Rui Shu

About Blog GitHub Publications Doodles

27 DEC 2016

TENSORFLOW GUIDE: BATCH NORMALIZATION

Update [11-21-2017]: Please see this code snippet for my current preferred implementation.

I recently made the switch to TensorFlow and am very happy with how easy it was to get things done using this awesome library. Tensorflow has come a long way since I first experimented with it in 2015, and I am happy to be back.

Since I am getting myself re-acquainted with TensorFlow, I decided that I should write a post about how to do batch normalization in TensorFlow. It's kind of weird that batch normalization still presents such a challenge for new TensorFlow users, especially since TensorFlow comes with invaluable functions like `tf.nn.moments` , `tf.nn.batch_normalization` , and even `tf.contrib.layers.batch_norm` . One would think that using batch normalization in TensorFlow will be a cinch. But alas, confusion still crops up from time to time, and the devil really lies in the details.

Batch Normalization The Easy Way

Perhaps the easiest way to use batch normalization would be to simply use the `tf.contrib.layers.batch_norm` layer. So let's give that a go! Let's get some imports and data loading out of the way first.

```
import numpy as np
import tensorflow as tf
from tensorflow.examples.tutorials.mnist import input_data
from utils import show_graph
mnist = input_data.read_data_sets("MNIST_data/", one_hot=True)
```

Next, we define our typical `fully-connected + batch normalization + nonlinearity` set-up

```
def dense(x, size, scope):
    return tf.contrib.layers.fully_connected(x, size,
                                             activation_fn=None,
                                             scope=scope)

def dense_batch_relu(x, phase, scope):
    with tf.variable_scope(scope):
        h1 = tf.contrib.layers.fully_connected(x, 100,
                                               activation_fn=None,
                                               scope='dense')
        h2 = tf.contrib.layers.batch_norm(h1,
                                          center=True, scale=True,
                                          is_training=phase,
                                          scope='bn')
        return tf.nn.relu(h2, 'relu')
```

One thing that might stand out is the `phase` term. We are going to use as a placeholder for a boolean which we will insert into `feed_dict` . It will serve as a binary indicator for whether we are in training `phase=True` or testing `phase=False` mode. Recall that batch normalization has distinct behaviors during training verus test time:

Training

1. Normalize layer activations according to mini-batch statistics.
2. *During the training step*, update population statistics approximation via moving average of mini-batch statistics.

Testing

1. Normalize layer activations according to estimated population statistics.
2. Do *not* update population statistics according to mini-batch statistcs from test data.

Now we can define our very simple neural network for MNIST classification.

```
tf.reset_default_graph()
x = tf.placeholder('float32', (None, 784), name='x')
y = tf.placeholder('float32', (None, 10), name='y')
phase = tf.placeholder(tf.bool, name='phase')

h1 = dense_batch_relu(x, phase,'layer1')
h2 = dense_batch_relu(h1, phase, 'layer2')
logits = dense(h2, 10, 'logits')

with tf.name_scope('accuracy'):
    accuracy = tf.reduce_mean(tf.cast(
            tf.equal(tf.argmax(y, 1), tf.argmax(logits, 1)),
            'float32'))

with tf.name_scope('loss'):
    loss = tf.reduce_mean(
        tf.nn.softmax_cross_entropy_with_logits(logits, y))
```

Now that we have defined our data and computational graph (a.k.a. model), we can train the model! Here is where we need to notice a very important note in the `tf.contrib.layers.batch_norm` documentation:

> Note: When is_training is True the moving_mean and moving_variance need to be updated, by default the update_ops are placed in tf.GraphKeys.UPDATE_OPS so they need to be added as a dependency to the train_op, example:
>
> update_ops = tf.get_collection(tf.GraphKeys.UPDATE_OPS) if update_ops: updates = tf.group(*update_ops) total_loss = control_flow_ops.with_dependencies([updates], total_loss)

If you are comfortable with TensorFlow's underlying graph/ops mechanism, the note is fairly straight-forward. If not, here's a simple way to think of it: when you execute an operation (such as `train_step`), only the subgraph components relevant to `train_step` will be executed. Unfortunately, the `update_moving_averages` operation is not a parent of `train_step` in the computational graph, so we will never update the moving averages! To get around this, we have to explicitly tell the graph:

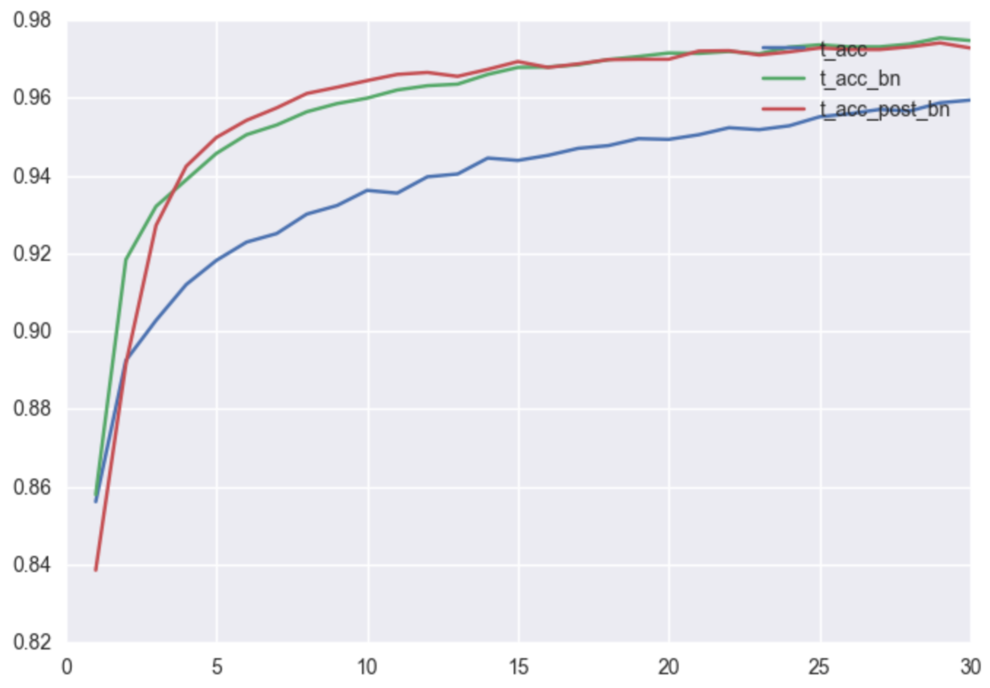
> Hey graph, update the moving averages before you finish the training step!

Unfortunately, the instructions in the documentation are a little out of date. Furthermore, if you think about it a little more, you may conclude that attaching the `update` ops to `total_loss` may not be desirable if you wish to compute the `total_loss` of the test set during test time. Personally, I think it makes more sense to attach the `update` ops to the `train_step` itself. So I modified the code a little and created the following training function

```
def train():
    update_ops = tf.get_collection(tf.GraphKeys.UPDATE_OPS)
    with tf.control_dependencies(update_ops):
        # Ensures that we execute the update_ops before performing the train_step
        train_step = tf.train.GradientDescentOptimizer(0.01).minimize(loss)
    sess = tf.Session()
    sess.run(tf.global_variables_initializer())

    history = []
    iterep = 500
    for i in range(iterep * 30):
        x_train, y_train = mnist.train.next_batch(100)
        sess.run(train_step,
                 feed_dict={'x:0': x_train,
                            'y:0': y_train,
                            'phase:0': 1})
        if (i + 1) %  iterep == 0:
            epoch = (i + 1)/iterep
            tr = sess.run([loss, accuracy],
                          feed_dict={'x:0': mnist.train.images,
                                     'y:0': mnist.train.labels,
                                     'phase:0': 1})
            t = sess.run([loss, accuracy],
                         feed_dict={'x:0': mnist.test.images,
                                    'y:0': mnist.test.labels,
                                    'phase:0': 0})
            history += [[epoch] + tr + t]
            print history[-1]
    return history
```

And we’re done! We can now train our model and see what happens. Below, I provide a comparison of the model without batch normalization, the model with pre-activation batch normalization, and the model with post-activation batch normalization.



As you can see, batch normalization really does help with training (not always, but it certainly did in this simple example).

Additional Remarks

You have the choice of applying batch normalization either before or after the non-linearity, depending on your definition of the "activation distribution of interest" that you wish to normalize. It will probably end up being a hyperparameter that you'll just have to tinker with.

There is also the question of what it means to *share* the same batch normalization layer across multiple models. I think this question should be treated with some care, and it really depends on what you think you'll gain out of sharing the batch normalization layer. In particular, we should be careful about sharing the mini-batch statistics across multiple data streams if we expect the data streams to have distinct distributions, as is the case when using batch normalization in recurrent neural networks. As of the moment, the `tf.contrib.layers.batch_norm` function does not allow this level of control.

But I do!… Kind of. It's a fairly short piece of code, so it should be easy to modify it to fit your own purposes. **runs away**

CODE ON GITHUB

JUPYTER NOTEBOOK

End of post

Tweet Like 7

10 Comments **ruishu.io** 1 **Login**

Recommend 5 Share Sort by Best

Join the discussion…

LOG IN WITH

OR SIGN UP WITH DISQUS ?

Name

Mohammad Reza Loghmani • 19 days ago

Thank you for the post!
I was trying to use the lower level implementation of the batch normalization layer, tf.nn.batch_normalization, but I am confused about how to distinguish between training and evaluation phase. In practice, what is the corresponding way of setting is_training = True in tf.layers.batch_normalization?

• Reply • Share ›

rohola zandie Mohammad Reza Loghmani • 19 days ago

I think you should set is_training=True as feed_dict when you are reading from training and otherwise(during test or evaluation) False. btw, can you explain more coz I think maybe you have another problem.

1 • Reply • Share ›

Sriram Vasu • 6 months ago

Does the boolean have to be a tensor? I am facing issues when I just use an if-else statement.

• Reply • Share ›

tal • 8 months ago

Hi,
This is a great post.

Can you tell me if it's possible to assign a value to batch normalization layer in TF? I'd like to initialize my model by a pre-trained model I have in caffe.

• Reply • Share ›

zoltan • a year ago

Should use reuse=None in training phase and reuse=True in testing phase?

About Blog GitHub Publications Doodles

built with Jekyll using Scribble theme