A Beginner's guide to Deep Learning based Semantic Segmentation using Keras

Divam Gupta · 06 Jun 2019

Pixel-wise image segmentation is a well-studied problem in computer vision. The task of semantic image segmentation is to classify each pixel in the image. In this post, we will discuss how to use deep convolutional neural networks to do image segmentation. We will also dive into the implementation of the pipeline – from preparing the data to building the models.

I have packaged all the code in an easy to use repository:
https://github.com/divamgupta/image-segmentation-keras

Deep learning and convolutional neural networks (CNN) have been extremely ubiquitous in the field of computer vision. CNNs are popular for several computer vision tasks such as Image Classification, Object Detection, Image Generation, etc. Like for all other computer vision tasks, deep learning has surpassed other approaches for image segmentation

What is semantic segmentation

Semantic image segmentation is the task of classifying each pixel in an image from a predefined set of classes. In the following example, different entities are classified.

Semantic segmentation of a bedroom image

In the above example, the pixels belonging to the bed are classified in the class “bed”, the pixels corresponding to the walls are labeled as “wall”, etc.

In particular, our goal is to take an image of size W x H x 3 and generate a W x H matrix containing the predicted class ID’s corresponding to all the pixels.

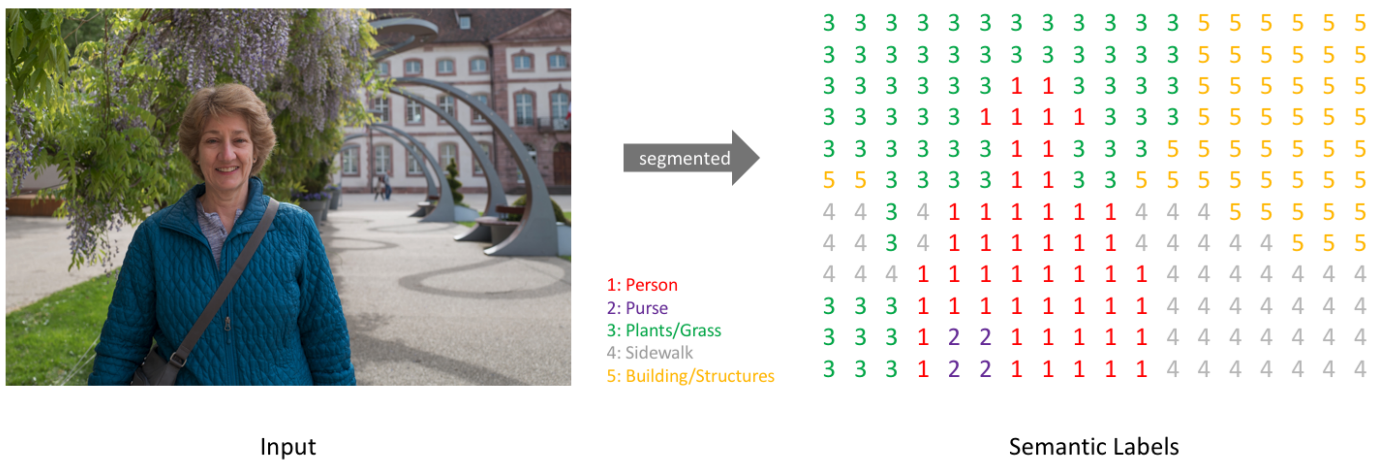
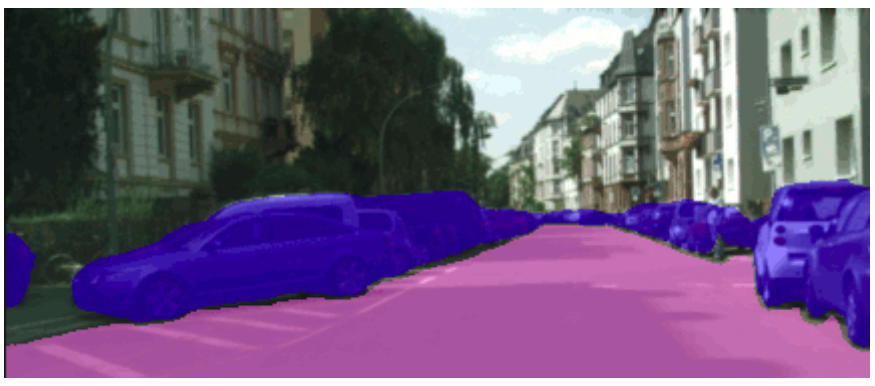


Image source: jeremyjordan.me

Usually, in an image with various entities, we want to know which pixel belongs to which entity, For example in an outdoor image, we can segment the sky, ground, trees, people, etc.

Semantic segmentation is different from object detection as it does not predict any bounding boxes around the objects. We do not distinguish between different instances of the same object. For example, there could be multiple cars in the scene and all of them would have the same label.



An example where there are multiple instances of the same object class

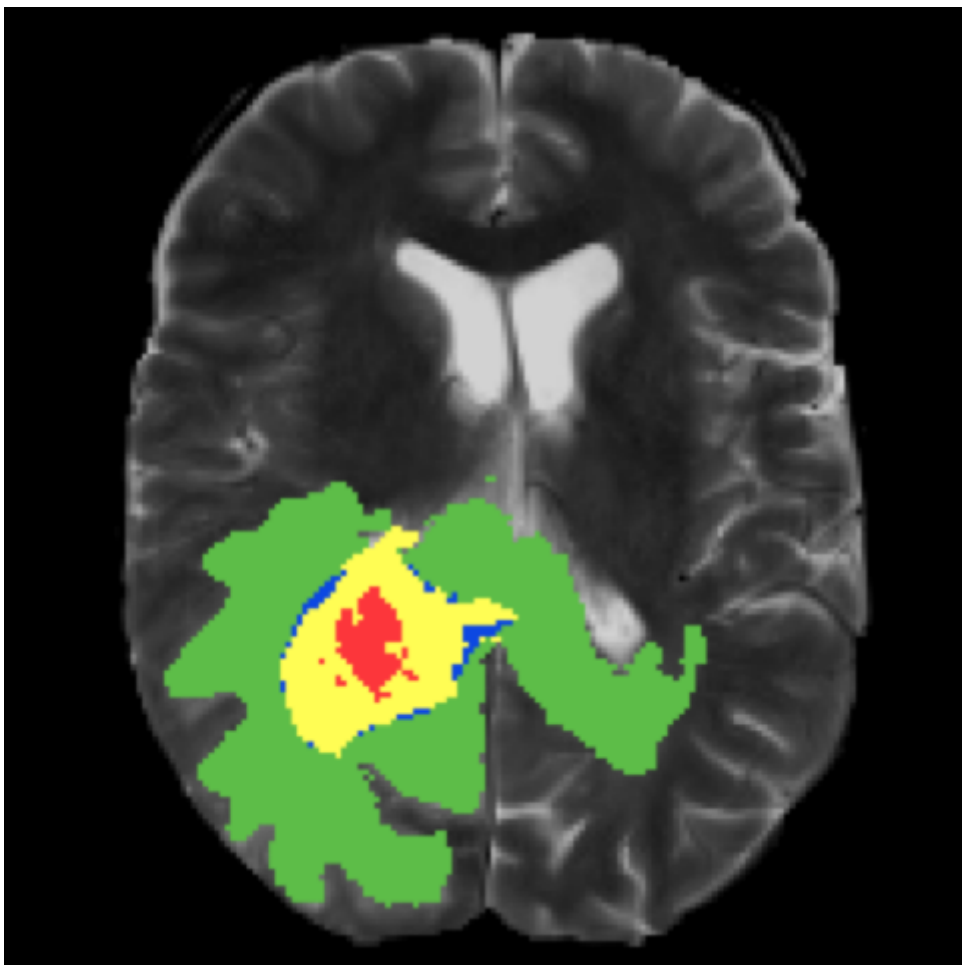
In order to perform semantic segmentation, a higher level understanding of the image is required. The algorithm should figure out the objects present and also the pixels which correspond to the object. Semantic segmentation is one of the essential tasks for complete scene understanding.

Applications

There are several applications for which semantic segmentation is very useful.

Medical images

Automated segmentation of body scans can help doctors to perform diagnostic tests. For example, models can be trained to segment tumor.



Tumor segmentation of brain MRI scan. Image source

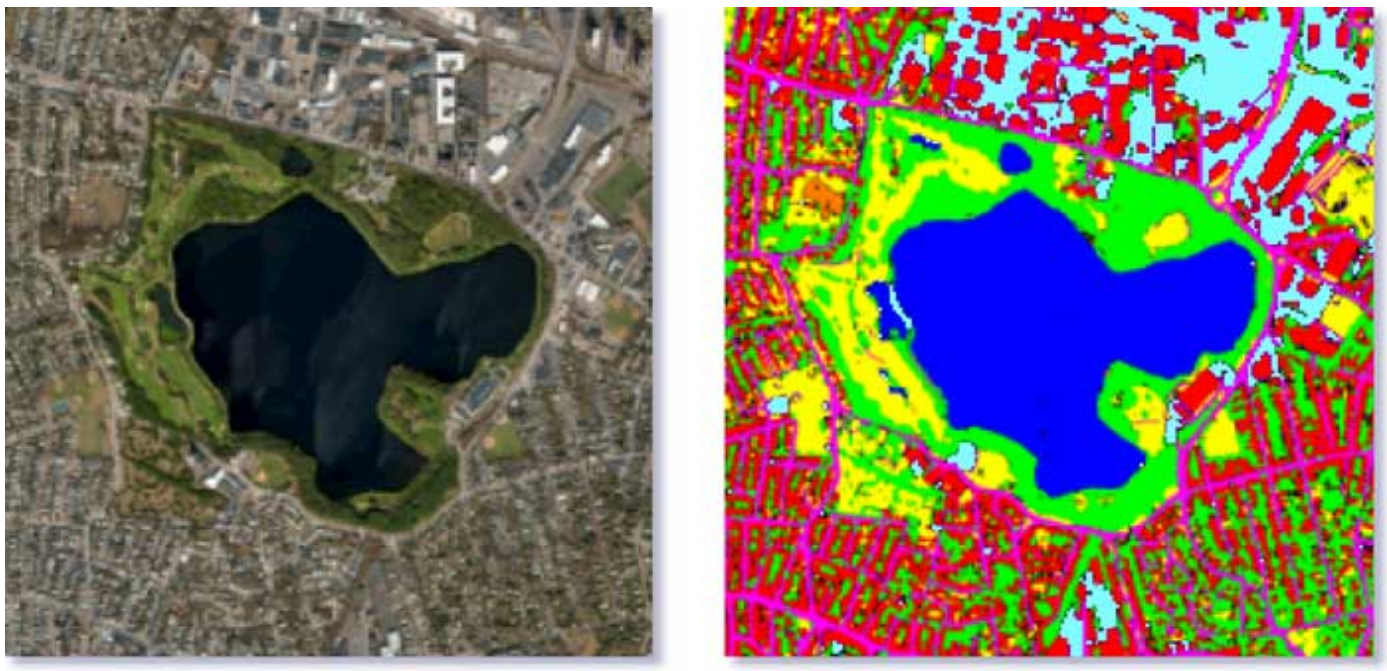
Autonomous vehicles

Autonomous vehicles such as self-driving cars and drones can benefit from automated segmentation. For example, self-driving cars can detect drivable regions.

Segmentation of a road scene Image source

Satellite image analysis

Aerial images can be used to segment different types of land. Automated land mapping can also be done.



Segmentation of a satellite image Image source

Image segmentation using deep learning

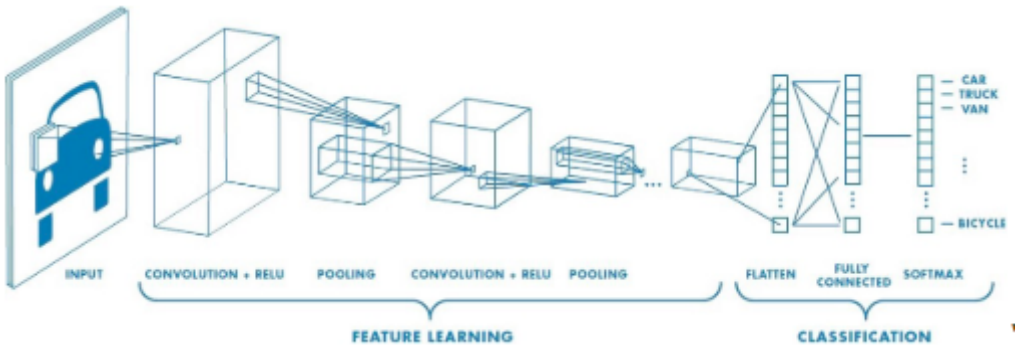
Like most of the other applications, using a CNN for semantic segmentation is the obvious choice. When using a CNN for semantic segmentation, the output is also an image rather than a fixed length vector.

Convolutional neural networks for segmentation

Usually, the architecture of the model contains several convolutional layers, non-linear activations, batch normalization, and pooling layers. The initial layers learn the low-level concepts such as edges and colors and the later level layers learn the higher level concepts such as different objects.

At a lower level, the neurons contain information for a small region of the image, whereas at a higher level the neurons contain information for a large region of the image. Thus, as we add more layers, the size of the image keeps on decreasing and the number of channels keeps on increasing. The downsampling is done by the pooling layers.

For the case of image classification, we need to map the spatial tensor from the convolution layers to a fixed length vector. To do that, fully connected layers are used, which destroy all the spatial information.

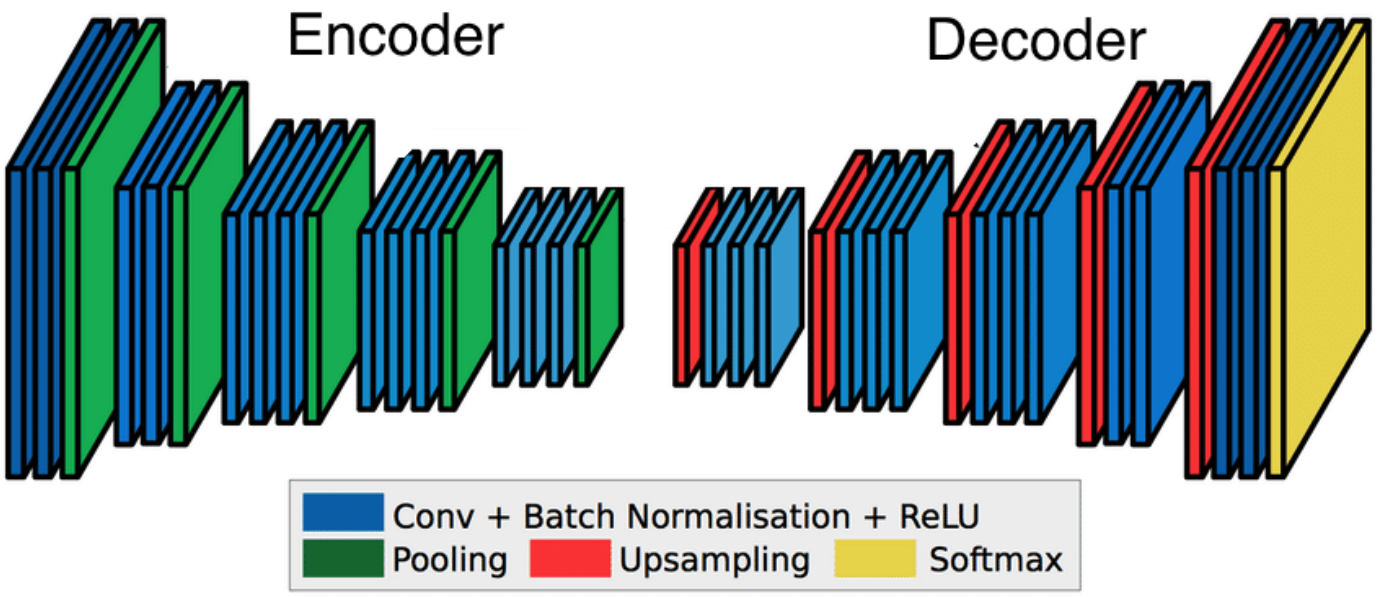


Spatial tensor is downsampled and converted to a vector Image source

For the task of semantic segmentation, we need to retain the spatial information, hence no fully connected layers are used. That's why they are called **fully convolutional networks**. The convolutional layers coupled with downsampling layers produce a low-resolution tensor containing the high-level information.

Taking the low-resolution spatial tensor, which contains high-level information, we have to produce high-resolution segmentation outputs. To do that we add more convolution layers coupled with upsampling layers which increase the size of the spatial tensor. As we increase the resolution, we decrease the number of channels as we are getting back to the low-level information.

This is called an **encoder-decoder** structure. Where the layers which downsample the input are the part of the encoder and the layers which upsample are part of the decoder.



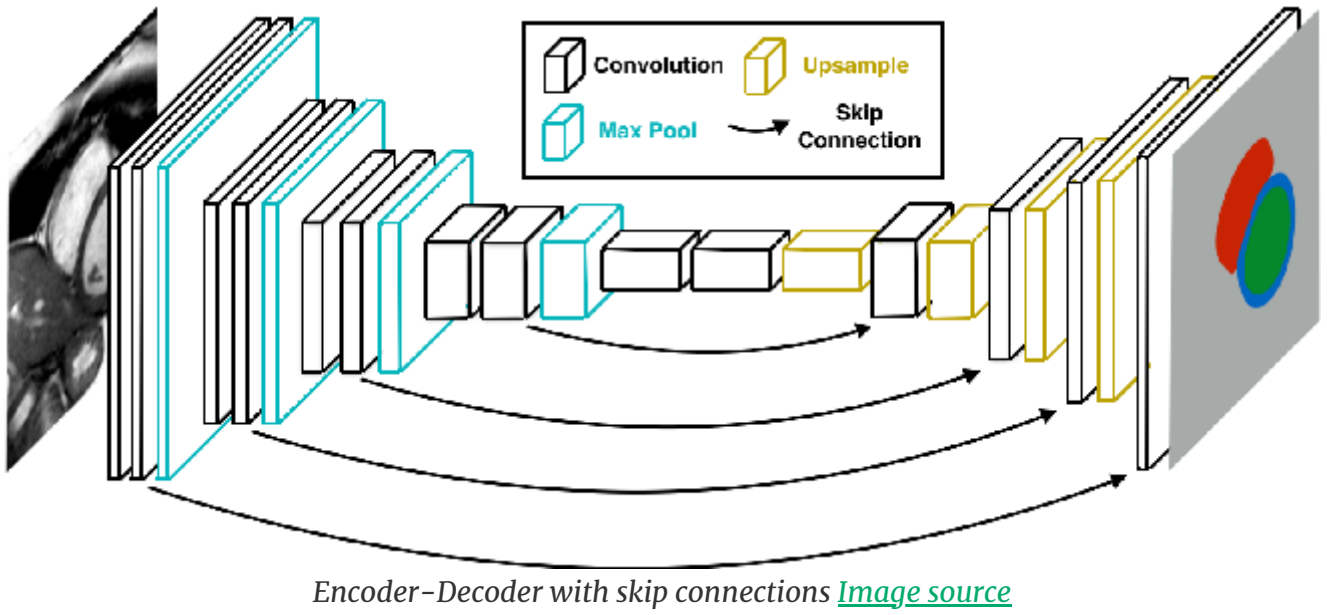
Encoder-Decoder architecture Image source

When the model is trained for the task of semantic segmentation, the encoder outputs a tensor containing information about the objects, and its shape and size. The decoder takes this information and produces the segmentation maps.

Skip connections

If we simply stack the encoder and decoder layers, there could be loss of low-level information. Hence, the boundaries in segmentation maps produced by the decoder could be inaccurate.

To make up for the information lost, we let the decoder access the low-level features produced by the encoder layers. That is accomplished by **skip connections**. Intermediate outputs of the encoder are added/concatenated with the inputs to the intermediate layers of the decoder at appropriate positions.

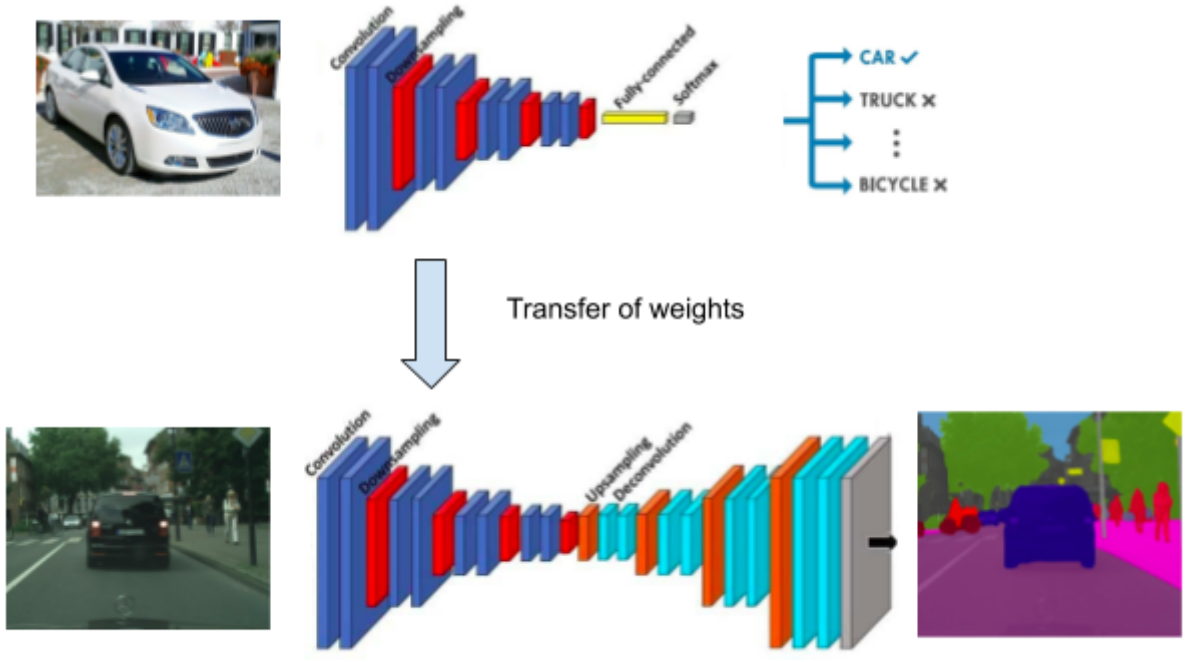


Encoder-Decoder with skip connections Image source

The skip connections from the earlier layers provide the necessary information to the decoder layers which is required for creating accurate boundaries.

Transfer learning

The CNN models trained for image classification contain meaningful information which can be used for segmentation as well. We can re-use the convolution layers of the pre-trained models in the encoder layers of the segmentation model. Using Resnet or VGG pre-trained on ImageNet dataset is a popular choice. You can read more about transfer learning here.



Transfer learning for Segmentation

Loss function

Each pixel of the output of the network is compared with the corresponding pixel in the ground truth segmentation image. We apply standard cross-entropy loss on each pixel.

Implementation

We will be using Keras for building and training the segmentation models. First, install keras_segmentation which contains all the utilities required.

```
pip install keras-segmentation
```

Dataset

The first step in training our segmentation model is to prepare the dataset. We would need the input RGB images and the corresponding segmentation images. If you want to make your own dataset, a tool like labelme or GIMP can be used to manually generate the ground truth segmentation masks.

Assign each class a unique ID. In the segmentation images, the pixel value should denote the class ID of the corresponding pixel. This is a common format used by most of the datasets and keras_segmentation. For the segmentation

maps, do not use the jpg format as jpg is lossy and the pixel values might change. Use bmp or png format instead. And of course, the size of the input image and the segmentation image should be the same.

In the following example, pixel (0,0) is labeled as class 2, pixel (3,4) is labeled as class 1 and rest of the pixels are labeled as class 0.

```
import cv2
import numpy as np

ann_img = np.zeros((30,30,3)).astype('uint8')
ann_img[ 3 , 4 ] = 1 # this would set the label of pixel 3,4 as 1
ann_img[ 0 , 0 ] = 2 # this would set the label of pixel 0,0 as 2

cv2.imwrite( "ann_1.png" ,ann_img )
```

After generating the segmentation images, place them in the training/testing folder. Make separate folders for input images and the segmentation images. The file name of the input image and the corresponding segmentation image should be the same.

Refer to the format below :

```
dataset/
        train_images/
                - img0001.png
                - img0002.png
                - img0003.png
        train_segmentation/
                - img0001.png
                - img0002.png
                - img0003.png
        val_images/
                - img0004.png
                - img0005.png
                - img0006.png
        val_segmentation/
                - img0004.png
                - img0005.png
                - img0006.png
```

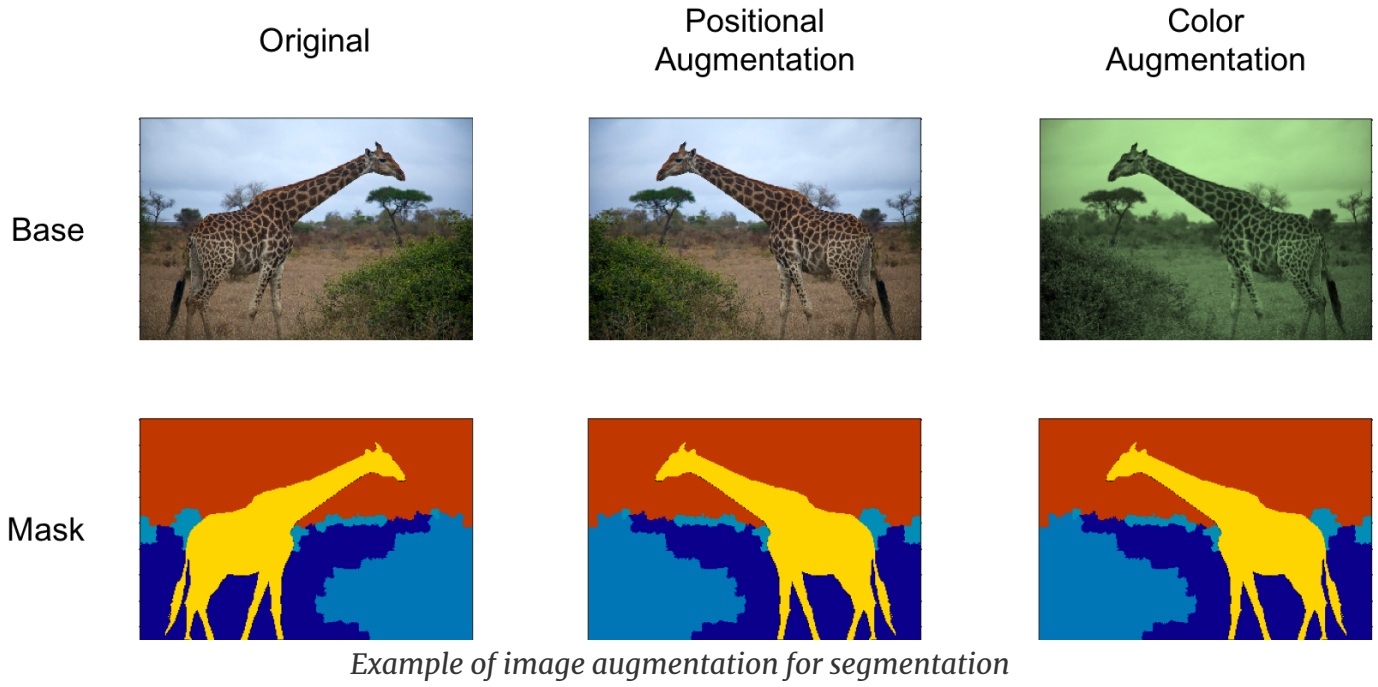
You can refer to a sample dataset here.

For this tutorial we would be using a data-set which is already prepared. You can download it from here.

Data augmentation

If you have less number of training pairs, the results might not be good be because the model might overfit. We can increase the size of the dataset by applying random transformations on the images. We can change the color properties like hue, saturation, brightness, etc of the input images. We can also

apply transformations such as rotation, scale, and flipping. For the transformations which change the location of the pixels, the segmentation image should also be transformed the same way.



Example of image augmentation for segmentation

Imgaug is an amazing tool to perform image augmentation. Refer to the code snippet below which would apply Crop, Flip and GaussianBlur transformation randomly.

```
import imgaug as ia
import imgaug.augmenters as iaa

seq = iaa.Sequential([
    iaa.Crop(px=(0, 16)), # crop images from each side by 0 to 16px (randomly chosen)
    iaa.Fliplr(0.5), # horizontally flip 50% of the images
    iaa.GaussianBlur(sigma=(0, 3.0)) # blur images with a sigma of 0 to 3.0
])

def augment_seg( img , seg  ):

        aug_det = seq.to_deterministic()
        image_aug = aug_det.augment_image( img )

        segmap = ia.SegmentationMapOnImage( seg , nb_classes=np.max(seg)+1 ,
shape=img.shape )
        segmap_aug = aug_det.augment_segmentation_maps( segmap )
        segmap_aug = segmap_aug.get_arr_int()

        return image_aug , segmap_aug
```

Here `aug_det` defines the parameters of the transformation, which is applied both to input image `img` and the segmentation image `seg`.

After preparing the dataset, you might want to verify it and also visualize it.

```
python -m keras_segmentation verify_dataset \
 --images_path="dataset_path/images_prepped_train/" \
 --segs_path="dataset_path/annotations_prepped_train/"  \
 --n_classes=50
```

```
python -m keras_segmentation visualize_dataset \
 --images_path="dataset_path/images_prepped_train/" \
 --segs_path="dataset_path/annotations_prepped_train/"  \
 --n_classes=50
```

Building the model

Now, let's use the Keras API to define our segmentation model with skip connections.

Let's define the encoder layers. Here, each block contains two convolution layers and one max pooling layer which would downsample the image by a factor of two.

```
img_input = Input(shape=(input_height,input_width , 3 ))

conv1 = Conv2D(32, (3, 3), activation='relu', padding='same')(img_input)
conv1 = Dropout(0.2)(conv1)
conv1 = Conv2D(32, (3, 3), activation='relu', padding='same')(conv1)
pool1 = MaxPooling2D((2, 2))(conv1)

conv2 = Conv2D(64, (3, 3), activation='relu', padding='same')(pool1)
conv2 = Dropout(0.2)(conv2)
conv2 = Conv2D(64, (3, 3), activation='relu', padding='same')(conv2)
pool2 = MaxPooling2D((2, 2))(conv2)
```

`conv1` and `conv2` contain intermediate the encoder outputs which will be used by the decoder. `pool2` is the final output of the encoder.

Let's define the decoder layers. We concatenate the intermediate encoder outputs with the intermediate decoder outputs which are the skip connections.

```
conv3 = Conv2D(128, (3, 3), activation='relu', padding='same')(pool2)
conv3 = Dropout(0.2)(conv3)
conv3 = Conv2D(128, (3, 3), activation='relu', padding='same')(conv3)

up1 = concatenate([UpSampling2D((2, 2))(conv3), conv2], axis=-1)
conv4 = Conv2D(64, (3, 3), activation='relu', padding='same')(up1)
conv4 = Dropout(0.2)(conv4)
conv4 = Conv2D(64, (3, 3), activation='relu', padding='same')(conv4)

up2 = concatenate([UpSampling2D((2, 2))(conv4), conv1], axis=-1)
conv5 = Conv2D(32, (3, 3), activation='relu', padding='same')(up2)
conv5 = Dropout(0.2)(conv5)
conv5 = Conv2D(32, (3, 3), activation='relu', padding='same')(conv5)
```

Here `conv1` is concatenated with `conv4`, and `conv2` is concatenated with `conv3`. To get the final outputs, add a convolution with filters the same as the number of classes. (similar to what we do for classification) .

```
out = Conv2D( n_classes, (1, 1) , padding='same')(conv5)

from keras_segmentation.models.model_utils import get_segmentation_model

model = get_segmentation_model(img_input ,  out ) # this would build the segmentation model
```

> *keras_segmentation contains several ready to use models, hence you don't need to write your own model when using an off-the-shelf one.*

Choosing the model

There are several models available for semantic segmentation. The model architecture shall be chosen properly depending on the use case. There are several things which should be taken into account:

1. The number of training images
2. Size of the images
3. The domain of the images

Usually, deep learning based segmentation models are built upon a base CNN network. A standard model such as ResNet, VGG or MobileNet is chosen for the base network usually. Some initial layers of the base network are used in the encoder, and rest of the segmentation network is built on top of that. For most of the segmentation models, any base network can be used.

Choosing the base model

For selecting the segmentation model, our first task is to select an appropriate base network. For many applications, choosing a model pre-trained on ImageNet is the best choice.

ResNet: This is the model proposed by Microsoft which got 96.4% accuracy in the ImageNet 2016 competition. ResNet is used as a pre-trained model for several applications. ResNet has large number of layers along with residual connections which make it's training feasible.

VGG-16: This is the model proposed by Oxford which got 92.7% accuracy in the ImageNet 2013 competition. Compared to Resnet it has lesser layers, hence it is much faster to train. For most of the existing segmentation benchmarks, VGG does not perform as good as ResNet in terms of accuracy. Before ResNet, VGG was the standard pre-trained model in for a large number of applications.

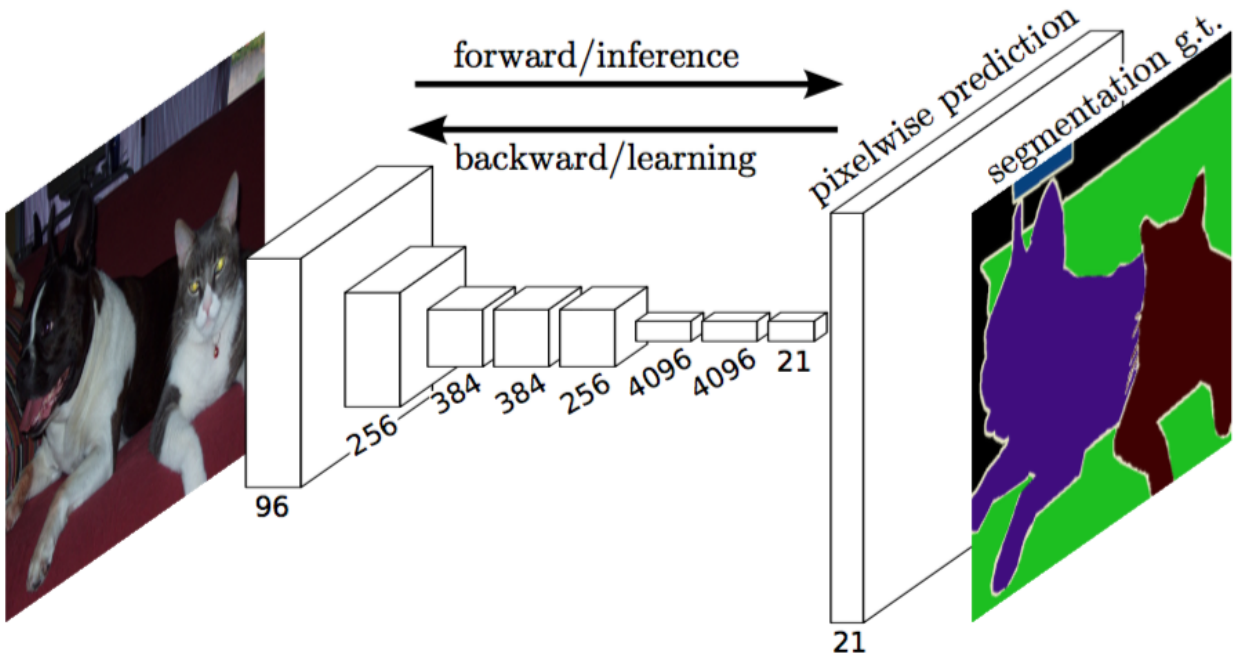
MobileNet: This model is proposed by Google which is optimized for having a small model size and faster inference time. This is ideal to run on mobile phones and resource-constrained devices. Due to the small size, there could be a small hit in the accuracy of the model.

Custom CNN: Apart from using an ImageNet pre-trained model, a custom network can be used as a base network. If the segmentation application is fairly simple, ImageNet pre-training is not necessary. Another advantage of using a custom base model is that we can customize it according to the application.

If the domain of the images for the segmentation task is similar to ImageNet then ImageNet pre-trained models would be beneficial. For input images of indoor/ outdoor images having common objects like cars, animals, humans, etc ImageNet pre-training could be helpful. The pre-trained model can also be trained on other datasets depending on the type of input images for the task.

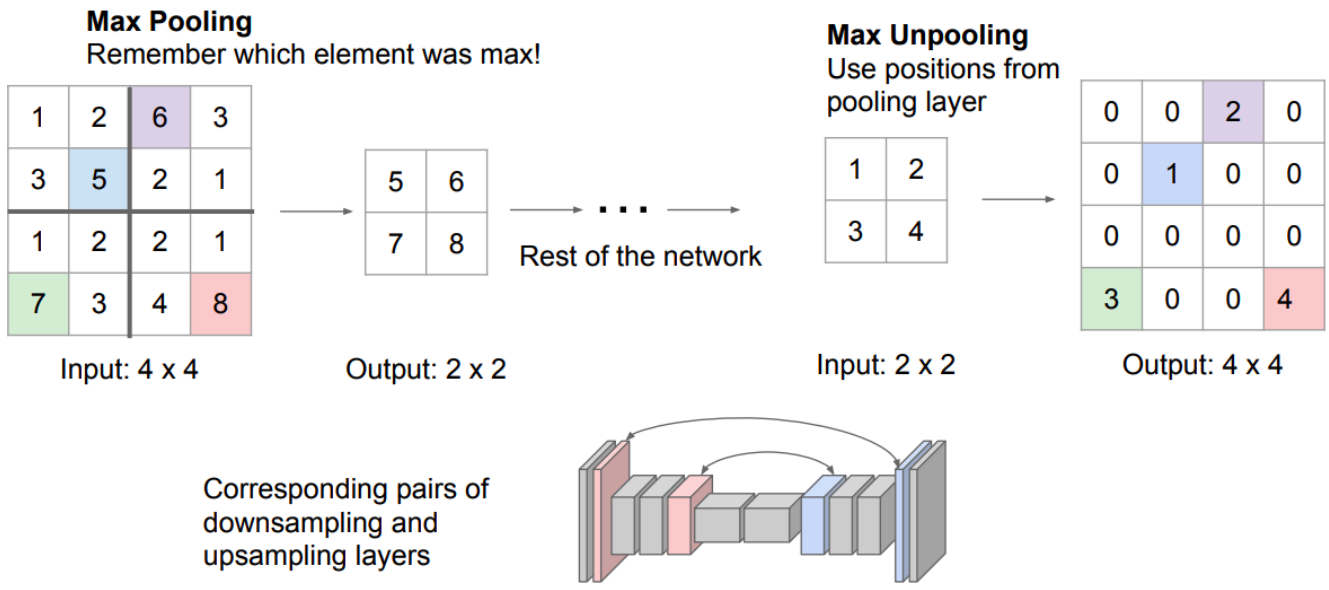
After selecting the base network we have to select the segmentation architecture. Let's go over some popular segmentation models.

FCN : FCN is one of the first proposed models for end-to-end semantic segmentation. Here standard image classification models such as VGG and AlexNet are converted to fully convolutional by making FC layers 1x1 convolutions. At FCN, transposed convolutions are used to upsample, unlike other approaches where mathematical interpolations are used. The three variants are FCN8, FCN16 and FCN32. In FCN8 and FCN16, skip connections are used.

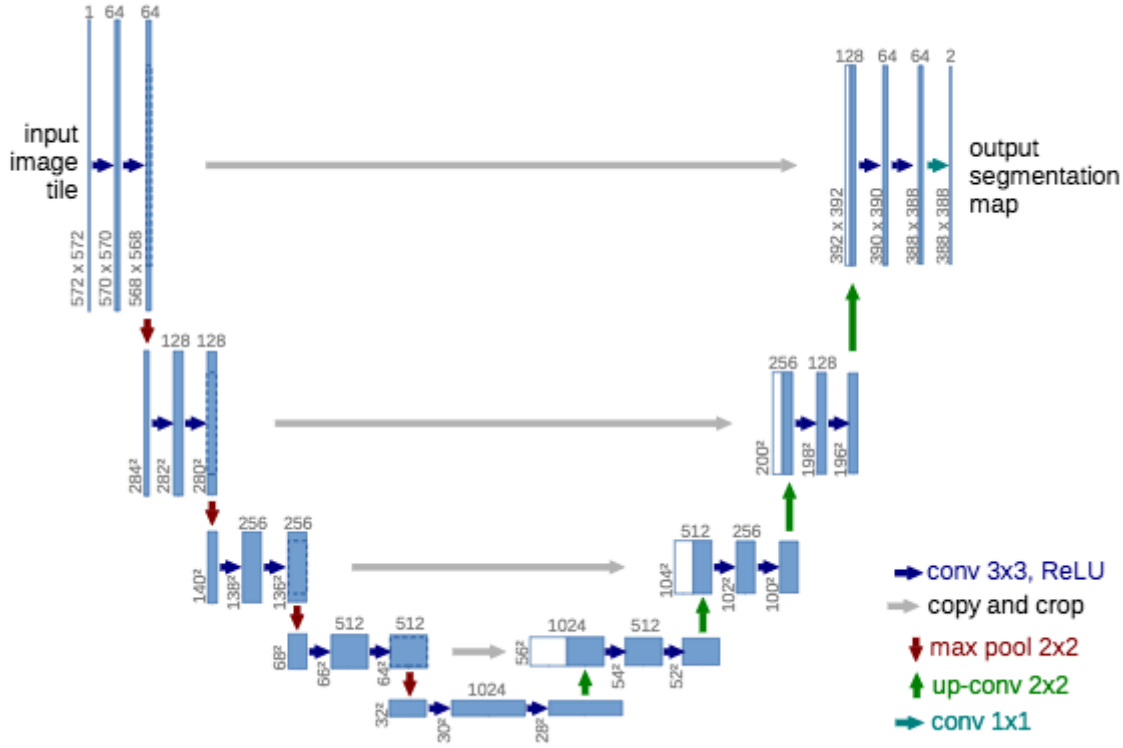


Architecture of FCN32 Image source

SegNet : The SegNet architecture adopts an encoder-decoder framework. The encoder and decoder layers are symmetrical to each other. The upsampling operation of the decoder layers use the max-pooling indices of the corresponding encoder layers. SegNet does not have any skip connections. Unlike FCN, no learnable parameters are used for upsampling.

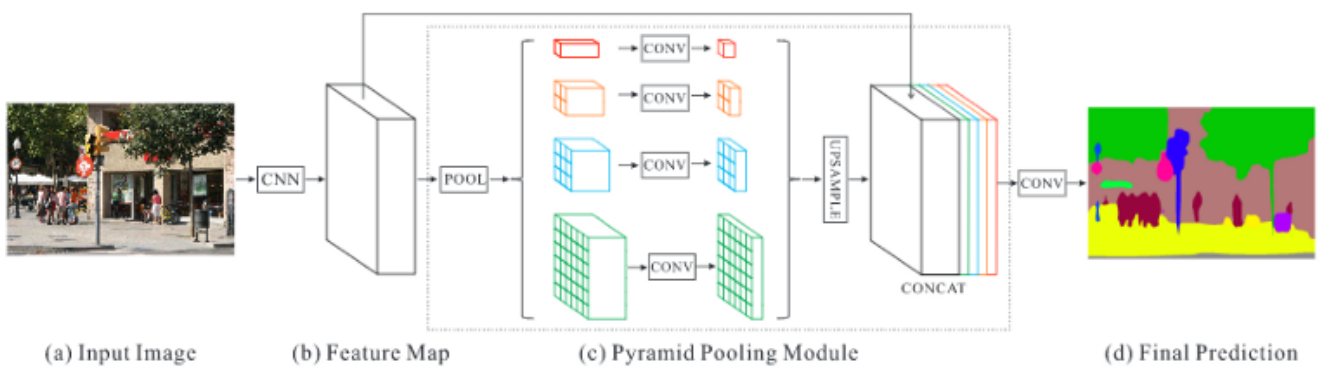


UNet : The UNet architecture adopts an encoder-decoder framework with skip connections. Like SegNet, the encoder and decoder layers are symmetrical to each other.



Architecture of UNet Image source

PSPNet : The Pyramid Scene Parsing Network is optimized to learn better global context representation of a scene. First, the image is passed to the base network to get a feature map. The the feature map is downsampled to different scales. Convolution is applied to the pooled feature maps. After that, all the feature maps are upsampled to a common scale and concatenated together. Finally a another convolution layer is used to produce the final segmentation outputs. Here, the smaller objects are captured well by the features pooled to a high resolution, whereas the large objects are captured by the features pooled to a smaller size.



Architecture of PSPNet Image source

For images containing indoor and outdoor scenes, PSPNet is preferred, as the objects are often present in different sizes. Here the model input size should be fairly large, something around 500x500.

For the images in the medical domain, UNet is the popular choice. Due to the skip connections, UNet does not miss out the tiny details. UNet could also be useful for indoor/outdoor scenes with small size objects.

For simple datasets, with large size and a small number of objects, UNet and PSPNet could be an overkill. Here simple models such as FCN or Segnet could be sufficient.

It is best advised to experiment with multiple segmentation models with different model input sizes.

If you don't want to write your own model, you can import ready to use models from keras_segmentation.

```
from keras_segmentation.models.unet import vgg_unet


model = vgg_unet(n_classes=51 ,  input_height=416, input_width=608  )
```

Choosing the input size

Apart from choosing the architecture of the model, choosing the model input size is also very important. If there are a large number of objects in the image, the input size shall be larger. In some cases, if the input size is large, the model should have more layers to compensate. The standard input size is somewhere from 200x200 to 600x600. A model with a large input size consumes more GPU memory and also would take more time to train.

Training

After preparing the dataset and building the model we have to train the model.

```
model.train(
    train_images =  "dataset_path/images_prepped_train/",
    train_annotations = "dataset_path/annotations_prepped_train/",
    checkpoints_path = "checkpoints/vgg_unet_1" , epochs=5
)
```

Here, `dataset` is the directory of the training images and `checkpoints` is the directory where all the model weights would be saved.

Now we can see the output of the model on a new image which is not present in the training set.

```
out = model.predict_segmentation(
    inp="dataset_path/images_prepped_test/0016E5_07965.png",
    out_fname="output.png"
)
```

We can also get predictions from a saved model, which would automatically load the model and with the weights.

```
from keras_segmentation.predict import predict


predict(
        checkpoints_path="checkpoints/vgg_unet_1",
        inp="dataset_path/images_prepped_test/0016E5_07965.png",
        out_fname="output.png"
)
```

To get predictions of multiple images in a directory.

```
from keras_segmentation.predict import predict_multiple


predict_multiple(
        checkpoints_path="checkpoints/vgg_unet_1",
        inp_dir="dataset_path/images_prepped_test/",
        out_dir="outputs/"
)
```

Conclusion

In this post, we discussed the concepts of deep learning based segmentation. We then discussed various popular models used. Using Keras, we implemented the complete pipeline to train segmentation models on any dataset. We discussed how to choose the appropriate model depending on the application. If you have any questions or want to suggest any changes feel free to contact me via twitter or write a comment below.

06 Jun 2019

image-segmentation

« An Introduction to Virtual Adversarial Training

An Introduction to Pseudo-semi-supervised Learning for Unsupervised Clustering »