Follow-Me Project

Congratulations on reaching the final project of the Robotics Nanodegree!

Previously, you worked on the Semantic Segmentation lab where you built a deep learning network that locates a particular human target within an image. For this project, you will utilize what you implemented and learned from that lab and extend it to train a deep learning model that will allow a simulated quadcopter to follow around the person that it detects!

Most of the code below is similar to the lab with some minor modifications. You can start with your existing solution, and modify and improve upon it to train the best possible model for this task.

You can click on any of the following to quickly jump to that part of this notebook:

Data Collection

We have provided you with a starting dataset for this project. Download instructions can be found in the README for this project's repo. Alternatively, you can collect additional data of your own to improve your model. Check out the "Collecting Data" section in the Project Lesson in the Classroom for more details!

In [36]:

```
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
```

FCN Layers

In the Classroom, we discussed the different layers that constitute a fully convolutional network (FCN). The following code will introduce you to the functions that you need to build your semantic segmentation model.

Separable Convolutions

The Encoder for your FCN will essentially require separable convolution layers, due to their advantages as explained in the classroom. The 1x1 convolution layer in the FCN, however, is a regular convolution. Implementations for both are provided below for your use. Each includes batch normalization with the ReLU activation function applied to the layers.

In [37]:
```
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, str
ides=strides,
                      padding='same', activation='relu')(input_layer)

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

Bilinear Upsampling

The following helper function implements the bilinear upsampling layer. Upsampling by a factor of 2 is generally recommended, but you can try out different factors as well. Upsampling is used in the decoder block of the FCN.

In [38]:
```
def bilinear_upsample(input_layer):
    output_layer = BilinearUpSampling2D((2,2))(input_layer)
    return output_layer
```

Build the Model

In the following cells, you will build an FCN to train a model to detect and locate the hero target within an image. The steps are:

- Create an `encoder_block`
- Create a `decoder_block`
- Build the FCN consisting of encoder block(s), a 1x1 convolution, and decoder block(s). This step requires experimentation with different numbers of layers and filter sizes to build your model.

Encoder Block

Create an encoder block that includes 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 [39]:
```
def encoder_block(input_layer, filters, strides):

    # TODO Create a separable convolution layer using the separable_conv2d_bat
chnorm() function.
    output_layer = separable_conv2d_batchnorm(input_layer, filters, strides)
    return output_layer
```

Decoder Block

The decoder block is comprised of three parts:

- A bilinear upsampling layer using the upsample_bilinear() 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 [40]:
```
def decoder_block(small_ip_layer, large_ip_layer, filters):

    # TODO Upsample the small input layer using the bilinear_upsample() functi
on.
    upsample = bilinear_upsample(small_ip_layer)
    # TODO Concatenate the upsampled and large input layers using layers.conca
tenate
    concat = layers.concatenate([upsample, large_ip_layer])
    # TODO Add some number of separable convolution layers
    output_layer = separable_conv2d_batchnorm(concat, filters)

    return output_layer
```

Model

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

There are three steps:

- Add encoder blocks to build the encoder layers. This is similar to how you added regular convolutional layers in your CNN lab.
- Add a 1x1 Convolution layer using the conv2d_batchnorm() function. Remember that 1x1 Convolutions require a kernel and stride of 1.
- Add decoder blocks for the decoder layers.

In [41]:
```
def fcn_model(inputs, num_classes):

    # TODO Add Encoder Blocks.
    # Remember that with each encoder layer, the depth of your model (the numb
er of filters) increases.
    filter1 = 32
    filter2 = 64
    filter3 = 128
    enc1 = encoder_block(inputs, filter1, 2)
    enc2 = encoder_block(enc1, filter2, 2)
    # TODO Add 1x1 Convolution layer using conv2d_batchnorm().
    conv = conv2d_batchnorm(enc2, filter3)
    # TODO: Add the same number of Decoder Blocks as the number of Encoder Blo
cks
    dec1 = decoder_block(conv, enc1, filter2)
    dec2 = decoder_block(dec1, inputs, filter1)

    x = dec2

    # The function returns the output layer of your model. "x" is the final la
yer obtained from the last decoder_block()
    return layers.Conv2D(num_classes, 1, activation='softmax', padding='same')
(x)
```

Training

The following cells will use the FCN you created and define an ouput layer based on the size of the processed image and the number of classes recognized. You will define the hyperparameters to compile and train your model.

Please Note: For this project, the helper code in `data_iterator.py` will resize the copter images to 160x160x3 to speed up training.

In [42]:
```
"""
DON'T MODIFY ANYTHING IN THIS CELL THAT IS BELOW THIS LINE
"""

image_hw = 160
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)
```

Hyperparameters

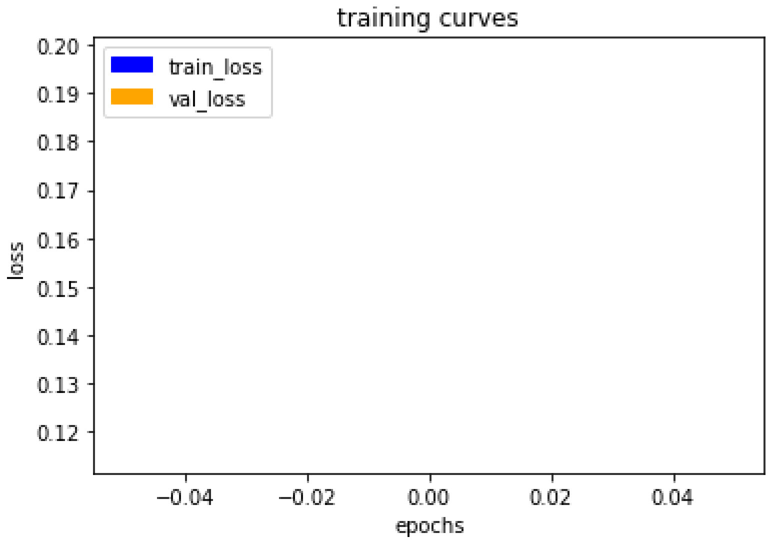
Define and tune your hyperparameters.

- **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.

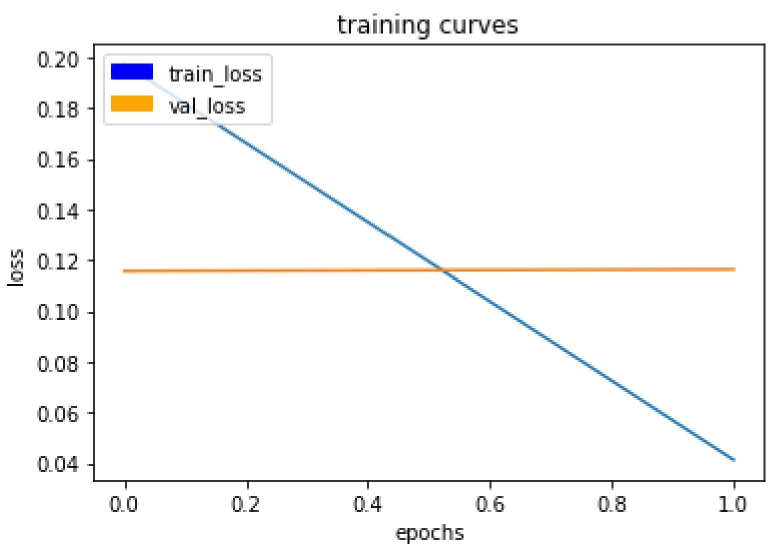
In [43]:
```
learning_rate = 0.005
batch_size = 16
num_epochs = 10
steps_per_epoch = 125
validation_steps = 50
workers = 3
```

In [45]:
```
"""
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='categorica
l_crossentropy')

# Data iterators for loading the training and validation data
train_iter = data_iterator.BatchIteratorSimple(batch_size=batch_size,
                                               data_folder=os.path.join('..',
'data', 'train_combined'),
                                               image_shape=image_shape,
                                               shift_aug=True)

val_iter = data_iterator.BatchIteratorSimple(batch_size=batch_size,
                                             data_folder=os.path.join('..', 'd
ata', 'validation'),
                                             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 batch
es to validate on
                    callbacks=callbacks,
                    workers = workers)
```

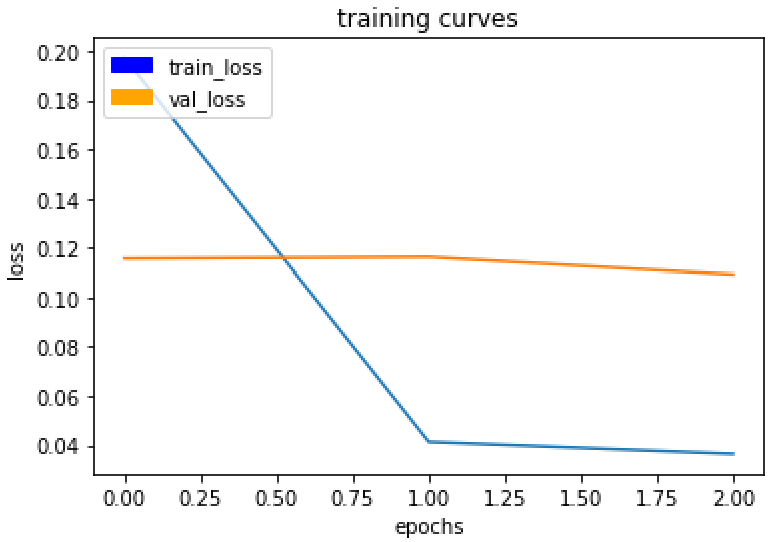
```
Epoch 1/10
124/125 [===========================>.] - ETA: 34s - loss: 0.1982
```



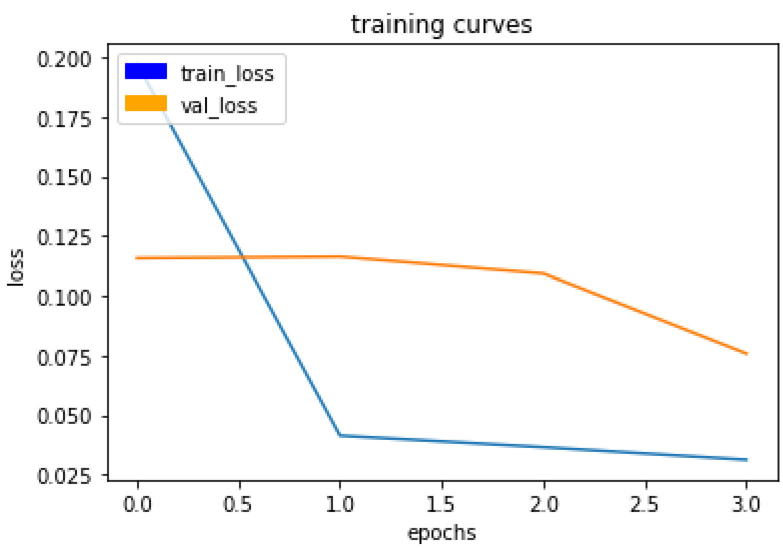

```
125/125 [=============================] - 4669s - loss: 0.1969 - val_loss:
0.1157
Epoch 2/10
124/125 [===========================>.] - ETA: 34s - loss: 0.0414
```



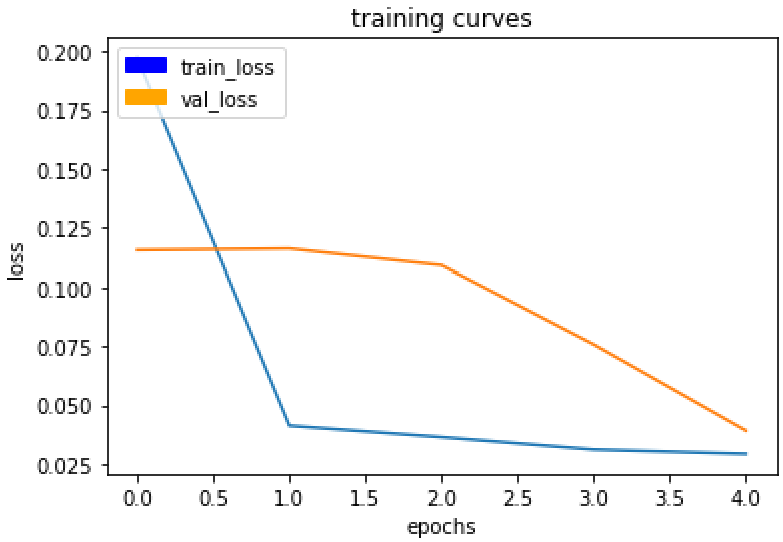

```
125/125 [=============================] - 4645s - loss: 0.0413 - val_loss:
0.1163
Epoch 3/10
124/125 [===========================>.] - ETA: 34s - loss: 0.0365
```



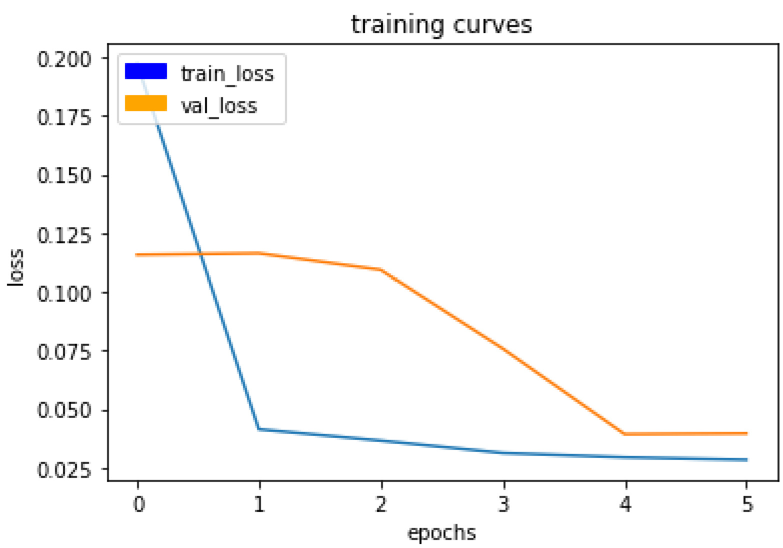

```
125/125 [==============================] - 4644s - loss: 0.0364 - val_loss:
0.1093
Epoch 4/10
124/125 [============================>.] - ETA: 34s - loss: 0.0311
```



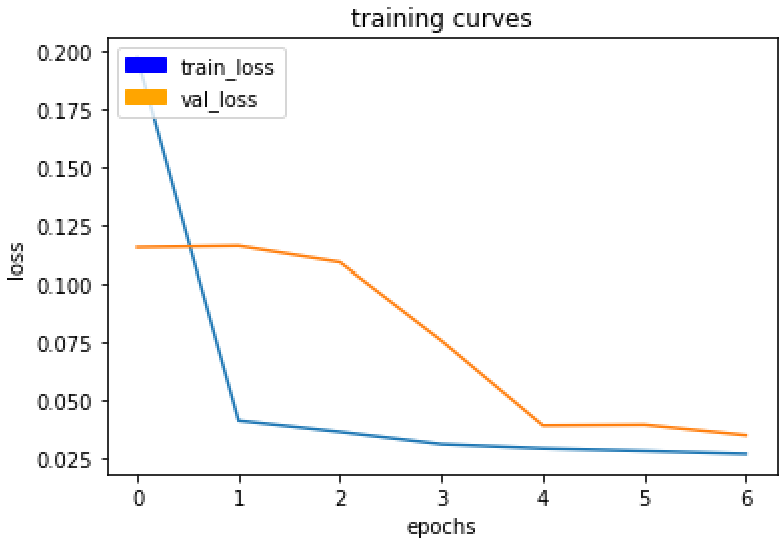

```
125/125 [==============================] - 4650s - loss: 0.0311 - val_loss:
0.0758
Epoch 5/10
124/125 [============================>.] - ETA: 34s - loss: 0.0293
```



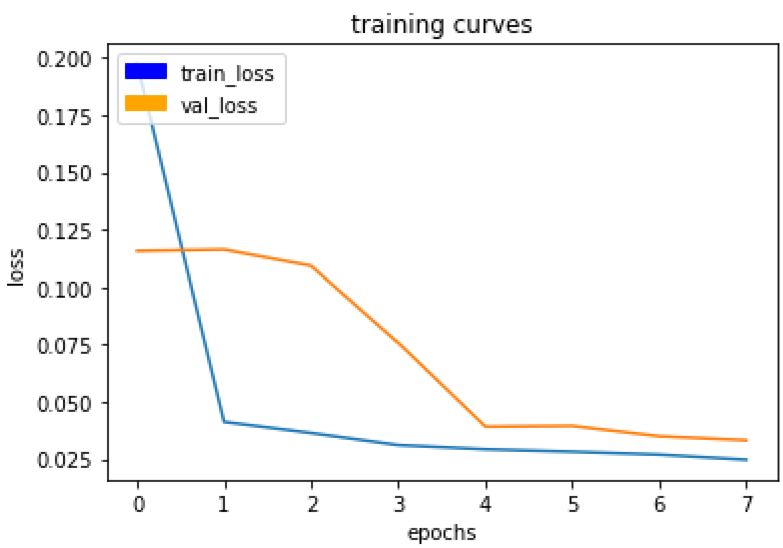

```
125/125 [==============================] - 4641s - loss: 0.0294 - val_loss:
0.0392
Epoch 6/10
124/125 [============================>.] - ETA: 34s - loss: 0.0283
```



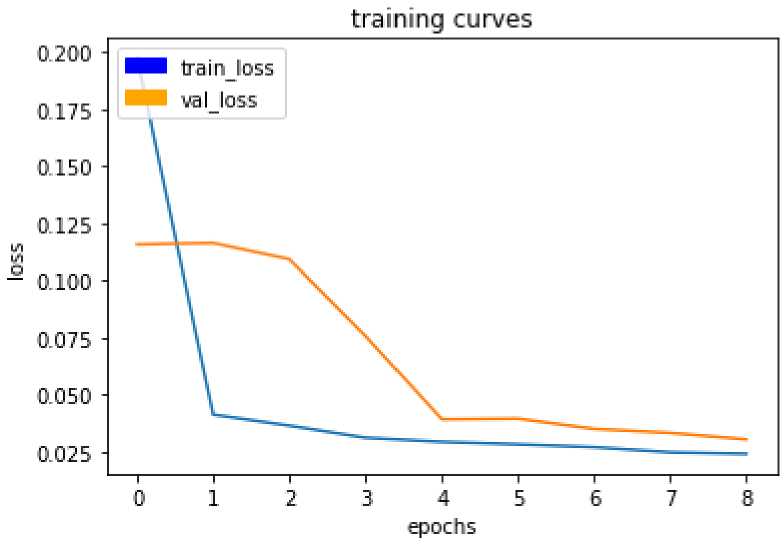

```
125/125 [==============================] - 4646s - loss: 0.0282 - val_loss:
0.0395
Epoch 7/10
124/125 [============================>.] - ETA: 34s - loss: 0.0270
```



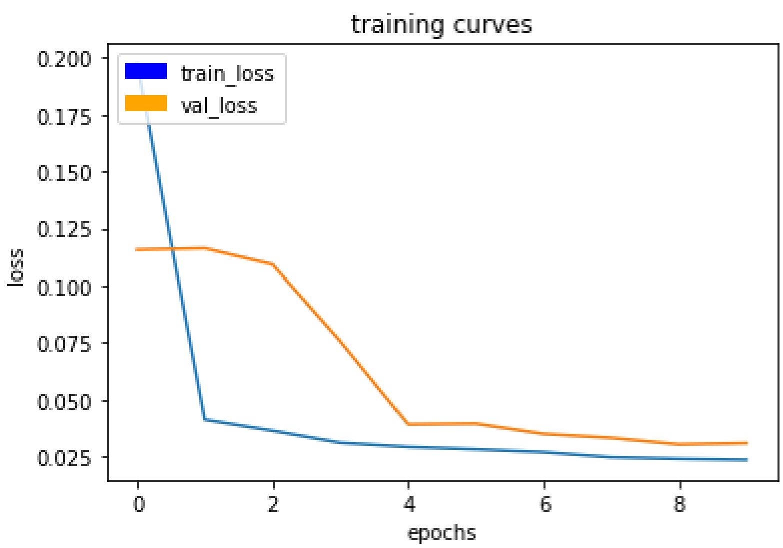

```
125/125 [==============================] - 4645s - loss: 0.0270 - val_loss:
0.0350
Epoch 8/10
124/125 [============================>.] - ETA: 34s - loss: 0.0248
```




```
125/125 [==============================] - 4637s - loss: 0.0248 - val_loss:
0.0333
Epoch 9/10
124/125 [============================>.] - ETA: 34s - loss: 0.0241
```




```
125/125 [==============================] - 4643s - loss: 0.0240 - val_loss:
0.0304
Epoch 10/10
124/125 [============================>.] - ETA: 34s - loss: 0.0236
```




```
125/125 [==============================] - 4659s - loss: 0.0236 - val_loss:
0.0310
```

Out[45]: <tensorflow.contrib.keras.python.keras.callbacks.History at 0x1c98e5c8240>

In [46]:
```
# Save your trained model weights
weight_file_name = 'model_weights'
model_tools.save_network(model, weight_file_name)
```

Prediction

Now that you have your model trained and saved, you can make predictions on your validation dataset. These predictions can be compared to the mask images, which are the ground truth labels, to evaluate how well your model is doing under different conditions.

There are three different predictions available from the helper code provided:

- **patrol_with_targ**: Test how well the network can detect the hero from a distance.
- **patrol_non_targ**: Test how often the network makes a mistake and identifies the wrong person as the target.
- **following_images**: Test how well the network can identify the target while following them.

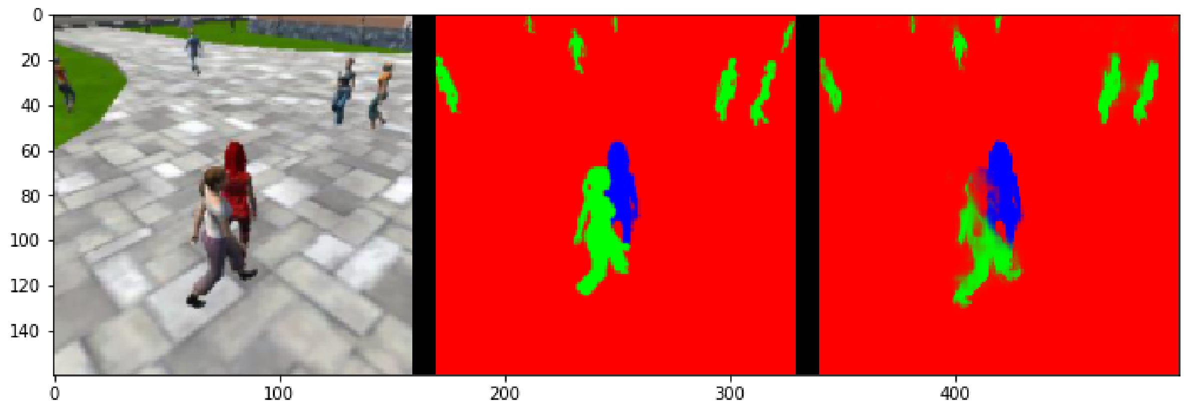
In [ ]:
```
# If you need to load a model which you previously trained you can uncomment t
he codeline that calls the function below.

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

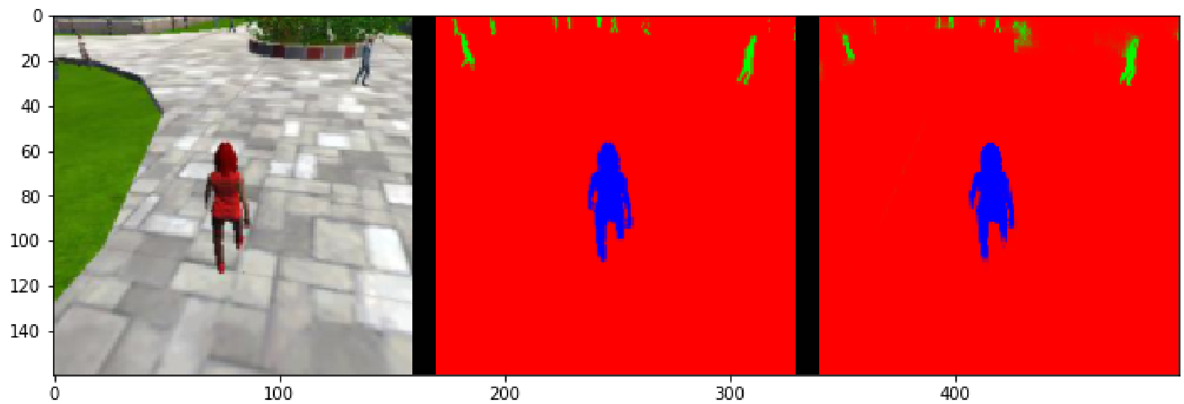
The following cell will write predictions to files and return paths to the appropriate directories. The `run_num` parameter is used to define or group all the data for a particular model run. You can change it for different runs. For example, 'run_1', 'run_2' etc.

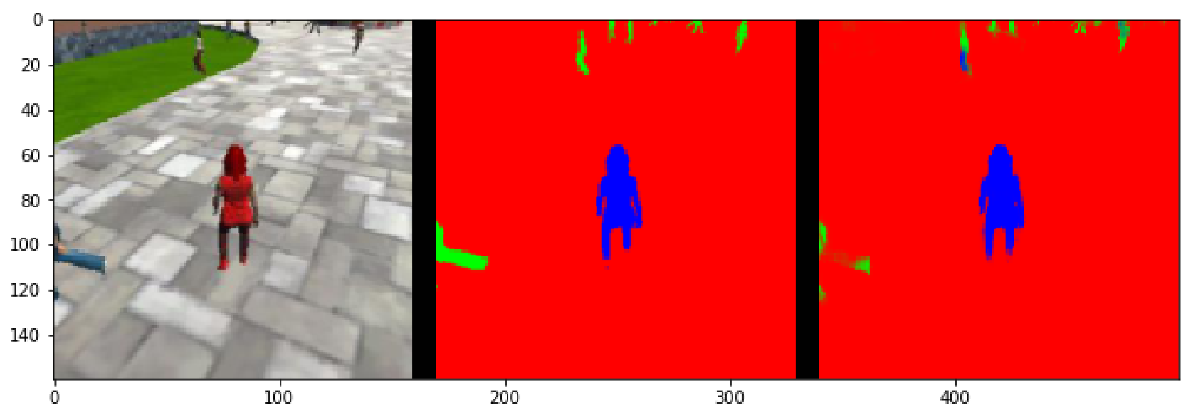
In [47]:
```
run_num = 'run_2'

val_with_targ, pred_with_targ = model_tools.write_predictions_grade_set(model,
                                        run_num,'patrol_with_targ', 'sample_ev
aluation_data')

val_no_targ, pred_no_targ = model_tools.write_predictions_grade_set(model,
                                        run_num,'patrol_non_targ', 'sample_eva
luation_data')

val_following, pred_following = model_tools.write_predictions_grade_set(model,
                                        run_num,'following_images', 'sample_ev
aluation_data')
```

Now lets look at your predictions, and compare them to the ground truth labels and original images. Run each of the following cells to visualize some sample images from the predictions in the validation set.

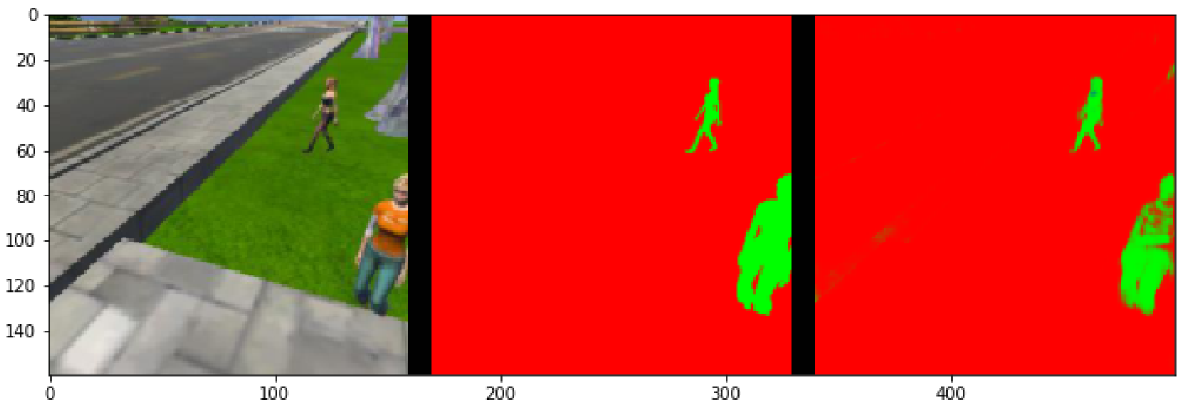
In [48]:
```
# images while following the target
im_files = plotting_tools.get_im_file_sample('sample_evaluation_data','followi
ng_images', run_num)
for i in range(3):
    im_tuple = plotting_tools.load_images(im_files[i])
    plotting_tools.show_images(im_tuple)
```

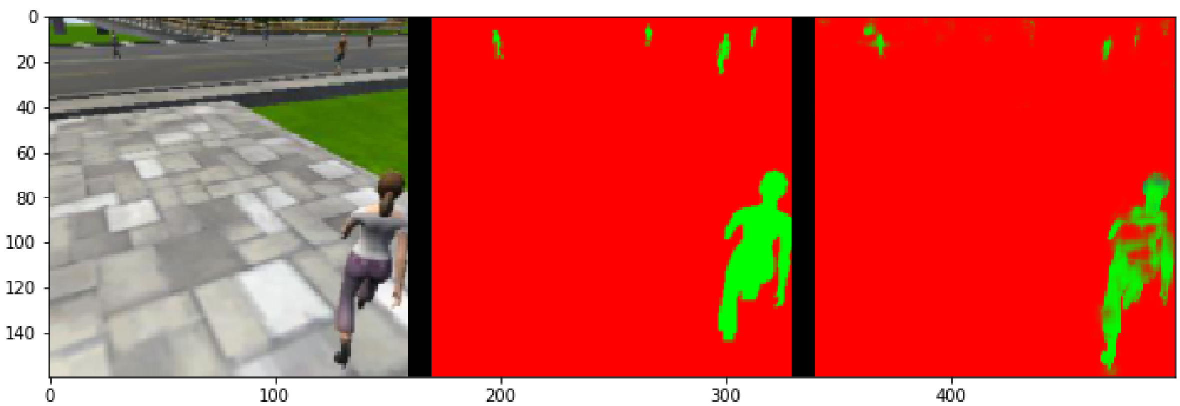


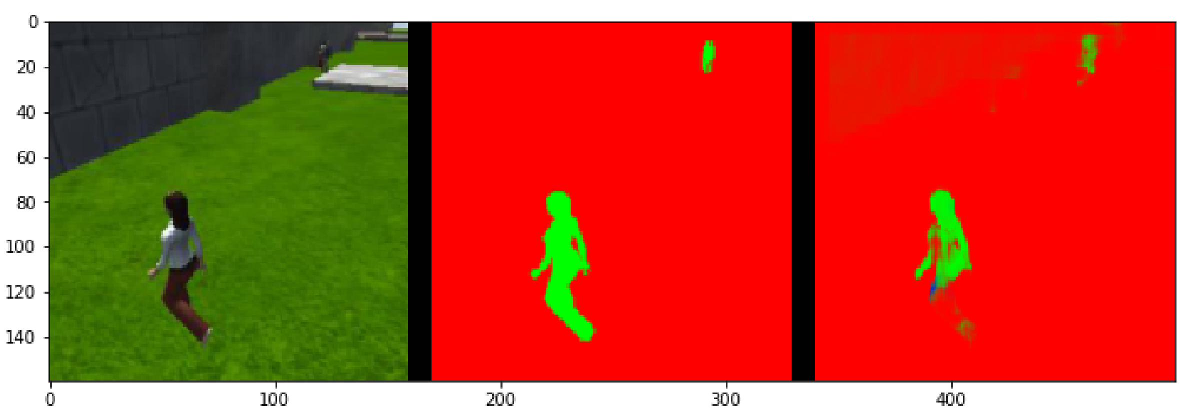




In [49]:
```
# images while at patrol without target
im_files = plotting_tools.get_im_file_sample('sample_evaluation_data','patrol_
non_targ', run_num)
for i in range(3):
    im_tuple = plotting_tools.load_images(im_files[i])
    plotting_tools.show_images(im_tuple)
```

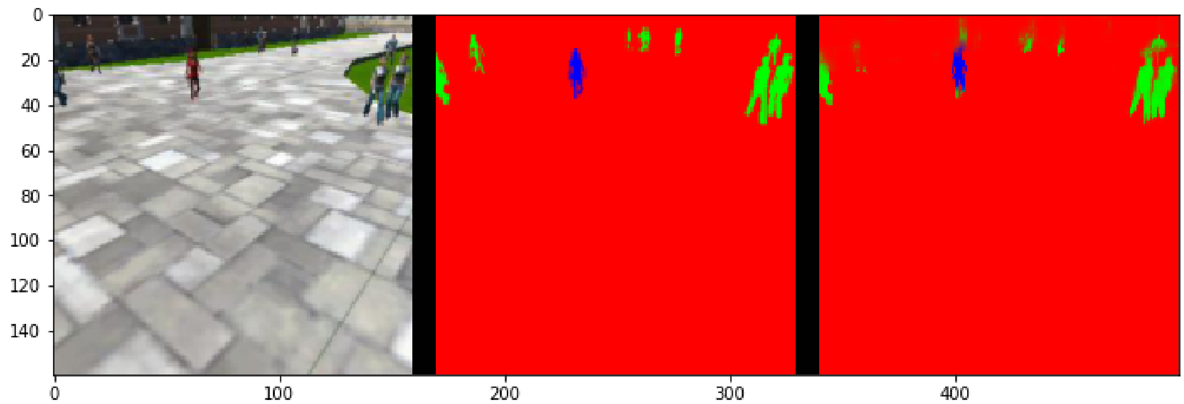


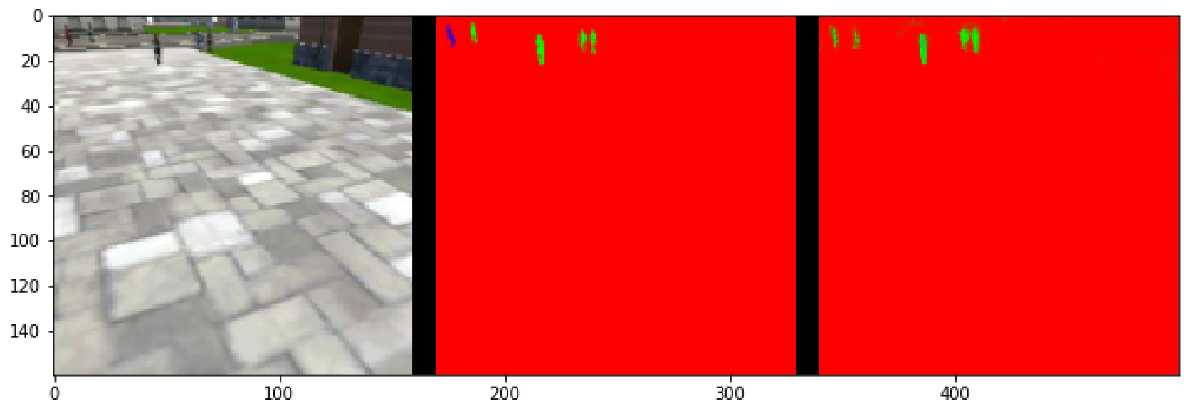


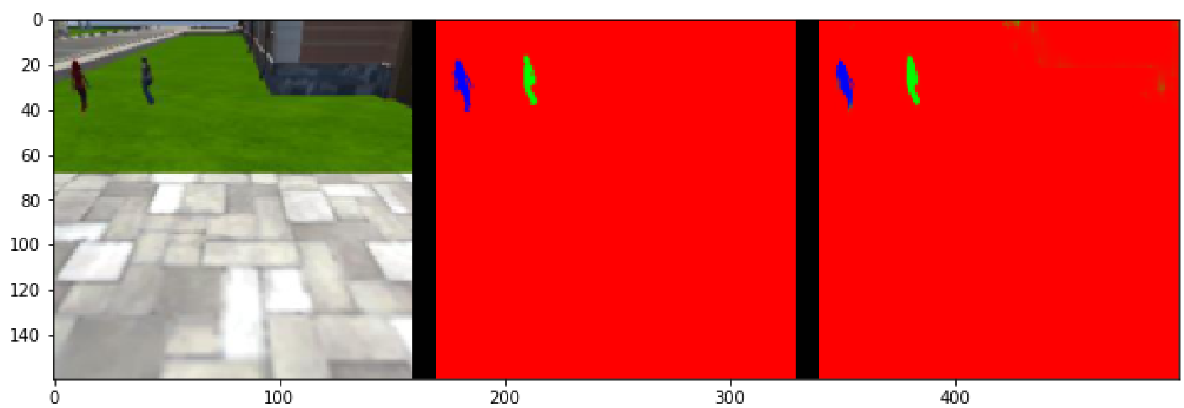


In [50]:

```
# images while at patrol with target
im_files = plotting_tools.get_im_file_sample('sample_evaluation_data','patrol_
with_targ', run_num)
for i in range(3):
    im_tuple = plotting_tools.load_images(im_files[i])
    plotting_tools.show_images(im_tuple)
```







Evaluation

Evaluate your model! The following cells include several different scores to help you evaluate your model under the different conditions discussed during the Prediction step.

In [51]:
```
# Scores for while the quad is following behind the target.
true_pos1, false_pos1, false_neg1, iou1 = scoring_utils.score_run_iou(val_foll
owing, pred_following)
```

```
number of validation samples intersection over the union evaulated on 542
average intersection over union for background is 0.9933777922537484
average intersection over union for other people is 0.31698465124950037
average intersection over union for the hero is 0.8926976943743484
number true positives: 539, number false positives: 0, number false negative
s: 0
```

In [52]:
```
# Scores for images while the quad is on patrol and the target is not visable
true_pos2, false_pos2, false_neg2, iou2 = scoring_utils.score_run_iou(val_no_t
arg, pred_no_targ)
```

```
number of validation samples intersection over the union evaulated on 270
average intersection over union for background is 0.9819542457761469
average intersection over union for other people is 0.6438639418494311
average intersection over union for the hero is 0.0
number true positives: 0, number false positives: 87, number false negatives:
0
```

In [53]:
```
# This score measures how well the neural network can detect the target from f
ar away
true_pos3, false_pos3, false_neg3, iou3 = scoring_utils.score_run_iou(val_with
_targ, pred_with_targ)
```

```
number of validation samples intersection over the union evaulated on 322
average intersection over union for background is 0.9948104388193848
average intersection over union for other people is 0.4048146735999907
average intersection over union for the hero is 0.24242440895040798
number true positives: 148, number false positives: 3, number false negative
s: 153
```

In [54]:
```
# Sum all the true positives, etc from the three datasets to get a weight for
 the score
true_pos = true_pos1 + true_pos2 + true_pos3
false_pos = false_pos1 + false_pos2 + false_pos3
false_neg = false_neg1 + false_neg2 + false_neg3

weight = true_pos/(true_pos+false_neg+false_pos)
print(weight)
```

```
0.7387096774193549
```

In [55]:
```
# The IoU for the dataset that never includes the hero is excluded from gradin
g
final_IoU = (iou1 + iou3)/2
print(final_IoU)
```

```
0.567561051662
```

In [56]:

```
# And the final grade score is
final_score = final_IoU * weight
print(final_score)
```

```
0.419262841389
```

In [ ]:

```
```