Follow Me Project

First follow readme of the Udacity Project to clone project depository.

```
$ git clone https://github.com/udacity/RoboND-DeepLearning-Project.git
```

There are two major pieces of work for the project.

- Train data by simulator
- Create a Fully Convolution Neural Network (FCN) for image segmentation

Train Data

By downloading QuadSim simulator from https://github.com/udacity/RoboND-DeepLearning-Project/releases/latest, we can start simulator application to train data by setting “spawn people” option and clicking “DL Training” as shown in Fig. 1.

Fig. 1 Using Simulator to start train data

Thereafter, in the simulator, we can hit “H” to set it to the local control mode and use arrow keys or AWSD to move drone around. “C” and Space are used to move down or up. We can create patrol route by using “P” to add patrol points (green ball), “O” to add hero walking points (pink ball) , “I” to add spawning people point (blue ball), and “M” to start spawn random people. Once you are satisfied with the three routes, you can click “H” to start patrolling mode. “R” is the switch to start and stop recording images from Quad-drone.

It’s good to save recording images under ~DeepLearning-Project/data/raw_sim_data/train/[run number], i.e. run7. After images are properly saved, you can run preprocess_ims.py to convert images from PNG to JPEG and generate masks under ~DeepLearning-Project/data/preprocess_ims/train directory.

A common problem in simulator images is missing target when patrol area is too large. To accommodate the problem, I tried couple ways as shown in Fig. 2 and Fig. 3. Fig. 2 shows a linear patrol line with zig-zag hero walking with dense traffic. Fig. 3 shows circle like area for area and hero walking with dense spawning people.

Fig. 2 & 3 Trainings on Simulator

Fully Convolutional Networks for Segmentation

Paper of Fully Convolutional Networks (FCN) for Semantic Segmentation by Long, Shelhamer, and Darrell, https://people.eecs.berkeley.edu/~jonlong/long_shelhamer_fcn.pdf is referred.

It took me more than five hours on my MacBook Pro to run the training notebook. I switched to AWS GPU instance to train data with Udacity Robot Lab AMI. The aws.sh included in the code directory installs TensorFlow and necessary libraries in an Anaconda environment.

Fig. 4 illustrates an FCN. The left encoder is a convolutional neuron network. Instead of connecting to a flat-fully-connected layer, it connects through an 1x1 convolutional network to a decoder. 1x1 convolutional network has only kernel and stride size one and stores information in deep layer of filters. It helps preserve spatial information from the image and is useful for semantic segmentation of images. I used 1024 filters for the 1x1 convolutional layer.

The decoder upsampling or de-convoluting to make it back to higher dimension. Bilinear upsampling which utilizes the weighted average of four nearest known pixels located diagonally to a given pixel to estimate a new pixel intensity is used for it.

In all these layers, batch normalization is used. During training, each layer's inputs are normalized to mean and variance of the values in the current mini-batch. I specified kernel regularizer with tf.contrib.layers.l2_regularizer(scale=0.1) for the convolution network layer to avoid over-fitting.

Skipping layer is also used to keep signals from front layers to later decoding layers to improve the resolution of results.

```
def fcn_model(inputs, num_classes):
    # Add Encoder Blocks.
    enc1 = encoder_block(inputs, filters=16, strides=2)
    enc2 = encoder_block(enc1, filters=32, strides=2)
    enc3 = encoder_block(enc2, filters=64, strides=2)
    enc4 = encoder_block(enc3, filters=128, strides=2)

    # Add 1x1 Convolution layer using conv2d_batchnorm().
    conv_1x1 = conv2d_batchnorm(enc4, 1024, kernel_size=1, strides=1)

    # Add the same number of Decoder Blocks as the number of Encoder Blocks
    dec1 = decoder_block(conv_1x1, enc3, 128)
    dec2 = decoder_block(dec1, enc2, 64)
    dec3 = decoder_block(dec2, enc1, 32)
    x = decoder_block(dec3, inputs, 16)
    # 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)
```

In the train, I added one more encoding layer and a corresponding decoding layer which helps final score.

Table 1 shows some tries and errors that I went through with parameters and results.
Here are the hyper parameters that I used after tuning.

`learning_rate = 0.002`
Learning rate is the delta increase in each step along gradient to reach minimum. Big rate might lead to big errors. Too small rate can take longer time and might lead to local minimum.

`batch_size = 128`
Batch size is the amount of data processing at one time. It's good to make as big as memory can sustain. Small batch size can lead to bigger errors. I ran out of memory with 192 and 256.

`num_epochs = 20`
Number of epochs specifies number of batches that entire dataset going propagated through the network.

`steps_per_epoch = 80`
Number of batches of training images go through the network in 1 epoch. 8165(training images)/128(batch size) = 63.8

`validation_steps = 20`
Number of batches of validation images go through the network in 1 epoch. 1187 (validation images) / 128 (batch size) = 9.27

`workers = 8`
Eight workers was spinning up in an AWS GPU instance in training.

Besides the provided training data, I included additional training data, train31.zip, train37.zip, and train39.zip in the data directory.

`The final score is 0.417073817766.`

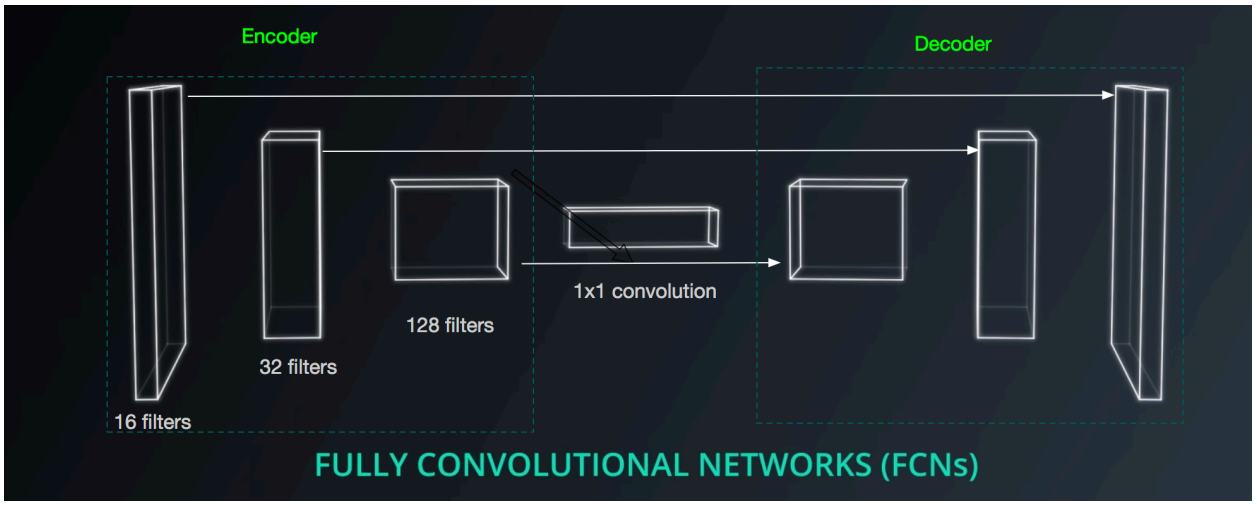
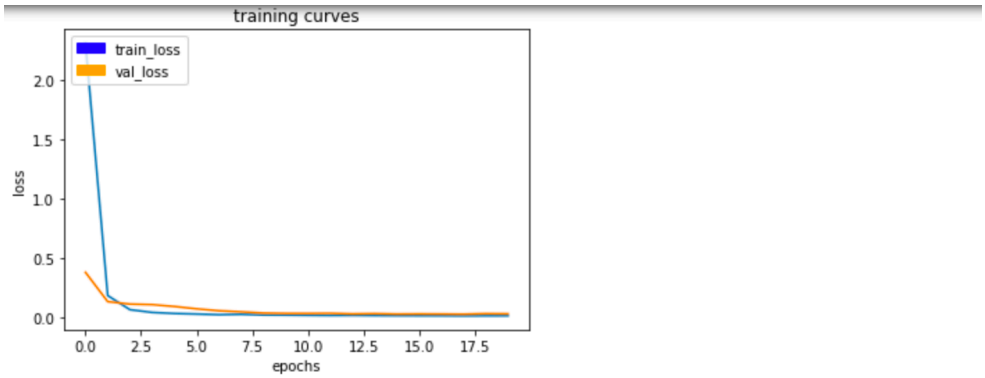


Fig. 4 Fully Convolutional Networks




```
80/80 [==============================] - 137s - loss: 0.0158 - val_loss: 0.0324
```

Fig. 5 Training Curve

Table 1 shows some tries and errors for hyperparameter tuning.

Table 1 Hyperparameters Tuning

	following behind the target	quad is on patrol and the target is not visable	detect the target from far away	final grade score
learning rate=0.001, batch=128, epochs=10, steps=200, validation=50	539/0/0	0/52/0	63/0/238	0.327769287903
kernel regularizer 0.05	538/0/1	0/26/0	38/1/263	0.307679649116
kernel regularizer 0.1	539/0/0	0/32/0	90/1/211	0.368747903307
kernel/bias regularizer 0.1	539/0/0	0/67/0	98/0/203	0.353568457077
add train31 (5071)	539/0/0	0/39/0	107/0/194	0.396784535836
add train32,33 (7048) batch = 64, epoch = 12	539/0/0	0/51/0	106/2/195	0.339338314819
add train37 and 38 (10883)	539/0/0	0/55/0	96/2/205	0.362178190961

With a trained model, I used it to test with simulator. Fig. 6 shows running Simulator with the trained model.

Fig. 6 Running Simulator with the trained model

Future Improvement

- To better identify far away target
- To improve situation when no target is not visible
- Catch more training data with high quality

This is a good project. It proves to me how important of high quality training data is and how important hyper-parameters are.