

Lecture note 7: Playing with convolutions in TensorFlow

“CS 20SI: TensorFlow for Deep Learning Research” (cs20si.stanford.edu)

Prepared by Chip Huyen (huyenn@stanford.edu)

This lecture note is an unfinished draft. Proceed with care!

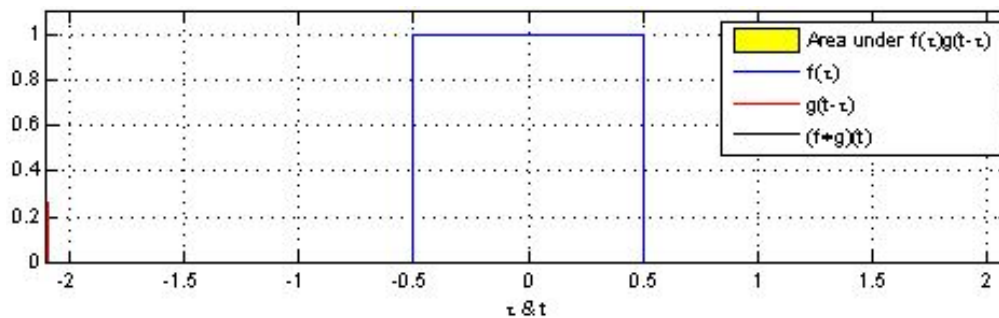
Last class, we had a wonderful guest lecture by Justin Johnson. I hope that by now, you're more or less familiar with common layers of a convolutional neural networks. Today, we will make a (crude) attempt to understand convolution and what kind of support TensorFlow has for convolutions.

Understanding convolutions

If you've taken maths or physics, you're probably familiar with this term. Oxford dictionary define the mathematical term convolution as:

a function derived from two given functions by integration that expresses how the shape of one is modified by the other.

And that's pretty much what convolution means in the neural networks setting. Convolution is how the original input (in the first convolutional layer, it's part of the original image) is modified by the kernel (or filter). To better understand convolutions, you can refer to this [wonderful blog post](#) by Chris Olah at Google Brain.



[Gif from Wikipedia](#)

TensorFlow has great support for convolutional layers. The most popular one is `tf.nn.conv2d`.

```
tf.nn.conv2d(input, filter, strides, padding, use_cudnn_on_gpu=None, data_format=None,
              name=None)
```

Input: Batch size x Height x Width x Channels

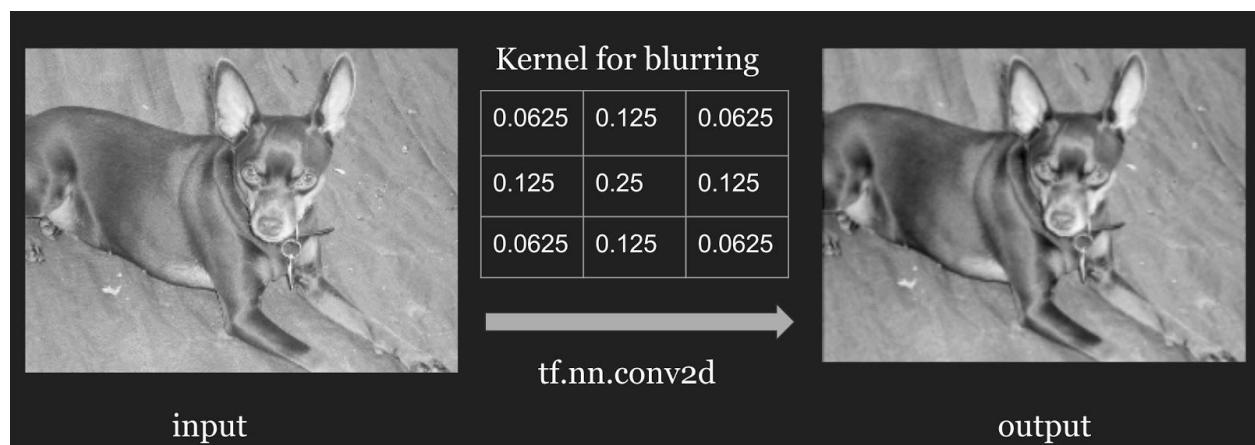
Filter: Height x Width x Input Channels x Output Channels
(e.g. `[5, 5, 3, 64]`)

Strides: 4 element 1-D tensor, strides in each direction
(often `[1, 1, 1, 1]` or `[1, 2, 2, 1]`)

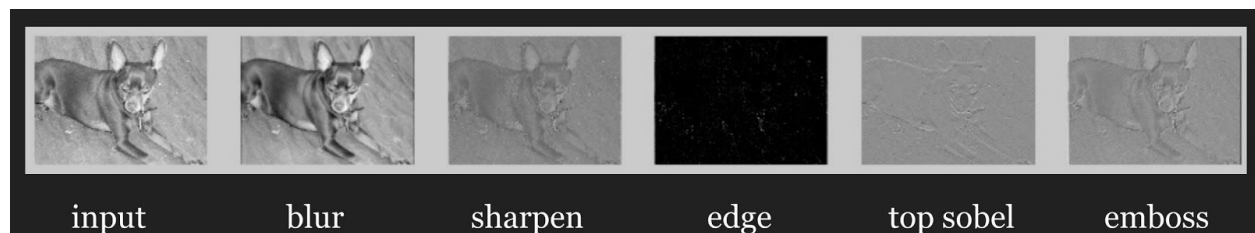
```
Padding: 'SAME' or 'VALID'  
Data_format: default to NHWC
```

Generally, for strides, you don't want to use any number other than 1 in the first and the fourth dimension,

Because of this property of convolution, we can do convolutions without training anything. We can simply choose a kernel and see how that kernel affects our image. For example, the kernel often used for blurring an image is as below. You can do the element-wise multiplication to see how this kernel helps blurring an image.



Just as a fun exercise, you can see several other popular kernels in the kernels.py file on the class GitHub repository, and see how to use them in 07_basic_filters.py
let's see what the Blurring kernel does to this super cute image of a Chihuahua.



There are also several other built-in convolutional operations. Please refer to [the official documentation](#) for more information.


```
conv2d: Arbitrary filters that can mix channels together.  
depthwise_conv2d: Filters that operate on each channel independently.  
separable_conv2d: A depthwise spatial filter followed by a pointwise filter.
```

In this case, we hard code our kernels. When training a convnet, we don't know what the values for our kernels and therefore have to figure them out by learning them. We'll go through the process of learning the kernels through a simple convnet with our old friend MNIST.

Convnet on MNIST

We've done logistic regression on MNIST and the result is abysmal. Since MNIST dataset contains of just images, let's see how much better we can do with convnet on MNIST.

For MNIST, we will be using two convolutional layers, each followed by a relu and a maxpool layers, and one fully connected layer. See the example on GitHub under the name 07_convnet_mnist.py

<div>Original Image 28 x 28 x 1</div> 	<div>Conv1 Filter: 5 x 5 x 1 x 32 Stride: 1, 1, 1, 1 Out: 28 x 28 x 32 Relu Maxpool (2 x 2 x 1) Out: 14 x 14 x 32</div>	<div>Conv2 Filter: 5 x 5 x 32 x 64 Stride: 1, 1, 1, 1 Out: 14 x 14 x 64 Relu Maxpool (2 x 2 x 1) Out: 7 x 7 x 64</div>	<div>Fully connected W: 7*7*64 x 1024 Out: 1 x 1024 Relu Out: 1 x 1024</div>	<div>Softmax W: 1024 x 10 Out: 1 x 10 Softmax 1 x 10</div>
---	---	--	---	---

Variable scope

Since we'll be dealing with multiple layers, it's important to introduce variable scope. Think of a variable scope something similar to a namespace. A variable name 'weights' in variable scope 'conv1' will become 'conv1-weights'. The common practice is to create a variable scope for each layer, so that if you have variable 'weights' in both convolution layer 1 and convolution layer 2, there won't be any name clash.

In variable scope, we don't create variable using `tf.Variable`, but instead use `tf.get_variable()`

```
tf.get_variable(<name>, <shape>, <initializer>)
```

If a variable with that name already exists in that variable scope, we use that variable. If a variable with that name doesn't already exists in that variable scope, TensorFlow creates a new variable. This setup makes it really easy to share variables across architecture. This will come in extremely handy when you build complex models and you need to share large sets of variables. Variable scopes help you initialize all of them in one place.

Nodes in the same variable scope will be grouped together, and therefore you don't have to use name scope any more. To declare a variable scope, you do it the same way you do name scope:

```
with tf.variable_scope('conv1') as scope:
```

For example:

```
with tf.variable_scope('conv1') as scope:

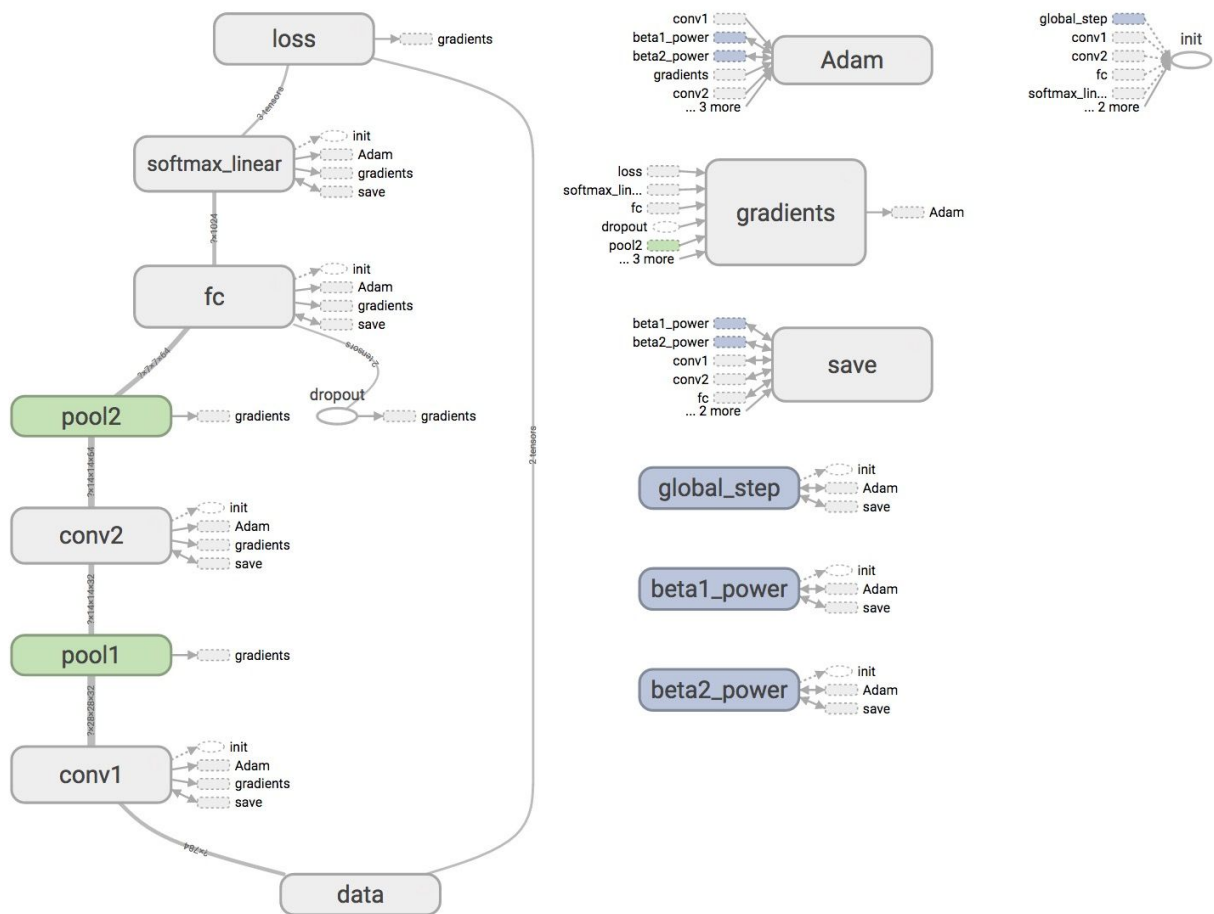
    w = tf.get_variable('weights', [5, 5, 1, 32])
    b = tf.get_variable('biases', [32], initializer=tf.random_normal_initializer())
    conv = tf.nn.conv2d(images, w, strides=[1, 1, 1, 1], padding='SAME')
    conv1 = tf.nn.relu(conv + b, name=scope.name)

with tf.variable_scope('conv2') as scope:

    w = tf.get_variable('weights', [5, 5, 32, 64])
    b = tf.get_variable('biases', [64], initializer=tf.random_normal_initializer())
    conv = tf.nn.conv2d(conv1, w, strides=[1, 1, 1, 1], padding='SAME')
    conv2 = tf.nn.relu(conv + b, name=scope.name)
```

You see that with variable scope, we now have neatly block of code that can be broken into smaller function for reusability.

For more information on variable scope, please refer to [the official documentation](#).



Below is the training result for MNIST with covnets:

Epochs	Accuracy
1	0.9111
2	0.9401
3	0.9494
5	0.9549
10	0.9692
25	0.9736
40	0.9793
50	0.9804