SEMANTIC SEGMENTATION

Semantic image segmentation seeks to assign a class to each pixel in an image that corresponds to the concept it is intended to convey. This task is sometimes known as "dense prediction" because we are making predictions for each and every pixel in the image.

The fact that we are just interested in the category of each pixel is crucial to keep in mind because we are not distinguishing between instances of the same class. To put it another way, the segmentation map does not automatically recognize two objects of the same category as independent ones if they are present in the input image. Instance segmentation models are a different class of models that can discriminate between different items belonging to the same class.

Each "block" in the architecture is represented by a set of convolution operations in the typical U-Net model. There are a variety of more sophisticated "blocks" that can be used instead of stacked convolutional layers, as I covered in my piece on popular convolutional network topologies.

Drozdzal et al. substitute residual blocks for the fundamental stacked convolution blocks. In addition to the long skip connections that already exist between the relevant feature maps of the encoder and decoder modules in the normal U-Net layout, this residual block also includes short skip connections (inside the block). They claim that training can be completed more quickly and with deeper models because to the brief skip connections.

Jegou et al. elaborated on this by suggesting the use of dense blocks, still having a U-Net structure, saying that the "characteristics of DenseNets make them a very good fit for semantic segmentation as they naturally produce skip connections and multi-scale supervision." These dense blocks are advantageous because they allow for very effective feature reuse by carrying lower level features from earlier layers side by side with higher level features from more recent layers.

Facebook AI Research (FAIR) came up with this advanced library, which gave amazing results on object detection and segmentation problems. Detectron2 is based upon the maskrcnn benchmark. Its implementation is in PyTorch. It requires CUDA due to the heavy computations involved.It supports multiple tasks such as bounding box detection, instance segmentation, keypoint detection, densepose detection, and so on. It provides pre-trained models which you can easily load

Many pre-trained models of Detectron2 can be accessed at [model zoo](https://github.com/facebookresearch/detectron2/blob/master/MODEL_ZOO.md). These models have been trained on different datasets, and are ready to be used.

Even when people are training their custom dataset, they use these pre-trained weights to initialize their model. It has proven to reduce the training time and improve the performance. The model we’ll be using is pretrained on the COCO dataset.

First, we have to define the complete configuration of the object detection model. We imported the ‘get\_cfg’ function from the detectron2.config module, we will be using it now. I have chosen the Coco Instance segmentation configuration (YAML file). There are other options available too. You also have to set the model’s threshold score (usually set between 0.4 to 0.6). You can load the pretrained weights for the configuration from the checkpoint