Write-up for the Project 4: Follow Me

Preparations

Layers Definitions

We start with the code changes required for the definition of the encoder and decoder blocks. First the **encoder block**:

```
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
```

We simply create a separable 2d convolution layer with batch normalisation using the convenience function already defined in the utilities include.

For the **decoder**:

```
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
    merge_layer = layers.concatenate([upsample_layer, large_ip_layer])

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

    return output_layer
```

We upsample the smaller layer using the convenience function from the utils include, then merge the result with the larger layer and apply a separable 2d convolution with batch normalisation with stride 1 (we keep the same resulting image size) but applying a number of filters that is provided as a parameter – and we will aim to reduce the dimensionality of that axis.

For the actual model we will discuss a number of tests and the results and investigate the impact of the size of the layers or their number on the performance of the network.

Small Changes to Utils

The callback function that is used to plot the training results is a little distracting as it draws the diagram after each epoch and soon the output becomes very messy. I have updated the function so that the plot is produced every 10 epochs:

```
    def on_epoch_end(self, epoch, logs=None):
        if logs is not None:
            for k in self.params['metrics']:
                if k in self.totals:
                    # Make value available to next callbacks.
                    logs[k] = self.totals[k] / self.seen

            self.hist_dict['loss'].append(logs['loss'])

            if 'val_loss' in self.params['metrics']:
                self.hist_dict['val_loss'].append(logs['val_loss'])
            # show graph every 10 epochs
            if (epoch+1) % 10 == 0:
                train_val_curve(self.hist_dict['loss'], self.hist_dict['val_loss'])
            #else:
            #    train_val_curve(self.hist_dict['loss'])
```

Also I have changed the train_val_curve method to print a larger version of the graph, and to reduce the range on the Y axis to 0 – 0.1 so that we only look at the range relevant for the end of the training. Otherwise the curves will have a range too big (they will start at about 0.9 for the first epoch) so the values for the last epochs will be crowded at the bottom of the graph and will be very hard to analyse the data efficiently:

```
def train_val_curve(train_loss, val_loss=None):
    plt.figure(figsize=(16,12))
    plt.ylim(ymin=0, ymax=0.1) # so that we focus on the finer details of training
    train_line = plt.plot(train_loss, label='train_loss')
    train_patch = mpatches.Patch(color='blue',label='train_loss')
    handles = [train_patch]
    if val_loss:
        val_line = plt.plot(val_loss, label='val_loss')
        val_patch = mpatches.Patch(color='orange',label='val_loss')
        handles.append(val_patch)

    plt.legend(handles=handles, loc=2)
    plt.title('training curves')
    plt.ylabel('loss')
    plt.xlabel('epochs')
    plt.show()
```

Hyper Parameters

To conveniently work for the batch size and the steps per epoch I have defined a variable:

```
training_set_size = 4000
```

That will hold the number of training samples used in the respective run (in this case I use the provided training set of approximately 4000 images).

Then the steps per epoch are calculated as:

```
steps_per_epoch = training_set_size / batch_size + 1
```

Thus, giving us only the choice of setting the batch size. For the validation we use:

```
validation_steps = steps_per_epoch / 4
```

Deciding the batch_size

When doing some testing with various model sizes we encountered from time to time memory allocation issues when batch size was larger than 150. So we will consider empirically as an upper limit for the training environment we have.

In terms of training speed we have not seem major impacts from using a larger batch size (at least not on the GPU server). So the decision ultimately will come from the impact on the training. There are relatively conflicting guidance about using a smaller or larger batch size in training with some advocates asking for smaller values, justified by the fact that with a smaller batch there are more updates to the weights and there is less likely to get stuck quickly in a unoptimal minimum. In my testing I have seen an improvement in the validation loss (val_loss decreasing) numbers when using a batch_size of 50 compared with 150. This suggests that, at least in this case of the model we are implementing, a smaller batch size helps with reducing overfitting, especially when the model becomes larger and having more parameters. So I have used consistently a **batch size of 50** in this project for all models and training sets.

Learning Rate

Empirically we noticed that using a learning rate of 0.01 is too aggressive and results in very fast decrease in the training loss but with a very slow follow-up of the validation loss. So we have settled for a **training rate of 0.001 to start with**.

In addition, we have found it useful to implement a decreasing learning rate callback and add it in the model definition using the Keras ReduceLROnPlateau function as follows:

```
reduceLR = keras.callbacks.ReduceLROnPlateau(monitor='val_loss',
                                             factor=0.5,
                                             patience=5,
                                             verbose=1,
                                             mode='auto')
```

This reduces the learning rate by half (factor=0.5) every time when the metric `val_loss` is stagnant for 5 epochs (patience=5). That means if for 5 epoch the minimum value obtained for the val_loss is not improved the learning rate is decreased. This in principle helps with the progress of the learning and allows further improvement of training and validation when they plateau. We might revisit later the fact that we use the `val_loss` metric for the decision to reduce the learning rate depending on the result we obtain in training.

This callback is added to the list that is passed to the model when training:

```
callbacks = [reduceLR, logger_cb]
```

Number of epochs

We found that the majority of models train quickly in the first 25 epochs or so and then slowly continue to train (depending on the model and amount of data). But we did not see any significant benefit in

using more than 50 epochs in training. So, in general we have used 50 epochs and if a particular model training was done for more than 50 epochs I will indicate it specifically in the text.

Overview

Here is the code of the hyper parameters as used in the majority of the trainings for this project:

```
learning_rate = 0.001
training_set_size = 4000
batch_size = 50
num_epochs = 50
steps_per_epoch = training_set_size / batch_size + 1
validation_steps = steps_per_epoch / 4
workers = 2
```

Training Data

We have created a larger set of data for training consisting of 15,277 images produced in multiple runs around the map. The original training set (aprox 4000 images) was kept in a directory called train-orig while the new data has been paced in data/train. When training the model we indicate in the data iterator one or the other directory depending if we want to do the training on the smaller or larger set. We will indicate specifically bellow which set we have used in the training. When the larger training set is used the variable `training_set_size` needs to be updated too before the notebook is run.

Models

We started the project by bashing some large models and trying to see how it goes and we realised that we have some significant problems with overfitting the data and in general about how the model behaves. So, we decided to have a more pragmatic approach, starting with a simple model and improving it in various ways, then understanding the impact of those changes and further refining the model towards a better performance.

Start Model (Model1)

We start our investigation with a small model that has the following structure:

- 2 encoder blocks
- A 1x1 2d conv layer to reduce dimensionality
- 2 decoder blocks
- Final 2d convolution with depth 3 and softmax activation

Here is the code with the definition of the model:

```
def fcn_model(inputs, num_classes):

    # TODO Add Encoder Blocks.
    # Remember that with each encoder layer, the depth of your model
    # (the number of filters) increases.
    #                                                    input: 160x160x3
    en1 = encoder_block(inputs, filters=12, strides=2)     # en1: 80x80x12
```

```
    en2 = encoder_block(en1, filters=48, strides=2)        # en2: 40x40x48

    # TODO Add 1x1 Convolution layer using conv2d_batchnorm().
    mid = conv2d_batchnorm(en2, filters=48, kernel_size=1, strides=1)
    # mid: 40x40x48

    # TODO: Add the same number of Decoder Blocks as the number of Encoder Blocks
    de2 = decoder_block(mid, en1, filters=48)              # de2: 80x80x48
    de1 = decoder_block(de2, inputs, filters=12)           # de1: 160x160x12

    # The function returns the output layer of your model
    return layers.Conv2D(num_classes, 1, activation='softmax', padding='same')(de1)
```

The diagram of the model is the following:

Layer (type)	Output Shape	Param #	
input_1 (InputLayer)	(None, 160, 160, 3)	0	
separable_conv2d_keras_1 (Se	(None, 80, 80, 12)	75	en1
batch_normalization_1 (Batch	(None, 80, 80, 12)	48	
separable_conv2d_keras_2 (Se	(None, 40, 40, 48)	732	en2
batch_normalization_2 (Batch	(None, 40, 40, 48)	192	
conv2d_1 (Conv2D)	(None, 40, 40, 48)	2352	mid
batch_normalization_3 (Batch	(None, 40, 40, 48)	192	
bilinear_up_sampling2d_1 (Bi	(None, 80, 80, 48)	0	de2
concatenate_1 (Concatenate)	(None, 80, 80, 60)	0	
separable_conv2d_keras_3 (Se	(None, 80, 80, 48)	3468	
batch_normalization_4 (Batch	(None, 80, 80, 48)	192	
bilinear_up_sampling2d_2 (Bi	(None, 160, 160, 48)	0	de1
concatenate_2 (Concatenate)	(None, 160, 160, 51)	0	
separable_conv2d_keras_4 (Se	(None, 160, 160, 12)	1083	
batch_normalization_5 (Batch	(None, 160, 160, 12)	48	
conv2d_2 (Conv2D)	(None, 160, 160, 3)	39	

Total params: 8,421
Trainable params: 8,085
Non-trainable params: 336

We can see the dimensions of the layers and the number of parameters in the above diagram. Each encoding layer reduces the W and H of the images due to the `strides=2` parameter in the call of the encoder block. Each convolution is followed by a batch normalisation layer that preserves the dimensions and adds minimal numbers of training parameters.

The mid layer includes a 1x1 convulsion with stride 1 thus preserving the original shape of the image, but we can control with the filter parameter the size of the output and reduce it (the depth) in order to simplify processing down flow and reduce overfitting (due to too many training parameters). This obviously also has an impact on the training speed. In the case of this particular model we have chosen not to reduce the depth of the input because it is already shallow but we will perform this reduction later when the models are getting more complex and the size of the layers increase.

Each decoder group is formed by an upsampling layer that doubles the W and H of the input (as expected) and preserves the number of channels. We then concatenate the output from the upsampling layer with the output of the decoder layer with the same image size (decoder 2 uses encoder 1 output and decoder 1 uses the original image). The concatenated output is then passed through a separable 2d convolution this time with the stride 1 so that they maintain the image size (unlike the encoder layers) but where we decrease progressively the number of channels relatively in line with the number of channels in the encoder layers. The convolution layer is followed by a batch normalization layer as a final step in the encoder module, a layer that leaves unchanged the dimensionality of the data.

We can see that overall this model has 8,085 trainable parameters – a very low number to produce any notable performance, but a good start to evaluate the task as hand.

Let's train the model using the original (4000 images) training set. The code for the training will be the same for all models unless mentioned specifically for that particular model:

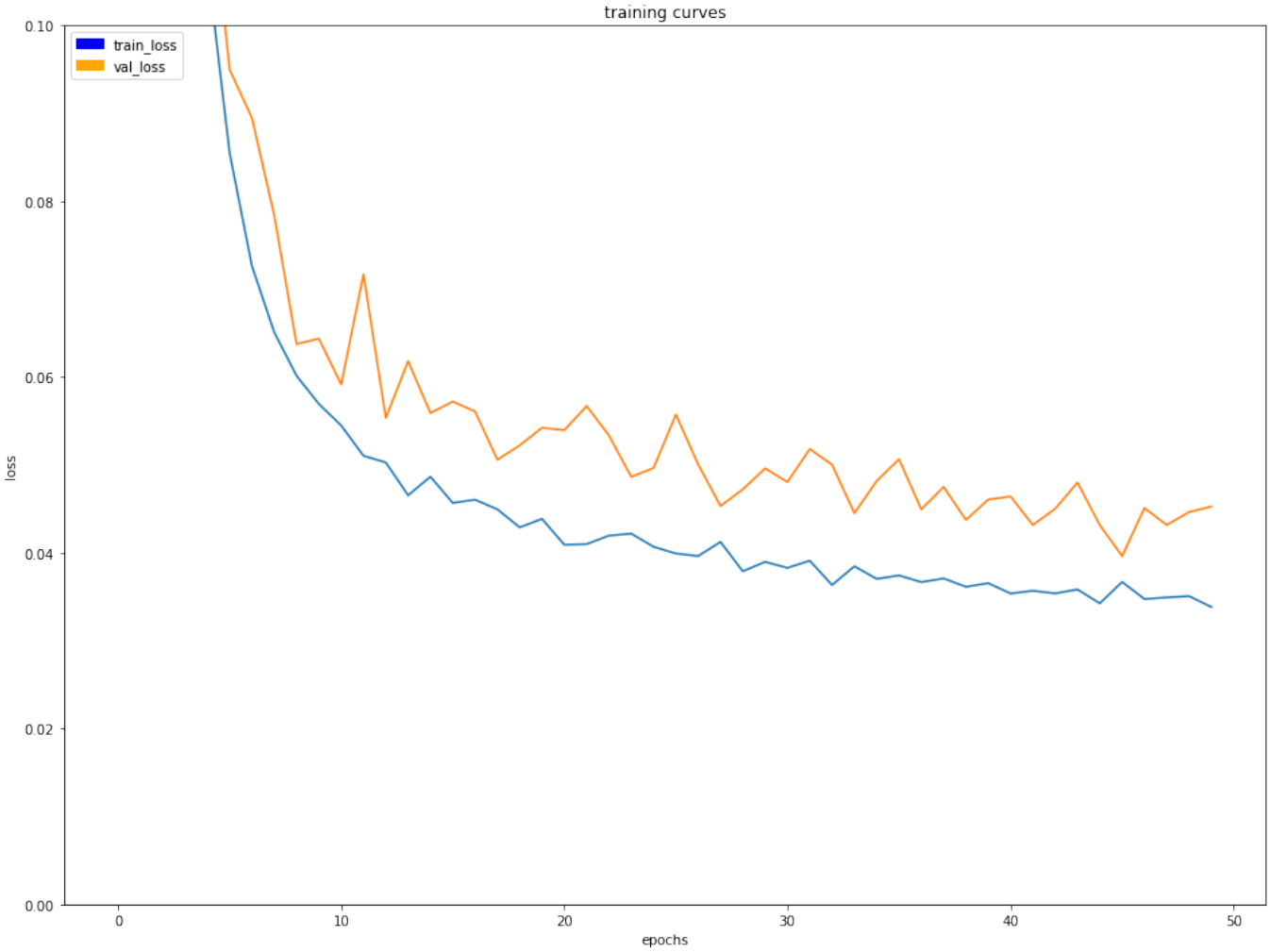
```
# Data iterators for loading the training and validation data
train_iter = data_iterator.BatchIteratorSimple(batch_size=batch_size,
                                               data_folder=os.path.join('..', 'data', 'train-orig'),
                                               image_shape=image_shape,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

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

logger_cb = plotting_tools.LoggerPlotter()
reduceLR = keras.callbacks.ReduceLROnPlateau(monitor='val_loss',
                                             factor=0.5,
                                             patience=5,
                                             verbose=1,
                                             mode='auto')
#callbacks = [logger_cb]
callbacks = [reduceLR, 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 validate on
                    callbacks=callbacks,
                    workers = workers)
```

We trained for 50 epochs. The full result of the training can be seen in the provided attachments in the file `aws_model_training_model1_run1.html`. The following diagram shows the training and validation loss as reported from Keras (I've changed the code for the representation of the training curves so that the Y axis shows only the range 0-0.1 so that we can focus on the lower details towards the end of the training):



The final training loss was 0.0338 (which was also the best) and the val_loss was 0.0453 (where a minimum of 0.0396 was produced during training).

The shape of the curves suggest that we might be able to still get some more improvements by adding more epochs and that the model does not overfit the data. Also, during training, although we have included a callback for reduction of the learning rate, this was not called meaning the choice for the learning rate was good and the combination of number of steps and learning rate was good, avoiding a steep decrease in the learning loss (which might be bad as we might end up in a local minimum).

Evaluation

The shape of the training shown above is very healthy and you can see that the training loss continues to improve even towards the end of the 50 epochs and the validation loss is closely aligned without any signs of overfitting (as expected considering the small number of parameters for the model). But the overall loss is too high meaning that most likely the model will not be very accurate in predictions.

The results of the evaluation of the model are presented below:

In [17]:
```
# Scores for while the quad is following behind the target.
true_pos1, false_pos1, false_neg1, iou1 = scoring_utils.score_run_iou(val_following, pred_following)
```

```
number of validation samples intersection over the union evaulated on 542
average intersection over union for background is 0.9900945692793426
average intersection over union for other people is 0.20551245273120788
average intersection over union for the hero is 0.6579089330174628
number true positives: 539, number false positives: 0, number false negatives: 0
```

In [18]:
```
# 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_targ, pred_no_targ)
```

```
number of validation samples intersection over the union evaulated on 270
average intersection over union for background is 0.9721309089029142
average intersection over union for other people is 0.41189500610000124
average intersection over union for the hero is 0.0
number true positives: 0, number false positives: 66, number false negatives: 0
```

In [19]:
```
# This score measures how well the neural network can detect the target from far 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 0.9940476458577336
average intersection over union for other people is 0.29963445335428185
average intersection over union for the hero is 0.09397528040578904
number true positives: 93, number false positives: 3, number false negatives: 208
```

In [20]:
```
# 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)
```

```
0.6952695269526953
```

In [21]:
```
# The IoU for the dataset that never includes the hero is excluded from grading
final_IoU = (iou1 + iou3)/2
print(final_IoU)
```

```
0.375942106712
```

In [22]:
```
# And the final grade score is
final_score = final_IoU * weight
print(final_score)
```

```
0.261381090695
```

This very simple mode still shows a remarkably good result for the identification of the hero in follow-up mode (from behind the target) but has very bad results for the cases where the hero is far away.

It's obvious that the model can be improved, and we have 3 options:

1. Train with more data
2. Increase the depth of the existing layers to increase the number of parameters
3. Add additional layers to increase the number of parameters

For the time being the option 1 is not useful as the training curve indicates that the model would not benefit from additional data. We'll try to approach the other 2 options and see the impact on the performance of the model.

Summary

	Model 1
Encoders	2

	Model 1
No of channels	12, 48
Mid layer channels	48
Decoders	2
No of channels	48, 12
Parameters	8,085
Training time (4000 img)	27[min]
Loss	0.0338 (0.0338)
Val_loss (min)	0.0453 (0.0396)
Evaluation	
Weight	0.6953
IoU	0.3759
Final score	0.2614

Increase the Depth of Layers (Model2)

The first change we will do is to increase the depth of the encoder and decoder layers. In the first model the layers were (encoder 1, encoder 2, middle, decoder 2, decoder 1): 12, 48, 48, 48, 12. We will double the number of channels for those layers (including the middle). The code for the model is:

```
def fcn_model(inputs, num_classes):

    # TODO Add Encoder Blocks.
    # Remember that with each encoder layer, the depth of your model
    # (the number of filters) increases.
    #                                                  input: 160x160x3
    en1 = encoder_block(inputs, filters=24, strides=2)     # en1: 80x80x24
    en2 = encoder_block(en1, filters=96, strides=2)        # en2: 40x40x96

    # TODO Add 1x1 Convolution layer using conv2d_batchnorm().
    mid = conv2d_batchnorm(en2, filters=96, kernel_size=1, strides=1)
    # mid: 40x40x96

    # TODO: Add the same number of Decoder Blocks as the number of Encoder Blocks
    de2 = decoder_block(mid, en1, filters=96)              # de2: 80x80x96
    de1 = decoder_block(de2, inputs, filters=24)           # de1: 160x160x24

    # The function returns the output layer of your model
    return layers.Conv2D(num_classes, 1, activation='softmax', padding='same')(de1)
```

The model is exactly the same as before so all comments that I have made in the previous section are valid. The only change is that the layers have a deeper channel structure, thus accommodating more training parameters and therefore providing more learning capability. Since the number of channels in the mid section of the model is still modest the mid convolution still does not reduces the number of channels thus preserving the 96 channels from the second encoding layer.

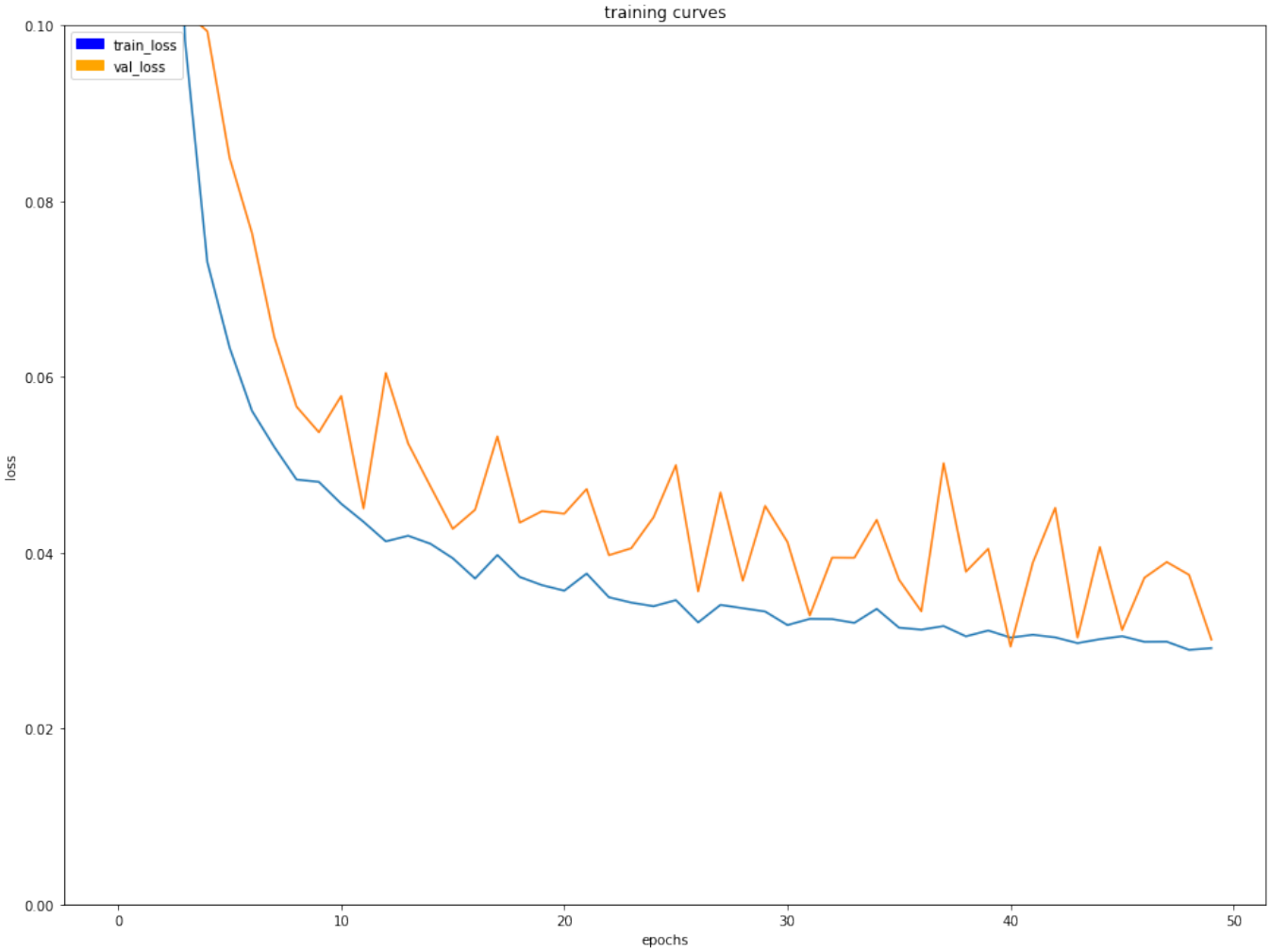
The printout of the model is shown bellow:

Layer (type)	Output Shape	Param #
input_1 (InputLayer)	(None, 160, 160, 3)	0
separable_conv2d_keras_1 (Se	(None, 80, 80, 24)	123
batch_normalization_1 (Batch	(None, 80, 80, 24)	96
separable_conv2d_keras_2 (Se	(None, 40, 40, 96)	2616
batch_normalization_2 (Batch	(None, 40, 40, 96)	384
conv2d_1 (Conv2D)	(None, 40, 40, 96)	9312
batch_normalization_3 (Batch	(None, 40, 40, 96)	384
bilinear_up_sampling2d_1 (Bi	(None, 80, 80, 96)	0
concatenate_1 (Concatenate)	(None, 80, 80, 120)	0
separable_conv2d_keras_3 (Se	(None, 80, 80, 96)	12696
batch_normalization_4 (Batch	(None, 80, 80, 96)	384
bilinear_up_sampling2d_2 (Bi	(None, 160, 160, 96)	0
concatenate_2 (Concatenate)	(None, 160, 160, 99)	0
separable_conv2d_keras_4 (Se	(None, 160, 160, 24)	3291
batch_normalization_5 (Batch	(None, 160, 160, 24)	96
conv2d_2 (Conv2D)	(None, 160, 160, 3)	75

Total params: 29,457
Trainable params: 28,785
Non-trainable params: 672

It’s obvious that the increase has increased the number of parameters by approximately three and a half times. That should give us in principle better results during training and validation.

The code for the training is exactly the same and the results are:



We still have a good training curve with the validation loss tightly following the training loss and with performance improving across the epochs. By the look of the graph the training might have beneficiated from some additional epochs and if we would increase that parameter we might be able to squeeze some additional performance from this model although it is worth noticing that the learning rate has been adjusted 3 times: at epoch 22 to 0.0005, at epoch 38 to 0.00025 and at epoch 47 to 0.000125. This might suggest that a next run for higher number of epoch might be better off if we start with a smaller learning rate at the beginning (ex. 0.0005). We will not do that now as we will tweak other parameters before deciding to do a training with more epochs.

The final loss is 0.0292 (best score was 0.0290 in epoch 49) and the validation loss is 0.0301 (with scores of 0.0293 being recorded during training). This results are significantly better than the previous model and this is also visible in the evaluation.

Evaluation

In [17]:
```
# Scores for while the quad is following behind the target.
true_pos1, false_pos1, false_neg1, iou1 = scoring_utils.score_run_iou(val_following, pred_following)
```

```
number of validation samples intersection over the union evaulated on 542
average intersection over union for background is 0.9932585957428202
average intersection over union for other people is 0.2422662580532505
average intersection over union for the hero is 0.8222454216330001
number true positives: 539, number false positives: 0, number false negatives: 0
```

In [18]:
```
# 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_targ, pred_no_targ)
```

```
number of validation samples intersection over the union evaulated on 270
average intersection over union for background is 0.9768848084723873
average intersection over union for other people is 0.511309923112217
average intersection over union for the hero is 0.0
number true positives: 0, number false positives: 103, number false negatives: 0
```

In [19]:
```
# This score measures how well the neural network can detect the target from far 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 0.9947635009015158
average intersection over union for other people is 0.3362814669033054
average intersection over union for the hero is 0.14246332768145992
number true positives: 120, number false positives: 4, number false negatives: 181
```

In [20]:
```
# 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)
```

```
0.6958817317845829
```

In [21]:
```
# The IoU for the dataset that never includes the hero is excluded from grading
final_IoU = (iou1 + iou3)/2
print(final_IoU)
```

```
0.482354374657
```

In [22]:
```
# And the final grade score is
final_score = final_IoU * weight
print(final_score)
```

```
0.33566159757
```

We can see that the overall score improved by 28% (from 0.2614 to 0.3357) with a 28% improvement of the IoU factor (from 0.405 to 0.489). We still have a large number of false negatives for identification of the target when far away (182 in a total set of 322 images) and the changes to the model did not seem to have any impact on the weight KPI meaning it does not change the number of false negatives or false positives. The impact of the changes in the model seems to have impact on how much more the of the image is recognized thus improving the IoU indicator.

The detail for this model are included in the document `aws_model_training_model2_run1.html` in the attachments.

Summary

	Model 1	Model 2
Encoders	2	2
No of channels	12, 48	24, 96

	Model 1	Model 2
Mid layer channels	48	96
Decoders	2	2
No of channels	48, 12	96, 24
Parameters	8,085	28,785
Training time (4000 img)	27[min]	44[min]
Loss	0.0338 (0.0338)	0.0292 (0.0290)
Val_loss (min)	0.0453 (0.0396)	0.0301 (0.0293)
Evaluation		
Weight	0.6953	0.6959
IoU	0.3759	0.4824
Final score	0.2614	0.3357

Adding Layers (Model 3)

Let's try now the impact of adding one more encoding and one more decoding layer in the model.

```
def fcn_model(inputs, num_classes):

    # TODO Add Encoder Blocks.
    # Remember that with each encoder layer, the depth of your model
    # (the number of filters) increases.
    #                                                  input: 160x160x3
    en1 = encoder_block(inputs, filters=12, strides=2)     # en1: 80x80x12
    en2 = encoder_block(en1, filters=48, strides=2)        # en2: 40x40x48
    en3 = encoder_block(en2, filters=192, strides=2)       # en3: 20x20x192

    # TODO Add 1x1 Convolution layer using conv2d_batchnorm().
    mid = conv2d_batchnorm(en3, filters=96, kernel_size=1, strides=1)
    # mid: 20x20x96

    # TODO: Add the same number of Decoder Blocks as the number of Encoder Blocks
    de3 = decoder_block(mid, en2, filters=72)              # de3: 40x40x72
    de2 = decoder_block(de3, en1, filters=48)              # de2: 80x80x48
    de1 = decoder_block(de2, inputs, filters=12)           # de1: 160x160x12

    # 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')(de1)
```

We have added one more encoder layer with 192 filters (we're increasing the number of channels by 4 roughly to accommodate the decrease of the other two dimensions HxW by 2). This time the mid layer acts as a bottleneck layer and reduces the depth from 192 to 96 to reduce the number of parameters that we need to train and positively impact the overfitting predisposition of the model. The decoder layer now reduce in half the number of channels resulted from the summation (ex. decoder 3 results from the summation of the 96 channels of the upscalled mid layer with the 48 channels of the encoder 2 – total 144 and in the last convolution layer of the decoder we reduce that to 72).

Here is the summary of the model as printed by Keras:

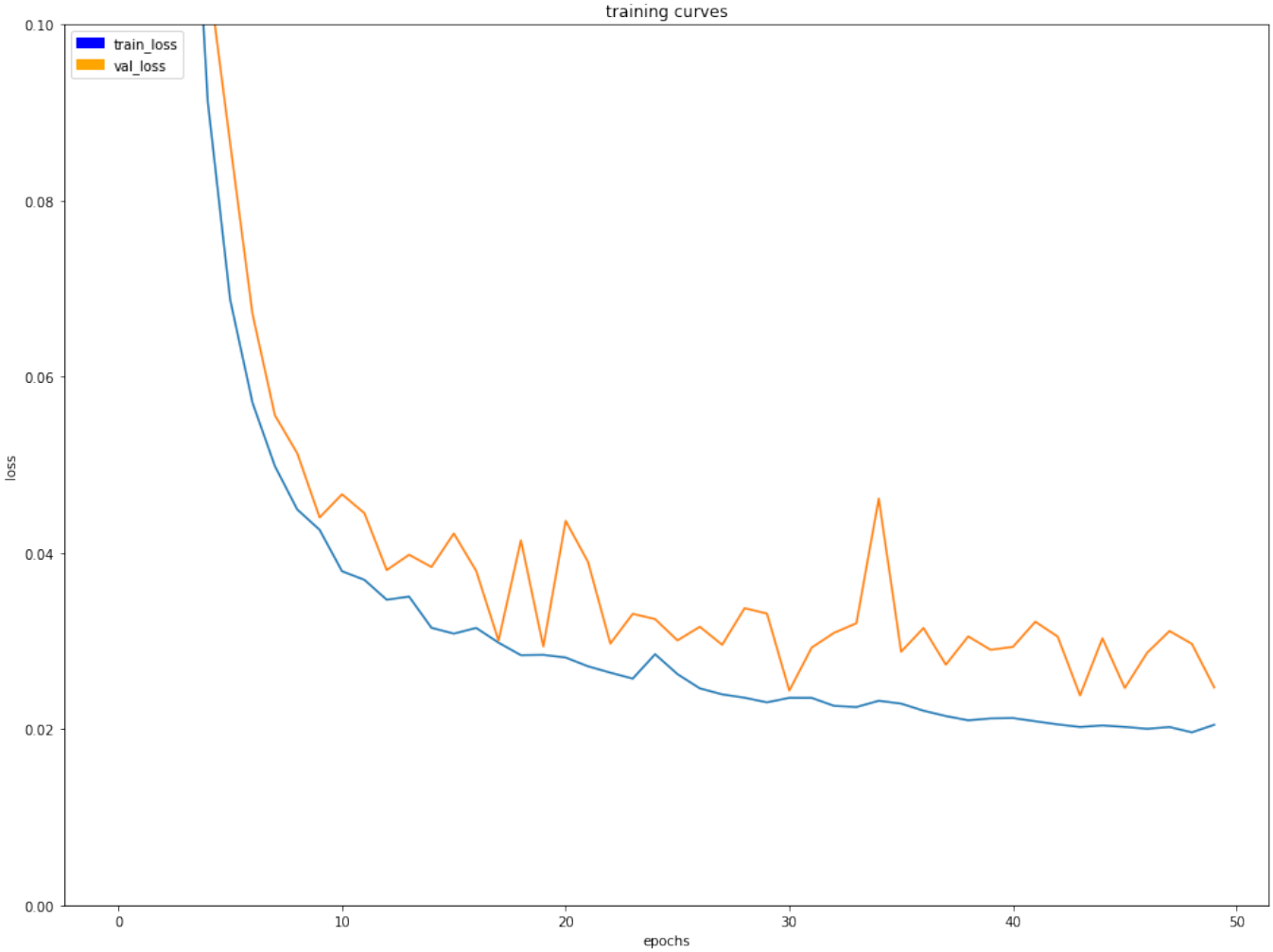
Layer (type)	Output Shape	Param #	
input_1 (InputLayer)	(None, 160, 160, 3)	0	
separable_conv2d_keras_1 (Se	(None, 80, 80, 12)	75	
batch_normalization_1 (Batch	(None, 80, 80, 12)	48	
separable_conv2d_keras_2 (Se	(None, 40, 40, 48)	732	
batch_normalization_2 (Batch	(None, 40, 40, 48)	192	
separable_conv2d_keras_3 (Se	(None, 20, 20, 192)	9840	en3
batch_normalization_3 (Batch	(None, 20, 20, 192)	768	
conv2d_1 (Conv2D)	(None, 20, 20, 96)	18528	
batch_normalization_4 (Batch	(None, 20, 20, 96)	384	
bilinear_up_sampling2d_1 (Bi	(None, 40, 40, 96)	0	
concatenate_1 (Concatenate)	(None, 40, 40, 144)	0	
separable_conv2d_keras_4 (Se	(None, 40, 40, 72)	11736	
batch_normalization_5 (Batch	(None, 40, 40, 72)	288	
bilinear_up_sampling2d_2 (Bi	(None, 80, 80, 72)	0	
concatenate_2 (Concatenate)	(None, 80, 80, 84)	0	
separable_conv2d_keras_5 (Se	(None, 80, 80, 48)	4836	
batch_normalization_6 (Batch	(None, 80, 80, 48)	192	
bilinear_up_sampling2d_3 (Bi	(None, 160, 160, 48)	0	de3
concatenate_3 (Concatenate)	(None, 160, 160, 51)	0	
separable_conv2d_keras_6 (Se	(None, 160, 160, 12)	1083	
batch_normalization_7 (Batch	(None, 160, 160, 12)	48	
conv2d_2 (Conv2D)	(None, 160, 160, 3)	39	

Total params: 48,789
Trainable params: 47,829
Non-trainable params: 960

The number of parameters has gone up to 47,829 almost double the number compared with the increased depth model and almost 6 times higher than the number of parameters in the first model. This should in principle improve our results.

We are training the model using the same 4000 images data set and the same hyper parameters as in the previous 2 cases.

The loss graph is shown below:



The shape is still good and although there is a little bit more jiggle in the validation loss compared with the previous examples there is not that bad. There seems to be a little bit of overfitting as we're finishing with a significantly lower training loss but the validation loss is very similar to the one in Model 2. We might need to increase the training data set to improve on this metric.

The training loss at the end was 0.0205 (with the best value at epoch 49: 0.0196) and the validation loss was 0.0247 (with a best value of 0.0238 being recorded in epoch 44). The fact that validation loss is not significantly better than the second model suggest that we might get some value from using a larger training set and therefore reduce the chance of overfitting (usually a larger discrepancy between training and validation sets is an indication that might be the case). We'll keep this in mind for deciding how to improve the model.

Evaluation

The evaluation of the model produces the following:

In [17]:
```
# Scores for while the quad is following behind the target.
true_pos1, false_pos1, false_neg1, iou1 = scoring_utils.score_run_iou(val_following, pred_following)
```

```
number of validation samples intersection over the union evaulated on 542
average intersection over union for background is 0.994560686125762
average intersection over union for other people is 0.3045883230652517
average intersection over union for the hero is 0.8725867562250729
number true positives: 539, number false positives: 0, number false negatives: 0
```

In [18]:
```
# 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_targ, pred_no_targ)
```

```
number of validation samples intersection over the union evaulated on 270
average intersection over union for background is 0.9834638297238424
average intersection over union for other people is 0.656054528595288
average intersection over union for the hero is 0.0
number true positives: 0, number false positives: 64, number false negatives: 0
```

In [19]:
```
# This score measures how well the neural network can detect the target from far 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 0.9956316584687024
average intersection over union for other people is 0.39004354354272114
average intersection over union for the hero is 0.11858350315486328
number true positives: 94, number false positives: 1, number false negatives: 207
```

In [20]:
```
# 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)
```

```
0.6994475138121546
```

In [21]:
```
# The IoU for the dataset that never includes the hero is excluded from grading
final_IoU = (iou1 + iou3)/2
print(final_IoU)
```

```
0.49558512969
```

In [22]:
```
# And the final grade score is
final_score = final_IoU * weight
print(final_score)
```

```
0.346635786844
```

We have a minor improvement of the overall score compared with the model 2 and this is rather surprising considering that the number of parameters is almost double. But this might be also due to the fact that this last model still has good potential when trained with a larger dataset and for longer.

The detail for this model are in the attachment `aws_model_training_model3_run1.html`.

Summary

At this moment it should be good to have an overview of the 3 models:

	Model 1	Model 2	Model 3
Encoders	2	2	3
No of channels	12, 48	24, 96	12, 48, 192
Mid layer channels	48	96	96
Decoders	2	2	3
No of channels	48, 12	96, 24	72, 48, 12
Parameters	8,085	28,785	47, 829

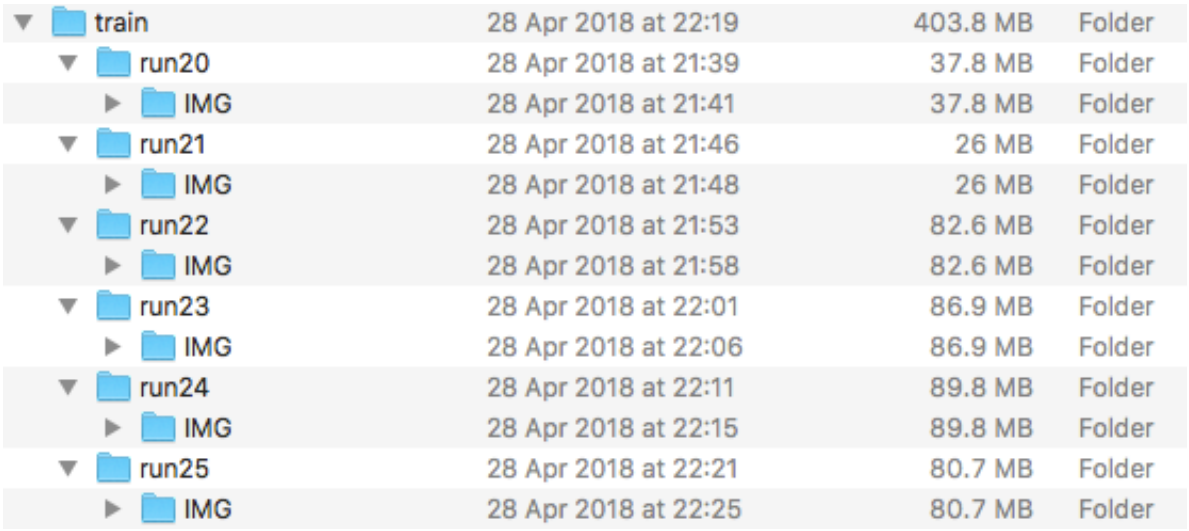
	Model 1	Model 2	Model 3
Training time (4000 img)	27[min]	44[min]	33 [min]
Loss	0.0338 (0.0338)	0.0292 (0.0290)	0.0205 (0.0196)
Val_loss (min)	0.0453 (0.0396)	0.0301 (0.0293)	0.0247 (0.0238)
Evaluation			
Weight	0.6953	0.6959	0.6994
IoU	0.3759	0.4824	0.4956
Final score	0.2614	0.3357	0.3466

Given the details above (and the training graph) model 3 seems to slightly overfit the data – the training loss is much better than the other models but the improvement in the validation loss is more modest.

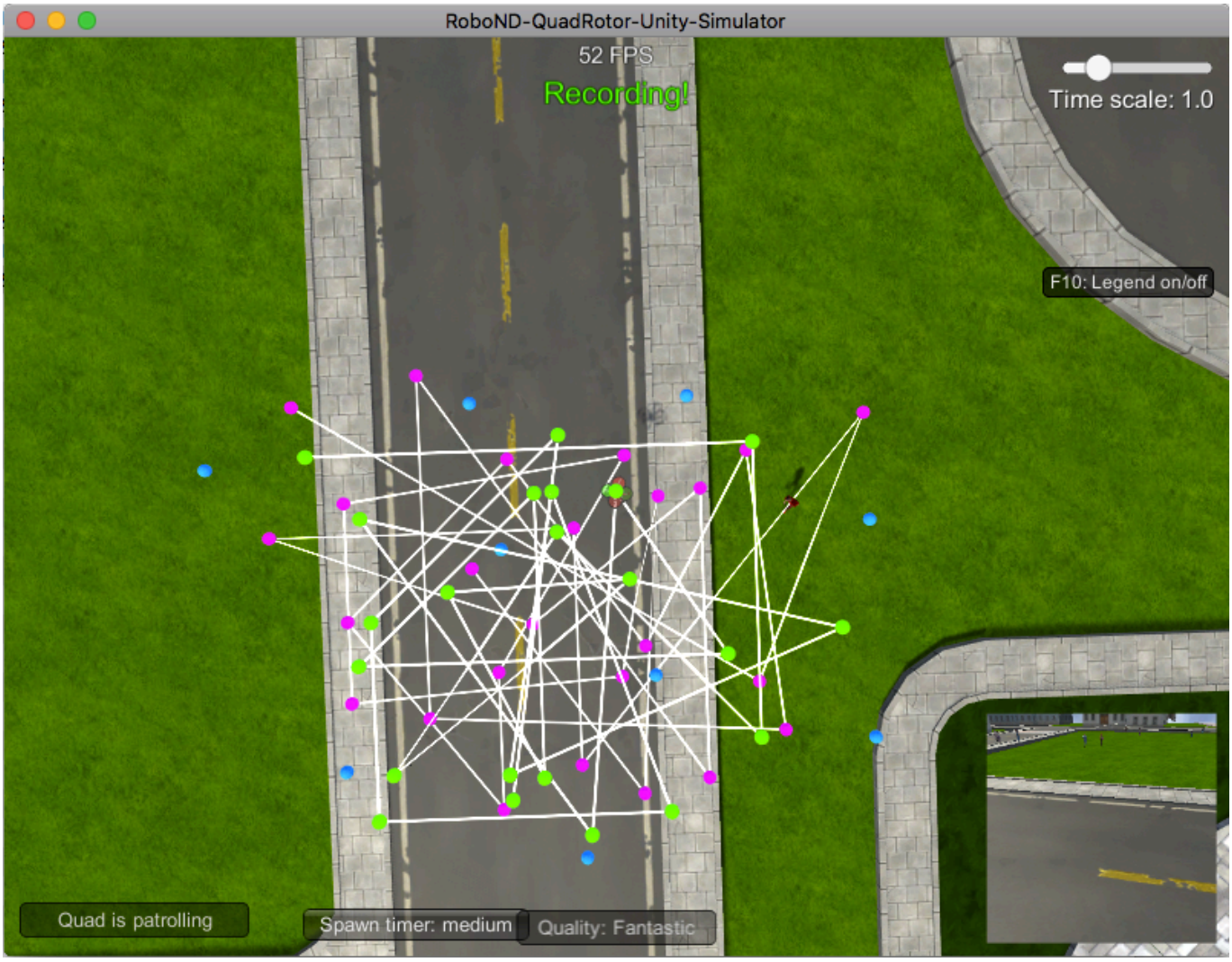
One way to check this model's potential is to throw more data at it.

Adding Data (Model 3 – Run 2)

I have created some more test data by producing 6 runs in the simulator with very random paths for hero and drone:



The type of paths I produced are like the following:



There are some important lessons learned from the generation of the data:

1. The simulator should be in Quality: Fantastic; this ensures that the train data images are as good as possible before they are down sampled to 160x160 and converted to jpeg. A lower quality would produce artefacts that are very problematic in our case as we perform pixel processing with the NN.
2. I called the folders in the `raw_sim_data` `run20`, `run21`, etc. so that the converted images have completely different names from the original ones (the run number is included in the file name). This way we can mix the two sets without risk of overwriting files.
3. The paths for the hero and the quad should be as random as possible and there should be as many points as we have time for. In some initial data sets I only created very simple paths (a rectangle for example) and the problem with this approach is that too much of the data is relatively repetitive. The NN will have no problem to learn with this data – as a matter of fact it will get very good training scores, but will do miserable when it comes to validation.
4. The number of images generated for a run should not be too big – in my case I stopped after the simulator places max 3,000 images in the IMG folder; since there are 5 images saved for each real frame (the camera image and 4 masks) that means the actual number of images produced for training is the number of images in IMG folder divided by 5; in my case aprox 600.

5. Once images are processed with the python script I have zipped them and copied to the aws server. There I created a separated folder `train-ext` where I copied the original training data (4100 images) plus my new data (3600 images) therefore having a training set of 7700 images.

I have used the same Model 3 as before without any changes and we have trained it on the new extended 7700 images set. Because we have more images we also can play with the hyper parameters a little. Here are the changes I have made for this training session:

- I've updated the way the steps per epoch are calculated for validation set so that it is derived from the number of validation images (1100) and the predetermined batch size:

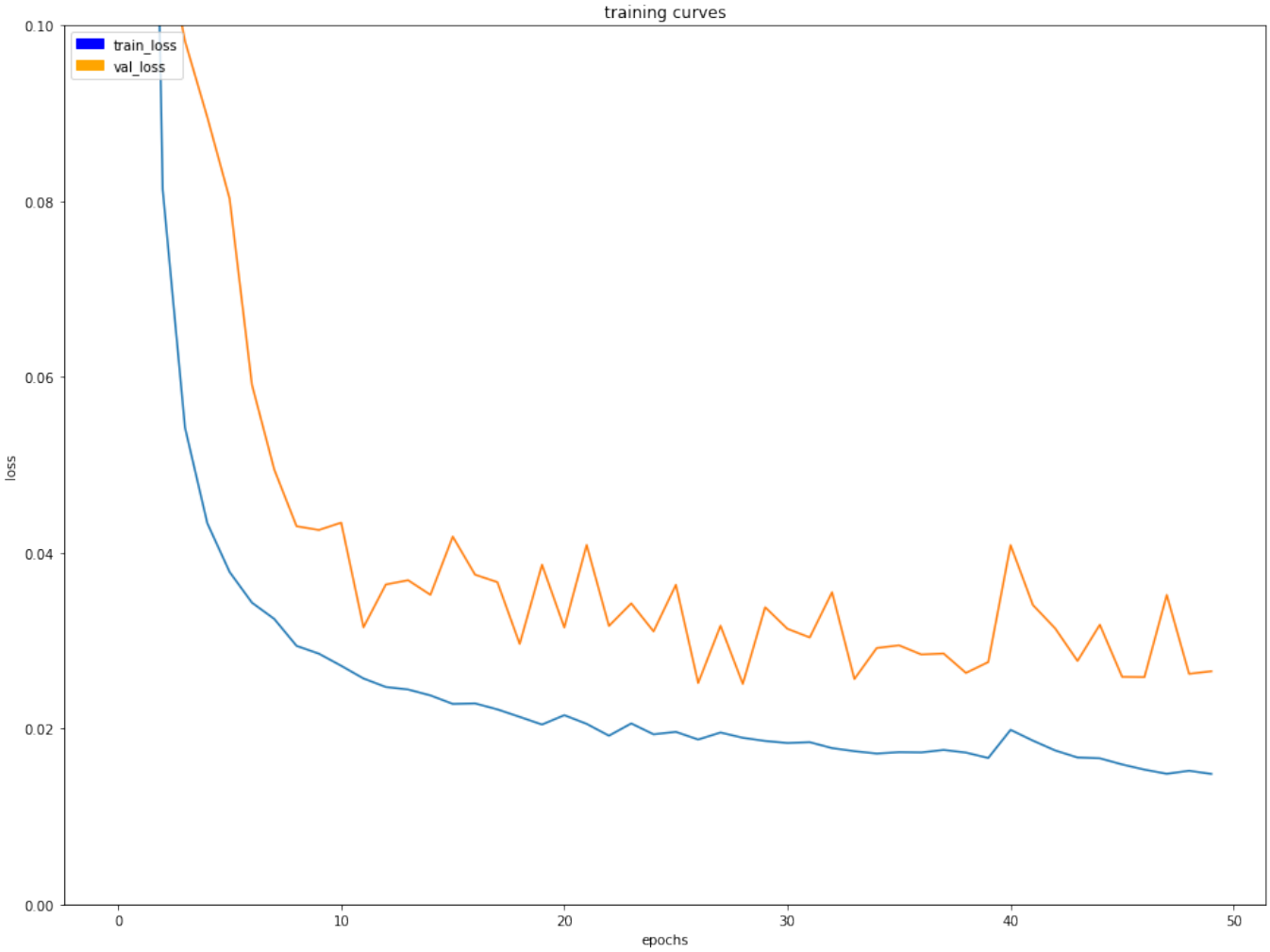
```
training_set_size = 7779
validation_set_size = 1184
batch_size = 100
num_epochs = 50
steps_per_epoch = training_set_size / batch_size + 1
validation_steps = validation_set_size / batch_size + 1
```

- The previous choice to use the val_loss metric as a guide for the decreasing learning rate in the callback seems now not very good, looking at the graphs. The val_loss is pretty volatile, and we have experienced the decrease of the learning rate too early in the previous trainings. Therefore, this time we will switch the metric to `loss` (the training loss itself). But since this metric is more stable we have also reduced the patience parameter from 5 to 4):

```
reduceLR = keras.callbacks.ReduceLROnPlateau(monitor='loss',
                                             factor=0.5,
                                             patience=4,
                                             verbose=1,
                                             mode='auto')
```

- The initial learning rate I increased to 0.002 – providing a slightly more aggressive training than before.

Here is the training graph (we continued to train over 50 epochs):



You can see that the training loss decreases nicely while the validation loss follows really close, although with a slightly more fluctuations than in the previous training of this Model. Nevertheless, the results are much better, both in the overall training loss as well as in validation. The training might have benefited from more epochs – we might consider that for the next execution.

Evaluation

The evaluation results are presented below:

In [17]:
```
# Scores for while the quad is following behind the target.
true_pos1, false_pos1, false_neg1, iou1 = scoring_utils.score_run_iou(val_following, pred_following)
```

```
number of validation samples intersection over the union evaulated on 542
average intersection over union for background is 0.9953554824876033
average intersection over union for other people is 0.34346724747637797
average intersection over union for the hero is 0.8837605377120039
number true positives: 539, number false positives: 0, number false negatives: 0
```

In [18]:
```
# 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_targ, pred_no_targ)
```

```
number of validation samples intersection over the union evaulated on 270
average intersection over union for background is 0.9870007339429052
average intersection over union for other people is 0.7323902720927059
average intersection over union for the hero is 0.0
number true positives: 0, number false positives: 57, number false negatives: 0
```

In [19]:
```
# This score measures how well the neural network can detect the target from far 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 0.9963658295779271
average intersection over union for other people is 0.43398700300178494
average intersection over union for the hero is 0.21045816476051585
number true positives: 128, number false positives: 2, number false negatives: 173
```

In [20]:
```
# 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)
```

```
0.7419354838709677
```

In [21]:
```
# The IoU for the dataset that never includes the hero is excluded from grading
final_IoU = (iou1 + iou3)/2
print(final_IoU)
```

```
0.547109351236
```

In [22]:
```
# And the final grade score is
final_score = final_IoU * weight
print(final_score)
```

```
0.40591984124
```

You can see we have passes the 40% threshold for the over final score! Both the weights and the IoU scores have improved. We seem to be on the right path.

Summary

At this moment it should be good to have an overview of the 3 models:

	Model 1	Model 2	Model 3	Model 3 (run 2)
Encoders	2	2	3	3
No of channels	12, 48	24, 96	12, 48, 192	12, 48, 192
Mid layer channels	48	96	96	96
Decoders	2	2	3	3
No of channels	48, 12	96, 24	72, 48, 12	72, 48, 12
Parameters	8,085	28,785	47, 829	47, 829
Training time	27[min]	44[min]	33 [min]	60 [min]
Data set [imgs]	4100	4100	4100	7700

Loss	0.0338 (0.0338)	0.0292 (0.0290)	0.0205 (0.0196)	0.0148 (0.0148)
Val_loss (min)	0.0453 (0.0396)	0.0301 (0.0293)	0.0247 (0.0238)	0.0265 (0.0251)
Evaluation				
Weight	0.6953	0.6959	0.6994	0.7420
IoU	0.3759	0.4824	0.4956	0.5471
Final score	0.2614	0.3357	0.3466	0.4059

We got robust improvements in the weight and IoU against a doubling of the training time due to extra training data.

The train graph suggests we might get some extra value from training the model a little longer – but this is unlikely to bring the overall score over 0.5. We will need to work on the model and increase the complexity. The current Model 3 doesn't seem to be powerful enough for the task at hand and the relatively low number of parameters are insufficient for a good score. We will therefore increase the complexity of the model.

The details for this model training are in the file `aws_model_training_model3_run2.html` in the appendix folder.

Model 4 (increase channels)

In the previous testing Model 2 showed a very interesting improvement of results by increasing the number of channels in the encoder and decoder layers. That seems reasonable as this means more parameters and also more information that is preserved and passed through the layers.

We increased the number of channels for the 3 encoder and decoders as follows:

```
def fcn_model(inputs, num_classes):

    # TODO Add Encoder Blocks.
    # Remember that with each encoder layer, the depth of your model
    # (the number of filters) increases.
    #                                                  input: 160x160x3
    en1 = encoder_block(inputs, filters=48, strides=2)     # en1: 80x80x48
    en2 = encoder_block(en1, filters=96, strides=2)        # en2: 40x40x96
    en3 = encoder_block(en2, filters=192, strides=2)       # en3: 20x20x192

    # TODO Add 1x1 Convolution layer using conv2d_batchnorm().
    mid = conv2d_batchnorm(en3, filters=96, kernel_size=1, strides=1)
    # mid: 20x20x96

    # TODO: Add the same number of Decoder Blocks as the number of Encoder Blocks
    de3 = decoder_block(mid, en2, filters=192)             # de3: 40x40x192
    de2 = decoder_block(de3, en1, filters=96)              # de2: 80x80x96
    de1 = decoder_block(de2, inputs, filters=48)           # de1: 160x160x12

    # The function returns the output layer of your model.
    return layers.Conv2D(num_classes, 1, activation='softmax', padding='same')(de1)
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_1 (InputLayer)         (None, 160, 160, 3)       0
_________________________________________________________________
separable_conv2d_keras_1 (Se (None, 80, 80, 48)        219
_________________________________________________________________
batch_normalization_1 (Batch (None, 80, 80, 48)        192
_________________________________________________________________
separable_conv2d_keras_2 (Se (None, 40, 40, 96)        5136
_________________________________________________________________
batch_normalization_2 (Batch (None, 40, 40, 96)        384
_________________________________________________________________
separable_conv2d_keras_3 (Se (None, 20, 20, 192)       19488
_________________________________________________________________
batch_normalization_3 (Batch (None, 20, 20, 192)       768
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 20, 20, 96)        18528
_________________________________________________________________
batch_normalization_4 (Batch (None, 20, 20, 96)        384
_________________________________________________________________
bilinear_up_sampling2d_1 (Bi (None, 40, 40, 96)        0
_________________________________________________________________
concatenate_1 (Concatenate)  (None, 40, 40, 192)       0
_________________________________________________________________
separable_conv2d_keras_4 (Se (None, 40, 40, 192)       38784
_________________________________________________________________
batch_normalization_5 (Batch (None, 40, 40, 192)       768
_________________________________________________________________
bilinear_up_sampling2d_2 (Bi (None, 80, 80, 192)       0
_________________________________________________________________
concatenate_2 (Concatenate)  (None, 80, 80, 240)       0
_________________________________________________________________
separable_conv2d_keras_5 (Se (None, 80, 80, 96)        25296
_________________________________________________________________
batch_normalization_6 (Batch (None, 80, 80, 96)        384
_________________________________________________________________
bilinear_up_sampling2d_3 (Bi (None, 160, 160, 96)      0
_________________________________________________________________
concatenate_3 (Concatenate)  (None, 160, 160, 99)      0
_________________________________________________________________
separable_conv2d_keras_6 (Se (None, 160, 160, 48)      5691
_________________________________________________________________
batch_normalization_7 (Batch (None, 160, 160, 48)      192
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 160, 160, 3)       147
=================================================================
Total params: 116,361
Trainable params: 114,825
Non-trainable params: 1,536
```

The structure of the model is the same, what has changed is the depth of the layers (third dimension). As a result we also have a significantly larger amount of parameters in the network – therefore in theory being able to learn more stuff.

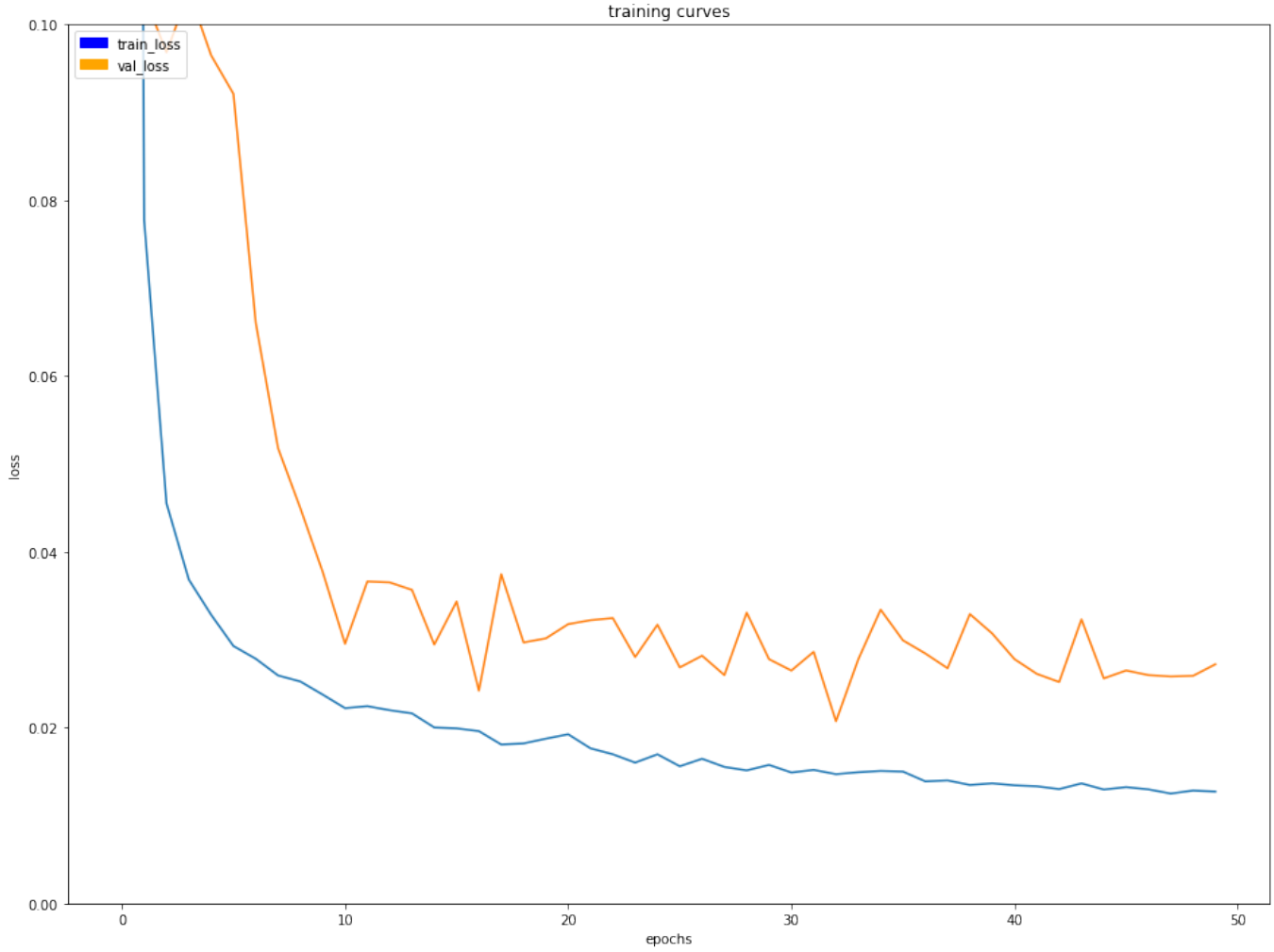
We have used the same settings as we did for the run 2 training of Model 3:

```
learning_rate = 0.002
training_set_size = 7779
validation_set_size = 1184
batch_size = 100
num_epochs = 50
steps_per_epoch = training_set_size / batch_size + 1
validation_steps = validation_set_size / batch_size + 1
workers = 2
```

In terms of learning rate callback we have reduced even more the patience parameter, as we have noticed that the fluctuations for the training loss are very small and we would like to transition to a lower learning rate a little sooner when no progress is made:

```
reduceLR = keras.callbacks.ReduceLROnPlateau(monitor='loss',
                                             factor=0.5,
                                             patience=2,
                                             verbose=1,
                                             mode='auto')
```

With these settings the learning graph over 50 epochs looks like this:



Again, we see that we could have probably trained the model for a little longer as the training loss continues to decrease. The learning rate has been decreased 3 times: at epoch 21 to 0.001, at epoch 36 to 0.0005 and at epoch 46 to 0.00025. The fact that this had a positive impact on training (in the sense

that it allowed for the training to improve and for the loss to decrease) is a good signal that we are doing the right thing with the LR reduction. The fact that the validation loss seems to stay a little too high is a little troubling, although it might be possible that a longer training might decrease that loss too.

Evaluation

The results of the evaluation for this model are:

In [18]:
```
# Scores for while the quad is following behind the target.
true_pos1, false_pos1, false_neg1, iou1 = scoring_utils.score_run_iou(val_following, pred_following)
```

```
number of validation samples intersection over the union evaulated on 542
average intersection over union for background is 0.9958133516149913
average intersection over union for other people is 0.36281511459904714
average intersection over union for the hero is 0.9057981172304144
number true positives: 539, number false positives: 0, number false negatives: 0
```

In [19]:
```
# 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_targ, pred_no_targ)
```

```
number of validation samples intersection over the union evaulated on 270
average intersection over union for background is 0.9881148929283664
average intersection over union for other people is 0.7562404214339161
average intersection over union for the hero is 0.0
number true positives: 0, number false positives: 58, number false negatives: 0
```

In [20]:
```
# This score measures how well the neural network can detect the target from far 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 0.9966054765799973
average intersection over union for other people is 0.4479522915694425
average intersection over union for the hero is 0.2524396957063291
number true positives: 136, number false positives: 5, number false negatives: 165
```

In [21]:
```
# 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)
```

```
0.7475083056478405
```

In [22]:
```
# The IoU for the dataset that never includes the hero is excluded from grading
final_IoU = (iou1 + iou3)/2
print(final_IoU)
```

```
0.579118906468
```

In [23]:
```
# And the final grade score is
final_score = final_IoU * weight
print(final_score)
```

```
0.432896192543
```

We see again very good improvements of the IoU factor but only a marginal improvement of the weight factor, and the final score is still short of the 0.50 target that we have.

Summary

In the summary I have removed the Model 1 that is no longer of interest:

	Model 2	Model 3	Model 3 (run 2)	Model 4
Encoders	2	3	3	3

	Model 2	Model 3	Model 3 (run 2)	Model 4
No of channels	24, 96	12, 48, 192	12, 48, 192	48, 96, 192
Mid layer channels	96	96	96	96
Decoders	2	3	3	3
No of channels	96, 24	72, 48, 12	72, 48, 12	192, 96, 48
Parameters	28,785	47, 829	47, 829	114,825
Training time	44[min]	33 [min]	60 [min]	113 [min]
Data set [imgs]	4100	4100	7700	7700
Epochs	50	50	50	50
Loss	0.0292 (0.0290)	0.0205 (0.0196)	0.0148 (0.0148)	0.0127 (0.0125)
Val_loss (min)	0.0301 (0.0293)	0.0247 (0.0238)	0.0265 (0.0251)	0.0272 (0.0207)
Evaluation				
Weight	0.6959	0.6994	0.7420	0.7476
IoU	0.4824	0.4956	0.5471	0.5791
Final score	0.3357	0.3466	0.4059	0.4329

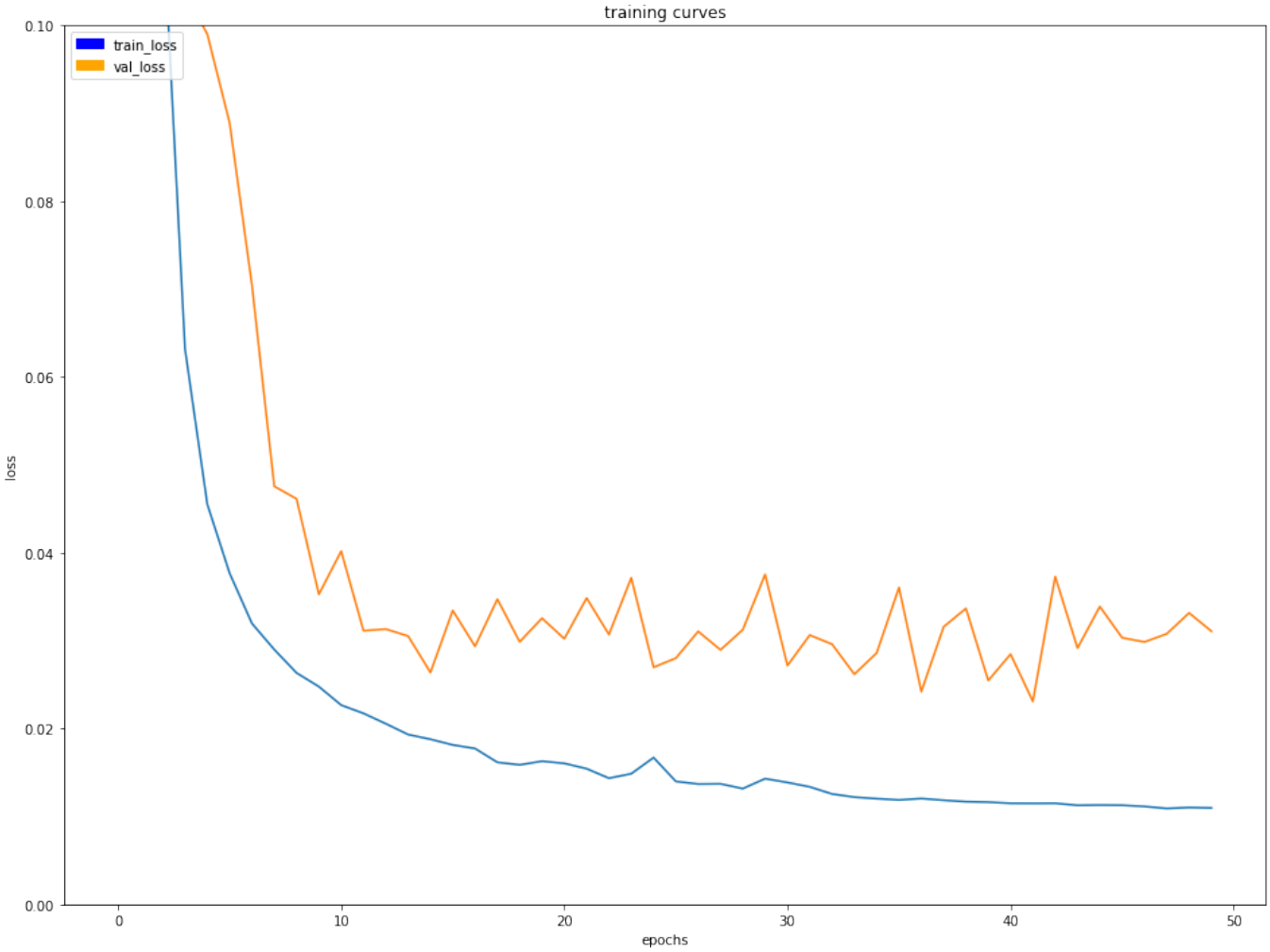
As indicated earlier this new model improved the IoU rating but had very low impact on the Weight factor.

It seems that the depth of the layers (the third dimension) has an impact on the IoU score, which seems reasonable because by having more channels we preserve more information at the pixel level and we have a better result in the individual pixel classifications. On the other hand the number of layers seem to impact the weight factor, again a sensible assumption due to the fact that more layers means more abstract information can be encoded and therefore the classification of the appearance of the 3 categories is better managed.

Now, we already have two models (Model 3 run 2 and Model 4) that beat the target of 0.4 for the final score. Can we do better than that?

The observations above lead to an obvious change for the next model: we would add one more layer in the encoder and the decoder and we keep on increasing the number of channels in this layer.

We've done this (adding a layer 4 for encoder and decoder) and the results were a decrease in performance. The number of parameters for the model went to approx. 250,000 and we suspect that the number of training samples is insufficient for the size of the model which over-fits the training data. We obtained a training loss of 0.0110 but a very bad validation loss with a training curve like this:



Obviously, this model badly over-fits the data and the results show this with a lower scoring on all fronts:

In [21]:
```
# 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)
```

```
0.7109634551495017
```

In [22]:
```
# The IoU for the dataset that never includes the hero is excluded from grading
final_IoU = (iou1 + iou3)/2
print(final_IoU)
```

```
0.54212992971
```

In [23]:
```
# And the final grade score is
final_score = final_IoU * weight
print(final_score)
```

```
0.385434567967
```

So, this path does not seem to be a productive one. Can we do something else? In fact, yes: in the definition of the decoder blocks we can chose to use two separable convolutional layers. This adds a relatively low number of parameters (thus having a lower change of over-fitting) but improves the ability to process the information spatially. Let's try this.

Model 5 (improving decoder)

We have updated the decoder block code in two ways:

- We added one more separable convolutional layers at the end
- We changed the code for the skip connection (the concatenation) so that if the "large" layer is `None` then we do not do any concatenation. This is to allow the construction of decoder blocks that do not use skip connection. We will only use skip connection for the first two decoders (from the "centre" of the network) and we will not use skip connection for the last decoder. This is also the approach taken in other architectures and is motioned very briefly in the videos for the session.

The new code for the decoder looks thus:

```
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
    if large_ip_layer != None:
        merge_layer = layers.concatenate([upsample_layer, large_ip_layer])
    else:
        merge_layer = upsample_layer

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

    return output_layer
```

Now in the definition of the model, the only think that we changed is that the last decoder block passes a None to the constructor, thus no longer using a skip connection there:

```
def fcn_model(inputs, num_classes):

    # TODO Add Encoder Blocks.
    # Remember that with each encoder layer, the depth of your model
    # (the number of filters) increases.
    #                                                  input: 160x160x3
    en1 = encoder_block(inputs, filters=48, strides=2)     # en1: 80x80x48
    en2 = encoder_block(en1, filters=96, strides=2)        # en2: 40x40x96
    en3 = encoder_block(en2, filters=192, strides=2)       # en3: 20x20x192

    # TODO Add 1x1 Convolution layer using conv2d_batchnorm().
    mid = conv2d_batchnorm(en3, filters=96, kernel_size=1, strides=1)
    # mid: 20x20x96

    # TODO: Add the same number of Decoder Blocks as the number of Encoder Blocks
    de3 = decoder_block(mid, en2, filters=192)              # de3: 40x40x192
    de2 = decoder_block(de3, en1, filters=96)              # de2: 80x80x96
    de1 = decoder_block(de2, None, filters=48)           # de1: 160x160x48
```

```
    # 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')(de1)
```

You can see the added layers and the lack of a concatenation layer in the last decoder in the model summary:

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_1 (InputLayer)         (None, 160, 160, 3)       0
_________________________________________________________________
separable_conv2d_keras_1 (Se (None, 80, 80, 48)        219
_________________________________________________________________
batch_normalization_1 (Batch (None, 80, 80, 48)        192
_________________________________________________________________
separable_conv2d_keras_2 (Se (None, 40, 40, 96)        5136
_________________________________________________________________
batch_normalization_2 (Batch (None, 40, 40, 96)        384
_________________________________________________________________
separable_conv2d_keras_3 (Se (None, 20, 20, 192)       19488
_________________________________________________________________
batch_normalization_3 (Batch (None, 20, 20, 192)       768
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 20, 20, 96)        18528
_________________________________________________________________
batch_normalization_4 (Batch (None, 20, 20, 96)        384
_________________________________________________________________
bilinear_up_sampling2d_1 (Bi (None, 40, 40, 96)        0
_________________________________________________________________
concatenate_1 (Concatenate)  (None, 40, 40, 192)       0
_________________________________________________________________
separable_conv2d_keras_4 (Se (None, 40, 40, 192)       38784
_________________________________________________________________
batch_normalization_5 (Batch (None, 40, 40, 192)       768
_________________________________________________________________
separable_conv2d_keras_5 (Se (None, 40, 40, 192)       38784
_________________________________________________________________
batch_normalization_6 (Batch (None, 40, 40, 192)       768
_________________________________________________________________
bilinear_up_sampling2d_2 (Bi (None, 80, 80, 192)       0
_________________________________________________________________
concatenate_2 (Concatenate)  (None, 80, 80, 240)       0
_________________________________________________________________
separable_conv2d_keras_6 (Se (None, 80, 80, 96)        25296
_________________________________________________________________
batch_normalization_7 (Batch (None, 80, 80, 96)        384
_________________________________________________________________
separable_conv2d_keras_7 (Se (None, 80, 80, 96)        10176
_________________________________________________________________
batch_normalization_8 (Batch (None, 80, 80, 96)        384
_________________________________________________________________
bilinear_up_sampling2d_3 (Bi (None, 160, 160, 96)      0
_________________________________________________________________
separable_conv2d_keras_8 (Se (None, 160, 160, 48)      5520
_________________________________________________________________
batch_normalization_9 (Batch (None, 160, 160, 48)      192
_________________________________________________________________
separable_conv2d_keras_9 (Se (None, 160, 160, 48)      2784
_________________________________________________________________
batch_normalization_10 (Batc (None, 160, 160, 48)      192
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 160, 160, 3)       147
=================================================================
Total params: 169,278
Trainable params: 167,070
Non-trainable params: 2,208
```

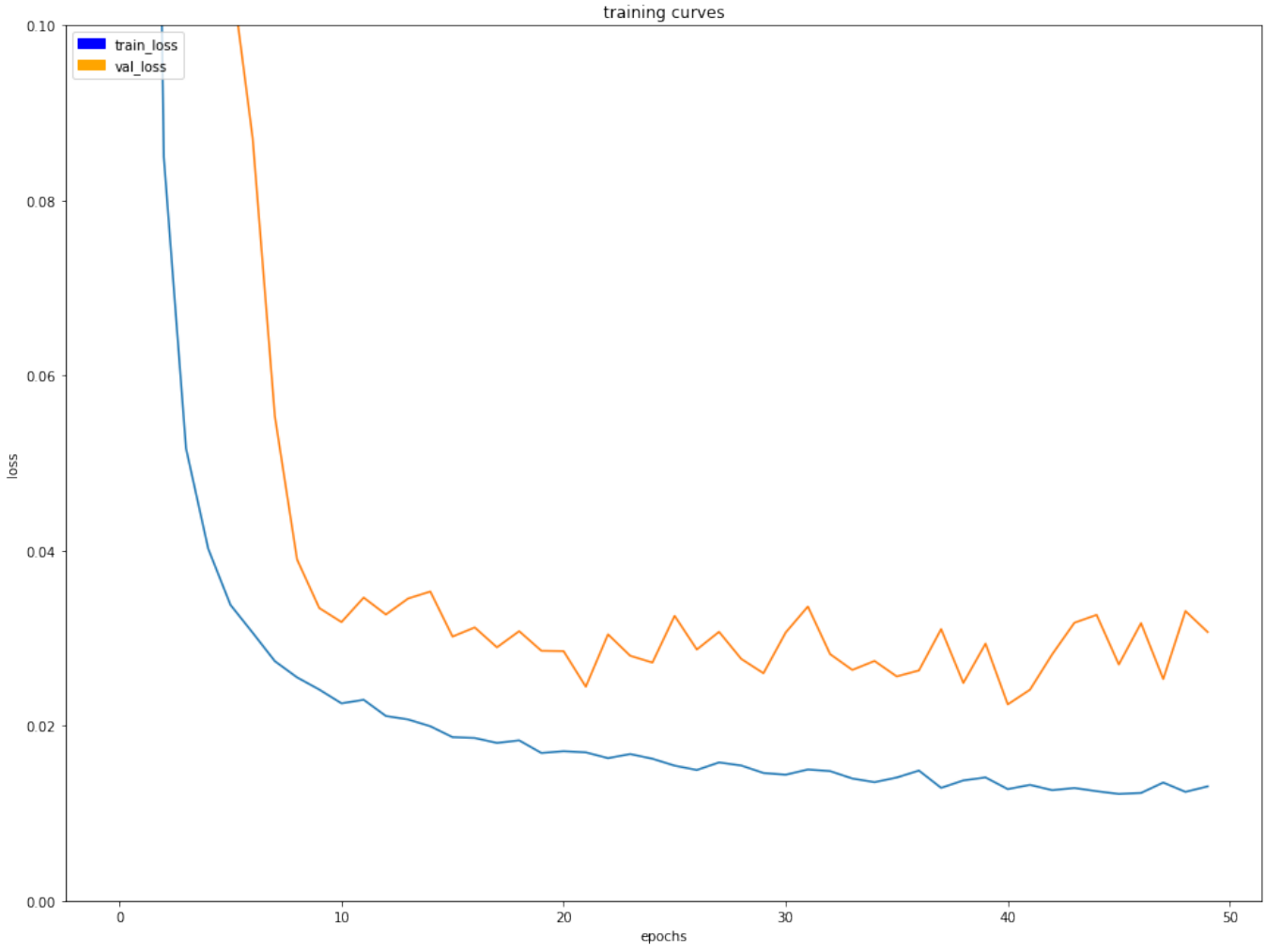
We do now the classic 50 epochs training using the full set of 7700 images. The settings for the paramters are very similar to the ones before, except that I decided to start with a lower learning rate (0.001):

```
learning_rate = 0.001
training_set_size = 7779
validation_set_size = 1184
batch_size = 100
num_epochs = 50
steps_per_epoch = training_set_size / batch_size + 1
validation_steps = validation_set_size / batch_size + 1
workers = 2
```

And the control of change of LR was refined to patience = 3 and I included a minimum LR of 0.0001 in order to avoid getting in situations where the model gets stuck because of lack of progress:

```
reduceLR = keras.callbacks.ReduceLROnPlateau(monitor='loss',
                                             factor=0.5,
                                             patience=3,
                                             verbose=1,
                                             mode='auto',
                                             min_lr = 0.0001)
```

The learning graph is the following:



With a final loss of 0.0131 (best 0.0122 in epoch 46) and a validation loss at end of 0.0307 (best 0.0224 at epoch 41). The interesting thing about this model is that it does not necessarily feels better than the Model 4 we had before, but the validation is more consistent and on average has better values than the ones in the previous model 4. This is also visible in the evaluation:

Evaluation

In [17]:
```
# Scores for while the quad is following behind the target.
true_pos1, false_pos1, false_neg1, iou1 = scoring_utils.score_run_iou(val_following, pred_following)
```

```
number of validation samples intersection over the union evaulated on 542
average intersection over union for background is 0.9946354240679545
average intersection over union for other people is 0.3650031662708496
average intersection over union for the hero is 0.9025074195936135
number true positives: 539, number false positives: 0, number false negatives: 0
```

In [18]:
```
# 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_targ, pred_no_targ)
```

```
number of validation samples intersection over the union evaulated on 270
average intersection over union for background is 0.9871873513143762
average intersection over union for other people is 0.7513464098885752
average intersection over union for the hero is 0.0
number true positives: 0, number false positives: 76, number false negatives: 0
```

In [19]:
```
# This score measures how well the neural network can detect the target from far 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 0.9961376382469861
average intersection over union for other people is 0.4631128945230213
average intersection over union for the hero is 0.28817325231933233
number true positives: 156, number false positives: 2, number false negatives: 145
```

In [20]:
```
# 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)
```

```
0.7570806100217865
```

In [21]:
```
# The IoU for the dataset that never includes the hero is excluded from grading
final_IoU = (iou1 + iou3)/2
print(final_IoU)
```

```
0.595340335956
```

In [22]:
```
# And the final grade score is
final_score = final_IoU * weight
print(final_score)
```

```
0.450720624717
```

The overall score improved from 0.4329 to 0.4507 a quite healthy increase.

As a matter of fact, if we pay close attention to the training graph you can see that after epoch 40 the model seems to be doing worse as visible in the training loss but especially in the validation loss.

The detail output of this model is included in the file aws_model_training_model5_run1.html in the appendix.

We have provided the weights and the definition of the model for this last model in the files:
`model_weights_model5_run1_e50`
`config_model_weights_model5_run1_e50`
in the weights directory.

Summary

Let's see now for the last time the summary of the models (I have kept only the latest 3 ones):

	Model 3	Model 3 (run 2)	Model 4	Model 5
Encoders	3	3	3	3
No of channels	12, 48, 192	12, 48, 192	48, 96, 192	48, 96, 192
Mid layer channels	96	96	96	96
Decoders	3	3	3	3
No of channels	72, 48, 12	72, 48, 12	192, 96, 48	192, 96, 48
Parameters	47, 829	47, 829	114,825	167,070
Training time	33 [min]	60 [min]	113 [min]	157 [min]
Data set [imgs]	4100	7700	7700	7700
Epochs	50	50	50	50
Loss	0.0205 (0.0196)	0.0148 (0.0148)	0.0127 (0.0125)	0.0131 (0.0122)
Val_loss (min)	0.0247 (0.0238)	0.0265 (0.0251)	0.0272 (0.0207)	0.0307 (0.0221)
Evaluation				
Weight	0.6994	0.7420	0.7476	0.7571
IoU	0.4956	0.5471	0.5791	0.5953
Final score	0.3466	0.4059	0.4329	0.4507