Project: tensorflow

2016-02-16T13:18:56Z cesarsalgado 127comments

Easy to use batch norm layer.

Web url: https://github.com/tensorflow/tensorflow/issues/1122

API url: https://api.github.com/repos/tensorflow/tensorflow/issues/1122

Many non-experts are using the following code http://stackoverflow.com/questions/33949786/how-could-i-use-batch-normalization-in-tensorflow?answertab=votes#tab-top.

It would be nice to have an official batch norm layer given its importance in training DNNs.

-------------------------------------------------------------------------

2016-02-16T14:19:56Z vincentvanhoucke

I'm working on some parts of that.

-------------------------------------------------------------------------

2016-06-09T16:12:25Z vincentvanhoucke

There is now a `batch\_norm` layer:

https://github.com/tensorflow/tensorflow/blob/b826b79718e3e93148c3545e7aa3f90891744cc0/tensorflow/contrib/layers/python/layers/layers.py#L100

-------------------------------------------------------------------------

2016-06-18T07:55:49Z Mahdizade

I think some thing wrong with this layer. in training every thing is OK and loss decrease very good. but in testing I get zero accuracy.

By the way in testing when I use is\_training=False, I get zero acc.

I know batch normalization behave different in train and test phase, as describe in [How does batch normalization behave differently at training time and test time? - Quora](https://www.quora.com/How-does-batch-normalization-behave-differently-at-training-time-and-test-time). I think this implementation is unclear

-------------------------------------------------------------------------

2016-06-20T21:43:28Z pawni

Same here, I have experienced some unexpected behavior with is\_training=False. What is the correct way to change this flag? I am currently using a `tf.cond` because it does not take `tf.placeholders` by itself.

-------------------------------------------------------------------------

2016-06-20T21:51:13Z ppwwyyxx

@pawni You have to use a Python boolean for `is\_training`. It cannot be a `tf.cond`.

-------------------------------------------------------------------------

2016-06-20T21:55:28Z pawni

@ppwwyyxx well I am doing `tf.cond(placeholder, batch\_norm(.., is\_training = True), batch\_norm(.., is\_training = False))` or is one just supposed to do a `batch\_norm(.., is\_training=variable)` and change that outside of the graph when needed?

-------------------------------------------------------------------------

2016-06-20T22:12:49Z ppwwyyxx

Oh I thought you were doing `batch\_norm(.., is\_training=tf.cond(placeholder))`, which is incorrect.

Your current way might have problems as well. You'll need to double check that the two `batch\_norm` op you created share the same scope, otherwise they won't share the underlying mean/variance statistics.

To do this the `reuse` argument might help, but I'm not sure because I use my own version of bn layer.

-------------------------------------------------------------------------

2016-06-20T22:21:03Z pawni

I am using the same scope and `reuse=True`. It seems to work sometimes but I am not too sure. It would be great if the layer could be added to the documentation with a short explanation how to best handle the change from training to test.

-------------------------------------------------------------------------

2016-06-21T01:29:05Z vincentvanhoucke

@sguada FYI

-------------------------------------------------------------------------

2016-06-21T04:05:45Z sguada

Currently batch\_norm requires a python boolean, but we are working in adding the option of passing a Tensor.

-------------------------------------------------------------------------

2016-06-21T04:07:39Z sguada

@pawni If you don't want to worry about about updating moving\_mean and moving\_variance set updates\_collections=None to make sure they are updated in place, otherwise you need to make sure the update\_ops added to tf.GraphKeys.UPDATE\_OPS are run during training.

-------------------------------------------------------------------------

2016-06-21T04:30:18Z Mahdizade

I think tensorflow need 2 hyper methods that change the model state, something like torch. [change model state](https://github.com/torch/nn/blob/master/doc/module.md#training). I think it is very straightforward.

-------------------------------------------------------------------------

2016-07-11T20:30:40Z brando90

is there a small script with a very simple NN that shows what is the proper way of using this "official" BN layer? I'd really appreciate it.

-------------------------------------------------------------------------

2016-07-11T20:39:28Z brando90

sorry if this is a little repetitive, but it seems the API talks about BN in a different interface: https://www.tensorflow.org/versions/r0.9/api\_docs/python/nn.html#batch\_normalization

is that not the official way to use BN? I am confused on how to use it and the SO seems to be outdated and then there is a layer in a different link from the API, just how exactly does one do this? I am unclear if to go to SO or ask here.

-------------------------------------------------------------------------

2016-07-12T00:51:27Z brando90

sorry for the spamming, but what is wrong with just using something like this:

```

def standard\_batch\_norm(l, x, n\_out, phase\_train, scope='BN'):

"""

Batch normalization on feedforward maps.

Args:

x: Vector

n\_out: integer, depth of input maps

phase\_train: boolean tf.Varialbe, true indicates training phase

scope: string, variable scope

Return:

normed: batch-normalized maps

"""

with tf.variable\_scope(scope+l):

#beta = tf.Variable(tf.constant(0.0, shape=[n\_out], dtype=tf.float64 ), name='beta', trainable=True, dtype=tf.float64 )

#gamma = tf.Variable(tf.constant(1.0, shape=[n\_out],dtype=tf.float64 ), name='gamma', trainable=True, dtype=tf.float64 )

init\_beta = tf.constant(0.0, shape=[n\_out], dtype=tf.float64)

init\_gamma = tf.constant(1.0, shape=[n\_out],dtype=tf.float64)

beta = tf.get\_variable(name='beta'+l, dtype=tf.float64, initializer=init\_beta, regularizer=None, trainable=True)

gamma = tf.get\_variable(name='gamma'+l, dtype=tf.float64, initializer=init\_gamma, regularizer=None, trainable=True)

batch\_mean, batch\_var = tf.nn.moments(x, [0], name='moments')

ema = tf.train.ExponentialMovingAverage(decay=0.5)

def mean\_var\_with\_update():

ema\_apply\_op = ema.apply([batch\_mean, batch\_var])

with tf.control\_dependencies([ema\_apply\_op]):

return tf.identity(batch\_mean), tf.identity(batch\_var)

mean, var = tf.cond(phase\_train, mean\_var\_with\_update, lambda: (ema.average(batch\_mean), ema.average(batch\_var)))

normed = tf.nn.batch\_normalization(x, mean, var, beta, gamma, 1e-3)

return normed

```

then its simple to tell tensorflow which one to use with a feed dictionary as in:

```

feed\_dict = {x: Xminibatch, y\_: Yminibatch, phase\_train: True}

sess.run(fetches=[merged,train\_step], feed\_dict=feed\_dict)

```

since its unclear if the implementation will change, I wanted to give a suggestion (note its easy to extend to convolutions and stuff I just didn't paste that code).

-------------------------------------------------------------------------

2016-07-12T01:33:15Z brando90

@pawni @ppwwyyxx did you guys decide if you had to use reuse to true to solve the scoping issue?

-------------------------------------------------------------------------

2016-07-12T02:01:40Z pawni

@brando90 currently I am doing something like:

```

def BatchNorm(inputT, is\_training=True, scope=None):

return tf.cond(isTraining,

lambda: batch\_norm(inputT, is\_training=True,

center=False, updates\_collections=None, scope=scope),

lambda: batch\_norm(inputT, is\_training=False,

updates\_collections=None, center=False, scope=scope, reuse = True))

```

However, I think that #3265 would basically want to implement it like this. A reference could be the dropout implementation here: https://github.com/tensorflow/tensorflow/blob/master/tensorflow/contrib/layers/python/layers/layers.py#L433-L435

-------------------------------------------------------------------------

2016-07-12T22:31:13Z sguada

When the updates\_collections=None then the updates happens in-place and it is easier to use a tf.cond() to allow is\_training being a Tensor a bit more complicated is when the updates are delayed and the the update\_ops are run later.

I will try to get the first part in soon.

-------------------------------------------------------------------------

2016-07-14T01:49:35Z nmhkahn

@brando90 @pawni he's code works good, but have to change like below

``` python

def BatchNorm(inputT, is\_training=True, scope=None):

# Note: is\_training is tf.placeholder(tf.bool) type

return tf.cond(is\_training,

lambda: batch\_norm(inputT, is\_training=True,

center=False, updates\_collections=None, scope=scope),

lambda: batch\_norm(inputT, is\_training=False,

updates\_collections=None, center=False, scope=scope, reuse = True))

```

And when run in training or test time,

``` python

# when training

sess.run([opt, loss], feed\_dict={x: bx, y: by, is\_training=True})

# when test

sess.run([opt, loss], feed\_dict={x: bx, y: by, is\_training=False})

```

This code works, but like [#3265](https://github.com/tensorflow/tensorflow/issues/3265) says it will be great if `tf.contrib.layers.batch\_norm` get `is\_training` variable as a `tf.plcaeholer`.

-------------------------------------------------------------------------

2016-07-21T22:55:26Z diegoAtAlpine

@nmhkahn @pawni thanks for the code snippets. They were very useful in adding batch normalization to my convolution network. Training seems to work very well. Testing is not. In some versions of the code training accuracies are much higher than testing accuracies, which probably mean I am not sharing batch normalization parameters. In other versions of the code I get "ValueError: Variable conv1/beta already exists, disallowed. Did you mean to set reuse=True in VarScope?" which seem to indicate that I am trying to relearn the parameter... when I was trying to reuse.

Can someone provide an example of how to call the "def BatchNorm" function during training and testing so that variable sharing happen correctly.

Thanks for any help.

UPDATE July 25, 2016:

@nmhkahn @pawni thanks for your comments. After taking a closer look at the code in contrib I realized what my problem was. During training and testing we are either updating or reusing four variables (beta, gamma, moving\_mean and moving\_variance). To make those unique I had to set a scope per layer. I did it like this:

conv1 = tf.nn.relu(batch\_norm\_layer(conv2d\_stride2\_valid(data, W\_conv1) + b\_conv1, train\_phase, scope="conv1"))

where batch\_norm\_layer is similar to the examples from @nmhkahn @pawni, conv2d\_stride2\_valid is just a def to define a convolutional layer, and W\_conv1 and b\_conv1 are variables holding the weights and biases. I could probably remove the bias term because we are using batch normalization.

The net is working well now. I noticed after plotting accuracies in training and test mode that the testing accuracies start climbing after the training accuracies. In retrospect it make sense since we are collecting dataset statistics for testing. But it appeared as if I was doing something wrong during my initial tests. Thanks for your comments and making batch normalization available to the community.

-------------------------------------------------------------------------

2016-07-22T20:00:07Z brando90

@nmhkahn how is it different from pawni's suggestion?

-------------------------------------------------------------------------

2016-07-22T20:22:54Z pawni

@brando90 I had a small error in my version which was fixed by nmhkahn (changing `isTraining` to `is\_training`)

@diegoAtAlpine I found the same problems - not sure why this is the case though. However, the ValueError should be resolved by the code snippet. Not sure what you want to see how to call it as nmhkahn's examples seems to do the job?

-------------------------------------------------------------------------

2016-07-22T20:55:01Z brando90

@nmhkahn @pawni @ when you do:

`sess.run([opt, loss], feed\_dict={x: bx, y: by, is\_training=True})`

doesn't that mean that your using `is\_training` as a placeholder? People have commented that they want `is\_training` to be a placer holder but thats what I had for my version of it:

```

def batch\_norm\_layer(x,train\_phase,scope\_bn):

bn\_train = batch\_norm(x, decay=0.999, center=True, scale=True,

is\_training=True,

reuse=None, # is this right?

trainable=True,

scope=scope\_bn)

bn\_inference = batch\_norm(x, decay=0.999, center=True, scale=True,

is\_training=False,

reuse=True, # is this right?

trainable=True,

scope=scope\_bn)

z = tf.cond(train\_phase, lambda: bn\_train, lambda: bn\_inference)

return z

```

is that not correct?

-------------------------------------------------------------------------

2016-07-22T20:59:49Z sguada

I have already extended tf.contrib.layers.batch\_norm to allow passing a Tensor or a Placeholder for is\_training. It will be merged in TF contrib soon.

Now available in

https://github.com/tensorflow/tensorflow/commit/9da5fc8e6425cabd61fc36f0dcc1823a093d5c1d#diff-94bbcef0ec8a5cdef55f705e99c2b2ed

-------------------------------------------------------------------------

2016-07-22T21:51:10Z brando90

is it just me or does adding this BN layer noticeably slows down training of a single epoch?

-------------------------------------------------------------------------

2016-07-22T21:56:55Z pawni

@brando90 It slows down training for me as well but I think that this is expected as it needs to calculate some statistics. And your version looks good to me.

-------------------------------------------------------------------------

2016-07-22T22:17:59Z omoindrot

BatchNorm is currently very slow (because of all the statistics computed), but they are working on adding a cudnn batchnorm op as said [here](https://github.com/tensorflow/tensorflow/pull/1759#issuecomment-228856467).

-------------------------------------------------------------------------

2016-07-25T20:56:03Z brando90

@nmhkahn quick question. When you wrote (for testing):

`sess.run([opt, loss], feed\_dict={x: bx, y: by, is\_training=False})`

in theory, can bx and by be any data set? i.e. it can still be the \*\*training\*\* set even though we are not training? (i.e. just to track the train error)

-------------------------------------------------------------------------

2016-07-26T04:23:08Z nmhkahn

@brando90 you're right.

-------------------------------------------------------------------------

2016-07-26T23:08:51Z papadopav

I am also confused regarding is\_training and reuse flags. I have created a program following the CIFAR example, where my code is structured as in CIFAR:

- Inference

- Loss

- Train

And I am running it in a multi-gpu fashion (for training).

So I have one script for training (similar to cifar10\_multigpu.py) and one for testing (similar to cifar10\_eval.py).

So

```

for ii in xrange(2): # Num of GPU

with tf.device('/gpu:%d' % ii):

with tf.name\_scope('device\_%d' % ii) as scope:

data\_batch, label\_batch = factory.GetShuffleBatch(batch\_size)

unnormalized\_logits = factory.MyModel(dataBatch=data\_batch, numClasses=numClasses,

isTraining=True)

More stuff happening

tf.get\_variable\_scope().reuse\_variables()

```

The inference happens with the function MyModel. (below is an example of the function, in reality i use more layers and neurons).

```

def MyModel(data\_batch, num\_classes, feature\_dim):

# Hidden Layer 1

with tf.variable\_scope('hidden1') as scope:

weights = variable\_on\_cpu('weights',[feature\_dim, 256], tf.truncated\_normal\_initializer(stddev=0.04))

biases = variable\_on\_cpu('biases', [256], tf.constant\_initializer(0.001))

hidden1 = tf.nn.relu(tf.matmul(data\_batch, weights) + biases, name=scope.name)

# Hidden Layer 2

with tf.variable\_scope('hidden2') as scope:

weights = variable\_on\_cpu('weights',[256, 256], tf.truncated\_normal\_initializer(stddev=0.04))

biases = variable\_on\_cpu('biases', [256], tf.constant\_initializer(0.001))

hidden2 = tf.nn.relu(tf.matmul(hidden1, weights) + biases, name=scope.name)

# output, unnormalized softmax

with tf.variable\_scope('softmax\_unnorm') as scope:

weights = variable\_on\_cpu('weights', [256, num\_classes], tf.truncated\_normal\_initializer(stddev=1/num\_classes))

biases = variable\_on\_cpu('biases', [num\_classes], tf.constant\_initializer(0.0))

softmax\_un = tf.add(tf.matmul(hidden2, weights), biases, name=scope.name)

return softmax\_un

```

I want to perform batch nomalization. So when I did:

```

def MyModel(data\_batch, num\_classes, feature\_dim, isTraining):

with tf.variable\_scope('bnormalization') as scope:

norm\_data\_batch = tcl.batch\_norm(inputs=dataBatch, epsilon=0.0001, is\_training=isTraining,

reuse=True, scope=scope)

# Hidden Layer 1

with tf.variable\_scope('hidden1') as scope:

weights = variable\_on\_cpu('weights',[feature\_dim, 256], tf.truncated\_normal\_initializer(stddev=0.04))

biases = variable\_on\_cpu('biases', [256], tf.constant\_initializer(0.001))

hidden1 = tf.nn.relu(tf.matmul(data\_batch, weights) + biases, name=scope.name)

```

I got the following error in the training phase:

Variable bnormalization/beta does not exist, disallowed. Did you mean to set reuse=None in VarScope?

From what I 've been reading in this thread in the training phase I should be using reuse=None. Have I got this part correct? If this is true, then since I am using two GPUS, should I do reuse=None in the first GPU and reuse=True in the second? Or since I am doing tf.get\_variable\_scope().reuse\_variables() it takes care of itself?

Finally, in the testing phase, should I have is\_training=False and reuse=True?

Any help is greatly appreciated.

-------------------------------------------------------------------------

2016-07-27T18:50:29Z sguada

Now tf.contrib.layers.batch\_norm accepts a Tensor, Variable or Placeholder as is\_training

https://github.com/tensorflow/tensorflow/commit/9da5fc8e6425cabd61fc36f0dcc1823a093d5c1d#diff-94bbcef0ec8a5cdef55f705e99c2b2ed

-------------------------------------------------------------------------

2016-07-28T05:57:03Z brando90

Is it normal that Batch Normalization makes my experiments \*\*worse\*\*? I tried it on a 2 layered NN network based on the MNIST beginner tutorial and I consistently get worse results when BN is present: with BN (one with scale and center trained and the other not) accuracy is 0.8423, 0.8221 and without BN accuracy is 0.9477.

My script is present here https://github.com/brando90/tensor\_flow\_experiments/blob/master/tf\_tutorials/beginner\_tutorial\_MNIST\_BN.py

anyone has experienced these problems or is BN just like this and I need to do something else to make it work?

-------------------------------------------------------------------------

2016-07-28T15:22:08Z sguada

The latest version of [tf.contrib.layers.batch\_norm](https://github.com/tensorflow/tensorflow/blob/master/tensorflow/contrib/layers/python/layers/layers.py#L110) now accepts a placeholder for is\_training so not need to do it yourself.

But what it is important is that either you pass [updates\_collections=None](https://github.com/tensorflow/tensorflow/blob/master/tensorflow/contrib/layers/python/layers/layers.py#L142) so the moving\_mean and moving\_variance are updated in-place, otherwise you will need gather the update\_ops and make sure they are run.

I would like to encourage you to use [`tf.contrib.layers`](https://github.com/tensorflow/tensorflow/tree/master/tensorflow/contrib/layers) or [`tf.contrib.slim`](https://github.com/tensorflow/tensorflow/blob/master/tensorflow/contrib/slim) to build your model.

```

slim = tf.contrib.slim

def build\_NN\_two\_hidden\_layers(x, is\_training):

batch\_norm\_params = {'is\_training': is\_training, 'decay': 0.9, 'updates\_collections': None}

with slim.arg\_scope([slim.fully\_connected],

activation\_fn=tf.nn.relu,

weigths\_initializer=tf.contrib.layers.xavier\_initializer(),

biases\_initializer=tf.constant\_initializer(0.1),

normalizer\_fn=slim.batch\_norm,

normalizer\_params=batch\_norm\_params):

net = slim.fully\_connected(x, 50, scope='A1')

net = slim.fully\_connected(net, 49, scope='A2')

y = slim.fully\_connected(net, 10, activation\_fn=tf.nn.softmax, normalizer\_fn=None, scope='A3')

return y

```

-------------------------------------------------------------------------

2016-07-28T17:03:29Z brando90

@sguada I changed my old one where I manually tell it to train or not (based on a tf.cond) and now it seems the accuracy is up to ~95's again. Why was it that I needed to change updates\_collections to be None? Do you mind explaining me why that gave such a big accuracy difference? Its seems like a non-trivial change (should it None be its default value then if it matters so much?). Thanks! :)

Also, I noticed you said it was a placeholder and I didn't need to do it manually. However, when I passed a placeholder for is\_training it said

`TypeError: Using a`tf.Tensor`as a Python`bool`is not allowed. Use`if t is not None:`instead of`if t:`to test if a tensor is defined, and use the logical TensorFlow ops to test the value of a tensor.`

and pointed to batch\_norm code. Maybe It could be nice to show how this placeholder thing should be used because it seems I don't understand how its suppose to be used. Thanks! :)

-------------------------------------------------------------------------

2016-07-28T17:30:25Z papadopav

@brando90

The relevant part of the code is here [L227-256](https://github.com/tensorflow/tensorflow/blob/98d63de3bb2bab7c9a81f83c8ca864741399300c/tensorflow/contrib/layers/python/layers/layers.py#L227-L256).

As you will notice is there is a `with ops.control\_dependencies` statement that forces the updates. I believe that for the code to be used "right out of the box" the default should be None.

As for my comment above [1122](https://github.com/tensorflow/tensorflow/issues/1122#issuecomment-235433645), I figured out that tf.get\_variable\_scope().reuse\_variables() takes care of the issue, so in the training phase the argument reuse of batch\_norm should be None. It has to do with the statement variable\_op\_scope (read its documentation in tensorflow)

-------------------------------------------------------------------------

2016-07-29T00:59:48Z sguada

Use of batch\_norm with tf.placeholder

```

x = tf.placeholder(tf.float32, [None, 784])

is\_training = tf.placeholder(tf.bool, [], name='is\_training')

y = build\_NN\_two\_hidden\_layers(x, is\_training)

# For training

sess.run(y, {is\_training: True, x: train\_data})

# For eval

sess.run(y, {is\_training: False, x: eval\_data})

```

-------------------------------------------------------------------------

2016-07-29T01:04:51Z sguada

The problem before was that you were not updating the `moving\_mean` and `moving\_variance` after each step, when updates\_collections is None it forces the updates as part of the computation.

However when a network has many batch\_norm layers it is more efficient to collect all the update ops and run them together, so each layer don't need to wait for the update to finish.

```

y = build\_model\_with\_batch\_norm(x, is\_training)

update\_ops = tf.group(tf.get\_collection(tf.GraphKeys.UPDATE\_OPS))

sess.run([y, update\_ops])

```

-------------------------------------------------------------------------

2016-08-08T03:19:34Z ghost

Has there been any progress made with speeding up batch norm?

-------------------------------------------------------------------------

2016-08-10T09:47:42Z brando90

I was trying to use batch norm with a 2 layered densely connected NN with the (flatten) MNIST (and relu units) data set for the task of auto-encoding and I keep getting a NaN error. Anyone know why might this be? Is this ever possible with BN? seem fishy, but it couldn't be my learning set up, rate etc. (but I'd assume it shouldn't because BN should be sort of rubust to this)

-------------------------------------------------------------------------

2016-08-26T16:25:32Z jpiabrantes

@sguada I am not understanding the right way of using `batch\_norm` specially concerning the flag `updates\_collections`. If I understood correctly if the flag is `None` the network is not efficient, so I should let `updates\_collections=tf.GraphKeys.UPDATE\_OPS` and then I should collect all the batch\_norm updates and run them together.

You collect the batch\_norms updates by doing: `update\_ops = tf.group(tf.get\_collection(tf.GraphKeys.UPDATE\_OPS))`.

I have many different models that use different batch\_norm layers, this wouldn't work right?:

``` python

#model 1

y1 = build\_model\_with\_batch\_norm(x, is\_training)

update\_ops1 = tf.group(tf.get\_collection(tf.GraphKeys.UPDATE\_OPS))

sess.run([y1, update\_ops1])

#model 2

y2 = build\_model\_with\_batch\_norm(x, is\_training)

update\_ops2 = tf.group(tf.get\_collection(tf.GraphKeys.UPDATE\_OPS))

sess.run([y2, update\_ops2])

```

Could you explain this part with a bit more details? Thank you very much.

-------------------------------------------------------------------------

2016-09-08T17:05:14Z bsautermeister

Just put it in seperate collection-keys:

```

# While building your 1st model...

tf.contrib.layers.batch\_norm(..., updates\_collection="updates-model1")

# same for 2nd model with key "updates-model2"

```

```

#model 1

y1 = build\_model\_with\_batch\_norm(x, is\_training)

update\_ops1 = tf.group(tf.get\_collection("updates-model1"))

sess.run([y1, update\_ops1])

#model 2

y2 = build\_model\_with\_batch\_norm(x, is\_training)

update\_ops2 = tf.group(tf.get\_collection("updates-model1"))

sess.run([y2, update\_ops2])

```

-------------------------------------------------------------------------

2016-09-08T17:10:54Z bsautermeister

Nevertheless, the documentation seams to be out-dated. It tells to do the following:

```

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)

```

But:

- \_tf.group()\_ does not accept a list. I replaced it with \_tf.tuple()\_

- ~~I don't know how to access \_control\_flow\_ops.with\_dependencies()\_. How can I access functions within control\_flow\_ops module? I have seen other examples just using tf.with\_dependecies(), but I cannot do that with Tensorflow 0.10.~~ I found it here: \_tf.python.control\_flow\_ops.with\_dependencies()\_

\*\*EDIT:\*\*

The documentation should be updated to s.th. like this:

```

from tensorflow.python import control\_flow\_ops

update\_ops = tf.get\_collection(tf.GraphKeys.UPDATE\_OPS)

if update\_ops:

updates = tf.tuple(update\_ops)

total\_loss = control\_flow\_ops.with\_dependencies(updates, total\_loss)

```

\*\*EDIT 2:\*\*

After doing some runs on my network, I have to say that ~~I can not see any performance difference between using \_updates\_collections=None\_ in contrast to manually fetching \_tf.GraphKeys.UPDATE\_OPS\_ while graph construction~~. Even with heavy use of batch normalization (in total, my \_tf.get\_collection(tf.GraphKeys.UPDATE\_OPS)\_ returns 140 Update-Ops, all of them are BN-ops only)

Edit: Hard to say, if my results are correct, but the whole network indeed seams to be 1.5x faster. As far as I know, BN-statistics are calculated on CPU, not GPU so far.

Can anyone of you see any performance benefits as well? Please share your results :)

-------------------------------------------------------------------------

2016-09-16T18:35:06Z brando90

Coming back to the performance issue, does the current batch norm layer benfit at all from GPU usage? Anyone has experienced benefits from GPUs with this batch norm implementation?

-------------------------------------------------------------------------

2016-09-16T18:47:33Z vincentvanhoucke

You can test for yourself:

https://github.com/tensorflow/tensorflow/blob/4addf4b5806cd731949c6582a83f5824599cd1ef/tensorflow/python/ops/batch\_norm\_benchmark.py

-------------------------------------------------------------------------

2016-09-19T03:38:42Z brando90

Sorry for the spam, but the documentation doesn't really explain how to use this BN with convolution (maybe should be provided somewhere?). In short how does it figure out that it should apply and learn the same parameters per feature (rather than per activation)?

(Is there at least a code snippet to do this?)

-------------------------------------------------------------------------

2016-09-19T14:37:11Z vincentvanhoucke

The slim batch\_norm wrapper normalizes over the last dimension of your input tensor. So if it's a 2D input tensor coming from a fully connected layer, it normalizes over batch, and thus performs per-activation normalization. If it's a 4D tensor coming from a convolution, it will normalize over the three first dimensions (batch, width, depth), and thus perform per-feature normalization. @sguada maybe forth being a bit more descriptive about this.

-------------------------------------------------------------------------

2016-11-02T21:32:24Z zhongyuk

@nmhkahn Regarding your code snippet, may I ask why is `reuse` set to be `None` when `is\_training=True`? Wouldn't that trigger the scaling parameter `gamma` and the offset parameter `beta` be re-initialized in every training step? I thought in the original paper, `beta` and `gamma` are "learned along with the original model parameters". To do that, shouldn't they be only initialized once and then reused in all training steps?

`tf.cond(is\_training,

lambda: batch\_norm(inputT, is\_training=True, updates\_collections=None, scope=scope),

lambda: batch\_norm(inputT, is\_training=False, updates\_collections=None, scope=scope, reuse = True))`

-------------------------------------------------------------------------

2016-11-10T18:04:48Z davek44

I greatly appreciate the work that the TF team has put in here to make batch\_norm available and effective. From my searching, this thread is the best resource for how to use it. There are many different problems and ideas flying around here, and it's difficult to figure out the consensus advice for the simplest standard case of how to use the batch\_norm layer. I think there'd be a lot of value in expanding the documentation to specify the exact recommended usage.

My best attempt to figure that out brought me to the following code:

```

is\_training\_ph = tf.placeholder(tf.bool)

...

with tf.variable\_scope('bn\_test\_layer') as vs:

layer\_output = tf.cond(is\_training\_ph,

lambda: tf.contrib.layers.batch\_norm(layer\_input, is\_training=True, center=True, scale=True, activation\_fn=tf.nn.relu, updates\_collections=None, scope=vs),

lambda: tf.contrib.layers.batch\_norm(layer\_input, is\_training=False, center=True, scale=True, activation\_fn=tf.nn.relu, updates\_collections=None, scope=vs, reuse=True))

```

Then I set is\_training\_ph to True for training and False for testing. This doesn't work for me. The model trains fine, but the test performance is terrible. In contrast, if I maintain is\_training\_ph=True for test time, it works great. Thus, I'm guessing I still have a scope issue so that it's not finding the proper existing variables.

-------------------------------------------------------------------------

2016-11-10T19:22:39Z zhongyuk

@davek44 I'm using the same code framework that you are using and I observed the same thing: when turns on `is\_training=True` during training phase and turns off `is\_training=False` for validation and/or testing phase, the model trains well like the paper described (model converges faster and I was able to use a larger learning rate), however the testing performance is terrible. If I turns on `is\_training=True` all the time, the model trains the same as without inserting batch norm layer. I haven't figured out what I did wrong, I'm planning to use TensorBoard to monitor the parameters. Would you please update if you diagnose the cause of this behavior?

-------------------------------------------------------------------------

2016-11-10T23:05:26Z sguada

tf.contrib.layers.batch\_norm can take tensor as is\_training, so not need to do anything especial.

```

is\_training\_ph = tf.placeholder(tf.bool)

outputs = tf.contrib.layers.batch\_norm(layer\_input, is\_training=is\_training\_ph, center=True, scale=True, activation\_fn=tf.nn.relu, updates\_collections=None, scope='batch\_norm'),

```

-------------------------------------------------------------------------

2016-11-11T05:50:14Z davek44

I see the same poor test performance with that code.

-------------------------------------------------------------------------

2016-11-12T01:33:48Z sguada

Without more details is impossible to know, my guesses are that you only train for a few iterations, so the moving\_mean and moving\_average haven't converge yet.

You can change the batch\_size during test to see how the performance degrades as you make your batch smaller.

-------------------------------------------------------------------------

2016-11-16T10:20:57Z nmduc

> I see the same poor test performance with that code.

I had exactly the same problem either with tf.slim batchnorm or with tf.cond and input is\_training as a placeholder.

In the former case, when investigating the trained model, I found out that the moving mean and moving variance consist of all zeros.

In the latter case, the moving mean and variance look more reasonable (with different values), but if I use is\_training=False in test time, the performance is also really bad. Using is\_training=True, it works better but I think it only uses the moving mean and variance inside the test batch.

-------------------------------------------------------------------------

2016-11-16T19:09:16Z zhongyuk

@nmduc @davek44 I wrote some code to track the moving mean and moving variance computed in `tf.contrib.layers.batch\_norm` during training and testing. I found out that the value of `decay` matters a lot (they use exponential decay to compute moving average and moving variance), with a `decay` setting closer to 1.0 (i.e. `decay=.999`), moving mean drops to a value closer to 0. I did 2 test runs with the exact same code but different `decay` settings in the `tf.contrib.layers.batch\_norm`, and my validation/test accuracies seemed more reasonable.

The test run results with `decay=0.9`

<img width="784" alt="screen shot 2016-11-16 at 1 51 51 pm" src="https://cloud.githubusercontent.com/assets/6901075/20361517/dd5dbbd8-ac05-11e6-85ac-5a9e2dec3a2b.png">

The test run results with `decay=0.999` (`decay=0.999` is the default setting in `tf.contrib.layers.batch\_norm`)

<img width="784" alt="screen shot 2016-11-16 at 2 03 58 pm" src="https://cloud.githubusercontent.com/assets/6901075/20361605/31729f5e-ac06-11e6-9736-eb9ad2f15de1.png">

(also seems like larger decay value would require the model to train longer to see validation accuracy change )

-------------------------------------------------------------------------

2016-11-17T06:03:50Z davek44

Yup that fixed it. Thanks for sharing your analysis @zhongyuk!

I encourage the developers to consider making decay=0.9 the default. Even 0.99 doesn't work well for me. That's the default value in Torch's implementation, too; see the momentum parameter in https://github.com/torch/nn/blob/master/BatchNormalization.lua

-------------------------------------------------------------------------

2016-11-17T08:50:04Z nmduc

@zhongyuk Thanks a lot for sharing . It works for me now.

-------------------------------------------------------------------------

2016-11-17T17:12:42Z vincentvanhoucke

This seems important. @sguada we should consider the right course of action here before 1.0. In the short term, can one of the interested parties send me a PR documenting the fact that `decay` might have to be significantly lowered when experiencing poor eval performance? I am pretty sure I've never had to tweak that parameter, but it might be a side effect of the distributed setting.

-------------------------------------------------------------------------

2016-11-17T21:40:42Z sguada

We could change the default to 0.9 or document better its impact in smaller datasets or few updates.

@vincentvanhoucke in our distributed setting we usually do millions of updates so it is ok, however in other cases like the one here which does only a few hundreds of updates it makes a big difference:

For example using decay=0.999 has a 0.36 bias after 1000 updates, but that bias goes down to 0.000045 after 10000 updates and to 0.0 after 50000 updates.

-------------------------------------------------------------------------

2016-11-21T15:53:18Z dominikandreas

Just wanted to note that I also have the problem of poor test performance, specifically using small batch sizes (anything smaller than 10 instead of the 200 I used for training diminishes test accuracy). I've used a tf.placeholder to switch between testing/training mode.

It's great that this batch normalization layer works for better training convergence, but if you can't apply the model in production, there isn't much of a point in using it. Can anyone confirm good test performance with small or single data samples using this batch norm layer?

-------------------------------------------------------------------------

2016-11-21T17:35:42Z sguada

I can confirm that test performance is good when using is\_training=False with small batches and even with batch\_size=1, since it is not using statistic from the batch, but the statistic learnt during training. Just need to make sure that the statistics have converged with default decay=0.999 that implies at least 50k updates.

-------------------------------------------------------------------------

2016-11-21T21:32:53Z zhongyuk

To follow up with TF developer's confirmation, I track the convergence of the statistics with two different `decay` settings (and training batch\_size=1). With `decay=0.99`, the statistics converge (bias<0.001) after 550~600 steps of learning/updates. With `decay=0.9`, the statistics converge (biase<0.001) within within 100 steps of learning/updates.

-------------------------------------------------------------------------

2016-11-21T22:55:00Z dominikandreas

@sguada thanks, does that also mean the output is actually independent of the batch size? because I'm noticing very slight changes with big impact on my accuracy (maybe my definition of performance is just more easily affected by this slight change). To be precise, all values in my 128 dimensional output tensor increase such that the total vector length scales almost linearly with the batch size. Per value this isn't that much of a difference, but has a big impact when computing vector distances in latent spaces.

@zhongyuk thanks, I've run about 5k updates with `decay=0.9`, so it should've converged and testing performance using large batch sizes is fine. But even if it didn't, would it result in a difference between training a testing? I'd be seeing bad performance during training \*and\* testing if it hadn't converged, right?

I will investigate some more and see if I can reproduce the issue on another task. Thanks for the quick feed back so far!

-------------------------------------------------------------------------

2016-11-22T00:16:14Z zhongyuk

@dominikandreas If your poor testing performance is caused by statistics not converging, you'd see reasonably good training performance but bad testing performance. Because during training, the batch normalization is done using the training batch statistics only. However, during testing time, it's using the moving average statistics of all the training batches to normalize the input tensor.

-------------------------------------------------------------------------

2016-11-23T07:12:14Z dominikandreas

I found and error in my code, batch normalization is working fine now :-) thanks for your support

-------------------------------------------------------------------------

2016-11-30T19:19:22Z rogertrullo

Hi @zhongyuk , how did you keep track of the moving mean and variance?

Thanks!

-------------------------------------------------------------------------

2016-11-30T22:46:23Z zhongyuk

@rogertrullo Generally I setup TensorBoard to track moving mean and variance. Other than that, I also tried fetching statistics through `tf.get\_variable("moving\_mean")` within scope during training and reference to monitor the bias.

-------------------------------------------------------------------------

2017-02-02T10:15:22Z ishaybee

hi,

I have same problem as other described that I have good training results but validation/testing is bad after using batch\_norm.

I use the function like this:

conv\_normed1 = tf.contrib.layers.batch\_norm(conv1 + block1\_layer3\_1\_biases, updates\_collections=None, scale=True, decay=batch\_norm\_decay, center=True, is\_training=is\_training )

decay value is 0.9

do I need to set the reuse flag?

I will glad for any help.

-------------------------------------------------------------------------

2017-02-05T20:33:27Z mshunshin

I have been using batch\_norm as described in this thread (with a tf.bool for training; and ops.GraphKeys.UPDATE\_OPS) and everything works.

When saving and restoring using:

saver = tf.train.Saver()

it works,

but when saving using:

saver = tf.train.Saver(tf.trainable\_variables() + [global\_step])

so that I can save storage space (by not saving the gradients etc)

on restore there is an error:

"uninitialized value unpool4/convc/bn/moving\_mean"

Obviously this is because moving\_mean (and I suppose moving\_variance) hasn't been saved for any of the layers. As I have lots of them (nested in many layers) - what is the most efficient way of adding them to the list of values to be saved? Also, given that these are trainable variables, why are they not addded to the trainable\_variables collection?

-------------------------------------------------------------------------

2017-02-05T23:45:40Z DrSleep

@mshunshin moving mean and variance are not trainable variables: there are no gradients coming to them, they are just accumulating statistics across minibatches of examples.

To save/restore them, you can use tf.global\_variables()

-------------------------------------------------------------------------

2017-02-06T08:41:08Z ishaybee

for me things started to work when I used this wrapper:

`def batch\_norm\_wrapper(x, phase, decay, scope, reuse):

with tf.variable\_scope(scope, reuse=reuse):

normed = tf.contrib.layers.batch\_norm(x, center=True, scale=True, decay=decay, is\_training=phase, scope='bn',updates\_collections=None, reuse=reuse)

return normed`

the whole using of scopes and reuse is not clear in this thread for my opinion.

-------------------------------------------------------------------------

2017-02-06T14:26:08Z mshunshin

Many thanks. With tf.global\_variables() the save files are much larger as I think it includes the gradients; in the end I used:

saver = tf.train.Saver([x for x in tf.global\_variables() if 'Adam' not in x.name])

and because the session manager init doesn't initialise them properly:

sess.run(tf.variables\_initializer([x for x in tf.global\_variables() if 'Adam' in x.name]))

(Using tf.train.AdamOptimizer)

-------------------------------------------------------------------------

2017-02-07T00:59:54Z sguada

You can also use tf.model\_variables() which contains the variables of the model, i.e. moving\_mean

-------------------------------------------------------------------------

2017-02-16T10:09:57Z soloice

@sguada Sorry for trouble you, but is it possible to make an example on how to use slim.batch\_norm when combined with slim.conv2d/slim.fully\_connect in readme.md?

I'm using slim.batch\_norm, but get good training performance and poor validation/test performance. I think it must be due to improper use of `reuse` or `scope` or some other parameters. Though there are many issues on batch normalization, it's hard to find a complete code snippet on how to use it, esp. for how to pass different parameters in different phase.

Say, in my [mnist\_bn](https://github.com/soloice/mnist-bn/blob/master/mnist\_bn.py) code, I controlled dependencies using `tf.GraphKeys.UPDATE\_OPS` and set up `is\_training` as a placeholder. But validation performance still is poor if I feed {is\_training: False}.

I would greatly appreciate it if there's an official and complete (which means training, validating, testing are all included) batch normalization example.

Thank you in advance!

-------------------------------------------------------------------------

2017-02-16T10:28:40Z ishaybee

hi,

you need to set different scope for every time you use batch norm and give it the reuse input according to the training/test phase(TRUE when test FALSE when train) that works for me.

-------------------------------------------------------------------------

2017-02-16T13:07:51Z soloice

@ishaybee Thanks for you help. I've found my problem= = \*\*It's due to the cold start of moving\_mean/moving\_variance.\*\*

Since I haven't trained enough steps, the estimated moving mean/variance is not that stable. The result turns out to be: the model performs pretty well on training mini-batches (you know at the beginning loss goes down quickly), but validation performance is erratic (because the estimated population mean/variance are not stable enough).

When I trained the model longer, validation accuracy becomes prettier, too.

\*\*Another important thing is, be sure to use `slim.learning.create\_train\_op` to create train op\*\*. Do not use tf native `tf.train.GradientDescentOptimizer(0.1).minimize(loss)`.

So the answer is, I'm using batch normalization correctly, but I haven't fully understood its dynamics during training.

================

What's more:

1. [Here is a full example](https://github.com/soloice/mnist-bn) on how to use BN layer on MNIST dataset.

2. Use a smaller decay value will accelerate the warm-up phase. The default decay is 0.999, for small datasets such like MNIST, you can choose 0.99 or 0.95, and it warms up in a short time.

-------------------------------------------------------------------------

2017-02-17T13:44:05Z pavelbulanov

@soloice , notice, how in about [comment](https://github.com/tensorflow/tensorflow/issues/1122#issuecomment-235928564) the following parameter is passed inside to the layer for calling batch\_norm:

> batch\_norm\_params = {'is\_training': is\_training, 'decay': 0.9, 'updates\_collections': None}

Without `updates\_collections `set to None (so mean updates are done in place inside BatchNorm), I won't expect surrounding layer (e.g. conv2d) to somehow execute tf.GraphKeys.UPDATE\_OPS needed for BatchNorm layer to update running mean and therefore be able to do run on test data later.

Or you may try to run UPDATE\_OPS yourself explicitly as one [here](https://github.com/tensorflow/tensorflow/issues/7469#issuecomment-279646674)

```

update\_ops = tf.get\_collection(tf.GraphKeys.UPDATE\_OPS)

if update\_ops:

updates = tf.group(\*update\_ops)

cross\_entropy = control\_flow\_ops.with\_dependencies([updates], cross\_entropy)

```

Update - I found that I quoted exactly your code and you do use UPDATE\_OPS.

As for "cold start", as you see above in discussiion, decreasing BatchNorm running average decay (input param) from default 0.999 to something like 0.95 can speed-up start-up

-------------------------------------------------------------------------

2017-02-17T17:35:11Z soloice

@pavelbulanov It's very kind of you to help me with this! I'll try a smaller value of `decay` to see how this helps.

================

Update: use a small decay (say, 0.9 or 0.95) does help a lot. Validation loss goes down very quickly when I set `decay` to 0.9. However, the drawback of small decay is that its effective range is small: The result is dominated by a few recent samples thus it's not a good estimation of population mean/variance. One needs to balance between quick start (small decay) and a longer effective range (large decay).

-------------------------------------------------------------------------

2017-04-01T18:17:58Z Alexivia

Hi,

I tried to implement a batch normalisation layer with the help of the suggestions in this issue, but I still have a >70% error in validation and testing... I do have a lower decay for non-training calls...

Here is my code:

```python

def BatchNorm(inputT, is\_training=False, scope=None):

return tf.cond(

is\_training,

lambda: tf.contrib.layers.batch\_norm(inputT, is\_training=True, reuse=None, decay=0.999, epsilon=1e-5, center=True, scale=True, updates\_collections=None, scope=scope),

lambda: tf.contrib.layers.batch\_norm(inputT, is\_training=False, reuse=True, decay=0.900, epsilon=1e-5, center=True, scale=True, updates\_collections=None, scope=scope)

)

```

Thank you in advance.

-------------------------------------------------------------------------

2017-04-02T02:31:05Z soloice

@Alexivia It seems that you are using two different batch normalization layers? You should use only one BN layer (of course, with different `is\_training` parameter).

-------------------------------------------------------------------------

2017-04-02T12:25:32Z Alexivia

Thank you for your advice @soloice.

I tried now with just different `is\_training` and `reuse` parameters:

```python

lambda: tf.contrib.layers.batch\_norm(inputT, is\_training=True, reuse=None, decay=0.9, epsilon=1e-5, center=True, scale=True, updates\_collections=None, scope=scope),

lambda: tf.contrib.layers.batch\_norm(inputT, is\_training=False, reuse=True, decay=0.9, epsilon=1e-5, center=True, scale=True, updates\_collections=None, scope=scope)

```

still don't get good validation and testing results... >70%...

-------------------------------------------------------------------------

2017-04-02T19:17:02Z ishaybee

hi,

please see my wrapper above.

you should use "with tf.variable\_scope(scope, reuse=reuse):" I think.

-------------------------------------------------------------------------

2017-04-03T16:23:41Z Alexivia

Hi @ishaybee,

I followed your advice, now my code is:

```python

def BatchNorm(inputT, is\_training=False, reuse=True, scope=None):

with tf.variable\_scope(scope, reuse=reuse):

return tf.contrib.layers.batch\_norm(inputT, is\_training=is\_training, reuse=reuse, scope=scope, updates\_collections=None, decay=0.9, center=True, scale=True)

```

and I feed `is\_training` and `reuse` through the feed\_dict, but now I get the error `ValueError("The reuse parameter must be True or False or None.")`

-------------------------------------------------------------------------

2017-04-03T16:41:49Z ishaybee

try to feed reuse as a python variable (input of the model) and as placeholder.

-------------------------------------------------------------------------

2017-04-03T18:57:48Z Alexivia

I tried that, and now it stopped complaining about the value... but I think that the placeholder value is not being used, because I see no change if I force values to `batch\_norm` function, and in TensorBoard it's not connected to the graph... (see attached image)

![screen shot 2017-04-03 at 19 54 54](https://cloud.githubusercontent.com/assets/23476569/24625718/cbe90f88-18a7-11e7-80a6-f087d56dd9ad.png)

-------------------------------------------------------------------------

2017-04-03T19:06:19Z Alexivia

My code is like this now:

\*\*Batch Normalisation wrapper\*\*

```python

def BatchNorm(inputT, is\_training=False, reuse=None, scope=None):

with tf.variable\_scope(scope):

return tf.contrib.layers.batch\_norm(inputT, is\_training=is\_training, reuse=reuse, scope=scope, updates\_collections=None, decay=0.9, center=True, scale=True)

```

\*\*Model definition\*\*

```python

def model(data, train=False, is\_training=False, reuse=None):

# 1st conv layer

with tf.name\_scope('conv1') as scope:

conv = tf.nn.conv2d(

<...>

norm = BatchNorm(pool, is\_training=is\_training, reuse=reuse, scope=scope)

```

\*\*Training\*\*

```python

feed\_dict = {train\_data\_node: batch\_data,

train\_labels\_node: batch\_labels,

is\_training: True,

reuse: None}

# Run the optimizer to update weights.

sess.run(optimizer, feed\_dict=feed\_dict)

```

\*\*Validation\*\*

```python

batch\_predictions = sess.run(eval\_prediction, feed\_dict={eval\_data: data[-EVAL\_BATCH\_SIZE:, ...], is\_training: False, reuse: True})

```

-------------------------------------------------------------------------

2017-04-04T00:08:04Z sguada

Although is\_traning can a placeholder reuse has to be a bool, and it cannot be a tensor nor a placeholder.

I'm not sure what are you trying to do, in most cases using static values solve the problem. For example this pattern works well:

```

def model(data, is\_training=False, reuse=None, scope='my\_model'):

# Define a variable scope to contain all the variables of your model

with tf.variable\_scope(scope, 'model', data, reuse=reuse):

# 1 layer

net = tf.contrib.layers.conv2d(data, ....)

....

net = tf.contrib.layers.batch\_norm(net, is\_training)

return net

train\_outputs = model(train\_data, is\_training=True)

eval\_outputs = model(eval\_data, is\_training=False, reuse=True)

eval\_predictions = sess.run(eval\_outputs, feed\_dict={eval\_data: data[-EVAL\_BATCH\_SIZE:, ...]})

```

Unless you need to change the behavior of the model dynamically, you don't need to use a placeholder for is\_training. The trick is to build the model twice, but sharing the variables the second time.

-------------------------------------------------------------------------

2017-04-04T10:49:44Z Alexivia

Thank you @sguada ! After applying your suggestions, I finally made it to work!

-------------------------------------------------------------------------

2017-04-05T03:10:30Z danrsc

It would be helpful if the API 1.0 documentation reflected that you need to manually add update ops to the graph. Being a newer tf user, I found that my test error was crazy and then had to spend a fair amount of time debugging my graph until I realized that batch normalization was the problem. Then I had to spend more time figuring out that by default the variables tracking the moments don't update unless you use a contrib function for optimization. Since in 1.0 there is no option to set the update\_collections to None, there is no indicator from the documentation that this might even be an issue. Additionally, it seems like it might make sense to have a parameter to add the control flow dependencies to the op that runs in the training case.

-------------------------------------------------------------------------

2017-04-05T04:41:58Z soloice

@danrsc Exactly. The usage of BN layer is quite confusing. I suggested to add documents or a complete official tutorial on batch normalization, but unfortunately got no response = =

-------------------------------------------------------------------------

2017-04-05T13:53:44Z alquraishi

Completely agree. I think BN usage is very tricky and the documentation is currently beyond inadequate. This ought to be fixed for such a commonly used layer.

-------------------------------------------------------------------------

2017-04-06T17:26:01Z vincentvanhoucke

Reopening for visibility of the documentation issues.

-------------------------------------------------------------------------

2017-04-06T17:29:18Z vincentvanhoucke

@sguada assigning to you for triaging. Might be worth getting a tech writer on the case.

-------------------------------------------------------------------------

2017-04-23T01:03:51Z ronghanghu

Just got confused by this problem last week and wasted 3 days of training... Hope the docs can be fixed soon, and an official batch normalization example can be added in the API docs.

-------------------------------------------------------------------------

2017-05-04T08:27:22Z MisayaZ

@sguada I have noticed that you said" tf.contrib.layers.batch\_norm can take tensor as is\_training, so not need to do anything especial".

Howerver, the comment in the code is

If `is\_training` doesn't have a constant value, because it is a `Tensor`,

# a `Variable` or `Placeholder` then is\_training\_value will be None and

# `needs\_moments` will be true.

Does it mean that nees\_moments will be true even in test phase if i set is\_training as a placeholder?

As far as I know, the moments is not needed while testing.

-------------------------------------------------------------------------

2017-05-05T22:39:04Z sguada

So if `is\_training` is a `Variable` or a `Placeholder`, it means it can change, so the graph to compute the moments is needed, so the layer builds it.

Then in running time depending on the value being `True` or `False` would use the batch `moments` or the `moving\_mean` and `moving\_variance`.

So during testing you would set the value to `False` and the `moments` won't be used.

-------------------------------------------------------------------------

2017-05-06T07:51:03Z MisayaZ

@sguada @brando90

```

def batch\_norm\_layer(self, x,train\_phase, scope\_bn):

bn\_train = batch\_norm(x, decay=0.9, center=False, scale=True,

updates\_collections=None,

is\_training=True,

reuse=None,

variables\_collections= [UPDATE\_OPS\_COLLECTION],

trainable=True,

scope=scope\_bn)

bn\_inference = batch\_norm(x, decay=0.9, center=False, scale=True,

updates\_collections=None,

is\_training=False,

reuse=True,

variables\_collections= [UPDATE\_OPS\_COLLECTION],

trainable=True,

scope=scope\_bn)

z = tf.cond(train\_phase, lambda: bn\_train, lambda: bn\_inference)

return z

```

I build batchnorm like this, however, the moving mean and moving variable are updated during test, I can not find the reason.

-------------------------------------------------------------------------

2017-05-06T11:34:51Z OktayGardener

I tried creating two models like @sguada said, however, my model where is\_training=False just crashes.

```

W tensorflow/core/framework/op\_kernel.cc:993] Not found: Key fully\_connected\_5/weights not found in checkpoint

W tensorflow/core/framework/op\_kernel.cc:993] Not found: Key fully\_connected\_6/weights not found in checkpoint

W tensorflow/core/framework/op\_kernel.cc:993] Not found: Key fully\_connected\_7/biases not found in checkpoint

W tensorflow/core/framework/op\_kernel.cc:993] Not found: Key fully\_connected\_6/biases not found in checkpoint

W tensorflow/core/framework/op\_kernel.cc:993] Not found: Key fully\_connected\_7/weights not found in checkpoint

W tensorflow/core/framework/op\_kernel.cc:993] Not found: Key history\_embeddings\_1 not found in checkpoint

W tensorflow/core/framework/op\_kernel.cc:993] Not found: Key global\_step\_1 not found in checkpoint

```

I feel like maybe there should be a concrete example of how to do a batch norm with a fully connected net, as well as with CNNs. Sucks that I've trained models for days expecting things to work before seeing that everyone trying to use this feature going crazy.

Interestingly enough, it takes a zillion years to get the model restored after training with batch\_norm as well. Will most likely wait until TF 2.0 to try something like this again.

-------------------------------------------------------------------------

2017-05-06T19:21:29Z sguada

@MisayaZ you don't need to create two batch\_norm layers you can just pass train\_phase (assuming it is a tf.bool) to batch\_norm. Also you are passing UPDATE\_OPS\_COLLECTION variables\_collections, which changes which collections are the variables added to.

The following should work:

```

z = batch\_norm(x, decay=0.9, center=False, scale=True, updates\_collections=None,

is\_training=train\_phase, scope=scope\_bn)

```

-------------------------------------------------------------------------

2017-05-06T19:32:36Z sguada

@OktayGardener not sure what model are you trying to create, it seems that the variables are not saved in your checkpoint.

batch\_norm also works with fully\_connected layers.

```

slim = tf.contrib.slim

def model(data, is\_training=False, reuse=None, scope='my\_model'):

# Define a variable scope to contain all the variables of your model

with tf.variable\_scope(scope, 'model', data, reuse=reuse):

# Configure arguments of fully\_connected layers

with slim.arg\_scope([slim.fully\_connected],

activation\_fn=tf.nn.relu,

normalizer\_fn=slim.batch\_nom):

# Configure arguments of batch\_norm layers

with slim.arg\_scope([slim.batch\_norm],

decay=0.9, # Adjust decay to the number of iterations

update\_collections=None, # Make sure updates happen automatically

is\_training=is\_training, # Switch behavior from training to non-training):

net = slim.fully\_connected(data, 100, scope='fc1')

net = slim.fully\_connected(net, 200, scope='fc2')

....

# Don't use activation\_fn nor batch\_norm in the last layer

net = slim.fully\_connected(net, 10, activation\_fn=None, normalizer\_fn=None, scope='fc10')

return net

```

-------------------------------------------------------------------------

2017-05-07T10:52:56Z MisayaZ

@sguada Thanks, I build a network with bathnorm which is implemented as you mentioned above

```

z = batch\_norm(x, decay=0.9, center=False, scale=True, updates\_collections=None,

is\_training=train\_phase, scope=scope\_bn)

```

the speed is slow, I use tensorflow benchmark to get the computation time as below:

I tensorflow/core/util/stat\_summarizer.cc:392] ============================== Top by Computation Time ==============================

I tensorflow/core/util/stat\_summarizer.cc:392] [node type] [start] [first] [avg ms] [%] [cdf%] [mem KB] [Name]

I tensorflow/core/util/stat\_summarizer.cc:392] Conv2D 106.164 51.354 51.004 23.145% 23.145% 692.224 conv8/Conv2D

I tensorflow/core/util/stat\_summarizer.cc:392] Conv2D 85.187 19.115 19.283 8.750% 31.896% 692.224 conv7/Conv2D

I tensorflow/core/util/stat\_summarizer.cc:392] SquaredDifference 11.967 15.105 14.331 6.503% 38.399% 11075.584 conv1/batch\_norm/moments/sufficient\_statistics/SquaredDifference

I tensorflow/core/util/stat\_summarizer.cc:392] Mul 11.970 14.162 13.495 6.124% 44.523% 11075.584 conv1/batch\_norm/batchnorm/mul\_1

I tensorflow/core/util/stat\_summarizer.cc:392] Conv2D 3.948 8.170 7.986 3.624% 48.146% 11075.584 conv1/Conv2D

I tensorflow/core/util/stat\_summarizer.cc:392] Sub 11.960 10.176 7.943 3.604% 51.751% 11075.584 conv1/batch\_norm/moments/sufficient\_statistics/Sub

I tensorflow/core/util/stat\_summarizer.cc:392] SquaredDifference 45.570 5.908 7.177 3.257% 55.007% 5537.792 conv2/batch\_norm/moments/sufficient\_statistics/SquaredDifference

I tensorflow/core/util/stat\_summarizer.cc:392] Mul 45.574 7.755 6.902 3.132% 58.140% 5537.792 conv2/batch\_norm/batchnorm/mul\_1

I tensorflow/core/util/stat\_summarizer.cc:392] Conv2D 40.692 5.408 4.845 2.199% 60.338% 5537.792 conv2/Conv2D

I tensorflow/core/util/stat\_summarizer.cc:392] Sub 45.563 6.067 4.784 2.171% 62.509% 5537.792 con

I don't understand why some op in moment are executed during test and it cost a lot of time, such as conv1/batch\_norm/moments/sufficient\_statistics/SquaredDifference.

The moment is not needed in test, why are some ops under moment executed?

-------------------------------------------------------------------------

2017-05-10T15:20:05Z raghavgoyal14

Hi,

Using the above `batch\_norm` layer in `contrib.layers`, I'm getting `nan` as an output for validation graph while the train graph runs seamlessly. Is there anything that I might be missing ?

I'm using:

```python

def batchnormlayer(inputs, numout, train\_model):

with tf.variable\_scope("batch\_norm") as scope\_bn:

epsilon = 1e-3

return tf.contrib.layers.batch\_norm(inputs, decay=0.9, updates\_collections=None,

scale=True, scope=scope\_bn,

is\_training=train\_model, epsilon=epsilon,

fused=True, reuse=scope\_bn.reuse)

```

Thanks

-------------------------------------------------------------------------

2017-05-11T13:00:11Z raghavgoyal14

As a follow up, I'm reusing 16 layers of batch\_norm.

However, I found that reusing 4 layers works.

-------------------------------------------------------------------------

2017-05-11T15:14:28Z danrsc

I've just been noticing that if I kill the tensorflow process and restart it, my error gets worse for a few epochs (i.e. worse than it should be at the last checkpoint). I also observe that if I remove batch\_norm, this problem goes away. After looking at the code for a while, I think this may be because the values of the variables are not restored from the shadow variables as they would be if the ExponentialMovingAverages class were used to manage the moving averages. This also means that if I use a separate process to evaluate, I'm getting whatever the last value of the variable was and not the moving average. Am I interpreting this correctly and is this the intended behavior? It seems like you want the shadow variable values to be restored...

-------------------------------------------------------------------------

2017-05-12T10:52:49Z raghavgoyal14

I caught the problem, the moving variance in my case goes negative after some iterations.

The output of the tensor : `Model/clip\_logits/batch\_norm/moving\_variance:0` present in `tf.model\_variables()` is

```

Moving variance (shape = (101,)) =

[ 214.70379639 95.36338043 0.57885742 189.49542236 102.72473145

137.14886475 286.57333374 111.06427002 154.98750305 167.75219727

207.83955383 211.14007568 158.23495483 171.61665344 116.81361389

115.77380371 43.59399796 137.75064087 181.75245667 161.37339783

215.21934509 92.88521576 191.23846436 336.3946228 259.85919189

299.47039795 186.23222351 165.19311523 262.82446289 170.11567688

233.56843567 209.35050964 115.96807861 154.34109497 295.5770874

123.6055603 295.76187134 296.88583374 240.88217163 247.32983398

87.15661621 217.69897461 133.00698853 -4.80375671 344.77462769

291.50601196 117.77174377 265.83712769 207.90093994 194.186203

220.21418762 178.03738403 115.27571869 196.62184143 228.8089447

191.53205872 331.36807251 151.55435181 197.2951355 179.67504883

181.09727478 90.09922791 173.30133057 102.6836853 160.9434967

236.59512329 168.05305481 403.36340332 41.14326096 185.93409729

130.57434082 266.31509399 101.44387817 163.88059998 290.25015259

244.52597046 229.86647034 158.14352417 202.68774414 187.78227234

248.78218079 126.0978241 171.41891479 274.40740967 119.84254456

202.53045654 200.20608521 214.04730225 111.53284454 222.03184509

244.81187439 172.23052979 187.09806824 194.62802124 255.26345825

293.63598633 307.91036987 210.86982727 308.88919067 144.94792175

229.69013977]

```

As you can see, there's negative variance for one of the dimension. How is this even possible ?

P.S. The batch norm layer is used just after the last fully connected layer of the network and before softmax.

-------------------------------------------------------------------------

2017-05-20T12:13:29Z abred

@raghavgoyal14 are you using it with fused=True? Had a similar problem and it went away when I used the fused version

-------------------------------------------------------------------------

2017-05-20T12:44:21Z raghavgoyal14

@abred : Yes, I used `fused=True`, same problem.

-------------------------------------------------------------------------

2017-07-10T03:40:08Z zmlmanly

@sguada Hi, sguada, I have a problem.

The definition of contrib.layers.batch\_norm in tensorflow:

def batch\_norm(inputs,

decay=0.999,

center=True,

scale=False,

epsilon=0.001,

activation\_fn=None,

param\_initializers=None,

param\_regularizers=None,

updates\_collections=ops.GraphKeys.UPDATE\_OPS,

is\_training=True,

reuse=None,

variables\_collections=None,

outputs\_collections=None,

trainable=True,

batch\_weights=None,

fused=False,

data\_format=DATA\_FORMAT\_NHWC,

zero\_debias\_moving\_mean=False,

scope=None,

renorm=False,

renorm\_clipping=None,

renorm\_decay=0.99):

scale: If True, multiply by gamma. If False, gamma is

not used. When the next layer is linear (also e.g. nn.relu), this can be

disabled since the scaling can be done by the next layer.

If I use tf.contrib.layers.batch\_norm(input, scale=False) , the"scale =False" means whether the gamma is zero in "y = gamma\*x+beta" while training. Thank you very much.

-------------------------------------------------------------------------

2017-07-10T03:43:50Z ppwwyyxx

When scale=False, gamma is a constant 1.

-------------------------------------------------------------------------

2017-07-10T07:21:05Z zmlmanly

@ppwwyyxx Thank you very much for your help. I use tf.contrib.layers.batch\_norm(input, scale=False) in Tensorflow, and now I am convering the batchnorm of Tensorflow to Caffe. How to set the param of BatchNormLayer and ScaleLayer in Caffe?

Thank you very much.

-------------------------------------------------------------------------

2017-07-20T14:14:05Z tano297

@MisayaZ I was having the same behavior using Batchnorm with a placeholder for "is\_training". I see in the trace that the moments are being calculated even at test time, so I decided to go into the source code and I found this:

```python

# If `is\_training` doesn't have a constant value, because it is a `Tensor`,

# a `Variable` or `Placeholder` then is\_training\_value will be None and

# `needs\_moments` will be true.

is\_training\_value = utils.constant\_value(is\_training)

need\_moments = is\_training\_value is None or is\_training\_value

if need\_moments:

# here it defines the moments

```

It looks like when "is\_training" is a variable or a placeholder the moments get defined and also get calculates them at runtime, even when you set the placeholder to "False". I would have preferred to leave it as a placeholder because this way I can do periodic testing during training without redefining the graph, but I decided to use it as a constant and define different behaviors for train vs test, and now the moments are not calculated at test time.

-------------------------------------------------------------------------

2017-07-21T09:56:14Z MisayaZ

@tano297 Thank you. I now also use 'is\_training' as a constant. Leave it as a placeholder and do periodic testing will change the value of moving mean and moving variance. And the inference time will be longer for it will calculate the mean and variance of the inputs and update the moving mean and moving variance. The right way to do testing is to define different behaviors for train and test as you mentioned.

-------------------------------------------------------------------------

2017-07-21T11:28:03Z abred

@tano297 @MisayaZ

but doesn't the "smart\_cond" in

```

is\_training\_value = utils.constant\_value(is\_training)

need\_updates = is\_training\_value is None or is\_training\_value

if need\_updates:

...

outputs = utils.smart\_cond(is\_training, \_force\_updates, no\_updates)

```

make sure that the updates are only calculated and applied if is\_training evaluates to True?

-------------------------------------------------------------------------

2017-07-21T15:56:55Z tano297

@abred Yes indeed, but you are referring to line 391, where it does the update of the moving average within \_fused\_batch\_norm():

```py

# If `is\_training` doesn't have a constant value, because it is a `Tensor`,

# a `Variable` or `Placeholder` then is\_training\_value will be None and

# `need\_updates` will be true.

is\_training\_value = utils.constant\_value(is\_training)

need\_updates = is\_training\_value is None or is\_training\_value

if need\_updates:

...

outputs = utils.smart\_cond(is\_training, \_force\_updates, no\_updates)

...

```

I am talking about line 753 within batch\_norm():

```py

# If `is\_training` doesn't have a constant value, because it is a `Tensor`,

# a `Variable` or `Placeholder` then is\_training\_value will be None and

# `needs\_moments` will be true.

is\_training\_value = utils.constant\_value(is\_training)

need\_moments = is\_training\_value is None or is\_training\_value

if need\_moments:

...

mean, variance = utils.smart\_cond(is\_training,

\_force\_updates,

moving\_vars\_fn)

...

```

The smart condition in that case (as far as I am concerned) decides wether or not to update the moving averages, but the moments still get calculated.

-------------------------------------------------------------------------

2017-07-21T17:29:23Z abred

@tano297 you right about that, I was in the wrong place, but still:

line 755-770 calculate the moments, but the moments are only used in \_force\_updates which is only executed if is\_training evaluates to True, aren't they?

And thus

```

mean, variance = utils.smart\_cond(is\_training, \_force\_updates, moving\_vars\_fn)

```

should be equivalent to line 804:

```

mean, variance = moving\_mean, moving\_variance

```

if is\_training evalutes to False and thus the "moments"-part of the graph is never used and thus shouldn't be executed

but I haven't tested, so I might be wrong about that :)

-------------------------------------------------------------------------

2017-07-24T08:34:08Z MisayaZ

@tano297 @abred you right. The moving mean and moving variance are changed when I used batchnorm like this:

def batch\_norm\_layer(self, x,train\_phase, scope\_bn):

bn\_train = batch\_norm(x, decay=0.9, center=False, scale=True,

updates\_collections=None,

is\_training=True,

reuse=None,

variables\_collections= [UPDATE\_OPS\_COLLECTION],

trainable=True,

scope=scope\_bn)

bn\_inference = batch\_norm(x, decay=0.9, center=False, scale=True,

updates\_collections=None,

is\_training=False,

reuse=True,

variables\_collections= [UPDATE\_OPS\_COLLECTION],

trainable=True,

scope=scope\_bn)

z = tf.cond(train\_phase, lambda: bn\_train, lambda: bn\_inference)

return z

If you use like following:

z = batch\_norm(x, decay=0.9, center=False, scale=True, updates\_collections=None,

is\_training=train\_phase, scope=scope\_bn)

The moving mean and moving variance will not be changed during test, but the speed is very slow.

-------------------------------------------------------------------------

2017-08-04T11:26:51Z tyshiwo

Hi @zhongyuk ,

I also met the problem that I could get good results when using is\_training=True for both training and inference, but get bad results when setting is\_training=False during inference (worse than the case using is\_training=True). According to your analysis, If I understand correctly, by simply setting decay=0.9 in BN can solve this problem. Am I right?

BTW, do I need to retrain the model using decay=0.9 from scratch? Or resuming training from the checkpoint (i.e., trained when decay=0.999) is also ok?

Thanks!

-------------------------------------------------------------------------

2017-08-05T04:25:13Z tyshiwo

@nmduc @davek44

Hi, I also met the problem that I could get good results when using is\_training=True for both training and inference, but get bad results when setting is\_training=False during inference (worse than the case using is\_training=True). Have you guys solved this problem? Thanks!

-------------------------------------------------------------------------

2017-08-05T07:58:30Z nmduc

@tyshiwo I just set decay=0.9 for batch\_norm and it works well so far.

-------------------------------------------------------------------------

2017-08-10T13:54:40Z issa-s-ayoub

I was confused after all these comments on how to properly use Batch Norm: So here is what I have. Please correct me if I'm wrong.

`batch\_norm = tf.contrib.layers.batch\_norm(conv,

center=True,

scale=True,

reuse=phase\_train\_py,

scope='bn',

is\_training=is\_training)`

where phase\_train\_py is a python boolean variable and is\_training is a placeholder taking a boolean variable. I guess using tf.cond is wrong, otherwise would did the function came with a boolean parameters. In other words, if `tf.cond` is true, then we should a `batch\_norm` function for training and another one for testing. So, developers allow us to change these boolean variables in order to change the behavior of the function. So What I am doing is: setting `phase\_train\_py` to False while training while `is\_training` to True. And the opposite while Testing. Since we can only change tensors or placeholders with `sess.run`, I changed `phase\_train\_py` intentionally before running the graph. Ex:

` if condition:

phase\_train\_py = False

sess.run(to\_run\_list, feed\_dict={phase\_train: True})

else:

phase\_train\_py = True

sess.run(to\_run\_list, feed\_dict={phase\_train: False})`

-------------------------------------------------------------------------

2017-09-16T03:51:25Z zhimengfan1990

+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

MAYBE YOU NEED READ THIS

+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

It seems there are still problems with TF v1.3. I'm sure I note the following details, but still failed to use the official `tf.contrib.layers.batch\_norm`, with `is\_training=False` during evaluation(but when I keep `is\_training=True` unchanged during evaluation, it is ok):

1.` decay`, exponential moving average is actually alpha filter in signal processing, the time to converge is approximately 1/(1-decay) steps of train. For decay=0.999, you need 1/0.001=1000 steps to converge. So set the appropriate decay for your training step numbers.

2. using placeholder to switch between train and test evaluation

3. use` updates\_collections=None` if you don't want to add control dependencies of update op to train\_op

4. set `reuse` to appropriate value.

It seems the only way to use the official batch\_norm is to build two graphs, one for train and one for evaluation, with `is\_training=True` and `is\_training=False`, respectively. In this way, you don't need to switch dynamically between train and evaluation. But this is a stupid way since you need to build more than one graph.

Finally, I write a moving average by myself, and I find it worked! It's as follows(based on code on the web and modified by myself)

```

def bn\_layer(x, scope, is\_training, epsilon=0.001, decay=0.99, reuse=None):

"""

Performs a batch normalization layer

Args:

x: input tensor

scope: scope name

is\_training: python boolean value

epsilon: the variance epsilon - a small float number to avoid dividing by 0

decay: the moving average decay

Returns:

The ops of a batch normalization layer

"""

with tf.variable\_scope(scope, reuse=reuse):

shape = x.get\_shape().as\_list()

# gamma: a trainable scale factor

gamma = tf.get\_variable("gamma", shape[-1], initializer=tf.constant\_initializer(1.0), trainable=True)

# beta: a trainable shift value

beta = tf.get\_variable("beta", shape[-1], initializer=tf.constant\_initializer(0.0), trainable=True)

moving\_avg = tf.get\_variable("moving\_avg", shape[-1], initializer=tf.constant\_initializer(0.0), trainable=False)

moving\_var = tf.get\_variable("moving\_var", shape[-1], initializer=tf.constant\_initializer(1.0), trainable=False)

if is\_training:

# tf.nn.moments == Calculate the mean and the variance of the tensor x

avg, var = tf.nn.moments(x, np.arange(len(shape)-1), keep\_dims=True)

avg=tf.reshape(avg, [avg.shape.as\_list()[-1]])

var=tf.reshape(var, [var.shape.as\_list()[-1]])

#update\_moving\_avg = moving\_averages.assign\_moving\_average(moving\_avg, avg, decay)

update\_moving\_avg=tf.assign(moving\_avg, moving\_avg\*decay+avg\*(1-decay))

#update\_moving\_var = moving\_averages.assign\_moving\_average(moving\_var, var, decay)

update\_moving\_var=tf.assign(moving\_var, moving\_var\*decay+var\*(1-decay))

control\_inputs = [update\_moving\_avg, update\_moving\_var]

else:

avg = moving\_avg

var = moving\_var

control\_inputs = []

with tf.control\_dependencies(control\_inputs):

output = tf.nn.batch\_normalization(x, avg, var, offset=beta, scale=gamma, variance\_epsilon=epsilon)

return output

def bn\_layer\_top(x, scope, is\_training, epsilon=0.001, decay=0.99):

"""

Returns a batch normalization layer that automatically switch between train and test phases based on the

tensor is\_training

Args:

x: input tensor

scope: scope name

is\_training: boolean tensor or variable

epsilon: epsilon parameter - see batch\_norm\_layer

decay: epsilon parameter - see batch\_norm\_layer

Returns:

The correct batch normalization layer based on the value of is\_training

"""

#assert isinstance(is\_training, (ops.Tensor, variables.Variable)) and is\_training.dtype == tf.bool

return tf.cond(

is\_training,

lambda: bn\_layer(x=x, scope=scope, epsilon=epsilon, decay=decay, is\_training=True, reuse=None),

lambda: bn\_layer(x=x, scope=scope, epsilon=epsilon, decay=decay, is\_training=False, reuse=True),

)

```

Just use the `bn\_layer\_top` function during building a graph, the is\_training parameter is a `tf.placeholder`

. Then you are free to switch the placeholder to True during train and False during evaluation, with `feed\_dict`.

Hope it helps the community.

-------------------------------------------------------------------------

2017-12-07T14:25:18Z tasx0823

When you use slim.batch\_norm,be sure to use "slim.learning.create\_train\_op" instead of "tf.train.GradientDecentOptimizer(lr).minimize(loss)" or other optimizer. Try it to see if it works!

-------------------------------------------------------------------------

2017-12-10T10:05:16Z ZahlGraf

@vincentvanhoucke You wrote in another post in this thread:

> The slim batch\_norm wrapper normalizes over the last dimension of your input tensor. So if it's a 2D input tensor coming from a fully connected layer, it normalizes over batch, and thus performs per-activation normalization. If it's a 4D tensor coming from a convolution, it will normalize over the three first dimensions (batch, width, depth), and thus perform per-feature normalization. @sguada maybe forth being a bit more descriptive about this.

Do you mean with "slim batch\_norm wrapper" the function `tf.contrib.layers.batch\_norm`? If so, I would suggest to add this information to the documentation text of this function. Thus it gets very clear, that this function performs the batch normalization exactly like described in the paper... for both FC-Layer and Conv2D-Layer. At the moment there is only the text "Can be used as a normalizer function for conv2d and fully\_connected.", where it is not clear if this is related to the normalization axis topic.

-------------------------------------------------------------------------

2017-12-11T16:20:58Z vincentvanhoucke

@ZahlGraf I'll happily consider a PR that clarifies the documentation. We've been at this for so long that I no longer have a good sense of what's obvious or not, and would welcome clarifying documentation for someone with a fresh perspective on the topic.

-------------------------------------------------------------------------

2017-12-27T10:50:05Z Netzeband

@vincentvanhoucke

I created a PR with a more detailed description, mainly based on your statement in this thread:

https://github.com/tensorflow/tensorflow/pull/15653

-------------------------------------------------------------------------

2018-01-11T19:01:28Z tensorflowbutler

Please remove the assignee, as this issue is inviting external contributions. Otherwise, remove the `contributions welcome` label. Thank you.

-------------------------------------------------------------------------

2018-02-06T07:48:54Z tensorflowbutler

Please remove the assignee, as this issue is inviting external contributions. Otherwise, remove the `contributions welcome` label. Thank you.

-------------------------------------------------------------------------

2018-02-07T23:17:07Z annarev

Closing this bug since the original request to add a batch norm layer has been addressed. Some of the more recent issues with documentation seem to have their own PRs

If you see any issue with batch\_norm, please either ask a question on StackOverflow or open another issue.

-------------------------------------------------------------------------

bel. Thank you.

-------------------------------------------------------------------------

2018-02-07T23:17:07Z annarev

Closing this bug since the original request to add a batch norm layer has been addressed. Some of the more recent issues with documentation seem to have their own PRs

If you see any issue with batch\_norm, please either ask a question on StackOverflow or open another issue.

-------------------------------------------------------------------------