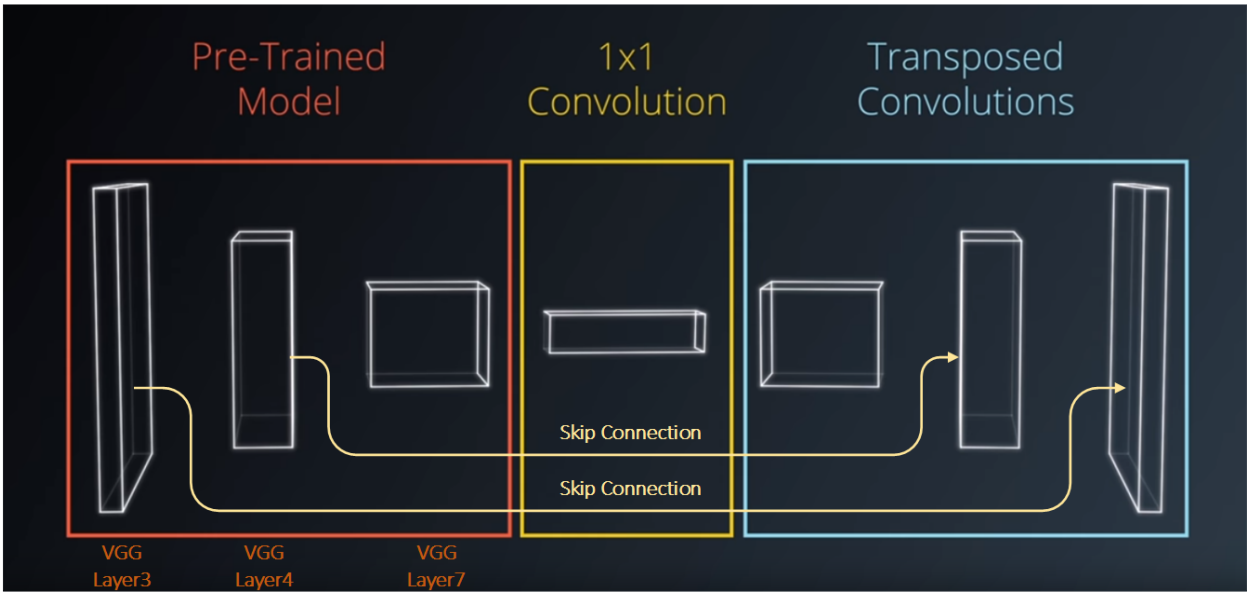
Carnd - term 3 - project 2 - semantic segmentation project

Overview

The goal of this project is to build and train a neural network that can identify the road of the vehicle it is travling on. The neural network is composed of: an encoder and a decoder. The encoder is basically a pre-trained VGG16 model followed by a 1x1 convlution layer, while the decoder contains the transposed layers to upsample the pixels of the object and the skip connection layers that retains the background information surround the object in the image.



Semantic Segmentation Neural Network Pipeline

Model is trained over 289 road images from Kitti Road dataset (http://www.cvlibs.net/datasets/kitti/eval_road.php) which also provides the corresponding ground true images that labels the road pixels vs background pixels

Images will be processed by the neural network in order to produce an image with a green mask on the road the vechicle is traveling on.

Project Repository

All resource are located in Udacity's project repository CarND-Semantic-Segmentatio (https://github.com/udacity/CarND-Semantic-Segmentation)

Project Submission

All modified code including results are committed to my personal github page carnd_t3_p2_semantic_segmentation_project (https://github.com/chriskcheung/carnd_t3_p2_semantic_segmentation_project)

Key Files

main.py

consists of key functions to support building, training and testing the nerual network pipeline: load_vgg(), layers(), optimize(), train_nn(), and run().

load_vgg() - downloads the pre-trained VGG16 neural net model and extracts its tensorflow graphs as well as its image layer, keep_prob layer, pooling layer 3, 4, and 7.

layers() - takes the extracted pooling layers from VGG16 model, connecting each following the Semantic Segmentation Nueral Network Pipeline diagram as shown above. Each layer in the decoder section is upsampled in order to bring the pixels of the object back to its original image width and length. We will go over the detail in the later section.

optimize() - uses cross_entrophy_loss function to calculate the loss between trained image and its ground true image. Adam optimizer is used to minimumize loss during training.

train_nn() - trains the neural network by getting batches of images and labals by calling get_batches_fn(batch_size) function from helper.py, and feeding them into train_op and entropy_loss.

run() - put together a processing pipeline with the functions above. After training a model, a set of testing images are feed through the semantic segmentation neural network and the results are saved into runs directory for inspection.

helper.py

maybe_download_pretrained_vgg() - detects if VGG16 pre-trained model is not previously downloaded, it will download and install it accordingly

gen_batch_function() - a generate function that creates batches of training images base on batch_size provides

gen_test_output() - generates test output using test images

save_inference_samples() - as its name says, it feeds the test images obtain from gen_test_output() and saves the result of the semantic segmentation

Project Setup

Due to the limitation of my PC laptop, I requested an AWS GPU instance to perform my semantic segmentation neural network training and testing. The project needs the a few more Tensorflow framework and Python packages to be installed than those mentioned in lecture. Road dataset comes from Kitti Road dataset (http://www.cvlibs.net/datasets/kitti/eval_road.php).

```
Python 3, TensorFlow-gpu, NumPy, SciPy, TQDM, Pillow
```

Details on how to run this project is documented in CarND-Semantic-Segmentatio (https://github.com/udacity/CarND-Semantic-Segmentation).

Implementation Challenge

VGG16 Pre-trained Model

To take advantage of the existing Pretrained Model, we use VGG16 Neural Network, which can be downloaded here (https://s3-us-west-1.amazonaws.com/udacity-selfdrivingcar/vgg.zip).

Before using VGG16, it is better to understand each of its pooling layers size, in order for skip connections to match the layers they are connecting. In this project, we use its vgg_layer_3, vgg_layer_4, and vgg_layer_7, and each has an output class size of 256, 512, and 4096, respectively.

The lecture never really talks about their width and length. This is actually dictated by the input_image tensor shape, which is based on the training image width and length.

```
    image_shape = (160, 576)
    input_image   = tf.placeholder(tf.float32, (None, image_shape[0], image_
shape[1], num_classes))
    correct_label = tf.placeholder(tf.float32, (None, image_shape[0], image_
shape[1], num_classes))
```

Semantic Segmentation Neural Network Pipeline Implementation

One thing worth pointing out is, there are multiple ways of constructing the pipeline. I tried two and did some comparison before finalizing to one approach. I basically applied 1 x 1 convulation on vgg_layer7_out to a 1 x 1 x num_classes layer, then upsample the resulting layer using transposed convulation with stride of 2 as to upsample the result by 2 times. I added the upsample layer with vgg_layer4_out as skip connection to retain background information from the original vgg_layer4_out. The same step from upsampling and skip connection was repeated with vgg_layer3_out instead.

The trick to match the output size of two layers during the additional of them for skip connection is to make sure the output of previous convolution layers matches the next one in the additional, by using the next vgg layer's shape[3] as the num_classes output during convolution.

Stride size controls how much a layer can be upsampled. Stride of 2 upsample the width and heigth of the pixel by 2 times. Same goes on as stride of 8 upsamples the pixel width and height by 8 times. This model is based on a series of upsample sequence of 2 times, 2 times, and 8 times at each VGG layers that we used.

```
    #taking the last layer of vgg16 to build a 1x1 fully convolutional laye
r,
    #VGG layer tensors with 4D size [batch_size, original_height, original_w
idth, num_classes]
    k_size = 1  # kernal_size
    stride = 1
    fconv8 = tf.layers.conv2d(vgg_layer7_out,
                              num_classes,
                              k_size,
                              stride,
                              padding='same',
                              kernel_regularizer=tf.contrib.layers.l2_regula
rizer(1e-3))

    #upsampling 2 times from ?xHxWx4096 to ?x2Hx2Wx512
    #as upsampling is dictated by stride, adjust stride to 2
    k_size = 4  # kernal_size
    stride = 2
    upscl9 = tf.layers.conv2d_transpose(fconv8,
                                        vgg_layer4_out.shape[3],
                                        k_size,
                                        stride,
                                        padding='same',
                                        kernel_regularizer=tf.contrib.layers
.l2_regularizer(1e-3))

    #skip connections by adding
    skip10 = tf.add(upscl9, vgg_layer4_out)

    #upsampling 2 times from ?x2Hx2Wx512 to ?x4Hx4Wx256
    k_size = 4  # kernal_size
    stride = 2
    upscl11 = tf.layers.conv2d_transpose(skip10,
                                         vgg_layer3_out.shape[3],
                                         k_size,
                                         stride,
                                         padding='same',
                                         kernel_regularizer=tf.contrib.layer
s.l2_regularizer(1e-3))

    #skip connections by adding
    skip12 = tf.add(upscl11, vgg_layer3_out)

    #upsampling 8 times from ?x4Hx4Wx256 to ?x32Hx32Wx2
    k_size = 16  # kernal_size
    stride = 8
    upscl13 = tf.layers.conv2d_transpose(skip12,
                                         num_classes,
                                         k_size,
                                         stride,
```

```
                                        padding='same',
                                        kernel_regularizer=tf.contrib.layer
    s.l2_regularizer(1e-3))
        return upscl13
```

Alternative pipeline implemenation turns all vgg layers output into output classes of 2 by applied 1 x 1 convolution. This keeps every layers of the pipeline to deal with the same output classes instead of needing to worry about using vgg_layer3_out.shape[3] and vgg_layer4_out.shape[3] at upsampling layers. This also works but the results show diffused edges of the green mask rather than solid edges.

Training

I started with a small epochs number, 5, and a small batch_size, 10, to get any idea of how the model behaves. The early trails found that the segmentation fell into the area of the road on the image. However, some output images showed the edges of the green mask was not fully extended to the edge of the road, while some output images showed the green mask sometimes covered areas that it shouldn't covered, like car, pedestrian walkway, etc.

Adjusting the batch_size, keep_prob, and epochs seems to be improve the trained model. Here are a few setting tried

Epochs	Batch Size	Keep Prob	Pipeline	Result	Images
5	10	0.5	Alternative	Diffusion at edge	
10	15	0.5	Alternative	Diffusion at edge	
20	16	0.5	Final	Teeth at edge	
30	16	0.5	Alternative	Diffusion at edge	
40	8	0.5	Final	Best	
40	16	0.5	Final	Overfitting	
50	16	0.5	Alternative	Overfitting	

Results

Most of the info was provided in the class material and the walktrough video. The results still not the best at the current implementation, but it can be improved with larger training dataset, as well as picking cleaning dataset, and updating batch_size. Below, I capture a few screenshots to illustrate how my Semantic Segmentation Neural Network performs over 300 images.


umm_000012.png
umm_000013.png
umm_000014.png
umm_000015.png
umm_000016.png
umm_000017.png
umm_000018.png
umm_000019.png
umm_000020.png
umm_000021.png
umm_000022.png
umm_000023.png
umm_000024.png
umm_000025.png
umm_000026.png
umm_000027.png
umm_000028.png
umm_000029.png
umm_000030.png
umm_000031.png
umm_000032.png
umm_000033.png
umm_000034.png
umm_000035.png
umm_000036.png
umm_000037.png
umm_000038.png
umm_000039.png
umm_000040.png
umm_000041.png
umm_000042.png
umm_000043.png
umm_000044.png
umm_000045.png
umm_000046.png
umm_000047.png
umm_000048.png
umm_000049.png
umm_000050.png
umm_000051.png
umm_000052.png
umm_000053.png
umm_000054.png
umm_000055.png
umm_000056.png
umm_000057.png
umm_000058.png
umm_000059.png
umm_000060.png
umm_000061.png
umm_000062.png
umm_000063.png
umm_000064.png
umm_000065.png
umm_000066.png
umm_000067.png
umm_000068.png
umm_000069.png
umm_000070.png
umm_000071.png
umm_000072.png
umm_000073.png
umm_000074.png
umm_000075.png
umm_000076.png
umm_000077.png
umm_000078.png
umm_000079.png
umm_000080.png
umm_000081.png
umm_000082.png
umm_000083.png
umm_000084.png
umm_000085.png
umm_000086.png
umm_000087.png
umm_000088.png
umm_000089.png
umm_000090.png
umm_000091.png
umm_000092.png
umm_000093.png
uu_000000.png
uu_000001.png
uu_000002.png
uu_000003.png
uu_000004.png
uu_000005.png
uu_000006.png
uu_000007.png
uu_000008.png
uu_000009.png
uu_000010.png
uu_000011.png
uu_000012.png
uu_000013.png
uu_000014.png
uu_000015.png
uu_000016.png
uu_000017.png
uu_000018.png
uu_000019.png
uu_000020.png
uu_000021.png
uu_000022.png
uu_000023.png
uu_000024.png
uu_000025.png
uu_000026.png
uu_000027.png
uu_000028.png
uu_000029.png
uu_000030.png
uu_000031.png
uu_000032.png
uu_000033.png
uu_000034.png
uu_000035.png
uu_000036.png
uu_000037.png
uu_000038.png
uu_000039.png
uu_000040.png
uu_000041.png
uu_000042.png
uu_000043.png
uu_000044.png
uu_000045.png
uu_000046.png
uu_000047.png
uu_000048.png
uu_000049.png
uu_000050.png
uu_000051.png
uu_000052.png
uu_000053.png
uu_000054.png
uu_000055.png
uu_000056.png
uu_000057.png
uu_000058.png
uu_000059.png
uu_000060.png
uu_000061.png
uu_000062.png
uu_000063.png
uu_000064.png
uu_000065.png
uu_000066.png
uu_000067.png
uu_000068.png
uu_000069.png
uu_000070.png
uu_000071.png
uu_000072.png
uu_000073.png
uu_000074.png
uu_000075.png
uu_000076.png
uu_000077.png
uu_000078.png
uu_000079.png


uu_000080.png


uu_000081.png


uu_000082.png


uu_000083.png


uu_000084.png


uu_000085.png


uu_000086.png


uu_000087.png


uu_000088.png


uu_000089.png


uu_000090.png


uu_000091.png


uu_000092.png


uu_000093.png


uu_000094.png


uu_000095.png


uu_000096.png


uu_000097.png


uu_000098.png


uu_000099.png