PIXELWISE SEMANTIC SEGMENTATION Using Deep Convolutional Encoder Decoder Network

Summer Semester 2019 Master's in IT

Presented by,

Suraj Sanath Kumar Bharadwaj

Narayan Narvekar

CONTENTS

INTRODUCTION

Segmentation is extremely important for image analysis tasks. Semantic segmentation is the process of mapping each pixel of an image with a class label, (such as ocean, tree, road, person, sky or car). Semantic segmentation is an important step towards understanding and inferring different objects and their arrangements observed in a scene.

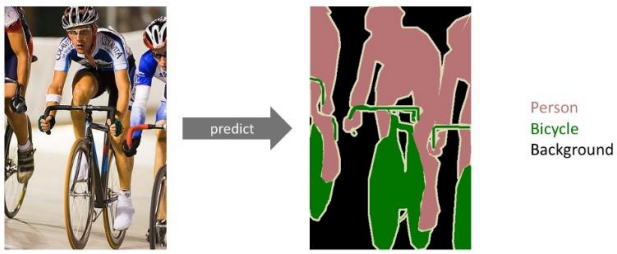

Figure 1: An example of semantic segmentation, where the goal is to predict class labels for each pixel in the image [1]

So one interesting thing about semantic segmentation is that it does not differentiate instances. Figure 2 on the right, the image with two cows where they're standing right next to each, all the pixels are labeled independently for what is the category of that pixel. So in the case like this where there are two cows right next to each other, the output does not distinguish between these two cows. Instead a whole mass of pixels that are all labeled as cow.

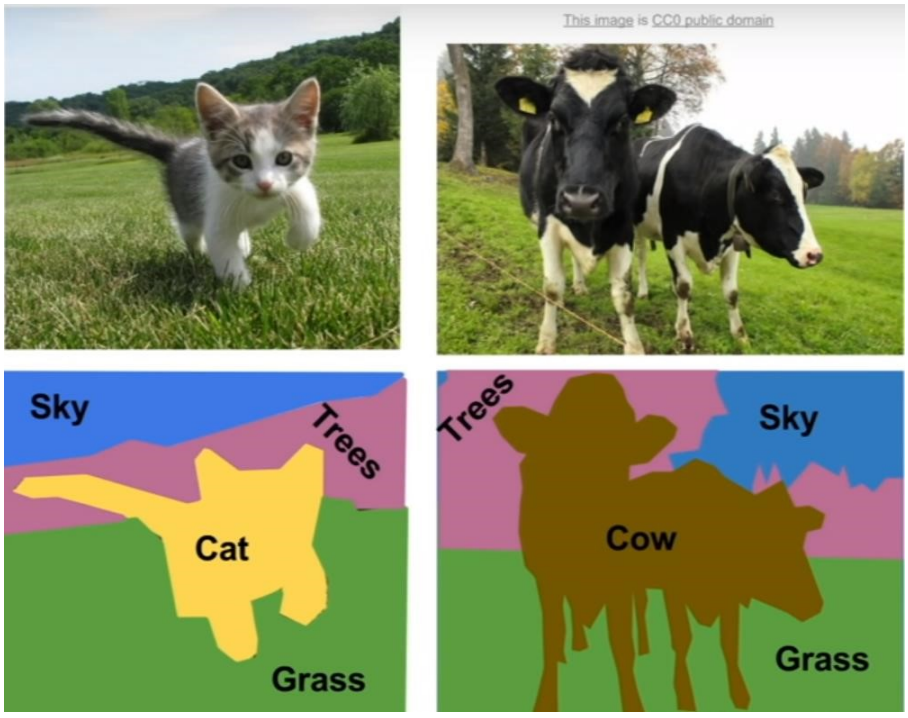

Figure 2: Semantic Segmentation example- Don't differentiate instances, only care about pixels

APPLICATIONS

It has a wide array of applications ranging from scene understanding, industrial inspection, Classification of terrain seen in satellite imagery, medical imaging analysis and inferring support-relationships among objects to autonomous driving. Previous methods which relied on low level vision cues have fast been supplanted by popular machine learning algorithms. In particular, deep learning has seen huge improvement lately in detecting objects in images, categorizing whole images speech, and handwritten digit recognition. Presently there is a huge interest for semantic pixel-wise labelling.

MOTIVATION

The motivation which lead to design of SegNet architecture arises from the requirement to map low resolution features to input resolution for pixel-wise classification. This mapping produces features which are useful for precise boundary localization. The primary motivation was the road scene understanding applications which require the ability to model the appearance (building, road), shape (pedestrians, cars) and understand the spatial-relationship (context) between different classes such as road and side-walk. Generally in road scenes, a majority of the pixels contributes to large classes such as road, building and therefore the network must produce smooth segmentations. Hence it is important to retain boundary information in the extracted image representation. From a computational viewpoint, it is fundamental for the system to be efficient regarding both computational time and memory during inference.

In SegNet, the encoder network is identical to the convolutional layers in VGG16. The fully connected layer is removed in order for training to be easy, by reducing the number of parameters. The decoder network consists of a hierarchy of decoders which corresponds to each encoder. Non-linear upsampling of input feature map is performed by each decoder utilizing the max-pooling indices resulted from the corresponding encoder. This idea was inspired from an architecture designed for unsupervised feature learning. [2]

Advantages of Reusing max-pooling indices in the decoding process:

- Boundary delineation is improved.
- End-to-end training achieved faster as the number of parameters are reduced.
- This form of upsampling can be incorporated into any encoder-decoder architecture. [3]

ARCHITECTURE

SegNet consists of an encoder network and corresponding decoder network, there is a final pixelwise classification layer in the end called softmax. This architecture is depicted in Figure 3.

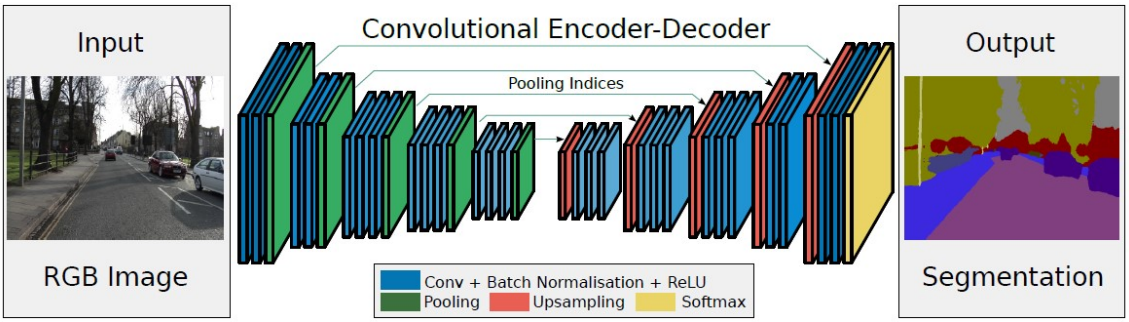

Figure 3: Segnet Architecture

The encoder network has 13 convolutional layers which correspond to the first 13 convolutional layers in the VGG16 network [4] designed for object classification. The fully connected layers is discarded from VGG16 network so that higher resolution feature maps is obtained at the deepest encoder output. As a result, the number of parameters is reduced in the encoder network significantly (from 134M to 14.7M) as compared to Fully Convolutional Networks and deconvolution network [5], [6].

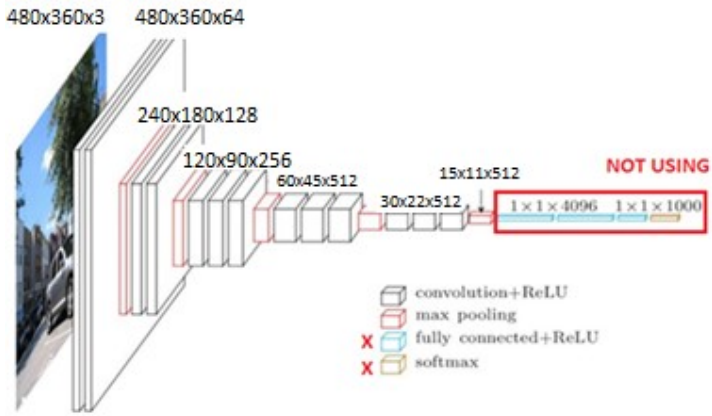

Figure 4: VGG16 Architecture. The first 13 convolutional layers are used. Fully Connected layer and softmax layer is discarded

The input to First convolutional layer is of fixed size 480*360 RGB image. As seen from Figure 3, each encoder carries out convolution with a filter bank. The image is passed through a stack of convolutional layers, where filters of size 3×3 is used, to have a very small receptive field (because smallest size to capture the notion of left or right, up or down and center). The convolution stride is fixed to 1 pixel. The padding is 1-pixel for 3×3 convolutional layers in order to preserve spatial padding after convolution.

The result of convolution with a filter bank is a set of feature maps. These feature maps are then Batch Normalized [7]. After Batch Normalization, an element-wise rectified linear non-linearity (ReLU) max(0, x) is applied. Following this, max-pooling with a 2x2 window having stride 2 is performed. The output is sub-sampled by factor of 2. Max pooling leads to a loss of spatial resolution of the feature maps. A lossy image representation is not favorable for segmentation where boundary delineation is very important. Therefore it is necessary to store boundary information of encoder feature maps before subsampling is performed. But storing this would require a lot of memory resources which is not feasible in practical applications. A more efficient way to do this is to store only the max pooling indices, i.e. the locations of maximum value in each pooling window. This value is memorized for each of the encoder feature map. It is implemented by making use of 2 bits for every 2x2 pooling window and hence very efficient as compared to memorizing the feature maps.

The Decoder network has 13 layers corresponding to each encoder layer. Every decoder in the decoder network up-samples its input feature map using the max-pooling indices which were memorized in the encoder layer from the corresponding encoder feature map. This step produces sparse feature map. This decoding technique is shown in Fig 5

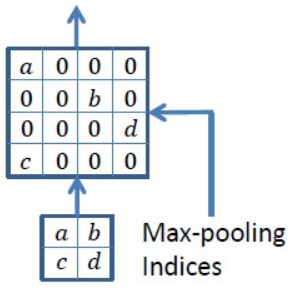

Figure 5: Decoding technique in SegNet

To produce dense feature maps, the sparse feature maps are then convolved with a trainable decoder filter bank. For each of these maps, a batch normalization step is applied. The decoder which corresponds to the first encoder i.e. the one which is just after input image, gives an output which is a multi-channel feature map, in spite of the fact that its encoder input has 3 channels (RGB). The number of channels in this feature map is 64. Except this, the other decoders in the network produces feature maps which have the same number of channels and size as their encoder inputs.

After the end of the decoder section, a final 1x1 convolution layer is applied to the decoder output to generate the predictions of our network. Each image prediction has the same height and width as the original image and a depth equal to the number of classes, i.e., 12. Every pixel of each of the 12 channels contains probabilistic values, with a higher value indicating a higher probability of that class being

classified for the pixel. For e.g., if for a particular pixel, the channel 5 has the highest probability value, and class 5 represents “Tree”, then that pixel will be classified as a tree in our prediction.

MAX POOLING

Max pooling is a type of operation that's typically added to CNN's following individual convolutional layers. When added to a model max pooling reduces the dimensionality of images, by reducing the number of pixels in the output from the previous convolutional layer.

Max Pooling works like this: An n x n region as a corresponding filter for the max pooling operation is defined. Let’s use 2x2 for an example, then a stride is defined, meaning by how many pixels must the filter move as it slides across the image, a value of 2 is used for this as well. From the convolutional output, the first 2x2 region is taken and the max value from each value in this 2x2 block is calculated, then the value which is going to be used to make up the full output from this max pooling operation is stored.

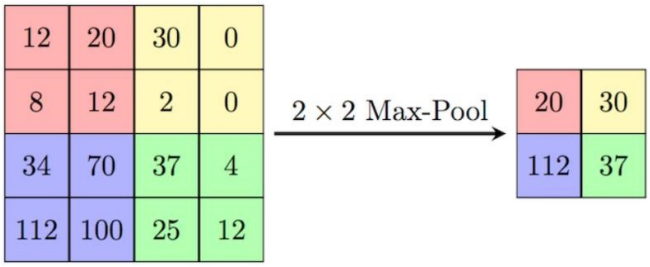

Figure 6: Max pooling

The resulting output is this 2x2 block here so our input dimensions were again reduced by a factor of 2.

Advantages of adding Max Pooling to the network:

- Since max pooling is reducing the resolution of the given output of a convolutional layer, the network will be looking at larger areas of the image at a time going forward, which reduces the amount of parameters in the network and consequently reduces computational load.
- Max pooling may also help to reduce overfitting.
- To achieve translation invariance over small spatial shifts in the input image.

Now the intuition for why max pooling works is that, for a particular image the network will be looking to extract some particular features. For e.g., if it's trying to identify numbers from the MNIST data sets, it's looking for edges and curves and circles. From the output of the convolutional layer, the higher

valued pixels are the ones that are the most activated. With max pooling, as we are going over each region from the convolutional output we are able to pick out the most activated pixels and preserve these high values going forward while discarding the lower valued pixels that are not as activated.

UPSAMPLING - MAX UNPOOLING

The strategy for increasing the size of a feature map inside the network is max unpooling, where for each unpooling, we add an upsampling layer which is associated with one of the pooling layers from the encoder section. In the encoder section, we remember all the indices of the max pooled pixels. In the decoder section, these stored indices are used to perform the upsampling. While upsampling, all the neighbouring pixels are filled with zeros. The unpooling operation is shown in Figure 7.

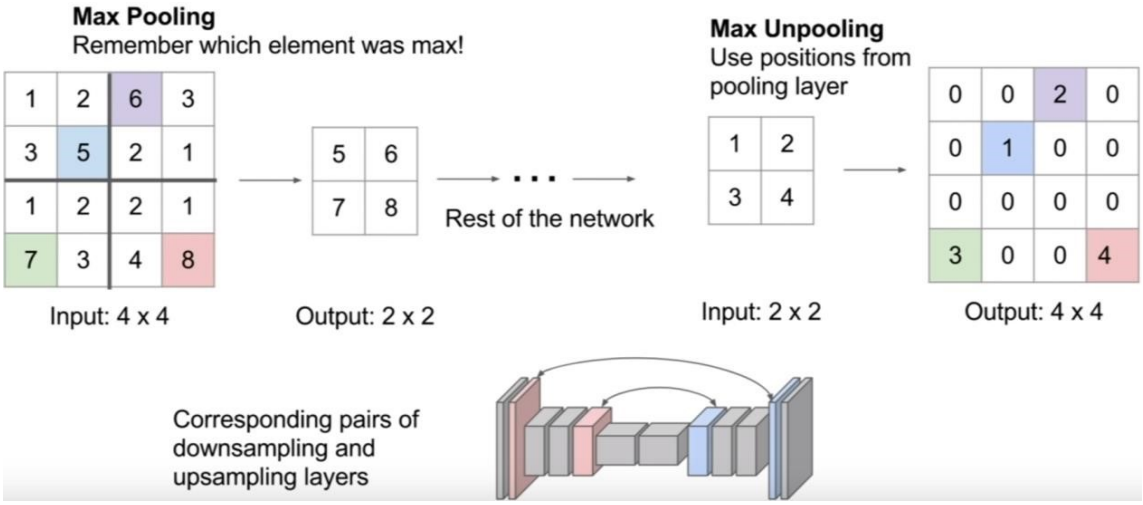

Figure 7: Unpooling Operation

The idea is that when semantic segmentation is done, the predictions should be pixel perfect. To get those sharp boundaries and those tiny details in the predictive segmentation, if max pooling is done, there is a sort of heterogeneity that's happening inside the feature map where, from the low resolution image, spatial information is lost in some sense but it is not known where that feature vector came from in the local receptive field after max pooling. If unpooling is done by putting the vector in the same slot it will help handle these fine details a little bit better and help preserve some of that spatial information that was lost during max pooling.

LOCAL RESPONSE NORMALIZATION

Before entering the images to the architecture, we perform *Local Response Normalization* on the images. Local Response Normalization layer implements a type of *lateral inhibition* that is useful while dealing

with ReLU neurons. *Lateral inhibition* is a concept from neurobiology that refers to the capacity of an excited neuron to subdue its neighbors. We need a significant peak so that there is some form of local maxima. This results in the creation of a contrast in that area, thereby increasing the sensory perception which is an advantage for the convolutional neural networks. Since ReLU neurons have unbounded activations, we need LRN to normalize that. We want to detect high frequency features with a large response. Normalizing around the local neighborhood of the excited neuron, results in it becoming even more sensitive as compared to its neighbors.

BATCH NORMALIZATION

Preprocessing to input data is done before training a neural network. If normalization of the data is not performed, then there will be some numerical data points in the data set that might be very high and others that might be very low. As a result these two pieces of data will not be on the same scale. The larger data points in these non-normalized datasets can cause instability in neural networks because the relatively large inputs can cascade down through the layers in the network which may cause imbalance gradients which may therefore cause the famous exploding gradient problem and this imbalanced non-normalized data may cause problems with our network that make it drastically harder to train. All the data are put on the same scale to increase training speed as well as avoid the problem.

There's another problem that can arise even with normalized data. When the neural network learns, weights in model become updated over each epoch during training. During training, in order to avoid one of the weights become drastically larger than the other weights which will then cause the output from its corresponding neuron to be extremely large and this imbalance will again continue to cascade through the neural network causing instability, batch normalization is applied. This problem is known as **internal covariate shift**.

During training time, a batch normalization layer performs the following:

1. Calculate the mean and variance of the input layers.

$$\mu_B = \frac{1}{m}\sum_{i=1}^{m} x_I \qquad \text{Batch Mean}$$

$$\sigma_B^2 = \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_B)^2 \qquad \text{Batch Variance}$$

2. Normalize the layer inputs using the previously calculated batch statistics.

$$\overline{x}_i = \frac{x_i - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}}$$

3. Scale and shift in order to obtain the output of the layer.

$$y_i = \gamma \overline{x}_i + \beta$$

γ and β are learned during training along with the original parameters of the network.

ADAM OPTIMIZER

Adaptive Moment Estimation (Adam) computes adaptive learning rates for each parameter. Like Adadelta and RMSprop which stores an exponentially decaying average of past squared gradients v_t, in addition Adam also stores an exponentially decaying average of past gradients m_t, similar to momentum. Whereas momentum can be seen as a ball running down a slope, Adam behaves like a heavy ball with friction, which thus prefers flat minima in the error surface [8].

Decaying averages of past and past squared gradients m_t and v_t are computed respectively:

$$m_t = \beta_1 m_{t-1} + (1-\beta_1) g_t$$

$$v_t = \beta_2 v_{t-1} + (1-\beta_2) g_t^2$$

m_t and v_t are estimates of the first moment (mean) and the second moment (variance) of the gradients respectively, hence the name of the method. m_t and v_t are initialized as vectors of 0's, and therefore it is observed that they are biased towards zero, during the initial time steps, and when the decay rates are small (when β_1 and β_2 are close to 1).

The bias-corrected first and second moment estimates are computed as:

$$\hat{m}_t = \frac{m_t}{1-\beta 1^t}$$

$$\hat{v}_t = \frac{v_t}{1-\beta 2^t}$$

Then these parameters are used to update the parameters, which yields the Adam update rule:

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\hat{v}_t + \epsilon}} \hat{m}_t$$

The default values of 0.9 for β_1, 0.999 for β_2, and 10^{-8} for ϵ are proposed in this paper [9].

MEDIAN FREQUENCY BALANCING

Cross-entropy loss [5] as an objective function is used for training the network. For all the pixels in a mini-batch, the loss is added up. Class balancing is performed whenever there is a huge variation in the number of pixels, in each class in the training set. For e.g. building, road and sky pixels influence the CamVid dataset, so it is necessary to weight the loss differently based on the true class.

Median frequency balancing [10] is process of assigning the weight to a class in the loss function, which is the ratio of the median of class frequencies calculated on the entire training set(median_freq) divided by the class frequency(freq(c)).

Median Frequency Balancing: alpha_c = median_freq/freq(c).

freq(c) is the number of pixels of class c divided by the total number of pixels in images where c is present.

Median Frequency balancing interpretation:

1. “Number of pixels of class c”: Represents the total number of pixels of class c across all images in the dataset.
2. “The total number of pixels in images where c is present”: Represents the total number of pixels across all images, where there is at least one pixel of class c, of the dataset.
3. "Median frequency is the median of these frequencies": Calculated by sorting the frequencies calculated above and picking the median.
4. The larger classes in the training set have a weight smaller than 1 and the weights of the smallest classes have the highest values.

IMPLEMENTATION DETAILS

This section provides an insight on the important files, classes and methods which are a part of the algorithm. The project is developed using the open source machine learning library Tensorflow Version 1.13.1.

config.json:

This file contains the following configuration parameters which will be read by the python script:

TRAIN_FILE – Contains the directory path of the text file which contains the path to the training images and corresponding labels.

VAL_FILE – Contains the directory path of the text file which contains the path to the validation images and corresponding labels.

TEST_FILE – Contains the directory path of the text file which contains the path to the test images and corresponding labels.

IMG_PREFIX – Prefix that needs to be added to the image file path before passing it as an argument to generate the random batch of images.

LABEL_PREFIX – Prefix that needs to be added to the file path of labels before passing it as an argument to generate the random batch of labels.

OPT – The optimizer that will be used to perform backpropagation.

INPUT_HEIGHT – Number of rows of images.

INPUT_WIDTH – Number of columns of images.

INPUT_CHANNELS – Number of channels of images. E.g., Red, Blue and Green.

NUM_CLASSES – Number of independent classes present in the images. E.g., Road, Building, etc.

USE_VGG – Flag that decides whether to use the trained VGG model weights for initializing the network parameters.

VGG_FILE – File from which the weights and biases of the VGG model are read in case USE_VGG is true.

TB_LOGS – Name of the folder where the tensor board logs will be saved.

SAVE_MODEL_DIR – Contains the directory path where the trained models will be saved at regular intervals.

segnet.py:

This python file contains the *SegNet* class which includes the tensorflow computational graph the methods to perform the training, validation and testing, and also methods to visualize the results after the network is trained. A brief description of the methods contained within the class is as follows:

- *def __init__(self, conf_file="config.json"):*

  This method acts as the constructor of the SegNet class and contains instructions to build the tensorflow graph of our network.

  The following placeholders are defined in our network:

  - *batch_size_pl* – This placeholder shall contain the number of images that should be processed in an iteration. The type of elements fed to the tensor is *int64* and the shape of the tensor is [].
  - *inputs_pl* – This placeholder shall contain the batch of images that shall be input to the training, validation or testing algorithm. The type of elements fed to the tensor is *float32* and the shape of the tensor is [batchSize, INPUT_HEIGHT, INPUT_WIDTH, NUM_CLASSES].
  - *labels_pl* – This placeholder shall contain the labels corresponding to the input images that shall be input to the training, validation or testing algorithm. The type of elements fed to the tensor is *int64* and the shape of the tensor is [batchSize, INPUT_HEIGHT, INPUT_WIDTH, 1].

  The computational graph is then built by adding sequential function calls to every encoder layer that is a part of the network, followed by every decoded layer. After the end of the decoder section, a final convolution layer is added to the graph to obtain the logits. The shape of the logits is [batchSize, INPUT_HEIGHT, INPUT_WIDTH, NUM_CLASSES].

- *def train(self, max_steps, batch_size):*

  This method contains the instructions to deploy the graph built in the constructor method, to train the network. The input arguments to this method are *max steps* and *batch size,* which indicate the total number of iterations to be executed and the number of images to be trained in each iteration.

  As a first step, the training and validation image file paths are read and input to functions that return a random batch of images and corresponding labels. This operation is added to the graph. The operation that calculates the *loss, accuracy* and *prediction* of the training images, and the method that performs the gradient optimization is also added to the graph. The graph is then deployed inside a session by using *sess.run().* The two operations defined above in this method are thus evaluated using *sess.run().* After every iteration, the Iteration Number, Training Loss and Training Accuracy are printed. After every 1000 iterations, we evaluate our network on the validation dataset and print out the Validation Loss and Validation Accuracy. Also, after every 1000 iterations, we save our model at the path provided in the configuration file. This saved model contains the entire computational graph and the parameters of the network, which can then be restored at any point of time to resume training with the saved weights and biases.

For debugging purpose, we also use TensorBoard, and we write the merged summaries to tensorboard after every 100 iterations.

- *def test(self):*

  This method contains the instructions to deploy the graph built in the constructor method, to evaluate the performance of the network on the test images. The operation that calculates the *loss, accuracy* and *prediction* of the test images is added to the graph. The operation that returns the list of test images and corresponding labels is also added to the graph. These two operations are thus evaluated using *sess.run().* For every test image and label that have been evaluated, we print the Image Index, Test Loss and Test Accuracy.

- *def visual_results(self, dataset_type, images_index):*

  This method can be invoked to visually observe the segmentation performance of our trained network. The input arguments to this method are *dataset_type* and *images_index,* which indicate the type of the dataset from which you want to visualize the predictions (train/validation/test) and the number of images that should be randomly selected from the specified dataset to perform the predictions. The operation that performs the prediction of the selected images is added to the graph. This operation is then evaluated using sess.run().

layers_object.py:

This python file contains methods that are called from the computational graph to build the encoder and decoder layers. A brief description of the methods is as follows:

- *def max_pool(inputs, name):*

  This method is used to perform the max pooling operation. The input arguments to this method are *inputs* and *name,* which specify the input tensor on which max pooling should be performed and the name of the encoder layer which is used to define the scope of the max pooling operation respectively. The method returns the max pooled tensor, the indices of max pooled values and the shape of the input tensor.

- *def conv_layer(input_map, name, shape, use_vgg, vgg_param_dict):*

  This method is used to perform the convolution operation. The input arguments to this method are *input_map, name, shape, use_vgg, vgg_param_dict,* which specify the input tensor on which convolution is to be performed, name of the encoder/decoder layer which is used to define the scope of the parameters, shape of the convolution filter, flag to indicate if vgg parameters shall be used

and the loaded vgg parameters respectively. The method returns the convoluted output. The method also contains three histogram summaries for weights, biases and the convolution outputs.

- *def batch_norm(bias_input, scope):*
  This method is used to perform the batch normalization. The input arguments to this method are *bias_input* and *scope,* which indicate the activations on which the batch normalization is to be applied and the scope of the variables. The method returns the batch normalized output.

- *def up_sampling(pool, ind, output_shape, batch_size, name):*
  This method is used to perform the upsampling operation, from the decoder part of the graph. The input arguments to this method are *pool, ind, output_shape, batch_size, name,* which indicate the input tensor that should be unpooled, the max pool indices, the expected shape of the unpooled tensor, the number of images to be unpooled at a time, and the name of the decoder layer which is used to define the scope of the operation respectively. The method returns the unpooled tensor.

- *def initialization(k, c):*
  This method takes the filter size and the number of channels in the filter as input arguments and generates a truncated normal distribution using these values to initialize the weights of that filter.

evaluation_object.py:

This python file contains methods to evaluate the losses, accuracies and perform gradient optimization through backpropagation. The methods contained in this file are described as follows:

- *def cal_loss(logits, labels):*
  This method assigns weights to the individual classes and returns the loss, accuracy and predictions of the *logits* that are received as an input parameter to this method. The second input parameter is *labels* which represent the actual labels of the images whose logits are calculated.

- *def weighted_loss(logits, labels, number_class, frequency):*
  This method is invoked from the train method to calculate the weighted loss of the logits that are evaluated during the training and validate phase of the network. The input parameters of this method are the *logits, labels, number_class, frequency,* which indicate the final layer predictions of our network, the labels corresponding to the images whose predictions are calculated, the number of classes read from the configuration file and the frequency of occurrence assigned to every class respectively. This method returns the loss, accuracy and predictions of the *logits* that are received as an input parameter to this method.

- *def normal_loss(logits, labels, number_class):*

  This method is invoked from the test method to calculate the normal loss of the logits that are evaluated during the testing phase of the network. The input parameters of this method are the *logits, labels, frequency,* which indicate the final layer predictions of our network, the labels corresponding to the images whose predictions are calculated and the number of classes read from the configuration file respectively. This method returns the loss, accuracy and predictions of the *logits* that are received as an input parameter to this method.

- *def train_op(total_loss, opt):*

  This method is invoked from the train method to perform the back propagation operation by computing the gradients using the Adam optimizer or Stochastic Gradient Descent optimizer. The input parameters to this method are *total_loss* and *opt,* which indicate the total loss calculated by the other functions defined in this file and the optimizer to be used as defined in the configuration file respectively.

inputs_object.py:

This python file contains methods which assist in providing access to the various datasets and create image and label batches for the training process. A brief description of the methods contained in the file is as follows:

- *def get_filename_list(path, config):*

  This method is invoked from the methods defined in *SegNet* class, to return the list of images and its labels. The input parameters of this method are *path* and *config,* which indicate the path of the dataset file read from the configuration file and the configuration file content respectively.

- *def dataset_inputs(image_filenames, label_filenames, batch_size, config):*

  This method is invoked from the train method to retrieve a random batch of images and labels. The input parameters to this method are *image_filenames, label_filenames, batch_size* and *config,* which indicate the list of image file paths, list of label file paths, number of images to be retrieved and the configuration file content respectively.

- *def dataset_reader(filename_queue, config):*

  This method returns the pixel values of one image and label from the list of images and labels received as input to the method. The other input parameter is *config,* which indicates the content of the configuration file.

- *def get_all_test_data(im_list, la_list):*

  This method returns the pixel values of all the test images and their corresponding labels. The input arguments to this method are *im_list* and *la_list,* which indicates the file paths of the test images and their corresponding labels respectively.

- *def _generate_image_and_label_batch(image, label, min_queue_examples, batch_size, shuffle):*
  This method constructs and returns a queued batch of images and labels. The input arguments to this method are *image, label, min_queue_examples, batch_size* and *shuffle,* which indicate the 3-D image tensor, the 3-D label tensor, minimum number of samples to retain in the queue that provides the batches of examples, number of images in a batch and a flag to decide whether to use a shuffling queue or not respectively.

drawings_object.py:

This python file contains methods that are invoked from the *visual_results()* method to plot the predictions of our neural network. The methods contained in this file are as follows:

- *def draw_plots_bayes(images, labels, predicted_labels):*

  This method plots the input image, the color coded ground truth image, and the color coded output image that is predicted by our network. The input parameters to this method are *images, labels, predicted_labels,* which indicate list of images to be displayed, list of corresponding labels and list of corresponding predictions of our network for these images.

- *def writeImage(image):*

  This method is used to assign the label and predicted images to colored images by using a unique RGB color code for each of the twelve classes and plot the resulting colored images. The input parameter to this method is *image,* which indicate the label or predicted image pixel values that need to be color coded.

- *def display_color_legend():*

  This method is used to assign and plot legends corresponding to each of the 12 classes.

TRAINING

The neural network uses the CamVid road scenes dataset. This dataset consists of 367 training, 233 testing images and 101 validation RGB images. The network is designed to classify each pixel of the images to one of the 12 classes. Every image has associated with it a labelled image. Since there are 12 classes, the pixel values of the labelled image are equal to the class indices, i.e. all the pixel values range from 0 to 11 for the labels. From these labels, we generate the ground truth images by assigning a color code to each of the 12 classes. An example of the original image, its corresponding label and ground truth is shown below:

Figure 8: (a) Input Image, (b) Labelled Image, (c) Ground Truth

The encoder and decoder filter weights have been initialized with the technique explained in [11]. The values for a particular filter are generated from a truncated normal distribution, whose standard deviation is given as follows:

$$\text{Standard deviation} = \sqrt{\frac{2}{k^2 * c}}$$

where, k is the filter size and c is the number of input channels for that filter.

The training is started by feeding the images and its labels to the tensorflow graph via placeholders. Before entering the images to the architecture, we perform *Local Response Normalization* on the images as explained in a previous chapter. The network was trained for a total of 19000 iterations, with 3 images processed in every iteration. Since there are 367 images in the training dataset, it corresponds to a total of 155 epochs. After every 1000 iterations, the network is evaluated on the validation dataset in order to estimate how well the network has been trained.

The cross entropy loss is used as the objective function to train the network. The loss is added up over all the pixels present in a mini-batch. Since there is a possibility of a large variation in the number of pixels corresponding to each class in the training set (e.g., sky, road and building dominate the CamVid dataset), a need arises to weight the loss differently based on the true class. This is known as class balancing and we use *median frequency balancing* which is explained in one of the above chapters.

Google Colaboratory:

In order to speed up training, we used the Google Colaboratory which is a free cloud service that provides a possibility of utilizing a free GPU for training the network. Colaboratory works with our own google drive. This requires us to store all the project relevant files and datasets in our google drive. We then need to mount the google drive by running the following code snippet:

from google.colab import drive

drive.mount('/content/drive')

This lets us access any file store in our google drive from Colaboratory, and thus we can proceed with the usual python commands to train the model. Also, since the Colaboratory resources are available for usage only for twelve hours at a time, we save our model at regular intervals. The interval that we selected was 1000 iterations. Thus, when we restart a new Colaboratory session, we first load the last saved model through which we can initialize the parameters of our network with the values from the saved model, thereby preventing the need to restart training from scratch after every twelve hours. The following figure shows a snapshot of the Google Colaboratory user interface, with the mounted google drive and the python notebook:

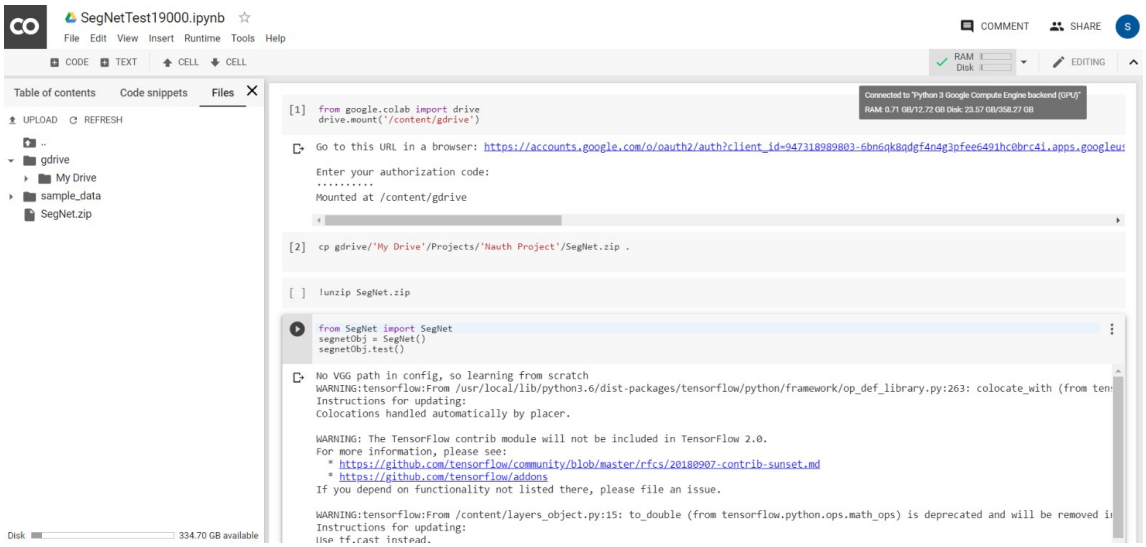

Figure 9: Google Colaboratory snapshot

ANALYSIS

To analyze the performance of our network, we consider the global accuracy which provides a measure of the percentage of pixels that were correctly classified in the dataset. The training and validation accuracy over a total of 19000 iterations is as shown in the following graphs.

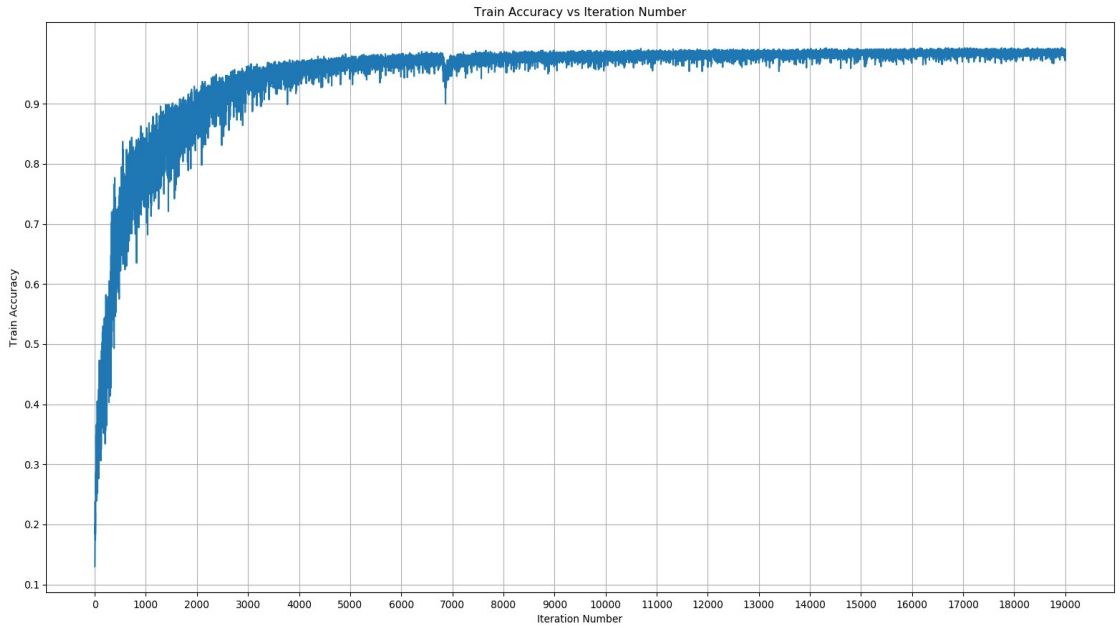

Figure 10: Train Accuracy vs Iteration Number

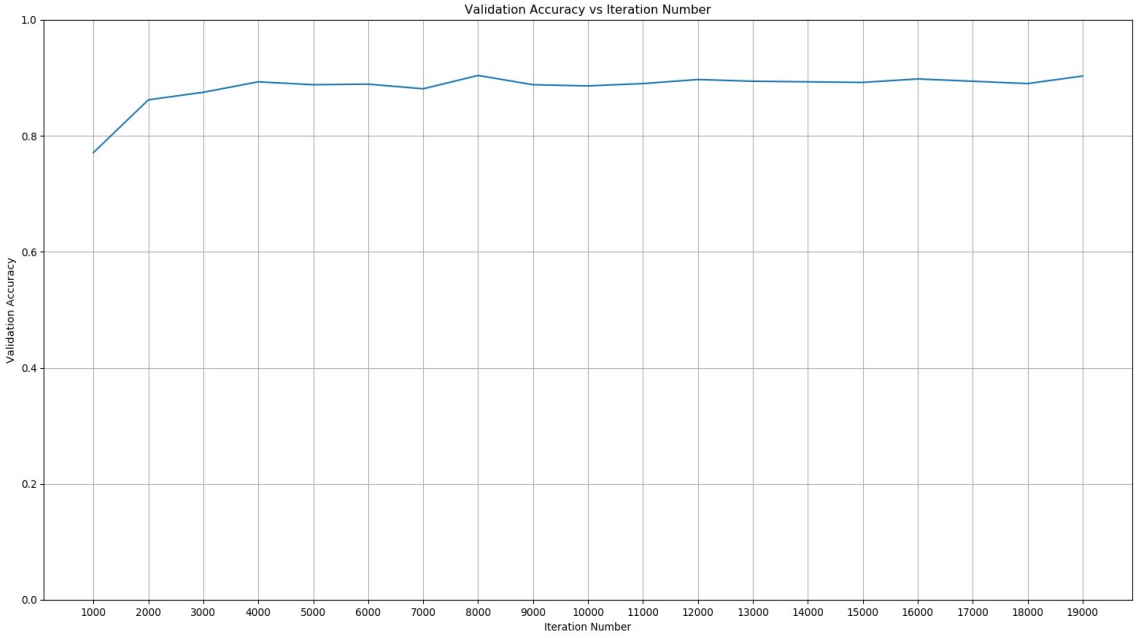

Figure 11: Validation Accuracy vs Iteration Number

The training and validation losses over all the iterations are also calculated and are shown in the following graphs. We can see how the losses decrease over each iteration, just as we had expected, which is a good indication that the network is being optimized at every iteration.

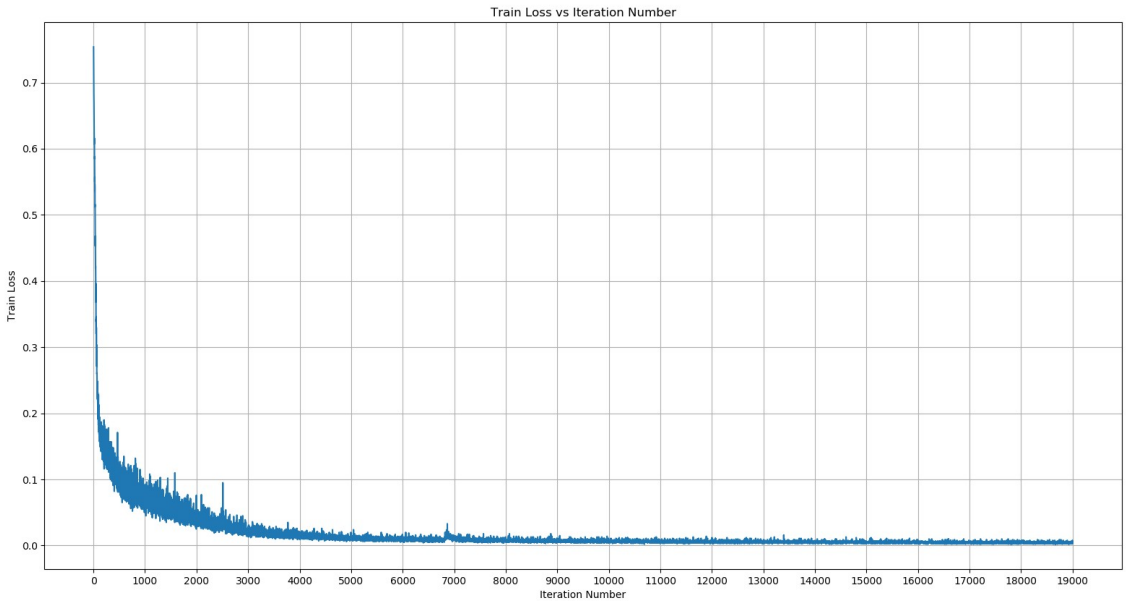

Figure 12: Train Loss vs Iteration Number

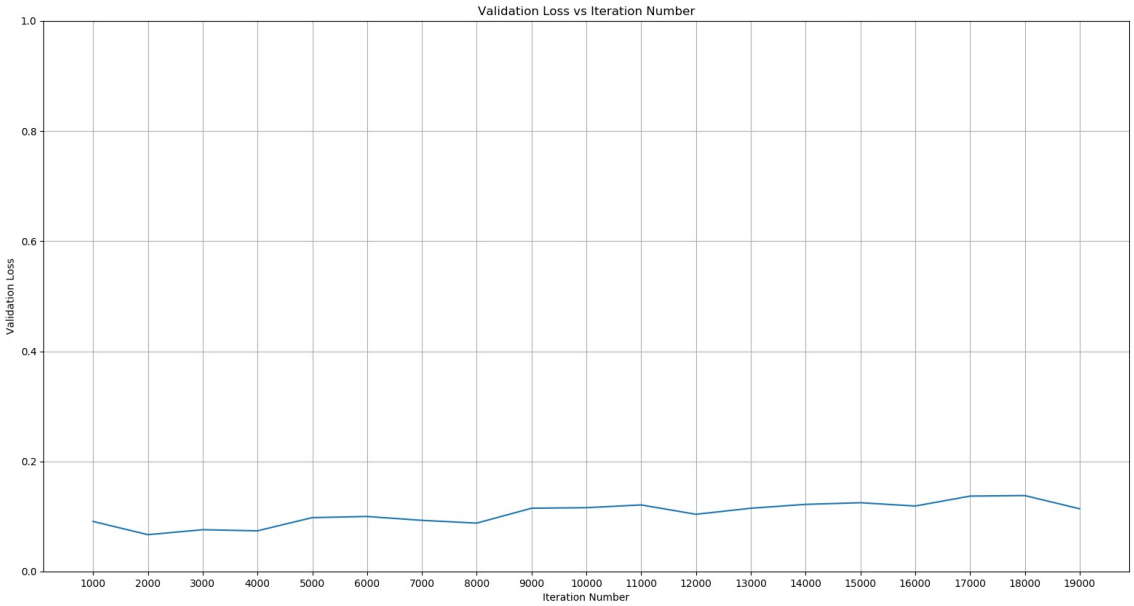

Figure 13: Validation Loss vs Iteration Number

Since after 19000 iterations, we don't observe any further improvement in our validation accuracy, we abort the training. The trained model is then evaluated on a new dataset which the network has never seen before. This is the test dataset which contains a total of 233 images as specified earlier. The final

test accuracy that we obtain is 83.47%, which is calculated by averaging the individual accuracies of each of the test images.

Finally, we show the predicted class labels for different images in the train, validation and test datasets. This can be done by using the *visual_results* method from the *SegNet* class. For each displayed image, we show the original image, its ground truth and the predicted output of our model. Also displayed are the color legends for the different classes.

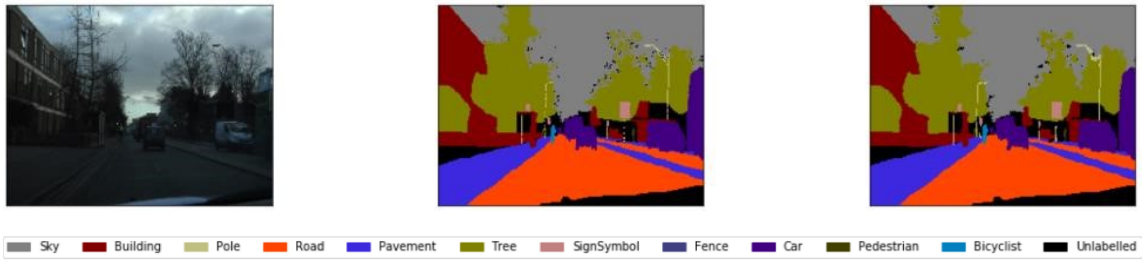

Figure 14: Train dataset, (a) Input image, (b) Ground Truth, (c) Predicted Output

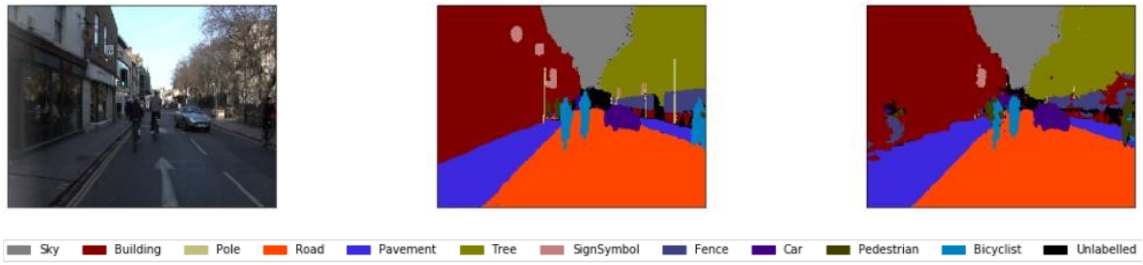

Figure 15: Validation dataset, (a) Input image, (b) Ground Truth, (c) Predicted Output

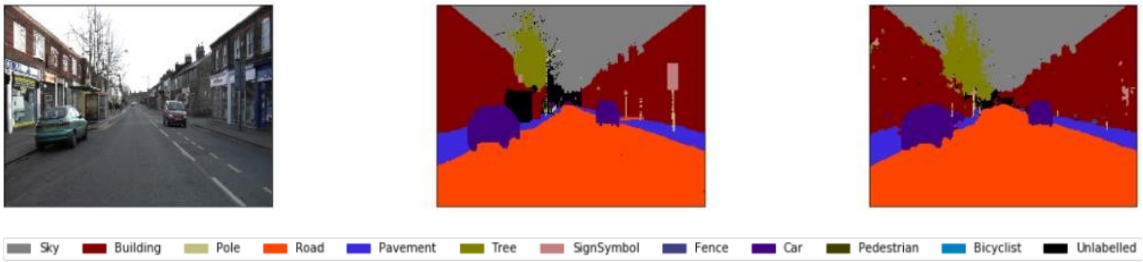

Figure 16: Test dataset, (a) Input image, (b) Ground Truth, (c) Predicted Output

CONCLUSION

In this project, we demonstrated a deep convolutional network architecture for semantic segmentation. The main motivation behind this was the requirement to design an architecture which is effective for road scene understanding and efficient both in terms memory as well as computational time. Most of the semantic segmentation architectures store the complete encoder network feature maps. Although they perform well, they have the disadvantage of consuming more memory during inference time. The architecture proposed in this project is more efficient since it stores only the max-pooling indices of the feature maps which are later used in the decoder network to achieve good performance. On large and well known datasets, this architecture performs competitively, resulting in high accuracy scores for road scene understanding.

REFERENCES

[1] J. Jordan, "An overview of semantic image segmentation.," 20 May 2018. [Online]. Available: https://www.jeremyjordan.me/semantic-segmentation/.

[2] M. R. L. J. H. Y-Lan Boureau, "Unsupervised Learning of Invariant Feature Hierarchies with Applications to Object Recognition," *2007 IEEE Conference on Computer Vision and Pattern Recognition,* vol. 1, pp. 1-8, 2007.

[3] A. K. R. C. Vijay Badrinarayanan, "SegNet: A Deep Convolutional Encoder-Decoder Architecture for Image Segmentation," in *IEEEE*, 2016.

[4] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," 2014.

[5] E. S. a. T. D. J. Long, "Fully convolutional networks for semantic segmentation," 2015.

[6] S. H. a. B. H. H. Noh, "Learning deconvolution network for semantic segmentation," 2015.

[7] S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep S. Ioffe and C. Szegedy, “Batch normalization: Accelerating deep," vol. vol. abs/1502.03167, 2015.

[8] S. Ruder, "An overview of gradient descent optimization algorithms," 19 Jan 2016. [Online]. Available: http://ruder.io/optimizing-gradient-descent/index.html#fn14.

[9] D. P. &. B. J. L. Kingma, "Adam: a Method for Stochastic Optimization. International Conference on Learning Representations," pp. 1-13, 2015.

[10] D. E. a. R. Fergus, "Predicting depth, surface normals and semantic labels with a common multi-scale convolutional architecture," vol. 2650–2658, 2015.

[11] X. Z. S. R. a. J. S. K. He, "Delving deep into rectifiers: Surpassing human-level performance on imagenet classification,," *ICCV,* vol. 1026–1034, 2015.

[12] M. u. Hassan, "VGG16 – Convolutional Network for Classification and Detection," 20 November 2018. [Online]. Available: https://neurohive.io/en/popular-networks/vgg16/.