NC STATE UNIVERSITY

ECE 763 Project 3

SegNet: Deep Convolutional Encoder-Decoder Architecture for Image Segmentation

Aditya Rasam
200153631

Neeraj Vyas
200210310

Sreeraj Rajendran
200210462

Table of Contents

I. Introduction

The objective of this project is to study Semantic segmentation by selecting one of the well adopted deep architectures and then implement it in some other framework, with comparable performance, to have a better understanding of its architecture. Segmentation is based on image recognition, except the classifications occur at the pixel level as opposed to classifying entire images as with image recognition. Fundamentally segmentation models consist of a convolution and a deconvolution module. To make detail analysis of these modules we took the approach as follows.

To begin with, we selected a fully functional SegNet model with original implementation in CAFFE framework. Further, we converted the model into Keras with Theano backend with corresponding deep learning instruction set. Finally, we modified model parameters to achieve comparable results. The detail methodology and analysis are explained in the sections to follow.

II. Motivation

Semantic segmentation plays critical role in applications like scene understanding, inferring support relationships and autonomous driving. Segnet provides an efficient architecture with reliable results and is of high practical significance. It was decided to Implement the architecture in a different platform so that we get acquainted with the techniques of a state of the art model.

Before deep learning, approaches like Texton Forest and Random Forest based classifiers were used for semantic segmentation. Initial deep learning approaches were patch classification where each pixel was separately classified into classes using a patch of image around it. The results appear coarse, primarily because max pooling and sub-sampling reduce feature map resolution. One of the motivations for SegNet arises from this need to map low resolution features to input resolution for pixel-wise classification.

Road scene understanding applications require the ability to model appearance (road, building), shape (cars, pedestrians) and understand the spatial-relationship (context) between different classes such as road and side-walk. In typical road scenes, most of the pixels belong to large classes such as road, building and hence the network must produce smooth segmentations. The engine must also have the ability to delineate objects based on their shape despite their small size. Hence it is important to retain boundary information in the extracted image representation. From a computational perspective, it is necessary for the network to be efficient in terms of both memory and computation time during inference. The ability to train end-to-end to jointly optimize all the weights in the network using an efficient weight update technique such as stochastic gradient descent (SGD) is an additional benefit since it is more easily repeatable. The design of SegNet arose from a need to match these criteria.

III. Related works

While we implemented SegNet for this project, we also studied other works that were closely related to SegNet architecture. These models were FCN, DeepLab-LargFOV and DeconvNet. The encoder architecture for SegNet was derived from convolutional layer in VGG16. SegNet was developed by removing the Fully connected layers of VGG16 and converting it to fully convolutional layer.

IV. The Model

SegNet has a symetrical encoder-decoder architecture. Both the encoder and the decoder are made of convolutional blocks of two and three convolutional layers with a 3×3 kernel followed by batch normalization and rectified linear units (ReLU). ReLu is added to bring non-linearity to the architecture.

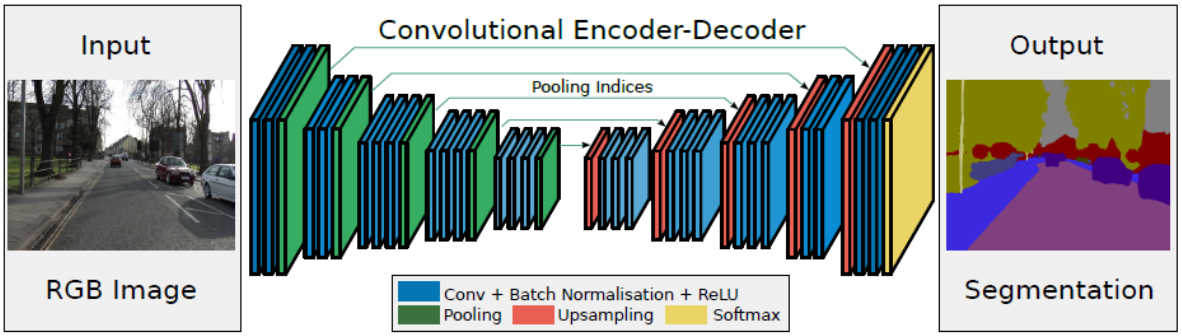


Figure 1: SegNet Architecture

Each block is then sent either into a max pooling layer (encoder) or an unpooling layer (decoder). To reduce dimensions and translation invariance the pooling operation is used. The unpooling operation replaces the pooling in the decoder and is the dual operation of the max pooling layer. It relocates the value of the activations into the mask of the maximum values computed at the pooling stage, which are fed-forward by a skip connection directly into the decoder. After pooling operation in the decoder side, we get a sparse feature map which is then densified by convolving with trainable decoder filter bank. This allows the network to upsample the feature activations coming out of the decoder up to the original input size, so that the final feature maps have the same dimensions as the input. Therefore, SegNet performs direct pixel-level inference.

As explained, SegNet architecture consists of the following:

i. Encoder network
ii. Decoder network
iii. Pixelwise classification layer

1. Encoder Network:

The encoder network consists of 13 convolutional layers which correspond to the first 13 convolutional layers in the VGG16 network designed for object classification. The training process can be initialized from weights trained for classification on large datasets. Each encoder in the encoder network performs convolution with a filter bank to produce a set of feature maps. These are then batch normalized. Then an element-wise rectified linear non-linearity (ReLU) max(0; x) is applied.

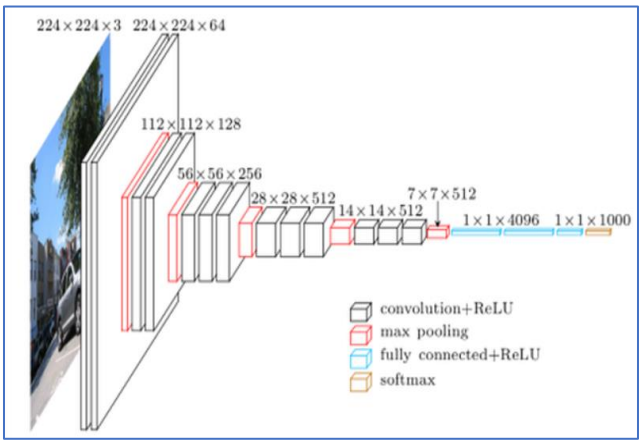


Figure 2: VGG 16 Encoder architecture.

A 2 x 2 max pooling window with stride of 2 is used and the resulting output is sub-sampled by a factor of 2. Max-pooling helps achieve translation invariance over small spatial shifts in the input image. Sub-sampling results in a large input image context (spatial window) for each pixel in the feature map.

Need to store Max pooling indices:

Storing of max pooling index is a novel technique proposed by Segnet. Several layers of max-pooling and sub-sampling can achieve more translation invariance for robust classification but correspondingly there is a loss of spatial resolution of the feature maps. Boundary delineation is vital for Segmentation tasks. . Therefore, it is necessary to capture and store boundary information in the encoder feature maps before sub-sampling is performed.

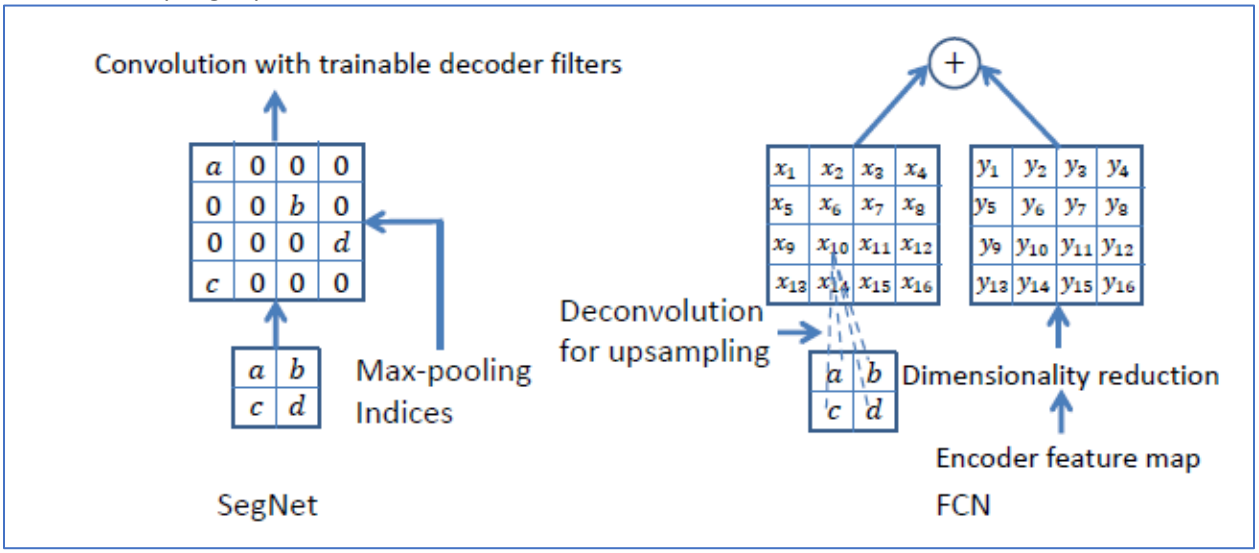


Figure 3: Max-pooling in SegNet

SegNet aims to optimize the memory usage and proposes an efficient way to store this information. It The max-pooling indices, i.e, the locations of the maximum feature value in each pooling window is memorized for each encoder feature map, are stored. This is done using 2 bits for each 2 x 2 pooling window and is thus much more efficient to store as compared to memorizing feature map(s) in float precision.

2. Decoder Network:

Each encoder layer has a corresponding decoder layer and hence the decoder network has 13 layers.

The appropriate decoder in the decoder network upsamples its input feature map(s) using the memorized max-pooling indices from the corresponding encoder feature map(s). This step produces sparse feature map(s). These feature maps are then convolved with a trainable decoder filter bank to produce dense feature maps. A batch normalization step is then applied to each of these maps. The decoder corresponding to the first encoder (closest to the input image) produces a multi-channel feature map. Other decoders in the network which produce feature maps with the same number of size and channels as their encoder inputs

3. Softmax Classification Layer:

The final decoder output is fed to a multi-class soft-max classifier to produce class probabilities for each pixel independently. The soft-max classifier is trainable and classifies each pixel independently. The output of the soft-max classifier is a K channel image of probabilities where K is the number of classes. The predicted segmentation corresponds to the class with maximum probability at each pixel.

4. Keras Implementation

The original paper was implemented in Caffe and our goal was to replicate the model in keras. We used Theano as backend for this. To train both encoder and decoder part it was taking lot of time. So, we used pretrained VGG-16 weights for our encoder part. For decoder part we tried two different topologies, one with thirteen convolution layer and other with five. Since they gave almost similar accuracy we opted for five convolution layer model. We then trained our model on the CamVid dataset to get our final weights for the model which we can use for the test image. We trained our model for different epochs and by tuning various hyper parameter to finally get an accuracy of almost 90%.

We have divided out implementation into four major section as follows: 1) Training, 2) Prediction, 3) Model definition, 4) Image processing. In training program various parameters are collected from the user and these input arguments are assigned to appropriate variables by parsing. Using these collected arguments, we then call our model definition code in which we have defined our model architecture. It also has predefined weights for VGG-16 model for our encoder. We then fit our dataset and finally get the weights for our entire model. In the prediction code we then call this weight and pass on the test data to get the semantic segmented images.

Layer (type)	Output Shape	Param #
input_1 (InputLayer)	(None, 3, 416, 608)	0
block1_conv1 (Conv2D)	(None, 64, 416, 608)	1792
block1_conv2 (Conv2D)	(None, 64, 416, 608)	36928
block1_pool (MaxPooling2D)	(None, 64, 208, 304)	0
block2_conv1 (Conv2D)	(None, 128, 208, 304)	73856
block2_conv2 (Conv2D)	(None, 128, 208, 304)	147584
block2_pool (MaxPooling2D)	(None, 128, 104, 152)	0
block3_conv1 (Conv2D)	(None, 256, 104, 152)	295168
block3_conv2 (Conv2D)	(None, 256, 104, 152)	590080
block3_conv3 (Conv2D)	(None, 256, 104, 152)	590080
block3_pool (MaxPooling2D)	(None, 256, 52, 76)	0
block4_conv1 (Conv2D)	(None, 512, 52, 76)	1180160
block4_conv2 (Conv2D)	(None, 512, 52, 76)	2359808
block4_conv3 (Conv2D)	(None, 512, 52, 76)	2359808

block4_conv2 (Conv2D)	(None, 512, 52, 76)	2359808
block4_conv3 (Conv2D)	(None, 512, 52, 76)	2359808
block4_pool (MaxPooling2D)	(None, 512, 26, 38)	0
zero_padding2d_2 (ZeroPaddin	(None, 512, 28, 40)	0
conv2d_27 (Conv2D)	(None, 512, 26, 38)	2359808
batch_normalization_27 (Batc	(None, 512, 26, 38)	152
up_sampling2d_6 (UpSampling2	(None, 512, 52, 76)	0
zero_padding2d_3 (ZeroPaddin	(None, 512, 54, 78)	0
conv2d_28 (Conv2D)	(None, 256, 52, 76)	1179904
batch_normalization_28 (Batc	(None, 256, 52, 76)	304
up_sampling2d_7 (UpSampling2	(None, 256, 104, 152)	0
zero_padding2d_4 (ZeroPaddin	(None, 256, 106, 154)	0
conv2d_29 (Conv2D)	(None, 128, 104, 152)	295040
batch_normalization_29 (Batc	(None, 128, 104, 152)	608
up_sampling2d_8 (UpSampling2	(None, 128, 208, 304)	0
zero_padding2d_5 (ZeroPaddin	(None, 128, 210, 306)	0

```
_________________________________________________________________
batch_normalization_28 (Batc (None, 256, 52, 76)       304
_________________________________________________________________
up_sampling2d_7 (UpSampling2 (None, 256, 104, 152)     0
_________________________________________________________________
zero_padding2d_4 (ZeroPaddin (None, 256, 106, 154)     0
_________________________________________________________________
conv2d_29 (Conv2D)           (None, 128, 104, 152)     295040
_________________________________________________________________
batch_normalization_29 (Batc (None, 128, 104, 152)     608
_________________________________________________________________
up_sampling2d_8 (UpSampling2 (None, 128, 208, 304)     0
_________________________________________________________________
zero_padding2d_5 (ZeroPaddin (None, 128, 210, 306)     0
_________________________________________________________________
conv2d_30 (Conv2D)           (None, 64, 208, 304)      73792
_________________________________________________________________
batch_normalization_30 (Batc (None, 64, 208, 304)      1216
_________________________________________________________________
conv2d_31 (Conv2D)           (None, 101, 208, 304)     58277
_________________________________________________________________
reshape_2 (Reshape)          (None, 101, 63232)        0
_________________________________________________________________
permute_1 (Permute)          (None, 63232, 101)        0
_________________________________________________________________
activation_26 (Activation)   (None, 63232, 101)        0
=================================================================
Total params: 11,604,365
Trainable params: 11,603,225
Non-trainable params: 1,140
_________________________________________________________________
```

V. Results

The results are explained with help of randomly selected test images.

Test image 159

Test image 119

Test image 959

The model was trained with following test cases:

1. Test Case 1:

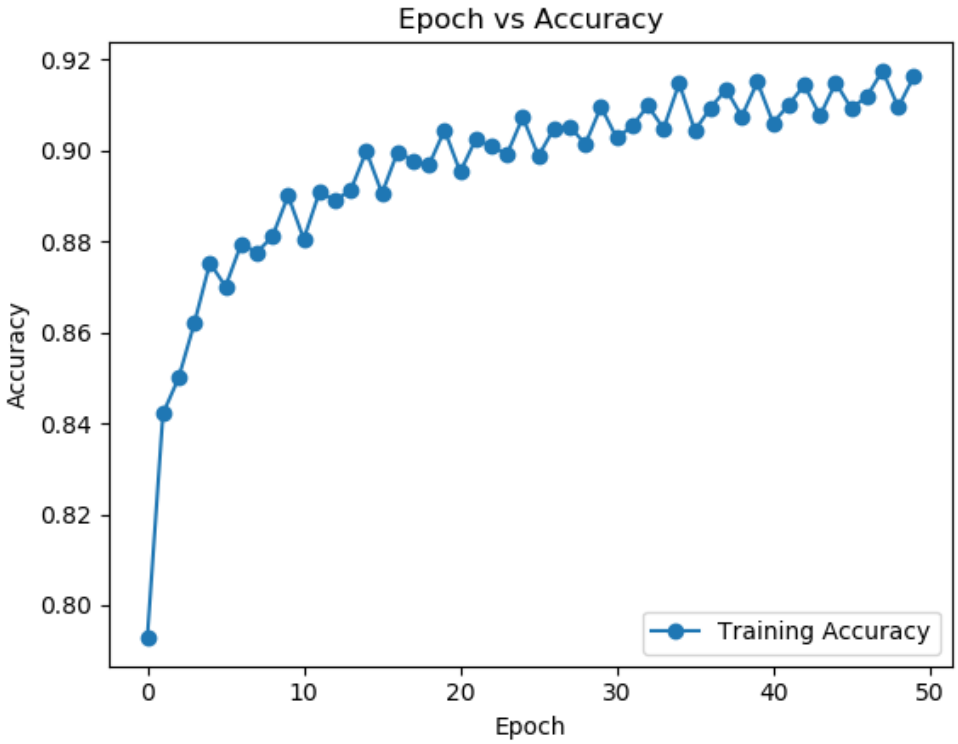
In this case we selected SGD as our optimizer and a learning rate of 0.001 was set. We trained the model with an epoch of 50 and the loss and accuracy after each epoch was as follows

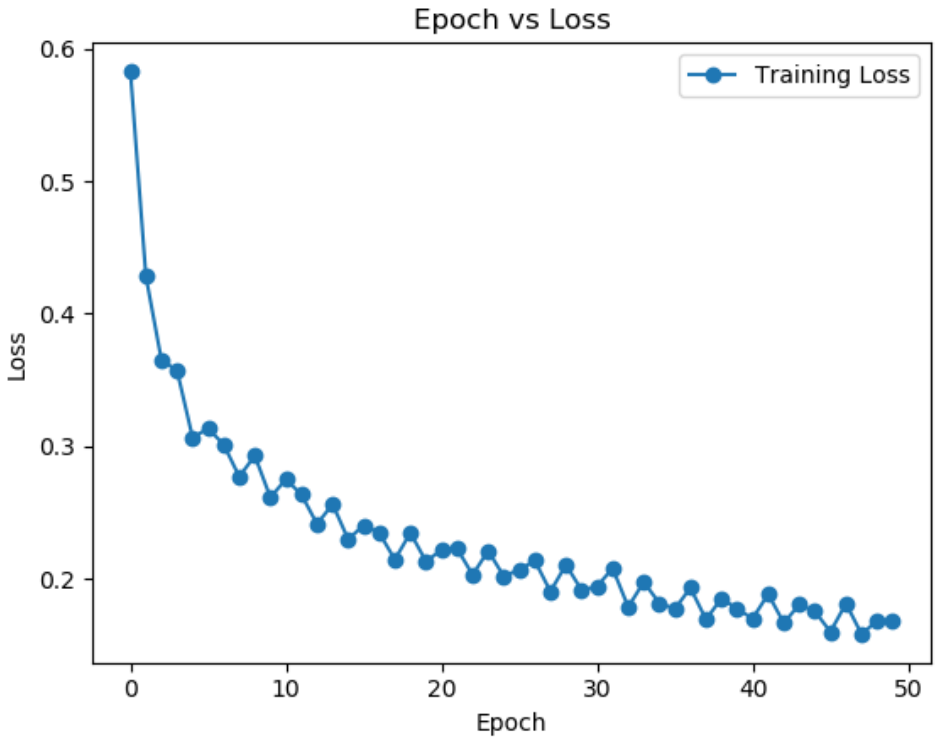
```
Epoch 1/50
512/512 [==============================] - 367s 717ms/step - loss: 0.5835 - acc: 0.7929
Epoch 2/50
512/512 [==============================] - 366s 716ms/step - loss: 0.4282 - acc: 0.8421
Epoch 3/50
512/512 [==============================] - 366s 716ms/step - loss: 0.3649 - acc: 0.8500
Epoch 4/50
512/512 [==============================] - 367s 716ms/step - loss: 0.3572 - acc: 0.8620
Epoch 5/50
512/512 [==============================] - 368s 718ms/step - loss: 0.3065 - acc: 0.8753
Epoch 6/50
512/512 [==============================] - 368s 718ms/step - loss: 0.3133 - acc: 0.8701
Epoch 7/50
512/512 [==============================] - 367s 717ms/step - loss: 0.3012 - acc: 0.8795
Epoch 8/50
512/512 [==============================] - 367s 717ms/step - loss: 0.2769 - acc: 0.8776
Epoch 9/50
512/512 [==============================] - 367s 716ms/step - loss: 0.2928 - acc: 0.8811
Epoch 10/50
512/512 [==============================] - 366s 716ms/step - loss: 0.2613 - acc: 0.8901
Epoch 11/50
512/512 [==============================] - 366s 716ms/step - loss: 0.2750 - acc: 0.8805
Epoch 12/50
512/512 [==============================] - 366s 715ms/step - loss: 0.2633 - acc: 0.8909
Epoch 13/50
512/512 [==============================] - 367s 716ms/step - loss: 0.2411 - acc: 0.8891
Epoch 14/50
512/512 [==============================] - 366s 716ms/step - loss: 0.2562 - acc: 0.8913
Epoch 15/50
512/512 [==============================] - 368s 718ms/step - loss: 0.2296 - acc: 0.8999
Epoch 16/50
512/512 [==============================] - 367s 717ms/step - loss: 0.2402 - acc: 0.8906
Epoch 17/50
512/512 [==============================] - 365s 712ms/step - loss: 0.2345 - acc: 0.8997
Epoch 18/50
512/512 [==============================] - 1522s 3s/step - loss: 0.2143 - acc: 0.8978
Epoch 19/50
512/512 [==============================] - 393s 769ms/step - loss: 0.2346 - acc: 0.8968
Epoch 20/50
512/512 [==============================] - 385s 752ms/step - loss: 0.2130 - acc: 0.9045
Epoch 21/50
512/512 [==============================] - 373s 729ms/step - loss: 0.2215 - acc: 0.8953
Epoch 22/50
512/512 [==============================] - 373s 729ms/step - loss: 0.2227 - acc: 0.9025
Epoch 23/50
512/512 [==============================] - 373s 728ms/step - loss: 0.2028 - acc: 0.9009
```

```
Epoch 35/50
512/512 [==============================] - 373s 728ms/step - loss: 0.1815 - acc: 0.9148
Epoch 36/50
512/512 [==============================] - 373s 728ms/step - loss: 0.1771 - acc: 0.9044
Epoch 37/50
512/512 [==============================] - 373s 728ms/step - loss: 0.1930 - acc: 0.9094
Epoch 38/50
512/512 [==============================] - 373s 728ms/step - loss: 0.1690 - acc: 0.9135
Epoch 39/50
512/512 [==============================] - 373s 729ms/step - loss: 0.1851 - acc: 0.9075
Epoch 40/50
512/512 [==============================] - 373s 728ms/step - loss: 0.1770 - acc: 0.9151
Epoch 41/50
512/512 [==============================] - 373s 728ms/step - loss: 0.1698 - acc: 0.9061
Epoch 42/50
512/512 [==============================] - 373s 728ms/step - loss: 0.1883 - acc: 0.9101
Epoch 43/50
512/512 [==============================] - 373s 728ms/step - loss: 0.1664 - acc: 0.9144
Epoch 44/50
512/512 [==============================] - 373s 729ms/step - loss: 0.1803 - acc: 0.9076
Epoch 45/50
512/512 [==============================] - 373s 729ms/step - loss: 0.1751 - acc: 0.9150
Epoch 46/50
512/512 [==============================] - 373s 729ms/step - loss: 0.1598 - acc: 0.9093
Epoch 47/50
512/512 [==============================] - 373s 728ms/step - loss: 0.1812 - acc: 0.9119
Epoch 48/50
512/512 [==============================] - 373s 728ms/step - loss: 0.1579 - acc: 0.9176
Epoch 49/50
512/512 [==============================] - 373s 728ms/step - loss: 0.1683 - acc: 0.9095
Epoch 50/50
512/512 [==============================] - 373s 728ms/step - loss: 0.1681 - acc: 0.9163
```

The highest possible accuracy was seen to be 91 %. And was the best result for our model.

Following curves shows the variation of the Accuracy and Loss function with number of epoch





Segmentation results were as follows

For test Image 159

For test Image 119

For test Image 959

2. Test Case 2:

For the second test case we selected SGD as our optimizer and a learning rate of 0.1 was set. We trained the model with an epoch of 50 and the loss and accuracy after each epoch was as follows

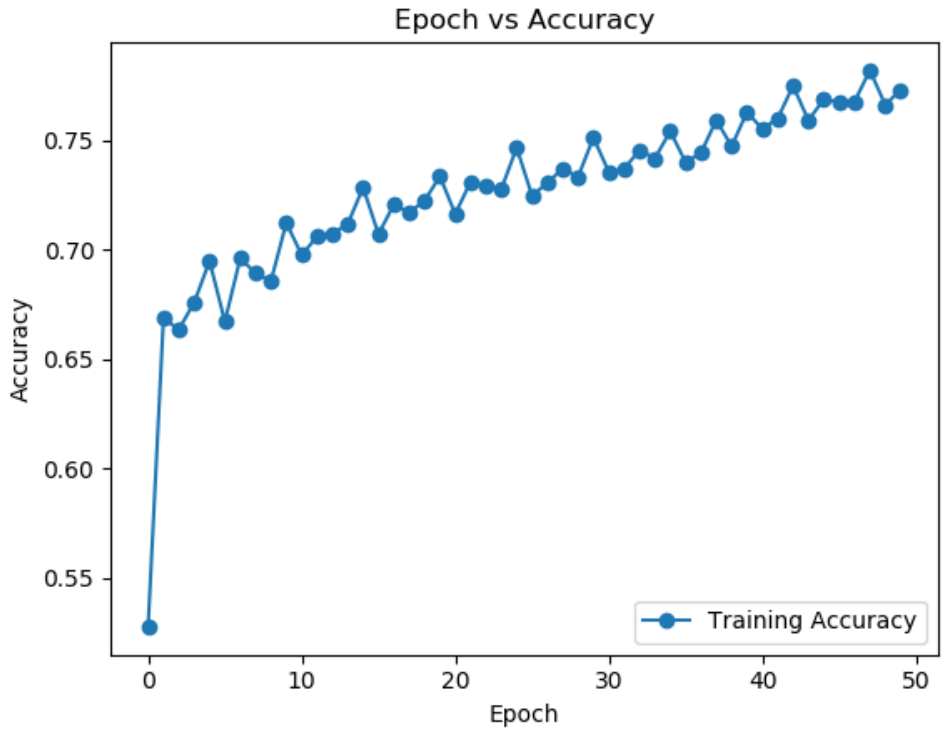
```
Epoch 1/50
512/512 [==============================] - 368s 718ms/step - loss: 2.8212 - acc: 0.5277
Epoch 2/50
512/512 [==============================] - 366s 716ms/step - loss: 1.0074 - acc: 0.6686
Epoch 3/50
512/512 [==============================] - 366s 715ms/step - loss: 0.9783 - acc: 0.6636
Epoch 4/50
512/512 [==============================] - 366s 716ms/step - loss: 0.9684 - acc: 0.6758
Epoch 5/50
512/512 [==============================] - 367s 716ms/step - loss: 0.9145 - acc: 0.6948
Epoch 6/50
512/512 [==============================] - 367s 716ms/step - loss: 0.9816 - acc: 0.6672
Epoch 7/50
512/512 [==============================] - 367s 716ms/step - loss: 0.9065 - acc: 0.6961
Epoch 8/50
512/512 [==============================] - 366s 716ms/step - loss: 0.9052 - acc: 0.6892
Epoch 9/50
512/512 [==============================] - 366s 716ms/step - loss: 0.9274 - acc: 0.6856
Epoch 10/50
512/512 [==============================] - 367s 716ms/step - loss: 0.8384 - acc: 0.7123
Epoch 11/50
512/512 [==============================] - 367s 716ms/step - loss: 0.8877 - acc: 0.6975
Epoch 12/50
512/512 [==============================] - 367s 717ms/step - loss: 0.8648 - acc: 0.7060
Epoch 13/50
512/512 [==============================] - 367s 716ms/step - loss: 0.8478 - acc: 0.7073
Epoch 14/50
512/512 [==============================] - 367s 717ms/step - loss: 0.8622 - acc: 0.7117
Epoch 15/50
512/512 [==============================] - 367s 717ms/step - loss: 0.8056 - acc: 0.7285
Epoch 16/50
512/512 [==============================] - 367s 716ms/step - loss: 0.8496 - acc: 0.7071
Epoch 17/50
512/512 [==============================] - 366s 716ms/step - loss: 0.8211 - acc: 0.7206
Epoch 18/50
512/512 [==============================] - 366s 715ms/step - loss: 0.8070 - acc: 0.7170
Epoch 19/50
512/512 [==============================] - 366s 715ms/step - loss: 0.8072 - acc: 0.7224
Epoch 20/50
512/512 [==============================] - 366s 715ms/step - loss: 0.7766 - acc: 0.7336
```

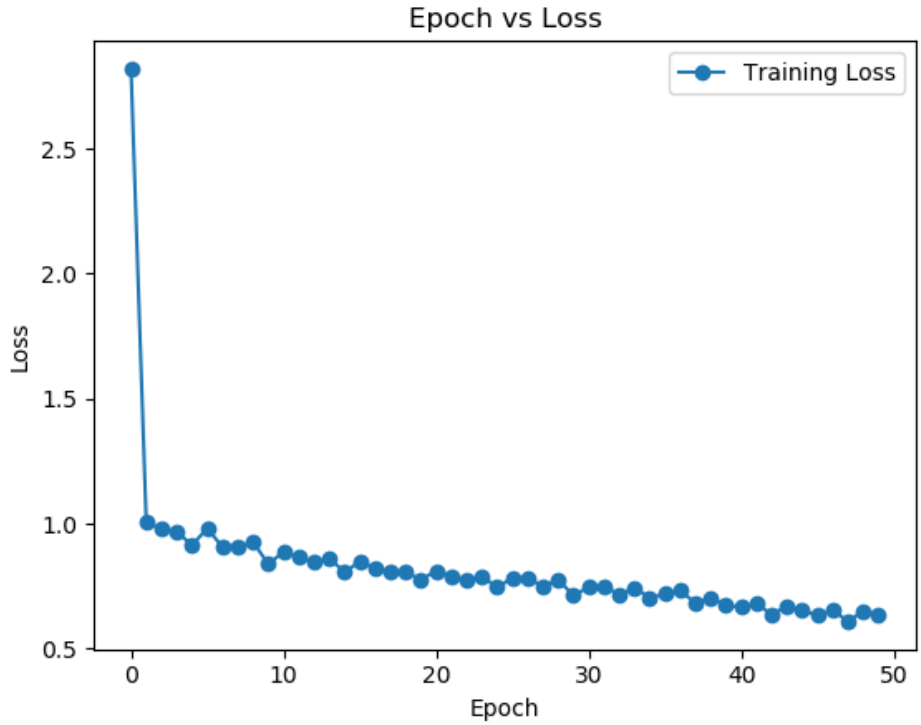
```
Epoch 19/50
512/512 [==============================] - 366s 715ms/step - loss: 0.8072 - acc: 0.7224
Epoch 20/50
512/512 [==============================] - 366s 715ms/step - loss: 0.7766 - acc: 0.7336
Epoch 21/50
512/512 [==============================] - 366s 715ms/step - loss: 0.8114 - acc: 0.7164
Epoch 22/50
512/512 [==============================] - 366s 716ms/step - loss: 0.7896 - acc: 0.7310
Epoch 23/50
512/512 [==============================] - 366s 715ms/step - loss: 0.7763 - acc: 0.7294
Epoch 24/50
512/512 [==============================] - 366s 716ms/step - loss: 0.7895 - acc: 0.7276
Epoch 25/50
512/512 [==============================] - 381s 743ms/step - loss: 0.7456 - acc: 0.7467
Epoch 26/50
512/512 [==============================] - 390s 761ms/step - loss: 0.7830 - acc: 0.7247
Epoch 27/50
512/512 [==============================] - 389s 759ms/step - loss: 0.7816 - acc: 0.7306
Epoch 28/50
512/512 [==============================] - 366s 715ms/step - loss: 0.7509 - acc: 0.7371
Epoch 29/50
512/512 [==============================] - 370s 722ms/step - loss: 0.7772 - acc: 0.7333
Epoch 30/50
512/512 [==============================] - 369s 720ms/step - loss: 0.7130 - acc: 0.7512
Epoch 31/50
512/512 [==============================] - 366s 716ms/step - loss: 0.7462 - acc: 0.7355
Epoch 32/50
512/512 [==============================] - 374s 731ms/step - loss: 0.7517 - acc: 0.7370
Epoch 33/50
512/512 [==============================] - 389s 760ms/step - loss: 0.7162 - acc: 0.7456
Epoch 34/50
512/512 [==============================] - 389s 760ms/step - loss: 0.7437 - acc: 0.7417
Epoch 35/50
512/512 [==============================] - 389s 759ms/step - loss: 0.7041 - acc: 0.7542
Epoch 36/50
512/512 [==============================] - 389s 759ms/step - loss: 0.7188 - acc: 0.7400
Epoch 37/50
512/512 [==============================] - 388s 758ms/step - loss: 0.7324 - acc: 0.7442
Epoch 38/50
512/512 [==============================] - 388s 759ms/step - loss: 0.6832 - acc: 0.7593
Epoch 39/50
512/512 [==============================] - 389s 759ms/step - loss: 0.7036 - acc: 0.7472
```

```
Epoch 33/50
512/512 [==============================] - 389s 760ms/step - loss: 0.7162 - acc: 0.7456
Epoch 34/50
512/512 [==============================] - 389s 760ms/step - loss: 0.7437 - acc: 0.7417
Epoch 35/50
512/512 [==============================] - 389s 759ms/step - loss: 0.7041 - acc: 0.7542
Epoch 36/50
512/512 [==============================] - 389s 759ms/step - loss: 0.7188 - acc: 0.7400
Epoch 37/50
512/512 [==============================] - 388s 758ms/step - loss: 0.7324 - acc: 0.7442
Epoch 38/50
512/512 [==============================] - 388s 759ms/step - loss: 0.6832 - acc: 0.7593
Epoch 39/50
512/512 [==============================] - 389s 759ms/step - loss: 0.7036 - acc: 0.7472
Epoch 40/50
512/512 [==============================] - 389s 759ms/step - loss: 0.6747 - acc: 0.7630
Epoch 41/50
512/512 [==============================] - 388s 758ms/step - loss: 0.6657 - acc: 0.7551
Epoch 42/50
512/512 [==============================] - 388s 758ms/step - loss: 0.6818 - acc: 0.7600
Epoch 43/50
512/512 [==============================] - 389s 759ms/step - loss: 0.6364 - acc: 0.7749
Epoch 44/50
512/512 [==============================] - 389s 759ms/step - loss: 0.6710 - acc: 0.7591
Epoch 45/50
512/512 [==============================] - 497s 970ms/step - loss: 0.6530 - acc: 0.7687
Epoch 46/50
512/512 [==============================] - 1087s 2s/step - loss: 0.6342 - acc: 0.7675
Epoch 47/50
512/512 [==============================] - 367s 717ms/step - loss: 0.6543 - acc: 0.7676
Epoch 48/50
512/512 [==============================] - 366s 714ms/step - loss: 0.6093 - acc: 0.7820
Epoch 49/50
512/512 [==============================] - 366s 716ms/step - loss: 0.6505 - acc: 0.7662
Epoch 50/50
512/512 [==============================] - 366s 716ms/step - loss: 0.6370 - acc: 0.7730
```

The highest possible accuracy was seen to be 77 %

Following curves shows the variation of the Accuracy and Loss function with number of epoch





Segmented Result for Test Image

For test Image 159

For test Image 119

For test Image 959

3. Test Case 3:

For the first test case we selected AdaDelta as our optimizer and a learning rate of 0.1 was set. We trained the model with an epoch of 40 and the loss and accuracy after each epoch was as follows

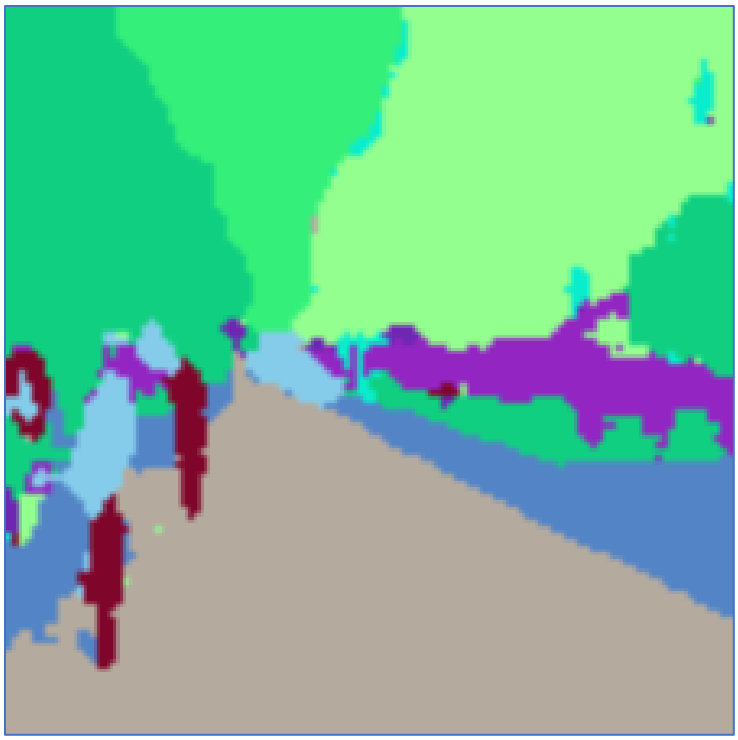
```
Epoch 1/1
512/512 [==============================] - 526s 1s/step - loss: 0.9793 - acc: 0.6501
Epoch 1/1
512/512 [==============================] - 527s 1s/step - loss: 0.8513 - acc: 0.6963
Epoch 1/1
512/512 [==============================] - 527s 1s/step - loss: 0.8008 - acc: 0.7150
Epoch 1/1
512/512 [==============================] - 527s 1s/step - loss: 0.7724 - acc: 0.7254
Epoch 1/1
512/512 [==============================] - 526s 1s/step - loss: 0.7502 - acc: 0.7302
Epoch 1/1
512/512 [==============================] - 527s 1s/step - loss: 0.7456 - acc: 0.7359
Epoch 1/1
512/512 [==============================] - 527s 1s/step - loss: 0.7295 - acc: 0.7421
Epoch 1/1
512/512 [==============================] - 526s 1s/step - loss: 0.7300 - acc: 0.7407
Epoch 1/1
512/512 [==============================] - 527s 1s/step - loss: 0.7036 - acc: 0.7490
Epoch 1/1
512/512 [==============================] - 527s 1s/step - loss: 0.6721 - acc: 0.7614
Epoch 1/1
512/512 [==============================] - 527s 1s/step - loss: 0.6455 - acc: 0.7685
Epoch 1/1
512/512 [==============================] - 526s 1s/step - loss: 0.6276 - acc: 0.7766
Epoch 1/1
512/512 [==============================] - 527s 1s/step - loss: 0.6068 - acc: 0.7861
Epoch 1/1
512/512 [==============================] - 527s 1s/step - loss: 0.5987 - acc: 0.7868
Epoch 1/1
512/512 [==============================] - 526s 1s/step - loss: 0.5802 - acc: 0.7887
Epoch 1/1
512/512 [==============================] - 527s 1s/step - loss: 0.5436 - acc: 0.8024
Epoch 1/1
512/512 [==============================] - 527s 1s/step - loss: 0.5250 - acc: 0.8085
Epoch 1/1
512/512 [==============================] - 527s 1s/step - loss: 0.5058 - acc: 0.8137
Epoch 1/1
512/512 [==============================] - 526s 1s/step - loss: 0.4794 - acc: 0.8263
Epoch 1/1
512/512 [==============================] - 527s 1s/step - loss: 0.4761 - acc: 0.8277
Epoch 1/1
512/512 [==============================] - 527s 1s/step - loss: 0.4626 - acc: 0.8295
Epoch 1/1
512/512 [==============================] - 526s 1s/step - loss: 0.4377 - acc: 0.8364
Epoch 1/1
512/512 [==============================] - 527s 1s/step - loss: 0.4147 - acc: 0.8442
```

```
512/512 [==============================] - 526s 1s/step - loss: 0.4377 - acc: 0.8364
Epoch 1/1
512/512 [==============================] - 527s 1s/step - loss: 0.4147 - acc: 0.8442
Epoch 1/1
512/512 [==============================] - 527s 1s/step - loss: 0.4005 - acc: 0.8482
Epoch 1/1
512/512 [==============================] - 526s 1s/step - loss: 0.3832 - acc: 0.8558
Epoch 1/1
512/512 [==============================] - 526s 1s/step - loss: 0.3768 - acc: 0.8585
Epoch 1/1
512/512 [==============================] - 527s 1s/step - loss: 0.3699 - acc: 0.8588
Epoch 1/1
512/512 [==============================] - 527s 1s/step - loss: 0.3538 - acc: 0.8611
Epoch 1/1
512/512 [==============================] - 526s 1s/step - loss: 0.3387 - acc: 0.8666
Epoch 1/1
512/512 [==============================] - 527s 1s/step - loss: 0.3279 - acc: 0.8702
Epoch 1/1
512/512 [==============================] - 527s 1s/step - loss: 0.3173 - acc: 0.8734
Epoch 1/1
512/512 [==============================] - 526s 1s/step - loss: 0.3093 - acc: 0.8781
Epoch 1/1
512/512 [==============================] - 527s 1s/step - loss: 0.3105 - acc: 0.8772
Epoch 1/1
512/512 [==============================] - 527s 1s/step - loss: 0.3052 - acc: 0.8766
Epoch 1/1
512/512 [==============================] - 527s 1s/step - loss: 0.2920 - acc: 0.8793
Epoch 1/1
512/512 [==============================] - 526s 1s/step - loss: 0.2814 - acc: 0.8828
Epoch 1/1
512/512 [==============================] - 527s 1s/step - loss: 0.2743 - acc: 0.8842
Epoch 1/1
512/512 [==============================] - 527s 1s/step - loss: 0.2714 - acc: 0.8870
Epoch 1/1
512/512 [==============================] - 527s 1s/step - loss: 0.2669 - acc: 0.8891
Epoch 1/1
512/512 [==============================] - 527s 1s/step - loss: 0.2682 - acc: 0.8869
```

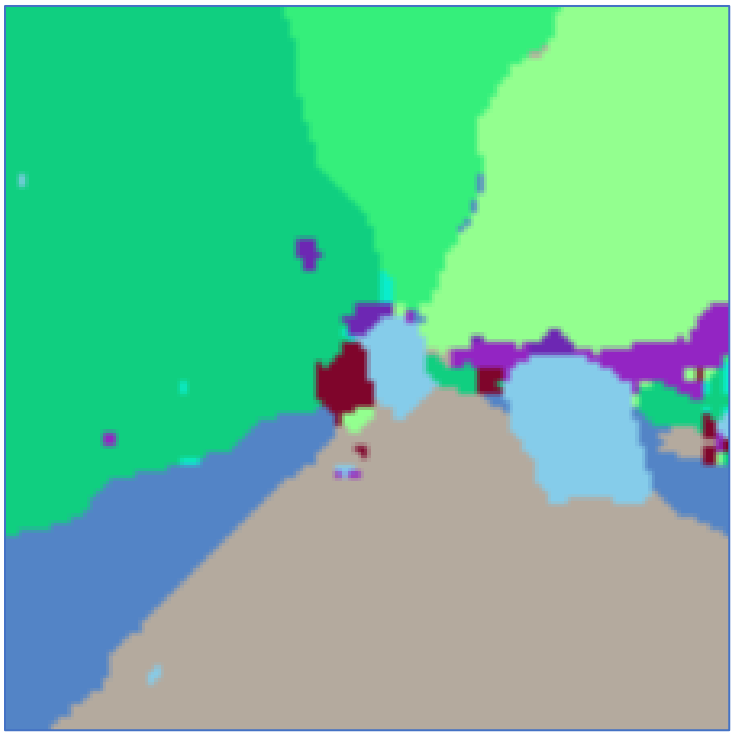
The highest possible accuracy was seen to be 88%

Segmentation results

For test Image 159



For test Image 119



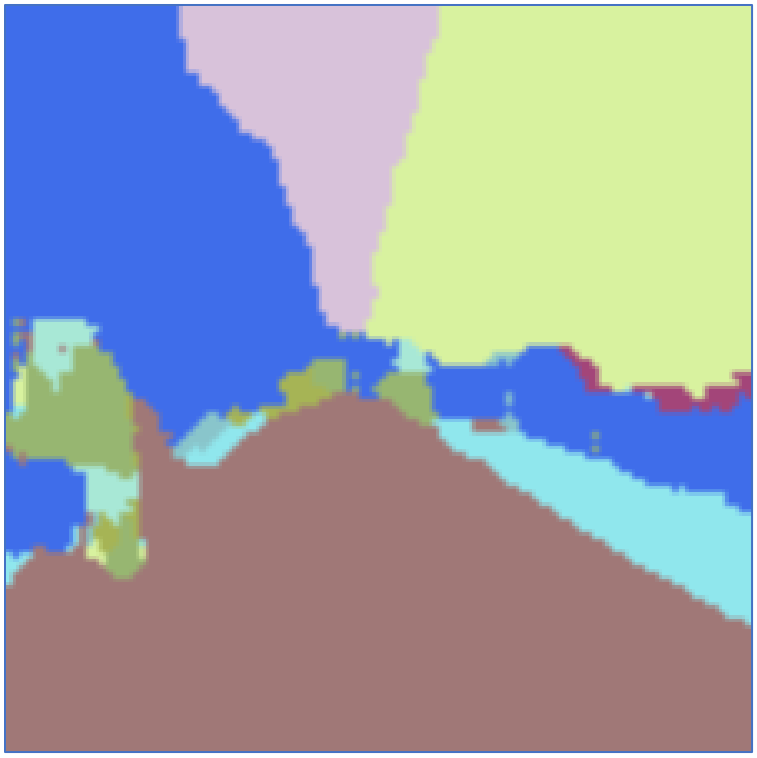
For test Image 959

4. Test Case 4:

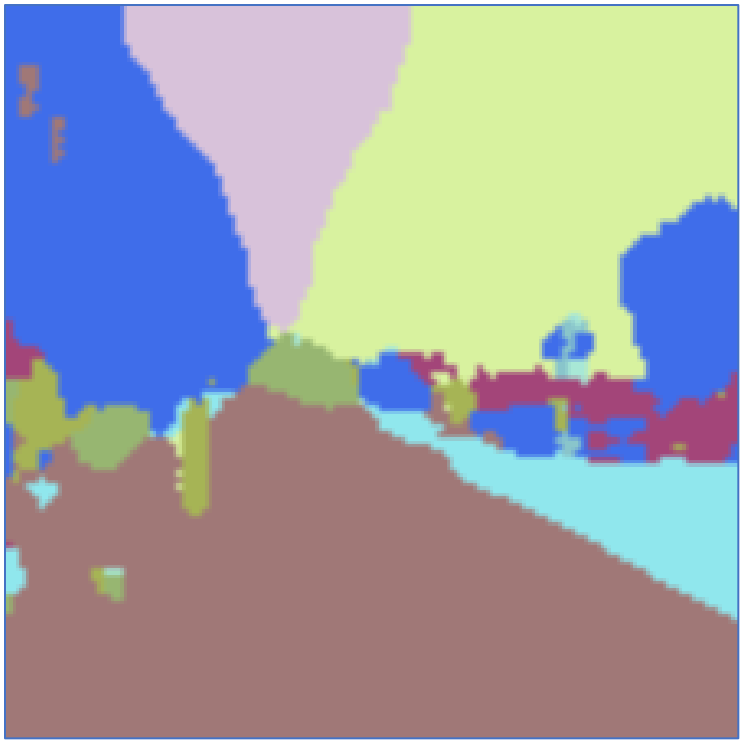
For the fourth test case we selected SGD as our optimizer and a learning rate of 0.1 was set. We trained the model with an epoch of 20 and the loss and accuracy after each epoch was as follows

```
Epoch 1/1
512/512 [==============================] - 366s 715ms/step - loss: 0.4897 - acc: 0.8201
Epoch 1/1
512/512 [==============================] - 369s 721ms/step - loss: 0.3541 - acc: 0.8642
Epoch 1/1
512/512 [==============================] - 370s 722ms/step - loss: 0.3256 - acc: 0.8632
Epoch 1/1
512/512 [==============================] - 372s 726ms/step - loss: 0.3174 - acc: 0.8748
Epoch 1/1
512/512 [==============================] - 372s 726ms/step - loss: 0.2788 - acc: 0.8799
Epoch 1/1
512/512 [==============================] - 371s 724ms/step - loss: 0.2860 - acc: 0.8820
Epoch 1/1
512/512 [==============================] - 370s 722ms/step - loss: 0.2575 - acc: 0.8920
Epoch 1/1
512/512 [==============================] - 369s 722ms/step - loss: 0.2676 - acc: 0.8828
Epoch 1/1
512/512 [==============================] - 370s 722ms/step - loss: 0.2562 - acc: 0.8934
Epoch 1/1
512/512 [==============================] - 370s 722ms/step - loss: 0.2381 - acc: 0.8884
Epoch 1/1
512/512 [==============================] - 370s 722ms/step - loss: 0.2543 - acc: 0.8931
Epoch 1/1
512/512 [==============================] - 370s 722ms/step - loss: 0.2218 - acc: 0.8983
Epoch 1/1
512/512 [==============================] - 369s 721ms/step - loss: 0.2424 - acc: 0.8934
Epoch 1/1
512/512 [==============================] - 370s 722ms/step - loss: 0.2181 - acc: 0.9034
Epoch 1/1
512/512 [==============================] - 369s 721ms/step - loss: 0.2284 - acc: 0.8936
Epoch 1/1
512/512 [==============================] - 369s 722ms/step - loss: 0.2231 - acc: 0.9026
Epoch 1/1
512/512 [==============================] - 368s 718ms/step - loss: 0.2043 - acc: 0.8990
Epoch 1/1
512/512 [==============================] - 369s 720ms/step - loss: 0.2261 - acc: 0.9009
Epoch 1/1
512/512 [==============================] - 369s 721ms/step - loss: 0.1985 - acc: 0.9066
Epoch 1/1
512/512 [==============================] - 370s 722ms/step - loss: 0.2194 - acc: 0.8986
```

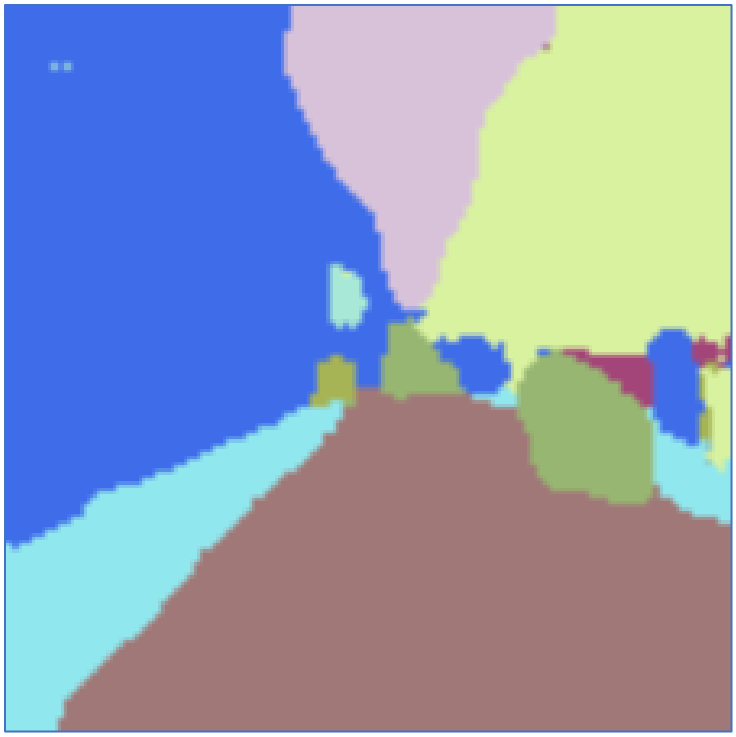
The highest accuracy was seen to be 89%



For test Image 159



For test Image 119



For test Image 959

VI. References

1. SegNet: A Deep Convolutional Encoder-Decoder Architecture for Image Segmentation Vijay Badrinarayanan, Alex Kendall, Roberto Cipolla

   https://arxiv.org/abs/1511.00561

2. Very Deep Convolutional Networks for Large-Scale Image Recognition Karen Simonyan, Andrew Zisserman

   https://arxiv.org/pdf/1409.1556.pdf

3. "Qure.ai Blog." *A 2017 Guide to Semantic Segmentation with Deep Learning*, blog.qure.ai/notes/semantic-segmentation-deep-learning-review.

4. 10152328293982999. "Learning Keras by Implementing the VGG Network From Scratch." *Hacker Noon*, Hacker Noon, 27 Mar. 2017, hackernoon.com/learning-keras-by-implementing-vgg16-from-scratch-d036733f2d5.
5. http://www.cs.toronto.edu/~urtasun/courses/CSC2541/07_segmentation.pdf

VII. Contribution

Research Work: Adityta, Neeraj, Sreeraj

Presentation: Neeraj, Sreeraj

Codes:

Training script : Neeraj

Predicting script: Sreeraj

Model Architecture: Aditya, Neeraj, Sreeraj

Processing script : Aditya

Although, the task were divided each of us also worked and helped other team members with their individual tasks. Training and prediction were executed by everyone on different laptops.

VIII. Link to Dataset

https://drive.google.com/drive/folders/1Uk1wqV6nXDKcd_o5TS99XV4VDX85EPEj