Semantic Segmentation Using Separable Convolutional Networks An Implementation of “Fast-SCNN: Fast Semantic Segmentation network”.

1. Introduction

Semantic Segmentation has been one of the most seminal applications of Deep learning. Image Segmentation is an integral part of scene understanding. Researchers are coming up with better and better models all the time. The aim of semantic segmentation is to categorize and label individual pixels of an image into its respective class or object. In contrast to the object recognition where we predict to which class an image may belong, we predict to which class each pixel belongs. Since we predict every pixel, we can also say that semantic segmentation is dense prediction.

Example of Semantic Segmentation task:

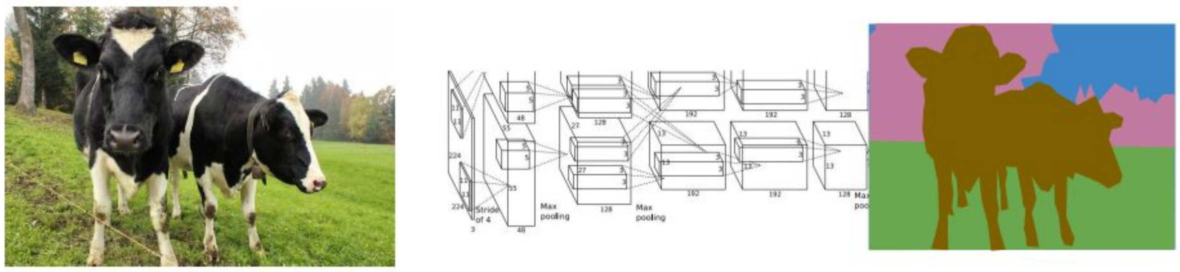
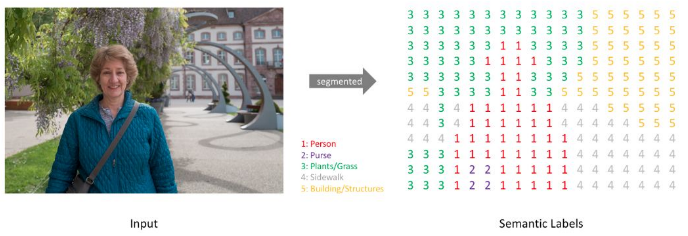


Figure 1: Semantic Segmentation using Convolutional Neural Networks. Source: Stanford CS231n

The applications of Semantic Segmentations are pervasive in the field of Autonomous Vehicles, Medical Image Diagnosis, Anomaly Detection, Satellite Imagery, Weather Analysis. Although the problem statement of Semantic Segmentation is quite old, the state of the art performance has evolved in recent years, specifically, through the development of Convolutional Neural Networks(CNNs). But there is a Caveat, training a CNN for the Semantic Segmentation is notoriously hard, and time consuming. And of course, like any other neural networks models, CNNs are also infamous for their overfitting if the proper measures are not undertaken. The output of the network is a mask of the size height x width x 1 where the height and width are the same as the input image but the input image may be color, i.e. 3 channels or grayscale, i.e. 1 channel. A pictorial representation is given below in Figure 2.



```
3 3 3 3 3 3 3 3 3 3 3 3 5 5 5 5 5 5
3 3 3 3 3 3 3 3 3 3 3 3 5 5 5 5 5 5
3 3 3 3 3 3 1 1 3 3 3 3 5 5 5 5 5 5
3 3 3 3 3 1 1 1 1 3 3 3 5 5 5 5 5 5
3 3 3 3 3 3 1 1 3 3 3 5 5 5 5 5 5 5
5 5 3 3 3 3 1 1 3 3 5 5 5 5 5 5 5 5
4 4 3 4 1 1 1 1 1 1 4 4 4 5 5 5 5 5
4 4 3 4 1 1 1 1 1 1 4 4 4 4 4 5 5 5
4 4 4 1 1 1 1 1 1 1 1 4 4 4 4 4 4 4
3 3 3 1 1 1 1 1 1 1 1 4 4 4 4 4 4 4
3 3 3 1 2 2 1 1 1 1 1 4 4 4 4 4 4 4
3 3 3 1 2 2 1 1 1 1 1 4 4 4 4 4 4 4
```

Figure 2: Input image and the mask Generated by the network. Source: Stanford CS231n

The mask generated by the network can also be represented as one-hot encoded labels of the class. In our implementation we consider 19 classes. The output of the network has 19 classes of the height and width of the input image. An example of this representation is as shown below in figure 3.

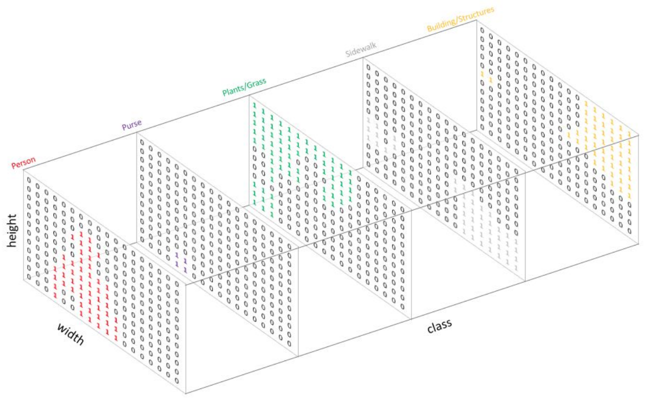


Figure 3: One-hot encoding of mask. Source: Stanford CS231n

The one-hot encoded image can be merged into a segmentation mask by taking the argmax of the depth wise vector.

Network Architecture

One way to build a network is to stack many convolutional layers and pad each layer to maintain the shape of the output as the input image. Over time, the network learns to produce the feature mask. As simple and good this may sound, this is not a practical approach as it is computationally quite taxing. One of the more practical approaches is to train an encoder/decoder model to produce the mask. In this type of model, we downsample the spatial resolution of the input image and develop the low level features which have the ability to distinguish between the classes and then upsample the features to the original size of the image to develop into a segmentation mask. One of the characteristic features of the Encoder-Decoder network is that they have multiple bridges to extract these features. An example is as shown in the below image in figure 4.1. But in this implementation of Fast-SCNN, the authors have used a much simpler two bridged approach. The architecture of the Fast-SCN is as shown in the figure 4.2 below.

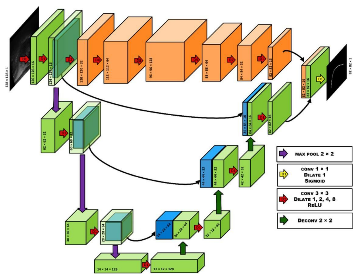

Max pool 2 × 2
Conv 1 × 1 Dilate 1 Sigmoid
Conv 3 × 3 Dilate 1, 2, 4, 8 ReLU
Deconv 2 × 2

Figure 4.1. Encoder-Decoder Network.

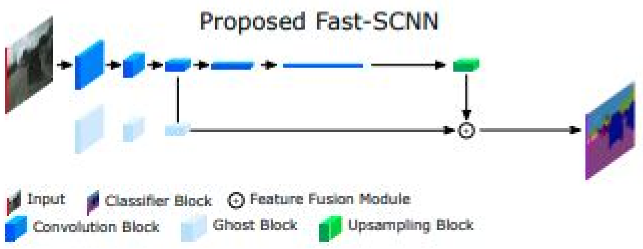


Figure 4.2. Fast-SCNN

Source: ScienceDirect Encoder-Decoder Model CNN Source: Fast-SCNN by Rudra PKP et. al.

The network can be better explained by breaking it down into four subcategories. Learning to Down-sample, Global Feature Extractor, Feature Fusion, Classifier. This is depicted in the below figure 5.

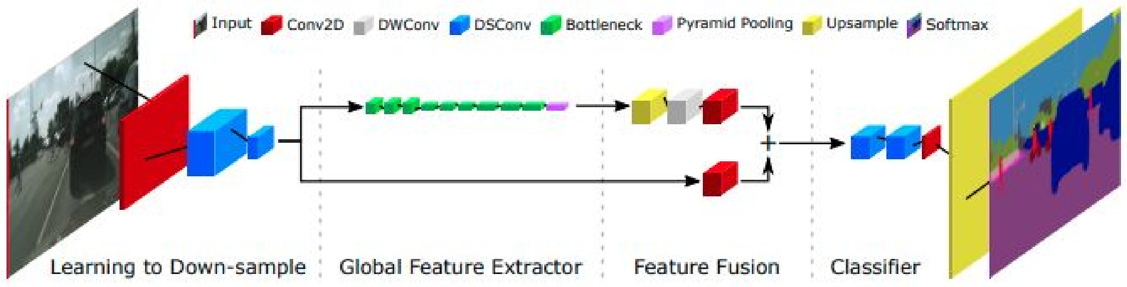


Figure 5. Subcategories of Fast-SCNN. Source: Fast-SCNN by Rudra PKP et. al.

1. **Learning to Down-sample:** The learning to downsample has 3 layers of convolutions. The first layer is the standard convolution with 32 filters of 3x3 size. The input to the first layer has only 3 channels, hence using the traditional convolution does not have huge computational impact. But for the second and third layer, separable convolutions are used as separable convolutions enjoy a huge computational boost when the depth of the channels increase. But there is a trade-off between the performance and the computational power when we use the separable convolutions. The goal of this block is to downsample the image to ensure the low level feature extraction.

2. **Global Feature Extractor:** From this block the input from previous layers split into two branches. This module takes the output of the previous block which is one-eighth of the input image and the separable convolutions derived from the architectures of MobileNet-V2 and Pyramid Pooling Modules are used to extract the low level features. The characteristic feature to be noted here is that due to efficient implementation of the Separable convolutions in all the layers results in the reduced floating point operations and lesser number of parameters. As a result of which, there is a computational performance improvement.

3. **Feature Fusion Module:** This block simply adds output of the two branches from the global feature extractor.

4. **Classifier:** The classifier block has two layers of the Separable convolutions. Adding more layers to the classifier can also boost the accuracy of the model. Stochastic Gradient Descent is used, so hence by extension softmax is used during the training.

Further details of the architecture such as dilations, strides, output dimension at each layer, name of each layer has been given in the table 1.

Input	Name	t	c	n	s
1024x 2048 x 3	Conv2D	-	32	1	2
512x1024x32	Sep. Conv.	-	48	1	2
256 x 512 x 48	Sep. Conv.	-	64	1	2
128 x 256 x 64	Bottleneck	6	64	3	2
64 x 128 x 64	Bottleneck	6	96	3	2
32 x 64 x 96	Bottleneck	6	238	3	1
32 x 64 x 128	PPM	-	128	-	-
32 x 64 x 128	Fusion	-	128	-	-
128 x 256 x 128	Sep. Conv	-	128	2	1
128 x 256 x 128	Conv2D	-	19	1	1

Table 1: Architecture parameters

Implementation Details

Hardwares and Hyperparameters:
The model has been trained on a Nvidia 2080 Ti for 100 epochs. A single CPU thread was used. The data was stored on a NVME M.2 Drive and hence the data accessing speed was increased due to which the 100 epochs were completed approximately 2 hours faster than training using data stored on a HDD. The model took one week to run 100 epochs. Model converged after 90 iterations. Stochastic Gradient Descent with momentum of 0.9. Although the paper implements a batch of 12, due to memory constraints, I have used 6 in this implementation. Although the Cross entropy loss was used, OHEM and Auxiliary losses have also been used in the implementation. The implementation of the OHEM and Auxiliary loss has been derived from the PyWick implementation. Dropout is used at the last layer. Aggressive data augmentation techniques also aids in avoiding overfitting of the model. BatchNormalization has been implemented before the activation function and ReLU is the activation function used.

Software Requirements:
The original paper has been implemented on TensorFlow by the authors. In this implementation I have used PyTorch. Loss functions and metrics from PyWick module implementations have been used as well. Although PyWick has not been imported, the loss and metrics implementation of the module has been used. Although the modules and interfaces used are python default, data augmentation has been implemented using imgaug module. The implementation environment is spyder IDE in Anaconda navigator.

Dataset:

Cityscapes dataset is one of the most used data sets. The data has training, validation and testing data. It is an open source data type. The data is available at the below link to download.

https://www.cityscapes-dataset.com/file-handling/?packageID=3.

In case the link does not work, Please navigate to the cityscapes website, download and download the 11gb leftImg8bit_trainvaltest data.

Results:

Figure 6.1 shows the input image to the network. 6.2 is the ground truth and the 6.3 is the mask generated by the network. In the paper the authors boast an accuracy rate of 90% pixel accuracy, 68% mean IoU class accuracy and 84.7% category mean IoU. In my paper, I have been able to produce a pixel accuracy of 80% and the mean IoU of 65%. By improving on the fusion modules and the upsampling techniques, I plan to enhance the accuracy rate more. In the Original paper implementation, the authors use the bilinear upsampling technique to upsample the image at the feature fusion blocks and after the classifier layer. I plan to further incorporate this project in my thesis work by adding the deconvolutions/ transposed convolutions for the upsampling.

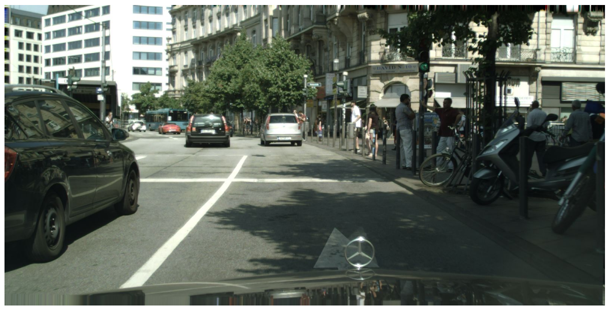


Figure 6.1. Input Image

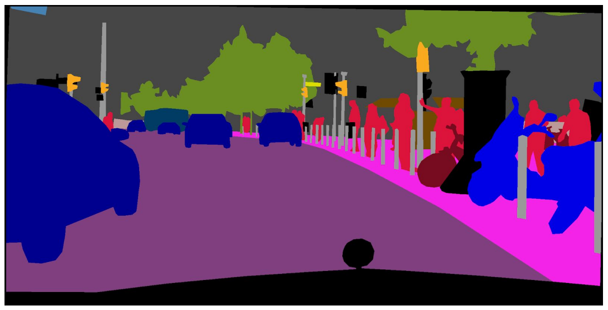


Figure 6.2 Ground Truth

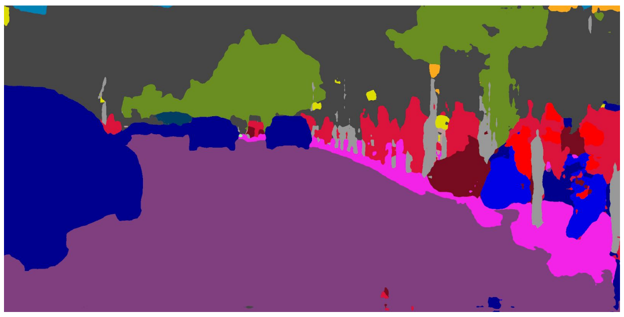


Figure 6.3. Predicted Mask