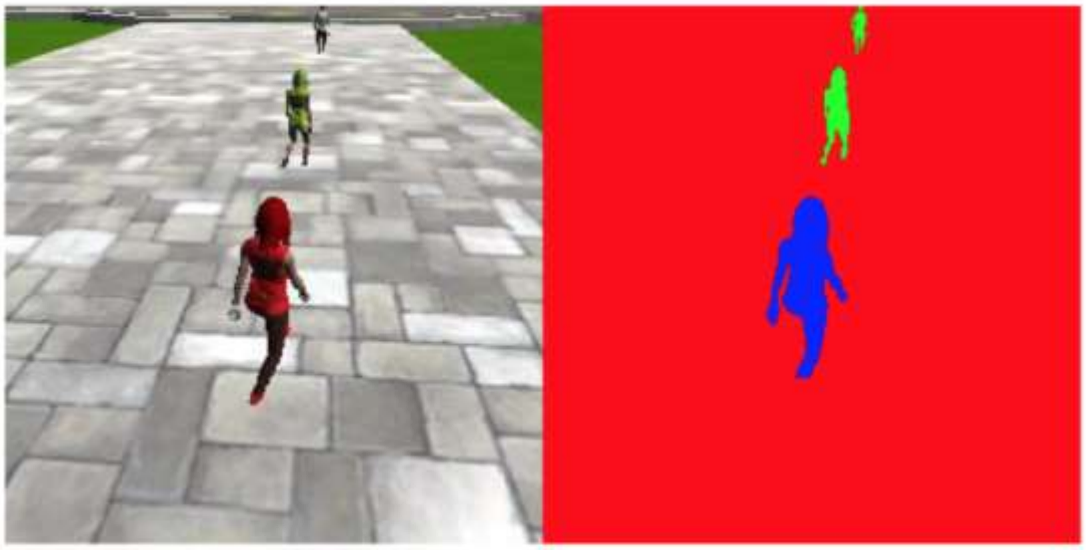
RoboND1- Lesson32 – Semantic Segmentation

05-Lab Semantic Segmentation

Lab Overview

Now that you've been introduced to the basics of segmentation networks, it's time for a concrete robotics example with a Jupyter notebook. In this lab, you'll train a deep neural network to identify a target person from images produced by a quadcoptor simulator. Once you have a trained network, you can use it to find the target in new images by **inference**.



In the final **Follow Me** project, the inference step must run continuously in simulation, just as quickly as the copter supplies images to it. Some additional code abstractions and performance enhancements are introduced in this lab that are needed when moving to the simulation, so be sure to do the lab before moving on to the project!

Specifically, the following key topics are covered in the next few concepts:

- Getting Started
- Keras layers in TensorFlow
- Encoder with Separable Convolutions
- Batch Normalization
- Decoder with Bilinear Upsampling

This lab uses actual data from the simulator, and at the end of it, you will have a nominally working network to incorporate into the **Follow Me** project. It will be up to you to maximize the network performance with additional layers, parameter tuning and even collecting more data!

06-Lab: Getting started

Setup

Make sure you have followed the instructions in the **classroom** to setup your environment or have followed along in the previous lab notebook setups. Please note that dependencies may be missing with a old setup. Check **here** for package requirements and manual installation of them.

Clone the Repository and Run the Notebook

Run the commands below to clone the lab repository and then run the notebook:

```
git clone https://github.com/udacity/RoboND-Segmentation-Lab.git
# Make sure your conda environment is activated!
jupyter notebook
```

The Jupyter interface will open in your browser. You can then access the cloned repo and the Jupyter Notebook from there. We are specifically working with the `segmentation_lab.ipynb` which can be found in following path `code/segmentation_lab.ipynb`.

Download the Data

After you have the notebook up and running be sure to download the **training** and **validation** data. Then put the respective folders in the `/data` directory.
Once the notebook is up and running and the data is downloaded, you can follow the instructions in the notebook and fill out the required pieces of code marked by `TODOs`. It is important to take time and read the comments in the notebook. On top of following along with the classroom lessons for guidance on how to fill out the `TODOs` be sure to check the notebook for relevant information as well. By the end you will have your first basic implementation of the network needed to get the project running!
It is important to note that some computer platforms may take up to 3 hours to train the network, depending on a few factors.

The recommended strategy for dealing with this problem is to complete the coding and debugging on your local system before moving to a faster system for the training portion. Once your network is running correctly you can then launch your notebook from your AWS instance in order to speed up training times. More information on running a Jupyter Notebook from AWS can be found **here.**

07 Lab: Keras

Q&A

Why Keras, after learning all about TensorFlow?

Keras allows you to spend more time interfacing with the data and particular actions you want to employ in your network without having to focus on a lot of little lower level thinking. It is, however, recommended that you understand the basics of neural networks first (which is what we covered in lessons before this) because this makes your use of Keras much more efficient.

What are some of the changes we will notice using Keras?

- The training routine is simplified, and supplied by a standard interface as seen **here.**
- You don't have to manage the tensorflow session yourself.
- The data generators have a standard interface, that are well documented **here.**
- The layers are classes, that have the syntax `layer()(prev_layer)`, instead of a functional interface where it is `layer(prev_layer)`, the naming convention for layers is pretty standard between them.
- Keras has additional layers for things like upsampling and other small features, whereas in tensorflow these are scattered about.
- If you want to add new functionality to a network then the functionality needs to be wrapped in a **custom layer** or written as a **lambda layer.**

08 Lab: Encoder

FCN: Encoder

In the **previous lesson**, the FCN (Fully Convolutional Network) architecture was introduced. Recall that an FCN is comprised of an encoder and decoder. The encoder portion is a convolution network that reduces to a deeper *1x1 convolution layer*, in contrast to a flat fully connected layer that would be used for basic classification of images. This difference has the effect of preserving spacial information from the image.
Separable Convolutions, introduced here, is a technique that reduces the number of parameters needed, thus increasing efficiency for the encoder network.

Separable Convolutions

Separable convolutions, also known as depthwise separable convolutions, comprise of a convolution performed over each channel of an input layer and followed by a 1x1 convolution that takes the output channels from the previous step and then combines them into an output layer.

This is different than regular convolutions that we covered before, mainly because of the reduction in the number of parameters. Let's consider a simple example.

Suppose we have an input shape of 32x32x3. With the desired number of 9 output channels and filters (kernels) of shape 3x3x3. In the regular convolutions, the 3 input channels get traversed by the 9 kernels. This would result in a total of 9*3*3*3 features (ignoring biases). That's a total of 243 parameters.

In case of the separable convolutions, the 3 input channels get traversed with 1 kernel each. That gives us 27 parameters (3*3*3) and 3 feature maps. In the next step, these 3 feature maps get traversed by 9 1x1 convolutions each. That results in a total of 27 (9*3) parameters. That's a total of 54 (27 + 27) parameters! Way less than the 243 parameters we got above. And as the size of the layers or channels increases, the difference will be more noticeable.

The reduction in the parameters make separable convolutions quite efficient with improved runtime performance and are also, as a result, useful for mobile applications. They also have the added benefit of reducing overfitting to an extent, because of the fewer parameters.

Additional Resources

- **Xception: Deep Learning with Depthwise Separable Convolutions**
- **Medium Post: An Introduction to different Types of Convolutions in Deep Learning**

Coding Separable Convolutions

An optimized version of separable convolutions has been provided for you in the `utils` module of the provided repo. It is based on the `tf.contrib.keras` **function definition** and can be implemented as follows:

```
output = SeparableConv2DKeras(filters, kernel_size, strides,
                             padding, activation)(input)
```

Where,

`input` is the input layer,
`filters` is the number of output filters (the depth),
`kernel_size` is a number that specifies the (width, height) of the kernel,
`padding` is either "same" or "valid", and
`activation` is the activation function, like "relu" for example.
The following is an example of how you can use the above function:

```
output = SeparableConv2DKeras(filters=32, kernel_size=3, strides=2,
                             padding='same', activation='relu')(input)
```

In the lab's notebook, the above has already been included for you in the `separable_conv2d_batchnorm()` function. You will use this function to define one or more layers for your encoder. Just like you used regular convolutional layers in the CNN lab.

But before that, there's another important part of this function that we will cover in the upcoming section. **Batch Normalization**.

09Lab: Batch Normalization

Batch Normalization

In the **TensorFlow for Deep Learning lesson**, Vincent Vanhoucke explained the importance of normalizing our inputs and how this helps the network learn. Batch normalization is an additional way to optimize network training. Batch normalization is based on the idea that, instead of just normalizing the inputs to the network, we normalize the inputs to layers within the network. It's called "batch" normalization because during training, we normalize each layer's inputs by using the mean and variance of the values in the current mini-batch.

A network is a series of layers, where the output of one layer becomes the input to another. That means we can think of any layer in a neural network as the first layer of a smaller network.

For example, imagine a 3 layer network. Instead of just thinking of it as a single network with inputs, layers, and outputs, think of the output of layer 1 as the input to a two layer network. This two layer network would consist of layers 2 and 3 in our original network.

Likewise, the output of layer 2 can be thought of as the input to a single layer network, consisting only of layer 3. When you think of it like that - as a series of neural networks feeding into each other - then it's easy to imagine how normalizing the inputs to each layer would help. It's just like normalizing the inputs to any other neural network, but you're doing it at every layer (sub-network).

Additional Resources

- **Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift**

Coding Batch Normalization

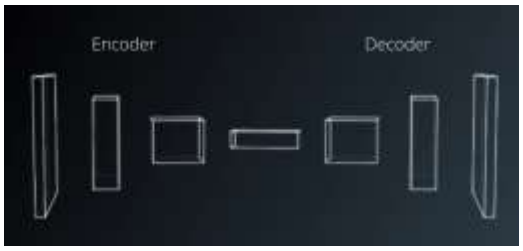
In `tf.contrib.keras`, batch normalization can be implemented with the following **function definition**:

```
from tensorflow.contrib.keras.python.keras import layers
output = layers.BatchNormalization()(input)
```

In your notebook, the `separable_conv2d_batchnorm()` function adds a batch normalization layer after the separable convolution layer. Introducing the batch normalization layer presents us with quite a few advantages. Some of them are:

- **Networks train faster** – Each training iteration will actually be slower because of the extra calculations during the forward pass. However, it should converge much more quickly, so training should be faster overall.
- **Allows higher learning rates** – Gradient descent usually requires small learning rates for the network to converge. And as networks get deeper, their gradients get smaller during back propagation so they require even more iterations. Using batch normalization allows us to use much higher learning rates, which further increases the speed at which networks train.
- **Simplifies the creation of deeper networks** – Because of the above reasons, it is easier to build and faster to train deeper neural networks when using batch normalization.
- **Provides a bit of regularization** – Batch normalization adds a little noise to your network. In some cases, such as in Inception modules, batch normalization has been shown to work as well as dropout.

10Lab: Decoder



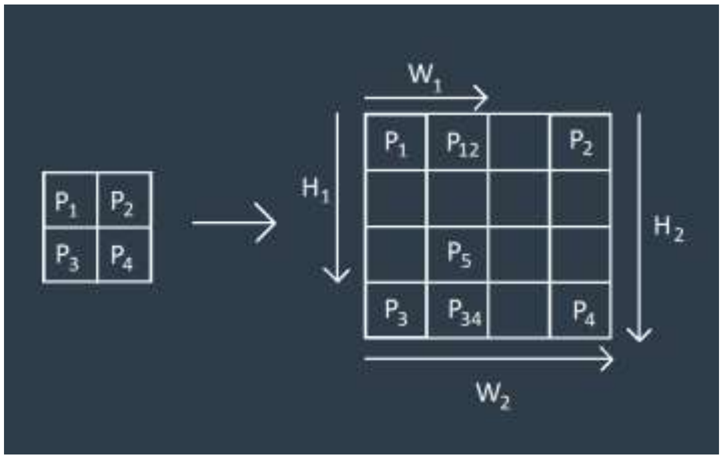
FCN: Decoder

Earlier, we covered transposed convolutions as one way of upsampling layers to higher dimensions or resolutions. There are, in fact, several ways to achieve upsampling. In this section, we will cover another method called bilinear upsampling or bilinear interpolation.

Bilinear Upsampling

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.

Let's consider the scenario where you have 4 known pixel values, so essentially a 2x2 grayscale image. This image is required to be upsampled to a 4x4 image. The following image gives a better idea of this process.



The unmarked pixels shown in the 4x4 illustration above are essentially whitespace. The bilinear upsampling method will try to fill out all the remaining pixel values via interpolation. Consider the case of P_5 to understand this algorithm.
We initially calculate the pixel values at P_{12} and P_{34} using linear interpolation.
That gives us,

$P_{12} = P_1 + W_1*(P_2 - P_1)/W_2$
and

$P_{34} = P_3 + W_1*(P_4 - P_3)/W_2$
Using P_{12} and P_{34}, we can obtain P_5:
$P_5 = P_{12} + H_1*(P_{34} - P_{12})/H_2$
For simplicity's sake, we assume that $H_2 = W_2 = 1$
After substituting for P_{34} and P_{12} the final equation for the pixel value of P_5 is:
$P_5 = P_1*(1 - W_1)*(1 - H_1) + P_2*W_1*(1 - H_1) + P_3*H_1*(1 - W_1) + P_4*W_1*H_1$
While the math becomes more complex, the above technique can be extended to RGB images as well.

QUIZ QUESTION

Given the equation above, what would be the value for P_5 if $P_1 = 150$, $P_2 = 200$, $P_3 = 65$, $P_4 = 100$, $W_1 = 0.5$, $H_1 = 0.7$?

- ○ 150.5
- 110.25
- ○ 120.25
- ○ 160.5

SUBMIT

Coding Bilinear Upsampler

An optimized version of a bilinear upsampler has been provided for you in the `utils` module of the provided repo and can be implemented as follows:

```
output = BilinearUpSampling2D(row, col)(input)
```

Where,

`input` is the input layer,
`row` is the upsampling factor for the rows of the output layer,
`col` is the upsampling factor for the columns of the output layer, and
`output` is the output layer.
The following is an example of how you can use the above function:

```
output = BilinearUpSampling2D((2,2))(input)
```

The bilinear upsampling method does not contribute as a learnable layer like the transposed convolutions in the architecture and is prone to lose some finer details, but it helps speed up performance.

1Lab: Layer Concatenation

In the last lesson, you covered skip connections which are a great way to retain some of the finer details from the previous layers as we decode or upsample the layers to the original size. Previously, we discussed one way to carry this out, using an element-wise addition operation to add two layers. We will go over another simple technique now where we will concatenate two layers instead of adding them.

Concatenating two layers, the upsampled layer and a layer with more spatial information than the upsampled one, presents us with the same functionality. Implementing this functionality is quite straightforward as well.

Using the `tf.contrib.keras` **function definition**, it can be implemented as follows:

```
from tensorflow.contrib.keras.python.keras import layers
output = layers.concatenate(inputs)
```

Where,

`inputs` is a list of the layers that you are concatenating.
In the lab's notebook, you will be able to implement the above in the `decoder_block()` function using a similar format as follows:

```
output = layers.concatenate([input_layer_1, input_layer_2])
```

One added advantage of concatenating the layers is that it offers a bit of flexibility because the depth of the input layers need not match up unlike when you have to add them. Which helps simplify the implementation as well.

While layer concatenation in itself is helpful for your model, it can often be better to add some regular or separable convolution layers after this step for your model to be able to learn those finer spatial details from the previous layers better. In the lab, you will be able to try this out yourself!

In the next section, we will cover an important metric that can help gauge the performance of a semantic segmentation network!

12,13 IOU

TensorFlow IoU

Let's look at the `tf.metrics.mean_iou` function. Like all the other **TensorFlow metric functions**, it returns a Tensor for the metric result and a Tensor Operation to generate the result. In this case it returns `mean_iou`for the result and `update_op` for the update operation. Make sure to run `update_op` before getting the result from `mean_iou`.

```
sess.run(update_op)
sess.run(mean_iou)
```

The other characteristic of **TensorFlow metric functions** is the usage of **local TensorFlow variables**. These are temporary TensorFlow Variables that must be initialized by running `tf.local_variables_initializer()`. This is similar to `tf.global_variables_initializer()`, but for different TensorFlow Variables.

IoU Quiz

In this quiz, you'll use this **documentation** to apply mean IoU to an example prediction.

```
import oldtensorflow as tf


def mean_iou(ground_truth, prediction, num_classes):
    # TODO: Use `tf.metrics.mean_iou` to compute the mean IoU.
    iou, iou_op = tf.metrics.mean_iou(ground_truth, prediction, num_classes)
    return iou, iou_op


ground_truth = tf.constant([
    [0, 0, 0, 0],
    [1, 1, 1, 1],
    [2, 2, 2, 2],
    [3, 3, 3, 3]], dtype=tf.float32)
prediction = tf.constant([
    [0, 0, 0, 0],
    [1, 0, 0, 1],
    [1, 2, 2, 1],
    [3, 3, 0, 3]], dtype=tf.float32)

# TODO: use `mean_iou` to compute the mean IoU
iou, iou_op = mean_iou(ground_truth, prediction, 4)

with tf.Session() as sess:
        sess.run(tf.global_variables_initializer())
        # need to initialize local variables for this to run `tf.metrics.mean_iou`
        sess.run(tf.local_variables_initializer())

        sess.run(iou_op)
        # should be 0.53869
        print("Mean IoU =", sess.run(iou))
```