RoboND-Follow-Me-Project

Abstract

In this project we train a Fully Convolutional Network (FCN) to be able to identify a target in a simulated drone environment.

A Fully Convolutional Neural Network (FCN) consist of three parts encoder which is a series of convolution neural network followed by 1 x 1 convolutional then last part is decoder that up-sample the layers of encoder. adds Skip Connections which allow us to view the convolutional filters at different dimensions. This gives us a better special understanding of the inputs at higher resolutions.

How each layer of the network architecture and the role that it plays in the overall network?

A Fully Convolutional Network (FCN) is a network that can be used to run inference on every single pixel in an image and apply this technology to train super-fast semantic network to real-time pixelwise classification on the images.

FCN's are built up of a set of convolution layers ending with a 1x1 convolution. This is the first half of the network called the encoder. The shape of the 1x1 convolution will be a small in height and width (pixels) but with lots of depth (filters). The second half of the network is the decoder.

Which case have we use Fully convolutional layers?

In classic convolution network with **Fully Connected Layers** final layers, the size of the input constrained by the size of fully connected layer.

However, **FCNs** don't care about the size of input image (work on any image of any size).

Why do we name it "encoder". Encoding, in general, means compressing with or without loss of information.?

Because it Encodes the features from the image in a series of Separable Convolutions layers and used these layers on our project.

What are the Separable Convolutions layers?

Separable Convolutions: is a technique that reduces the number of parameters needed, thus increasing efficiency for the encoder network with improved runtime performance and reducing overfitting because of fewer parameters. Beside Batch normalization is an additional way to optimize network training and help network to learn.

Which information do we lose? (for example, you have a picture of a person, a hero in our case. If you were trying to compress the image, what would you leave out, what information would you unify?)

Fully connected layer used to flatten output to 2D as a result of that we **loss** spatial information, because there **is no information** about the **location** of the **pixels**.

The reason for using a **1x1 convolution** is that it acts just like a fully connected layer (reduce dimensionality), but it remains the spatial information of all the pixels (4D).

How it should be used a 1x1 convolution? And why not used a fully connected layer?

Fully connected layer used to flatten output to 2D as a result of that we loss spatial information, because there is no information the location of the pixels.

The reason for using a **1x1 convolution** is that it acts just like a fully connected layer (reduce dimensionality), but it remains the spatial information of all the pixels (4D).

Add another **advantage** of using **1x1 convolution** that **during inference** (testing your model) you can feed images of any size into your trained network.

Note that just because the last part of the encoder is a 1x1 convolution, this doesn't mean that the output shape will be 1 pixel high and 1 pixel wide. The 1x1 convolution height and width will be the same as the layer preceding as output of fully connected layer

Which information does the decoder recover (example in our case? How will the decoded picture look like compared to the original)?

Decoder: up-scale the output of the encoder such that it's the same of original image.

It is built up of layers of **transposed convolutions** whose outputs increase the height and width of the input while decreasing its depth. By multiplying each pixel of your input with kernel or filter.

Transposed Convolutions as one way of upsampling layers to higher dimensions or resolutions. There are several ways to achieve upsampling.

I used bilinear interpolation method.

Bilinear upsampling is a resampling technique that utilizes the weighted average of four nearest known pixels, located diagonally to a given pixel, to estimate a new pixel intensity value. The weighted average is usually distance dependent.

This how the decoder picture look like compared to the original?

Original image with (160,160,3) the output of decoder (160,160,32)

Which information is lost?

One effect on convolutional or encoding is reducing dimensionality,

As we narrow down the scope of the input image by looking closely at some pictures as a result of that loss the bigger pictures (information will be lost).

But if we using **Skip connections** that are also used at all points in the decoder which in this case **concatenates** the input to the encoder layer of the same size (output of one layer) with the decoded layer (non-adjacent layer). This allows the model to learn from multiple resolutions which helps it retain details that could have been lost. Thus, increasing our accuracy on our segmentation.

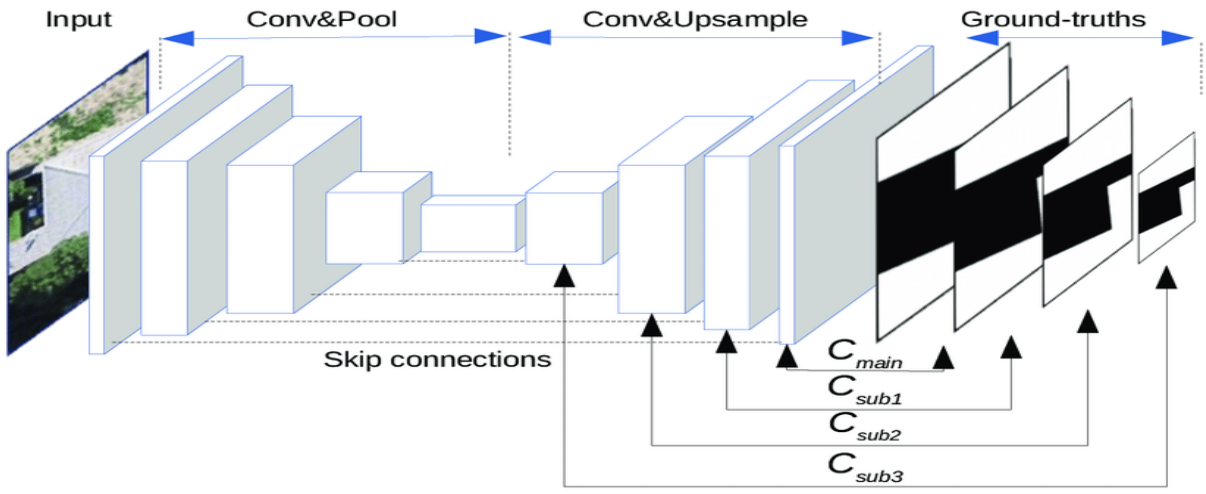
Model

I decided to use a filter of 32 and created 3 encoder layers, 1 convolutional layer, and 3 decoder layers. I created 3 encoder and decoder layers.

	Inputs	H- size	W-size	D-size
Features layer	**?**	**160**	**160**	**3**
Encoder-layer1	**?**	**80**	**80**	**32**
Encoder-layer2	**?**	**40**	**40**	**64**
Encoder-layer3	**?**	**20**	**20**	**128**
convolution 1 x 1	**?**	**20**	**20**	**128**
Decoder-layer1	**?**	**40**	**40**	**128**
Decoder-layer2	**?**	**80**	**80**	**64**
Decoder-layer	**?**	**160**	**160**	**32**

This how the process works?

Input layer → Max_pooling→Sparable_Covnet→1X1 Covnet →Transposed Covnet →Output



Network Parameter & Training at Udacity's GPU

I lost the data I had on some of the models, including some images, but I initially started by tweeking the learning rate.

1-The initial **learning_rate** was "0.001" with **batch_size** "128"→ **final_score**: 0.3504

2-I tweaked **learning_rate** "0.005" and settled on **batch_size** "20" → **final_score**: 0.3912

3-I tweaked again **learning_rate** "0.007" and **batch_size** "20" → **final_score**: 0.4080

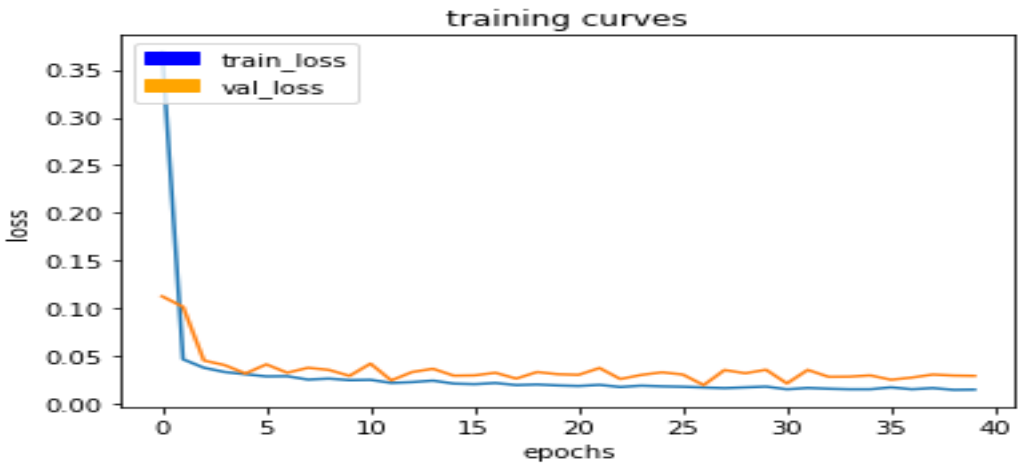
the most accurate. I started the as being small enough for a stable amount of data. I felt that the smaller batch sizes would be more consistent in training and decrease the validation loss. I initially set the number of epochs to 10 then 40 epochs it starts to stagnate and you see very little change. I set the steps_per_epoch to be the number of images / batch size approximately 250.

These are the current parameters I got to.

learning_rate = 0.0007
batch_size = 20
num_epochs = 40
steps_per_epoch = 250
validation_steps = 50
workers = 2

with loss: 0.0148 - val_loss: 0.0292

with final score : 0.408059713897



Performance on Simulator

When operating on the simulator, the model is able to pick up the hero from a reasonable distance, and once it closes in never lose sight of it. This model had 100% performance once it got close to the hero (evidenced by 539 true positives), and thus never lost it even in big crowds.

The models performance could be improved by using more training data, particularly when the hero is far away. Other than the model performs really well.

I believe this model could be used to also identify and track and another object (based on suggestions of using this model for tracking a dog or car) but training data specific too these objects would be required.