Published in Towards Data Science

Vinayak Nayak
Dec 12, 2020 · 6 min read · Listen

Semantic Image Segmentation with DeepLabv3-pytorch

Build your own blurring/background modification functionality in videos

Photo by Rodion Kutsaev on Unsplash

Introduction

With the surge in use of video calling services during the COVID lockdown, many players are offering a service where the user of the service could blur the background or add a custom background etc. (Zoom, MS-Teams etc.) This is a classic use case of image segmentation where the object of interest is located and the pixels barring this region are modified/substituted.

There are many deep learning architectures which could be used to solve the instance segmentation problem and today we're going to useDeeplab-v3 which is a State of the Art semantic image segmentation model which comes in many flavors. Its goal is to assign semantic labels (e.g., person, sheep, airplane and so on) to every pixel in the input image. We are going to particularly be focusing on using the Deeplabv3 model with a Resnet-101 backbone that is offered out of the box with the torch library.

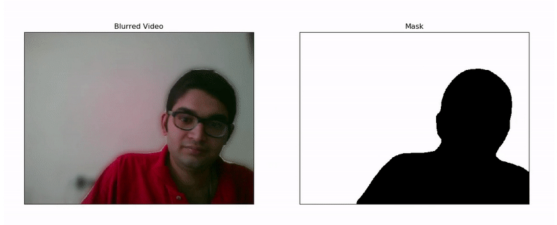

Image by Vinayak

At the end of this post, you'll be able to build something as shown above. We'll go about building this in a step by step fashion as follows:

- Dependencies needed
- Using cv2 to set up a video session and prepare output screens
- Using deeplab for predicting labels
- Applying segmentation (labels) map and blurring
- Bonus: Using custom background

So without any further ado, let's get started!

. . .

Dependencies

For reproducing the code below, we will need to have python installed on our system and in addition to that we will also need the following:

- torch
- Pillow
- numpy

- matplotlib
- cv2

These could be simply installed by using the command

```
pip install packagename
```

If you're working with conda, I would recommend you create a separate environment for this project using the following command. You can choose a higher version for python as well by changing the last portion to python=3.7 or python=3.8 etc.

```
conda create -n deeplabenv python=3.6
```

Once you install the above packages, we're ready to go to the next step.

. . .

Getting Input & Setting the stage for Output

We will be using opencv to interface a webcam for reading in input from our screens and we'll use matplotlib's pyplot module to render the processed video feed to output.

```
import cv2

# Start a video cam session
video_session = cv2.VideoCapture(0)

# Given a video capture object, read frames from the same and convert it to RGB
def grab_frame(cap):
  _, frame = cap.read()
  return cv2.cvtColor(frame, cv2.COLOR_BGR2RGB)
```

input_prep.py hosted with ❤ by GitHub view raw

opencv's VideoCapture object is used to get the image input for video. If you have multiple webcams you could create multiple such objects by passing the appropriate index; by default nowadays, most monitors have one inbuilt camera which could be indexed at 0th position.

Subsequently, opencv reads images in a BGR format but while rendering we need to show it in RGB format; so we've written a tiny function that captures a frame in realtime and converts it from BGR format to RGB format above. With this, we're set with the input preprocessing steps. Let's look at how we'll set the stage for output now.

```
import matplotlib.pyplot as plt
from utils import grab_frame

# Define two axes for showing the mask and the true video in realtime
# And set the ticks to none for both the axes
fig, (ax1, ax2) = plt.subplots(1, 2, figsize = (15, 8))
ax2.set_title("Mask")

ax1.set_xticks([])
ax1.set_yticks([])
ax2.set_xticks([])
ax2.set_yticks([])

# Create two image objects to picture on top of the axes defined above
im1 = ax1.imshow(grab_frame(video_session))
im2 = ax2.imshow(grab_frame(video_session))

# Switch on the interactive mode in matplotlib
plt.ion()
plt.show()
```

output_prep.py hosted with ❤ by GitHub view raw

Why use matplotlib.pyplot and not cv2.imshow? There are some inconsistencies with cv2.imshow when it comes to ubuntu distributions; I had one such issue which caused me to look at alternate methods and since pyplot is easy to implement and mostly included with all major python distributions like anaconda by default, I thought of using pyplot for rendering the output.

So basically we set up two subplots as seen in the gif on the top; one to see the blurred version and another one to actually look at the labels mask which the deeplab model has predicted. We switch on the interactive mode for pyplot and display the image captured directly at the very beginning of the stream. Now, that we have the stage set, let's discuss the part to obtain predictions from the deeplab-v3 model.

Deeplab-v3 Segmentation

The model offered at torch-hub for segmentation is trained on PASCAL VOC

dataset which contains 20 different classes of which the most important one for us is the person class with label 15.

```python
import torch
import torchvision

def load_model():
    # Load the DeepLab v3 model to system
    device = "cuda" if torch.cuda.is_available() else "cpu"
    model = torch.hub.load('pytorch/vision:v0.6.0', 'deeplabv3_resnet101', pretrained=True
    model.to(device).eval()
    return model
```

deeplab_v3_model_load.py hosted with ❤ by GitHub view raw

Using the above code we can download the model from torch-hub and use it for our segmentation task. Note that since this is based on top of a Resnet-101 model which is quite heavy, the inference will take time and rendering will lag heavily if you do not have a medium sized GPU if not a high end one. Now that we have the model loaded, let's discuss how we could get predictions from it.

```python
import torch
import torchvision

def get_pred(img, model):
  # See if GPU is available and if yes, use it
  device = "cuda" if torch.cuda.is_available() else "cpu"

  # Define the standard transforms that need to be done at inference time
  imagenet_stats = [[0.485, 0.456, 0.406], [0.485, 0.456, 0.406]]
  preprocess = torchvision.transforms.Compose([torchvision.transforms.ToTensor(),
                                               torchvision.transforms.Normalize(mean = ima
                                                                                std  = ima
  input_tensor = preprocess(img).unsqueeze(0)
  input_tensor = input_tensor.to(device)

  # Make the predictions for labels across the image
  with torch.no_grad():
      output = model(input_tensor)["out"][0]
      output = output.argmax(0)

  # Return the predictions
  return output.cpu().numpy()
```

deeplab_v3_model_predict.py hosted with ❤ by GitHub view raw

We will first load the image using Pillow or directly get it from the VideoCapture object of cv2 defined above. Once we have this, we will normalize it using imagenet stats and convert it to a tensor. Then if we have a GPU available at our disposal, we can move the tensor to GPU. All pre-trained models expect input images in mini-batches of 3-channel RGB images of shape `(N, 3, H, W)`, where `N` is the number of images, `H` and `W` are height and width of the two images respectively. Since our video capture object captures single frames, it's output is `(3, H, W).` We therefore unsqueeze the tensor along the first dimension to make it `(1, 3, H, W).`

Once we do that, we then put the model in eval mode and without autograd perform a forward pass through the same. The forward pass gives `aux` and `out` objects from which, the `out` object is of interest to us during inference. The `aux` values have loss value per pixel and are useful during train time. So, we get the out object and do a argmax along the 1st dimension to obtain the labels map which is of the same height and width as the original image with a single channel. This mask could be used for segmentation which we'll look at in the next section.

. . .

Human Segmentation

Now that we have the predictions, we can use them for segmenting the human and blurring the background.

```python
from utils import *
import numpy as np
import cv2

# Define the kernel size for applying Gaussian Blur
blur_value = (51, 51)

frame = utils.grab_frame(video_session)
# Read the frame's width, height, channels and get the labels' predictions from utilities
width, height, channels = frame.shape
labels = utils.get_pred(frame, model)

# Wherever there's empty space/no person, the label is zero
# Hence identify such areas and create a mask (replicate it across RGB channels)
mask = labels == 0
mask = np.repeat(mask[:, :, np.newaxis], channels, axis = 2)

# Apply the Gaussian blur for background with the kernel size specified in constants abov
blur = cv2.GaussianBlur(frame, blur_value, 0)
frame[mask] = blur[mask]
ax1.set_title("Blurred Video")

```

```
# Set the data of the two images to frame and mask values respectively
im1.set_data(frame)
im2.set_data(mask * 255)
plt.pause(0.01)
```

segment.py hosted with ❤ by GitHub view raw

The labels extracted out of the image are two dimensional. We need to replicate this mask across all three channels i.e. RGB. So, we use numpy to repeat this segmentation mask across all three channels.

Next, we use opencv's Gaussian Blur to blue the entire frame output and get a new image called blur. The extent of blur is determined by the `blur_value` variable which is a tuple representing the kernel size for blurring. Higher the value, more the blur and vice versa.

Next, with the help of mask, we replace the background pixels of the frame with the blurred frame's pixels. Now that we have both the mask and the resulting manipulated frame, we set the data of the two image variables defined in the *Setting the stage for output* section to these two frames respectively and define an interval for interactive pyplot object for refreshing the frame (think of it as an equivalent of frame rate).

This is how you could use DeepLabv3 for making your very own background blurring feature on custom videos or live vidcams with Image Segmentation.

Bonus: Background Substitution with custom image

Just like we blurred the background above, we could also substitute the background with a custom image and it only requires some minor modifications to the code.

```
from utils import *
import cv2
import numpy as np

# Read the background image to memory
bg_image = cv2.imread(BG_PTH)
bg_image = cv2.cvtColor(bg_image, cv2.COLOR_BGR2RGB)

# Start a video cam session
video_session = cv2.VideoCapture(0)

# Read frame from the video capture object and get the labels from get_pred function
frame = utils.grab_frame(video_session)
width, height, channels = frame.shape
labels = utils.get_pred(frame, model)

# Define a blurring value kernel size for cv2's Gaussian Blur
blur_value = (51, 51)

# The PASCAL VOC dataset has 20 categories of which Person is the 16th category
# Hence wherever person is predicted, the label returned will be 15
# Subsequently repeat the mask across RGB channels
mask = labels == 15
mask = np.repeat(mask[:, :, np.newaxis], 3, axis = 2)

# Resize the image as per the frame capture size
bg = cv2.resize(bg_image, (height, width))
bg[mask] = frame[mask]
frame = bg
ax1.set_title("Background Changed Video")
```

custom_background.py hosted with ❤ by GitHub view raw

We read a background image of our choice for substitution using cv2 and bring it to RGB format. Next, in the part where we replace the frame's non-human pixels with blurred pixels, we now use this background image pixels instead.

Basically, we resize the background image to fit the frame capture source or the video source, then mask all the human pixels in this image with the frame's human pixels. This gives the source human overlaid on top of the background image like shown in the gif below.

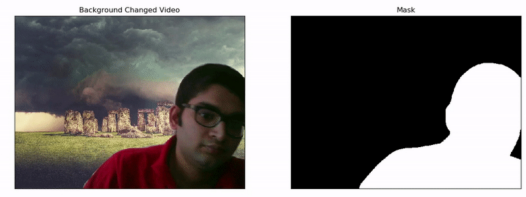

Image by Vinayak

Hope you enjoyed reading this post as much as I did while writing it. If you want to read further, I've mentioned some links in the resource section which may prove helpful. Cheers!

. . .

References

1. Deeplabv3-Resnet101 Pytorch
2. All the code for the above example
3. More on Image Segmentation
4. PyImageSearch Article on Instance Segmentation with opencv