Robotics Nanodegree
Project Report : Follow Me

Guangwei Wang

Oct. 17, 2017

1 Introduction

In this project, I built a deep neural network to identify and track a target in simulation. So-called follow me applications like this are key to many fields of robotics and the very same techniques applied here could be extended to scenarios like advanced cruise control in autonomous vehicles or human-robot collaboration in industry.

The conventional CNN approaches for classification consist a serial of convolutional layers and then followed by fully connected layers to label each pixel with the class of its enclosing object or region, see Fig. 1(a). one of the main problems with using CNNs for segmentation is pooling layers. Pooling layers increase the field of view and are able to aggregate the context while discarding the where information. However, semantic segmentation requires the exact alignment of class maps and thus, needs the where information to be preserved [1], like Fig. 1(b). Hence, by converting the fully-connected layers into convolutional layers and concatenating the intermediate score maps, a fully convolutional network (FCN) is proposed to delineate the boundaries of each object [2].

(a) Conventional classification

(b) Pixels to pixels semantic segmentation

Figure 1: Methods comparison.

2 Fully Convolutional Networks

There are three contributions of fully convolutional network:

1. Replace the fully connected layers by 1x1 convolution layer
2. Upsampling through the transpose convolutional layers
3. Skip connections, which allows to use information from multiple resolution scales

The FCN usually contains two parts: encoder and decoder. The encoder is a series of convolutional layers like VGG, ResNet to extract feature maps from image inputs, the decoder is used for upscaling the output of encoder to generate the same size as the original image. In the conventional CNNs, we always flatten the output of a convolutional layer into a fully connected layer into a 2D tensor, which results in the loss of spatial information, because no information about the location of the pixels is preserved. Hence, the 1x1 convolutions is adopted to replace it between encoder and decoder layer. Moreover, upsampling layers produce coarse segmentation maps because of loss of information during pooling. Therefore, shortcut/skip connections are introduced from higher resolution feature maps, see Fig. 2.

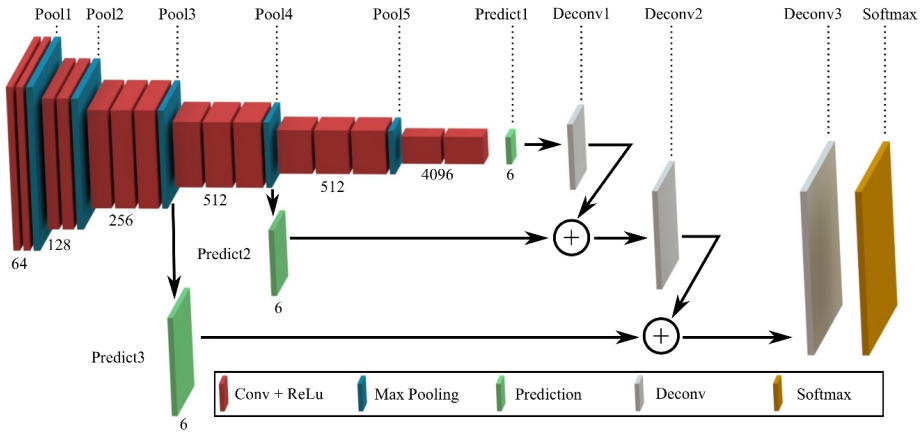


Figure 2: Fully convolutional networks architecture [3].

3 Global Convolutional Netwrok

Although FCN has outperformed a lot of traditional methods on semantic segmentation, several works have been proposed to improve the semantic segmentation task based on it. As mentioned in [4], Semantic segmentation can be considered as a per-pixel classification problem. There are two challenges in this task: 1) classification: an object associated to a specific semantic concept should be marked correctly; 2) localization: the classification label for a pixel must be aligned to the appropriate coordinates in output score map.

A encoder-decoder architecture with very large kernels convolutions is proposed to address the contradictory classification and localization problem in semantic segmentation. However, larger kernels are computationally expensive and have a lot of parameters. Therefore, k x k convolution is approximated with sum of 1 x k + k x 1 and k x 1 and 1 x k convolutions. This module is called as Global Convolutional Network (GCN). As for its architecture, ResNet without any dilated convolutions forms encoder part of the architecture while GCNs and upsampling form

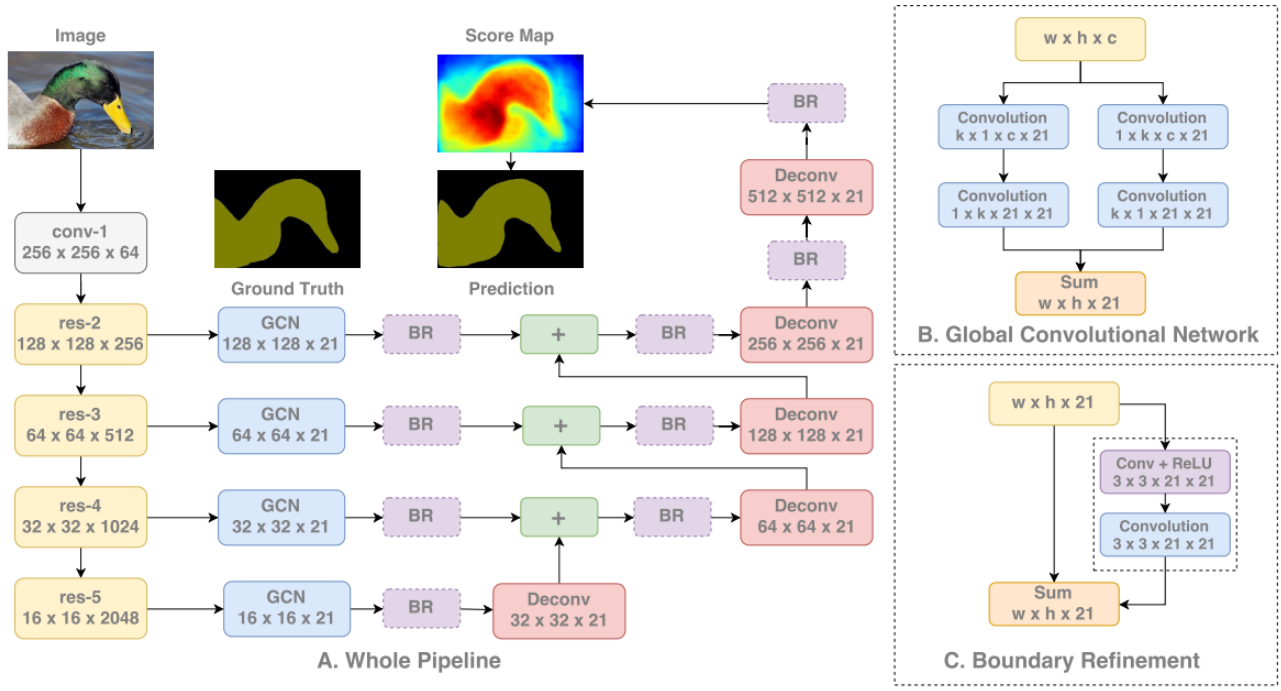


Figure 3: (a) Global convolutional networks architecture. (b) The details of Global Convolutional Network (GCN). (c) Boundary Refinement (BR) block.

decoder. A residual block called Boundary Refinement (BR) is also proposed to refine the object boundaries, the whole architecture of GCN model is shown in Fig. 3.

4 Implementation

4.1 FCN Implementation

My FCN model is implemented as following

```
def fcn_model(inputs, num_classes):
  # Conv layer
  conv1 = conv2d_batchnorm(inputs, filters=32, kernel_size=5, strides=2)
  # Encoder
  enc1 = encoder_block(conv1, filters=64, strides=2)
  enc2 = encoder_block(enc1, filters=128, strides=2)
  enc3 = encoder_block(enc2, filters=256, strides=2)
  # 1x1 convolutional layer
  conv4 = conv2d_batchnorm(enc3, filters=256, kernel_size=1, strides=1)
  # Decoder
  dec5 = decoder_block(conv4, enc2, filters=256)
  dec6 = decoder_block(dec5, enc1, filters=128)
  x = bilinear_upsample(dec6)
  x = separable_conv2d_batchnorm(x, filters=64)
  x = separable_conv2d_batchnorm(x, filters=64)
```

```
  x = bilinear_upsample(x)
  x = separable_conv2d_batchnorm(x, filters=32)
  x = separable_conv2d_batchnorm(x, filters=32)
  return layers.Conv2D(num_classes, 1, activation='softmax', padding='same
    ')(x)
```

4.2 GCN Implementation

4.2.1 GCN and BR algorithm

The GCN function and Boundary Refinement block

```
def GCN(input_layer, filters, ks=5, strides=1):
  '''Globale Convolution Networks'''
  conv_l1 = layers.Conv2D(filters=filters, kernel_size=(ks,1), strides=1,
  padding='same')(input_layer)
  conv_l2 = layers.Conv2D(filters=filters, kernel_size=(1,ks),strides=1,
  padding='same')(conv_l1)
  conv_r1 = layers.Conv2D(filters=filters, kernel_size=(1,ks), strides=1,
  padding='same')(input_layer)
  conv_r2 = layers.Conv2D(filters=filters, kernel_size=(ks,1), strides=1,
  padding='same')(conv_r1)
  output_layer = layers.Add()([conv_l2, conv_r2])
  return output_layer

def BR(input_layer, filters, ks=3, strides=1):
  '''Boundary Refinement'''
  conv1 = layers.BatchNormalization()(input_layer)
  conv1 = layers.Conv2D(filters=filters,kernel_size=ks, strides=strides,
  padding='same', activation='relu')(conv1)
  conv2 = layers.BatchNormalization()(conv1)
  conv2 = layers.Conv2D(filters=filters,kernel_size=ks, strides=strides,
  padding='same', activation='linear')(conv2)
  output_layer = layers.Add()([input_layer, conv2])
  return output_layer

def decoder_gcn(small_ip_layer, large_ip_layer, filters):
  # Upsample the small input layer using the bilinear_upsample() function.
  unsample_layer = BilinearUpSampling2D((2,2))(small_ip_layer)
  concat_layer = layers.Add()([unsample_layer, large_ip_layer])
  # Add BR layers
  output_layer = BR(concat_layer, filters, ks=3, strides=1)
  return output_layer
```

The whole GCN model in this project

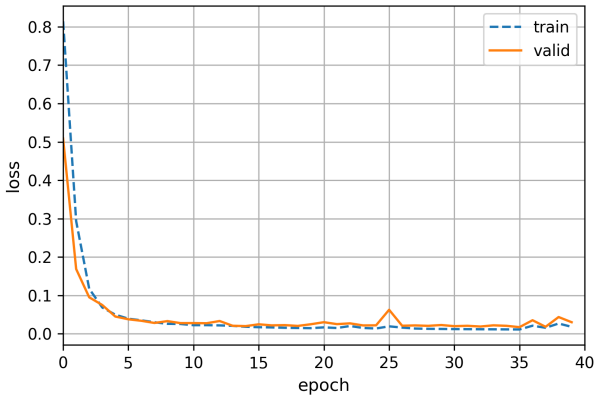
```
def gcn_model(inputs, num_classes):
  ks_gcn = 5
  # Conv layers
  conv1 = conv2d_batchnorm(inputs, filters=64, kernel_size=3, strides=2)
  conv2 = conv2d_batchnorm(conv1, filters=128, kernel_size=3, strides=2)
  conv3 = conv2d_batchnorm(conv2, filters=256, kernel_size=3, strides=2)
  conv4 = conv2d_batchnorm(conv3, filters=512, kernel_size=3, strides=2)
  # GCN and BR layers
```

```
gcn1 = GCN(conv2, filters=num_classes, ks=ks_gcn, strides=1)
gcn1 = BR(gcn1, filters=num_classes, ks=3, strides=1)
gcn2 = GCN(conv3, filters=num_classes, ks=ks_gcn, strides=1)
gcn2 = BR(gcn2, filters=num_classes, ks=3, strides=1)
gcn3 = GCN(conv4, filters=num_classes, ks=ks_gcn, strides=1)
gcn3 = BR(gcn3, filters=num_classes, ks=3, strides=1)
# Deconv layers
dec1 = decoder_gcn(gcn3, gcn2, filters=num_classes)
dec2 = decoder_gcn(dec1, gcn1, filters=num_classes)
output_layer = BilinearUpSampling2D((2,2))(dec2)
output_layer = BR(output_layer, filters=num_classes, ks=3, strides=1)
output_layer = BilinearUpSampling2D((2,2))(output_layer)
x = BR(output_layer, filters=num_classes, ks=3, strides=1)
return layers.Conv2D(num_classes, 1, activation='softmax', padding='same
')(x)
```

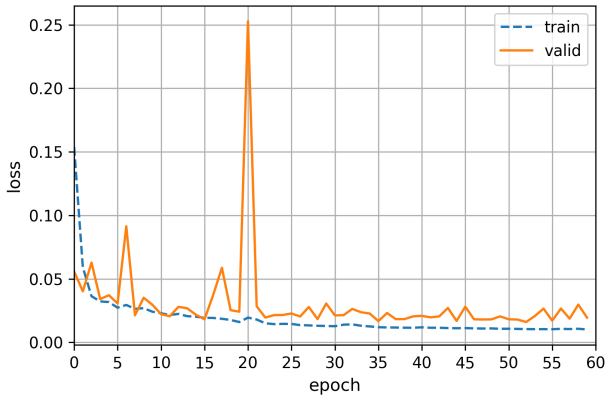
5 Parameters Chosen

Parameters selecting depends on trail and error method, main parameters are the following

- Learning rate = 0.002. Nadam optimizer is adopted here, the learning rate is selected as the default value in its paper.
- Epoch = 50. Because dropout is not adopted, and there would be overfitting when epochs larger than 50.
- Batch size = 16. The batch size 20 is used in FCN paper, so a similar one is selected.
- Workers = 4. p2.xlarge instance on AWS has 4 vCPU.



(a) Training results of FCN model with 40 epochs



(b) Training results of FCN model with 40 epochs

Figure 4: Model training results.

6 Results

The FCN model has a heavy computational burden. Each epoch costs about 254s with the above proposed model structure, the final score 0.432 is achieved with 40 epochs, as shown in Fig. 4(a). As for the GCN model, it has a faster training speed, which is about 140s for each epoch. The final score of 0.454 is achieved with 50 epochs, as shown in Fig. 4(b). The slight higher score is achieved with less training time.

Some prediction samples are illustrated in Fig. 5.

7 Discussion

According to evaluation scores (IoU1, IoU2 and IoU3), IoU3 has a low value while IoU1 has achieved score of 0.90. IoU3 measures how well the neural network can detect the target from far away, so some new images are collected to increase samples of the target from far away. In addition, the images without people are removed and then all the training data are also flipped to generating more data. IoU3 is a quite strict metric, the target from far away would be vague pixels in particular with a quite low resolution image in this project. Only IoU score of 0.233 is achieved finally.

The recording image is not match with images in follow me mode, as shown in Fig. 6. The top half of recording data is sky area, and we cannot crop it any more due to there is no extra margin for their size.

Large kernel didn't have a obvious improvement for GCN model here, the reason might be the image inputs is too small so that large kernel cannot work well.

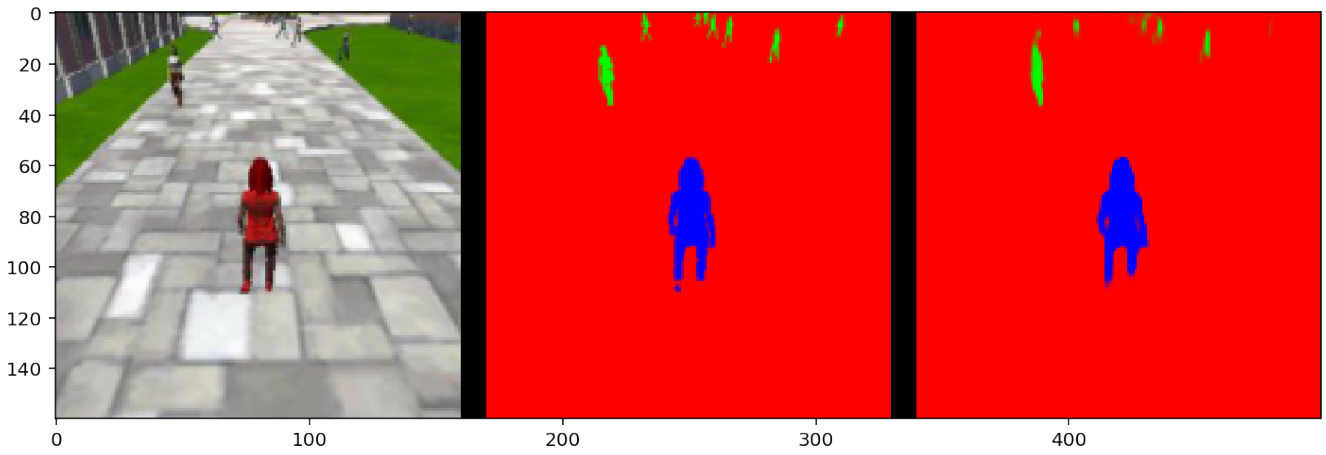
The encoder of original FCN and GCN models are based on pre-trained CNNs networks, such as VGG16 and Resnet152 model. The models built in this project are trained from scratch with a simple CNN layers. Finetuned method based on pre-trained networks would have a better performance.

The model test in follow me mode did not be conducted, because the python stopped working when I run 'python follower.py' on Win10 and the Simulator cannot work on VM ubuntu.

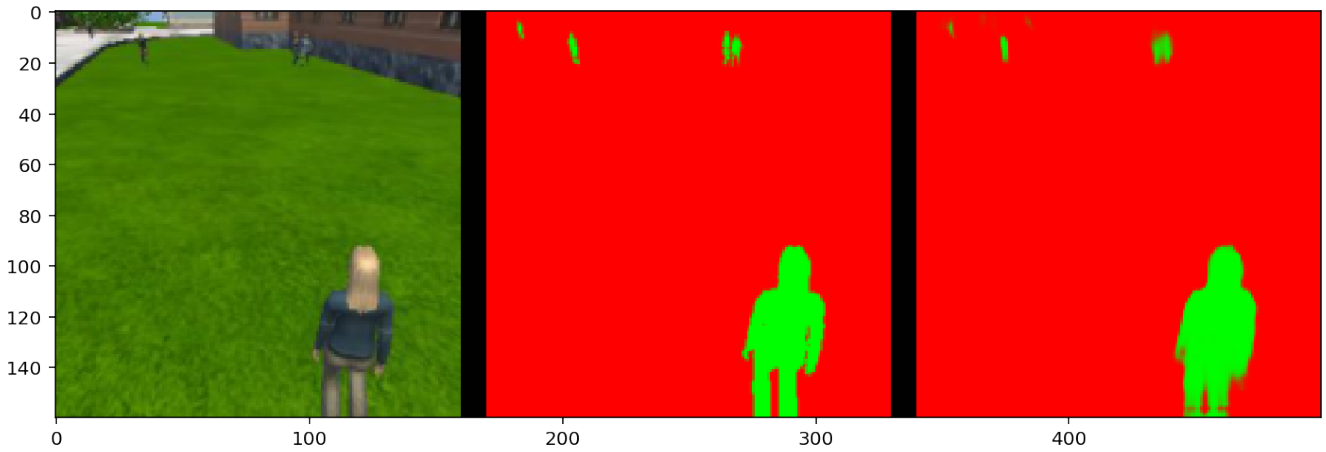
8 Improvements

There are lots of works to improve further, just to list some.

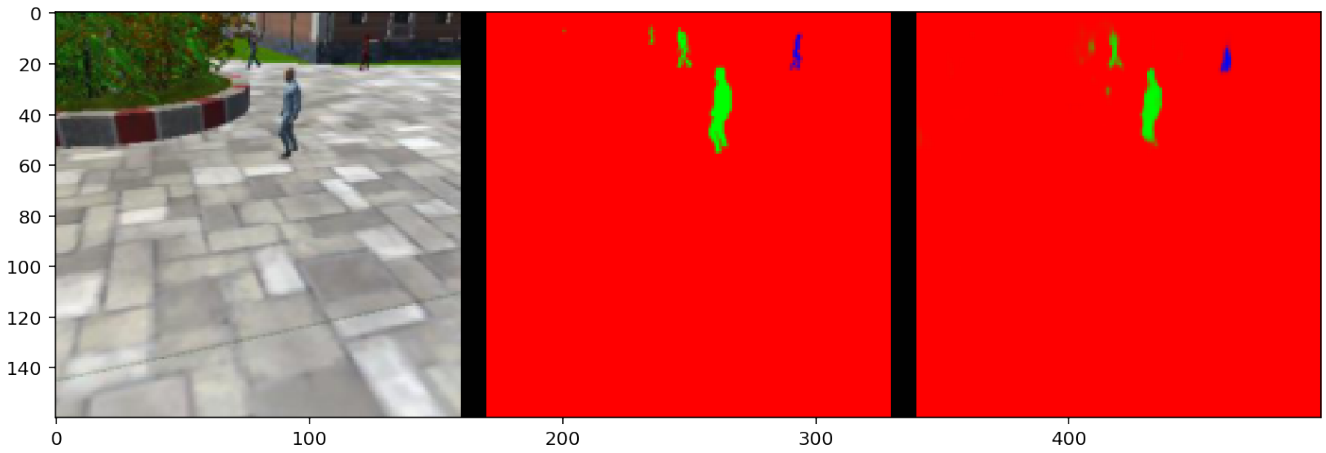
- More and high-quality training data are preferred.
- Pre-trained networks, such as GoolgeNet and Resnet, are preferred to implemented to generate feature maps in the future.
- Higher resolution of image inputs would achieve better performance but with heavier computational burden.
- Combine train and valid data together and use cross validation.



(a) result while following the target



(b) result while at patrol without target



(c) result while at patrol with target

Figure 5: Prediction results.

(a) Image from follow me mode

(b) Image from recording mode

Figure 6: Image difference between follow me and recording modes.

References

[1] S. Chilamkurthy, "A 2017 guide to semantic segmentation with deep learning," Tech. Rep., 2017.

[2] J. Long, E. Shelhamer, and T. Darrell, "Fully convolutional networks for semantic segmentation," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2015, pp. 3431–3440.

[3] L. Tai, H. Ye, Q. Ye, and M. Liu, "Pca-aided fully convolutional networks for semantic segmentation of multi-channel fmri," in *Advanced Robotics (ICAR), 2017 18th International Conference on*. IEEE, 2017, pp. 124–130.

[4] C. Peng, X. Zhang, G. Yu, G. Luo, and J. Sun, "Large kernel matters–improve semantic segmentation by global convolutional network," *arXiv preprint arXiv:1703.02719*, 2017.