Project Objective:

The project objective is to build a segmentation network, train it, validate it, and deploy it in the Follow Me project. A drone will use the trained NN model to track and follow a single hero target using the new network pipeline.

Project Steps:

Step:1 Data Collection

A starting dataset of 4131 images with mask for training and 1184 images with mask for validation was provided. An additional of 3679 images were collected using the QuadSim simulator (by Unity). A brief description of the simulator is given below:

The dialogue box in the simulator presents with two choices:

(1) DL Training and

(2) Follow Me!.

DL Training is used to gather the training data. The Follow Me! option is to evaluate the quality of trained network.

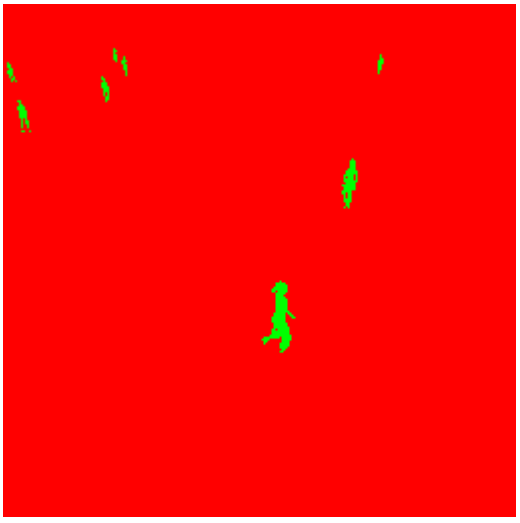
Step:2 Image Preprocessing

Before the network is trained, the images undergoes a preprocessing step. The preprocessing step transforms the depth masks from the simulator, into binary masks suitable for training a neural network. It also converts the images from .png to .jpeg to create a reduced sized dataset

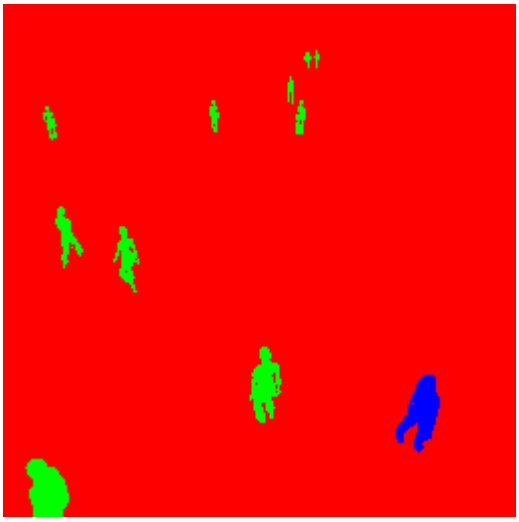
Data Sample:

1. Without target/human being in the back ground


2. With human being in the background

3. With target in the picture

Step:3 FCN Layers

Convolutional Network are Neural Networks that share their parameters across space. A patch/kernel of specific depth is being convoluted to input image/layer pixels to generate each layer in this network. CNN learns to recognize basic lines and curves, then shapes and blobs, and then increasingly complex objects within the image. A Fully Convolutional Network(FCN) is a special type of convolutional network which preserve the spatial information throughout the image space by decoding or de-convoluting the encoded or convoluted network.

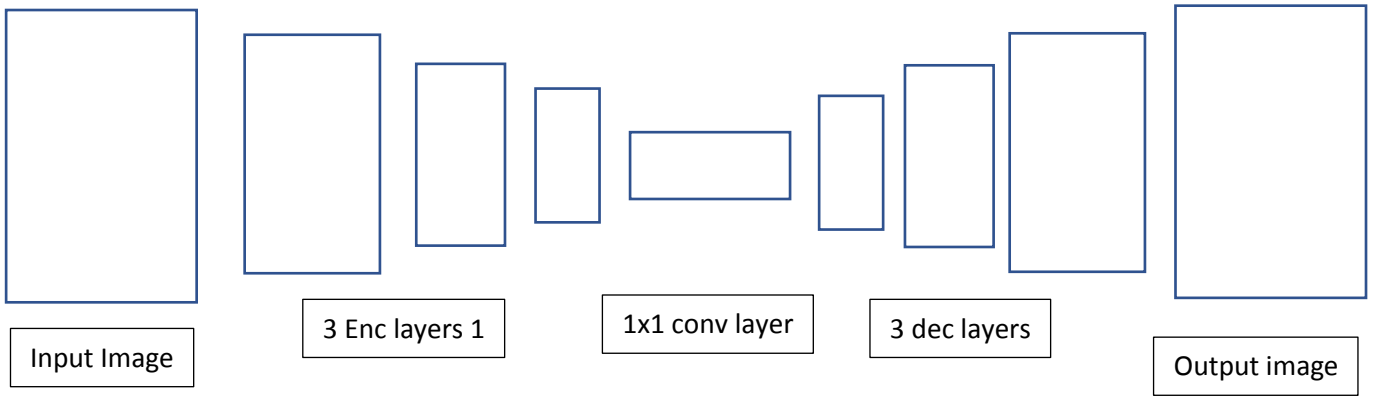


Fig. FCN with 3 layers of encoder and decoder

Step:4 Build the Model

In the following cells, an FCN is built 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).

Encoder Block

Encoder block includes a separable convolution layer using the `separable_conv2d_batchnorm()` function. The `filters` parameter defines the size or depth of the output layer.

```
def encoder_block(input_layer, filters, strides):

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

Separable Convolutions

Separable convolutions, also known as depthwise separable convolutions, comprise of a convolution performed over each channel of an input layer and followed by a 1x1 convolution that takes the output channels from the previous step and then combines them into an output layer. This is different than regular convolutions, mainly because of the reduction in the number of parameters. The 1x1 convolution layer in the FCN, however, is a regular convolution. Each includes batch normalization with the ReLU activation function applied to the layers.

Definition:

```
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
```

Decoder Block

The decoder block is comprised of three parts:

- A bilinear upsampling layer using the upsample_bilinear() function. The factor for upsampling is set to 2.
- A layer concatenation step. This step is similar to skip connections. It is used to 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.

```
def decoder_block(small_ip_layer, large_ip_layer, filters):

    # TODO Upsample the small input layer using the bilinear_upsample() function.
    upsampled_layer = bilinear_upsample(small_ip_layer)
    # TODO Concatenate the upsampled and large input layers using layers.concatenate
    concatenated_layer = layers.concatenate([upsampled_layer, large_ip_layer])
    # TODO Add some number of separable convolution layers
    output_layer = separable_conv2d_batchnorm(concatenated_layer, filters)
    return output_layer
```

Bilinear Upsampling

Bilinear upsampling is a resampling technique that utilizes the weighted average of four nearest known pixels, located diagonally to a given pixel, to estimate a new pixel intensity value. The weighted average is usually distance dependent. The following helper function implements the bilinear upsampling layer. Upsampling was done by a factor of 2 for decoding.
Definition:

```
def bilinear_upsample(input_layer):
    output_layer = BilinearUpSampling2D((2,2))(input_layer)
    return output_layer
```

Model

There are three steps:

- Add encoder blocks to build the encoder layers.
- Add a 1x1 Convolution layer using the conv2d_batchnorm() function. 1x1 Convolutions uses a kernel and stride of 1.
- Add decoder blocks for the decoder layers.

1x1 Convolution layer

Classic covnets are linear classfiers. But if we add 1x1 convolution in the middle, suddenly we have a mini neural network running over the patch instead of a linear classifier. Inter-sparsing your convolution with 1x1 conv. is a very inexpensive way to make our models deeper and have more parameters. Computation also becomes easier as well.
when we wish to feed the output of a convolutional layer into a fully connected layer, we flatten it into a 2D tensor. This results in the loss of spatial information, because no information about the location of the pixels is preserved.
We can avoid that by using 1x1 convolutions.
A 1x1 convolution is essentially convolving with a set of filters of dimensions:

1x1xfilter_size (HxWxD),
stride = 1, and
zero (same) padding.

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

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

def fcn_model(inputs, num_classes):
    # TODO Add Encoder Blocks.
    enc_layer_1 = encoder_block(inputs, filters = 16,strides = 2)
    enc_layer_2 = encoder_block(enc_layer_1, filters = 32, strides = 2)
    enc_layer_3 = encoder_block(enc_layer_2, filters = 64, strides = 2)

    # TODO Add 1x1 Convolution layer using conv2d_batchnorm().
    conv1x1_layer = conv2d_batchnorm(enc_layer_3, filters=64, kernel_size=1, strides=1)
    # TODO: Add the same number of Decoder Blocks as the number of Encoder Blocks
    dec_layer_3 = decoder_block(conv1x1_layer, enc_layer_2, filters = 64)
    dec_layer_2 = decoder_block(dec_layer_3, enc_layer_1, filters = 32)
    dec_layer_1 = decoder_block(dec_layer_2, inputs, filters = 16)
    x = dec_layer_1    # The function returns the output layer of your model. "x" is the
final layer obtained from the last decoder_block()
    return layers.Conv2D(num_classes, 1, activation='softmax', padding='same')(x)
```

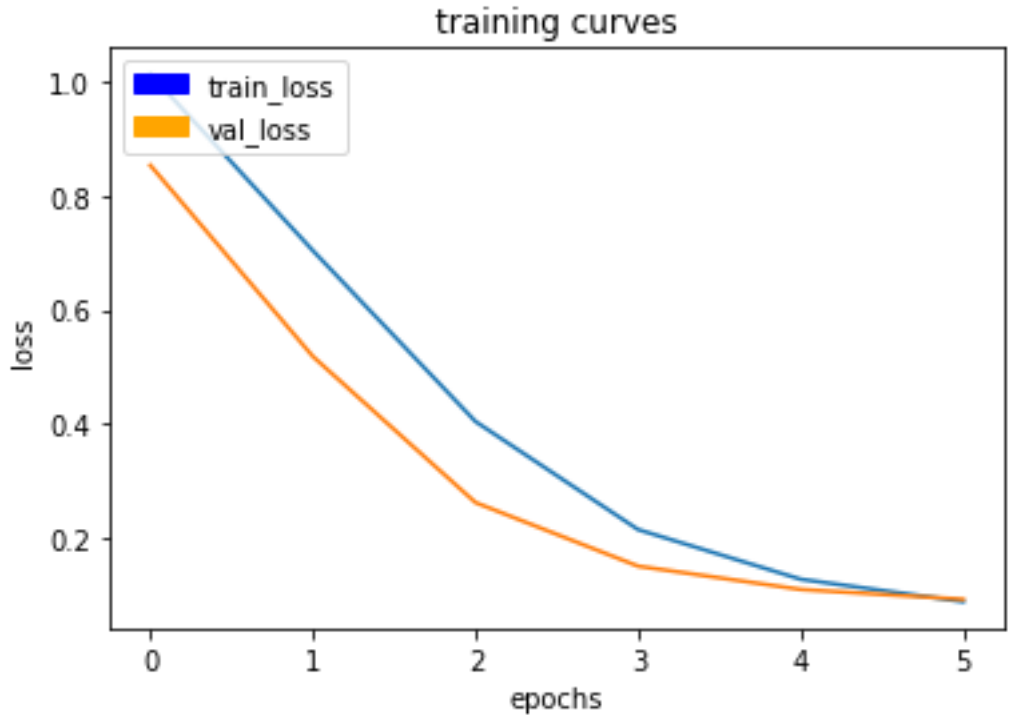
Step:4 Training

Training steps are:

- Define the Keras model and compile it for training
- Data iterators for loading the training and validation data

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

- Training analysis:



Hyperparameters

Different parameters used are:

learning_rate = 0.009
batch_size = 128
num_epochs = 60
steps_per_epoch = 61
validation_steps = 20
workers =8

Define hyperparameters:

- **Batch Size**: As we can see in the code that a model fit generator is being used to avoid duplicate data when using multiprocessing. A single output of the generator makes a single batch. It is important to define batch size as all the arrays in the output tuple of the generator has to be the same length. Usually number of training samples per images that go through the network is defined as batch size.
- **Number of Epochs:** It is defined as the total number of iterations of the sample data.
- **Steps Per Epoch**: It is calculated by total number of steps (batches of samples) to yield from generator in 1 epoch. It is typically equal to the number of samples of dataset divided by the batch size.
- **Validation Steps**: It is nothing but total number of batches of samples to yield from generator before stopping. This is similar to Steps per epoch, except validation steps is for the validation dataset.
- **Workers**: The word 'workers' here represent processes. It is calculated as the max. number of processes to spin up when using process-based threading. This can affect training speed and is dependent on hardware.

Step:5 Prediction

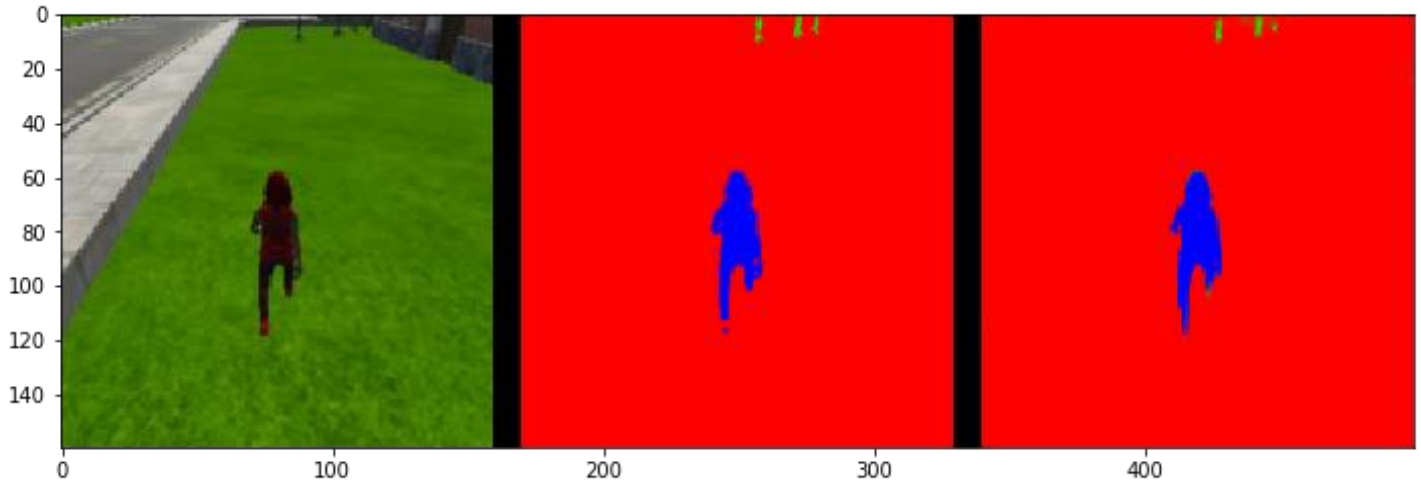
These predictions can be compared to the mask images, which are the ground truth labels, to evaluate how well your model is doing under different conditions.

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

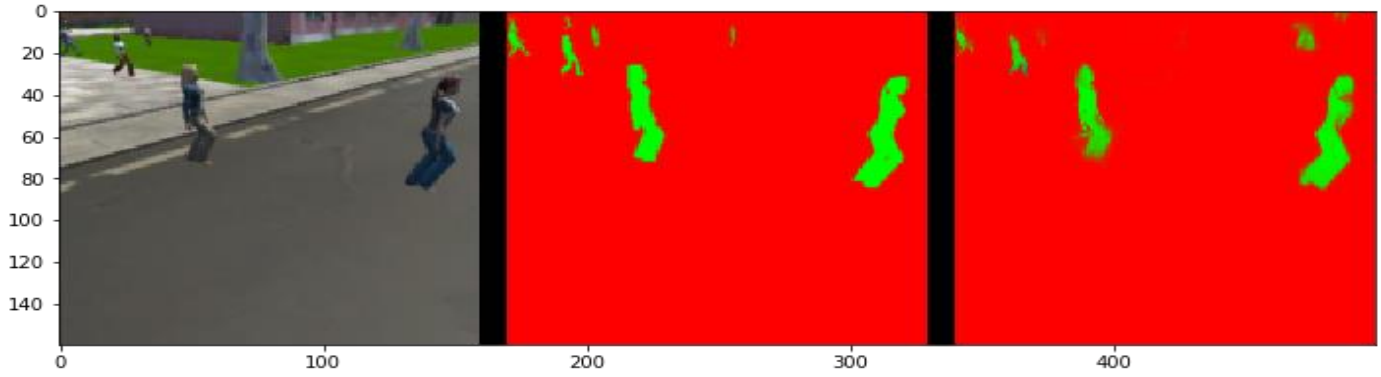
- **patrol_with_targ**: Test how well the network can detect the hero from a distance.
- **patrol_non_targ**: Test how often the network makes a mistake and identifies the wrong person as the target.
- **following_images**: Test how well the network can identify the target while following them.

Results:

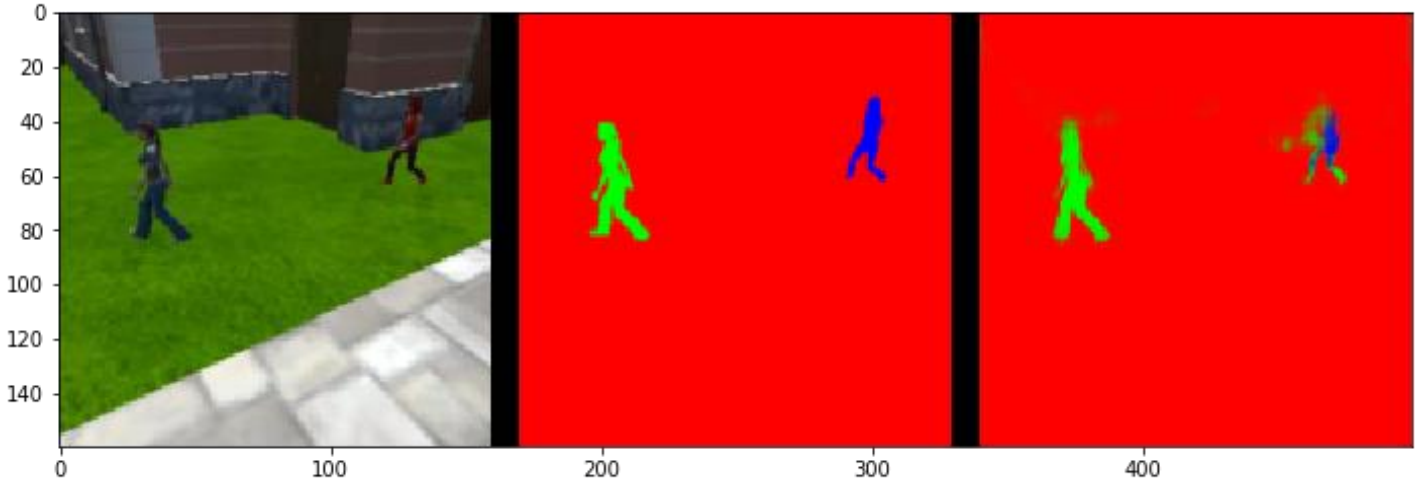
images while following the target



images while at patrol without target



images while at patrol with target



Training Improvement chart:

Training Attempt #	No. of Images in training set	Hyper Parameter Set	Prediction performance			Final Avg. IoU (Score), Lesson Learned(LL)
			Quad Following Target	Quad On Patrol, Target Not Visible	Quad On Patrol, Target Far Away	
1.	4131	Learning rate: 0.01 Batch_size:128 Epochs:30 Steps_per_epoch: 200 Validation_steps: 50	Avg. IoU of Target: 0.87 True Positives:539 False Positives:0 False Negatives:0	Avg. IoU of Target:0.0 True Positives:0 false Positives:135 False Negatives:0	Avg. IoU of Target: True Positives: False Positives: False Negatives:	0.366, LL: No. of training data need to be increased Or vary other parameters
2.	4158 (27 image added, but no relevant mask added)	Learning rate: 0.008 Batch_size:100 Epochs:20 Steps_per_epoch: 100 Validation_steps:50	Avg. IoU of Target:0.012 True Positives:127 False Positives:0 False Negatives:412	Avg. IoU of Target:0.0 True Positives:0 false Positives:1 False Negatives:0	Avg. IoU of Target:0.0003 True Positives:3 False Positives:0 False Negatives:298	0.0009, LL: Masks are must with images for training
3.	7810 (3679 images and mask added)	Learning rate: 0.01 Batch_size:120 Epochs:20 Steps_per_epoch: 65 Validation_steps:10	Avg. IoU of Target:0.775 True Positives:521 False Positives:0 False Negatives:18	Avg. IoU of Target:0.0 True Positives:0 false Positives:28 False Negatives:0	Avg. IoU of Target:0.15 True Positives:98 False Positives:4 False Negatives:203	0.328, LL: Epochs need to be increased

4	7810	Learning rate: 0.01 Batch_size:128 Epochs:30 Steps_per_epoch: 61 Validation_steps: 9	Avg. IoU of Target:0.84 True Positives:539 False Positives:0 False Negatives:0	Avg. IoU of Target:0.0 True Positives:0 false Positives:19 False Negatives:0	Avg. IoU of Target:0.16 True Positives:107 False Positives:1 False Negatives:194	0.38 LL: Lower Learning rate And more Epoch needed
5	7810	Learning rate: 0.009 Batch_size:128 Epochs:50 Steps_per_epoch: 61 Validation_steps:9	Avg. IoU of Target:0.88 True Positives:539 False Positives:0 False Negatives:0	Avg. IoU of Target:0.0 True Positives:0 false Positives:42 False Negatives:0	Avg. IoU of Target:0.23 True Positives:125 False Positives:1 False Negatives:176	0.42

Results:

- A Fully Convolutional Neural Network model was developed which consists of 3 hidden layers of the encoders, one 1x1 convolution layer and 3 hidden layers of the decoders.
- Segmentation technique: semantic
- Training and validation sets:
  No. of training images and mask: 7810
  No. of validation images and mask: 2504
- Optimization of the network model using following hyper-parameters (final values):
  learning_rate = 0.009
  batch_size = 128
  num_epochs = 60
  steps_per_epoch = 61
  validation_steps = 20
  workers = 8
- Accuracy measurement of the NN using Intersection over Union IoU metric (IoU): 0.42

Future Enhancement:

The project still has scopes to improve the prediction and tune hyper parameters. Those parameters should further be tuned to obtain 0 false positives and 0 false negatives. Training model could be tuned further to optimized to ensure detecting target object while in patrol mode more accurately.

The training data is used to train the FCN model to identify human shape and find the difference between target(and) non-target. In order to follow another object (dog, cat, car, etc.) instead of a human the model needs to be trained with the pictures or training data of the different target object.