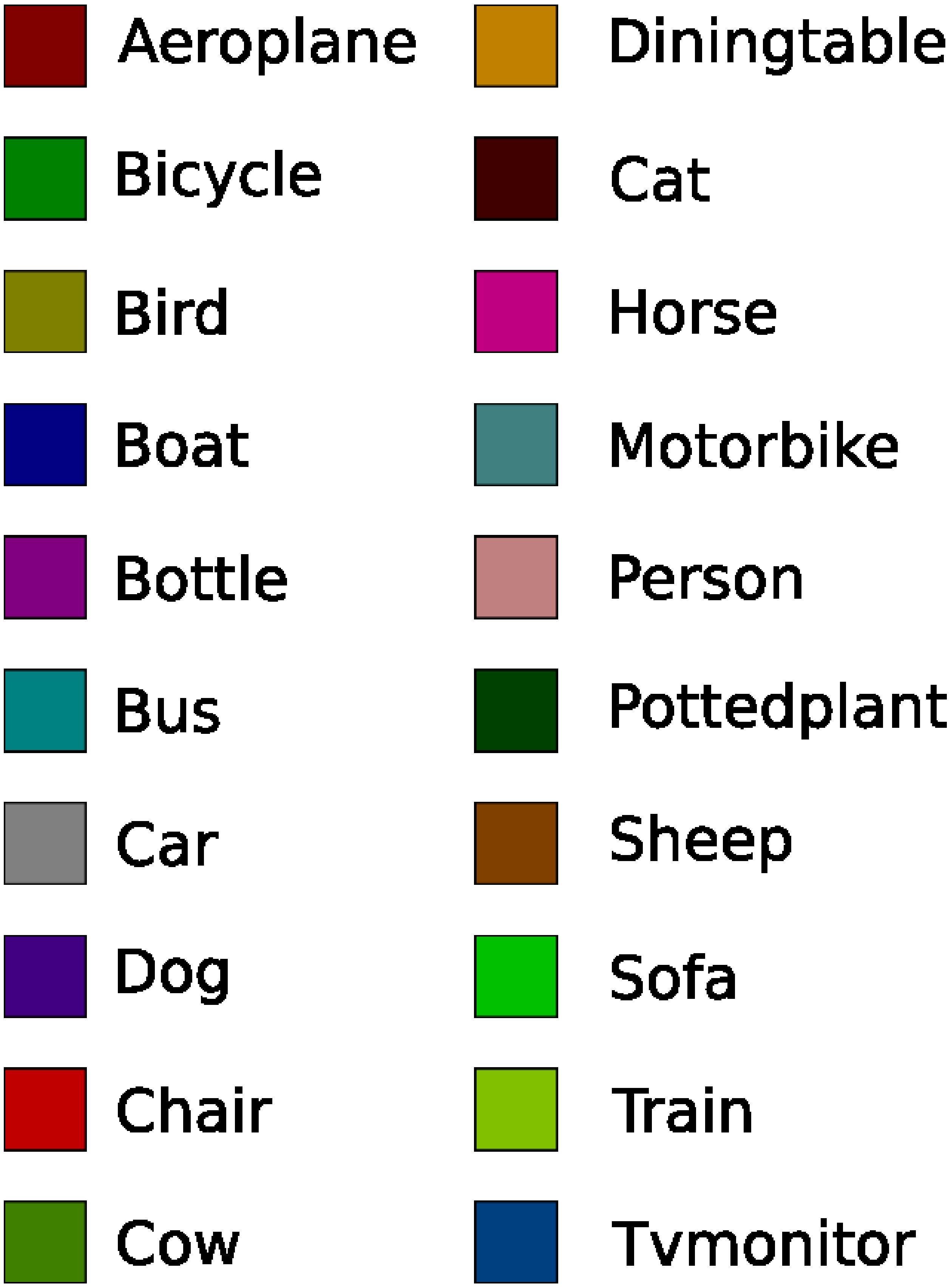
(continues on next page)

(continued from previous page)

```
segment_image = semantic_segmentation()
segment_image.load_pascalvoc_model("deeplabv3_xception_tf_dim_ordering_tf_kernels.h5")
segvalues, segoverlay = segment_image.segmentAsPascalvoc("sample1.jpg", overlay= True)
cv2.imwrite("img.jpg", segoverlay)
print(segoverlay.shape)
```

This xception model is trained on pascal voc dataset, a dataset with 20 object categories.

Objects and their corresponding colormaps.



🟥	Aeroplane	🟧	Diningtable
🟩	Bicycle	🟫	Cat
🟨	Bird	🟪	Horse
🟦	Boat	🟦	Motorbike
🟪	Bottle	🟥	Person
🟦	Bus	🟩	Pottedplant
⬜	Car	🟫	Sheep
🟪	Dog	🟩	Sofa
🟥	Chair	🟩	Train
🟩	Cow	🟦	Tvmonitor

Process opencv's frames

```python
import pixellib
from pixellib.semantic import semantic_segmentation
import cv2

segment_frame = semantic_segmentation()
segment_frame.load_pascalvoc_model("deeplabv3_xception_tf_dim_ordering_tf_kernels.h5")

capture = cv2.VideoCapture(0)
while True:
  ret, frame = capture.read()
  segment_video.segmentFrameAsPascalvoc(frame, output_image_name= "hi.jpg")
```

CHAPTER 5

Semantic Segmentation of videos with PixelLib using Pascalvoc model

PixelLib is implemented with Deeplabv3+ framework to perform semantic segmentation. Xception model trained on pascalvoc dataset is used for semantic segmentation.

Download the xception model from here.

Code to implement semantic segmentation of a video with pascalvoc model:

```
import pixellib
from pixellib.semantic import semantic_segmentation

segment_video = semantic_segmentation()
segment_video.load_pascalvoc_model("deeplabv3_xception_tf_dim_ordering_tf_kernels.h5")
segment_video.process_video_pascalvoc("video_path",  overlay = True, frames_per_
↪second= 15, output_video_name="path_to_output_video")
```

We shall take a look into each line of code.

```
import pixellib
from pixellib.semantic import semantic_segmentation

#created an instance of semantic segmentation class
segment_image = semantic_segmentation()
```

The class for performing semantic segmentation is imported from pixellib and we created an instance of the class.

```
segment_image.load_pascalvoc_model("deeplabv3_xception_tf_dim_ordering_tf_kernels.h5")
```

We called the function to load the xception model trained on pascal voc.

```
segment_video.process_video_pascalvoc("video_path",  overlay = True, frames_per_
↪second= 15, output_video_name="path_to_output_video")
```

This is the line of code that performs segmentation on an image and the segmentation is done in the pascalvoc's color format. This function takes in two parameters:

video_path: the path to the video file we want to perform segmentation on.

frames_per_second: this is parameter to set the number of frames per second for the output video file. In this case it is set to 15 i.e the saved video file will have 15 frames per second.

output_video_name: the saved segmented video. The output video will be saved in your current working directory.

sample_video

```
import pixellib
from pixellib.semantic import semantic_segmentation

segment_video = semantic_segmentation()
segment_video.load_pascalvoc_model("deeplabv3_xception_tf_dim_ordering_tf_kernels.h5")
segment_video.process_video_pascalvoc("sample_video1.mp4",  overlay = True, frames_
↪per_second= 15, output_video_name="output_video.mp4")
```

Output video

This is a saved segmented video using pascal voc model.

CHAPTER 6

Segmentation of live camera with pascalvoc model

We can use the same model to perform semantic segmentation on camera. This can be done by few modifications to the code to process video file.

```
import pixellib
from pixellib.semantic import semantic_segmentation
import cv2


capture = cv2.VideoCapture(0)

segment_video = semantic_segmentation()
segment_video.load_pascalvoc_model("deeplabv3_xception_tf_dim_ordering_tf_kernels.h5")
segment_video.process_camera_pascalvoc(capture,  overlay = True, frames_per_second=
↪15, output_video_name="output_video.mp4", show_frames= True,
frame_name= "frame")
```

We imported cv2 and included the code to capture camera's frames.

```
segment_video.process_camera_pascalvoc(capture,  overlay = True, frames_per_second=
↪15, output_video_name="output_video.mp4", show_frames= True,frame_name= "frame")
```

In the code for performing segmentation, we replaced the video's filepath to capture i.e we are going to process a stream camera's frames instead of a video file.We added extra parameters for the purpose of showing the camera frames:

show_frames: this parameter handles showing of segmented camera frames and press q to exist. *frame_name:* this is the name given to the shown camera's frames.

A demo showing the output of pixelLib's semantic segmentation of camera's feeds using pascal voc model. *Good work! It was able to successfully segment me and the plastic bottle in front of me.*

CHAPTER 7

Instance segmentation of images with PixelLib

Instance segmentation with PixelLib is based on MaskRCNN framework.

Download the mask rcnn model from here

Code to implement instance segmentation:

```
import pixellib
from pixellib.instance import instance_segmentation

segment_image = instance_segmentation()
segment_image.load_model("mask_rcnn_coco.h5")
segment_image.segmentImage("path_to_image", output_image_name = "output_image_path")
```

Observing each line of code:

```
import pixellib
from pixellib.instance import instance_segmentation

segment_image = instance_segmentation()
```

The class for performing instance segmentation is imported and we created an instance of the class.

```
segment_image.load_model("mask_rcnn_coco.h5")
```

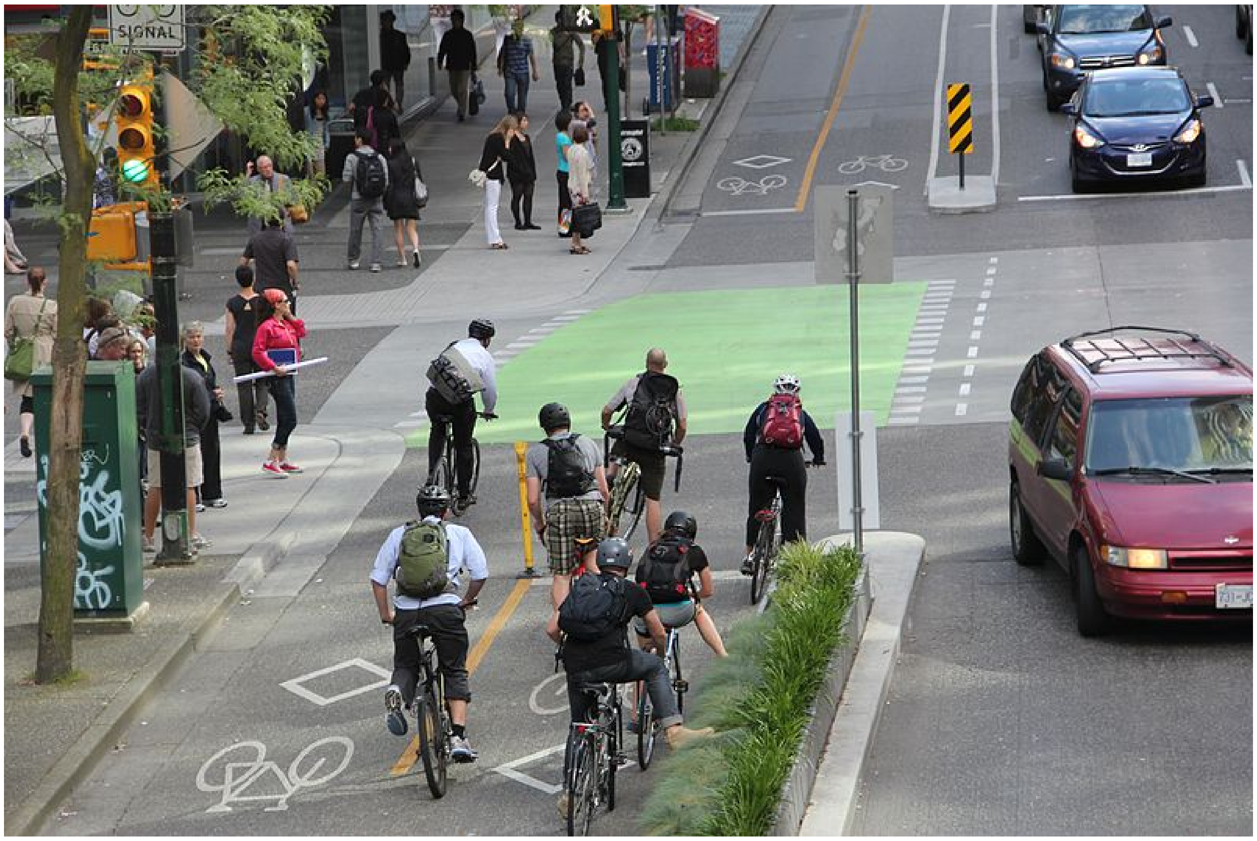
This is the code to load the mask rcnn model to perform instance segmentation.

```
segment_image.segmentImage("path_to_image", output_image_name = "output_image_path")
```

This is the code to perform instance segmentation on an image and it takes two parameters:

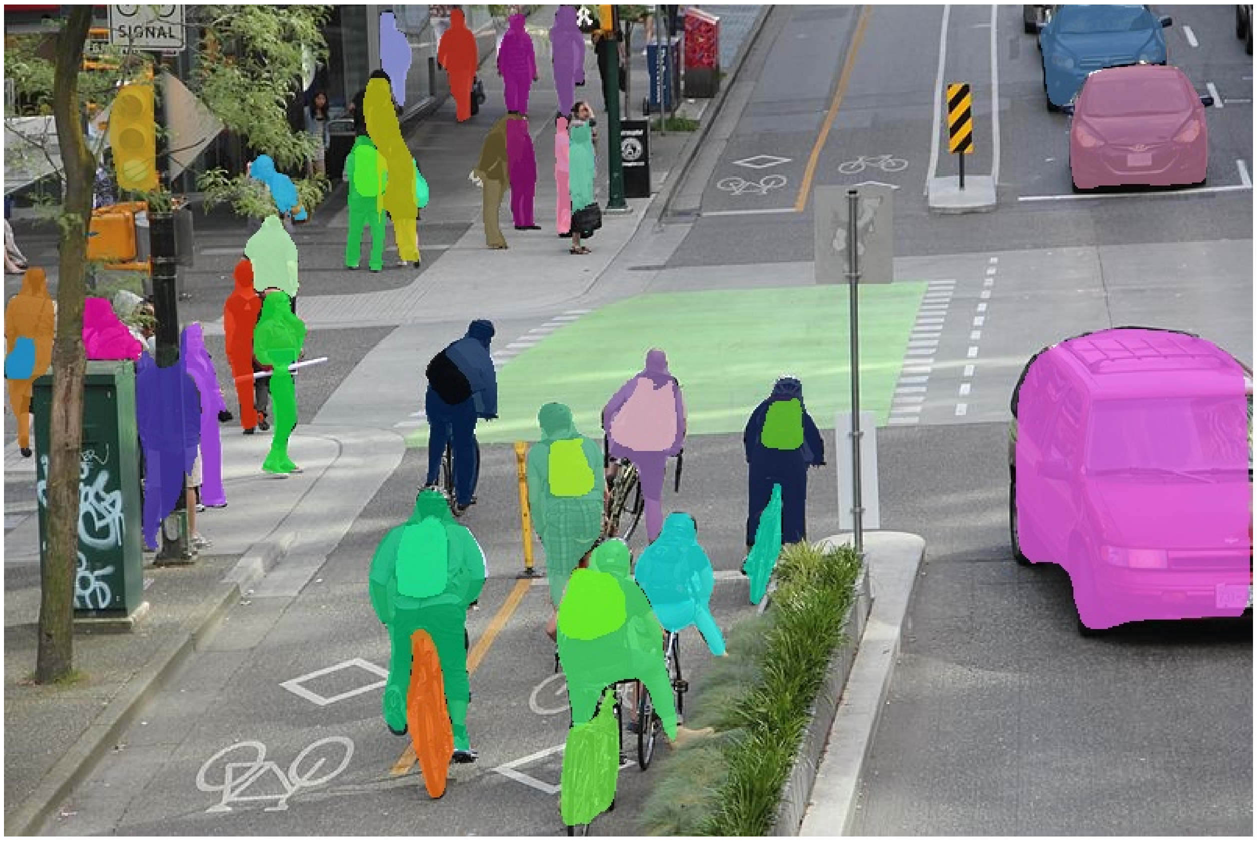
> *path_to_image*: The path to the image to be predicted by the model.
>
> *output_image_name*: The path to save the segmentation result. It will be saved in your current working directory.

Sample2.jpg



Image's source:Wikicommons

```python
import pixellib
from pixellib.instance import instance_segmentation

segment_image = instance_segmentation()
segment_image.load_model("mask_rcnn_coco.h5")
segment_image.segmentImage("sample2.jpg", output_image_name = "image_new.jpg")
```

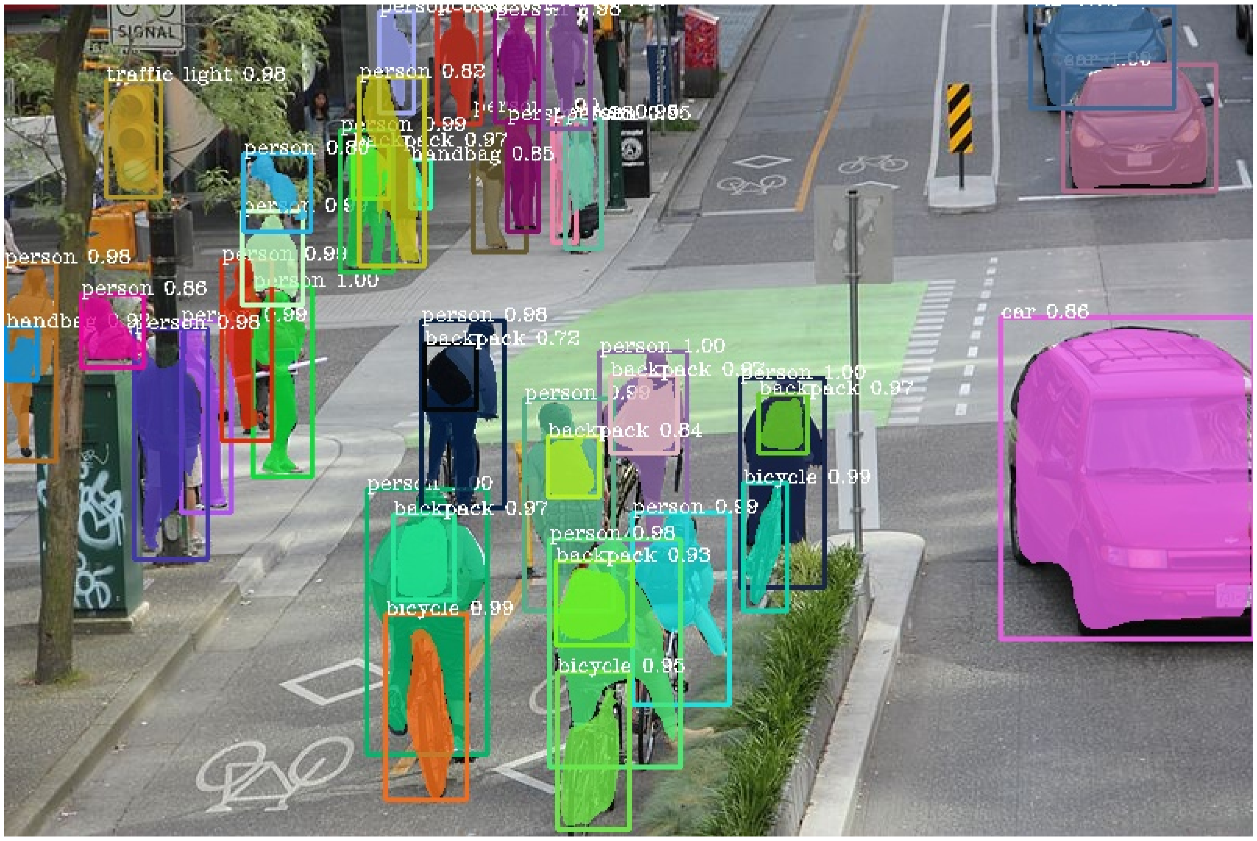


This is the saved image in your current working directory.

You can implement segmentation with bounding boxes. This can be achieved by modifying the code.

```
segment_image.segmentImage("sample2.jpg", output_image_name = "image_new.jpg", show_
↪bboxes = True)
```

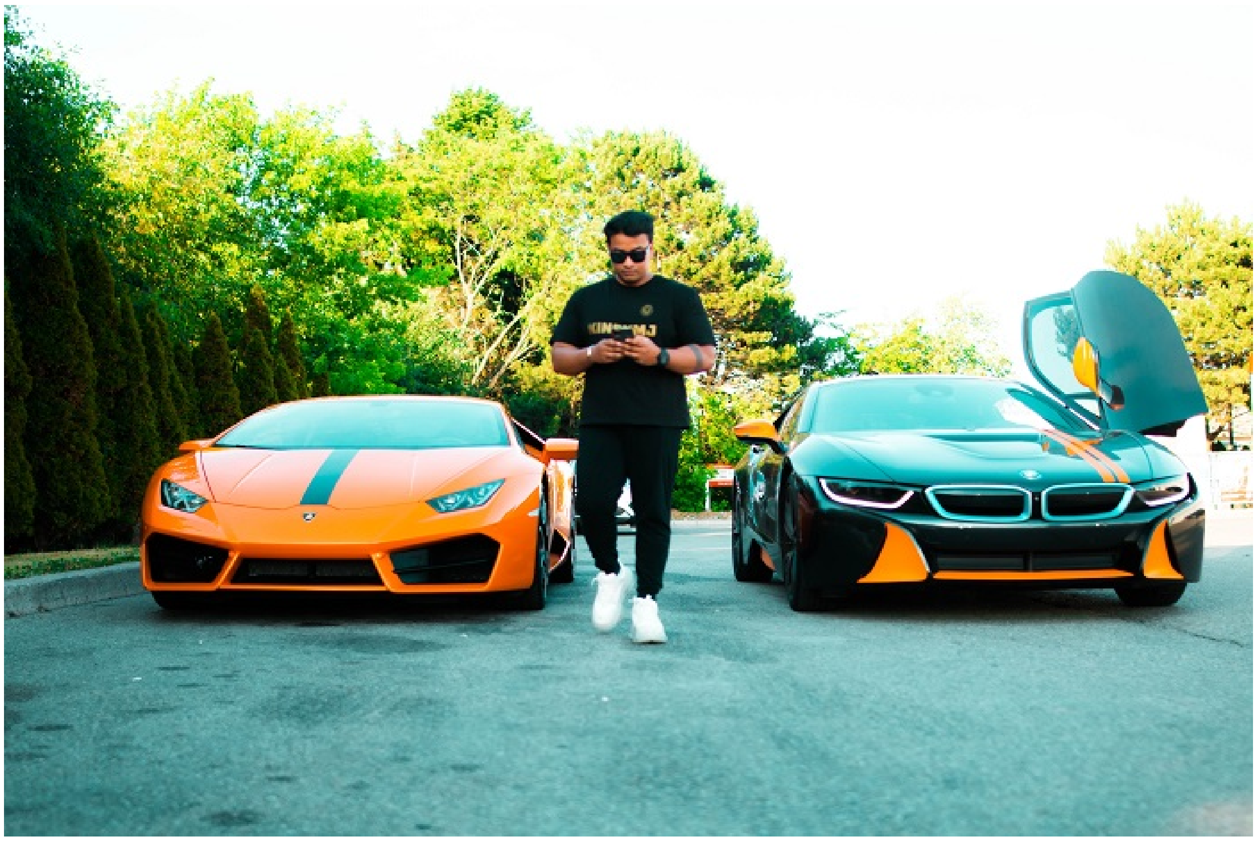
We added an extra parameter **show_bboxes** and set it to **true**, the segmentation masks are produced with bounding boxes.



You get a saved image with both segmentation masks and bounding boxes.

Extraction of Segmented Objects

PixelLib now makes it possible to extract each of the segmented objects in an image and save each of the object extracted as a separate image. This is the modified code below;



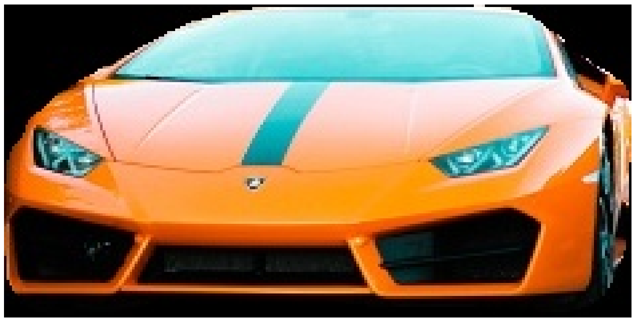
```
import pixellib
from pixellib.instance import instance_segmentation

seg = instance_segmentation()
seg.load_model("mask_rcnn_coco.h5")
seg.segmentImage("sample2.jpg", show_bboxes=True, output_image_name="output.jpg",
extract_segmented_objects= True, save_extracted_objects=True)
```

We introduced new parameters in the *segmentImage* function which are:

extract_segmented_objects: This parameter handles the extraction of each of the segmented object in the image.

save_extracted_objects: This parameter saves each of the extracted object as a separate image.Each of the object extracted in the image would be save with the name *segmented_object* with the corresponding index number such as *segmented_object_1*.

These are the objects extracted from the image above.



Detection of Target Classes

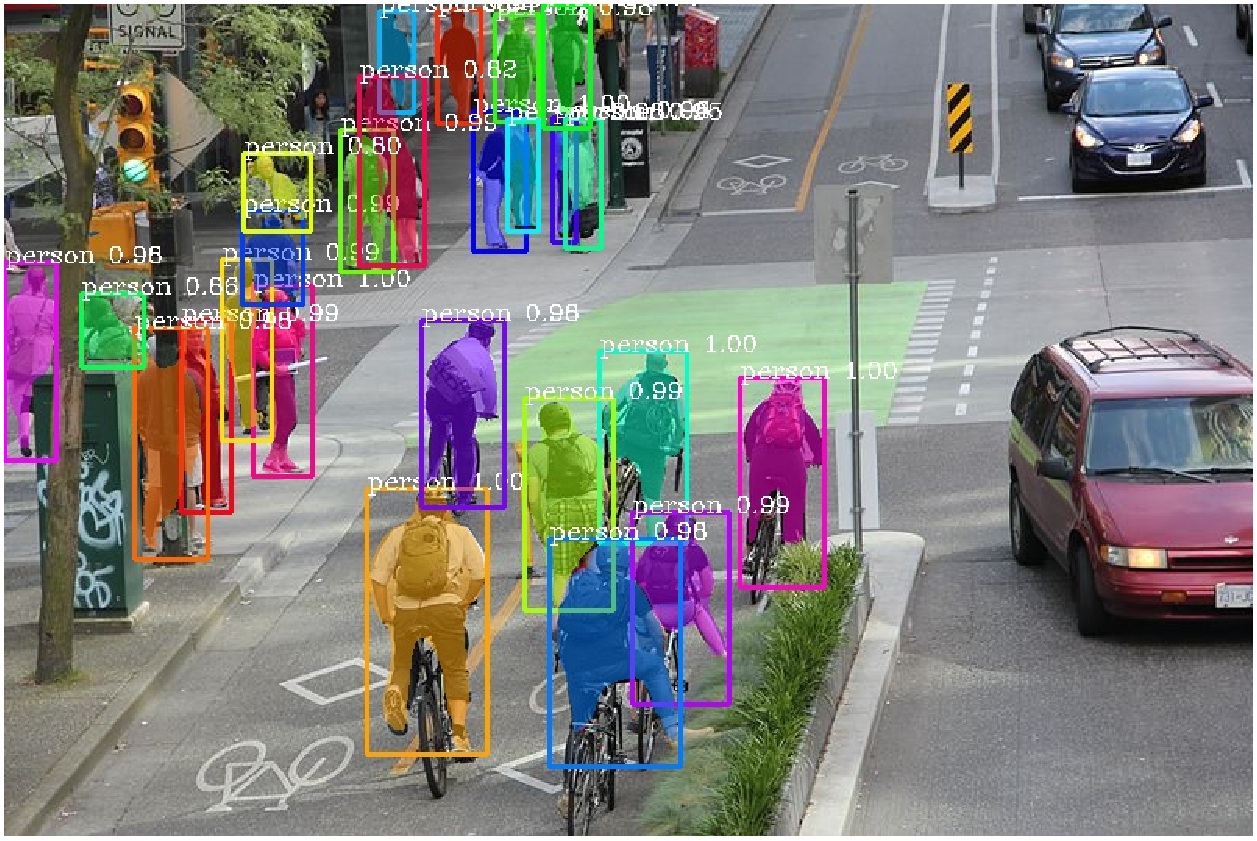
The pre-trained coco model used detects 80 classes of objects. PixelLib has made it possible to filter out unused detections and detect the classes you want.

Code to detect target classes

```
import pixellib
from pixellib.instance import instance_segmentation

seg = instance_segmentation()
seg.load_model("mask_rcnn_coco.h5")
target_classes = seg.select_target_classes(person=True)
seg.segmentImage("sample2.jpg", segment_target_classes= target_classes, show_
↪bboxes=True,  output_image_name="a.jpg")
```

```
target_classes = seg.select_target_classes(person=True)
seg.segmentImage("sample2.jpg", segment_target_classes= target_classes, show_
↪bboxes=True,  output_image_name="a.jpg")
```

We introduced a new function *select_target_classes* that determines the target class to be detected. In this case we want to detect only *person* in the image. In the function *segmentImage* we added a new parameter *segment_target_classes* to filter unused detections and detect only the target class.



Beautiful Result! We were able to filter other detections and detect only the people in the image.

Speed Adjustments for Faster Inference

PixelLib now supports the ability to adjust the speed of detection according to a user's needs. The inference speed with a minimal reduction in the accuracy of detection. There are three main parameters that control the speed of detection.

1 **average**

2 **fast**

3 **rapid**

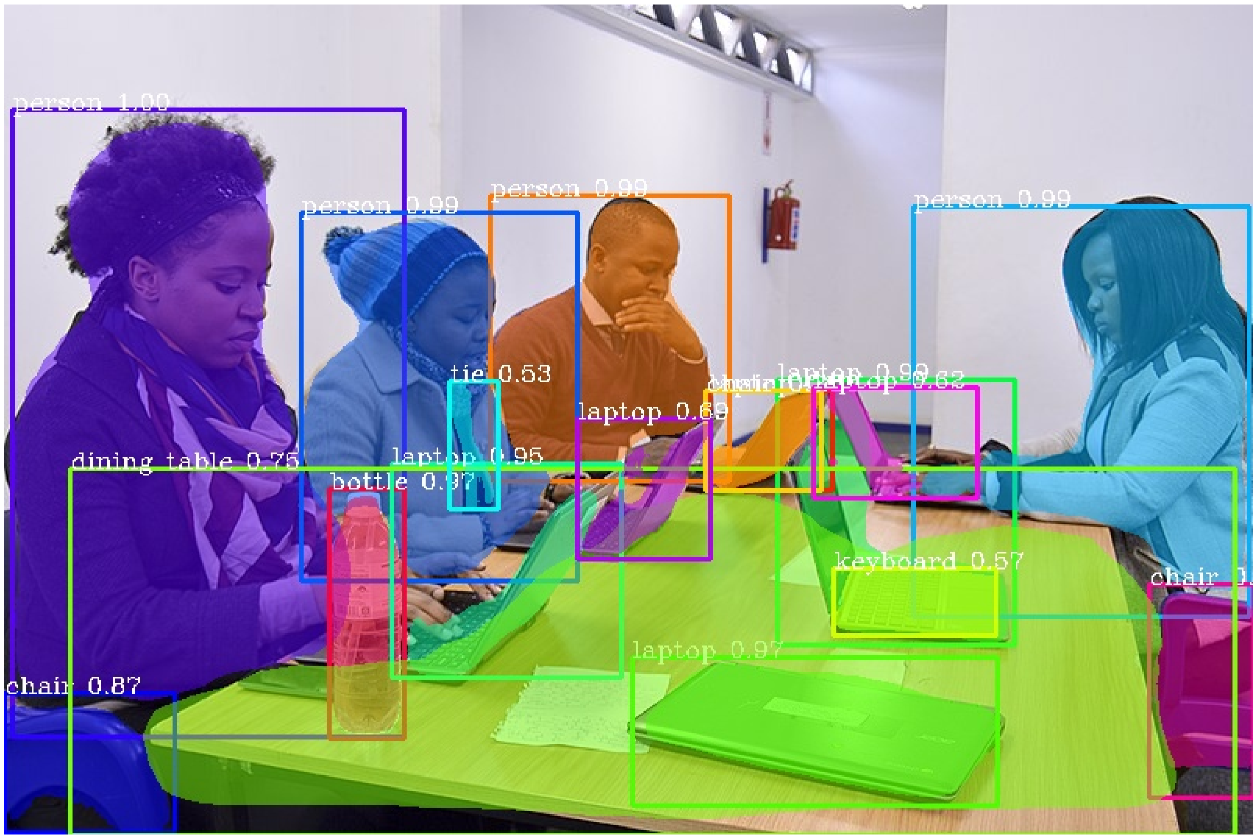
By default the detection speed is about 1 second for a processing a single image.

average detection mode

```
import pixellib
from pixellib.instance import instance_segmentation

segment_image = instance_segmentation(infer_speed = "average" )
segment_image.load_model("mask_rcnn_coco.h5")
segment_image.segmentImage("sample.jpg", show_bboxes = True, output_image_name = "new.
↪jpg")
```

In the modified code above within the class *instance_segmentation* we introduced a new parameter **infer_speed** which determines the speed of detection and it was set to **average**. The average value reduces the detection to half of its original speed, the detection speed would become *0.5* seconds for processing a single image.

Output Image



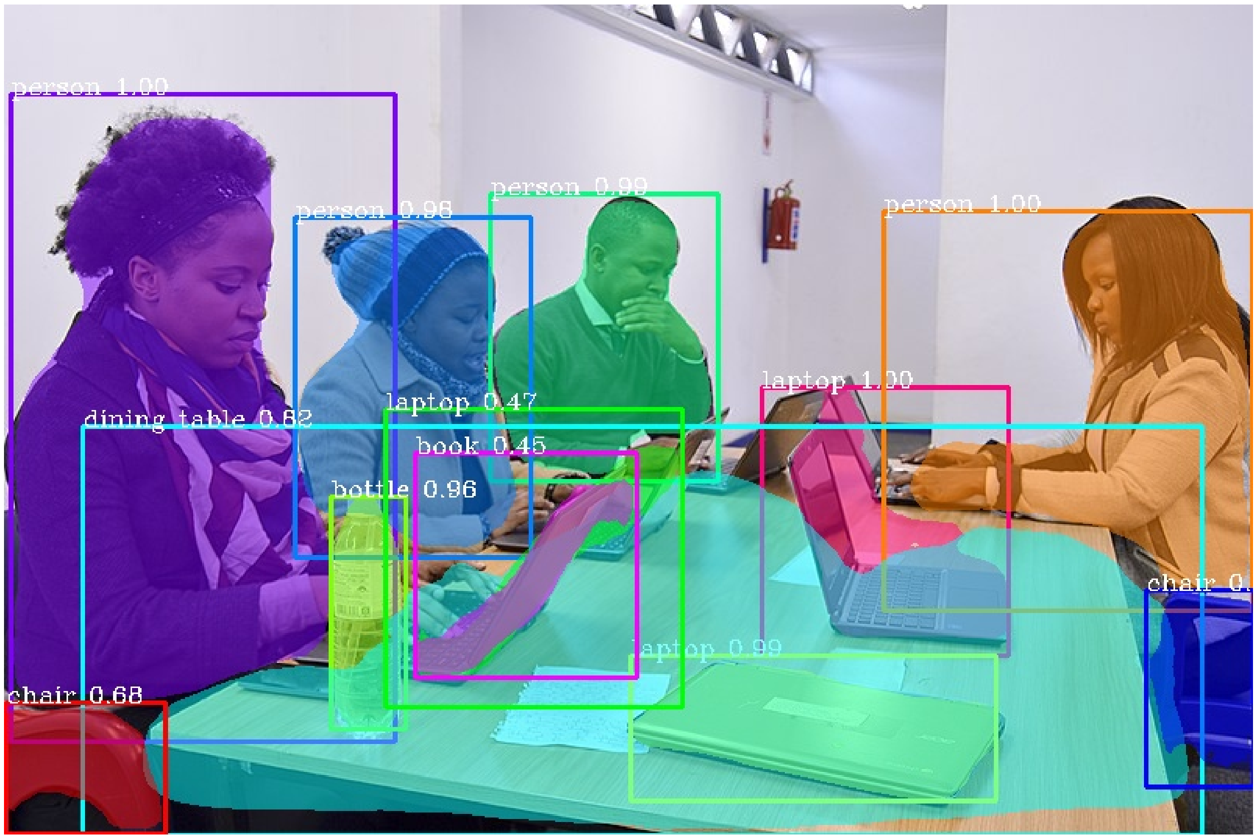
We obtained beautiful results with *average detection speed mode*.

fast detection mode

```
import pixellib
from pixellib.instance import instance_segmentation

segment_image = instance_segmentation(infer_speed = "fast" )
segment_image.load_model("mask_rcnn_coco.h5")
segment_image.segmentImage("sample.jpg", show_bboxes = True, output_image_name = "new.
↪jpg")
```

In the code above we replaced the **infer_speed** value to **fast** and the speed of detection is about *0.35* seconds for processing a single image.

Output Image



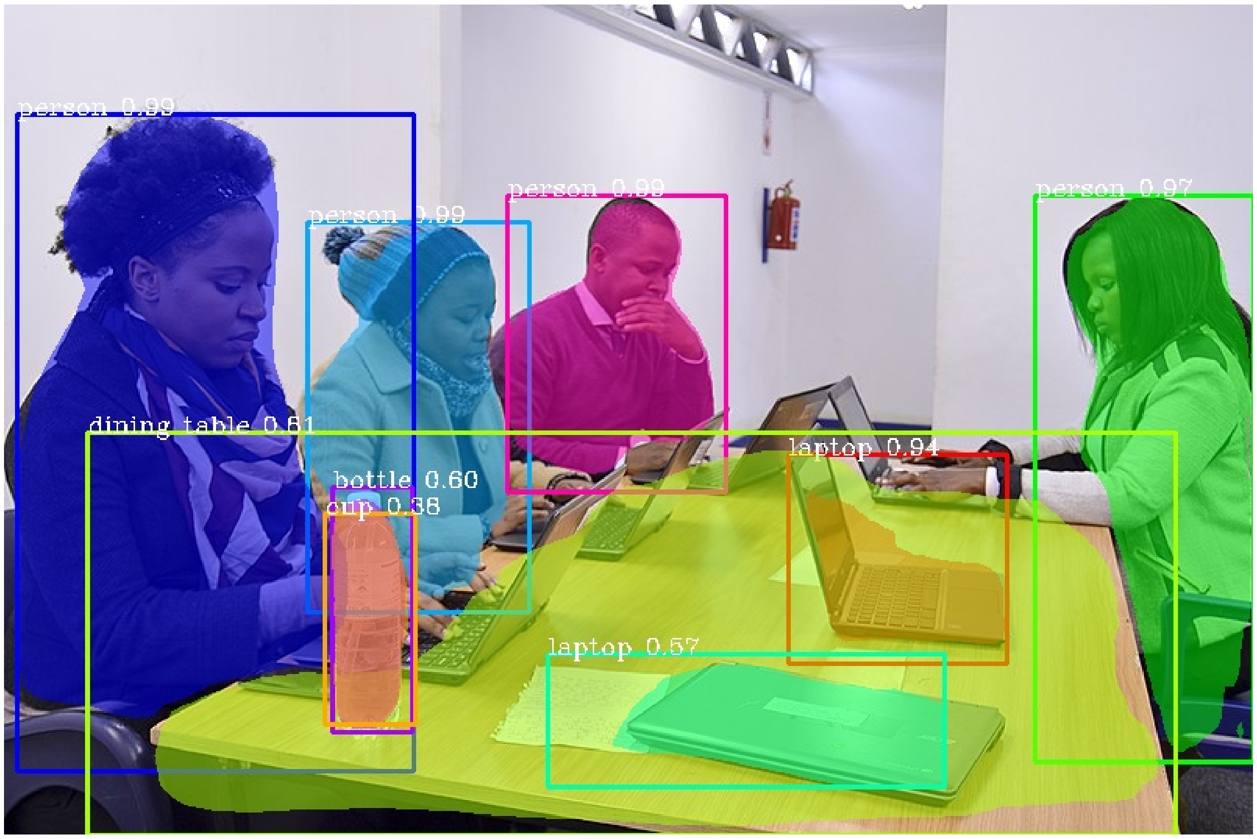
Our results are still wonderful with *fast detection speed mode*.

rapid detection mode

```python
import pixellib
from pixellib.instance import instance_segmentation

segment_image = instance_segmentation(infer_speed = "rapid" )
segment_image.load_model("mask_rcnn_coco.h5")
segment_image.segmentImage("sample.jpg", show_bboxes = True, output_image_name = "new.
↪jpg")
```

In the code above we replaced the **infer_speed** value to **rapid** which is the fastest detection mode. The speed of detection becomes *0.25* seconds for processing a single image.

Output Image



The *rapid detection speed mode* produces good results with the fastest inference speed.

Note These inference reports are obtained using Nvidia GeForce 1650.

Specialised uses of PixelLib may require you to return the array of the segmentation's output.

Obtain the following arrays:

-Detected Objects' arrays

-Objects' corresponding class_ids' arrays

-Segmentation masks' arrays

-Output's array

By using this code

```
segmask, output = segment_image.segmentImage()
```

- You can test the code for obtaining arrays and print out the shape of the output by modifying the instance segmentation code below.

```
import pixellib
from pixellib.instance import instance_segmentation
import cv2

instance_seg = instance_segmentation()
instance_seg.load_model("mask_rcnn_coco.h5")
segmask, output = instance_seg.segmentImage("sample2.jpg")
cv2.imwrite("img.jpg", output)
print(output.shape)
```

- Obtain arrays of segmentation with bounding boxes by including the parameter *show_bboxes*.

```
segmask, output = segment_image.segmentImage(show_bboxes = True)
```

```
import pixellib
from pixellib.instance import instance_segmentation
import cv2

instance_seg = instance_segmentation()
instance_seg.load_model("mask_rcnn_coco.h5")
segmask, output = instance_seg.segmentImage("sample2.jpg", show_bboxes= True)
cv2.imwrite("img.jpg", output)
print(output.shape)
```

Note: Access mask's values using *segmask['masks']*, bounding box coordinates using *segmask['rois']*, class ids using *segmask['class_ids']*.

```
segmask, output = segment_image.segmentImage(show_bboxes = True, extract_segmented_
↪objects= True )
```

Access the value of the extracted and croped segmented object using *segmask['extracted_objects']*

Process opencv's frames

```
import pixellib
from pixellib.instance import instance_segmentation
import cv2

segment_frame = instance_segmentation()
segment_frame.load_model("mask_rcnn_coco.h5")

capture = cv2.VideoCapture(0)
while True:
  ret, frame = capture.read()
  segment_video.segmentFrame(frame)
```

CHAPTER 8

Instance segmentation of videos with PixelLib

Instance segmentation with PixelLib is based on MaskRCNN framework.

Download the mask rcnn model from here

Code to implement instance segmentation of videos:

```
import pixellib
from pixellib.instance import instance_segmentation

segment_video = instance_segmentation()
segment_video.load_model("mask_rcnn_coco.h5")
segment_video.process_video("video_path", frames_per_second= 20, output_video_name=
↪"path_to_outputvideo")
```

```
import pixellib
from pixellib.instance

segment_video = instance_segmentation()
```

We imported in the class for performing instance segmentation and created an instance of the class.

```
segment_video.load_model("mask_rcnn_coco.h5")
```

We loaded the maskrcnn model trained on coco dataset to perform instance segmentation and it can be downloaded from here.

```
segment_video.process_video("sample_video2.mp4", frames_per_second = 20, output_video_
↪name = "output_video.mp4")
```

We called the function to perform segmentation on the video file.

It takes the following parameters:-

video_path: the path to the video file we want to perform segmentation on.

frames_per_second: this is parameter to set the number.of frames per second for the output video file. In this case it is set to 15 i.e the saved video file will have 15 frames per second.

output_video_name: the saved segmented video. The output video will be saved in your current working directory.

Sample video2

```
import pixellib
from pixellib.instance import instance_segmentation

segment_video = instance_segmentation()
segment_video.load_model("mask_rcnn_coco.h5")
segment_video.process_video("sample_video2.mp4", frames_per_second= 15, output_video_
↪name="output_video.mp4")
```

Output video

We can perform instance segmentation with object detection by setting the parameter *show_bboxes* to true.

```
import pixellib
from pixellib.instance import instance_segmentation

segment_video = instance_segmentation()
segment_video.load_model("mask_rcnn_coco.h5")
segment_video.process_video("sample_video2.mp4", show_bboxes = True, frames_per_
↪second= 15, output_video_name="output_video.mp4")
```

Output video with bounding boxes

CHAPTER 9

Extraction of objects in videos

PixelLib supports the extraction of objects in videos and camera feeds.

```
segment_video.process_video("sample.mp4", show_bboxes=True,  extract_segmented_
↪objects=True,save_extracted_objects=True, frames_per_second= 5,  output_video_name=
↪"output.mp4")
```

It still the same code except we introduced new parameters in the *process_video* which are:

extract_segmented_objects: this is the parameter that tells the function to extract the objects segmented in the image. It is set to true.

save_extracted_objects: this is an optional parameter for saving the extracted segmented objects.

Extracted objects from the video

CHAPTER 10

Segmentation of Specific Classes in Videos

The pre-trained coco model used detects 80 classes of objects. PixelLib has made it possible to filter out unused detections and detect the classes you want.

```
target_classes = segment_video.select_target_classes(person=True)
segment_video.process_video("sample.mp4", show_bboxes=True, segment_target_classes=␣
↪target_classes, extract_segmented_objects=True,save_extracted_objects=True, frames_
↪per_second= 5,  output_video_name="output.mp4")
```

The target class for detection is set to person and we were able to segment only the people in the video.

```
target_classes = segment_video.select_target_classes(car = True)
segment_video.process_video("sample.mp4", show_bboxes=True, segment_target_classes=␣
↪target_classes, extract_segmented_objects=True,save_extracted_objects=True, frames_
↪per_second= 5,  output_video_name="output.mp4")
```

The target class for detection is set to car and we were able to segment only the cars in the video.

Full code for filtering classes and object Extraction

CHAPTER 11

Instance Segmentation of Live Camera with Mask R-cnn.

We can use the same model to perform semantic segmentation on camera. This can be done by few modifications to the code used to process video file.

```
import pixellib
from pixellib.instance import instance_segmentation
import cv2


capture = cv2.VideoCapture(0)

segment_video = instance_segmentation()
segment_video.load_model("mask_rcnn_coco.h5")
segment_video.process_camera(capture, frames_per_second= 15, output_video_name=
↪"output_video.mp4", show_frames= True,
frame_name= "frame")
```

```
import cv2
capture = cv2.VideoCapture(0)
```

We imported cv2 and included the code to capture camera frames.

```
segment_video.process_camera(capture, show_bboxes = True, frames_per_second = 15,␣
↪output_video_name = "output_video.mp4", show_frames = True, frame_name = "frame")
```

In the code for performing segmentation, we replaced the video filepath to capture i.e we are going to process a stream camera frames instead of a video file.We added extra parameters for the purpose of showing the camera frames.

show_frames this parameter handles showing of segmented camera frames and press q to exist.

frame_name this is the name given to the shown camera's frame.

A demo showing the output of pixelLib's instance segmentation of camera's feeds using Mask-RCNN. *Good work! It was able to successfully detect me and my phone.*

Detection of Target Classes in Live Camera Feeds

This is the modified code below to filter unused detections and detect a target class in a live camera feed.

```
import pixellib
from pixellib.instance import instance_segmentation
import cv2


capture = cv2.VideoCapture(0)

segment_video = instance_segmentation()
segment_video.load_model("mask_rcnn_coco.h5")
target_classes = segment_video.select_target_classes(person=True)
segment_video.process_camera(capture, segment_target_classes=target_classes,  frames_
↪per_second= 10, output_video_name="output_video.mp4", show_frames= True,frame_name=
↪"frame")
```

Full code for Filtering classes and object Extraction in Camera feeds

```
import pixellib
from pixellib.instance import instance_segmentation
import cv2


capture = cv2.VideoCapture(0)

segment_video = instance_segmentation()
segment_video.load_model("mask_rcnn_coco.h5")
target_classes = segment_video.select_target_classes(person=True)
segment_video.process_camera(capture, segment_target_classes=target_classes,  frames_
↪per_second= 10, extract_segmented_objects=True,
save_extracted_objects=True,output_video_name="output_video.mp4", show_frames= True,
↪frame_name= "frame")
```

Speed Adjustments for Faster Inference

PixelLib now supports the ability to adjust the speed of detection according to a user's needs. The inference speed with a minimal reduction in the accuracy of detection. There are three main parameters that control the speed of detection.

1 **average**

2 **fast**

3 **rapid**

By default the detection speed is about 1 second for a processing a single image using Nvidia GeForce 1650.

Using Average Detection Mode

```
import pixellib
from pixellib.instance import instance_segmentation
import cv2


capture = cv2.VideoCapture(0)

segment_video = instance_segmentation(infer_speed = "average")
segment_video.load_model("mask_rcnn_coco.h5")
segment_video.process_camera(capture, frames_per_second= 10, output_video_name=
↪"output_video.mp4", show_frames= True,
frame_name= "frame")
```

In the modified code above within the class *instance_segmentation* we introduced a new parameter **infer_speed** which determines the speed of detection and it was set to **average**. The average value reduces the detection to half of its original speed, the detection speed would become *0.5* seconds for processing a single image.

Using fast Detection Mode

```
import pixellib
from pixellib.instance import instance_segmentation
import cv2


capture = cv2.VideoCapture(0)

segment_video = instance_segmentation(infer_speed = "fast")
segment_video.load_model("mask_rcnn_coco.h5")
segment_video.process_camera(capture, frames_per_second= 10, output_video_name=
↪"output_video.mp4", show_frames= True,
frame_name= "frame")
```

In the code above we replaced the **infer_speed** value to **fast** and the speed of detection is about *0.35* seconds for processing a single image.

Using rapid Detection Mode

```
import pixellib
from pixellib.instance import instance_segmentation
import cv2


capture = cv2.VideoCapture(0)

segment_video = instance_segmentation(infer_speed = "rapid")
segment_video.load_model("mask_rcnn_coco.h5")
segment_video.process_camera(capture, frames_per_second= 10, output_video_name=
↪"output_video.mp4", show_frames= True,
frame_name= "frame")
```

In the code above we replaced the **infer_speed** value to **rapid** the fastest the detection mode. The speed of detection becomes *0.25* seconds for processing a single image. The rapid detection speed mode achieves 4fps.

CHAPTER 12

Custom Training With PixelLib

Implement custom training on your own dataset using PixelLib's Library. In just seven Lines of code you can create a custom model for perform instance segmentation and object detection for your own application.

Prepare your dataset

Our goal is to create a model that can perform instance segmentation and object detection on butterflies and squirrels. Collect images for the objects you want to detect and annotate your dataset for custom training. Labelme is the tool employed to perform polygon annotation of objects. Create a root directory or folder and within it create train and test folder. Separate the images required for training (a minimum of 300) and test. Put the images you want to use for training in the train folder and put the images you want to use for testing in the test folder. You will annotate both images in the train and test folder. Download Nature's dataset used as a sample dataset in this article, unzip it to extract the images' folder. This dataset will serve as a guide for you to know how to organize your images. Ensure that the format of the directory of your own dataset directory is not different from it. Nature is a dataset with two categories butterfly and squirrel. There is 300 images for each class for training and 100 images for each class for testing i.e 600 images for training and 200 images for validation. Nature is a dataset with 800 images.

Read this article on medium and learn how to annotate objects with *Labelme*.

Note: Use labelme for annotation of objects. If you use a different annotation tool it will not be compatible with the library.

```
Nature >>train>>>>>>>>>>>>> image1.jpg
                            image1.json
                            image2.jpg
                            image2.json

    >>test>>>>>>>>>>>>>>>>> img1.jpg
                            img1.json
                            img2.jpg
                            img2.json
```

Sample of folder directory after annotation.

Visualize Dataset

Visualize a sample image before training to confirm that the masks and bounding boxes are well generated.

```
import pixellib
from pixellib.custom_train import instance_custom_training

vis_img = instance_custom_training()
vis_img.load_dataset("Nature")
vis_img.visualize_sample()
```

```
import pixellib
from pixellib.custom_train import instance_custom_training
vis_img = instance_custom_training()
```

We imported in pixellib, from pixellib import the class instance_custom_training and created an instance of the class.

```
vis_img.load_dataset("Nature")
```

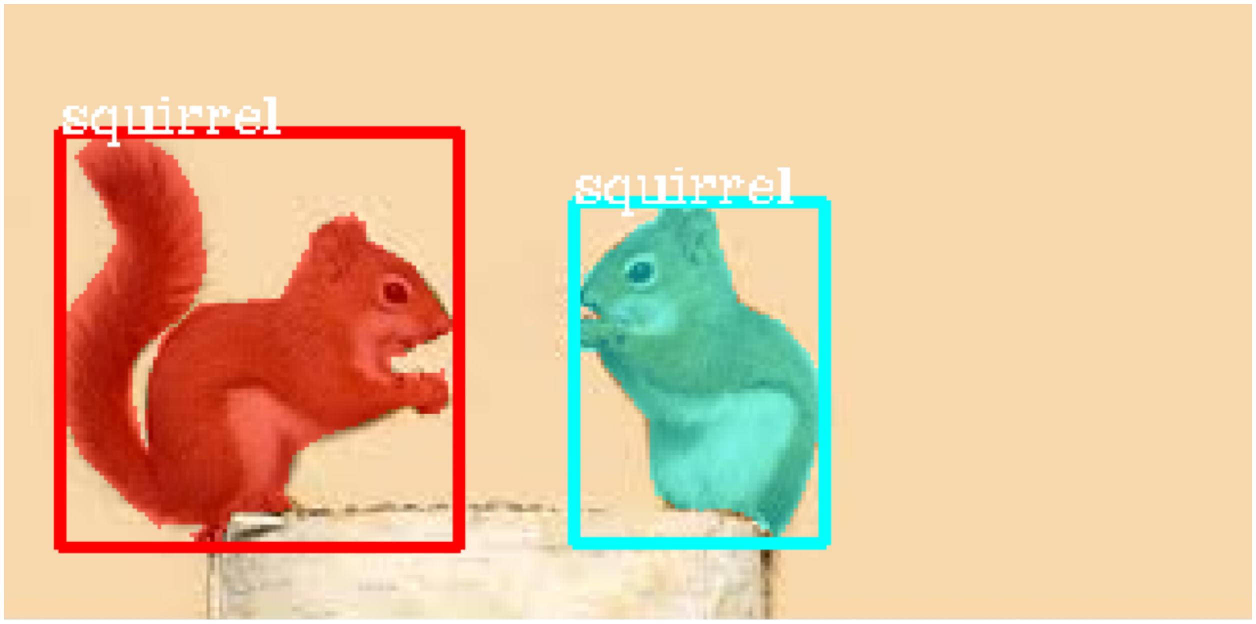
We loaded the dataset using *load_dataset function*. PixelLib requires polygon annotations to be in coco format, when you call the *load_data function* the individual json files in the train and test folder will be converted into a single *train.json* and *test.json* respectively. The train and test json files will be located in the root directory as the train and test folder. The new folder directory will now look like this:

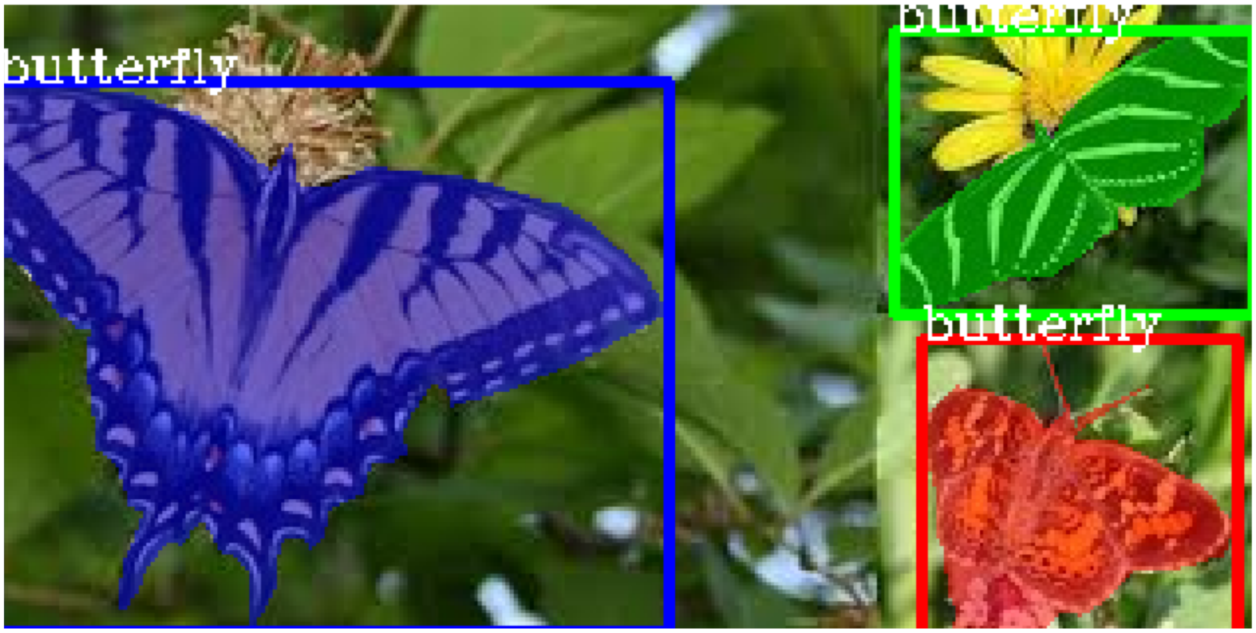
```
Nature >>>>>>>>train>>>>>>>>>>>>>>>> image1.jpg
           train.json                image1.json
                                     image2.jpg
                                     image2.json

   >>>>>>>>>>>test>>>>>>>>>>>>>>>>> img1.jpg
              test.json              img1.json
                                     img2.jpg
                                     img2.json
```

Inside the load_dataset function annotations are extracted from the jsons's files. Bitmap masks are generated from the polygon points of the annotations and bounding boxes are generated from the masks. The smallest box that encapsulates all the pixels of the mask is used as a bounding box.

```
vis_img.visualize_sample()
```

When you called this function it shows a sample image with a mask and bounding box.





Great! the dataset is fit for training, the load_dataset function successfully generates mask and bounding box for each object in the image. Random colors are generated for the masks in HSV space and then converted to RGB.

Train a custom model Using your dataset

```
import pixellib
from pixellib.custom_train import instance_custom_training

train_maskrcnn = instance_custom_training()
train_maskrcnn.modelConfig(network_backbone = "resnet101", num_classes= 2, batch_size␣
↪= 4)
train_maskrcnn.load_pretrained_model("mask_rcnn_coco.h5")
train_maskrcnn.load_dataset("Nature")
train_maskrcnn.train_model(num_epochs = 300, augmentation=True,  path_trained_models␣
↪= "mask_rcnn_models")
```

This is the code for performing training, in just seven lines of code you train your dataset.

```
train_maskrcnn.modelConfig(network_backbone = "resnet101", num_classes= 2, batch_size␣
↪= 4)
```

We called the function modelConfig, i.e model's configuration. It takes the following parameters:

network_backbone: This the CNN network used as a feature extractor for mask-rcnn. The feature extractor used is resnet101.

num_classes: We set the number of classes to the categories of objects in the dataset. In this case we have two classes(butterfly and squirrel) in nature's dataset.

batch_size: This is the batch size for training the model. It is set to 4.

```
train_maskrcnn.load_pretrained_model("mask_rcnn_coco.h5")
train_maskrcnn.load_dataset("Nature")
```

We are going to employ the technique of transfer learning for training the model. Coco model has been trained on 8O categories of objects, it has learnt a lot of features that will help in training the model. We called the function load_pretrained_model function to load the mask-rcnn coco model.We loaded the dataset using *load_dataset function*.

Download coco model from here

```
train_maskrcnn.train_model(num_epochs = 300, augmentation=True,path_trained_models =
↪"mask_rcnn_models")
```

Finally, we called the train function for training maskrcnn model. We called *train_model function*. The function takes the following parameters:

num_epochs: The number of epochs required for training the model. It is set to 300.

augmentation: Data augmentation is applied on the dataset, this is because we want the model to learn different representations of the objects.

path_trained_models: This is the path to save the trained models during training. Models with the lowest validation losses are saved.

```
Using resnet101 as network backbone For Mask R-CNN model
Train 600 images
Validate 200 images
Applying augmentation on dataset
Checkpoint Path: mask_rcnn_models
Selecting layers to train
Epoch 1/200
100/100 - 164s - loss: 2.2184 - rpn_class_loss: 0.0174 - rpn_bbox_loss: 0.8019 -␣
↪mrcnn_class_loss: 0.1655 - mrcnn_bbox_loss: 0.7274 - mrcnn_mask_loss: 0.5062 - val_
↪loss: 2.5806 - val_rpn_class_loss: 0.0221 - val_rpn_bbox_loss: 1.4358 - val_mrcnn_
↪class_loss: 0.1574 - val_mrcnn_bbox_loss: 0.6080 - val_mrcnn_mask_loss: 0.3572

Epoch 2/200
100/100 - 150s - loss: 1.4641 - rpn_class_loss: 0.0126 - rpn_bbox_loss: 0.5438 -␣
↪mrcnn_class_loss: 0.1510 - mrcnn_bbox_loss: 0.4177 - mrcnn_mask_loss: 0.3390 - val_
↪loss: 1.2217 - val_rpn_class_loss: 0.0115 - val_rpn_bbox_loss: 0.4896 - val_mrcnn_
↪class_loss: 0.1542 - val_mrcnn_bbox_loss: 0.3111 - val_mrcnn_mask_loss: 0.2554

Epoch 3/200
100/100 - 145s - loss: 1.0980 - rpn_class_loss: 0.0118 - rpn_bbox_loss: 0.4122 -␣
↪mrcnn_class_loss: 0.1271 - mrcnn_bbox_loss: 0.2860 - mrcnn_mask_loss: 0.2609 - val_
↪loss: 1.0708 - val_rpn_class_loss: 0.0149 - val_rpn_bbox_loss: 0.3645 - val_mrcnn_
↪class_loss: 0.1360 - val_mrcnn_bbox_loss: 0.3059 - val_mrcnn_mask_loss: 0.2493
```

(continues on next page)

(continued from previous page)

```
```

This is the training log it shows the network backbone used for training mask-rcnn which is *resnet101*, the number of images used for training and number of images used for validation. In the *path_to_trained models's* directory the models are saved based on decrease in validation loss, typical model name will appear like this: **mask_rcnn_model_25–0.55678**, it is saved with its *epoch number* and its corresponding *validation loss*.

Network Backbones: There are two network backbones for training mask-rcnn

1. Resnet101

2. Resnet50

Google colab: Google Colab provides a single 12GB NVIDIA Tesla K80 GPU that can be used up to 12 hours continuously.

Using Resnet101: Training Mask-RCNN consumes alot of memory. On google colab using resnet101 as network backbone you will be able to train with a batchsize of 4. The default network backbone is resnet101. Resnet101 is used as a default backbone because it appears to reach a lower validation loss during training faster than resnet50. It also works better for a dataset with multiple classes and much more images.

Using Resnet50: The advantage with resnet50 is that it consumes lesser memory, you can use a batch_size of 6 0r 8 on google colab depending on how colab randomly allocates gpu. The modified code supporting resnet50 will be like this.

Full code

```
import pixellib
from pixellib.custom_train import instance_custom_training

train_maskrcnn = instance_custom_training()
train_maskrcnn.modelConfig(network_backbone = "resnet50", num_classes= 2, batch_size␣
↪= 6)
train_maskrcnn.load_pretrained_model("mask_rcnn_coco.h5")
train_maskrcnn.load_dataset("Nature")
train_maskrcnn.train_model(num_epochs = 300, augmentation=True, path_trained_models =
↪"mask_rcnn_models")
```

The main differences from the original code is that in the model configuration function we set network_backbone to be *resnet50* and changed the batch size to 6.

The only difference in the training log is this:

```
Using resnet50 as network backbone For Mask R-CNN model
```

It shows that we are using *resnet50* for training.

Note: The batch_sizes given are samples used for google colab. If you are using a less powerful GPU, reduce your batch size, for example a PC with a 4G RAM GPU you should use a batch size of 1 for both resnet50 or resnet101. I used a batch size of 1 to train my model on my PC's GPU, train for less than 100 epochs and it produced a validation loss of 0.263. This is favourable because my dataset is not large. A PC with a more powerful GPU you can use a batch size of 2. If you have a large dataset with more classes and much more images use google colab where you have free access to a single 12GB NVIDIA Tesla K80 GPU that can be used up to 12 hours continuously. Most importantly try and use a more powerful GPU and train for more epochs to produce a custom model that will perform efficiently across multiple classes. Achieve better results by training with much more images. 300 images for each each class is recommended to be the minimum required for training.

Model Evaluation

When we are done with training we should evaluate models with lowest validation losses. Model evaluation is used to access the performance of the trained model on the test dataset. Download the trained model from here.

```
import pixellib
from pixellib.custom_train import instance_custom_training


train_maskrcnn = instance_custom_training()
train_maskrcnn.modelConfig(network_backbone = "resnet101", num_classes= 2)
train_maskrcnn.load_dataset("Nature")
train_maskrcnn.evaluate_model("mask_rccn_models/Nature_model_resnet101.h5")
```

output

```
mask_rcnn_models/Nature_model_resnet101.h5 evaluation using iou_threshold 0.5 is 0.
↪890000
```

The mAP(Mean Avearge Precision) of the model is *0.89*.

You can evaluate multiple models at once, what you just need is to pass in the folder directory of the models.

```
import pixellib
from pixellib.custom_train import instance_custom_training


train_maskrcnn = instance_custom_training()
train_maskrcnn.modelConfig(network_backbone = "resnet101", num_classes= 2)
train_maskrcnn.load_dataset("Nature")
train_maskrcnn.evaluate_model("mask_rccn_models")
```

Output log

```
mask_rcnn_models\Nature_model_resnet101.h5 evaluation using iou_threshold 0.5 is 0.
↪890000

mask_rcnn_models\mask_rcnn_model_055.h5 evaluation using iou_threshold 0.5 is 0.867500

mask_rcnn_models\mask_rcnn_model_058.h5 evaluation using iou_threshold 0.5 is 0.
↪8507500
```

```
import pixellib
from pixellib.custom_train import instance_custom_training


train_maskrcnn = instance_custom_training()
train_maskrcnn.modelConfig(network_backbone = "resnet50", num_classes= 2)
train_maskrcnn.load_dataset("Nature")
train_maskrcnn.evaluate_model("path_to_model path or models's folder directory")
```

Note: Change the network_backbone to resnet50 if you are evaluating a resnet50 model.

Visit Google Colab's notebook set up for training a custom dataset

Learn how how to perform inference with your custom model by reading this tutorial

CHAPTER 13

Inference With A Custom Model

We have trained and evaluated the model, the next step is to see the performance of the model on unknown images. We are going to test the model on the classes we have trained it on. If you have not download the trained model, download it from here.

sample1.jpg

```
import pixellib
from pixellib.instance import custom_segmentation

segment_image = custom_segmentation()
segment_image.inferConfig(num_classes= 2, class_names= ["BG", "butterfly", "squirrel
↪"])
segment_image.load_model("mask_rcnn_models/Nature_model_resnet101.h5")
segment_image.segmentImage("sample1.jpg", show_bboxes=True, output_image_name="sample_
↪out.jpg")
```

```
import pixellib
from pixellib.instance import custom_segmentation
segment_image =custom_segmentation()
segment_image.inferConfig(num_classes= 2, class_names= ["BG", "butterfly", "squirrel
↪"])
```

We imported the class custom_segmentation, the class for performing inference and created an instance of the class. We called the model configuration and introduced an extra parameter class_names.

```
class_names= ["BG", "butterfly", "squirrel"])
```

class_names: It is a list containing the names of classes the model is trained with.Butterfly has the class id 1 and squirrel has the class id 2 “BG”, it refers to the background of the image, it is the first class and must be available along the names of the classes.

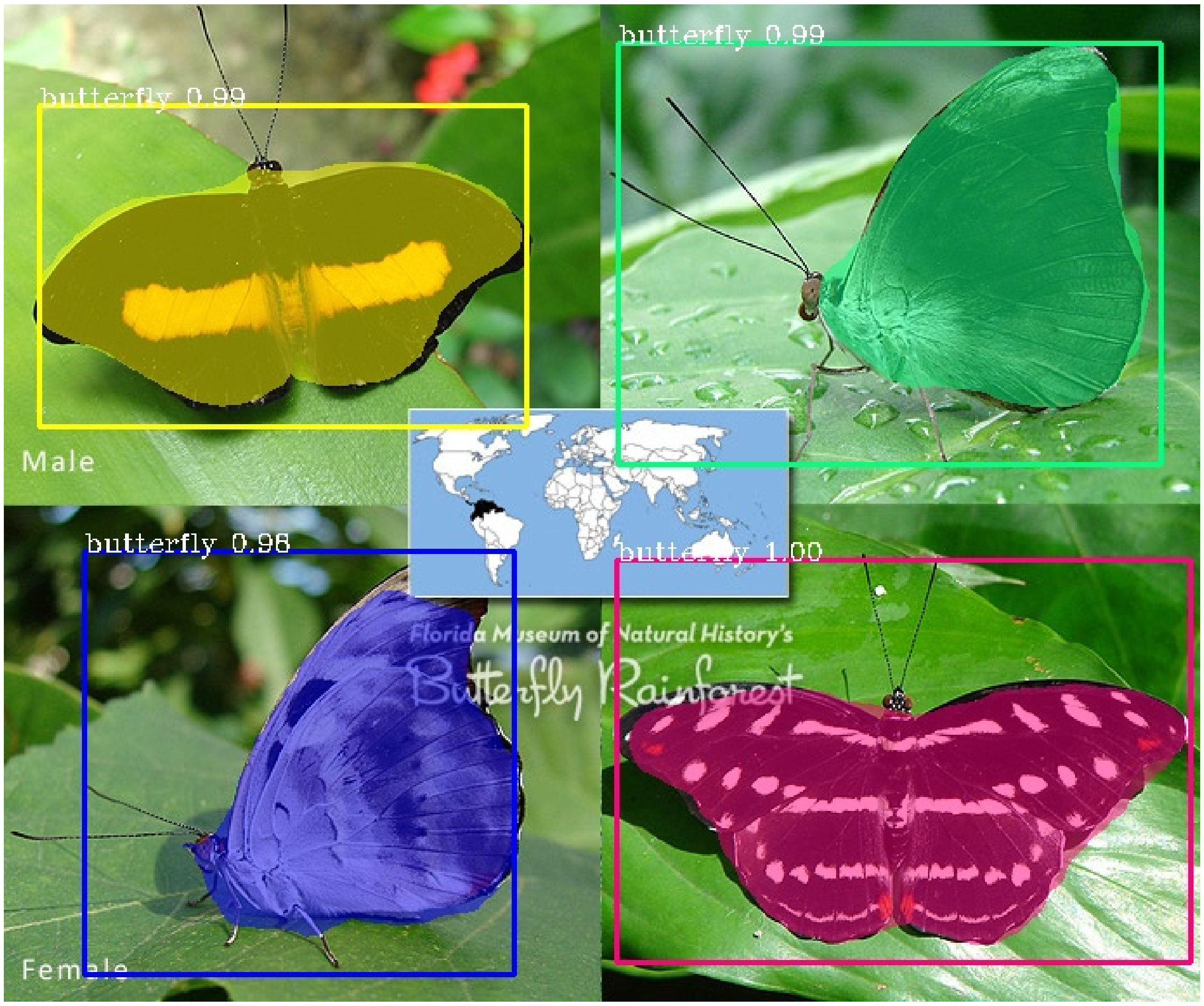
Note: If you have multiple classes and you are confused of how to arrange the classes’s names according to their class ids, in your test.json in the dataset’s folder check the categories’ list.

```
{
"images": [
{
"height": 205,
"width": 246,
"id": 1,
"file_name":
↪"C:\\Users\\olafe\\Documents\\Ayoola\\PIXELLIB\\Final\\Nature\\test\\butterfly (1).
↪png"
},
],
"categories": [
{
"supercategory": "butterfly",
"id": 1,
"name": "butterfly"
},
{
"supercategory": "squirrel",
"id": 2,
"name": "squirrel"
}
],
```

You can observe from the sample of the directory of test.json above, after the images’s list in your test.json is object categories’s list, the classes’s names are there with their corresponding class ids. Remember the first id “0” is kept in reserve for the background.

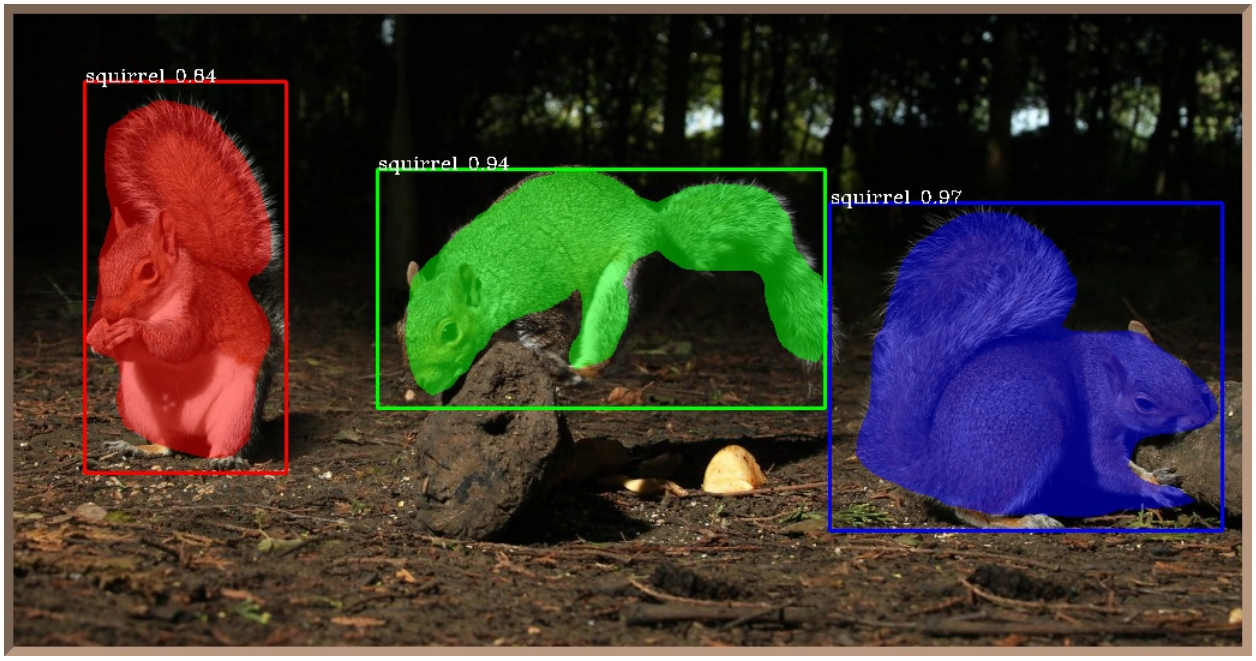
```
segment_image.load_model("mask_rcnn_model/Nature_model_resnet101.h5)

segment_image.segmentImage("sample1.jpg", show_bboxes=True, output_image_name="sample_
↪out.jpg")
```

The custom model is loaded and we called the function to segment the image.


butterfly 0.99
butterfly 0.99
Male
butterfly 0.98
butterfly 1.00
Florida Museum of Natural History's
Butterfly Rainforest
Female

sample2.jpg

```
test_maskrcnn.segmentImage("sample2.jpg",show_bboxes = True, output_image_name=
↪"sample_out.jpg")
```



WOW! We have successfully trained a custom model for performing instance segmentation and object detection on butterflies and squirrels.

Extraction of Segmented Objects

PixelLib now makes it possible to extract each of the segmented objects in an image and save each of the object extracted as a separate image. This is the modified code below;

```
import pixellib
```

(continues on next page)

(continued from previous page)

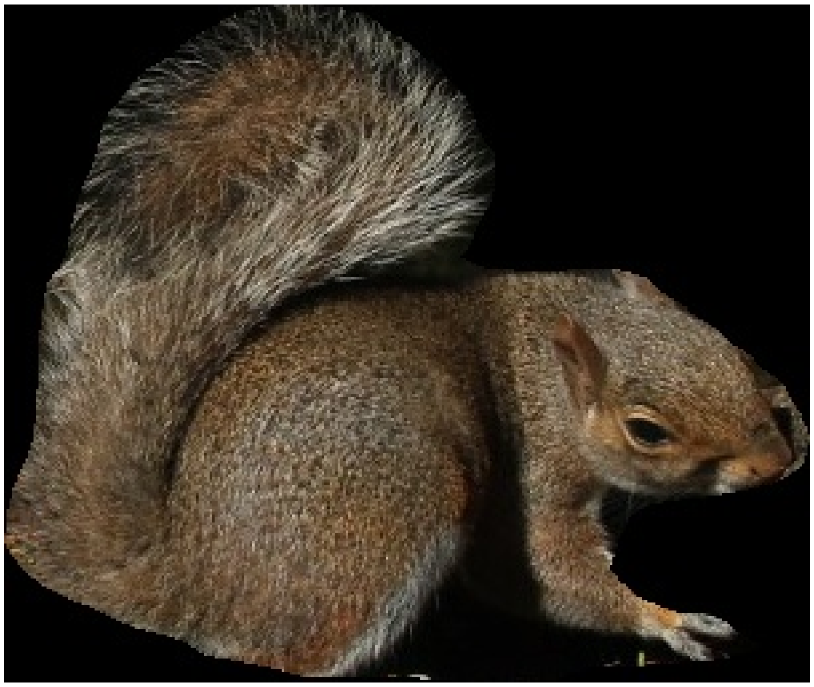
```
from pixellib.instance import custom_segmentation

segment_image = custom_segmentation()
segment_image.inferConfig(num_classes= 2, class_names= ["BG", "butterfly", "squirrel
↪"])
segment_image.load_model("mask_rcnn_model/Nature_model_resnet101.h5")
segment_image.segmentImage("sample2.jpg", show_bboxes=True, output_image_name="output.
↪jpg",
extract_segmented_objects= True, save_extracted_objects=True)
```

We introduced new parameters in the *segmentImage* function which are:

extract_segmented_objects: This parameter handles the extraction of each of the segmented object in the image.

save_extracted_objects: This parameter saves each of the extracted object as a separate image.Each of the object extracted in the image would be save with the name *segmented_object* with the corresponding index number such as *segmented_object_1*.

These are the objects extracted from the image above.



Specialised uses of PixelLib may require you to return the array of the segmentation's output.

Obtain the following arrays:

-Detected Objects' arrays

-Objects' corresponding class_ids' arrays

-Segmentation masks' arrays

-Output's array

By using this code

```
segmask, output = segment_image.segmentImage()
```

- You can test the code for obtaining arrays and print out the shape of the output by modifying the instance segmentation code below.

```
import pixellib
from pixellib.instance import custom_segmentation

segment_image = custom_segmentation()
segment_image.inferConfig(num_classes= 2, class_names= ["BG", "butterfly", "squirrel
↪"])
segment_image.load_model("mask_rcnn_model/Nature_model_resnet101.h5")
segmask, output = segment_image.segmentImage("sample2.jpg")
cv2.imwrite("img.jpg", output)
print(output.shape)
```

Obtain arrays of segmentation with bounding boxes by including the parameter *show_bboxes*.

```
segmask, output = segment_image.segmentImage(show_bboxes = True)
```

- Full code

```
import pixellib
from pixellib.instance import custom_segmentation

segment_image = custom_segmentation()
segment_image.inferConfig(num_classes= 2, class_names= ["BG", "butterfly", "squirrel
↪"])
segment_image.load_model("mask_rcnn_model/Nature_model_resnet101.h5")
segmask, output = segment_image.segmentImage("sample2.jpg", show_bboxes= True)
cv2.imwrite("img.jpg", output)
print(output.shape)
```

Note:

Access mask's values using *segmask['masks']*, bounding box coordinates using *segmask['rois']*, class ids using *segmask['class_ids']*.

```
segmask, output = segment_image.segmentImage(show_bboxes = True, extract_segmented_
↪objects= True )
```

Access the value of the extracted and croped segmented object using *segmask['extracted_objects']*

Video segmentation with a custom model.

sample_video1

We want to perform segmentation on the butterflies in this video.

```
import pixellib
from pixellib.instance import custom_segmentation

test_video = custom_segmentation()
test_video.inferConfig(num_classes=  2, class_names=["BG", "butterfly", "squirrel"])
test_video.load_model("Nature_model_resnet101")
test_video.process_video("sample_video1.mp4", show_bboxes = True,  output_video_name=
↪"video_out.mp4", frames_per_second=15)
```

```
test_video.process_video("video.mp4", show_bboxes = True,  output_video_name="video_
↪out.mp4", frames_per_second=15)
```

The function process_video is called to perform segmentation on objects in a video.

It takes the following parameters:-

video_path: this is the path to the video file we want to segment.

frames_per_second: this is the parameter used to set the number of frames per second for the saved video file. In this case it is set to 15 i.e the saved video file will have 15 frames per second.

output_video_name: this is the name of the saved segmented video. The output video will be saved in your current working directory.

Output_video

A sample of another segmented video with our custom model.

CHAPTER 14

Extraction of Segmented Objects in Videos

```
segment_video.process_video("sample.mp4", show_bboxes=True,  extract_segmented_
↪objects=True,save_extracted_objects=True, frames_per_second= 5,  output_video_name=
↪"output.mp4")
```

It still the same code except we introduced new parameters in the *process_video* which are:

extract_segmented_objects: this is the parameter that tells the function to extract the objects segmented in the image. It is set to true.

save_extracted_objects: this is an optional parameter for saving the extracted segmented objects.

Extracted objects from the video

You can perform live camera segmentation with your custom model making use of this code:

```
import pixellib
from pixellib.instance import custom_segmentation
import cv2


capture = cv2.VideoCapture(0)

segment_camera = custom_segmentation()
segment_camera.inferConfig(num_classes=2, class_names=["BG", "butterfly", "squirrel"])
segment_camera.load_model("Nature_model_resnet101.h5")
segment_camera.process_camera(capture, frames_per_second= 10, output_video_name=
↪"output_video.mp4", show_frames= True,
frame_name= "frame", check_fps = True)
```

You will replace the process_video funtion with process_camera function.In the function, we replaced the video's filepath to capture i.e we are processing a stream of frames captured by the camera instead of a video file. We added extra parameters for the purpose of showing the camera frames:

show_frames: this parameter handles the showing of segmented camera's frames.

frame_name: this is the name given to the shown camera's frame.

check_fps: You may want to check the number of frames processed, just set the parameter check_fps is true. It will print out the number of frames processed per second.

Full code for object extraction in camera feeds Using A Custom Model

```
import pixellib
from pixellib.instance import custom_segmentation
import cv2

capture = cv2.VideoCapture(0)
segment_frame = custom_segmentation()
segment_frame.inferConfig(num_classes=2, class_names=['BG', 'butterfly', 'squirrel'])
segment_frame.load_model("Nature_model_resnet101.h5")
```

(continues on next page)

(continued from previous page)

```
segment_frame.process_camera(capture, show_bboxes=True, show_frames=True, extract_
↪segmented_objects=True,
save_extracted_objects=True,frame_name="frame", frames_per_second=5, output_video_
↪name="output.mp4")
```

Process opencv's frames

```
import pixellib
from pixellib.instance import custom_segmentation
import cv2

segment_frame = custom_segmentation()
segment_frame.inferConfig(network_backbone="resnet101", num_classes=2, class_names=[
↪"BG", "butterfly", "squirrel"])
segment_frame.load_model("Nature_model_resnet101.h5")

capture = cv2.VideoCapture(0)
 while True:
   ret, frame = capture.read()
   segment_frame.segmentFrame(frame)
   cv2.imshow("frame", frame)
   if  cv2.waitKey(25) & 0xff == ord('q'):
      break
```

CHAPTER 15

Image Tuning With PixelLib

Image tuning is the change in the background of an image through image segmentation. The key role of image segmentation is to remove the objects segmented from the image and place it in the new background created. This is done by producing a mask for the image and combining it with the modified background. Deeplabv3+ model trained on pascalvoc dataset is used. The model supports 20 common object catefories which means you can change the background of these objects as you desired. The objects are

```
person,bus,car,aeroplane, bicycle, ,motorbike,bird, boat, bottle,  cat, chair, cow,␣
↪dinningtable, dog,
horse pottedplant, sheep, sofa, train, tv
```

Image Tuning features supported are:

Change the background of an image with a picture

Assign a distinct color to the background of an image

Grayscale the background of an image

Blur the background of an image

Change the background of an image with a picture

sample.jpg

background.jpg

We intend to change the background of our sample image with this image. We can do this easily with just five lines of code.

```
import pixellib
from pixellib.tune_bg import alter_bg
change_bg = alter_bg(model_type = "pb")
change_bg.load_pascalvoc_model("xception_pascalvoc.pb")
change_bg.change_bg_img(f_image_path = "sample.jpg",b_image_path = "background.jpg",␣
↪output_image_name="new_img.jpg")
```

```
import pixellib
from pixellib.tune_bg import alter_bg

change_bg = alter_bg(model_type = "pb")
change_bg.load_pascalvoc_model("xception_pascalvoc.pb")
```

Line 1-4: We imported pixellib and from pixellib we imported in the class *alter_bg*. Instance of the class is created and within the class we added a parameter model_type and set it to **pb**. we finally loaded the deeplabv3+ model. PixelLib supports two deeplabv3+ models, keras and tensorflow model. The keras model is extracted from the tensorflow model's checkpoint. The tensorflow model performs better than the keras model extracted from its checkpoint. We will make use of tensorflow model. Download the model from here.

```
change_bg.change_bg_img(f_image_path = "sample.jpg",b_image_path = "background.jpg",␣
↪output_image_name="new_img.jpg")
```

We called the function *change_bg_img* that handled changing the background of the image with a picture.

It takes the following parameter:

f_image_path: This is the foreground image, the image which background would be changed.

b_image_path: This is the image that will be used to change the backgroud of the foreground image.

output_image_name: The new image with a changed background.

Output Image

Wow! We have successfully changed the background of our image.

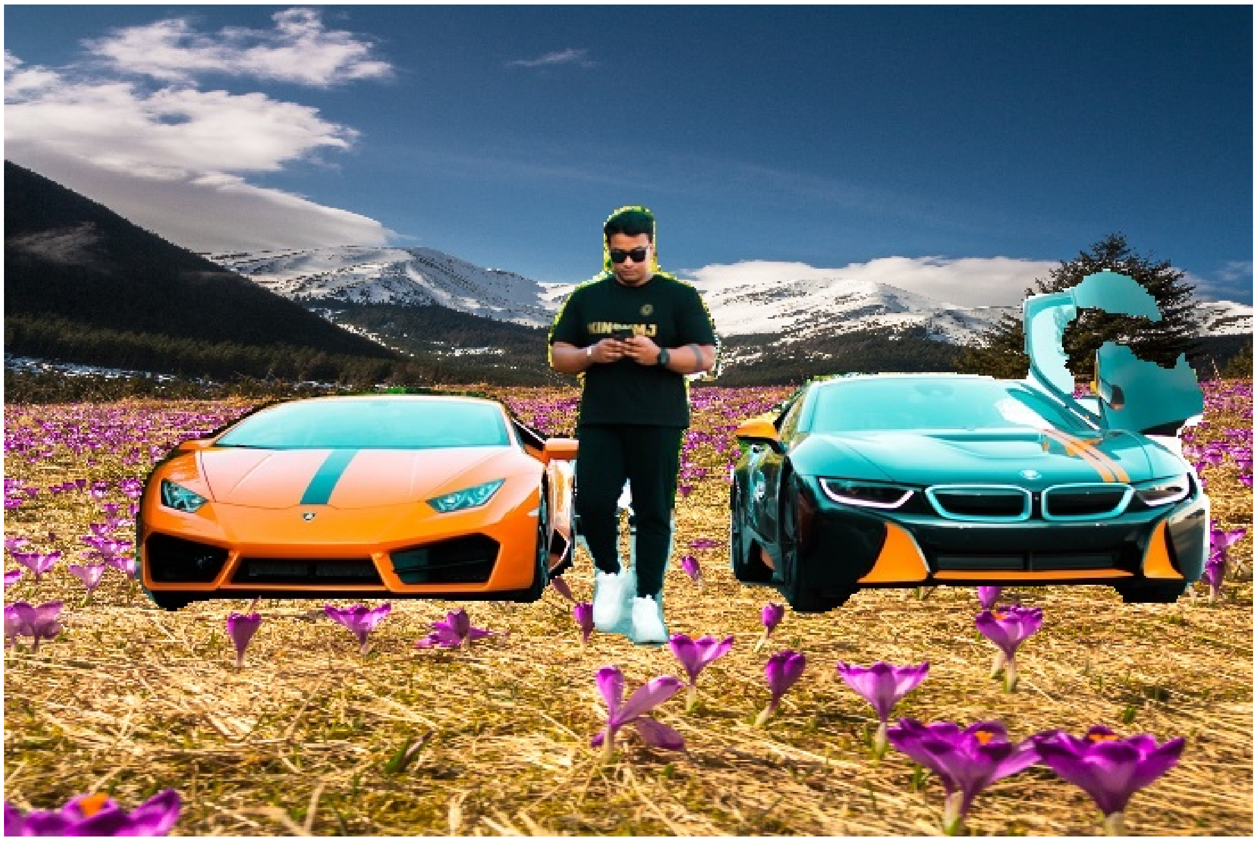
Detection of target object

In some applications you may not want to detect all the objects in an image or video, you may just want to target a particular object. By default the model detects all the objects it supports in an image or video. It is possible to filter other objects' detections and detect a target object in an image or video.

sample2.jpg

```
change_bg.change_bg_img(f_image_path = "sample2.jpg",b_image_path = "background.jpg",␣
↪output_image_name="new_img.jpg")
```

Output Image



It successfully change the image's background, but our goal is to change the background of the person in this image. We are not comfortable with the other objects showing, this is because car is one the objects supported by the model. Therefore there is need to modify the code to detect a target object.

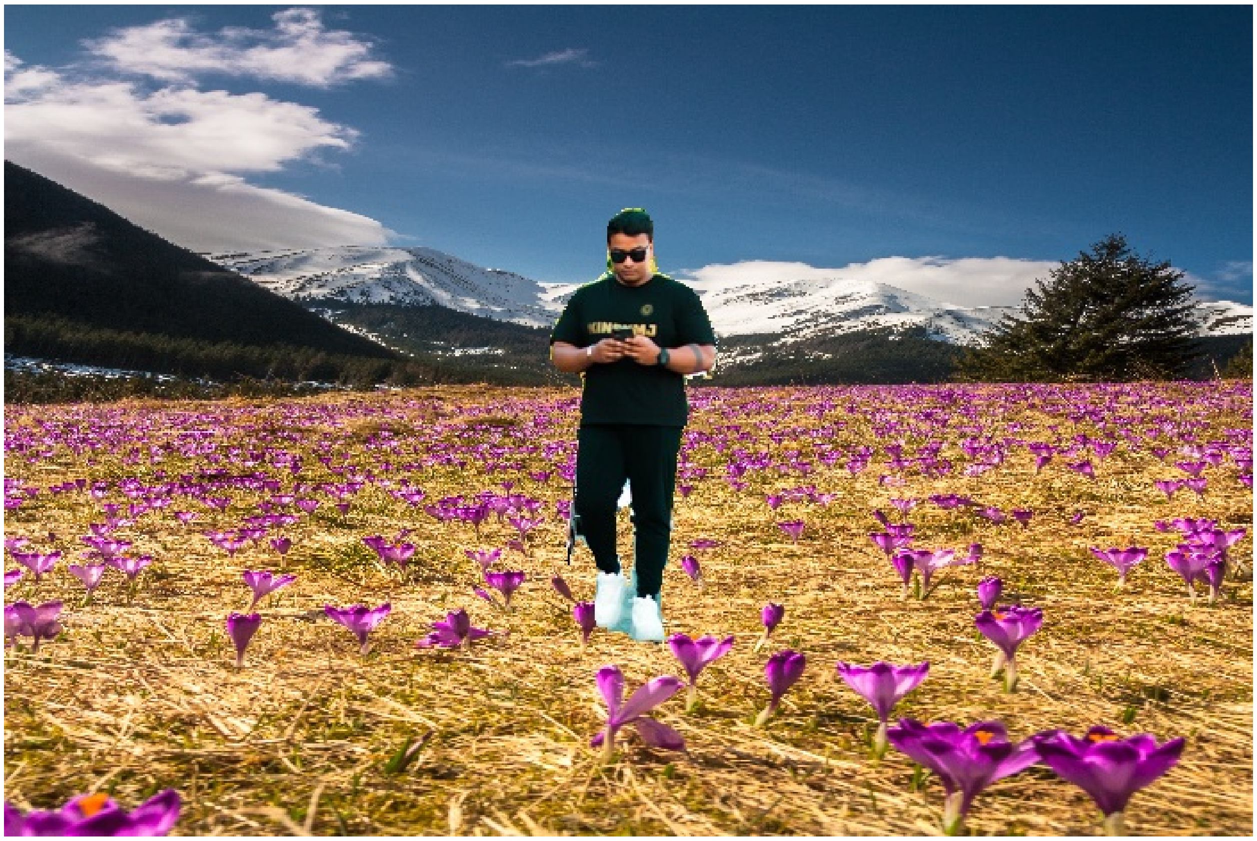
```
import pixellib
from pixellib.tune_bg import alter_bg

change_bg = alter_bg(model_type = "pb")
change_bg.load_pascalvoc_model("xception_pascalvoc.pb")
change_bg.change_bg_img(f_image_path = "sample2.jpg",b_image_path = "background.jpg",␣
↪output_image_name="new_img.jpg", detect = "person")
```

It is still the same code except we introduced an extra parameter detect in the function.

```
change_bg.change_bg_img(f_image_path = "sample2.jpg",b_image_path = "background.jpg",␣
↪output_image_name="new_img.jpg", detect = "person")
```

The parameter detect is set to *person*.

Output Image



This is the new image with only our target object shown. If we intend to show only the cars present in this image. We just have to change the value of the parameter person to *car*.

```
change_bg.change_bg_img(f_image_path = "sample2.jpg",b_image_path = "background.jpg",␣
↪output_image_name="new_img.jpg", detect = "car")
```

Output Image

Assign a distinct color to the background of an image

You can choose to assign any distinct color to the background of your image. This is also possible with five lines of code.

```
import pixellib
from pixellib.tune_bg import alter_bg

change_bg = alter_bg(model_type = "pb")
change_bg.load_pascalvoc_model("xception_pascalvoc.pb")
change_bg.color_bg("sample2.jpg", colors = (0,0,255), output_image_name="colored_bg.
↪jpg", detect = "person")
```

It is very similar to the code used above for changing the background of an image with a picture. The only difference is that we replaced the function *change_bg_img* to *color_bg* the function that will handle color change.

```
change_bg.color_bg("sample2.jpg", colors = (0, 0, 255), output_image_name="colored_bg.
↪jpg", detect = "person")
```

The function *color_bg* takes the parameter *colors* and we provided the RGB value of the color we want to use. We want the image to have a blue background and the color's RGB value is set to blue which is *(0, 0, 255)*.

Colored Image

Note: You can assign any color to the background of your image, just provide the RGB value of the color.

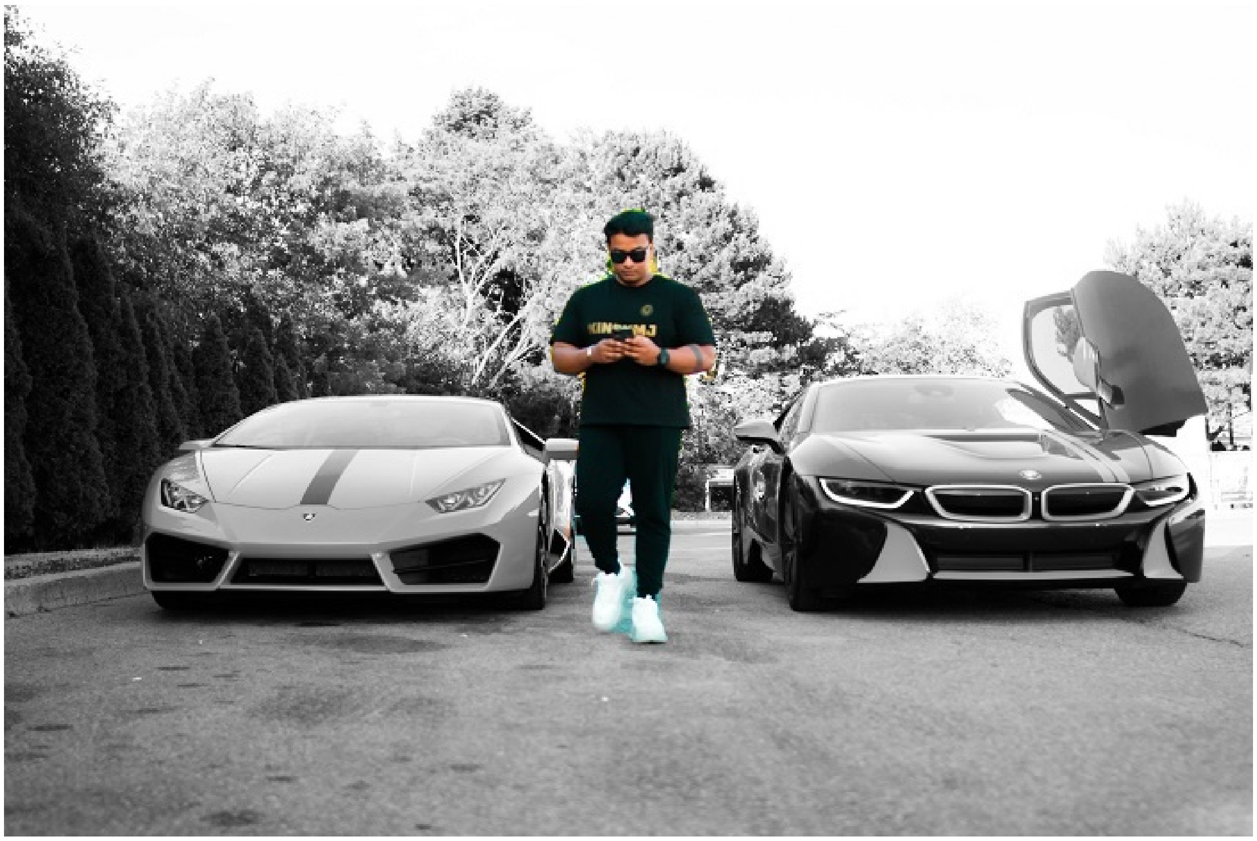
Grayscale the background of an image

```
import pixellib
from pixellib.tune_bg import alter_bg

change_bg = alter_bg(model_type = "pb")
change_bg.load_pascalvoc_model("xception_pascalvoc.pb")
change_bg.gray_bg("sample2.jpg",output_image_name="gray_img.jpg", detect = "person")
```

```
change_bg.gray_bg("sample.jpg",output_image_name="gray_img.jpg", detect = "person")
```

It is still the same code except we called the function *gray_bg* to grayscale the background of the image.

Output Image



Blur the background of an image

sample3.jpg

You can also apply the effect of bluring the background of your image. You can control how blur the background will be.

```
change_bg.blur_bg("sample2.jpg", low = True, output_image_name="blur_img.jpg")
```

We called the function *blur_bg* to blur the background of the image and set the blurred effect to be low. There are three parameters that control the degree in which the background is blurred.

low: When it is set to true the background is blurred slightly.

moderate: When it is set to true the background is moderately blurred.

extreme: When it is set to true the background is deeply blurred.

blur_low

The image is blurred with a low effect.

```
change_bg.blur_bg("sample2.jpg", moderate = True, output_image_name="blur_img.jpg")
```

We want to moderately blur the background of the image, we set *moderate* to *true*.

blur_moderate

The image is blurred with a moderate effect.

```
change_bg.blur_bg("sample2.jpg", extreme = True, output_image_name="blur_img.jpg")
```

We want to deeply blurred the background of the image and we set *extreme* to *true*.

blur_extreme

The image is blurred with a deep effect.

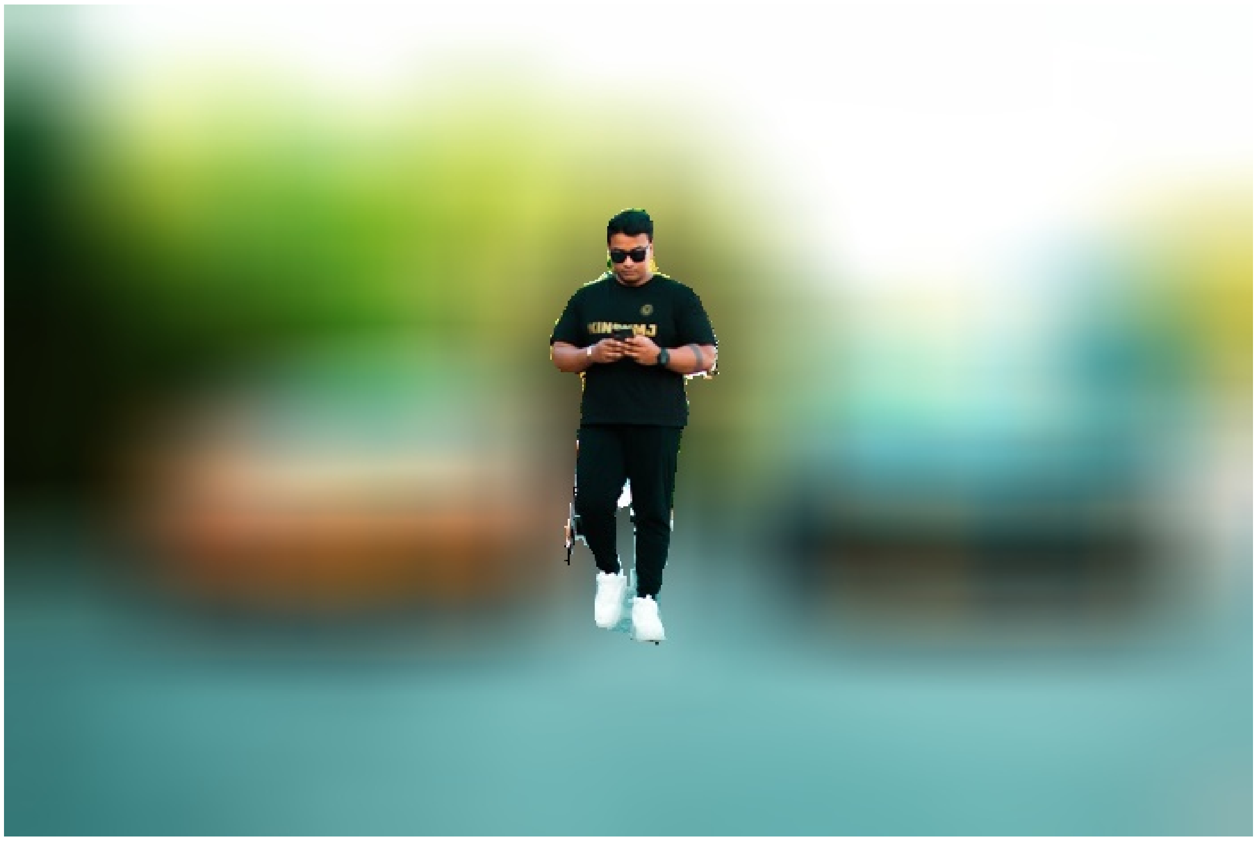
Full code

```
import pixellib
from pixellib.tune_bg import alter_bg

change_bg = alter_bg(model_type = "pb")
change_bg.load_pascalvoc_model("xception_pascalvoc.pb")
change_bg.blur_bg("sample2.jpg", moderate = True, output_image_name="blur_img.jpg")
```

Blur a target object in an image

```
change_bg.blur_bg("sample2.jpg", extreme = True, output_image_name="blur_img.jpg",␣
↪detect = "person")
```

Our target object is a person.



```
change_bg.blur_bg("sample2.jpg", extreme = True, output_image_name="blur_img.jpg",␣
↪detect = "car")
```

Our target object is a car.

Obtain output arrays

You can obtain the output arrays of your changed image. . . .

Obtain output array of the changed image array

```
import pixellib
from pixellib.tune_bg import alter_bg
import cv2

change_bg = alter_bg(model_type = "pb")
change_bg.load_pascalvoc_model("xception_pascalvoc.pb")
output = change_bg.change_bg_img(f_image_path = "sample.jpg",b_image_path =
↪"background.jpg", detect = "person")
cv2.imwrite("img.jpg", output)
```

Obtain output array of the colored image

```
import pixellib
from pixellib.tune_bg import alter_bg
import cv2

change_bg = alter_bg(model_type = "pb")
change_bg.load_pascalvoc_model("xception_pascalvoc.pb")
output = change_bg.color_bg("sample.jpg", colors = (0, 0, 255), detect = "person")
cv2.imwrite("img.jpg", output)
```

Obtain output array of the blurred image

```
import pixellib
from pixellib.tune_bg import alter_bg
import cv2

change_bg = alter_bg(model_type = "pb")
change_bg.load_pascalvoc_model("xception_pascalvoc.pb")
output = change_bg.blur_bg("sample.jpg", moderate = True, detect = "person")
cv2.imwrite("img.jpg", output)
```

Obtain output array of the grayed image

```
import pixellib
from pixellib.tune_bg import alter_bg
import cv2

change_bg = alter_bg(model_type = "pb")
change_bg.load_pascalvoc_model("xception_pascalvoc.pb")
output = change_bg.gray_bg("sample.jpg", detect = "person")
cv2.imwrite("img.jpg", output)
```

Process frames directly with Image Tuning…

Create a virtual background for frames

```
import pixellib
from pixellib.tune_bg import alter_bg
import cv2

change_bg = alter_bg(model_type = "pb")
change_bg.load_pascalvoc_model("xception_pascalvoc.pb")

capture = cv2.VideoCapture(0)
while True:
 ret, frame = capture.read()
 output = change_bg.change_frame_bg(frame, "flowers.jpg", detect = "person")
 cv2.imshow("frame", output)
 if  cv2.waitKey(25) & 0xff == ord('q'):
     break
```

Blur frames

```
import pixellib
from pixellib.tune_bg import alter_bg
import cv2

change_bg = alter_bg(model_type = "pb")
change_bg.load_pascalvoc_model("xception_pascalvoc.pb")

capture = cv2.VideoCapture(0)
while True:
 ret, frame = capture.read()
 output = change_bg.blur_frame(frame, extreme = True, detect = "person")
 cv2.imshow("frame", output)
 if  cv2.waitKey(25) & 0xff == ord('q'):
     break
```

Color frames

```
import pixellib
from pixellib.tune_bg import alter_bg
import cv2

change_bg = alter_bg(model_type = "pb")
change_bg.load_pascalvoc_model("xception_pascalvoc.pb")

capture = cv2.VideoCapture(0)
while True:
ret, frame = capture.read()
output = change_bg.color_frame(frame, colors = (255, 255, 255), detect = "person")
cv2.imshow("frame", output)
if  cv2.waitKey(25) & 0xff == ord('q'):
    break
```

Grayscale frames

```
import pixellib
from pixellib.tune_bg import alter_bg
import cv2

change_bg = alter_bg(model_type = "pb")
change_bg.load_pascalvoc_model("xception_pascalvoc.pb", detect = "person")

capture = cv2.VideoCapture(0)
while True:
 ret, frame = capture.read()
 output = change_bg.gray_frame(frame)
 cv2.imshow("frame", output)
 if  cv2.waitKey(25) & 0xff == ord('q'):
     break
```

Read the tutorial on blurring, coloring and grayscaling background of videos and camera's feeds.

CHAPTER 16

Change the Background of A Video

Blur Video Background

Blur the background of a video with five lines of code.

sample_video

Line 1-4: We imported pixellib and from pixellib we imported in the class *alter_bg*. Instance of the class is created and and within the class we added a parameter model_type and set it to **pb**. we finally loaded the deeplabv3+ model. PixelLib supports two deeplabv3+ models, keras and tensorflow model. The keras model is extracted from the tensorflow model's checkpoint. The tensorflow model performs better than the keras model extracted from its checkpoint. We will make use of tensorflow model. Download the model from here.

There are three parameters that control the degree in which the background is blurred.

low: When it is set to true the background is blurred slightly.

moderate: When it is set to true the background is moderately blurred.

extreme: When it is set to true the background is deeply blurred.

Detection of target object

In some applications you may not want to detect all the objects in a video, you may just want to target a particular object. By default the model detects all the objects it supports in an image or video. It is possible to filter other objects' detections and detect a target object in an image or video.

```
change_bg.blur_video("sample_video.mp4", extreme = True, frames_per_second=10, output_
↪video_name="blur_video.mp4", detect = "person")
```

This is the line of code that blurs the video's background. This function takes in four parameters:

video_path: the path to the video file we want to blur.

moderate: it is set to true and the background of the video would be moderatly blurred.

frames_per_second: this is the parameter to set the number of frames per second for the output video file. In this case it is set to 10 i.e the saved video file will have 10 frames per second.

output_video_name: the saved video. The output video will be saved in your current working directory.

detect (optional): this is the parameter that detects a particular object and filters out other detectons. It is set to detect *person* in the video.

Output Video

Blur the Background of Camera's Feeds

```
import pixellib
from pixellib.tune_bg import alter_bg
import cv2

capture = cv2.VideoCapture(0)
change_bg = alter_bg(model_type = "pb")
change_bg.load_pascalvoc_model("xception_pascalvoc.pb")
change_bg.blur_camera(capture, frames_per_second=10,extreme = True, show_frames =␣
↪True, frame_name = "frame",
output_video_name="output_video.mp4", detect = "person")
```

```
import cv2
capture = cv2.VideoCapture(0)
```

We imported cv2 and included the code to capture camera frames.

```
change_bg.blur_camera(capture, moderate = True, frames_per_second= 10, output_video_
↪name="output_video.mp4", show_frames= True,frame_name= "frame",  detect = "person")
```

In the code for blurring camera's frames, we replaced the video filepath to capture i.e we are going to process a stream of camera frames instead of a video file.We added extra parameters for the purpose of showing the camera frames:

show_frames: this parameter handles showing of segmented camera frames and press q to exist. **frame_name:** this is the name given to the shown camera's frame.

Output Video

A sample video of camera's feeds with my background blurred.

Create a Virtual Background for a Video

You can make use of any image to create a virtual background for a video.

sample video

Image to serve as background for a video

```
import pixellib
from pixellib.tune_bg import alter_bg

change_bg = alter_bg(model_type="pb")
change_bg.load_pascalvoc_model("xception_pascalvoc.pb")
change_bg.change_video_bg("sample_video.mp4", "bg.jpg", frames_per_second = 10,␣
↪output_video_name="output_video.mp4", detect = "person")
```

```
change_bg.change_video_bg("sample_video.mp4", "bg.jpg", frames_per_second = 10,␣
↪output_video_name="output_video.mp4", detect = "person")
```

It is still the same code except we called the function *change_video_bg* to create a virtual background for the video. The function takes in the path of the image we want to use as background for the video.

Output Video

Create a Virtual Background for Camera's Feeds

```
import pixellib
from pixellib.tune_bg import alter_bg
import cv2

cap = cv2.VideoCapture(0)

change_bg = alter_bg(model_type="pb")
change_bg.load_pascalvoc_model("xception_pascalvoc.pb")
change_bg.change_camera_bg(cap, "bg.jpg", frames_per_second = 10, show_frames=True,␣
↪frame_name="frame", output_video_name="output_video.mp4", detect = "person")
```

```
change_bg.change_camera_bg(cap, "bg.jpg", frames_per_second = 10, show_frames=True,␣
↪frame_name="frame", output_video_name="output_video.mp4", detect = "person")
```

It is similar to the code we used to blur camera's frames. The only difference is that we called the function *change_bg.change_camera_bg*. We performed the same routine, replaced the video filepath to capture and added the same parameters.

Output Video

PixelLib successfully created a virtual effect on my background.

Color Video Background

```
import pixellib
from pixellib.tune_bg import alter_bg

change_bg = alter_bg(model_type = "pb")
change_bg.load_pascalvoc_model("xception_pascalvoc.pb")
change_bg.color_video("sample_video.mp4", colors =  (0, 128, 0), frames_per_second=10,
↪ output_video_name="output_video.mp4", detect = "person")
```

```
change_bg.color_video("sample_video.mp4", colors =  (0, 128, 0), frames_per_second=10,
↪ output_video_name="output_video.mp4", detect = "person")
```

It is still the same code except we called the function *color_video* to give the video's background a distinct color. The function *color_bg* takes the parameter *colors* and we provided the RGB value of the color we want to use. We want the image to have a green background and the color's RGB value is set to green which is (0, 128, 0).

Output Video

Color the Background of Camera's Feeds

```
import pixellib
from pixellib.tune_bg import alter_bg
import cv2

capture = cv2.VideoCapture(0)
change_bg = alter_bg(model_type = "pb")
change_bg.load_pascalvoc_model("xception_pascalvoc.pb")
change_bg.color_camera(capture, frames_per_second=10,colors = (0, 128, 0), show_
↪frames = True, frame_name = "frame",
output_video_name="output_video.mp4", detect = "person")
```

```
change_bg.color_camera(capture, frames_per_second=10,colors = (0, 128, 0), show_
↪frames = True, frame_name = "frame",
output_video_name="output_video.mp4", detect = "person")
```

It is similar to the code we used to blur camera's frames. The only difference is that we called the function *color_camera*. We performed the same routine, replaced the video filepath to capture and added the same parameters.

Output Video

A sample video of camera's feeds with my background colored green.

Grayscale Video Background

```
import pixellib
from pixellib.tune_bg import alter_bg
```

(continues on next page)

(continued from previous page)

```

change_bg = alter_bg(model_type = "pb")
change_bg.load_pascalvoc_model("xception_pascalvoc.pb")
change_bg.gray_video("sample_video.mp4", frames_per_second=10, output_video_name=
↪"output_video.mp4", detect = "person")
```

```
change_bg.gray_video("sample_video.mp4", frames_per_second=10, output_video_name=
↪"output_video.mp4", detect = "person")
```

We are still using the same code but called a different function *gray_video* to grayscale the background of the video.

Output Video

Grayscale the Background of Camera's Feeds

```
import pixellib
from pixellib.tune_bg import alter_bg
import cv2

capture = cv2.VideoCapture(0)
change_bg = alter_bg(model_type = "pb")
change_bg.load_pascalvoc_model("xception_pascalvoc.pb")
change_bg.gray_camera(capture, frames_per_second=10, show_frames = True, frame_name =
↪"frame",
output_video_name="output_video.mp4", detect = "person")
```

It is similar to the code we used to color camera's frames. The only difference is that we called the function *gray_camera*. We performed the same routine, replaced the video filepath to capture and added the same parameters.

CONTACT INFO:

olafenwaayoola@gmail.com

Github.com

Twitter.com

Facebook.com

Linkedin.com