Machine Learning Engineer Nanodegree

Capstone Project

Andreas Braun

June 3rd, 2021

Semantic Segmentation with AWS SageMaker

I. Definition

I.1 Project Overview

Machine Learning (ML) or in this case Deep Learning (DL) as special case of ML are very magical things, where you're not programming an algorithm to calculate a result, but you're programming an algorithm to take a result and to show you patterns within the result. In this case you teach the system to visualize objects and put these objects into categories or classes (for different cases of object detection on Amazon Web Services (AWS) SageMaker as ML-tool please see the following post: https://medium.com/@kolungade.s/object-detection-image-classification-and-semantic-segmentation-using-aws-sagemaker-e1f768c8f57d). As I am working within the Automotive Industry for several years, I am really interested in putting the ideas of ML and DL into the Automotive Industry. Self-Driving cars on the streets, Self-Working robots within a manufacturing process, or Self Organized Logistics within a production plant are not future thinking anymore, but possible solutions for the Automotive Industry nowadays. Semantic segmentation is the basis for all these tasks. Before you learn how to fly, you'll need to learn, how to walk. Actually, I start walking now and am already excited, which next steps I'll make.

I.2 Problem Statement

According to tensorflow.org "the task of image segmentation is to train a neural network to output a pixel-wise mask of the image. This helps in understanding the image at a much lower level" (https://www.tensorflow.org/tutorials/images/segmentation).

Basically, semantic segmentation is needed to get an output image, where every pixel of the input image is assigned to a class, e.g. a car, bus, or a pedestrian. It's used to identify objects on a photo, video, etc. Semantic segmentation could let cars know the location of another car or person on the road for Autonomous Driving purposes or let robots know the location of other parts for Augmented Reality Applications in order to avoid collisions in a manufacturing environment.

In order to identify objects in a traffic environment I am using the Kitti Dataset for Lane/Road Detection Evaluation (http://www.cvlibs.net/datasets/kitti/eval_road.php). This allows me to perform a binary segmentation of road-/not-road-objects

This data then will be used to train a Neural Network (NN) on photos of streets including objects like cars or pedestrians with the help of Amazon Web Services (AWS) SageMaker. The NN then will identify, if the objects on new photos of a traffic environment belongs to a road or not (it divides the picture into different segments). After the identification the road-segments will be highlighted and test-pictures with highlighted roads will be saved into the runs-folder of the AWS SageMaker Jupyter Notebook-instance.

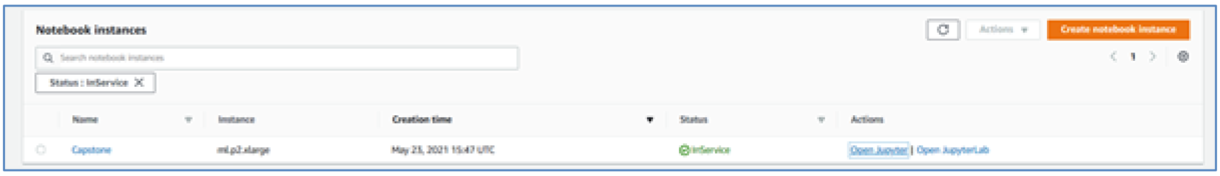
I.3 Metrics

The main metrics here used for checking the quality of the inference are the loss per batch as summation of the errors made for each example in training sets. Another metric that meets the eye is the visibility of the segments in the test-pictures. Can we clearly see the distinction between the road- and the non-road-elements? Feel free to look on your own.

II. Analysis

II.1 AWS setup

Before we can explore and analyse our data, we need to setup our environment for the data. In a first step I created a notebook-instance on AWS SageMaker that has some GPU-capacities and downloaded the data to the notebook (Check on: https://towardsdatascience.com/choosing-the-right-gpu-for-deep-learning-on-aws-d69c157d8c86 for GPU-usage on AWS).



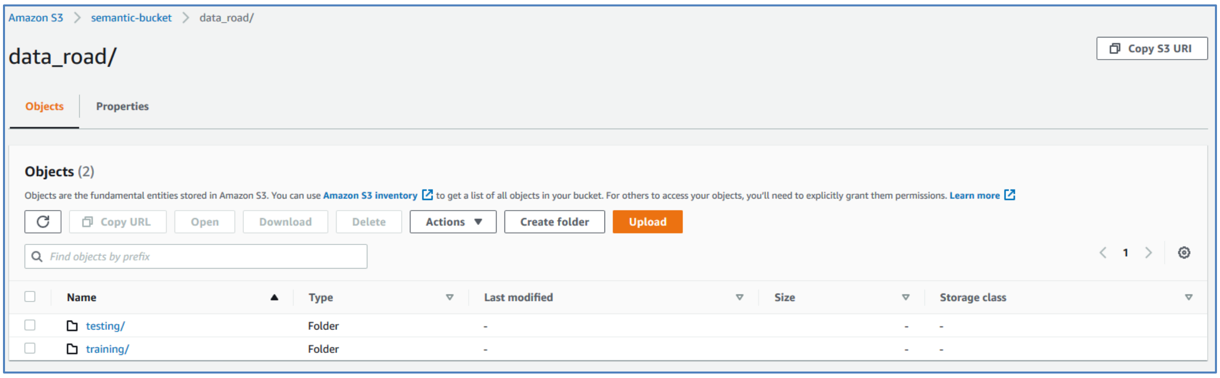
AWS SageMaker notebook-instance

“To identify the contents of an image at the pixel level, use an Amazon SageMaker Ground Truth semantic segmentation labelling task. When assigned a semantic segmentation labelling job, workers classify pixels in the image into a set of predefined labels or classes” (https://docs.aws.amazon.com/sagemaker/latest/dg/sms-semantic-segmentation.html). Lucky me, the labels have been already provided within the Kitti dataset.

```
P0: 7.215377000000e+02 0.000000000000e+00 6.095593000000e+02 0.000000000000e+00 0.000000000000e+00 7.215377000000e+02 1.728540000000e+02
0.000000000000e+00 0.000000000000e+00 0.000000000000e+00 1.000000000000e+00 0.000000000000e+00
P1: 7.215377000000e+02 0.000000000000e+00 6.095593000000e+02 -3.875744000000e+02 0.000000000000e+00 7.215377000000e+02
1.728540000000e+02 0.000000000000e+00 0.000000000000e+00 0.000000000000e+00 1.000000000000e+00 0.000000000000e+00
P2: 7.215377000000e+02 0.000000000000e+00 6.095593000000e+02 4.485728000000e+01 0.000000000000e+00 7.215377000000e+02 1.728540000000e+02
2.163791000000e-01 0.000000000000e+00 0.000000000000e+00 1.000000000000e+00 2.745884000000e-03
P3: 7.215377000000e+02 0.000000000000e+00 6.095593000000e+02 -3.395242000000e+02 0.000000000000e+00 7.215377000000e+02
1.728540000000e+02 2.199936000000e+00 0.000000000000e+00 0.000000000000e+00 1.000000000000e+00 2.729905000000e-03
R0_rect: 9.999239000000e-01 9.837760000000e-03 -7.445048000000e-03 -9.869795000000e-03 9.999421000000e-01 -4.278459000000e-03
7.402527000000e-03 4.351614000000e-03 9.999631000000e-01
Tr_velo_to_cam: 7.533745000000e-03 -9.999714000000e-01 -6.166020000000e-04 -4.069766000000e-03 1.480249000000e-02 7.280733000000e-04
-9.998902000000e-01 -7.631618000000e-02 9.998621000000e-01 7.523790000000e-03 1.480755000000e-02 -2.717806000000e-01
Tr_imu_to_velo: 9.999976000000e-01 7.553071000000e-04 -2.035826000000e-03 -8.086759000000e-01 -7.854027000000e-04 9.998898000000e-01
-1.482298000000e-02 3.195559000000e-01 2.024406000000e-03 1.482454000000e-02 9.998881000000e-01 -7.997231000000e-01
Tr_cam_to_road: 9.999570839814e-01 -5.508724949246e-03 -7.452906591504e-03 9.610489538319e-03 5.425697507328e-03 9.999234779341e-01
-1.111504746388e-02 -1.597134401910e+00 7.513565886504e-03 1.107413060494e-02 9.999104059534e-01 2.788606298060e-01
```

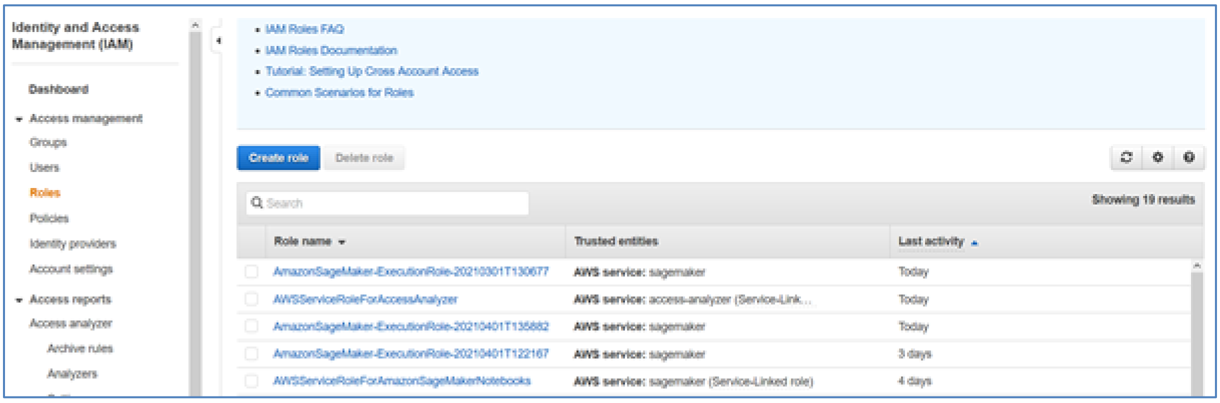
Labeling example

After setting up a notebook-instance the data can be uploaded to the AWS S3-bucket for storage purposes. S3 stands for Simple Storage Service and according to AWS it is „an object storage service that offers industry-leading scalability, data availability, security, and performance" (https://aws.amazon.com/s3).



AWS S3-bucket

In a next step you should make sure to give Amazon SageMaker access to the S3-bucket, where you stored the files used for the Semantic Segmentation (and vice versa).



S3 — Identity and Access Management (IAM)

After all is set up, you can start with the SageMaker-magic, upload the files from S3 to your SageMaker notebook, and check your files.

```
In [ ]: # Load data directly from the S3 bucket
        # Make sure to give your notebook instance access to S3 and vice versa by setting up IAM correctly
        # Also check on https://blog.codecentric.de/en/2020/01/aws-sagemaker-data-handling/
        # Kitti dataset downloaded from kaggle.com


        from botocore.exceptions import ClientError
        s3 = boto3.resource('s3', region_name='eu-central-1')
        bucket = s3.Bucket('semantic-bucket')
        for my_bucket_object in bucket.objects.all():
            key = my_bucket_object.key
            print(key)
            if not os.path.exists(os.path.dirname(key)):
                os.makedirs(os.path.dirname(key))
            try:
                bucket.download_file(key, key)
            except ClientError as e:
                if e.response['Error']['Code'] == "404":
                    print("No object with this key.")
                else:
                    raise
```

Upload Kitti-dataset from S3 to SageMaker-notebook

Furthermore, make sure to install the necessary packages (Python 3, TensorFlow, NumPy, SciPy) and use a TensorFlow P36 Kernel, when using the Jupyter Notebook on SageMaker.

II.2 Data Exploration

I order to identify objects in a traffic environment I am using the Kitti Dataset downloaded from Kaggle (https://www.kaggle.com/kerneler/starter-kitti-object-detection-162ff5be-6).

The Kitti Dataset was collected by driving around the mid-size city of Karlsruhe (Germany), equipped with two high-resolution cameras, and taking photos in rural areas as well as on highways. “Up to 15 cars and 30 pedestrians are visible per image” (www.cvlibs.net/datasets/kitti/). Basically, it consists out of 7518 photos in the test dataset and 7481 photos in the training dataset. The training dataset is labelled indicating objects like cars or pedestrians per pixel. Hence, it’s a case of supervised learning.

However, for the purpose of this task (binary segmentation), I am using the part from the Kitti dataset prepared for this task – the road and lane estimation benchmark (see: http://www.cvlibs.net/datasets/kitti/eval_road.php). It consists of 289 training and 290 test images and three different categories of road scenes:

The road and lane estimation benchmark “contains three different categories of road scenes: * uu - urban unmarked (98/100) * um - urban marked (95/96) * umm - urban multiple marked lanes (96/94) * urban - combination of the three above Ground truth has been generated by manual annotation for the images and is available for two different road terrain types: road - the road area, i.e, the composition of all lanes, and lane - the ego-lane, i.e., the lane the vehicle is currently driving on (only available for category "um"). Ground truth is provided for training images only” (https://paperswithcode.com/dataset/kitti-road).

```
# show data statistics

import os
from sklearn.datasets import load_files
from keras.utils import np_utils
import numpy as np
from glob import glob

# load train, test datasets
data_dir = './data'
kitti_dataset_path = os.path.join(data_dir, 'data_road')
train_files = glob(os.path.join(kitti_dataset_path, 'training/image_2/*.png'))
label_files = glob(os.path.join(kitti_dataset_path, 'training/gt_image_2/*_road_*.png'))
test_files = glob(os.path.join(kitti_dataset_path, 'testing/image_2/*.png'))


# print statistics about the dataset

print('There are %d training images.' % len(train_files))

print('There are %d test images.'% len(test_files))
print('There are %d total images.'%(len(train_files)+ len(test_files)))

print()
print('There are %d image labels.'%len(label_files))

There are 289 training images.
There are 290 test images.
There are 579 total images.

There are 289 image labels.
Using TensorFlow backend.
```

Road and lane estimation benchmark – overview over dataset

In order to understand the data more briefly, I would like to give an overview of my file structure on my AWS SageMaker notebook:

File structure on AWS SageMaker Notebook-instance

The data folder will be explained more detailed in the next section.
In the runs folder the final pictures will be saved, where the road-segmentation has been performed on.

The main folder is the main part of this work and will be explained more detailed during this report.
The helper.py-file can be found on Udacity's Github page (https://github.com/udacity/CarND-Semantic-Segmentation/blob/master/helper.py) and is useful for several supporting actions. It will be explained more detailed during this report, when needed.
The same goes for the project_test.py-file (https://github.com/udacity/CarND-Semantic-Segmentation/blob/master/project_tests.py).

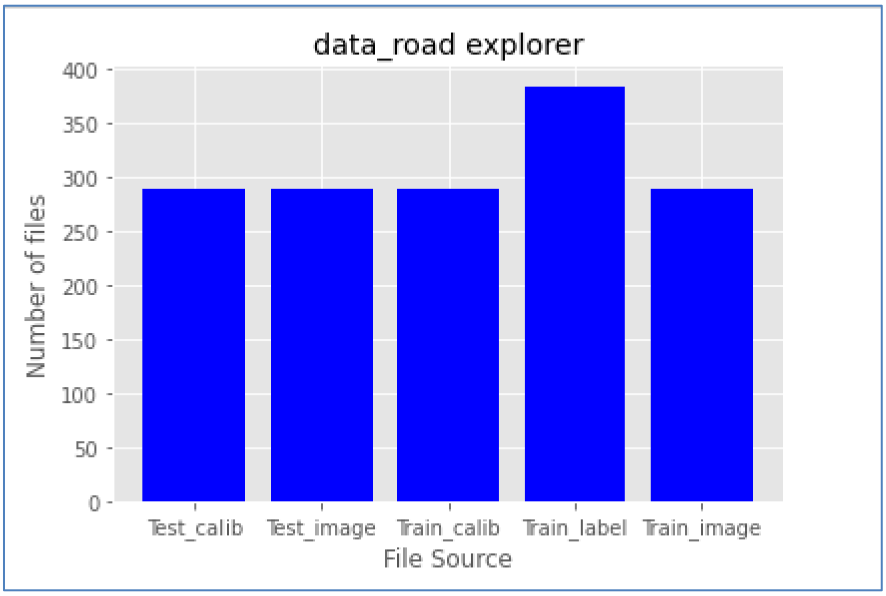
data
- data_road
  - testing
    - calib
      - um_000000.txt
      - um_000001.txt
      - um_000002.txt
      - (...)
    - image_2
      - um_000000.png
      - um_000001.png
      - um_000002.png
      - (...)
  - training
    - calib
      - um_000000.txt
      - um_000001.txt
      - um_000002.txt
      - (...)
    - gt_image_2
      - um_lane_000000.png
      - um_lane_000001.png
      - um_lane_000002.png
      - (...)
    - image_2
      - um_000000.png
      - um_000001.png
      - um_000002.png
      - (...)
- vgg
  - variables
  - saved_model.pb

File structure of data folder on AWS SageMaker Notebook Instance

The data folder basically is divided into the data_road- and the vgg-folder

- data_road: training and testing subfolder. The content will be explained more detailed in the exploratory-visualisation-section.
- vgg: I used a technique called Transfer Learning with the VGG16-pretrained-model (more precisely I used encoding-decoding as techniques). Transfer learning is a machine learning method where a model developed for a task is reused as the starting point for a model on a second task, especially for Computer Vision or Natural Language Processing tasks (in this case the VGG16-model is used as starting point for the training- and testing-steps). The VGG16-model will be explained more detailed in the algorithms-and-techniques-section.

II.3 Exploratory Visualization



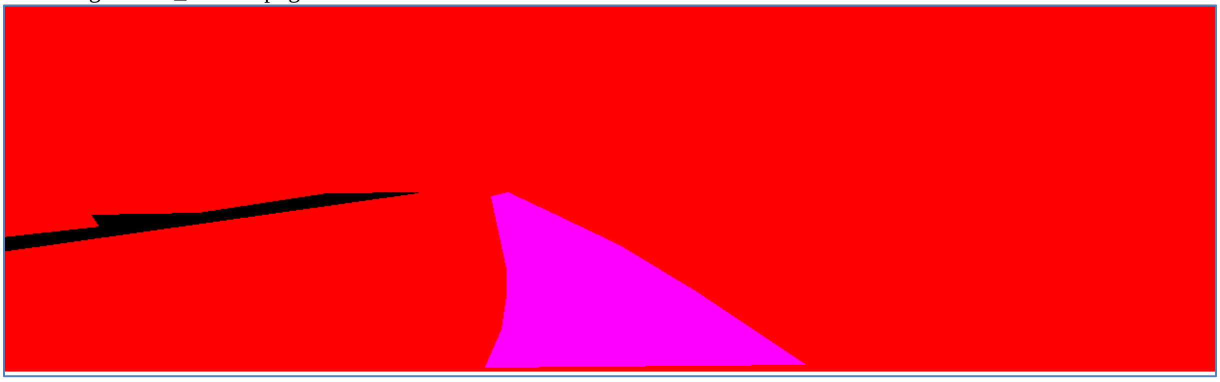
Structure of tests and training datasets

The data_road-folder consists of two testing and three training directories. There are 290 test-calib-txt-files, 290 test-images as png-file, 289 train-calib-txt-files, 384 train-label-files as png, and 289 train-images as png-file.

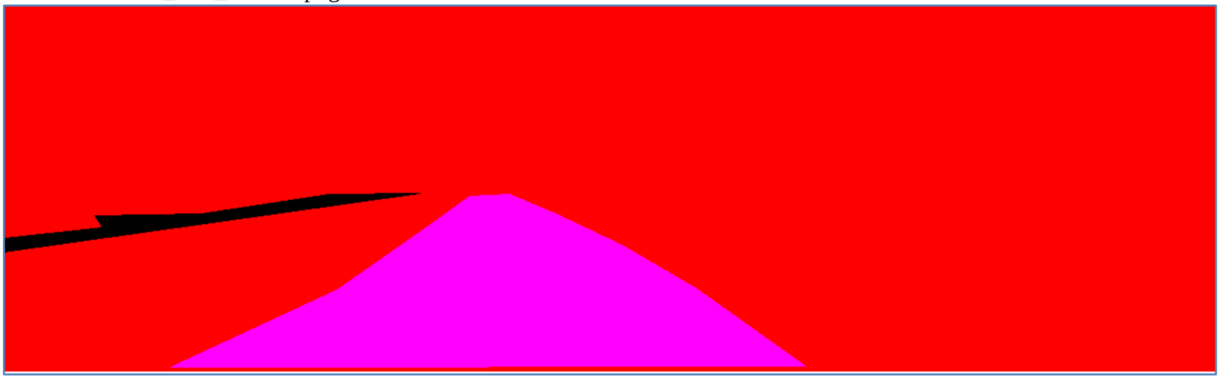
The images are images of traffic situations as shown in section III.2, the calib-files contain information about the precise position of the right and left cameras and their optical characteristics.

The train-label-files are labelled files differentiating between road and lane. See the following example-pictures:

train-image of um_000000.png



train-label of um_lane_000000.png



train-label of um_road_000000.png

```
P0: 7.215377000000e+02 0.000000000000e+00 6.095593000000e+02 0.000000000000e+00 0.000000000000e+00 7.215377000000e+02 1.728540000000e+02 0.000000000000e+00 0.000000000000e+00 0.000000000000e+00 1.000000000000e+00 0.000000000000e+00
P1: 7.215377000000e+02 0.000000000000e+00 6.095593000000e+02 -3.875744000000e+02 0.000000000000e+00 7.215377000000e+02 1.728540000000e+02 0.000000000000e+00 0.000000000000e+00 0.000000000000e+00 1.000000000000e+00 0.000000000000e+00
P2: 7.215377000000e+02 0.000000000000e+00 6.095593000000e+02 4.485728000000e+01 0.000000000000e+00 7.215377000000e+02 1.728540000000e+02 2.163791000000e-01 0.000000000000e+00 0.000000000000e+00 1.000000000000e+00 2.745884000000e-03
P3: 7.215377000000e+02 0.000000000000e+00 6.095593000000e+02 -3.395242000000e+02 0.000000000000e+00 7.215377000000e+02 1.728540000000e+02 2.199936000000e+00 0.000000000000e+00 0.000000000000e+00 1.000000000000e+00 2.729905000000e-03
R0_rect: 9.999239000000e-01 9.837760000000e-03 -7.445048000000e-03 -9.869795000000e-03 9.999421000000e-01 -4.278459000000e-03 7.402527000000e-03 4.351614000000e-03 9.999631000000e-01
Tr_velo_to_cam: 7.533745000000e-03 -9.999714000000e-01 -6.166020000000e-04 -4.069766000000e-03 1.480249000000e-02 7.280733000000e-04 -9.998902000000e-01 -7.631618000000e-02 9.998621000000e-01 7.523790000000e-03 1.480755000000e-02 -2.717806000000e-01
Tr_imu_to_velo: 9.999976000000e-01 7.553071000000e-04 -2.035826000000e-03 -8.086759000000e-01 -7.854027000000e-04 9.998898000000e-01 -1.482298000000e-02 3.195559000000e-01 2.024406000000e-03 1.482454000000e-02 9.998881000000e-01 -7.997231000000e-01
Tr_cam_to_road: 9.999570839814e-01 -5.508724949246e-03 -7.452906591504e-03 9.610489538319e-03 5.425697507328e-03 9.999234779341e-01 -1.111504746388e-02 -1.597134401910e+00 7.513565886504e-03 1.107413060494e-02 9.999104059534e-01 2.788606298060e-01
```

train-calib um_000000.txt

Hence, you have precisely labelled files distinguishing between road and lane per pixel as well as giving the precise position of other road user (in this case a bicycle or velo, as shown in the calib-txt-file above).

That means the NN will process the training data in the following way: With the detailed pixel-wise information regarding what's a road and what's, a lane, and what's not it learns to distinguish, how a road looks like or not. Then it will perform a batch-wise inference with new data and highlight the road-elements. The result then will be saved in the runs-folder.

```
# Save inference data using helper.save_inference_samples
print("Save inference samples..")
helper.save_inference_samples(runs_dir, data_dir, sess, image_shape, logits, keep_prob, input_image)
```

Saving inference data

II.4 Algorithms and Techniques

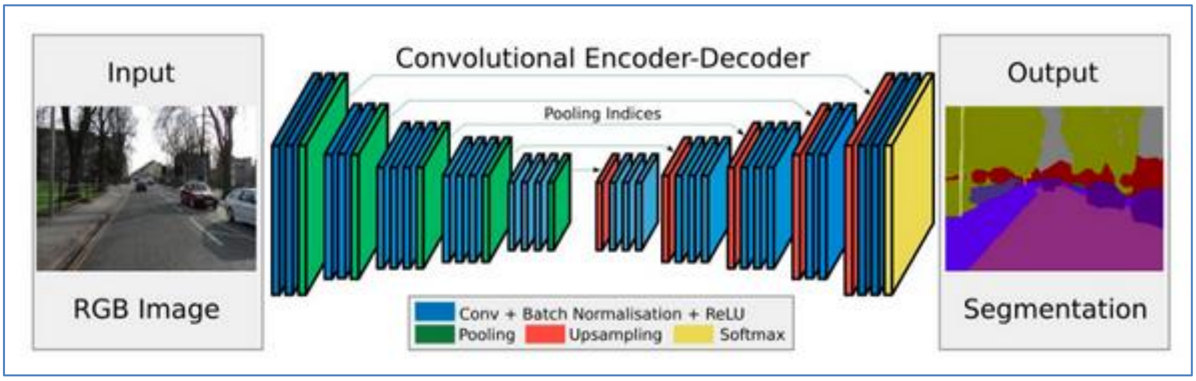
Answering the questions regarding algorithms and techniques used in this project is linked to the different process steps I made during this project and, hence, I present these algorithms and techniques while showing the process flow of the Session:

```
# Run tensorflow session
with tf.Session() as sess:
    # Path to vgg model
    vgg_path = os.path.join(data_dir, 'vgg')

    # Create function to get batches
    get_batches_fn = helper.gen_batch_function(os.path.join(data_dir, 'data_road/training'), image_shape)

    # Load pretrained VGG Model into TensorFlow and extract layers
    print("Loading VGG model as encoder..")
    input_image, keep_prob, layer3_out, layer4_out, layer7_out = load_vgg(sess, vgg_path)

    # Create our FCN model
    print("Creating our decoder part on top..")
    layer_output = layers(layer3_out, layer4_out, layer7_out, num_classes)

    # Build the TensorFlow loss and optimizer operations
    print("Create loss and optimizer..")
    correct_label = tf.placeholder(tf.float32, [None, image_shape[0], image_shape[1], num_classes])
    logits, train_op, cross_entropy_loss = optimize(layer_output, correct_label, learning_rate, num_classes)

    # Train our model using the train_nn function
    print("Train network..")
    train_nn(sess, epochs, batch_size, get_batches_fn, train_op, cross_entropy_loss,
             input_image, correct_label, keep_prob, learning_rate)

    # Save inference data using helper.save_inference_samples
    print("Save inference samples..")
    helper.save_inference_samples(runs_dir, data_dir, sess, image_shape, logits, keep_prob, input_image)
```

Project steps

In a first step of this Session, I build a vgg-path and later loaded the VGG16-model as so-called encoder building the basis for further processing. According to Jason Brownlee “the pre-trained model (..) can be integrated directly into a new neural network model. In this usage, (…) the weights may be updated during the training of the new model, perhaps with a lower learning rate, allowing the pre-trained model to act like a weight initialization scheme when training the new model.” (https://machinelearningmastery.com/how-to-use-transfer-learning-when-developing-convolutional-neural-network-models). And this is actually the functionality of VGG16 we would like to use during the training of the NN.

VGG16 is a convolutional neural network model or architecture. The image is passed through a stack of convolutional (Conv, see bottom) layers, where the filters were used with a very small receptive field (see also on https://neurohive.io/en/popular-networks/vgg16). It consists of 13 convolutional layers, 2 fully connected layers, and 1SoftMax classifier.



https://stackoverflow.com/questions/63465734/how-to-get-the-encoder-from-a-trained-vgg16-network

The first part of convolution and pooling is the part we call encoding, the part of convolution, upsampling and the usage of Softmax we call decoding.

```
In [19]: print(vgg16.summary())

_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_1 (InputLayer)         (None, 224, 224, 3)       0
_________________________________________________________________
block1_conv1 (Conv2D)        (None, 224, 224, 64)      1792
_________________________________________________________________
block1_conv2 (Conv2D)        (None, 224, 224, 64)      36928
_________________________________________________________________
block1_pool (MaxPooling2D)   (None, 112, 112, 64)      0
_________________________________________________________________
block2_conv1 (Conv2D)        (None, 112, 112, 128)     73856
_________________________________________________________________
block2_conv2 (Conv2D)        (None, 112, 112, 128)     147584
_________________________________________________________________
block2_pool (MaxPooling2D)   (None, 56, 56, 128)       0
_________________________________________________________________
block3_conv1 (Conv2D)        (None, 56, 56, 256)       295168
_________________________________________________________________
block3_conv2 (Conv2D)        (None, 56, 56, 256)       590080
_________________________________________________________________
block3_conv3 (Conv2D)        (None, 56, 56, 256)       590080
_________________________________________________________________
block3_pool (MaxPooling2D)   (None, 28, 28, 256)       0
_________________________________________________________________
block4_conv1 (Conv2D)        (None, 28, 28, 512)       1180160
_________________________________________________________________
block4_conv2 (Conv2D)        (None, 28, 28, 512)       2359808
_________________________________________________________________
block4_conv3 (Conv2D)        (None, 28, 28, 512)       2359808
_________________________________________________________________
block4_pool (MaxPooling2D)   (None, 14, 14, 512)       0
_________________________________________________________________
block5_conv1 (Conv2D)        (None, 14, 14, 512)       2359808
_________________________________________________________________
block5_conv2 (Conv2D)        (None, 14, 14, 512)       2359808
_________________________________________________________________
block5_conv3 (Conv2D)        (None, 14, 14, 512)       2359808
_________________________________________________________________
block5_pool (MaxPooling2D)   (None, 7, 7, 512)         0
_________________________________________________________________
flatten (Flatten)            (None, 25088)             0
_________________________________________________________________
fc1 (Dense)                  (None, 4096)              102764544
_________________________________________________________________
fc2 (Dense)                  (None, 4096)              16781312
_________________________________________________________________
predictions (Dense)          (None, 1000)              4097000
=================================================================
Total params: 138,357,544
Trainable params: 138,357,544
Non-trainable params: 0
_________________________________________________________________
None
```

VGG16-model overview

After setting up the VGG16 as encoder we create the layers of the NN with VGG16 as basis.

```
In [18]: # Create the layers for a FCN and using the vgg layers for skip layers
def layers(vgg_layer3_out, vgg_layer4_out, vgg_layer7_out, num_classes):
    # Define our kernel initializer and regularizer
    start = tf.truncated_normal_initializer(stddev = 0.01)
    reg = tf.contrib.layers.l2_regularizer(1e-3)

    # Do 1x1 convolutions on layer 3, 4 and 7 with L2 regularizer for the weights
    conv_layer3 = tf.layers.conv2d(vgg_layer3_out, num_classes, 1, padding='same',
                               kernel_initializer=start, kernel_regularizer=reg)
    conv_layer4 = tf.layers.conv2d(vgg_layer4_out, num_classes, 1, padding='same',
                               kernel_initializer=start, kernel_regularizer=reg)
    conv_layer7 = tf.layers.conv2d(vgg_layer7_out, num_classes, 1, padding='same',
                               kernel_initializer=start, kernel_regularizer=reg)

    # Do our first transposed convolution from layer 7
    deconv_1 = tf.layers.conv2d_transpose(conv_layer7, num_classes, 4, 2, padding='same',
                                      kernel_initializer=start, kernel_regularizer=reg)

    # Add the first skip connection from layer 4
    skip_1 = tf.add(deconv_1, conv_layer4)

    # Do our second transposed convolution on that result
    deconv_2 = tf.layers.conv2d_transpose(skip_1, num_classes, 4, 2, padding='same',
                                      kernel_initializer=start, kernel_regularizer=reg)

    # Add the second skip connection from layer 3
    skip_2 = tf.add(deconv_2, conv_layer3)

    # Do our third and last transposed convolution to match input image size
    deconv_3 = tf.layers.conv2d_transpose(skip_2, num_classes, 16, 8, padding='same',
                                      kernel_initializer=start, kernel_regularizer=reg)
    return deconv_3

tests.test_layers(layers)

WARNING:tensorflow:From <ipython-input-18-0615545ab307>:17: conv2d_transpose (from tensorflow.python.layers.convolutional)
is deprecated and will be removed in a future version.
Instructions for updating:
Use `tf.keras.layers.Conv2DTranspose` instead.
WARNING:tensorflow:From /home/ec2-user/SageMaker/project_tests.py:42: The name tf.assert_rank is deprecated. Please use tf.
compat.v1.assert_rank instead.

Tests Passed
```

Creating conv layer

In the next step we optimize our NN by reshaping our output labels to match the size, using the loss/cost-function to optimize the model during training by minimizing the loss function by minimizing the summation of the errors.

```
# Build the TensorFLow loss and optimizer operations
def optimize(nn_last_layer, correct_label, learning_rate, num_classes):
    # Logits is a 2D tensor where each row represents a pixel and each column a class
    logits = tf.reshape(nn_last_layer, (-1, num_classes))

    # Those are our output labels, reshaped to match size
    labels = tf.reshape(correct_label, (-1, num_classes))

    # We use standard cross-entropy-loss as our loss function
    cross_entropy_loss = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(logits=logits, labels=labels))

    # For the optimizer, we use Adam as it is a good general choice
    train_op = tf.train.AdamOptimizer(learning_rate).minimize(cross_entropy_loss)
    return logits, train_op, cross_entropy_loss
tests.test_optimize(optimize)
```

Optimization

Finally, everything comes together in the train_nn_function, where we build batches of the training data per epoch using VGG16 as encoder and defining the second part of the FCN.

The goal of the FCN is to assign each pixel to the right class.

```python
# Train neural network and print out loss

def train_nn(sess, epochs, batch_size, get_batches_fn, train_op, cross_entropy_loss, input_image,
             correct_label, keep_prob, learning_rate):

    # Init our global variables
    sess.run(tf.global_variables_initializer())

    # Go through all epochs
    for epoch in range(epochs):
        # Print out epoch
        print("Epoch {}".format(epoch + 1), "/ {} ..".format(epochs))

        # Go through all batches
        batch = 1
        for image, label in get_batches_fn(batch_size):
            # Print out batch number and raise it
            print("Batch {} ..".format(batch))
            batch = batch + 1

            # Train our model and get loss
            _, loss = sess.run([train_op, cross_entropy_loss],
                               feed_dict={input_image: image, correct_label: label,
                                          keep_prob: 0.8, learning_rate: 1e-4})

        # Print loss for each epoch
        print("Epoch {}".format(epoch + 1), " loss: {:.4f}".format(loss))
tests.test_train_nn(train_nn)
```

```
Tests Passed
```

Training of Neural Network

The helper function provided on Udacity's GitHub-page (https://github.com/udacity/CarND-Semantic-Segmentation/blob/master/helper.py) has a function called get_batches_fn linking the training-labels with the images performing a for loop for batch-wise creation of image-label-combinations (actually a Python dictionary will be created).

```python
def gen_batch_function(data_folder, image_shape):
    """
    Generate function to create batches of training data
    :param data_folder: Path to folder that contains all the datasets
    :param image_shape: Tuple - Shape of image
    :return:
    """
    def get_batches_fn(batch_size):
        """
        Create batches of training data
        :param batch_size: Batch Size
        :return: Batches of training data
        """
        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 batch_i in range(0, len(image_paths), batch_size):
            images = []
            gt_images = []
            for image_file in image_paths[batch_i:batch_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_fn
```

get_batches_function on helper.py

As you can see it's a batchwise training, where you bring the training-images as well as training-labels together and perform inference on the test data, assigning the pictures per pixel, belonging either to a road-element or not.

II.5 Benchmark

Our final benchmark is the loss per batch calculated in the main-function through the train_nn-function.

```
# We use standard cross-entropy-loss as our loss function
cross_entropy_loss = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(logits=logits, labels=labels))
```

Calculating loss with the help of TensorFlow

```
    # Train our model and get loss
    _, loss = sess.run([train_op, cross_entropy_loss],
                       feed_dict={input_image: image, correct_label: label,
                                  keep_prob: 0.8, learning_rate: 1e-4})

# Print loss for each epoch
print("Epoch {}".format(epoch + 1), " loss: {:.4f}".format(loss))
```

Calculation loss per epoch

The loss will be calculated and printed out per epoch within the different batches (see also IV.1 Model Evaluation and Validation)

III. Methodology

III.1 Data Preprocessing

In order to see, if you are on the right track, download the following files as help from the Udacity-Github page:

- helper.py (https://github.com/udacity/CarND-Semantic-Segmentation/blob/master/helper.py)
- project_test.py (https://github.com/udacity/CarND-Semantic-Segmentation/blob/master/project_tests.py)

The main preprocessing-steps were the following:

- Setting up AWS appropriately
- Data structure for appropriate file usage
- Usage of supporting function (incl. location where to find the functions) and VGG16
- Setting up the main-functions, the process flow and the supporting functions

III.2 Implementation

That means the NN will process the training data in the following way: With the detailed pixel-wise information regarding what's a road and what's a lane, and what's not.

Sample picture from the training dataset

Sample pictures from the test-dataset before and after inference

III.3 Refinement

Initially I had poor inference results and it was not clear, where the problems lie. Unfortunately, I did not save these pictures. The problem basically was due to the fact, that I didn't set up the link between labels and files appropriately.

```
feed_dict={input_image: image, correct_label: label,
           keep_prob: 0.8, learning_rate: 1e-4})
```

Dictonary-setup during training of NN

It took me a while to understand, that I get an empty dictionary during training of the NN, because the correct_label-element was not correctly assigned to the right picture. As I mentioned in the beginning: the file structure is one key element of this task.

IV. Results

IV.1 Model Evaluation and Validation

Our main evaluation criteria of seeing how our model is developing is to check the loss per epoch-development. Basically, loss is the summation of the errors made for each example in training sets.

```
Batch 286 ..
Batch 287 ..
Batch 288 ..
Batch 289 ..
Epoch 9  loss: 0.0200
Epoch 10 / 10 ..
Batch 1 ..
Batch 2 ..
Batch 3 ..
```

Lowest loss in Epoch 9

```
Batch 288 ..
Batch 289 ..
Epoch 10  loss: 0.0880
```

Highest loss in Epoch 10

In our training data our loss is varying between 2 % in Epoch9 as lowest loss and 8.8 % as highest loss in Epoch 10, which is basically not too bad.

As pictures are sometimes louder than words, see the final results from my runs-fuolder:

As you can see the model is able to distinguish between road and non-road elements.

IV.2 Justification

After adjusting the number of classes, image shape, as well as the hyper-paramaters a few times I found my ideal combination. It’s especially disturbing, because with every run on AWS you spent time for the calculation as well as money.

However, working with computers means to try out things and working due trial-and-error. Hence, I found my configuration in the end, that served my needs.

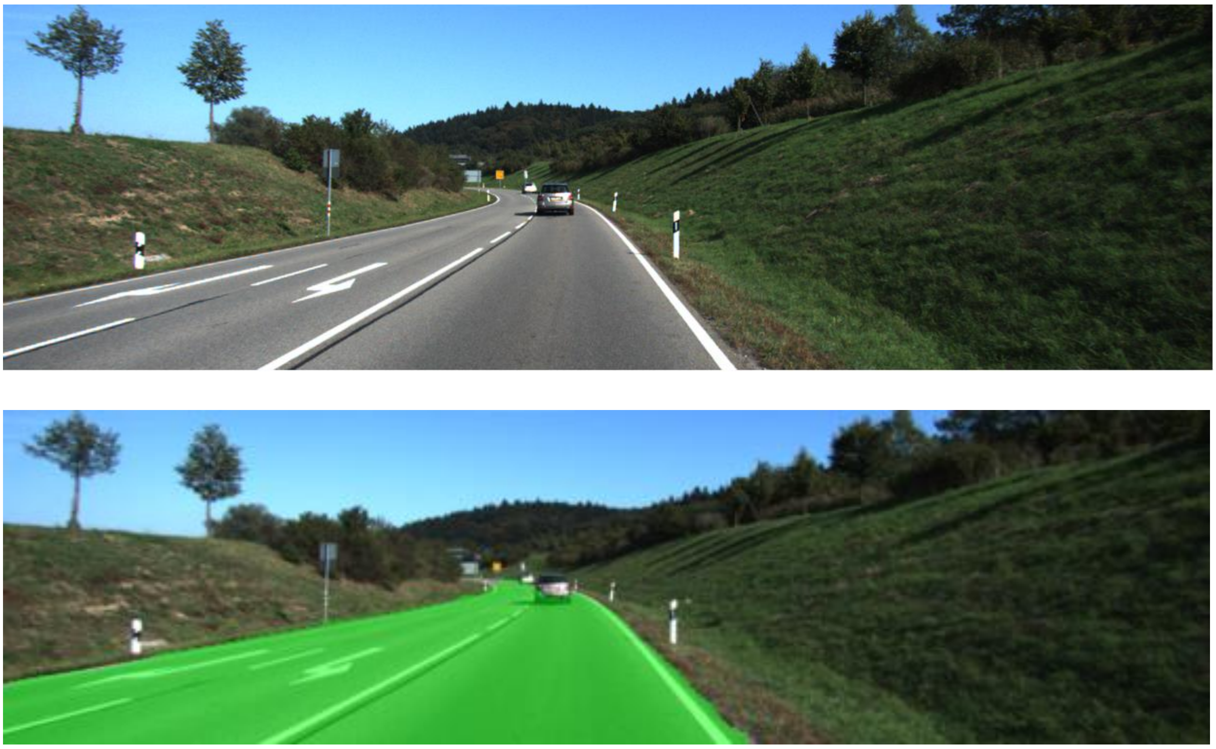
```
# Configuration
num_classes = 2
image_shape = (160, 576)
data_dir = os.path.abspath('data')
runs_dir = os.path.abspath('runs')

# Eventually download vgg model
helper.maybe_download_pretrained_vgg(data_dir)

# Hyper-Parameter
batch_size = 1
epochs = 10
learning_rate = tf.constant(1e-4)
```

Configuration

V. Conclusion

V.1 Free-Form Visualization

um00034.png



umm_000031.png

uu_000031

As you can see the NN is working quite well, however, there are still a few questions I would like to discuss in the next section.

V.2 Reflection

Semantic Segmentation is a good way to make computers visualize objects in multiple environments. Due to models like VGG16 it is possible to train NN without much data (in many cases of object classification with NN you need thousands of training images). However, before we can use this segmentation results in the real world for Autonomous Driving-purposes or Self-working-robots, you would need some further lidar-information for distance tracking and you should be sure, how to handle the exceptions. As you can see in the above picture, the Neural Network performs quite well with the street, but cannot really handle the occurring shadows. A human brain can tell that the shadowed part of the road is still a part of the road. However, our NN is not highlighting these parts of the road.

It is still a long way to go before our cars drive on their own and we don’t need humans for producing cars anymore, because the computer does not reach the kind of flexibility a human has. However, with semantic segmentation we have a first way teaching computers, how to visualize objects.

V.3 Improvement

As next course I really would like to choose a course about Computer Vision, because not only it is interesting and intellectually challenging. It is also very useful.

The first improvement here that comes to my mind is a semantic segmentation, where you really highlight every object on a street and not only performing a binary segmentation (road/not-road).

The next step the is to use lidar-information for distance tracking and 3D-segementation.

With these kinds of things your cars could drive automatically and more save than you and your robots would work faster as well as harder as you could (who is able to lift a sidepanel of a car?).

Still for me there is a lot to learn and I made another step for continuous improvement.