Project: Semantic Segmentation

Sean Pavlak

April 30 2018

1 Project

For this project I set out to better understand and hopefully implement semantic segmentation. This is a topic that is at the heart of autonomous vehicles, it also is a very interesting computer vision and deep learning problem.

Segmentation is essential for image analysis tasks. Semantic segmentation describes the process of associating each pixel of an image with a class label.

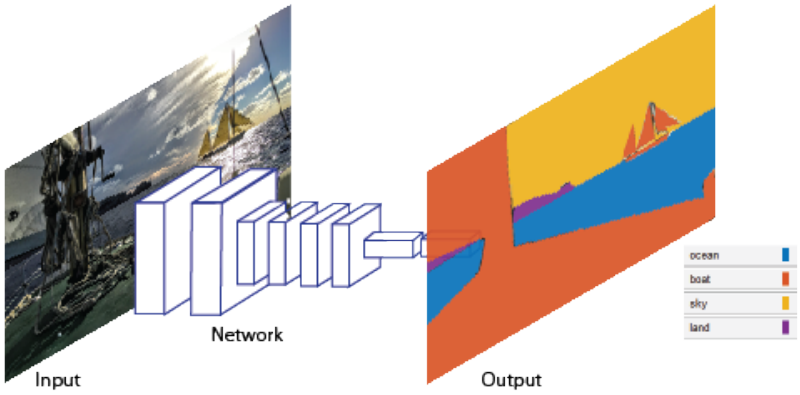


Figure 1: Demonstrating the process of semantic segmentation on an image.

As shown in figure 1, semantic segmentation begins with an image which is then processed by a trained network and outputs an color coded image. This image is then able to be deciphered and potentially yield more information than the original.

At this point in time deep learning is the chosen mechanism to perform semantic segmentation, but before deep learning took over,

people used approaches like TextonForest and Random Forest based classifiers. As with image classification, convolutional neural networks have had enormous success on segmentation problems.

One newer method of segmentation is Fully Convolutional Neural Networks. Fully convolutional indicates that the neural network is composed of convolutional layers without any fully-connected layers usually found at the end of the network. To semantic segment a image, a "fully convolutional" neural networks can take input of arbitrary size and produce correspondingly-sized output with efficient inference and learning, both learning and inference are performed whole-image-at-a-time by dense feedforward computation and backpropagation. In-network upsampling layers enable pixelwise prediction and learning in nets with subsampled pooling.

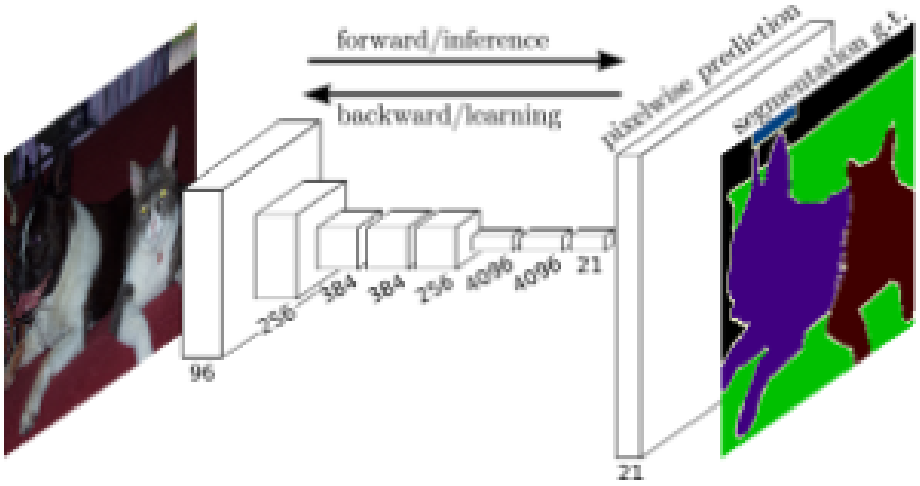


Figure 2: Demonstrating the process of fully convolutional neural network semantic segmentation on an image.

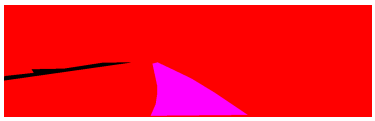
A general semantic segmentation architecture can be broadly thought of as an encoder network followed by a decoder network. The encoder is usually is a pre-trained classification network like VGG/ResNet followed by a decoder network. The decoder network/mechanism is mostly where these architectures differ. The task of the decoder is to semantically project the discriminating features (lower resolution) learned by the encoder onto the pixel space (higher resolution) to get a dense classification.

A convolutional encoder-decoder neural network is composed of a stack of encoders followed by a corresponding decoder stack which

feeds into a soft-max classification layer. The decoders help map low resolution feature maps at the output of the encoder stack to full input image size feature maps.

The dataset I am using is the KITTI road data. KITTI is a project of Karlsruhe Institute of Technology and Toyota Technological Institute at Chicago. The dataset includes three different categories of road; urban unmarked, urban marked, and urban multiple marked lanes. This data is also broken up into preset training and testing sets.

(a) Urban marked road training image from the KITTI dataset



(b) Colorized urban marked road training image from the KITTI dataset

Figure 3: (a) and (b) are the training image pair used to train the CNN model.

Using the images provided by KITTI and a CNN designed for semantic segmentation, I set out to train a model on this dataset.

1.1 Code Description

In order to perform semantic segmentation I defined a set of function what would load in my training data, setup a tensorflow model, and then use this defined model to generate a semantic segmentation image on the testing images proveded by the dataset.

```
def generate_batch_function(data_folder, image_shape):
    def get_batches_function(batch_size):
        image_paths = glob(os.path.join(data_folder, 'image_2',
        ↪  '*.png'))
        label_paths = {
            re.sub(r'_(lane|road)_', '_',
            ↪  os.path.basename(path)): path
            for path in glob(os.path.join(data_folder,
            ↪  'gt_image_2', '*_road_*.png'))}
        background_color = np.array([255, 0, 0])
        random.shuffle(image_paths)

```

```
        for i in range(0, len(image_paths), batch_size):
            images = []
            gt_images = []

            for image_file in image_paths[i:i+batch_size]:
                gt_image_file =
                ↪  label_paths[os.path.basename(image_file)]

                image =
                ↪  scipy.misc.imresize(scipy.misc.imread(image_file),
                ↪  image_shape)
                gt_image =
                ↪  scipy.misc.imresize(scipy.misc.imread(gt_image_file),
                ↪  image_shape)

                gt_bg = np.all(gt_image == background_color,
                ↪  axis=2)
                gt_bg = gt_bg.reshape(*gt_bg.shape, 1)
                gt_image = np.concatenate((gt_bg,
                ↪  np.invert(gt_bg)), axis=2)

                images.append(image)
                gt_images.append(gt_image)

            yield np.array(images), np.array(gt_images)
    return get_batches_function
```

The first functions generate_batch_function and get_batches_function simply generate two arrays containing the street images and their colored pairs, similar to the pair shown in figure 3.

```
def generate_test_output(sess, logits, keep_prob, image_pl,
↪  data_folder, image_shape):
    for image_file in glob(os.path.join(data_folder, 'image_2',
    ↪  '*.png')):
        image =
        ↪  scipy.misc.imresize(scipy.misc.imread(image_file),
        ↪  image_shape)

        im_softmax = sess.run([tf.nn.softmax(logits)],
        ↪  {keep_prob: 1.0, image_pl: [image]})
        im_softmax = im_softmax[0][:, 1].reshape(image_shape[0],
        ↪  image_shape[1])

```

```
        segmentation = (im_softmax > 0.5).reshape(image_shape[0],
        ↪  image_shape[1], 1)

        mask = np.dot(segmentation, np.array([[0, 255, 0, 127]]))
        mask = scipy.misc.toimage(mask, mode='RGBA')

        street_im = scipy.misc.toimage(image)
        street_im.paste(mask, box=None, mask=mask)

        yield os.path.basename(image_file), np.array(street_im)
```

The next function generate_test_output is of a similar nature. Rather than loading in training data, this function loads in the testing data. It also produces an array of images.

After the images are in memory, I defined a function, load_vgg, which then loads in the TensorFlow VGG architecture. This architecture is pre-trained to handle semantic segmentation. However, as shown in the results it still needs to be extensively trained in order to handle the task at hand.

```
def load_vgg(sess, vgg_path):
    model = tf.saved_model.loader.load(sess, ['vgg16'], vgg_path)

    graph = tf.get_default_graph()
    image_input = graph.get_tensor_by_name('image_input:0')
    keep_prob = graph.get_tensor_by_name('keep_prob:0')
    layer3 = graph.get_tensor_by_name('layer3_out:0')
    layer4 = graph.get_tensor_by_name('layer4_out:0')
    layer7 = graph.get_tensor_by_name('layer7_out:0')

    return image_input, keep_prob, layer3, layer4, layer7
```

This pre-trained model generates layers for the CNN that will train on the training data.

Once the pre-trained model is set and those layers are defined another function named layers takes those pre-trained layers and convolves them in order to generate a decoder layer. This layer will be used to semantically project the discriminating features learned by the encoding layers onto the pixel space.

```
def layers(vgg_layer3_out, vgg_layer4_out, vgg_layer7_out,
    num_classes = NUMBER_OF_CLASSES):
    layer3x = tf.layers.conv2d(inputs = vgg_layer3_out,
                               filters =  NUMBER_OF_CLASSES,
                               kernel_size = (1, 1),
                               strides = (1, 1),
                               name = 'layer3conv1x1')

    layer4x = tf.layers.conv2d(inputs = vgg_layer4_out,
                               filters =  NUMBER_OF_CLASSES,
                               kernel_size = (1, 1),
                               strides = (1, 1),
                               name = 'layer4conv1x1')

    layer7x = tf.layers.conv2d(inputs = vgg_layer7_out,
                               filters =  NUMBER_OF_CLASSES,
                               kernel_size = (1, 1),
                               strides = (1, 1),
                               name = 'layer7conv1x1')

    decoderlayer1 = tf.layers.conv2d_transpose(inputs = layer7x,
                                               filters =
                                               NUMBER_OF_CLASSES,
                                               kernel_size = (4,
                                               4),
                                               strides = (2, 2),
                                               padding = 'same',
                                               name =
                                               'decoderlayer1')
    decoderlayer2 = tf.add(decoderlayer1, layer4x, name =
    'decoderlayer2')

    decoderlayer3 = tf.layers.conv2d_transpose(inputs =
    decoderlayer2,
                                               filters =
                                               NUMBER_OF_CLASSES,
                                               kernel_size = (4,
                                               4),
                                               strides = (2, 2),
                                               padding = 'same',
                                               name =
                                               'decoderlayer3')

    decoderlayer4 = tf.add(decoderlayer3, layer3x, name =
    'decoderlayer4')
```

```
    decoderlayer_output = tf.layers.conv2d_transpose(inputs = decoderlayer4,
                                                     filters = NUMBER_OF_CLASSES,
                                                     kernel_size = (16, 16),
                                                     strides = (8, 8),
                                                     padding = 'same',
                                                     name = 'decoderlayer_output')

    return decoderlayer_output
```

After the layers of the CNN are defined, the optimizer must then be defined.

```
def optimize(nn_last_layer, correct_label, learning_rate, num_classes = NUMBER_OF_CLASSES):
    logits = tf.reshape(nn_last_layer, (-1, num_classes))
    class_labels = tf.reshape(correct_label, (-1, num_classes))

    cross_entropy = tf.nn.softmax_cross_entropy_with_logits_v2(logits = logits, labels = class_labels)
    cross_entropy_loss = tf.reduce_mean(cross_entropy)

    train_optimizer = tf.train.AdamOptimizer(learning_rate).minimize(cross_entropy_loss)

    return logits, train_optimizer, cross_entropy_loss
```

This function first, reshapes the 4D tensors to 2D. Then implements softmax cross entropy and optimizes using the adam algorithm. These properties of the model help find the weights/parameters that would yield correct pixel labels.

```
def train_nn(sess, epochs, batch_size, get_batches_function, train_optimizer, cross_entropy_loss, input_image,
correct_label, keep_prob, learning_rate):
```

```
    start = time.time()
    print('epoch: 0', '/', EPOCHS, 'training loss: N/A')

    for epoch in range(EPOCHS):
        losses, i = [], 0
        for images, labels in get_batches_function(BATCH_SIZE):
            i += 1
            feed = {input_image: images,
                   correct_label: labels,
                   keep_prob: DROPOUT,
                   learning_rate: LEARNING_RATE}

            _, partial_loss = sess.run([train_optimizer,
             ↪  cross_entropy_loss], feed_dict = feed)

            print('-> iteration: ', i, '/', NUMBER_OF_IMAGES,
             ↪  'partial loss: ', partial_loss)
            losses.append(partial_loss)

        training_loss = sum(losses) / len(losses)

        end = time.time()
        training_time = end - start

        print('epoch: ', epoch + 1, '/', EPOCHS, 'training loss:
         ↪  ', training_loss)
    print('training time: ', training_time)
```

Then I defined the train_nn function. Which iterates through each epoch and each training image pair in order to train the CNN over many separate cycles. This function also keeps track of how long each set of epochs took in order to train.

```
def save_inference_samples(runs_dir, data_dir, sess, image_shape,
 ↪  logits, keep_prob, input_image):
    output_dir = os.path.join(runs_dir, 'epochs
     ↪  {0}'.format(EPOCHS))

    if os.path.exists(output_dir):
        shutil.rmtree(output_dir)
    os.makedirs(output_dir)

    print('Training Finished. Saving test images to:
     ↪  {}'.format(output_dir))
```

```
    image_outputs = generate_test_output(sess, logits, keep_prob,
    ↪  input_image, os.path.join(data_dir, 'data_road/testing'),
    ↪  image_shape)

    for name, image in image_outputs:
        scipy.misc.imsave(os.path.join(output_dir, name), image)
```

Lastly, I defined a function that saves the semantic segmentation image for each test case. This allowed me to empirically determine how successful each set of epochs was.

Once all the functions were defined, I defined the paths and the learning parameters. Then training and testing was as simple as loading in the data and tensorflow model, training, and testing.

```
DATA_DIRECTORY = './data'
RUNS_DIRECTORY = './runs'
TRAINING_DATA_DIRECTORY ='./data/data_road/training'
NUMBER_OF_IMAGES =
↪  len(glob('./data/data_road/training/calib/*.*'))
VGG_PATH = './data/vgg'

NUMBER_OF_CLASSES = 2
IMAGE_SHAPE = (160, 576)

EPOCHS = 20
BATCH_SIZE = 1

LEARNING_RATE = 0.0001
DROPOUT = 0.75

correct_label = tf.placeholder(tf.float32, [None, IMAGE_SHAPE[0],
↪  IMAGE_SHAPE[1], NUMBER_OF_CLASSES])
learning_rate = tf.placeholder(tf.float32)
keep_prob = tf.placeholder(tf.float32)

get_batches_function =
↪  generate_batch_function(TRAINING_DATA_DIRECTORY, IMAGE_SHAPE)

with tf.Session() as session:
    image_input, keep_prob, layer3, layer4, layer7 =
    ↪  load_vgg(session, VGG_PATH)
    model_output = layers(layer3, layer4, layer7,
    ↪  NUMBER_OF_CLASSES)
```

```
25
26      logits, train_optimizer, cross_entropy_loss =
        ↪  optimize(model_output, correct_label, learning_rate,
        ↪  NUMBER_OF_CLASSES)
27
28      session.run([tf.global_variables_initializer(),
        ↪  tf.local_variables_initializer()])
29      train_nn(session, EPOCHS, BATCH_SIZE, get_batches_function,
        ↪  train_optimizer, cross_entropy_loss, image_input,
        ↪  correct_label, keep_prob, learning_rate)
30
31      save_inference_samples(RUNS_DIRECTORY, DATA_DIRECTORY,
        ↪  session, IMAGE_SHAPE, logits, keep_prob, image_input)
```

The very last step was calling save_inference_samples so that the test images that underwent semantic segmentation could be saved into a separate directory and reviewed.

1.2 Results

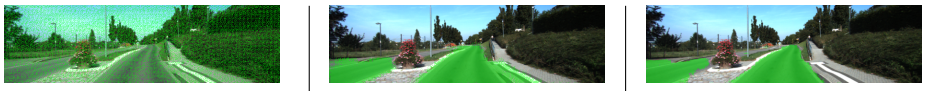
The final results of this project were more impressive than I originally expected. Semantic segmentation is significantly more accurate at defining objects in a view than most of the edge detectors we explored in this course.

The approach I took in order to evaluate this model was to generate three separate models, which differed only in their epochs for training. The first only went through a single epoch, and took a few minutes to train. The next was over 10 epochs, and took over an hour to train. The last went through 20 epochs, and took multiple hours to train.

If I were training on GPU rather than CPU and I wasn't running this code on a local machine I may have been able to train in significantly shorter times.

(a) 1 Epoch | (b) 10 Epochs | (c) 20 Epochs

Figure 4: (a),(b), and (c) are the generated semantic segmentation images on the urban unmarked road images.



(a) 1 Epoch (b) 10 Epochs (c) 20 Epochs

Figure 5: (a),(b), and (c) are the generated semantic segmentation images on the urban unmarked road images.

As shown in these images of urban unmarked road, it wasn't until the 10 epoch trial that some semantic segmentation began to take hold.

However, between the 10 and 20 epochs there does not appear to be very much improvement.

(a) 1 Epoch (b) 10 Epochs (c) 20 Epochs

Figure 6: (a),(b), and (c) are the generated semantic segmentation images on the urban marked road images.

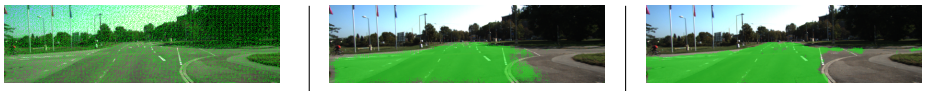
(a) 1 Epoch (b) 10 Epochs (c) 20 Epochs

Figure 7: (a),(b), and (c) are the generated semantic segmentation images on the urban marked road images.

When performed on the urban marked road images both the 1 epoch and 10 epochs cases fell short. Neither was able to accurately perform semantic segmentation on the lanes without extending past the bounds of the road.

The 20 epoch case was able to differentiate between road and non-road area well. This testing set still appears to give this model some difficulties shown in image c of figure 7. This image has a rough edge on the right side of the road along with errors caused by shadows on the road.

Potentially running a 30 epoch case may solve this, or tweaking the model may help.



(a) 1 Epoch (b) 10 Epochs (c) 20 Epochs

Figure 8: (a),(b), and (c) are the generated semantic segmentation images on the urban multiple marked lanes road images.

(a) 1 Epoch (b) 10 Epochs (c) 20 Epochs

Figure 9: (a),(b), and (c) are the generated semantic segmentation images on the urban multiple marked lanes road images.

Again semantic segmentation on this test set of urban multiple marked lanes images appears to only hold true for the 20 epochs case. This set does appear to be better suited for my model. Except when approaching turning lanes, but this may be corrected with more epochs or examples of turning lanes presented in the training data.

Overall all three cases for the 10 or 20 epochs performed very well. The only changes I would make would be to extend to a 30 epochs case and maybe define a function to compare the generated image with a hand labeled image to calculate an accuracy value for these images.

Unless the KITTI dataset has these labeled test images somewhere this may not be feasible as there are 300 images that would need labeled. Which wouldn't be time efficient. Regardless, without labels this project's accuracy can only be estimated.