Report of Tensorflow

Maurizio Franchi

February 2016

Introduction

In this report I discuss:

- the methodology used in the layers: convolution, max pooling, ReLU, dropout and softmax;
- how I adapt the code of the complete deep architecture as I remove one layer at the time and keep the other
- how I compare the performance of the deep convolution networks when removing layers;

Methodology

In this report I used five methodologies to create the deep network: convolution, max pooling, ReLU, dropout and softmax. Below I discuss each methodology separately.

Convolutional layer

The convolutional layer is the first layer in the deep convolution network. In this layer the parameters consist of a set of learnable filters (or kernels), which have a small receptive field, but extend through the full depth of the input volume. For the next step, each filter:

- is convolved across the width and height of the input volume;
- compute the dot product between the entries of the filter and the input, producing a 2-dimensional activation map of that filter.

From this it follows that the network learns filters that activate themselves when they see some specific type of feature at some spatial position in the input.
In this case the full output of the convolution layer is the filling of the activation maps for all filters along the depth dimension. Every entry in the output volume can be interpreted as an output of a neuron that looks at a small region in the input and shares parameters with neurons in the same activation map.

The convolution layer has three main characteristic:

- local connectivity;
- spatial arrangement;
- parameter sharing.

Local connectivity

If there are high-dimensional inputs such as images, it is impossible to connect neurons to all neurons in the volume because such a network architecture does not take the spatial structure of the data into account. For this reason the convolutional networks exploit spatially local correlation by enforcing a local connectivity pattern between neurons of adjacent layers and then each neuron is connected to only a small region of the input volume. The connections are local in space (along width and height) but always extend along the entire depth of the imput volume. Such an architecture ensures that the learnt filters produce the strongest response to a spatially local input pattern.

Spatial arrangement

In convolutional layer there are three hyperparameters that control the size of the output of this layer:

- `depth` of the output volume controls the number of neurons in the layer that connect to the same region of the input volume. All of these neurons will learn to activate for different features in the input.
- `stride` controls how depth columns around the spatial dimensions (width and height) are allocated. When the stride is 1 then a new depth columns is allocated to spatial positions only 1 spatial unit apart. In this case there are heavily overlapping receptive fields between the columns and this leads to large output volume. Conversely, if we use higher strides, the receptive fields will overlap less and then the output volume will have smaller dimensions spatially;
- `zero-padding` because sometimes it is convenient to pad the input with zeros on the border of the input volume. Zero padding allows to control the spatial size of the input volumes. In particular, sometimes it is desirable to exactly preserve the spatial size of the input volume.

The spatial size of the output volume can be computed as a function of the input volume size W, the receptive field size of the convolutional layer neurons F, the stride with which they are applied S and the amount of zero padding P used on the the bonder. The formula for calculating how many neurons fit in a given volume is given by $(W - F + 2P)/S + 1$. If this number is not an integer, then the strides are set incorrectly and the neurons cannot be tiled to fit across the input volume in a symmetric way. In general, setting zero paddng to be $P = (F - 1)/2$ when the stride is $S = 1$ ensures that the input volume and output volume will have the same size spatially.

Parameter sharing

Parameter sharing scheme is used in convolutional layer to control the number of free parameters. This layer relies on one reasonable assumption: if one patch feature is useful to compute at some spatial position then it is should also be useful to compute at a different position.
Since all neurons in a single depth slice are sharing the same parameter, then the next step in each depth slice of the convolutional layer can be computed as a convolution of the neuron's weights with the input volume. As a consequence, it is common to refer to the sets of weights as a filter, which is convolved with the input. The result of this convolution is an activation map and the set of activation maps for each different filter are stacked together along the depth dimension to produce the output volume. Parameter Sharing contributes to the transaction invariance of the convolutional neural network architecture.
It is important to notice that sometimes the parameter sharing assumption may not make sense. An example is the case when the input images to a convolutional neural network have some specific centered structure in which we expect completely different features to be learned on different spatial locations. An example of this is when the input are faces that have been centered in the image: we might expect different eye-specific or hair-specific features to be learned in different parts of the image. In that case it is common to relax the parameter sharing scheme and instead simply call the layer a locally connected layer.

Max pooling

Max pooling is a form of non linear down-sampling. This layer divides the input image into a set of non overlapping rectangles and outputs the maximum for each sub-region. The idea is that once a feature has been found, it is not important its exact location but its rough location in relation to other features.
The max pooling layer has two important tasks:

- progressively reduce the spatial size of the representation to reduce the amount of parameters and computation in the network
- control the overfitting

This two important tasks are correlated, in fact the second task is derived by the first task.
The max pooling layer operates independently on every depth slice of the input and resized it spatially.
A example of max pooling layer that is also the most common form is a max pooling layer with filters of size 2x2 applied with a stride of 2 downsamples at every depth slice in the input by 2 along both width and height, descarding 75% of the activations.

ReLU

ReLU (Reactified Linear Units) layer is a layer of neurons that applies the non saturation activation function $f(x) = max(0, x)$. It increases the nonlinear properties of the decision function and of the overall network without affecting the receptive fields of the convolution layer.
Compared to other functions the usage of ReLU is preferable because it results in the neural network training several faster without making a significant difference to generalisation accuracy.

Dropout

The dropout method is introduced to avoid overfitting. In this method at each step, individual nodes are in random way either "dropped out" of the network with probability $1 - p$ with them incoming and outcoming edges, or saved with probability p. Therefore after this we have a reduced network. Only the reduced network is trained on the data in that stage.

Softmax

Softmax layer has three important tasks:

- looks at a feature configuration coming from the layers below and gives probabilities for it being each digit
- evidences that a feature configuration is a particular class `i`:

$$evidence_i = \sum_j W_{ij} x_y + b_i \forall_i$$

- shape the evidence as a probability distribution over `i`:

$$softmax_i = \frac{exp(evidence_i)}{\sum_j exp(evidence_j)} \forall_i$$

Deep convolution network code

The deep convolution network code is made from five steps:

- preparation;
- weights initialization;
- convolution and pooling;
- modelling;
- optimization and evaluation.

Preparation

This step is divided into three important parts:

- `Imports` in which I import the file `input_data` with the following code:

```
import input_data
import tensorflow as tf
```

- `Load data` in which load the `MNIST dataset` with the following code:

```
mnist = input_data.read_data_sets('MNIST_data', one_hot=True)
```

- `Placeholders and data reshaping` in which:
  - set the values we will input during the computation with the following code:

```
x = tf.placeholder("float", shape=[None, 784])
y = tf.placeholder("float", shape=[None, 10])
```

  - reshape vectors of size 784 to squares of size 28x28 with the following code:

```
x_image = tf.reshape(x, [-1,28,28,1])
```

Weights initialization

In this step we define functions to initialize variables of the model two in particular:

- weight variable through the following code:

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

- bias variable through the following code:

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

Convolution and pooling

In this step we define:

- the function for the convolution through the code:

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

- the function for the max pooling through the code:

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

Model

In this step we define four layers that are:

- the first layer in which there is the convolution layer with max pooling and the code is:

```
W_conv1 = weight_variable([5,5,1,32])
b_conv1 = bias_variable([32])
h_conv1 = tf.nn.relu(conv2d(x_image, W_conv1) + b_conv1)
h_pool1 = max_pool_2x2(h_conv1)
```

- the second layer in which there is the convolution layer with max pooling and the code is:

```
W_conv2 = weight_variable([5,5,32,64])
b_conv2 = bias_variable([64])
h_conv2 = tf.nn.relu(conv2d(h_pool1, W_conv2) + b_conv2)
h_pool2 = max_pool_2x2(h_conv2)
```

- the third layer in which there are:
  - ReLU layer that composed by two parts:
    - fully connected layer through the code:

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

    - the flot of the output of the previous layer through the code:

      ```
      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)
      ```

  - the dropout part through the code:

    ```
    keep_prob = tf.placeholder("float")
    h_fc1_drop = tf.nn.dropout(h_fc1, keep_prob)
    ```

- the fourth layer in which there is the fully connected layer through the code:

```
W_fc2 = weight_variable([1024, 10])
b_fc2 = bias_variable([10])
y_hat = tf.nn.softmax(tf.matmul(h_fc1_drop, W_fc2) + b_fc2)
```

Optimization and evaluation

In this step we:

- define the cost with the cross-entropy: $H_y(\widehat{y}) = -\sum ylog(\widehat{y})$ through the code:

```
cross_entropy = -tf.reduce_sum(y*tf.log(y_hat))
```

- define the traning algorithm minimizing the cross-entropy through the code:

```
train_step = tf.train.AdamOptimizer(1e-4).minimize(cross_entropy)
```

- start a new session through the code:

```
sess = tf.InteractiveSession()
```

- initialize the variables with the code:

```
sess.run(tf.initialize_all_variables())
```

- define accuracy before training for monitoring with the code:

```
for n in range(20000):
    batch = mnist.train.next_batch(50)
    if n % 100 == 0:
        train_accuracy = accuracy.eval(feed_dict={x:batch[0], y:batch[1],
        keep_prob: 1.0})
        print "step %d, training accurancy %g" % (n, train_accuracy)
    sess.run(train_step, feed_dict={x:batch[0], y:batch[1], keep_prob: 0.5})
```

- evaluate the prediction with the code:

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

Changes of Deep convolution network code

In this section I explain how I change the deep convolution network code seen in the previous chapter to compare the performance of this network when removing layers.

Remove the first layer

If I remove the first layer I must modify:

- the second layer to enable the latter to read the input, therefore the code of the second layer becomes:

```
W_conv2 = weight_variable([5,5,1,64])
b_conv2 = bias_variable([64])
h_conv2 = tf.nn.relu(conv2d(max_pool_2x2(x_image), W_conv2) + b_conv2)
h_pool2 = max_pool_2x2(h_conv2)
```

- the third layer because now it takes as a input a image 14x14 and not 7x7, therefore the code of the third layer becomes:

```
W_fc1 = weight_variable([14*14*64, 1024])
b_fc1 = bias_variable([1024])
h_pool2_flat = tf.reshape(h_pool2, [-1,14*14*64])
h_fc1 = tf.nn.relu(tf.matmul(h_pool2_flat, W_fc1) + b_fc1)
keep_prob = tf.placeholder("float")
h_fc1_drop = tf.nn.dropout(h_fc1, keep_prob)
```

Remove the second layer

If I remove the second layer then I must change the weight variable of the third layer and so the code of the third layer is:

```
W_fc1 = weight_variable([14*14*32, 1024])
b_fc1 = bias_variable([1024])
h_pool2_flat = tf.reshape(h_pool1, [-1,14*14*32])
h_fc1 = tf.nn.relu(tf.matmul(h_pool2_flat, W_fc1) + b_fc1)
keep_prob = tf.placeholder("float")
h_fc1_drop = tf.nn.dropout(h_fc1, keep_prob)
```

Remove the third layer

If I remove the third layer then the fourth layer must be the tasks that was the third layer, therefore the fourth layer becomes:

```
W_fc1 = weight_variable([7*7*64, 1024])
b_fc1 = bias_variable([1024])
W_fc2 = weight_variable([1024, 10])
b_fc2 = bias_variable([10])
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)
keep_prob = tf.placeholder("float")
h_fc1_drop = tf.nn.dropout(h_fc1, keep_prob)

y_hat = tf.nn.softmax(tf.matmul(h_fc1_drop, W_fc2) + b_fc2)
```

Conclusion

After I run the four scripts:

- `model.py` in which there is the full model;
- `model_no_1` in with there is the model without the first layer;
- `model_no_2` in with there is the model without the second layer;
- `model_no_3` in with there is the model without the third layer;

for each script I obtain the test accuracy reported in the following table.

test accuracy of the model	0.9920
test accuracy of the model without first layer	0.9908
test accuracy of the model without second layer	0.9901
test accuracy of the model without third layer	0.9919

From this test accuracy I can conclude that if I remove one layer from the original model, then the test accuracy that is similar is the one of the model without the third layer, in fact the test accuracy of this model is practically the same of the original model.