Follow-Me Project

The project is a (deep learning) semantic segmentation task within the scene understanding field of computer vision.

The purpose is to be able to successfully track a target in simulation after having trained a model to recognize the target at pixel level in images where other people and a diversified background is also present.

To accomplish the task of classifying each and every pixel in images, a Fully Convolutional Network (FCN) is used. A CNN cannot do the job as it does not preserve the spatial information throughout the network.

Data Collection

The data used for training and validation of the FCN model was a combination of data provided by Udacity and data collected by me using a simulated quadcopter in QuadSim_Win software (provided by Udacity).

Data collection was an iterative process where I tried to collect more images depending on the score I wanted to improve (mainly patrolling with/without target).

The original training dataset comprised of 4,131 images and the validation dataset of 1,184 images. The total number of images in the last run were 5,780 for training and 2,481 for validation.

The strategy used to collect images was to 1) collect images of the target from afar, 2) surrounded by crowd, 3) in different locations, 4) near obstacles/building/corners, 5) from different quadcopter heights.

The model

The structure of the FCN comprises an encoder and a decoder portion.

Encoder Block

The aim of the encoder portion is to extract features from images (similarly to a CNN) AND preserve the location information for each pixel is in the original image. To do this we substitute the fully connected layer of a CNN (which requires a flattening of its input to a 2-d tensor) with a 1x1 convolutional layer using *conv2d_batchnorm* function provided in the notebook.

In order to improve runtime performance, two techniques were used in the encoder space:

1- **Separable Convolutions**. Separable convolutions require less parameters than a convolutional layer and because of this are also less prone to overfitting. To add separable convolutions to the encoder, I have used the *separable_conv2d_batchnorm* provided in the notebook.
2- **Batch Normalization**, where we normalize input to each layer. The improvement is given by a faster convergence to a local minimum and the use of higher learning rates (which in turn tend to converge faster to a local minimum). This is done in the notebook by calling *tensorflow.contrib.keras.python.keras.layers.BatchNormalization()* on a given output of previous layer.

The *separable_conv2d_batchnorm* function provided in the notebook performs both separable convolution and batch normalization and is called by the *encoder_block* function that will be used to define a layer in the model:

```
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
```

Decoder Block

The purpose of the decoder is to upscale the output of the encoder to the same size of the original image and to make prediction for each pixel.

One technique to do this is to use a transpose convolutional layer. Another technique, the one used in this project, is the "**bilinear upsampling**". To perform bilinear upsampling, I used the the *bilinear_upsample* function provided in the notebook.

During the decode phase we want to recall finer spatial details from earlier layers in the network. To do this, we link one output of a decoder layer with one of an encoder layer. This technique is called "skip connection" and can be achieved by summation of element-wise of two layers or, as done in this project, by **concatenating** the outputs of the two layers (which allows more flexibility in terms of depth of the output layers). This was done by using the *layers.concatenate* in *tensorflow.contrib.keras.python.keras.*

The decoder block used consisted of an upsampled layer, a concatenation and 2 separable convolutions.

```
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

    separable_layer = separable_conv2d_batchnorm(concatenated_layer,
                                                 filters,
                                                 strides=1)#rl 2/2


    output_layer = separable_conv2d_batchnorm(separable_layer,
                                                 filters,
                                                 strides=1)
    return output_layer
```

The FCN model used here consists of an encoder with 3 layers, a 1x1 conv layer and a decoder of 3 layers.

```
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.
    encoder_1_layer = encoder_block(inputs, 32, 2)
    encoder_2_layer = encoder_block(encoder_1_layer, 64, 2)
    encoder_3_layer = encoder_block(encoder_2_layer, 128, 2)

    #encoder_4_layer = encoder_block(encoder_3_layer, 256, 2)

    # TODO Add 1x1 Convolution layer using conv2d_batchnorm().
    conv_1_1_layer = conv2d_batchnorm(encoder_3_layer, 256, 1, 1)

    # TODO: Add the same number of Decoder Blocks as the number of Encoder Blocks
    decoder_1_layer = decoder_block(conv_1_1_layer, encoder_2_layer, 128)
    decoder_2_layer = decoder_block(decoder_1_layer, encoder_1_layer, 64)
    decoder_3_layer = decoder_block(decoder_2_layer, inputs, 32)

    #decoder_4_layer = decoder_block(decoder_3_layer, inputs, 32)

    # The function returns the output layer of your model. "x" is the final layer obtained from the last decoder_block()
    output_layer = layers.Conv2D(num_classes, 1, activation='softmax', padding='same')(decoder_3_layer)

    return output_layer
```

Experiment

I have run about 20 experiments before I could reach the score above 40%. 5 of these experiments consisted in adding new images to training and/or validation. After the first few experiments, I could see that the model converged to local optimum quite fast, around 20 epochs with a learning rate of 0.001 and 15 epochs with a learning rate 0.01. To allow for an extensive search of local optimum, I set the number of epochs to 50 and introduced to keras.callbacks to save the best model and perform an early stopping should the validation loss stop improving after a patience of 5 epochs. However I did a big mistake in the sense that I forgot to add the step that loads the best model found during training before running the prediction step, thus not being able to link changes to the model/data and hyperparameters with the final score. Once I noticed this big error, I realized that the best model found during training was not the one giving me a good final score, thus I thought that

the model was overfitting. Something I tried was to apply a dropout layer in the *separable_conv2d_batchnorm* function and played with several dropout probabilities (0.1,0.2,up to 0.5). The results were not that promising. I then removed the dropout layer altogether and simplified the network: rather than 4 encoders/4 decoders layers I used 3 encoders/decoders. That seemed to improve the final score on unseen data. Something else I reverted back to the original status was the generator of validation data (at a certain point I set the flag shift_aug=True). Also, at some point I set the optimizer to Adamax rather than Adam but then reverted it back to Adam. I have experimented with a couple of learning rates: 0.001 and 0.01, which impacted the time it took to converge. I set the batch size to 20 and 32 images and calculated calculated the steps per epoch (or validation steps) as #images in training (#images in validation) divided by batch size.
Further improvements: Few experiments ended in tensorflow/keras errors and thus I could not complete them this time.

In one experiment I wanted to use *keras.callbacks.ReduceLROnPlateau* to start with a relatively high learning rate (to converge quickly) and then reduce it by a certain factor when the validation loss stops improving at that level of learning rate (to keep the search going).

Another experiment I could not complete because of errors aimed at assigning the 3 classes different weights of importance (use of keras *sample_weight*), with the target class being the most important one (and also the least present in the image).

Also, one could experiment with transfer learning in the encoder portion of the network.

Evaluation

The metric used to estimate the performance of the network on unseen data is mean IoU, which is the mean IoU across all classes. IoU is defined at class level as # pixels belonging to that class and being correctly classified as belonging to that class divided by #pixels belonging to the class or being classified as belonging to the class. The final score was ca 43%.