segmentation_lab

February 18, 2019

1 Semantic Segmentation Lab

In this lab, you will build a deep learning network that locates a particular human target within an image. The premise is that a quadcopter (simulated) is searching for a target, and then will follow the target once found. It's not enough to simply say the target is present in the image in this case, but rather to know *where* in the image the target is, so that the copter can adjust its direction in order to follow.

Consequently, an image classification network is not enough to solve the problem. Instead, a semantic segmentation network is needed so that the target can be specifically located within the image.

You can click on any of the following to quickly jump to that part of this notebook: 1. Section **??** 2. Section **??** 3. Section **??** 4. Section 1.4 5. Section 1.5 6. Section 1.6

1.1 Data Collection

We have provided you with the dataset for this lab. If you haven't already downloaded the training and validation datasets, you can check out the README for this lab's repo for instructions as well.

```
In [1]: import os
        import glob
        import sys
        import tensorflow as tf

        from scipy import misc
        import numpy as np

        from tensorflow.contrib.keras.python import keras
        from tensorflow.contrib.keras.python.keras import layers, models

        from tensorflow import image

        from utils import scoring_utils
        from utils.separable_conv2d import SeparableConv2DKeras, BilinearUpSampling2D
        from utils import data_iterator
        from utils import plotting_tools
        from utils import model_tools
```

1.2 FCN Layers

In the Classroom, we discussed the different layers that constitute a fully convolutional network. The following code will introduce you to the functions that you will be using to build out your model.

1.2.1 Separable Convolutions

The Encoder for your FCN will essentially require separable convolution layers. Below we have implemented two functions - one which you can call upon to build out separable convolutions or regular convolutions. Each with batch normalization and with the ReLU activation function applied to the layers.

While we recommend the use of separable convolutions thanks to their advantages we covered in the Classroom, some of the helper code we will present for your model will require the use for regular convolutions. But we encourage you to try and experiment with each as well!

The following will help you create the encoder block and the final model for your architecture.

```
In [2]: def separable_conv2d_batchnorm(input_layer, filters, strides=1):
            output_layer = SeparableConv2DKeras(filters=filters,kernel_size=3, strides=strides,
                                        padding='same', activation='relu')(input_layer)

            output_layer = layers.BatchNormalization()(output_layer)
            return output_layer

        def conv2d_batchnorm(input_layer, filters, kernel_size=3, strides=1):
            output_layer = layers.Conv2D(filters=filters, kernel_size=kernel_size, strides=strid
                              padding='same', activation='relu')(input_layer)

            output_layer = layers.BatchNormalization()(output_layer)
            return output_layer
```

1.2.2 Bilinear Upsampling

The following helper function will help implement the bilinear upsampling layer. Currently, upsampling by a factor of 2 is recommended but you can try out different factors as well. You will use this to create the decoder block later!

```
In [3]: def bilinear_upsample(input_layer):
            output_layer = BilinearUpSampling2D((2,2))(input_layer)
            return output_layer
```

1.3 Build the Model

In the following cells, we will cover how to build the model for the task at hand.

- We will first create an Encoder Block, where you will create a separable convolution layer using an input layer and the size(depth) of the filters as your inputs.
- Next, you will create the Decoder Block, where you will create an upsampling layer using bilinear upsampling, followed by a layer concatentaion, and some separable convolution layers.

- Finally, you will combine the above two and create the model. In this step you will be able to experiment with different number of layers and filter sizes for each to build your model.

Let's cover them individually below.

1.3.1 Encoder Block

Below you will create a separable convolution layer using the separable_conv2d_batchnorm() function. The `filters` parameter defines the size or depth of the output layer. For example, 32 or 64.

```
In [4]: def encoder_block(input_layer, filters, strides):

            # TODO Create a separable convolution layer using the separable_conv2d_batchnorm() f
            output_layer = separable_conv2d_batchnorm(input_layer, filters, strides)

            return output_layer
```

1.3.2 Decoder Block

The decoder block, as covered in the Classroom, comprises of three steps -

- A bilinear upsampling layer using the bilinear_upsample() function. The current recommended factor for upsampling is set to 2.
- A layer concatenation step. This step is similar to skip connections. You will concatenate the upsampled small_ip_layer and the large_ip_layer.
- Some (one or two) additional separable convolution layers to extract some more spatial information from prior layers.

```
In [5]: def decoder_block(small_ip_layer, large_ip_layer, filters):

            # TODO Upsample the small input layer using the bilinear_upsample() function.
            upsample_layer = bilinear_upsample(small_ip_layer)

            # TODO Concatenate the upsampled and large input layers using layers.concatenate
            concated_layer = layers.concatenate([upsample_layer, large_ip_layer])

            # TODO Add some number of separable convolution layers
            output_layer = separable_conv2d_batchnorm(concated_layer, filters=filters, strides=1

            return output_layer
```

1.3.3 Model

Now that you have the encoder and decoder blocks ready, you can go ahead and build your model architecture!

There are three steps to the following: - Add encoder blocks to build out initial set of layers. This is similar to how you added regular convolutional layers in your CNN lab. - Add 1x1 Convolution layer using conv2d_batchnorm() function. Remember that 1x1 Convolutions require a kernel and stride of 1. - Add decoder blocks for upsampling and skip connections.

```
In [6]: def fcn_model(inputs, num_classes):

            # TODO Add Encoder Blocks.
            # Remember that with each encoder layer, the depth of your model (the number of filt
            encoder1 = encoder_block(inputs, filters=64, strides=2)
            encoder2 = encoder_block(encoder1, filters=64, strides=2)
            encoder3 = encoder_block(encoder2, filters=64, strides=2)
            encoder4 = encoder_block(encoder3, filters=128, strides=2)

            # TODO Add 1x1 Convolution layer using conv2d_batchnorm().
            onebyone_convolution_layer = conv2d_batchnorm(encoder4, filters=128, kernel_size=1,

            # TODO: Add the same number of Decoder Blocks as the number of Encoder Blocks
            decoder1 = decoder_block(onebyone_convolution_layer, encoder3, filters= 128)
            decoder2 = decoder_block(decoder1, encoder2, filters=64)
            decoder3 = decoder_block(decoder2, encoder1, filters=64)
            decoder4 = decoder_block(decoder3, inputs, filters=64)

            x = decoder4

            # The function returns the output layer of your model. "x" is the final layer obtain
            return layers.Conv2D(num_classes, 3, activation='softmax', padding='same')(x)
```

1.4 Training

The following cells will utilize the model you created and define an ouput layer based on the input and the number of classes.Following that you will define the hyperparameters to compile and train your model!

```
In [7]: """
        DON'T MODIFY ANYTHING IN THIS CELL THAT IS BELOW THIS LINE
        """

        image_hw = 128
        image_shape = (image_hw, image_hw, 3)
        inputs = layers.Input(image_shape)
        num_classes = 3

        # Call fcn_model()
        output_layer = fcn_model(inputs, num_classes)
```

1.4.1 Hyperparameters

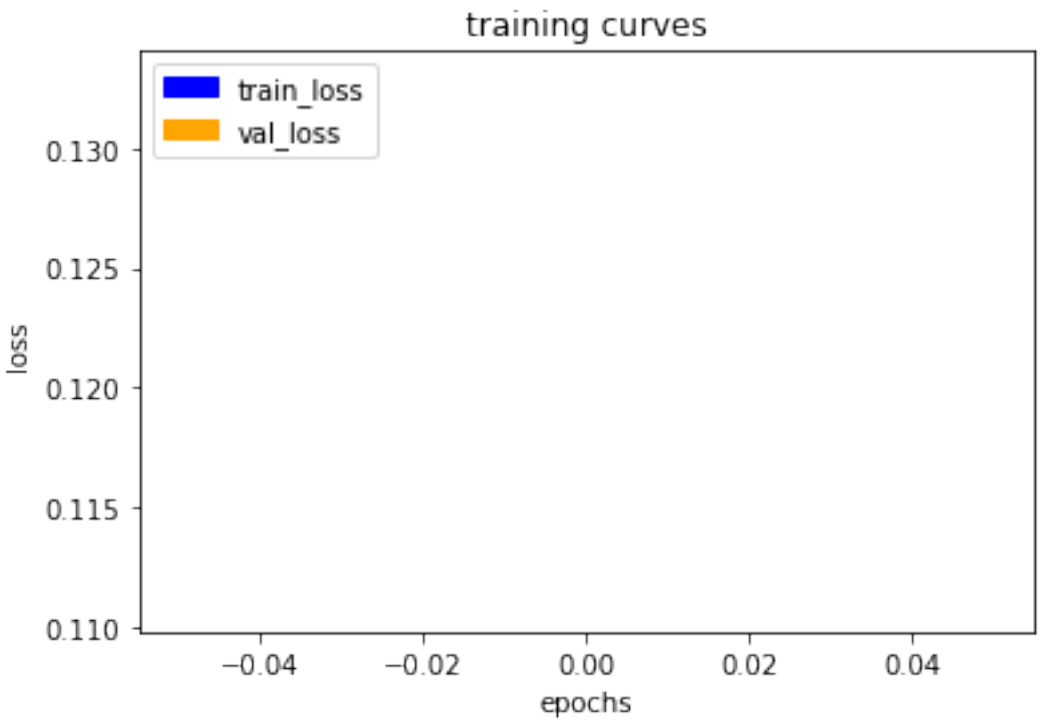
Define and tune your hyperparameters. - **batch_size**: number of training samples/images that get propagated through the network in a single pass. - **num_epochs**: number of times the entire training dataset gets propagated through the network. - **steps_per_epoch**: number of batches of training images that go through the network in 1 epoch. We have provided you with a default value. One recommended value to try would be based on the total number of images in training

dataset divided by the batch_size. - **validation_steps**: number of batches of validation images that go through the network in 1 epoch. This is similar to steps_per_epoch, except validation_steps is for the validation dataset. We have provided you with a default value for this as well. - **workers**: maximum number of processes to spin up. This can affect your training speed and is dependent on your hardware. We have provided a recommended value to work with.

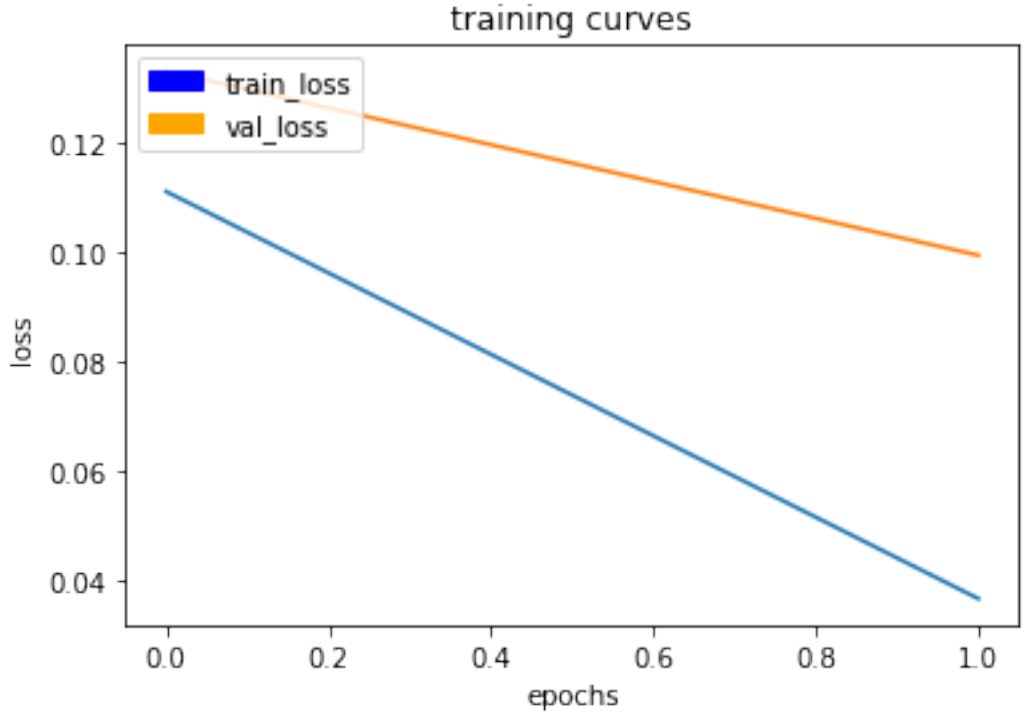
In [8]:
```
learning_rate = 0.01
batch_size = 64
num_epochs = 8
steps_per_epoch = 200
validation_steps = 50
workers = 2
```

In [9]:
```
"""
DON'T MODIFY ANYTHING IN THIS CELL THAT IS BELOW THIS LINE
"""
# Define the Keras model and compile it for training
model = models.Model(inputs=inputs, outputs=output_layer)

model.compile(optimizer=keras.optimizers.Adam(learning_rate), loss='categorical_crossent

# Data iterators for loading the training and validation data
train_iter = data_iterator.BatchIteratorSimple(batch_size=batch_size,
                                               data_folder=os.path.join('..', 'data', 't
                                               image_shape=image_shape,
                                               shift_aug=True)

val_iter = data_iterator.BatchIteratorSimple(batch_size=batch_size,
                                             data_folder=os.path.join('..', 'data', 'val
                                             image_shape=image_shape)

logger_cb = plotting_tools.LoggerPlotter()
callbacks = [logger_cb]

model.fit_generator(train_iter,
                    steps_per_epoch = steps_per_epoch, # the number of batches per epoch
                    epochs = num_epochs, # the number of epochs to train for,
                    validation_data = val_iter, # validation iterator
                    validation_steps = validation_steps, # the number of batches to vali
                    callbacks=callbacks,
                    workers = workers)
```

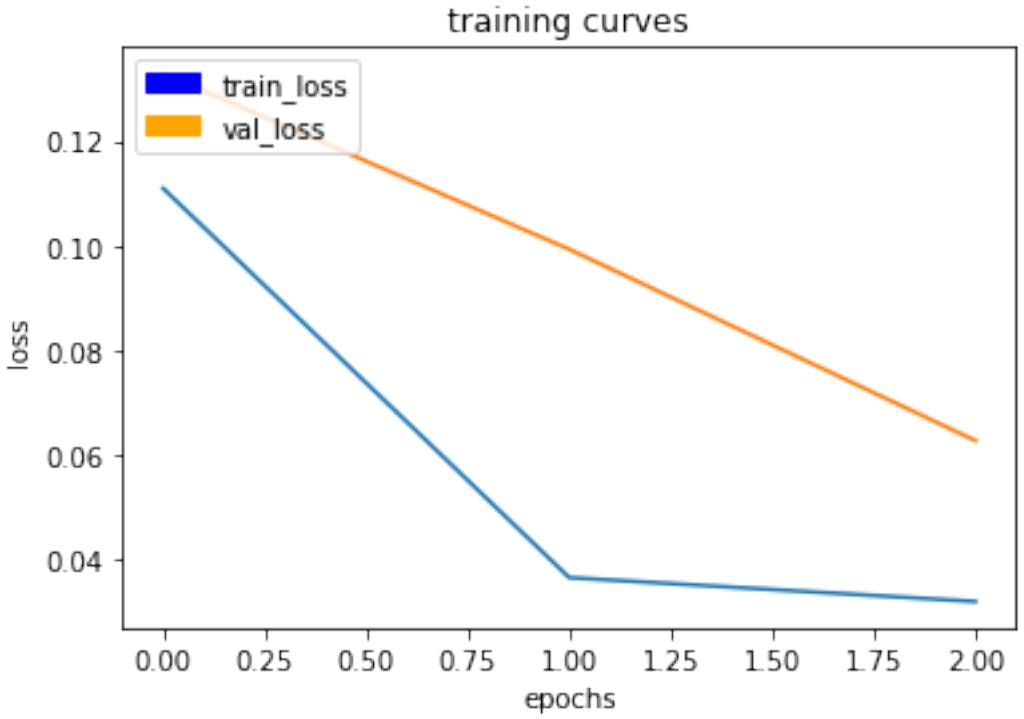
```
Epoch 1/8
199/200 [============================>.] - ETA: 0s - loss: 0.1108
```



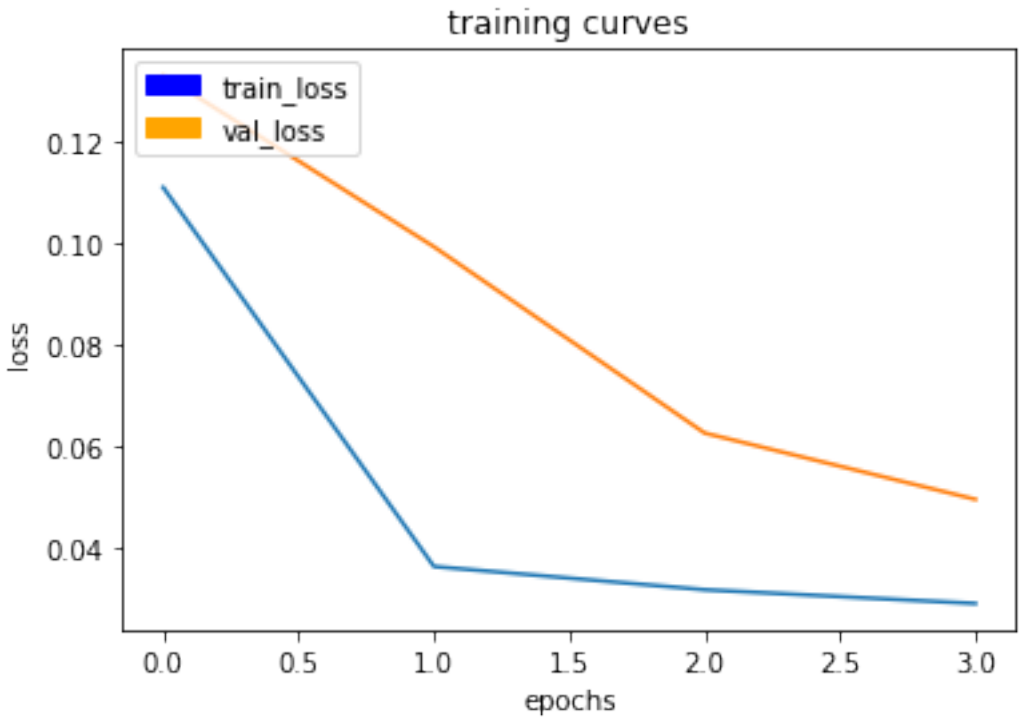

```
200/200 [==============================] - 134s - loss: 0.1105 - val_loss: 0.1330
Epoch 2/8
199/200 [==============================>.] - ETA: 0s - loss: 0.0365
```



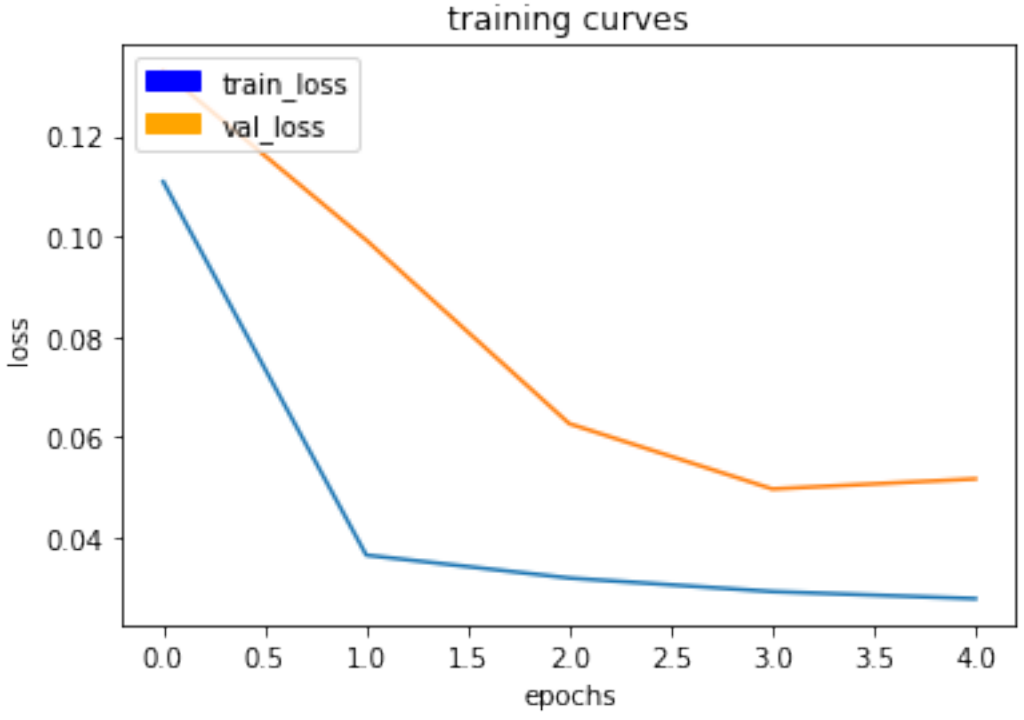
```
200/200 [==============================] - 128s - loss: 0.0364 - val_loss: 0.0993
Epoch 3/8
199/200 [============================>.] - ETA: 0s - loss: 0.0319
```



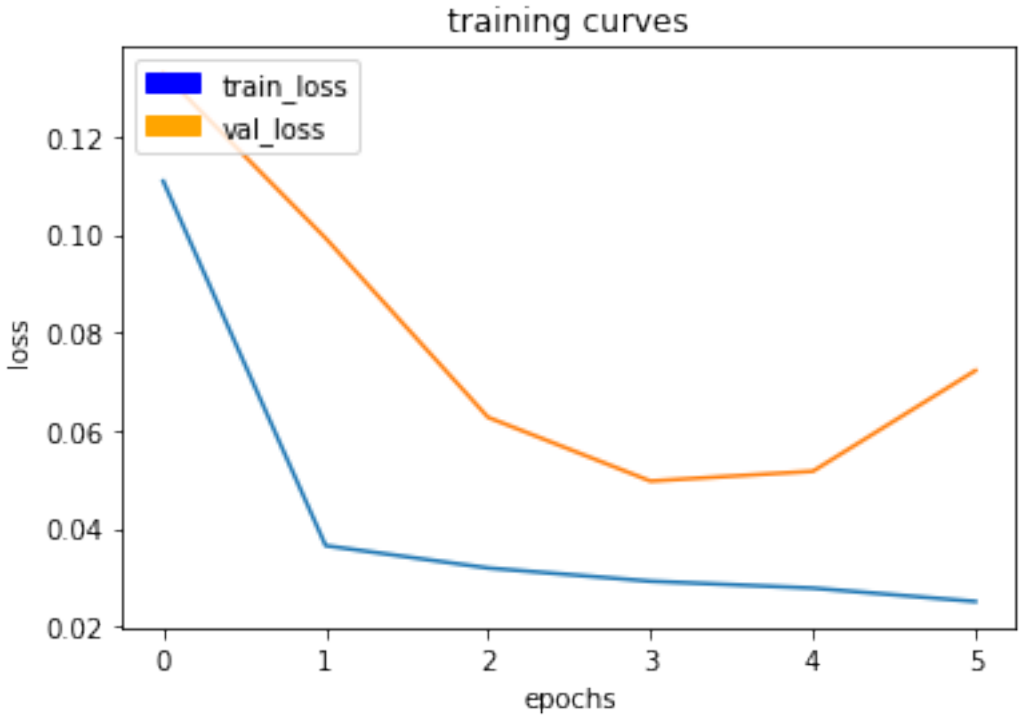

```
200/200 [==============================] - 128s - loss: 0.0319 - val_loss: 0.0627
Epoch 4/8
199/200 [============================>.] - ETA: 0s - loss: 0.0292
```



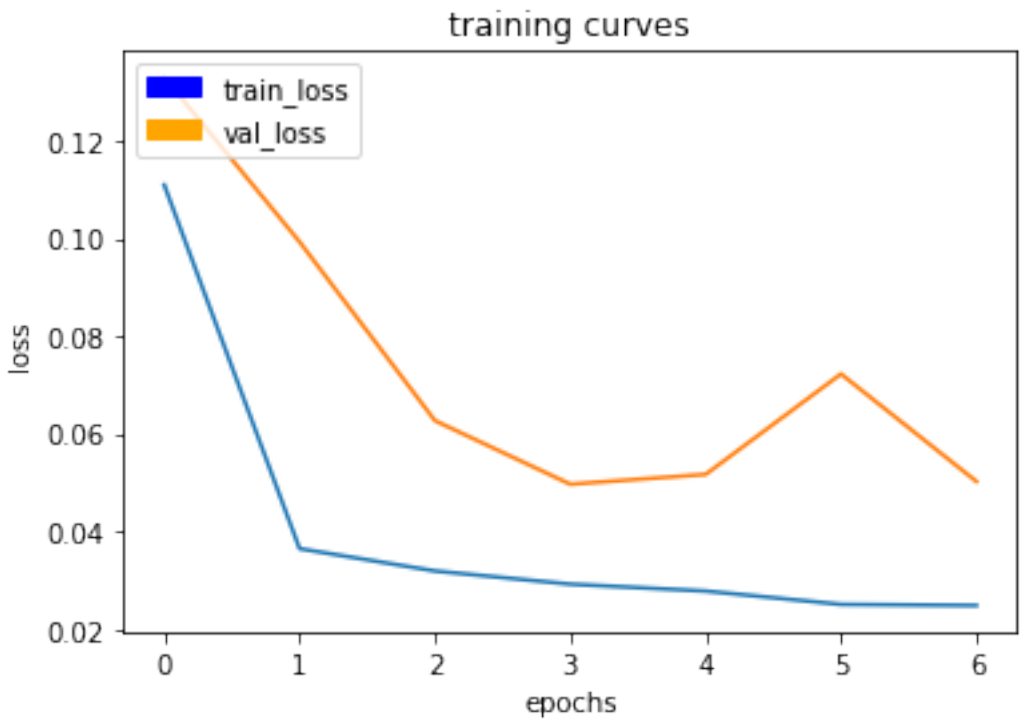

```
200/200 [==============================] - 131s - loss: 0.0292 - val_loss: 0.0497
Epoch 5/8
199/200 [==============================>.] - ETA: 0s - loss: 0.0278
```



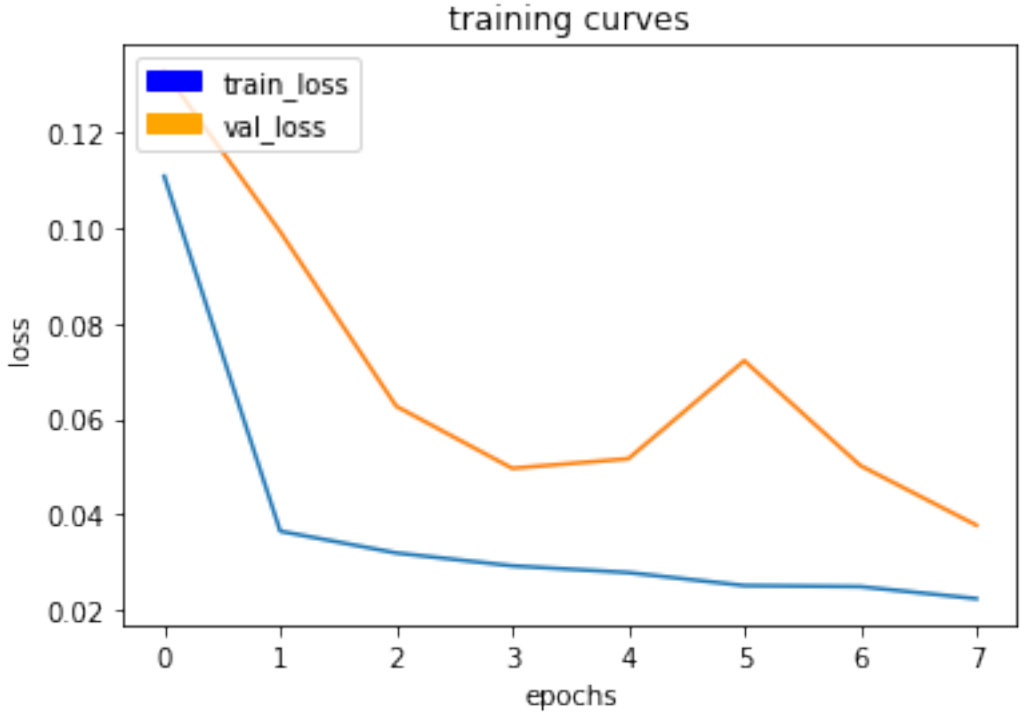
```
200/200 [==============================] - 132s - loss: 0.0278 - val_loss: 0.0517
Epoch 6/8
199/200 [============================>.] - ETA: 0s - loss: 0.0251
```




```
200/200 [==============================] - 134s - loss: 0.0251 - val_loss: 0.0722
Epoch 7/8
199/200 [============================>.] - ETA: 0s - loss: 0.0248
```




```
200/200 [==============================] - 134s - loss: 0.0248 - val_loss: 0.0502
Epoch 8/8
199/200 [==============================>.] - ETA: 0s - loss: 0.0223
```



```
200/200 [==============================] - 134s - loss: 0.0223 - val_loss: 0.0377
```

```
Out[9]: <tensorflow.contrib.keras.python.keras.callbacks.History at 0x7f69d76ca588>
```

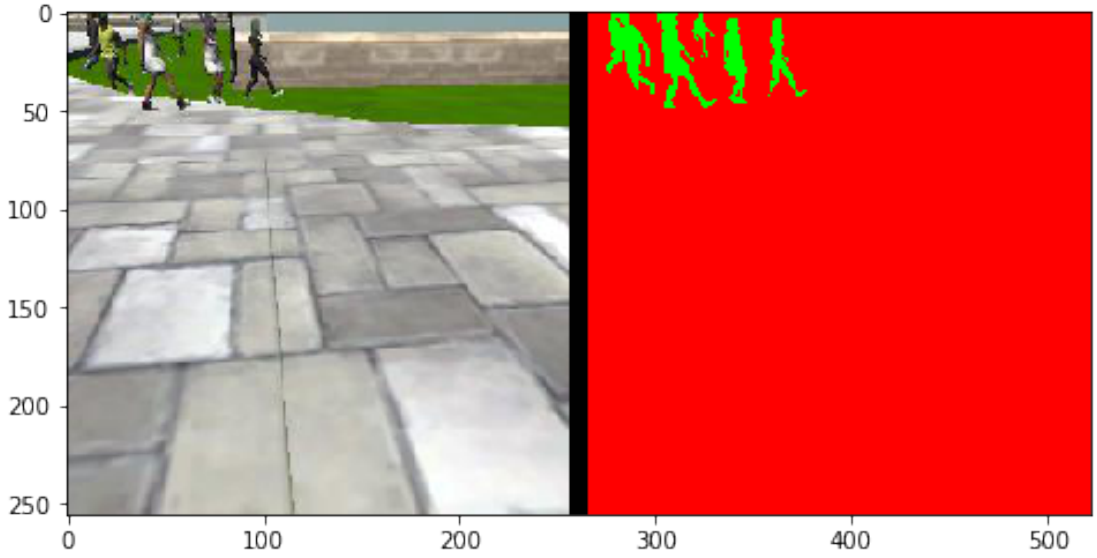
```
In [10]: # Save your trained model weights
         weight_file_name = 'model_weights'
         model_tools.save_network(model, weight_file_name)
```

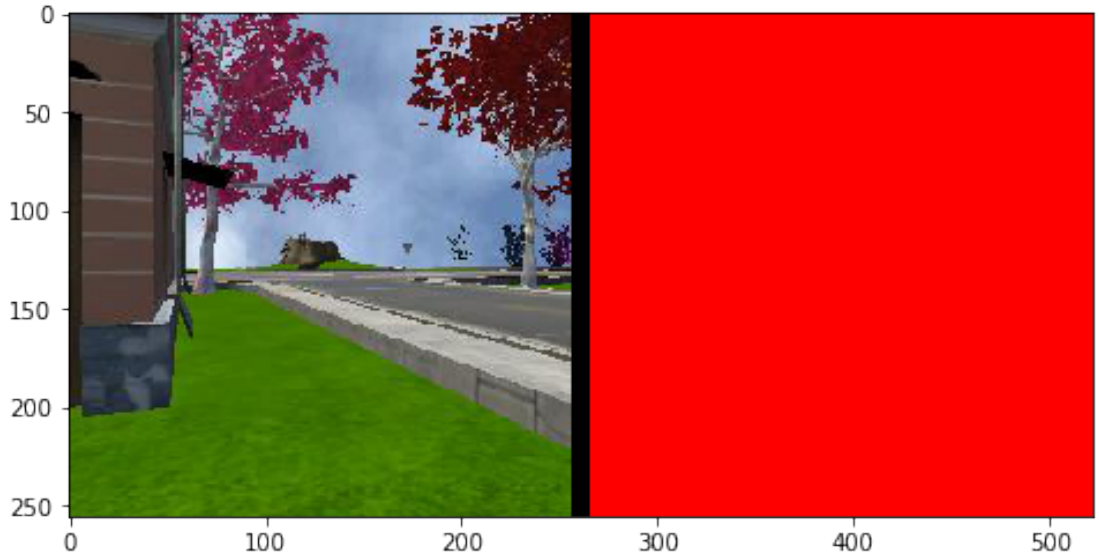
1.5 Prediction

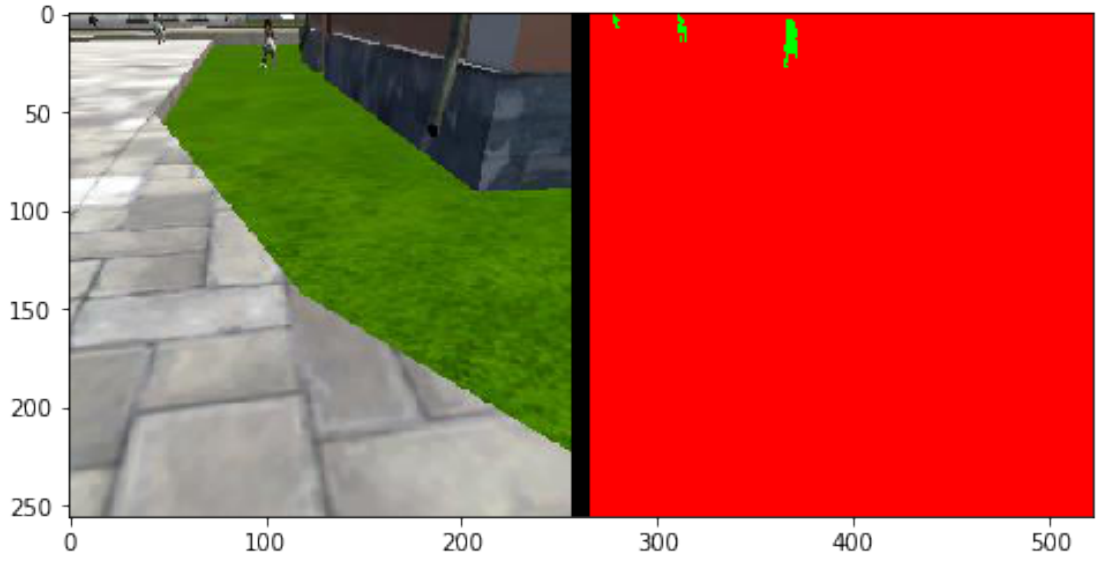
```
In [11]: # If you need to load a model which you previously trained you can uncomment the codeli

         # weight_file_name = 'model_weights'
         # restored_model = model_tools.load_network(weight_file_name)
```

```
In [12]: # generate predictions, save in the runs, directory.
         run_number = 'run1'
         validation_path, output_path = model_tools.write_predictions_grade_set(model,run_number
```

```
In [13]: # take a look at predictions
         # validation_path = 'validation'
         im_files = plotting_tools.get_im_file_sample(run_number,validation_path)
         for i in range(3):
             im_tuple = plotting_tools.load_images(im_files[i])
             plotting_tools.show_images(im_tuple)
```







1.6 Evaluation

Let's evaluate your model!

```
In [14]: scoring_utils.score_run(validation_path, output_path)
```

```
number of validation samples intersection over the union evaulated on 1184
average intersection over union for background is 0.9920874176681285
```

```
average intersection over union for other people is 0.296333515684606
average intersection over union for hero is 0.14118430939486792
global average intersection over union is 0.4765350809158675
```