Technical Note on Image Segmentation

In this write up we are going to tell our journey in building image segmentation algorithms and their deployment on Microsoft Azure Services. During this quest, we encountered many troubles that we took great pleasure in solving.

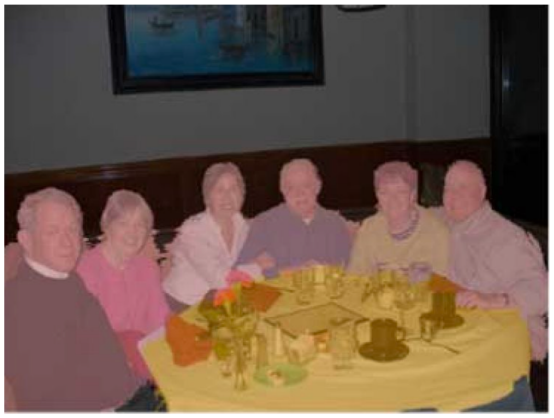
Along the way, we are going first to make an overview of image segmentation algorithms, explain the state of the art in the Deep Learning world in image segmentation for self driving car, and then go over on the way we prepared images in the data step; the training step with the use of colab pro and then Azure Services; the prediction or how we built a little pipeline from raw input images to output masked images; the training and deployment on Azure; and finally the use of Flask API for a deployment of a Web Application on top of Microsoft Azure Services.

Here below our guidelines:

1. An overview image segmentation algorithms
2. U-Net and Pre Trained Model for feature extractions
3. Data Preparation
4. Image Data Segment Generation
5. Data augmentation
6. Training and Prediction with U-Net
7. Training and Prediction with PreTrained Model
8. Comparison of U-Net and FCN8-VGG16
9. Azure Training and Deployment
10. FLask API
11. Azure Web Application

1. An overview image segmentation algorithms

At the start, the YOLO model was the state of the art in object detection for autonomous driving cars. It helps in detecting objects and plotting bounding boxes around them. And then thanks to Kaggle and the tremendous research in the area, we are using image segmentation models for autonomous driving car. This type of model was originally made for cancer detection. Nowadays image segmentation models are among the main types of models used in self-driving car algorithms. We have two types of image segmentation algorithm, semantic segmentation and instance segmentation. The first does the segmentation by classifying groups of similar objects and the second assigns a label to every object in an image. In this project,we are going to focus on semantic segmentation, assigning labels on groups of similar objects in an image.



Semantic Segmentation

Instance Segmentation

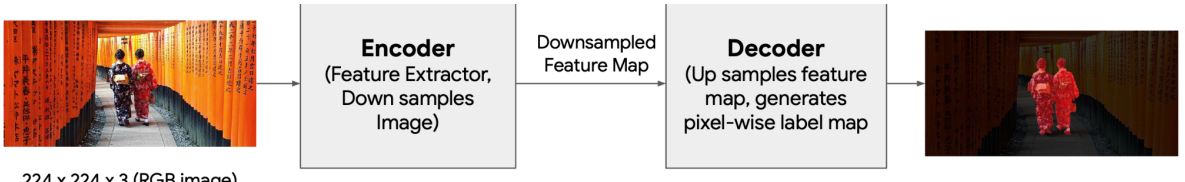
The motivation of image segmentation is that Instead of just predicting bounding boxes and detecting objects, how about predicting every pixel in an image?

In our case we used two types of algorithms of image semantic segmentation: the U-Net model and a mix of a Fully Connected Neural Networks with a pretrained part (VGG16).

2. The U-Net Model and the Pretrained Model for feature extractions

In this part we are going to explain the two models, their architecture, the way they take input images, the training and the prediction.

fig: Image Segmentation Basic Architecture.



224 x 224 x 3 (RGB image)

224 x 224 x n (Pixel map)
n = number of classes

As shown above, image segmentation type models are composed of an encoder part and a decoder part. The encoder part is aimed at aggregating low level features to high level features. It is generally a bunch of convolution layers without the fully connected part. The

decoder is generally a fully connected layer. It is aimed at Up Sampling images to original size to generate pixel mask

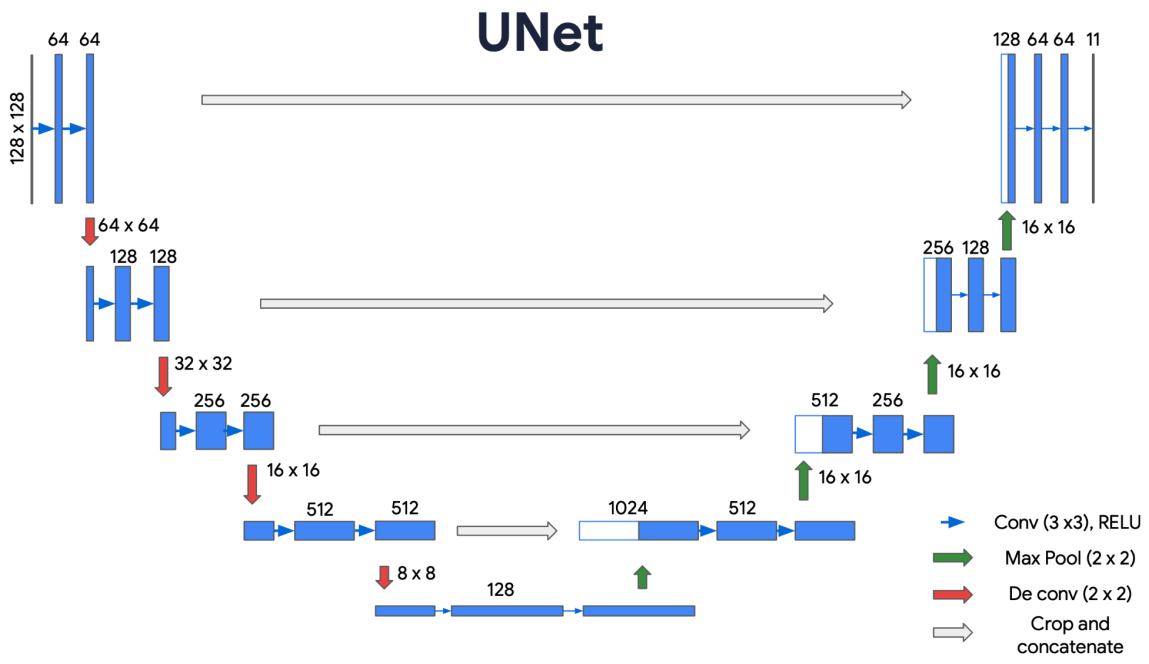
2.1. U-Net model

U-Net, named for its U-shape, was originally created in 2015 for tumor detection, but in the years since has become a very popular choice for other semantic segmentation tasks.

U-Net builds on a previous architecture called the Fully Convolutional Network, or FCN, which replaces the dense layers found in a typical CNN with a transposed convolution layer that up-samples the feature map back to the size of the original input image, while preserving the spatial information. This is necessary because the dense layers destroy spatial information (the "*where*" of the image), which is an essential part of image segmentation tasks. An added bonus of using transpose convolutions is that the input size no longer needs to be fixed, as it does when dense layers are used.

Unfortunately, the final feature layer of the FCN suffers from information loss due to downsampling too much. It then becomes difficult to upsample after so much information has been lost, causing an output that looks rough.

U-Net improves on the FCN, using a somewhat similar design, but differing in some important ways. Instead of one transposed convolution at the end of the network, it uses a matching number of convolutions for downsampling the input image to a feature map, and transposed convolutions for upsampling those maps back up to the original input image size. It also adds skip connections, to retain information that would otherwise become lost during encoding. Skip connections send information to every upsampling layer in the decoder from the corresponding downsampling layer in the encoder, capturing finer information while also keeping computation low. These help prevent information loss, as well as model overfitting.



Contracting path (Encoder containing downsampling steps):

Images are first fed through several convolutional layers which reduce height and width, while growing the number of channels.

The contracting path follows a regular CNN architecture, with convolutional layers, their activations, and pooling layers to downsample the image and extract its features. In detail, it consists of the repeated application of two 3 x 3 unpadded convolutions, each followed by a rectified linear unit (ReLU) and a 2 x 2 max pooling operation with stride 2 for downsampling. At each downsampling step, the number of feature channels is doubled.

Crop function: This step crops the image from the contracting path and concatenates it to the current image on the expanding path to create a skip connection.

Expanding path (Decoder containing upsampling steps):

The expanding path performs the opposite operation of the contracting path, growing the image back to its original size, while shrinking the channels gradually.

In detail, each step in the expanding path up-samples the feature map, followed by a 2 x 2 convolution (the transposed convolution). This transposed convolution halves the number of feature channels, while growing the height and width of the image.

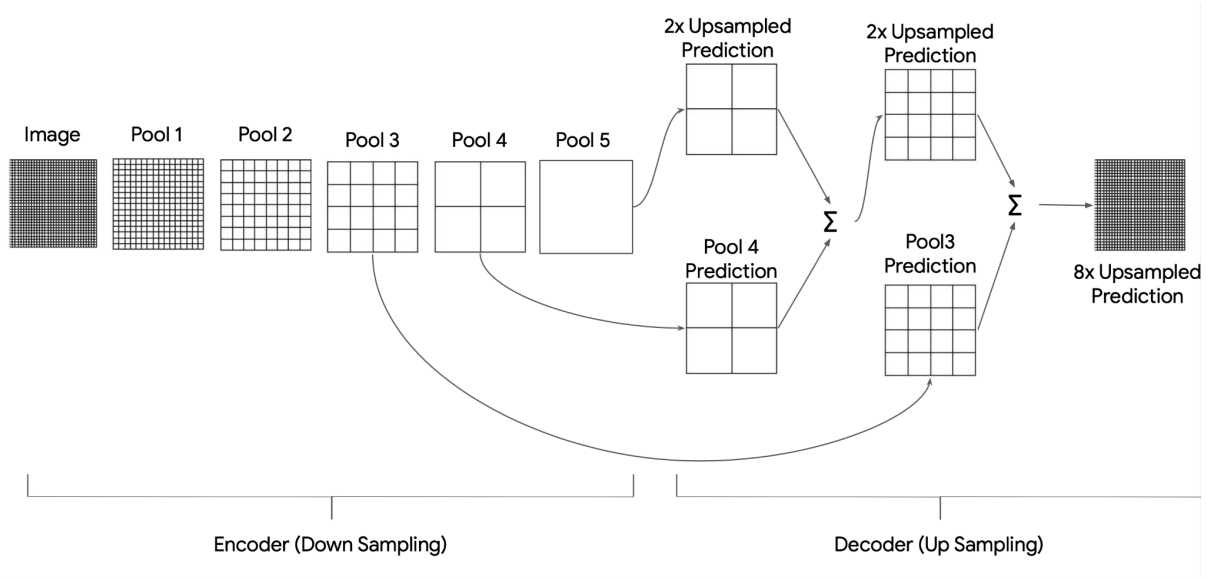
Next is a concatenation with the correspondingly cropped feature map from the contracting path, and two 3 x 3 convolutions, each followed by a ReLU. You need to perform cropping to handle the loss of border pixels in every convolution.

Final Feature Mapping Block: In the final layer, a 1x1 convolution is used to map each 64-component feature vector to the desired number of classes. The channel dimensions from the previous layer correspond to the number of filters used, so when you use 1x1 convolutions, you can transform that dimension by choosing an appropriate number of 1x1 filters. When this idea is applied to the last layer, you can reduce the channel dimensions to have one layer per class.

The U-Net network has 23 convolutional layers in total.

2.2. FCN VGG16

By contrast with the U-Net model algorithm, mixing a CNN feature extraction type model and Fully connected layers is a regular way to insert more flexibility of building a U-Net type model.
The schema below depicts the way pooling processes layers shrinks images between every convolution block in the VGG16 and how it is then up-sampled in the FCN8 part of the model architecture.



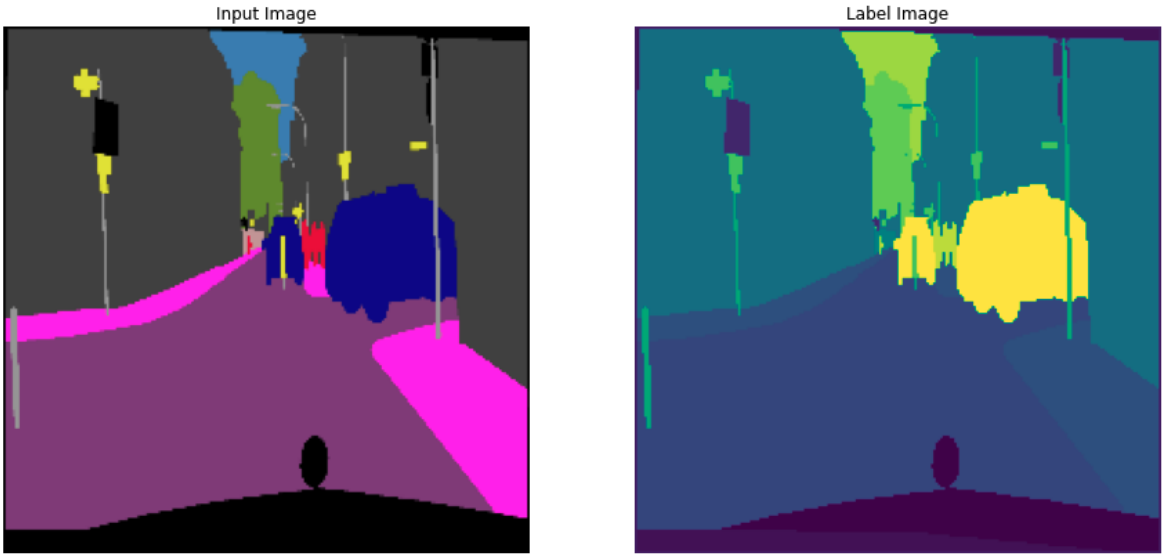
The encoder is made of the VGG16 blocks. To explain it short, the VGG16 is a Convolutional Neural Network that consists of 5 blocks with an increasing number of filters at each stage. The FCN decoding part is made mainly by successive up-samplings and summations feature maps. At the last layer, we have a regular Dense layer to classify 8 classes with naturally a softmax activation function.

3. Data Preparation

When we first started working with image segmentation we met masked or annotated images. In our case here, input samples images are masked too. There has been an image pre-treatment beforehand. Those images have four channels. The question was how to deal with them? How to read them? What are predictor sample images? But thanks to the literature out there and the help of our mentor, we know now that annotated images with only one channel are the ones that we are going to use as our label variables.

In the process of our data preprocessing we found that the fourth channel of iput images is empty, hence we got rid-off it. We did so, because the pretrained VGG16 contrainted us to give an input channel of 3. It is pretrained that way so, we adapt data input images. For uniformity sake we kept the same input image shape for the U-Net model too.

fig: input image and label image



In our project we are recommended to use 8 segments instead of 34 for prediction masks. We will treat that in the next part. But as illustration, we can see on the above image that we have 34 pixels type labels, the road and the pavement have different labels.

4. **Image Data Segment Generation**

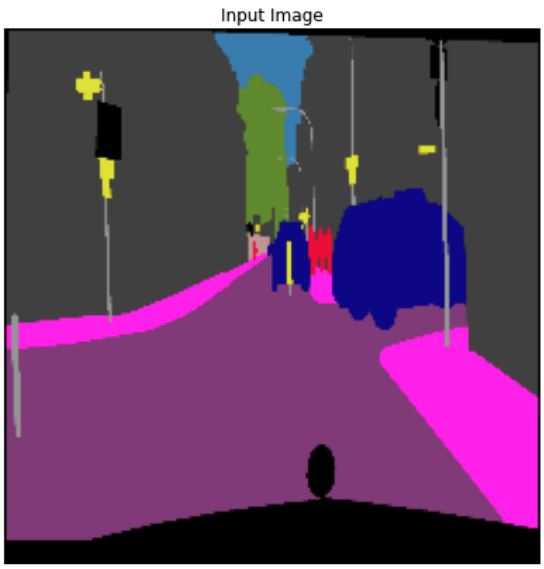
With 34 classes, 34 different number type pixels labelled from 0 to 33. And we are given a category correspondence dictionary, we are going to build a function that will help us to regroup classes, hence to generate new classes.

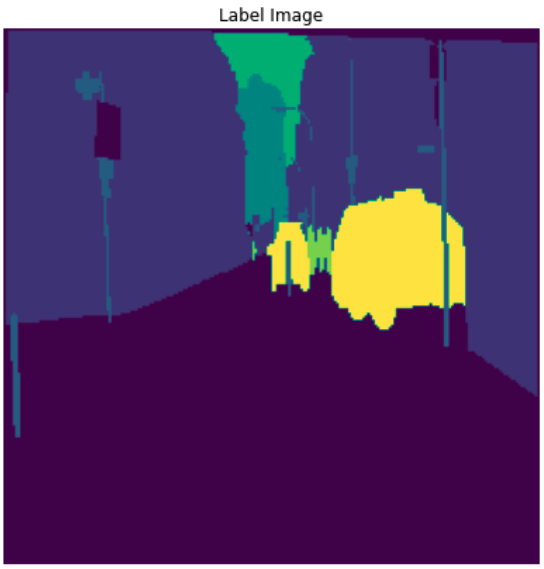
fig2: new segments and their regrouped names

```
# CATEGORIES TO CONSIDER IN THE DATA
cats = {'void': [0, 1, 2, 3, 4, 5, 6],
 'flat': [7, 8, 9, 10],
 'construction': [11, 12, 13, 14, 15, 16],
 'object': [17, 18, 19, 20],
 'nature': [21, 22],
 'sky': [23],
 'human': [24, 25],
 'vehicle': [26, 27, 28, 29, 30, 31, 32, 33, -1]}
```

Something that is worth notifying is that fortunately class 0 has to be regrouped in a void class. Therefore, in our developpement a numpy matrix initialization of zeros image pixels stays valid.

fig3: *input image and label image with new segments (8 classes instead* of 34 as above)



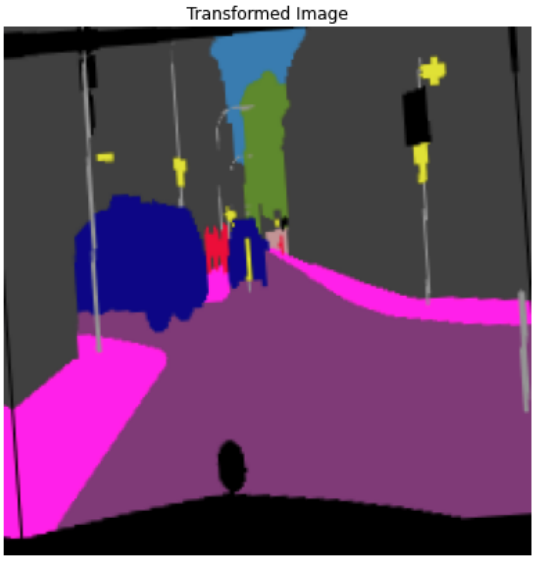


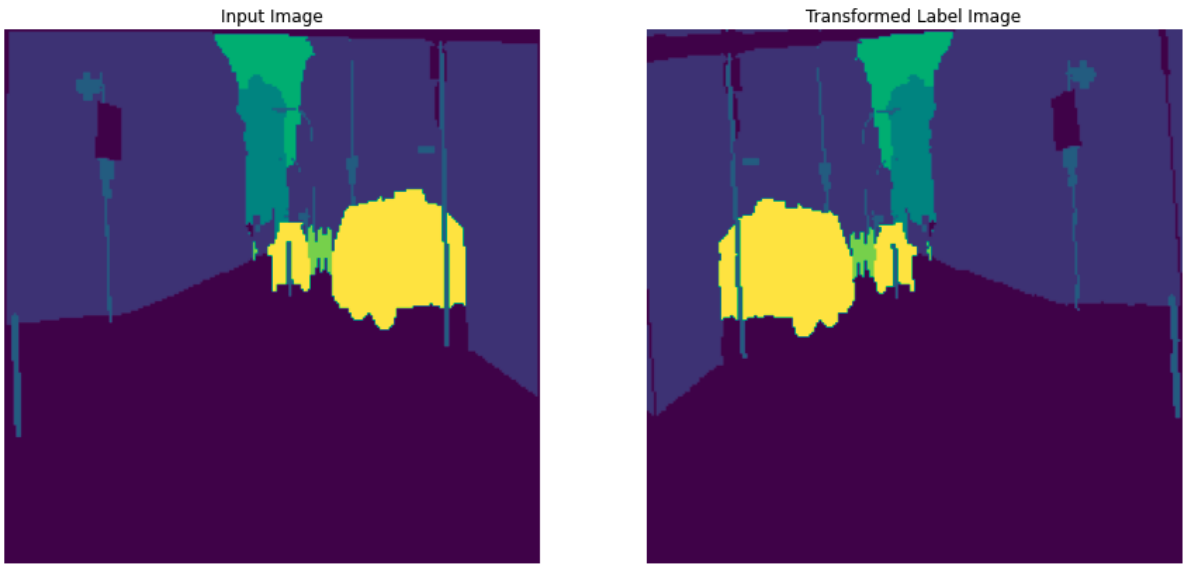
we can see that we have less class segments. For instance roads and pavements are now all in the same segment called “flat”.

5. **Data augmentation**

As we are big fans of tensorflow, we wanted to use the *ImageDataGeneration* library from tensorflow. It is very stable and well documented. It gives many functionalities to do image augmentation and the main advantage is the augmentation will only occur during training and raw images are not impacted. Unfortunately using this library did not give us the possibility to see and diagnose augmented images. We wanted to have this option since with this method, we were having weird image masks in our predictions. We could go for giving the possibility of the TensorFlow image data generator to persist augmented images before training, but that is not efficient. We end up trying **ALBUMENTIONS**. A very neat library for image augmentation. Easy to use and adapted for masked images. And we can easily plot augmented images and show the transformation we made as shown below.

fig: image before and after transformation for both image and mask





For the image augmentation, the transformations we implemented in the librairie are: horizontal flip, 10 degree rotation to the left and a little tweak in the image contrast. This is self explained on image above.

To sum up, in the data preparation part, we used three main big steps: the preprocess when reading by taking only three channels instead of four; the generation of new segments for our label images; and then the data augmentation step. The data augmentation is optionally used to assess its impact in the subsequent model training and prediction.

6. Training and Prediction with U-Net

Training and prediction with the U-Net is very straightforward. The model is very well designed, thanks to the community. We only need to adapt our input image shape and output label. Tensorflow Keras functional API makes the model easier to understand and flexible to use.

At the end of our data step, we converted our numpy data to Tensorflow dataset. This gives us more flexibility in adapting our input samples to our model way of input shape. A special thanks to tensorflow developers for providing *tensorflow.newaxis*. This method is magical. We can add a dimension to any tensor…Just magic!

What is worth noticing with our U-Net V2 implementation, is that the output label image mask should have dimension 1 (channel of size 1), instead of size 8 with one hot encoding along channels. Therefore, we collapsed our 8-dimensional masked images label to 1-channel 2-D images. We call it the magic of convolutions applied to the U-Net V2 model.

Below for the training of the U-Net, we used a batch size of one to anticipate for deployment. As we used a batch size of 32 that gave resource error when calling the endpoint during deployment. The choice was to either increase the deployment service container memory size or decrease the batch size.
With EarlyStopping implemented the model training stopped at the tenth iteration since the validation loss was not improving.

```
EPOCHS = 20
VAL_SUBSPLITS = 5
BUFFER_SIZE = 250
BATCH_SIZE = 1
steps_per_epoch=20
validation_steps=20
train_data_unet1 = train_data_unet.cache().shuffle(BUFFER_SIZE).batch(BATCH_SIZE)
val_data_unet1 = val_data_unet.cache().shuffle(BUFFER_SIZE).batch(BATCH_SIZE)
model_history = unet.fit(train_data_unet1,
          epochs=EPOCHS,
          validation_data=val_data_unet1,
          steps_per_epoch=steps_per_epoch,
          validation_steps=validation_steps,
          verbose=2,
          callbacks=[EarlyStopping(
              patience=7,
              min_delta=0.05,
              baseline=0.8,
              mode='min',
              monitor='val_loss',
              restore_best_weights=True,
              verbose=1)
          ])
Restoring model weights from the end of the best epoch: 10.
20/20 - 2s - loss: 0.0769 - accuracy: 0.9785 - val_loss: 0.1019 - val_accuracy:
0.9634 - 2s/epoch - 121ms/step
Epoch 00017: early stopping
```

fig: training and validation without data augmentation with U-Net

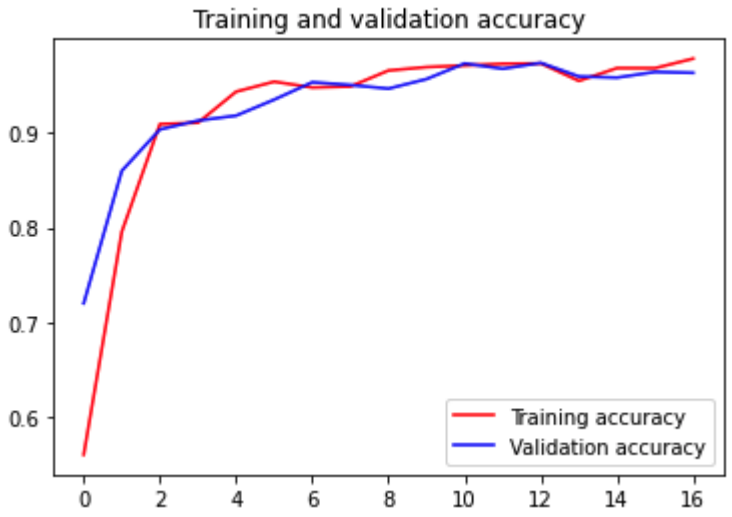
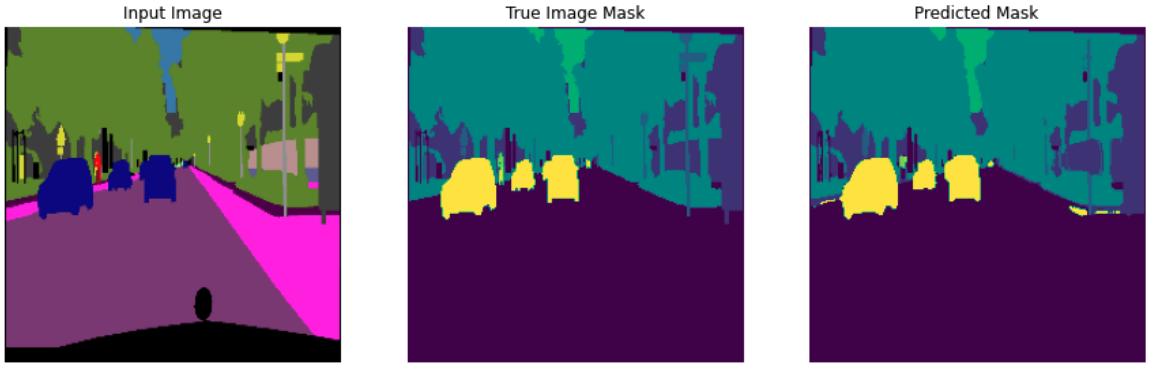


fig: an example of a predicted mask, a true mask and the input image



7. Training and Prediction with PreTrained Model

After the data step, it is time to implement our pretrained model. The process is the same as with the U-Net model, we adapted our input samples to the model, downloaded the weights, compiled the model and launched the training.

Thanks to the deep learning community and researchers in the world, Image feature extraction algorithms are reusable. We used the VGG16 weights extracted from Keras application library storage and used it for feature extraction. And then the Fully Connected Dense model tooks over in the second part with the Up Sampling process. The full model has many parameters.

fig: fcn8-vgg16 architecture, the two first and last layers of the model.

```
Model: "model_1"
_________________________________________________________________________________________________
 Layer (type)                Output Shape              Param #     Connected to
=================================================================================================
 input_1 (InputLayer)        [(None, 224, 224, 3       0           []
                             )]

 block1_conv1 (Conv2D)       (None, 224, 224, 64       1792        ['input_1[0][0]']
                             )

 conv2d_1 (Conv2D)           (None, 28, 28, 8)         2056        ['block3_pool3[0][0]']

 add_1 (Add)                 (None, 28, 28, 8)         0           ['cropping2d_1[0][0]',
                                                                    'conv2d_1[0][0]']

 conv2d_transpose_2 (Conv2DTran  (None, 224, 224, 8)   4096        ['add_1[0][0]']
 spose)

 activation (Activation)     (None, 224, 224, 8)       0           ['conv2d_transpose_2[0][0]']

=================================================================================================
Total params: 134,796,112
Trainable params: 134,796,112
Non-trainable params: 0
_________________________________________________________________________________________________
```

We can see above that the model has more than 134 million parameters to estimate in training. It is very compute intensive.

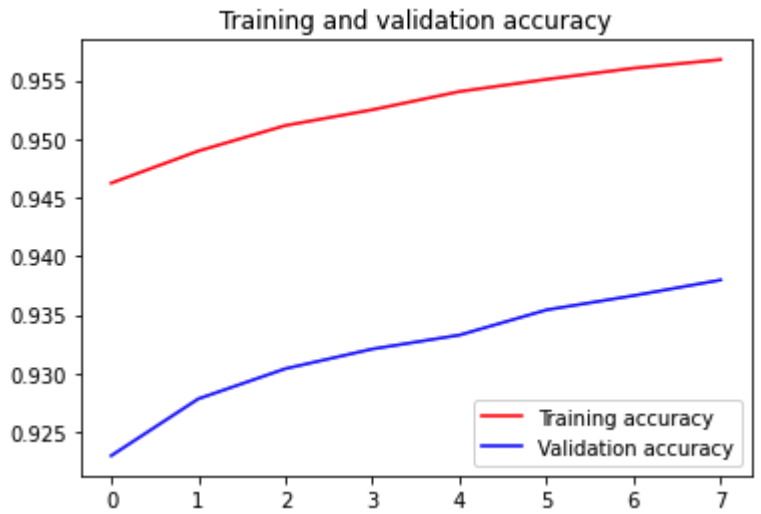
```
EPOCHS = 20
VAL_SUBSPLITS = 5
BUFFER_SIZE = 250
BATCH_SIZE = 32
train_data_vgg1 = train_data_vgg.cache().shuffle(BUFFER_SIZE).batch(BATCH_SIZE)
val_data_vgg1 = val_data_vgg.cache().shuffle(BUFFER_SIZE).batch(BATCH_SIZE)
print(train_data_vgg1.element_spec)

model_history = model_vgg_fcn.fit(train_data_vgg1,
          epochs=EPOCHS,
          validation_data=val_data_vgg1,
          verbose=2,
          callbacks=[EarlyStopping(
              patience=7,
              min_delta=0.05,
              baseline=0.8,
              mode='min',
              monitor='val_loss',
              restore_best_weights=True,
              verbose=1)
          ])

(TensorSpec(shape=(None, 224, 224, 3), dtype=tf.float32, name=None),
TensorSpec(shape=(None, 224, 224, 8), dtype=tf.int32, name=None))
Epoch 1/20
125/125 - 297s - loss: 0.1521 - accuracy: 0.9462 - val_loss: 0.2149 -
val_accuracy: 0.9230 - 297s/epoch - 2s/step
Epoch 8/20
Restoring model weights from the end of the best epoch: 1.
125/125 - 297s - loss: 0.1194 - accuracy: 0.9568 - val_loss: 0.1710 -
val_accuracy: 0.9380 - 297s/epoch - 2s/step
Epoch 00008: early stopping
```

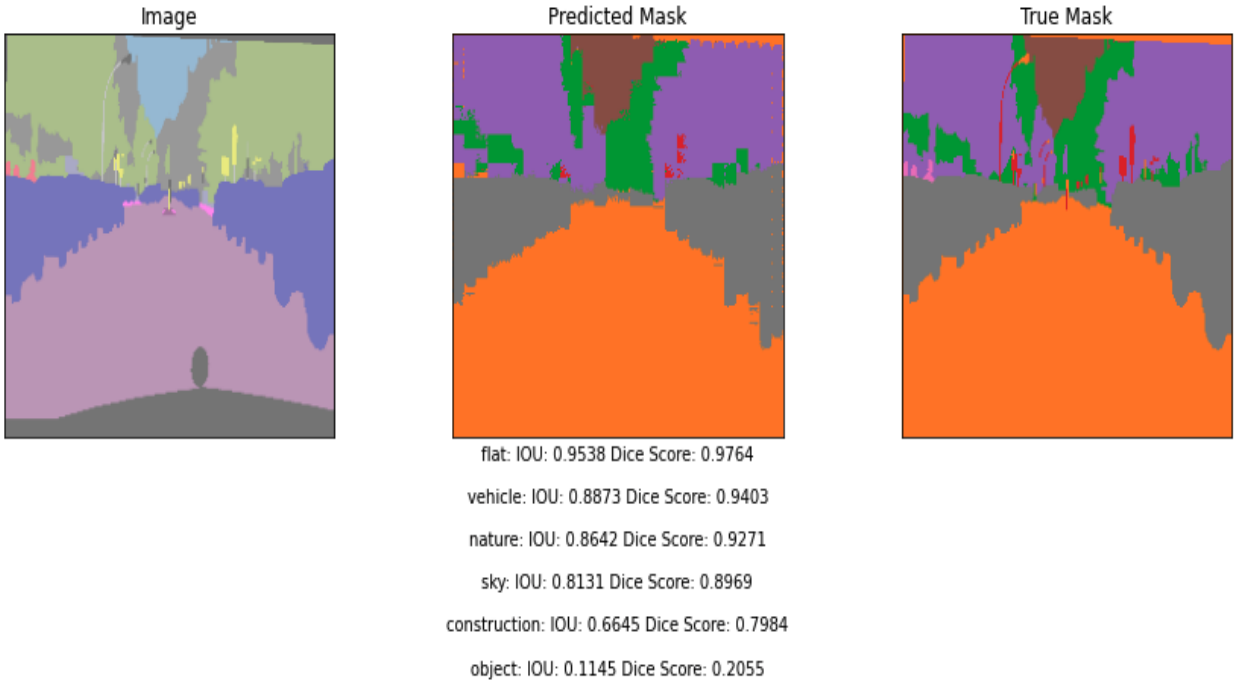
The prediction is done using label images of 8 classes with one hot encoding per slice. In contrast with the U-Net, predictions are probabilities of classes. We have to then collapse label images for visualization.

fig: fcn8-vgg16 training and validation accuracy



Below, we can see the prediction given by the model. And just under the predicted mask the IoU and Dice Metrics.

fig: fcn8-vgg16 prediction with metrics show below the predicted mask



8. Comparison of U-Net and FCN8-VGG16

Globally, the U-Net is more compact and easy to implement. It's very resource efficient.
And in contrast, the FCN8-VGG16 is very resource intensive, thus takes a lot of time to train. With that model, we lose flexibility of tweaking quickly and training again.

Nonetheless, the pretrained model with its complication in terms of architecture and model mixing should be more efficient. It brings a pre-trained part that could help to extract features and allow the training to start from somewhere. In our case, the simple U-Net does the job. Since it has some properties that Fully Connected Dense lack.

We could have added the Up Sampling part of the U-Net to the pre-trained model to see how it would have performed. But again, our use case is so simple and data so clean that we do not need any complications.

9. Azure Training and Deployment

Thanks to resource disposal, we can put our model to the test in a less resource constrained environment. Azure ML gives us the possibility to train our choosing model with a notebook. We gave it a try.
We encountered my troubles. But kept on going, despite the fact that we knew from other experiences that training a Machine Learning Model on Azure is more efficient using its SDK from our local machine. It is quicker and flexible. But for this project, we wanted to explore Notebook capabilities with Azure ML.

First problem was the latency. We were wondering why, since our compute instance is an optimized GPU machine. Why is our job running on CPU? And How to select the GPU or force the Notebook to use the GPU? The solution is the jupyter's kernel choice, but which one? Apparently the "python 3.8 Tensorflow". We picked it and it worked. We did not investigate further to see if we had the possibility to create our own GPU Kernel.

The easier part is the deployment. Since We were working on Azure ML through our Notebook, we have the possibility to test our endpoints inside. This gives us more flexibility. But what is the process to follow when deploying an ML model on Azure?
Since we did our training on Azure through Azure ML Notebook, our model after being well trained in a given Workspace, is ready for deployment. The is process is as follow:

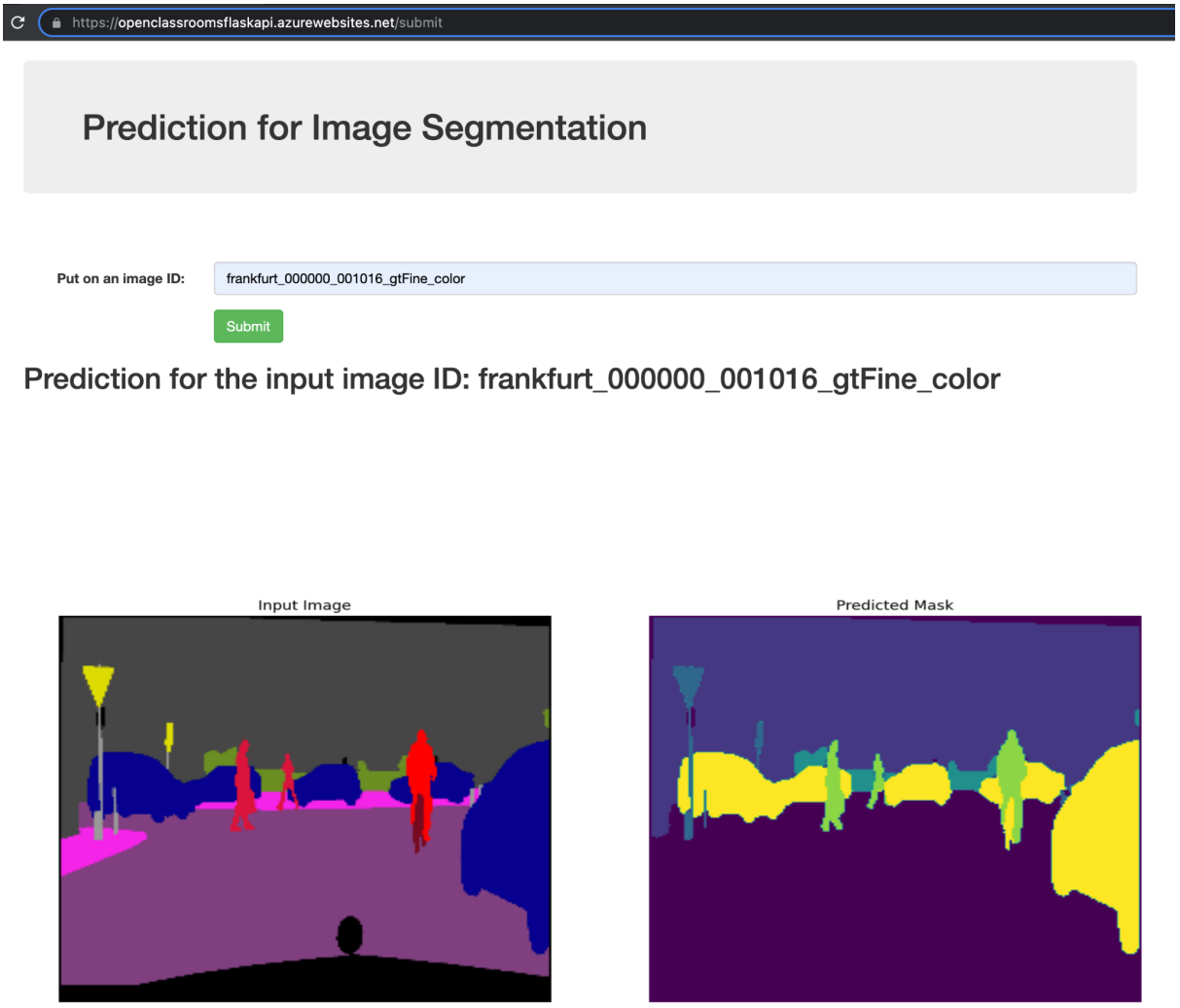
1. Creation of an Experiment: the entry point of any model training on Azure.
2. Registration of the model in the Experiment: the model needs to be registered
3. Creation of a container service: the deployment needs an isolated environment to run, such as docker image or Kubernetes service.
4. Creation of a score file: the registered model needs to know how to deal with input and output sample data. This is explained in the score dot py file through *init* and *run* method. the first for loading the model and its initialization, and the second for feeding the model with input and the way the prediction is returned.
5. Creation of a Python Environment Librairie file: for the trained model to run, it needs all libraries with which it's been built. Hence, a YML file with all necessary libraries is provided.
6. Launch a service deploy method of the Model API that takes the score file, the Env Lib file, the container service config as parameters to instantiate the creation of the deployment service. An Endpoint ready to be called upon is created at the end .

10. FLask API

At this step, our model is trained, validated, tested, deployed and running through Azure Microservices (in our case, a container).
Now it is time to encapsulate our very capable endpoint in an application, here a web application on top of Microsoft Azure Services.
We used a Flask API that we tested in our local machine. With the help of python VirtualEnv and Flask API command utilities, and a funny little tutorial found on the internet, we managed to build a local Web Application that uses our Endpoint to take an image ID as input and return an input image and its associated predicted mask.

11. Azure Web Application

After testing the web application using Flask on our local machine, we decided to deploy it with Azure Web App.
To do so, we took advantage of the smart integration between Azure Deployment Services and Github. Therefore, we pushed our flask local App directory with its requirements file built in an isolated environment on a our github repository; and then we built a Web App on Azure; finally we deployed our Application by telling the Azure newly created Web App to fetch our App on our Github repository.
Below the final outcome.
The Web Application takes an Image Raw input ID and throws back the prediction. We plotted side by side the input raw image and the predicted mask.
We have to note that we only have 3 raw images in our application.



12. Conclusion

To sum up, the image semantic segmentation project applied to self-driving cars with Azure Machine Learning Services has been a great journey. We know that we need to go more on reading research articles on Deep Learning to grasp more insight, but having the opportunity to build a Deep Neural Networks model almost from scratch is a great challenge. We took great pleasure in not only resolving bugs, but also unit testing and optimizing our models. We look forward to renewing the experience again.