Tensorflow CNN tutorial

2017/03/10
Lenet-5

[LeCun et al., 1998]
Today’s example
The slides are from
2. Lecture 7 & 12 in Stanford CS231n
Convolution Layer

32x32x3 image

32  height
32  width
3   depth
Convolution Layer

32x32x3 image

5x5x3 filter

**Convolve** the filter with the image i.e. “slide over the image spatially, computing dot products”
Convolution Layer

32x32x3 image

5x5x3 filter

Convolve the filter with the image i.e. “slide over the image spatially, computing dot products”

Filters always extend the full depth of the input volume
Convolution Layer

32x32x3 image
5x5x3 filter $w$

1 number:
the result of taking a dot product between the filter and a small 5x5x3 chunk of the image (i.e. $5 \times 5 \times 3 = 75$-dimensional dot product + bias)

$$w^T x + b$$
7x7 input (spatially)
assume 3x3 filter
7x7 input (spatially)
assume 3x3 filter
7x7 input (spatially)
assume 3x3 filter
7x7 input (spatially)
assume 3x3 filter
7x7 input (spatially)
assume 3x3 filter

=> 5x5 output
Convolution Layer

32x32x3 image
5x5x3 filter

convolve (slide) over all spatial locations

activation map
Convolution Layer

32x32x3 image
5x5x3 filter

Convolve (slide) over all spatial locations

Consider a second, green filter

Activation maps
For example, if we had 6 5x5 filters, we’ll get 6 separate activation maps:

We stack these up to get a “new image” of size 28x28x6!
**Preview:** ConvNet is a sequence of Convolution Layers, interspersed with activation functions.

![Diagram of ConvNet layers](image)

- **CONV**
- **ReLU**
- e.g. 6
- 5x5x3 filters
**Preview:** ConvNet is a sequence of Convolutional Layers, interspersed with activation functions.

![Diagram of ConvNet layers](image)

- **First Layer:**
  - Convolution (CONV) with ReLU activation function
  - Example: 6 filters of size $5 \times 5 \times 3$

- **Second Layer:**
  - Convolution (CONV) with ReLU activation function
  - Example: 10 filters of size $5 \times 5 \times 6$

- **Subsequent Layers:**
  - Convolution (CONV) with ReLU activation function
  - Example: 24 filters

The diagram shows the progression of the ConvNet architecture, with each layer reducing the spatial dimensions of the input data while increasing the number of feature maps.
32x32 input convolved repeatedly with 5x5 filters shrinks volumes spatially! (32 -> 28 -> 24 ...). Shrinking too fast is not good, doesn’t work well.
A closer look at spatial dimensions:

7x7 input (spatially)
assume 3x3 filter
A closer look at spatial dimensions:

7x7 input (spatially)
assume 3x3 filter
A closer look at spatial dimensions:

7x7 input (spatially)
assume 3x3 filter
A closer look at spatial dimensions:

7x7 input (spatially) assume 3x3 filter
A closer look at spatial dimensions:

7x7 input (spatially)
assume 3x3 filter

=> 5x5 output
A closer look at spatial dimensions:

7x7 input (spatially) assume 3x3 filter applied with stride 2
A closer look at spatial dimensions:

7x7 input (spatially) assume 3x3 filter applied *with stride 2*
A closer look at spatial dimensions:

7x7 input (spatially) assume 3x3 filter applied with stride 2 => 3x3 output!
A closer look at spatial dimensions:

7x7 input (spatially) assume 3x3 filter applied with stride 3?
A closer look at spatial dimensions:

- 7x7 input (spatially) assume 3x3 filter applied with stride 3?

  - doesn’t fit!
  - cannot apply 3x3 filter on 7x7 input with stride 3.
In practice: Common to zero pad the border

<table>
<thead>
<tr>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

e.g. input 7x7
3x3 filter, applied with **stride 1**
pad with **1 pixel** border => what is the output?

7x7 output!
In practice: Common to zero pad the border

- For example, input 7x7
- 3x3 filter, applied with **stride 1**
- **pad with 1 pixel** border => what is the output?

**7x7 output!**

- In general, common to see CONV layers with stride 1, filters of size FxF, and zero-padding with (F-1)/2. (will preserve size spatially)
- e.g. F = 3 => zero pad with 1
- F = 5 => zero pad with 2
- F = 7 => zero pad with 3
Pooling layer

- makes the representations smaller and more manageable
- operates over each activation map independently:
MAX POOLING

Single depth slice

max pool with 2x2 filters and stride 2
Tensorflow implementation

• Weight Initialization
• Convolution and Pooling
• Convolution layer
• Fully connected layer
• Readout Layer

• Reference and image source: https://www.tensorflow.org/get_started/mnist/pros
  (See section ‘Build a Multilayer Convolutional Network’)

Input (placeholder)

```python
x = tf.placeholder(tf.float32, shape=[None, input_size])
y = tf.placeholder(tf.float32, shape=[None, classes_num])
```

x is placeholder for input image.
y is label with one-hot representation, so second dimension of y is equal to number of classes.

*None* indicates that the first dimension, corresponding to the batch size, which can be any size.
Weight Initialization

```python
def weight_variable(shape):
    initial = tf.truncated_normal(shape, stddev=0.1)
    return tf.Variable(initial)

def bias_variable(shape):
    initial = tf.constant(0.1, shape=shape)
    return tf.Variable(initial)
```

tf.truncated_normal

These variable will be initialized when user run `tf.global_variables_initializer`. Now they are just nodes in a graph without any value.
Convolution and Pooling

```
def conv2d(x, W):
    return tf.nn.conv2d(x, W, strides=[1, 1, 1, 1], padding='SAME')

def max_pool_2x2(x):
    return tf.nn.max_pool(x, ksize=[1, 2, 2, 1],
                          strides=[1, 2, 2, 1], padding='SAME')
```

Strides is 4-d, following NHWC format.  
(Num_samples x Height x Width x Channels)

Recall strides and padding.  
padding = ‘SAME’ means apply padding to keep output size as same as input size.

Conv2d pads with zeros and max_pool pads with –inf.

```
  tf.nn.conv2d
  tf.nn.max_pool
```
Convolution layer

\[
\begin{align*}
\text{w\_conv1} &= \text{weight\_variable}([5, 5, 1, 32]) \\
\text{b\_conv1} &= \text{bias\_variable}([32]) \\
\text{x\_image} &= \text{tf\_reshape}(x, [-1, 28, 28, 1]) \\
\text{h\_conv1} &= \text{tf\_nn\_relu}(\text{conv2d}(\text{x\_image}, \text{w\_conv1}) + \text{b\_conv1}) \\
\text{h\_pool1} &= \text{max\_pool\_2x2}(\text{h\_conv1}) \\
\text{w\_conv2} &= \text{weight\_variable}([5, 5, 32, 64]) \\
\text{b\_conv2} &= \text{bias\_variable}([64]) \\
\text{h\_conv2} &= \text{tf\_nn\_relu}(\text{conv2d}(\text{h\_pool1}, \text{w\_conv2}) + \text{b\_conv2}) \\
\text{h\_pool2} &= \text{max\_pool\_2x2}(\text{h\_conv2})
\end{align*}
\]
Convolution layer

See how the code creates a model by wrapping layers.
Be care of shape of each layer.
-1 means match the size of that dimension is computed so that the total size remains constant.
Reshape

For example:
tensor ‘t’ is [[1, 2], [3, 4], [5, 6], [7, 8]], so t has shape [4, 2]

(1) reshape(t, [2,4]) ➔ [[1, 2, 3, 4], [5, 6, 7, 8]]
(2) reshape(t, [-1, 4]) ➔ [[1, 2, 3, 4], [5, 6, 7, 8]]

-1 would be computed and becomes ‘2’
Fully connected layer

W_fc1 = weight_variable([7 * 7 * 64, 1024])
b_fc1 = bias_variable([1024])

h_pool2_flat = tf.reshape(h_pool2, [-1, 7*7*64])
h_fc1 = tf.nn.relu(tf.matmul(h_pool2_flat, W_fc1) + b_fc1)

Flatten all the maps and connect them with fully connected layer. Again, be care of shape.
Use a layer to match output size.
Done!
Training and Evaluation (optional)

cross_entropy = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(y_conv, y_))
train_step = tf.train.AdamOptimizer(1e-4).minimize(cross_entropy)
correct_prediction = tf.equal(tf.argmax(y_conv,1), tf.argmax(y_,1))
accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))
sess.run(tf.global_variables_initializer())
for i in range(20000):
    batch = mnist.train.next_batch(50)
    if i%100 == 0:
        train_accuracy = accuracy.eval(feed_dict={
            x:batch[0], y_: batch[1], keep_prob: 1.0})
        print("step %d, training accuracy %g"%(i, train_accuracy))
        train_step.run(feed_dict={x: batch[0], y_: batch[1], keep_prob: 0.5})

print("test accuracy %g"%accuracy.eval(feed_dict={
    x: mnist.test.images, y_: mnist.test.labels, keep_prob: 1.0})))
Recommendation

• Search for each function, and you’ll what’s everything going on.