Follow Me Project

This project trains a deep neural network to identify and track a target in simulation, so-called "follow me" application. Fully Convolutional Neural Network (FCN) is used to segment images from drone to identify the target hero and everyone else. The same techniques could be extended to scenarios like advanced cruise control in autonomous vehicles or human-robot collaboration in industry.

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

```
$ 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 and run the included Jupyter Notebook with final score > 40%

Train Data

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

Fig. 1 Using Simulator to 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 drone down or up. We can create patrol route by using "P" to add patrol points (green ball) in the area, "O" to add hero walking points (pink ball) on the ground, "I" to add spawning people point (blue ball) on the ground, 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 or 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 python 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. It usually happens 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 path with dense traffic. Fig. 3 shows circle like area for patrol and hero walking with dense spawning people.

Fig. 2 & 3 Trainings on Simulator

Fully Convolutional Networks for Segmentation

Fully Convolutional Networks (FCN) for Semantic Segmentation is a better approach than traditional box approach. Paper by Long, Shelhamer, and Darrell, https://people.eecs.berkeley.edu/~jonlong/long_shelhamer_fcn.pdf is a common reference in FCN implementation.

Fig. 4 illustrates a typical FCN with an encoder, an 1x1 convolutional layer, and a decoder. The encoder on the left has convolutional neuron network layers. Instead of connecting to a flat-fully-connected layer, it connects through an 1x1 convolutional network to the decoder on the right. 1x1 convolutional network has only 1x1 kernel and stride size of one with deep layer of filters. It helps to preserve spatial information from the image. With dimension reduction from 1x1 convolution neuron network, it can save computation resources as well. It's useful for semantic segmentation of images.

Google proposed inception network in GoogLeNet. The main hallmark of the architecture is the improved utilization of the computing resources inside the network. This was achieved by a carefully crafted design that allows for increasing the depth and width of the network while keeping the computational budget constant. To optimize quality, the architectural decisions were based on the Hebbian principle and the intuition of multi-scale processing.

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.

Batch normalization was developed to prevent internal covariate shift, making sure the distribution of the inputs from activation function doesn't change over time due to parameter updates from each batch Internal covariate shift occurs when the distribution of the activations of a layer shifts significantly throughout training. It uses batch statistics to do the normalizing, and then uses the batch normalization parameters gamma and beta to make sure that the transformation inserted in the network can represent the identity transform.

I specified kernel regularizer with tf.contrib.layers.l2_regularizer(scale=0.1) for the convolution network layer to avoid over-fitting.

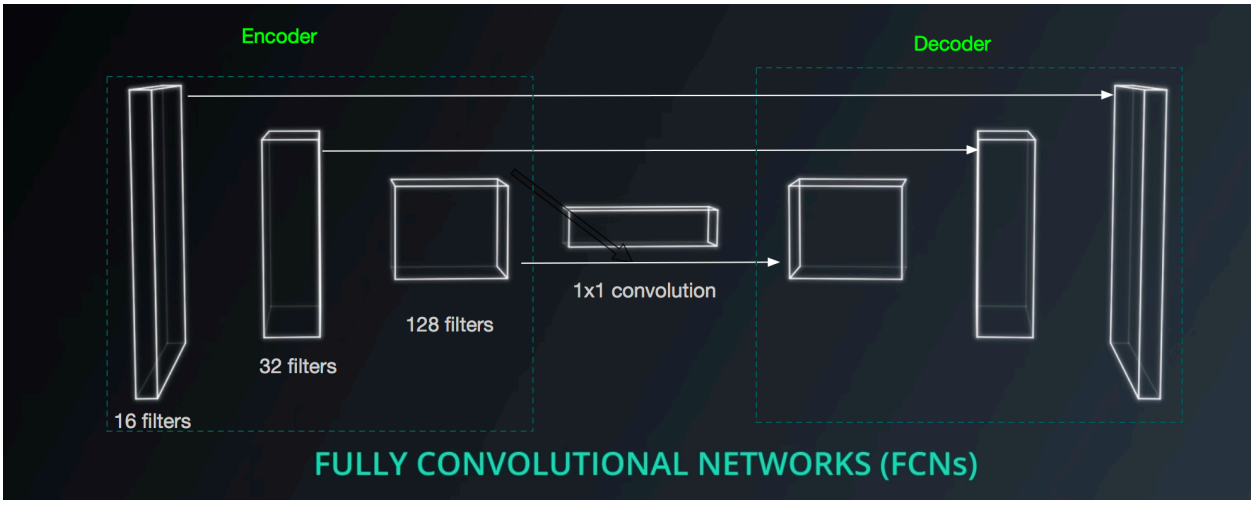


Fig. 4 Fully Convolutional Networks

The decoder layers use 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.

Skipping layer is also used to keep signals from front layers to later decoding layers to improve the resolution of results. This helps identify far away target.

FCN model was implemented in function fcn_model(). Table 1 shows functions to create each layer with dimension and filters. Input images are in 160x160 with color depth of three.

layer	dimension	filters
enc1 = encoder_block(inputs, filters=16, strides=2)	80x80	16
enc2 = encoder_block(enc1, filters=32, strides=2)	40x40	32
enc3 = encoder_block(enc2, filters=64, strides=2)	20x20	64
enc4 = encoder_block(enc3, filters=128, strides=2)	10x10	128
conv_1x1 = conv2d_batchnorm(enc4, 1024, kernel_size=1, strides=1)	10x10	1024
dec1 = decoder_block(conv_1x1, enc3, 128)	20x20	128
dec2 = decoder_block(dec1, enc2, 64)	40x40	64
dec3 = decoder_block(dec2, enc1, 32)	80x80	32
x = decoder_block(dec3, inputs, 16)	160x160	16

Table 1 FCN Layers

More layer can help identify far-away target better.In my experience, with enc4 and the corresponding decoding layer improved total score ~3% than without the enc4 layer.

Finally, after composing from dec3 and skipping layer from inputs, layer x is 160x160 with 16 filters. Layer x has the same dimension as the input layer, which is 160x160.

Hyperparameters

Table 2 shows results that I went through with tries and errors. From the third entry, We can see ~4% improvement in total score by setting kernel regularizer to 0.1, while bias regularizer doesn't seem to help. We use SeparableConv2DKeras and batch normalization, which seems to be good in handling overfitting.

Table 2 Hyperparameters Tries

	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

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.

AWS Training

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. To setup EC2 GPU instance, I opened both ssh and 8888 port used Jupyter Notebook for MyIP in security group,. Then I git clone my project https://github.com/TianHuaBooks/DeepLearning/tree/master/code to the AWS EC2 instance. The aws.sh included in the code directory installed TensorFlow and necessary libraries in an Anaconda environment called Deep.

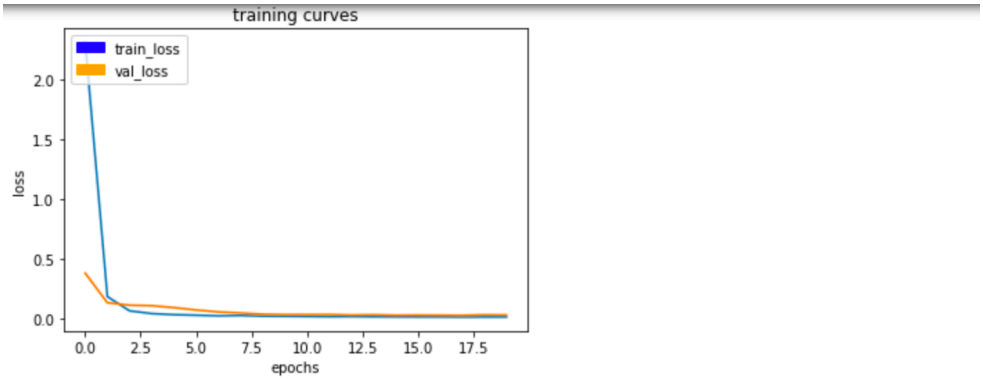
Besides the provided training data, I included additional training data, train31.zip, train37.zip, and train39.zip in the data directory. The included train.zip is too big to upload to my GitHub repository, you can use scp to copy it to EC2 or wget from Udacity repository. Unzip all data zip files under the data directory. You need to rename combined_train to train as Notebook looks for the train directory to load trained images and masks. Validation and evaluation directory names are fine. You can copy images and masks from train31, train37, and train39 to the train directory.

It's highly recommended to use HTTPS in communication between your browser and an EC2 instance. You an purchase an SSL certificate from CA or generate an self-signed SSL certificate for yourself. https://docs.aws.amazon.com/mxnet/latest/dg/setup-jupyter-configure-server.html has instruction to generate a self-signed SSL certificate and IPython password.

Then you can start a browser to connect to Notebook in your EC2 instance by port 8888, like https://[EC2 IP or FQDN]:8888. Your browser might complain certificate if you use a self-signed certificate. You need to allow exception to proceed. After entering a password that you created during the self-signed certificate with Jupyter Notebook, you should be able to run the notebook.

In my testing, I got a final score of 0.417073817766 in the Notebook.

```
The final score is 0.417073817766
```




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

Fig. 5

Training Curve

Running Trained Models on Simulator

With a trained model, I used it to test with simulator. By starting simulator with the follow me option and running terminal with python follower.py on a separate terminal, drone followed hero pretty well. Fig. 6 shows a screenshot of running Simulator with the trained model.

Fig. 6 Running Simulator with the trained model

The FCN mechanism can be applied to identify other targets for examples dog, cat, and panda. The networking layer can be reused with number classes according to number of animal kinds plus one. The images preprocessing needs to be adjusted accordingly.

In Udacity Self-Driving CarND, there is a project to use VGG encoding layers integrating with 1x1 convolutional layer and corresponding decoding layer. In the way, existing training data and models can be reused for scene understanding and other applications.

Future Improvement

- To better identify far away target
- To improve situation when no target is not visible
- Catch more training data with high quality
- Use data augmentation like flip, spraying random noise, etc to create more datasets

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

References

https://people.eecs.berkeley.edu/~jonlong/long_shelhamer_fcn.pdf
https://keras.io/layers/normalization/
https://github.com/dzt109/RoboND-DeepLearning/blob/master/code/model_training.ipynb
https://stats.stackexchange.com/questions/194142/what-does-1x1-convolution-mean-in-a-neural-network
https://arxiv.org/abs/1409.4842
https://en.wikipedia.org/wiki/Generalized_Hebbian_Algorithm