Udacity Self Driving Car Nanodegree – Project: Semantic Segmentation

By:

Saurabh Sohoni

Udacity profile: https://profiles.udacity.com/p/u183059

Contents

1. Introduction

This document acts in support with code and output model created in order to implement 'Semantic Segmentation' project. In this project, a fully convolutional deep learning network is trained using TensorFlow [1] on images of city and highway roads taken from the KITTI Road Dataset [2]. Loss encountered during training is noted and the model is applied on test images to segment sections of road from rest of the image. This segmented portion represents area on which a vehicle can be driven.

2. Project Goals

Following were the goals of this project:

a. Load city and highway road images taken from the KITTI Road Dataset [2].
b. Explore the dataset by checking the number of samples present in training and test sets.
c. Load an existing image classifier network such as VGG-16 [3], GoogLenet [4] or Resnet [5]. Explore the architecture and convert it to fully convolutional network for image segmentation.
d. Add a decoder network after last layer of image classifier network to get back original image with segmented pixels. This decoder network will consist of 1x1 convolutions, up samplers and skip connections.
e. Train the network on training dataset and calculate training loss using cross entropy. Tune different parameters such as the learning rate, number of epochs, batch size, weight initialization parameters, etc. to obtain minimum loss.
f. Test the network on test dataset and validate the segmented images on output. Ensure the network labels at least 80% of the road and labels no more than 20% of non-road pixels as road.

3. Implementation of Project Rubric Points

Following section lists down various rubric points for this project and also details out the implementation strategy followed:

3.1 Submission of code and output files

The submission includes following files:

1. main.py – This file contains code for loading of VGG-8 classifier, implementation of decoder, calculation of loss parameters, training on train data set and testing on test data set
2. project_tests.py – This file contains unit tests for each method in main.py
3. helper.py – This file contains helper functions used in main.py and project_tests.py
4. inference_images – This folder contains segmented images received as an output when the network is applied on test images
5. writeup_report.pdf – Current file summarizing about the project goals and results obtained

3.2 Implementation of Neural Network

Fully convolutional neural network was built by using FCN-8 [6] architecture shown below:

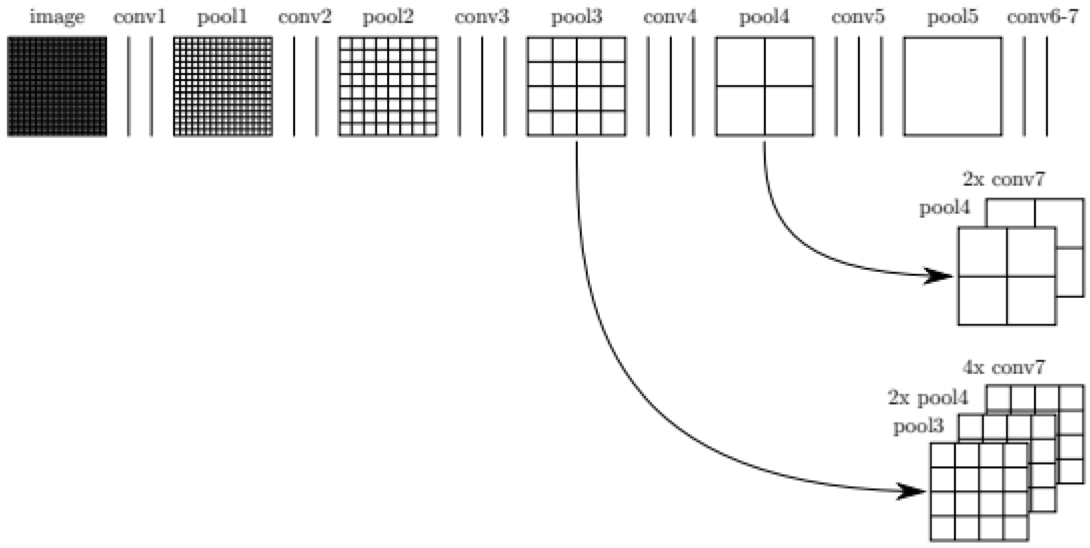

Figure 1: FCN-8 architecture [7a]. Image taken original paper titled 'Fully Convolutional Networks for Semantic Segmentation'

This architecture has following layers/features:

1. VGG-16 image classifier on the encoder end. The last fully connected layer was replaced by 1x1 convolution layer.
2. 1x1 convolutions on the decoder end. These layers preserve spatial dimensions of the input and at the same time can be used to tune the number of filters on the output. In this project, the number of output filters were 2 (one for road, one for non-road class).
3. Skip connection from layer 3 and layer 4 on encoder end to layers on decoder end.
4. Up sampling (or transpose convolution) on the decoder end in order to match the dimensions of layers used in skip connection and final inference image out of the network.

3.3 Python code for implementation

3.3.1 load_vgg()

This method is implemented from line 20 to line 45 in main.py. Code was responsible for performing the following:

1. Loading pre-trained VGG-16 (fully convolutional version) tensorflow model
2. Extraction of references to input image, layer 3, layer 4, dropout probability and layer 7 from the loaded graph

3.3.2 layers()

This method is implemented from line 49 to line 101 in main.py. Code was responsible for implementation of decoder network using the following:

1. Scaling of outputs of layer 3 and layer 4 for better learning of model as suggested in original implementation of FCN-8.
2. 1x1 convolution for layer 7 to scale down the number of output features from 4096 to 2 (classes for road and non-road) resulting in output conv_7_1x1
3. 2x up sampling of conv_7_1x1 to match dimensions of layer 4
4. 1x1 convolution for layer 4 to scale down the number of output features 2 (classes for road and non-road) resulting in output conv_4_1x1
5. Skip connection of conv_4_1x1 to conv_7_1x1 to obtain skip_layer_4
6. 2x up sampling of skip_layer_4 to match dimensions of layer 3
7. 1x1 convolution for layer 3 to scale down the number of output features 2 (classes for road and non-road) resulting in output conv_3_1x1
8. Skip connection of conv_3_1x1 to skip_layer_4 to obtain skip_layer_3
9. 8x up sampling of skip_layer_3 to obtain output layer upsample_8x which is the inference generated by the whole network

3.3.3 optimize()

This method is implemented from line 105 to line 134 in main.py. Code was responsible for calculating training loss and implementation of optimizer reduce the loss using following:

1. Softmax activation at for the inference output from the network.
2. Reduce mean algorithm to calculate training loss incurred by the network by using know labels.
3. Adam optimizer for training and reducing the training loss. Adam optimizer is an adaptive optimizer which decreases the learning rate as the number of epochs increases. This ensures that the model doesn't get stuck in local minima and network loss is decreased gradually. As it uses moving averages of the parameters (momentum) it enables Adam to use a larger effective step size, and the algorithm will converge to this step size without fine tuning.

3.3.4 train_nn()

This method is implemented from line 138 to line 161 in main.py. Code was responsible for training the network on tuned batches of data and for tuned number of epochs.

3.4 Tuning of Hyper parameters

Following were the tunable parameters for the network:

1. Dropout keep probability to avoid overfitting
2. Weight of L2 regularizer used in layers on decoder end
3. Batch size
4. Number of epochs
5. Learning rate

Network was trained for following combinations of hyper parameters before setting locking the final values:

Hyper parameters ⇨ / Iteration ⇩	Dropout Keep probability	L2 regularizer weight	Batch size	Number of epochs	Learning rate	Minimum training loss	Inference
1	0.5	0.0001	10	5	0.1	Nan	Learning rate too high
2	0.5	0.0001	10	5	0.01	Nan	Learning rate too high
3	0.5	0.0001	10	5	0.001	0.341	Loss has decreasing trend. Increase the number of epochs
4	0.5	0.0001	10	15	0.001	0.2986	Loss oscillates around 0.3. Reduce batch size of lower learning rate
5	0.5	0.0001	5	15	0.001	0.1357	Loss still oscillates. Lower learning rate
6	0.5	0.0001	5	15	0.0008	0.1203	Loss is decreased but not significantly. Increase epochs
7	0.5	0.0001	5	25	0.0008	0.0983	Loss is low. Try reducing dropout for better results
8	0.75	0.001	5	25	0.0008	0.08324	Loss is low

							but network performs worse on test images. Revert to original dropout
9	0.5	0.001	5	25	0.0008	0.09645	Final values

3.5 Model Inference on Test Images

The trained model was then applied on test images and inference images were obtained. Results are summarized below:

1. Model performed well for segmentation of pixels closer to camera than far away
2. Model performance was accurate in crowded places or where features such as cars, sign boards were prominent
3. Model did not perform well on images with shadow on road or poor with darker lighting conditions

Few inference images are shown below:

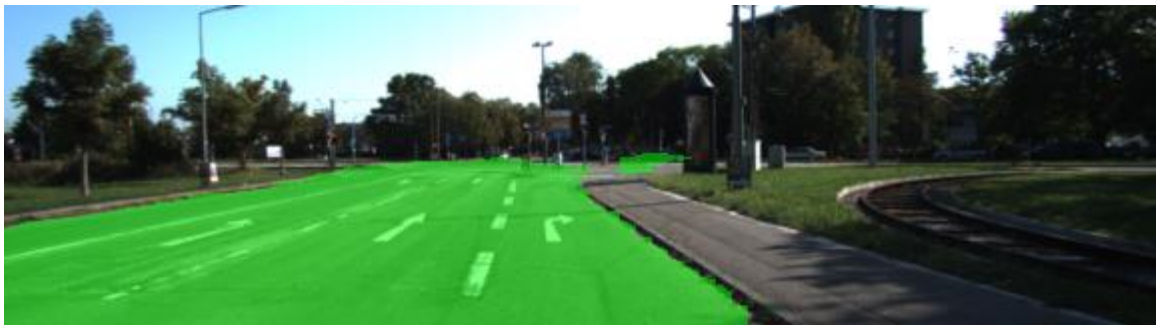

Figure 2: Inference on highway [7b]

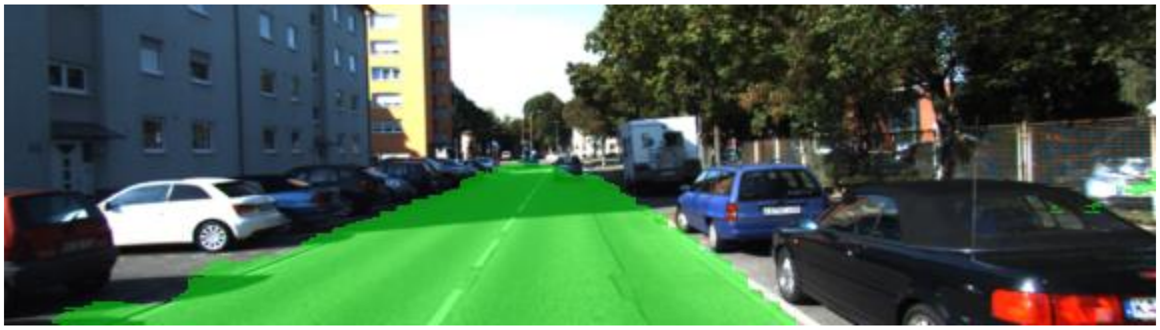

Figure 3: Inference on crowded feature rich scene [7c]

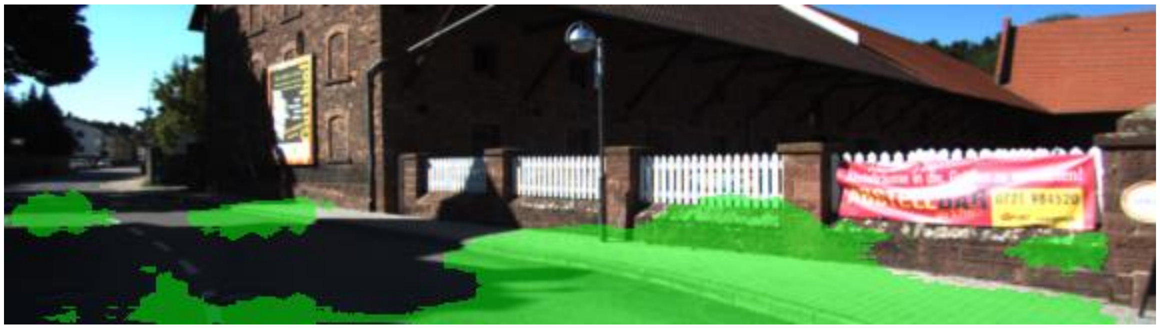

Figure 4: Inference of scene with shadows and darker lighting conditions [7d]

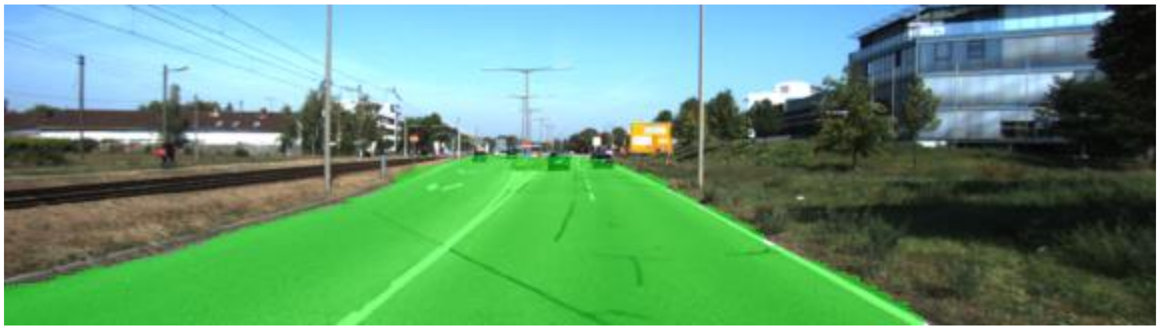

Figure 5: Inference on areas close to camera vs far away from camera [7e]

4. Reflection and Future Scope

The inference obtained from the trained model was accurate for more than 80% of the test images. Also, false positives occupied less than 20% of area in the image. However, the model performed poorly on images with dark lighting conditions. This can be corrected by using a combination of following techniques:

1. Pre-processing input images using mean subtraction and normalization of pixel values in the images
2. Pre-processing input images using PCA whitening which corrects the brightness and contrast range difference in the images
3. Creating contours from inference images to reject discontinuous false positives

5. References

1. [1] TensorFlow An open-source software library for Machine Intelligence
2. [2] KITTI city and highway roads data set
3. [3] Karen Simonyan, Andrew Zisserman. , 'Very Deep Convolutional Networks for Large-Scale Image Recognition, Conference paper at ICRL 2015
4. [4] Christian Szegedy et. al. 'Going Deeper with Convolutions'
5. [5] Kaiming He et al. 'Deep Residual Learning for Image Recognition'
6. [6] Jonathan Long et al. 'Fully Convolutional Networks for Semantic Segmentation'
7. [7] Image references:
    a. /image-resources/fcn-8-architecture.PNG
    b. /image-resources/inference-on-highway.png
    c. /image-resources/crowded-feature-rich-scene.png
    d. /image-resources/inference-shadows-dark-lighting.png
    e. /image-resources/inference-close-far-away.png