HW 2 Solutions

Author:
Fateme Noorzad

StudentID:
810198271

Deep Learning and Applications
Spring 2021

Contents

1 Introduction **1**

2 Question 1 **2**
2.1 Part A: Review on Papers 2
2.1.1 A Review on SegNet: A Deep Convolutional Encoder-Decoder Architecture for Image Segmentation 2
Architecture 3
2.1.2 SegNet: A Deep Convolutional Encoder-Decoder Architecture for Robust Semantic Pixel-Wise Labelling 4
2.2 Part B: CamVid Dataset 4
2.2.1 Class Names 4
2.2.2 CamVid Images 5
2.2.3 Segmented Images 5
2.2.4 Batch of Data 5
2.3 Part C: Implementation of SegNet-base 6
2.4 Part D: With Batch Normalization 12
2.5 Part F: Training with All Data 17

3 Question 2 **19**
3.1 Preprocessing Data 19
3.1.1 Train and Test Images 19
3.1.2 Train and Test Labels 20
3.1.3 Data Loading and Batches 20
3.2 Training the Network 20

4 Appendix: How to Run the Codes **24**
4.1 Problem 1 24
4.2 Problem 2 24

5 References **25**

Chapter 1

Introduction

In this homework, in the first problem, SegNet-base is implemented to find the segmentation of images. In the second problem, CCNN is implemented so that the number of individuals in a picture can be counted.

Chapter 2

Question 1

In this problem, given the CamVid dataset, we are to segment the given images based on the architecture proposed in a paper under the name of "SegNet: A Deep Convolutional Encoder-Decpder Architecture for Robust Semantic Pixel-Wise Labelling".

In the following sections, the questions regarding the stated goal and dataset are elaborated.

2.1 Part A: Review on Papers

2.1.1 A Review on SegNet: A Deep Convolutional Encoder-Decoder Architecture for Image Segmentation

Semantic segmentation has a wide array of applications ranging from scene understanding, inferring support relationships among objects to autonomous driving. Early methods which employed low-level vision cues have soon been replaced by popular machine learning algorithms, specially deep learning. This paper's motivation is to design SegNet in order to map low resolution features to input resolution for pixel-wise classification. In fact, this mapping must produce features which are useful for accurate boundary localization.

This architecture is designed to be an efficient architecture for pixel-wise semantic segmentation. The motivation in designing SegNet came from road scene understanding applications which require the ability to model the appearance and shape as well as understanding the spatial relationship between different classes such as road and side-walk. In other words, in typical road scenes, the majority of the pixels belong to large classes such as roads and buildings. Hence, the network must produce smooth segmentations for these cases. In addition, the engine must have the ability to delineate objects based on their shape despite their small size. Therefore, it is vital to retain boundary information in the extracted image representation. If we look at this matter through the lens of computations, it is necessary for the network to be efficient in terms of both memory and computation time during inference.

As another benefit of SegNet which its efficient design made it available is the ability of training end-to-end in order to jointly optimize all the weights in the network using an efficient weight update technique such as SGD which is easily repeatable.

The encoder network in SegNet is topologically identical to the convolutional layers in VGG16. However, the fully connected layers of VGG16 are removed in SegNet, making its encoder network significantly smaller and easier to train.

All the above features aside, the key component of SegNet is the decoder network

which consists of a hierarchy of decoders each corresponding to one encoder. In addition, the decoders employ the max-pooling indices received form the corresponding encoder to perform non-linear upsampling of their feature maps. [1]

Architecture

As has been discussed earlier, SegNet consists of an encoder networks with its corresponding decoder network, as well as a final pixelwise classification layer.

- **Encoder**

  The encoder network has 13 convolutional layers which correspond to the first 13 convolutional layers in VGG16 network, designed for classification of objects. Hence, the training process from weights trained for classification on large datasets can be used for initialization. In addition, in favor of higher resolution feature maps at the deepest encoder output, the fully connected layers are discarded. Besides, it is essential to note that this fact also reduces the number of parameters in the SegNet encoder significantly.

  Each encoder in the encoder network performs convolution with a filter bank to produce a set of feature maps. These are then followed by batch normalization layers. Afterwards, an element-wise ReLU is applied. Following that, max-pooling with a 2×2 window with stride 2 (non-overlapping window) is performed and the output is sub-sampled by a factor of 2. Max-pooling is used to 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.

  While it seems logical that several layers of max-pooling and sub-sampling can achieve more translation invariance for robust classification, there is always a loss of spatial resolution of the feature map. This downside is not desirable since boundary delineation is crucial in segmentation. Therefore, to overcome this matter, capturing and storing boundary information in the encoder feature maps before dub-sampling is carried out. Although in the ideal case all the encoder feature map is stored for future usage, in practical applications due to the lack of memory storage, another way is proposed. In such cases, only the max-pooling indices [2] is memorized for each encoder feature map. For instance, in case of 2×2 pooling window, 2 bits of information is stored for future usages. Although this results in slight decrease in accuracy, the efficient memory usage makes it a suitable solution for practical cases.

- **Decoder**

  As has been mentioned various times, each encoder layer has a corresponding decoder layer and hence the decoder network has 13 layers. The last output of the decoder is then fed to a multi-class soft-max classifier to produce class probabilities for each pixel independently.

  The approriate decoder in the decoder network upsamples its input feature maps using the memorizes max-pooling indices from the corresponding encoder feature maps, creating sparse feature maps. Afterwards, to produce dense feature maps, a trainable decoder filter bank is convolved with the sparse feature maps. On the grounds of benefits of employing batch normalization layer, one of each is applied to each of these maps.

  One essential point to mention here is that the decoder corresponding to

[1] This idea was inspired form an architecture designed for unsupervised feature learning.

[2] The locations of the maximum feature value in each pooling window

the first encoder produces a multi-channel feature map, unlike other decoders in the network which produce feature maps with the same size and channel as their encoder. This decoder's high dimensional feature map is fed to a trainable soft-max classifier, classifying each pixel independently. Therefore, it is crystal clear that the output of the soft-max classifier is an image with number of channels as the same size as the number of classes.

2.1.2 SegNet: A Deep Convolutional Encoder-Decoder Architecture for Robust Semantic Pixel-Wise Labelling

The architecture of this network is given as Table 2.1:

Layer Number	Layer Name	Number of In Channels	Number of Out Channels	Kernel Size	Stride Size	Padding Size
1	Encoder 1 (Conv2D)	3	64	7	1	3
		R	e	L	U	
	Max-Pooling 1	-	-	2	2	0
2	Encoder 2 (Conv2D)	64	64	7	1	3
		R	e	L	U	
	Max-Pooling 2	-	-	2	2	0
3	Encoder 3 (Conv2D)	64	64	7	1	3
		R	e	L	U	
	Max-Pooling 3	-	-	2	2	0
4	Encoder 4 (Conv2D)	64	64	7	1	3
		R	e	L	U	
	Max-Pooling 4	-	-	2	2	0
5	Max-Unpooling 4	-	-	2	2	0
	Decoder 4 (Conv2D)	64	64	7	1	3
6	Max-Unpooling 3	-	-	2	2	0
	Decoder 3 (Conv2D)	64	64	7	1	3
7	Max-Unpooling 2	-	-	2	2	0
	Decoder 2 (Conv2D)	64	64	7	1	3
8	Max-Unpooling 1	-	-	2	2	0
	Decoder 1 (Conv2D)	64	64	7	1	3
9	Soft-Max (Conv2D)	64	32	7	1	3

Table 2.1: The layers of SegBet-base

2.2 Part B: CamVid Dataset

2.2.1 Class Names

Among the given files, there is a file in .txt format with the name of the classes as well as the associated RGB code to each one of them. This information is read and stored in a dictionary to be used for transferring RBG into class number and vise versa.

2.2.2 CamVid Images

CamVid images' names and their file path are read and stored into a list via a function under "readNamesFromFolder" name. (This function also reads the segmented images and their path) As the second step, 80% of them are separated as train and the other 20% as test datasets. Afterwards, several transformations need to be carried out on them. First of all, images need to be represented in tensor format. Secondly, they are all normalized. To do so, each channel's mean and variance is normalized by "torchvision.transforms.Normalize". As the third step, the normalized tensor of images are cropped so that their size be ultered to 360×480 by "torchvision.transforms.CenterCrop". Finally, segmented labels need to be ready, so that batched of images and their corresponding images become ready.

2.2.3 Segmented Images

Same as CamVid images, their name an path are read and stored into a list via the mentioned function and separated into training and test datasets. Then, they are cropped into the same shape as CamVid images. However, another transformation is also needed for them. By employing the saved dictionary of the classes and their RGB value, instead of RBG value of each pixel, its class number is used. Therefore, we have a 360×480 while in each pixel a number between 0-31 is written.

2.2.4 Batch of Data

As the last step of preprocessing, the normalized, cropped tensor of images with labels in the new format are gathered into batches of size 4. This procedure is carried out for both training and test datasets, so that they are both ready as the inputs of SegNet.

A batch of the mentioned datasets are depicted below:

FIGURE 2.1: The sample batch of images from CamVid training dataset

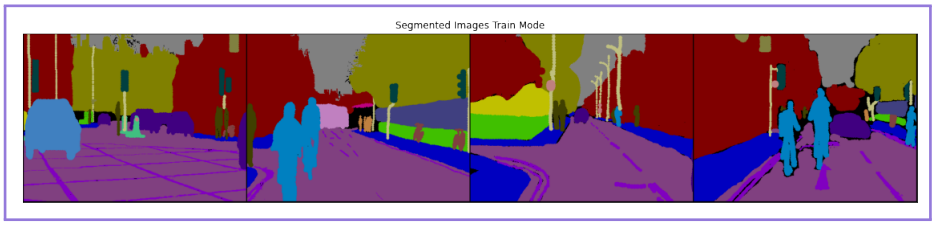

FIGURE 2.2: The sample batch of segmented images of CamVid from training dataset

FIGURE 2.3: The sample batch of images from CamVid test dataset

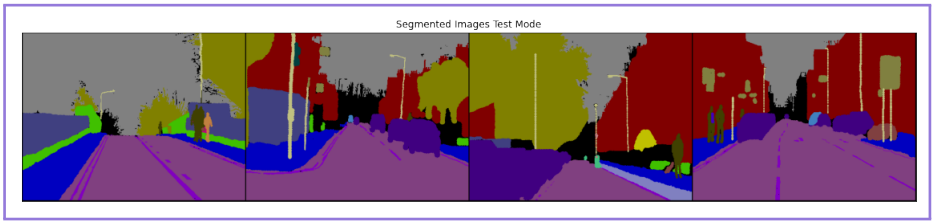

FIGURE 2.4: The sample batch of segmented images of CamVid from test dataset

2.3 Part C: Implementation of SegNet-base

In order to implement SegNet, two classes are implemented. One of which is "convBachNormReLU" with the goal of creating a unit based on the desired given conditions. The conditions are the number of input channels, output channels, kernel size, stride size, padding size, whether this unit has a batch normalization layer, bias condition, and whether the activation ReLU function is present or not. Based on these states, and by Sequence tool, a desired unit is created. With this unit in hand, encoder and decoders of SegNet can be employed easily.

As was discussed in Part A, encoder, decoder, max-pooling, max-unpooling, and softmax layers need to be implemented. For the first encoder, since the number of in channel is the same of the number of channels in the input channel, it is set as 3. However, for all the other encoders, the in and out channels are set to 64 [3]. The rest of the information regarding the mentioned layers are set based on the information in Table2.1.

This part is carried out with two different optimizers. The first one is the SGD optimizer with learning rate=0.001, momentum=0.9, and weight decay=0.005. With these parameters as well as "Cross Entropy Loss", without batch normalization, and in 30 epochs the results in Table2.2 can be seen:

	Epoch 1 Loss	**Epoch 10 Loss**	**Epoch 20 Loss**	**Epoch 30 Loss**
Iteration 10	3.04970	2.23234	2.33747	2.30516
Iteration 20	2.98063	2.24136	2.32445	2.27906
Iteration 30	2.90497	2.25548	2.29316	2.29133
Iteration 40	2.83691	2.26295	2.29259	2.28430

[3] This number is as the same as what has been stated in the paper and can be found in Table2.1

	Epoch 1 Loss	Epoch 10 Loss	Epoch 20 Loss	Epoch 30 Loss
Iteration 50	2.78454	2.27018	2.28698	2.28534
Iteration 60	2.73603	2.28616	2.27839	2.28482
Iteration 70	2.70228	2.28553	2.27591	2.28010
Iteration 80	2.66926	2.29044	2.27546	2.27970
Iteration 90	2.63779	2.28771	2.27906	2.27677
Iteration 100	2.60809	2.28938	2.28102	2.28059
Iteration 110	2.58734	2.29012	2.27889	2.28664
Iteration 120	2.56566	2.28787	2.28076	2.28621
Iteration 130	2.54635	2.28995	2.28497	2.28415
Iteration 140	2.52943	2.29413	2.28686	2.28363
Average of Train	2.52943	2.29413	2.28686	2.28363
Average of Test	2.38436	2.32106	2.30533	2.30713

TABLE 2.2: Loss of SegNet-base in some cases without batch normalization and with SGD optimizer

A better view of average loss of train and test through learning is depicted in Figure2.5 as below:

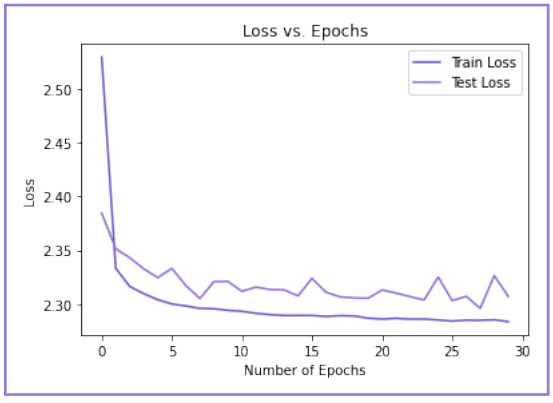

FIGURE 2.5: The loss function's behavior after 30 epochs

As this figure suggest, learning is not finished and more epochs are needed to reach a better result. This fact can also be seen in the below figures. In these figures the prediction of SegNet in comparision with the true segmented images can be seen.

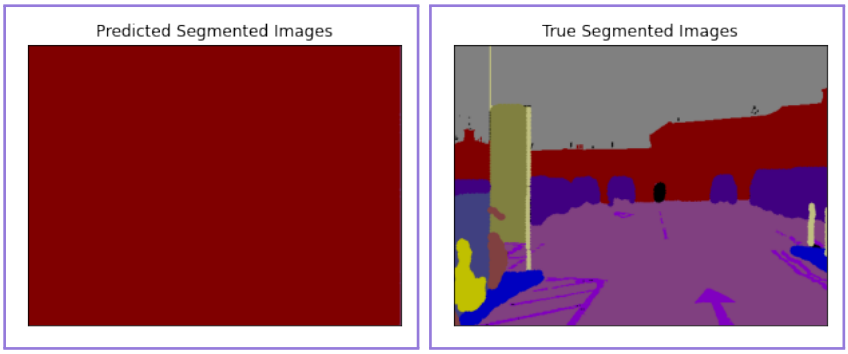

FIGURE 2.6: The true segmented image and the prediction of SegNet after the first epoch

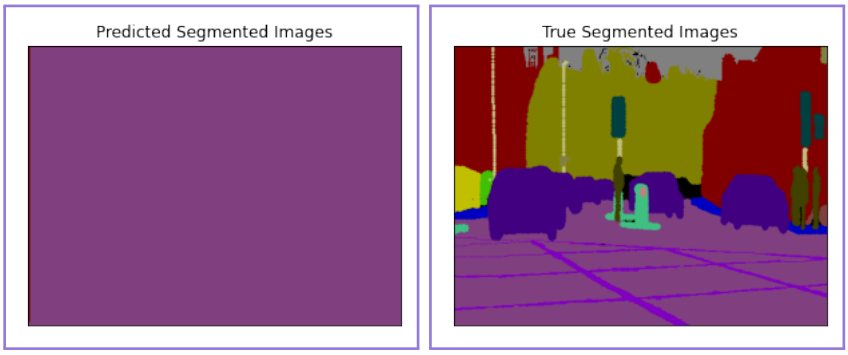

FIGURE 2.7: The true segmented image and the prediction of SegNet after 10 epochs

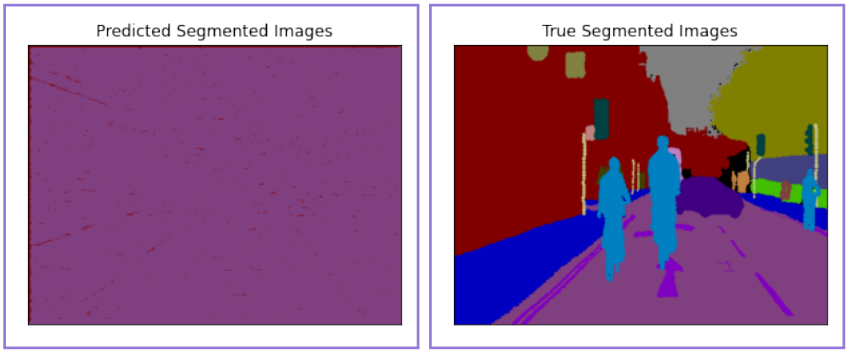

FIGURE 2.8: The true segmented image and the prediction of SegNet after 20 epochs

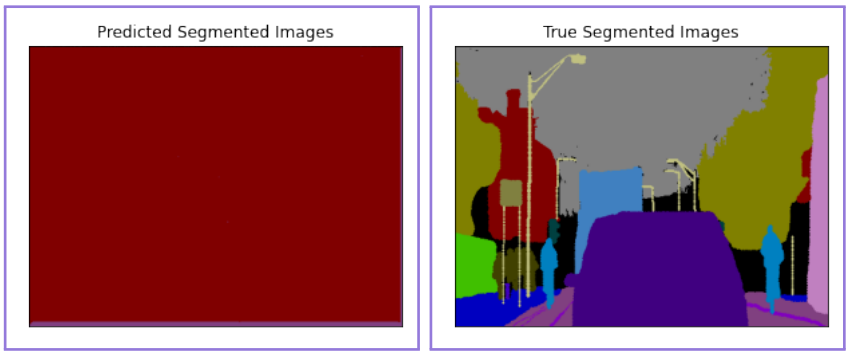

FIGURE 2.9: The true segmented image and the prediction of SegNet after 30 epochs

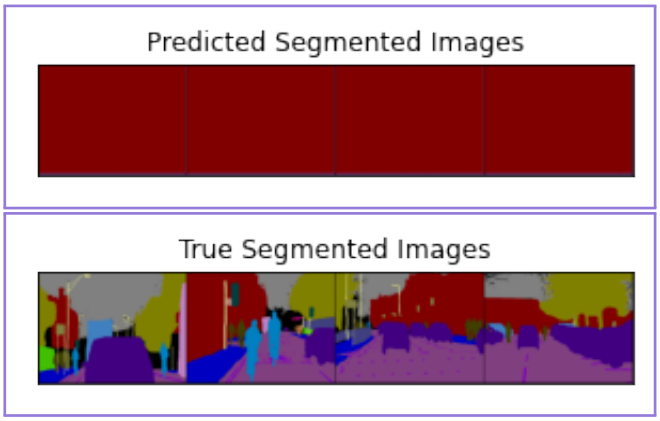

FIGURE 2.10: The true segmented images and the predictions of SegNet after 30 epochs

As all the above figures suggest, more epochs are needed. However, due to the fact that Google Colab's GPU is only available for short periods, more epochs could not be done. Therefore, this idea came to mind that since Adam optimizer reaches an answer faster (although it's accuracy is not as good as SGD in the end), this optimizer may be a better choice for limited GPU.
As a result, the same network with the same parameters but with Adam optimizer is trained. The results are represented in below:

	Epoch 1 Loss	**Epoch 10 Loss**	**Epoch 20 Loss**	**Epoch 30 Loss**
Iteration 10	2.79952	1.55222	1.25828	1.22876
Iteration 20	2.59289	1.49819	1.27321	1.24840
Iteration 30	2.51051	1.51485	1.27329	1.25681
Iteration 40	2.47979	1.55416	1.32244	1.25316
Iteration 50	2.45066	1.53420	1.32617	1.27149
Iteration 60	2.44415	1.50907	1.30707	1.26309

	Epoch 1 Loss	Epoch 10 Loss	Epoch 20 Loss	Epoch 30 Loss
Iteration 70	2.42683	1.49771	1.31311	1.24576
Iteration 80	2.39261	1.51257	1.29891	1.23486
Iteration 90	2.37590	1.50309	1.29616	1.23726
Iteration 100	2.35456	1.50971	1.29179	1.23267
Iteration 110	2.23505	1.51741	1.30481	1.22563
Iteration 120	2.33137	1.53058	1.31443	1.21850
Iteration 130	2.32009	1.52817	1.31259	1.21002
Iteration 140	2.29060	1.52369	1.31260	1.20026
Average of Train	2.29060	1.52369	1.31260	1.20026
Average of Test	1.85400	1.44223	1.22558	1.15225

TABLE 2.3: Loss of SegNet-base in some cases without batch normalization and with Adam optimizer

Comparing contents of Table2.3 with that of Table2.2 shows that in this number of epochs, Adam is a better optimizer than SGD. [4] In addition to the above table, the below figure gives a better insight regarding the decreament in loss of this implementation in 30 epochs:

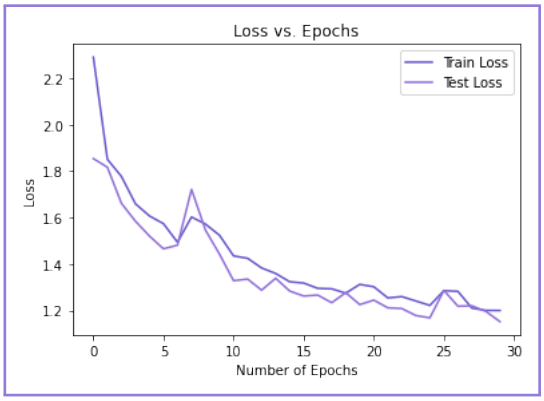

FIGURE 2.11: The loss function's behavior after 30 epochs

Besides, the evolution of prediction in term of segmented images ca also be found in the below figures:

[4] Note that in long term, im nost cases, SGD is a better optimizer

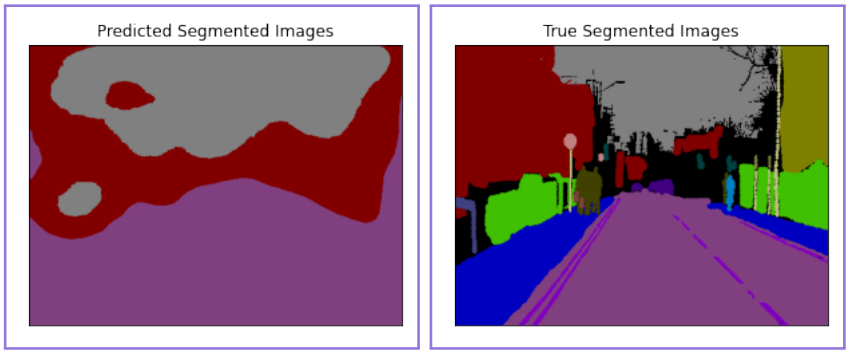

FIGURE 2.12: The true segmented image and the prediction of SegNet after the first epoch

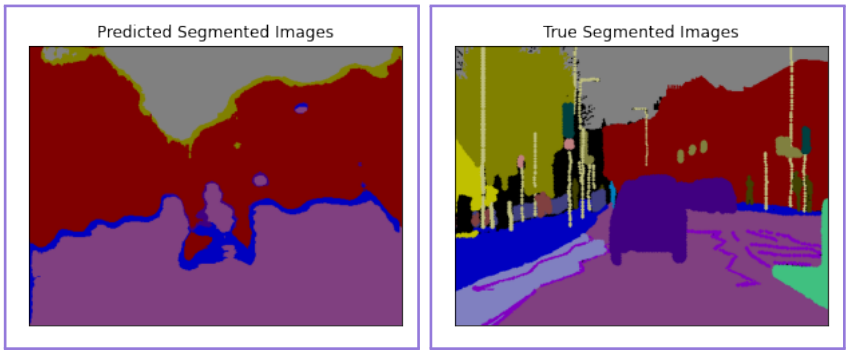

FIGURE 2.13: The true segmented image and the prediction of SegNet after 10 epochs

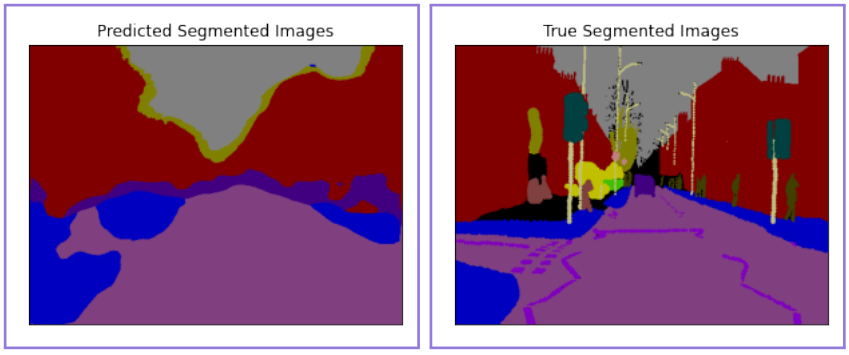

FIGURE 2.14: The true segmented image and the prediction of SegNet after 20 epochs

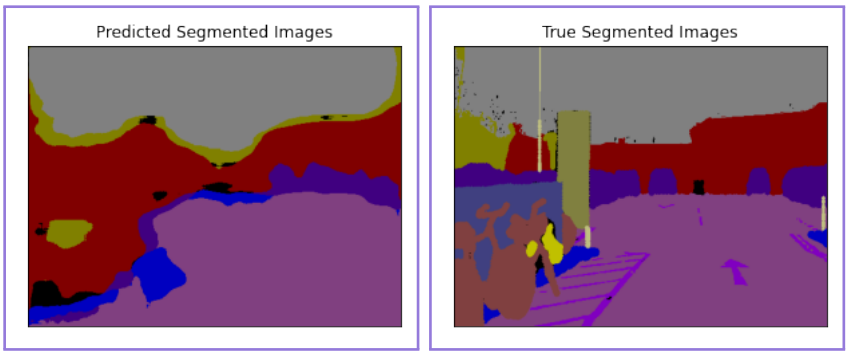

FIGURE 2.15: The true segmented image and the prediction of SegNet after 30 epochs

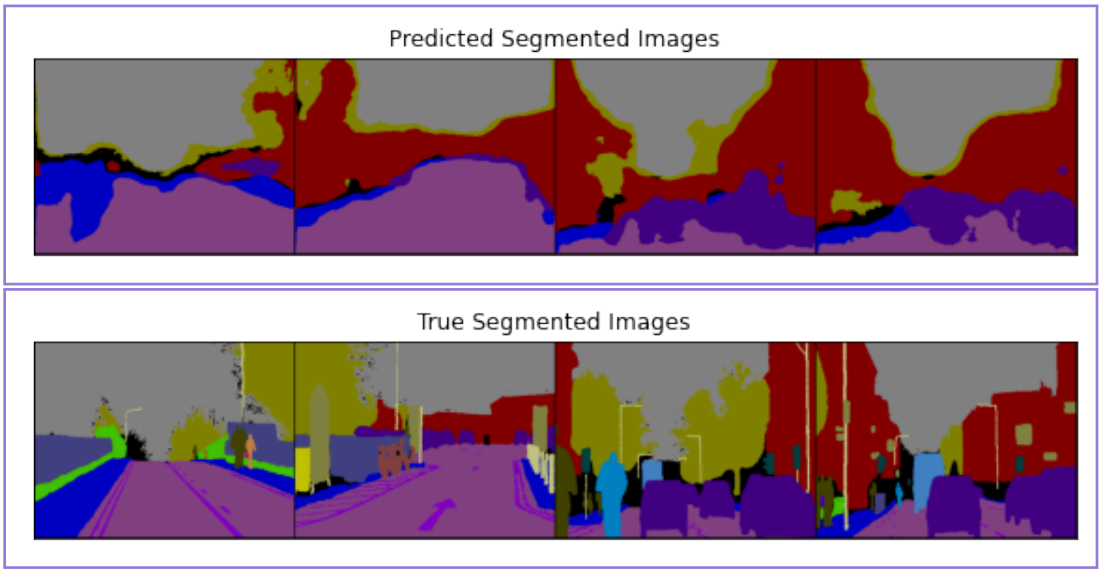

FIGURE 2.16: A batch of true segmented images and the predictions of SegNet after 30 epochs

As can be seen, the results are better than the SGD case, but they are not the best. In the next section, with Batch Normalization, significant improvements can be seen in the same number of epochs. In addition, it is worth mentioning that instability in Adam optimizer were seen more then SGD, this means that it is better to employ SGD while finding a way to make it faster!

2.4 Part D: With Batch Normalization

Same as the last section, SegNet is implemented, but with one difference. After each convolution layer, a batch normalization is set. With SGD as its optimizer and the same parameters of the last part, the below results were seen:

	Epoch 1 Loss	**Epoch 10 Loss**	**Epoch 20 Loss**	**Epoch 30 Loss**
Iteration 10	3.61472	0.79897	0.59692	0.63622
Iteration 20	3.57102	0.81136	0.65877	0.59609

	Epoch 1 Loss	Epoch 10 Loss	Epoch 20 Loss	Epoch 30 Loss
Iteration 30	3.40741	0.79962	0.69209	0.61359
Iteration 40	3.12213	0.81080	0.69898	0.62379
Iteration 50	2.89072	0.81410	0.71196	0.60773
Iteration 60	2.68721	0.83685	0.71811	0.60396
Iteration 70	2.57618	0.83199	0.71222	0.59428
Iteration 80	2.46250	0.84692	0.70780	0.58697
Iteration 90	2.36206	0.85286	0.70456	0.58386
Iteration 100	2.30606	0.85406	0.69762	0.58642
Iteration 110	2.23505	0.85495	0.70154	0.59269
Iteration 120	2.19505	0.86108	0.70263	0.59084
Iteration 130	2.14379	0.85837	0.69856	0.60469
Iteration 140	2.10022	0.85636	0.69878	0.60986
Average of Train	2.10022	0.85636	0.69878	0.60986
Average of Test	1.46723	0.85788	0.73653	0.78991

TABLE 2.4: Loss of SegNet-base in some cases with batch normalization and SGD optimizer

A comparision of Table2.4 with Table2.2 and Table2.3, shows in the same number of epochs, the last method which is a comparision of batch normalization and SGD optimizer resulted in a better performance. Training for 40 epochs results in Figure2.17 showing significant decrease in loss function.[5]

[5]Note that continuing learning will result in slightly better decreament, but due to the limitations of Google Clob, no more than 40 epochs could be run with GPU!

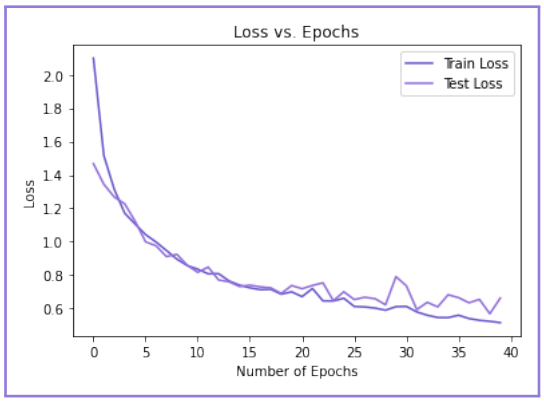

FIGURE 2.17: The loss function's behavior after 40 epochs

Now, we depict the predictions of this implementation and compare it with the true segmentation as below:

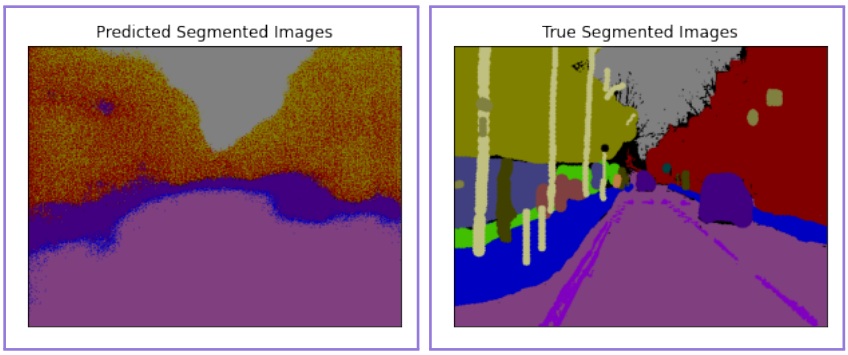

FIGURE 2.18: The true segmented image and the prediction of SegNet after the first epoch

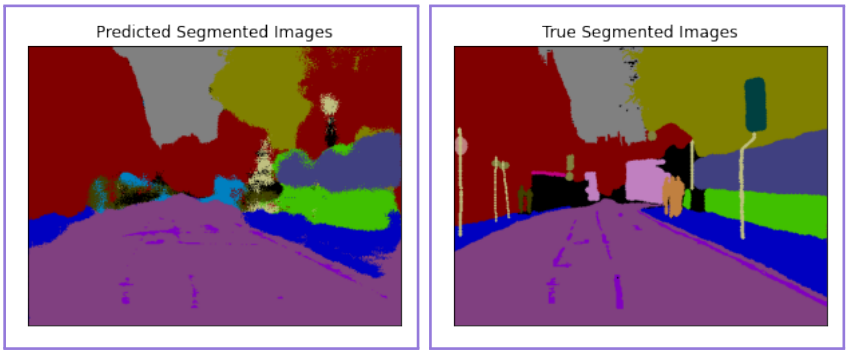

FIGURE 2.19: The true segmented image and the prediction of SegNet after 10 epochs

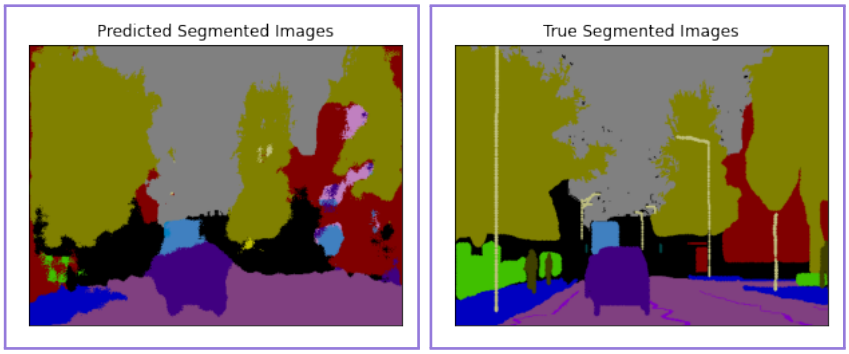

FIGURE 2.20: The true segmented image and the prediction of SegNet after 20 epochs

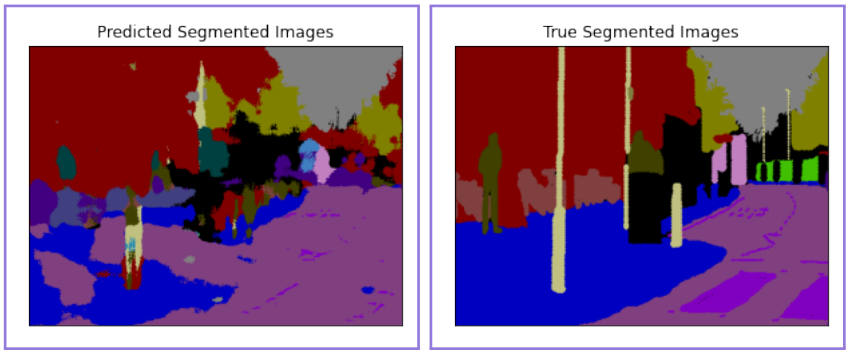

FIGURE 2.21: The true segmented image and the prediction of SegNet after 30 epochs

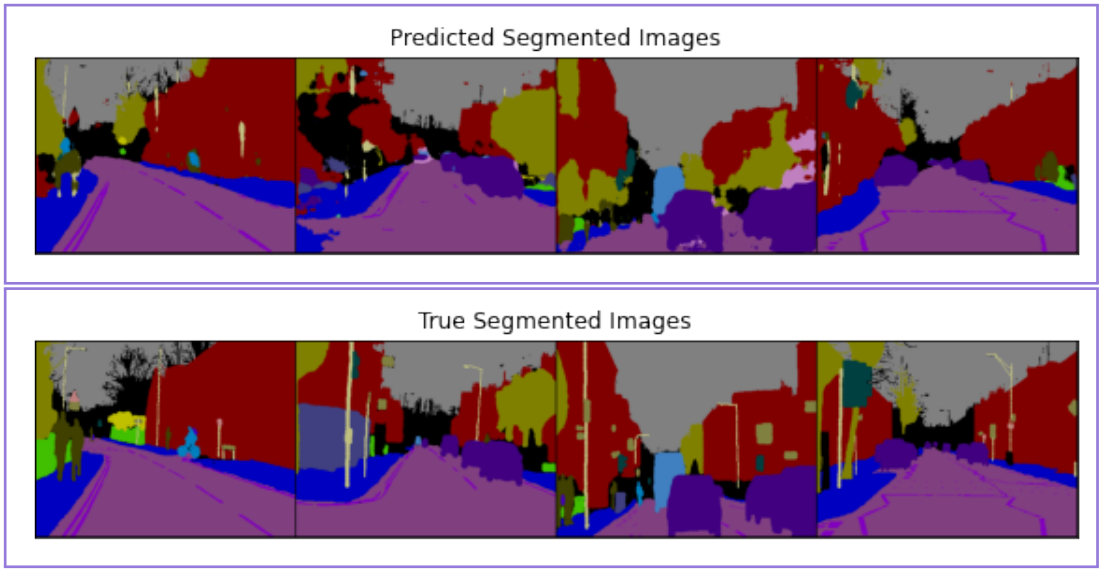

FIGURE 2.22: A batch of true segmented images and the predictions of SegNet after 40 epochs

In fact, looking at the results of the last 9 epochs will give great insight on how better this network is working in comparision with the last implementations:

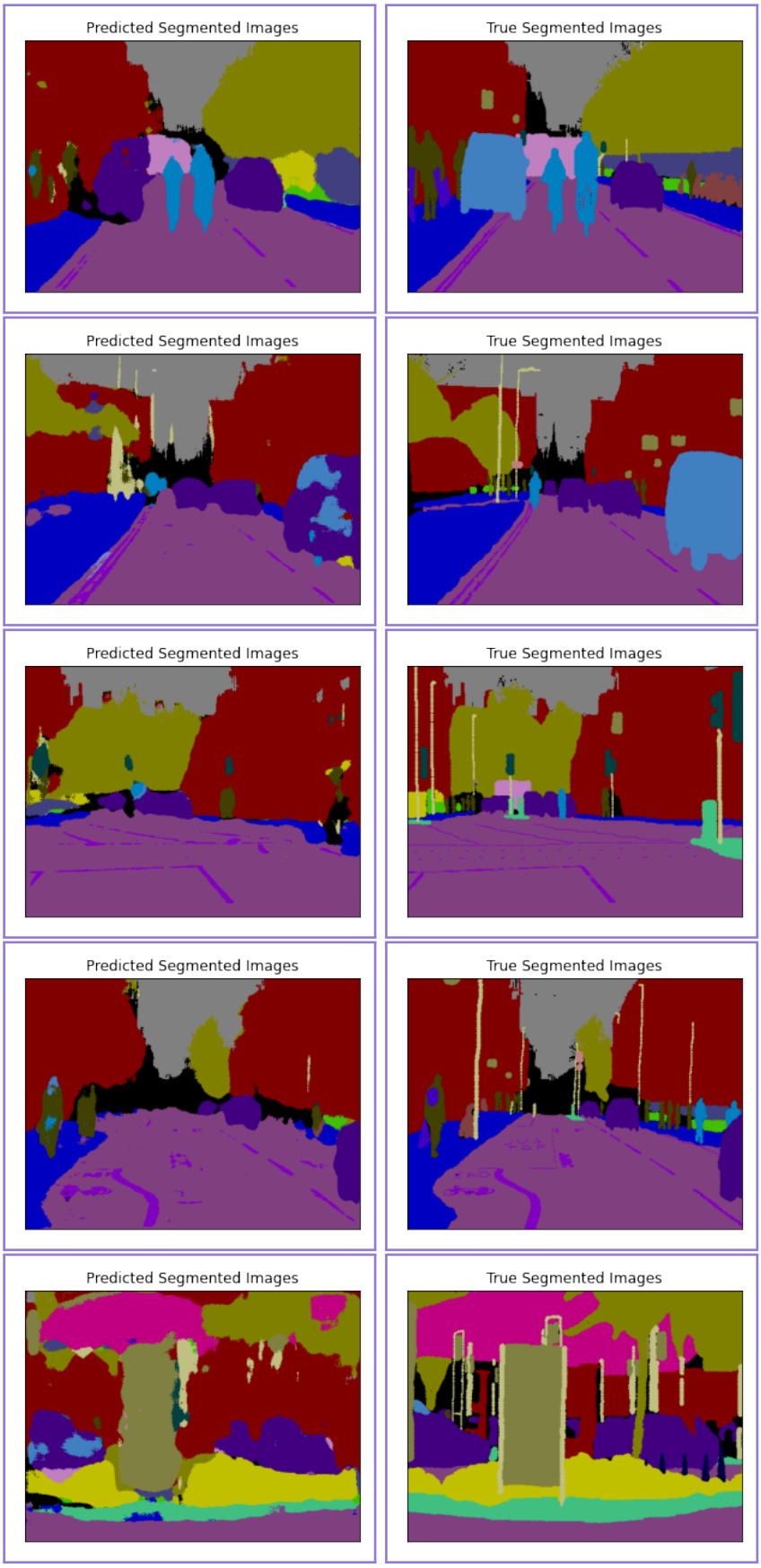

FIGURE 2.23: The true segmented image and the prediction of SegNet in the 31-35 epochs

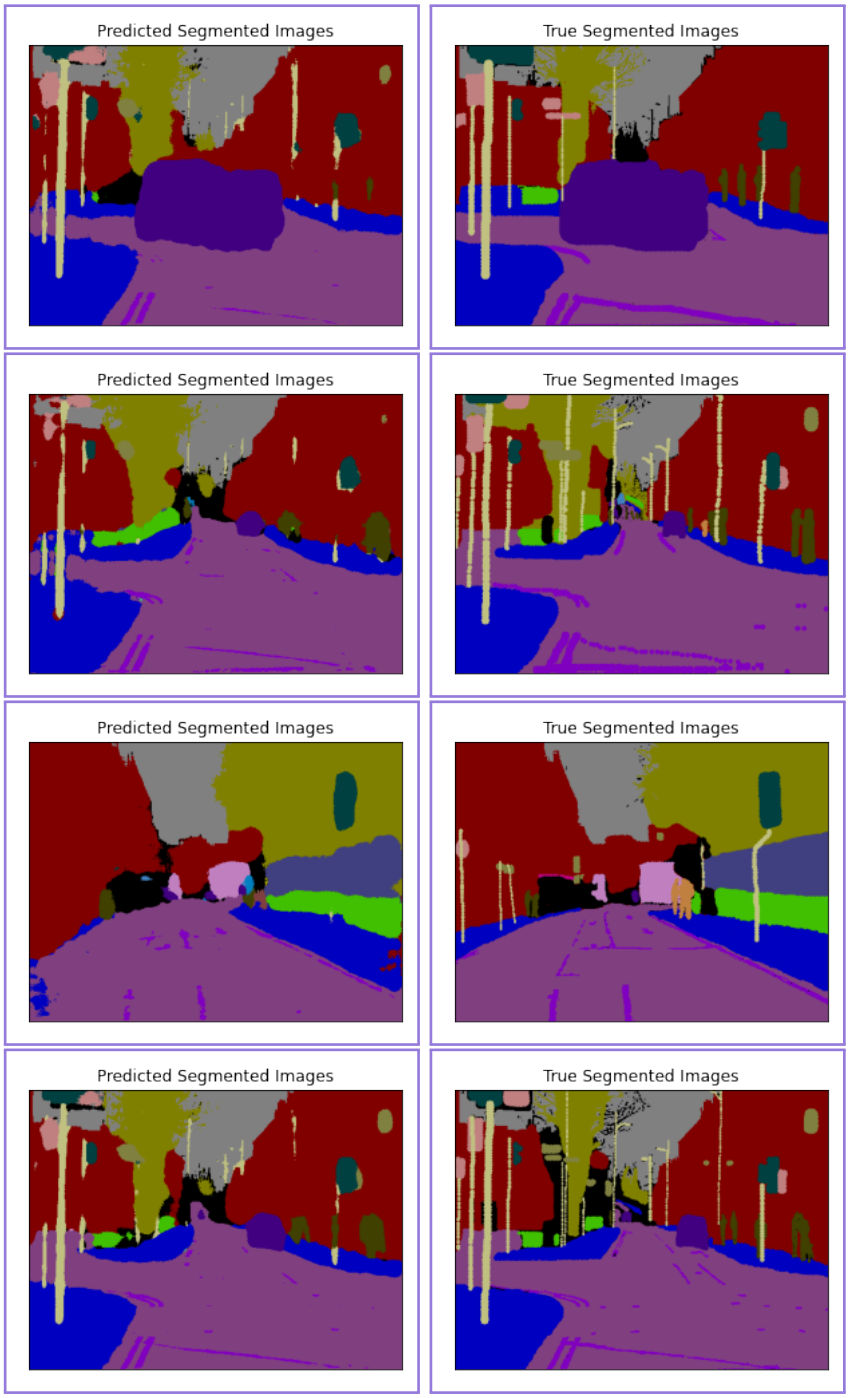

FIGURE 2.24: The true segmented image and the prediction of SegNet in the last 36-39 epochs

Therefore, in the cases where we have limitations in time or GPU computation, employing batch normalization will produce more accurate results. [6]

2.5 Part F: Training with All Data

Although it was not asked in the problem, the network was implemented with all the 700 data while the network included batch normalization layer. The result of the

[6]The same network but with Adam were implemented, but did not reach a good result!

last epochs can be seen below:

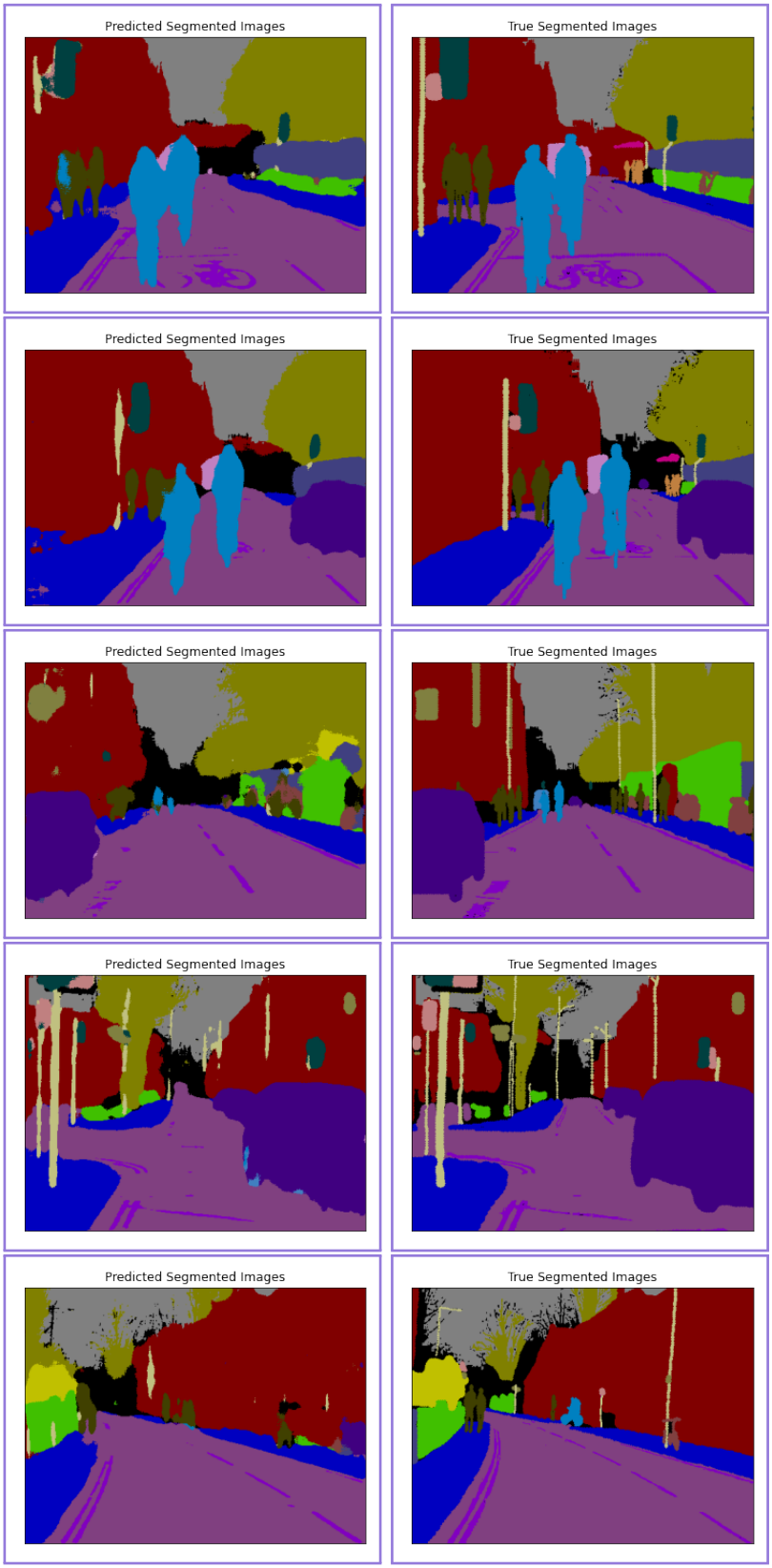

FIGURE 2.25: The true segmented image and the prediction of SegNet for all data

Chapter 3

Question 2

In this problem we are asked to implement a neural network that can count the number of individuals in a picture. To do so, some steps need to be taken to make the data ready for being the input of this NN. Therefore, in the following sections all the step to reach this goal is elaborated.

3.1 Preprocessing Data

3.1.1 Train and Test Images

First of all, the name and path of train data images are read and stored. Afterwards, we need to transform these images into gray scale. To do so, "RGB2Gray" is implemented. Besides, images needs to be patched into new images with size 256 $\times$ 256. Given the size of initial images, we can extract 12 images from it. Figure 3.1 shows the transformed image.

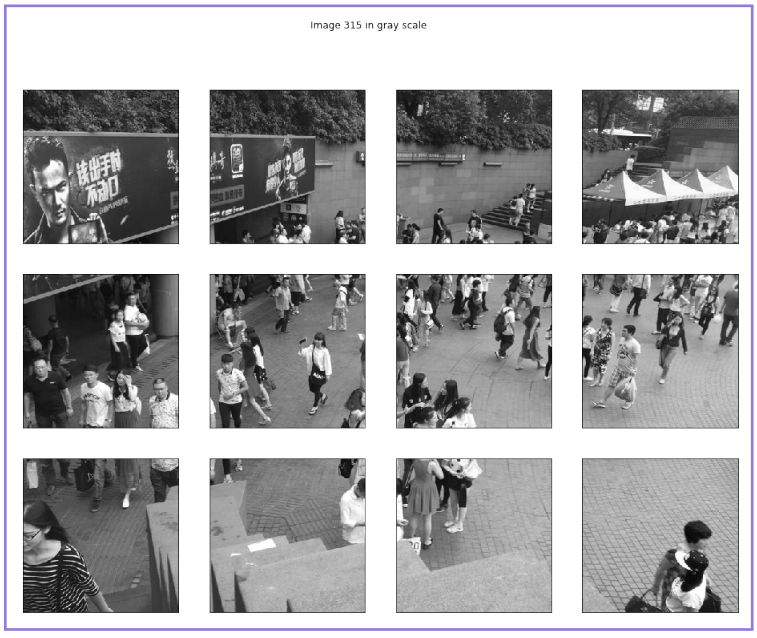

FIGURE 3.1: The 315th image in gray scale format with 12 patches

The same procedure is carried out for all the images and they are saved in a new location. What has been described is also implemented for test images and they are saved in the new location.

3.1.2 Train and Test Labels

For labels, we need to fit a Gaussian kernel on the given coordinates to make a density map of all the people in the given images. Afterwards, these images are cut into patches, same as what has been described in the previous section. If we do so, for the same image that was depicted in Figure 3.1 the density map will be as follows:

FIGURE 3.2: The 315th image's density map of people

Now the same procedure is done for all coordinates in training and test labels and are saved in new location.

3.1.3 Data Loading and Batches

The saved images and labels are then loaded to create batches. One thing to mention is that since in the CCNN has two max-pooling, the labels need to be re-sized (down-scaled) and then be the input for creating batches.

Now that data is ready, we can implement the NN with the given layers and loss function.

3.2 Training the Network

For implementing the network, the given architecture is used. The optimizer is set to SGD with learning rate = 0.001. After each 100 iteration, loss plot is depicted as well as a comparision of true and predicted labels.

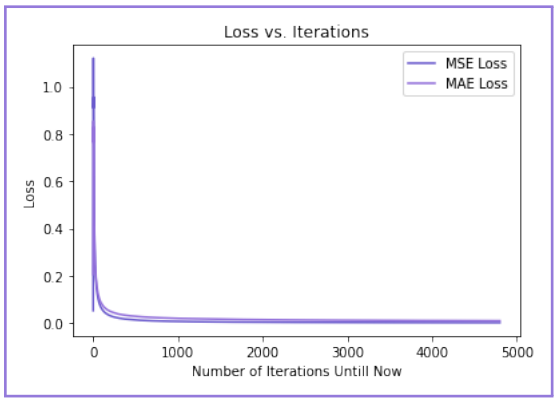

FIGURE 3.3: The two desired losses in the first epoch

Some of the predicted plots are as below:

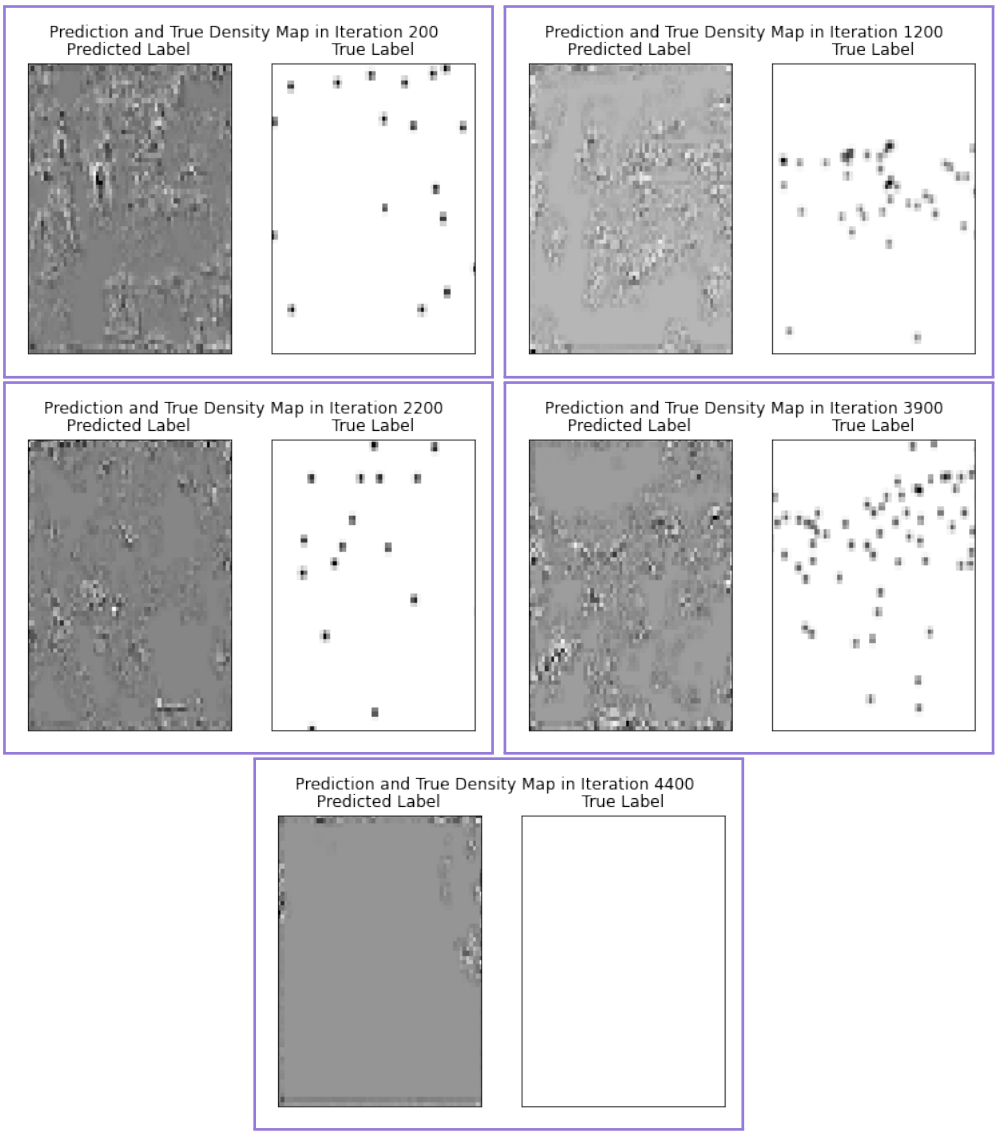

FIGURE 3.4: The true an predicted labels in the first epoch

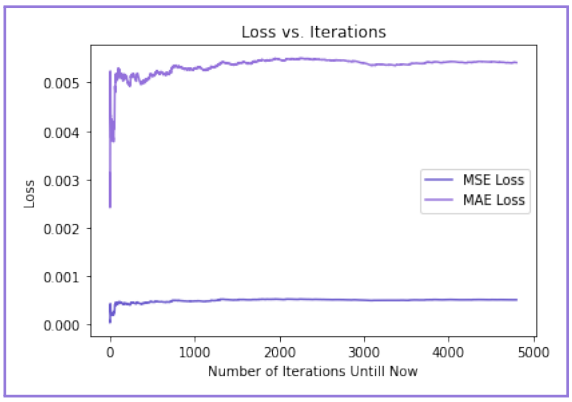

FIGURE 3.5: The two desired losses in the last epoch

Some of the predicted plots are as below:

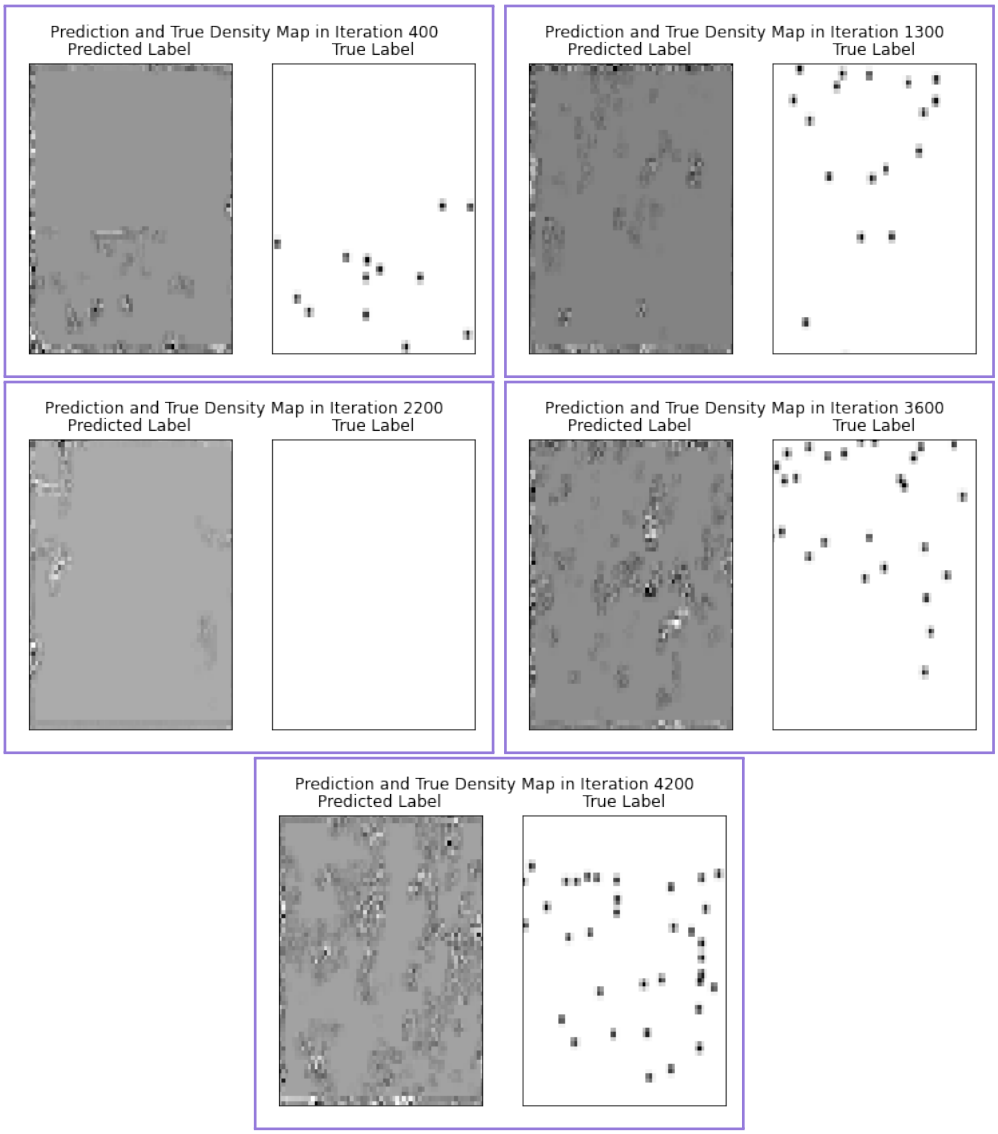

FIGURE 3.6: The true and predicted labels in the last epoch

As can be seen, the density map of various images are predicted in the best possible way. Due to the lack of GPU, only 5 epochs were run and the results are as

above. It is clear that with more epochs, the results could have been better.

The test loss after each epoch is set as the below:

	Epoch 1	**Epoch 2**	**Epoch 3**	**Epoch 4**	**Epoch 5**
MSE	0.00045	0.00053	0.00048	0.00060	0.00048
MAE	0.00600	0.00557	0.00545	0.00553	0.00511

TABLE 3.1: The loss of test data in the end of each epoch

As can be seen, through out the epochs, both losses on test dataset is decreasing.

Chapter 4

Appendix: How to Run the Codes

4.1 Problem 1

All the codes can be found in the codes folder. Note that first of all, the directory of files need to be set. Afterwards, running each notebook will show the results. Besides, my results can also be found in the codes as well as the .html formats of notebooks in case you did not have the time to run them!

4.2 Problem 2

All the codes can be found in the codes folder. Note that first of all, the directory of files need to be set. Afterwards, with "LoadTransformSave" part, let patched data be copied into the new direction. Then, you can run the cells regarding loading the datasets and then the network will start training.

Chapter 5

References

- Reading images with Pytorch
- Masking RGB images to indexes
- Source code for torchvision.transforms
- Transforms on PIL Image
- Crop on PIL Image
- Convert RGB image into grayscale
- Generating density map with fixed kernel
- Extract smaller patches from an image
- Crowd counting from scratch