Batch Normalization – Solutions

Batch normalization is most useful when building deep neural networks. To demonstrate this, we'll create a convolutional neural network with 20 convolutional layers, followed by a fully connected layer. We'll use it to classify handwritten digits in the MNIST dataset, which should be familiar to you by now.

This is **not** a good network for classfying MNIST digits. You could create a *much* simpler network and get *better* results. However, to give you hands-on experience with batch normalization, we had to make an example that was:

1. Complicated enough that training would benefit from batch normalization.
2. Simple enough that it would train quickly, since this is meant to be a short exercise just to give you some practice adding batch normalization.
3. Simple enough that the architecture would be easy to understand without additional resources.

This notebook includes two versions of the network that you can edit. The first uses higher level functions from the `tf.layers`package. The second is the same network, but uses only lower level functions in the `tf.nn` package.

The following cell loads TensorFlow, downloads the MNIST dataset if necessary, and loads it into an object named `mnist`. You'll need to run this cell before running anything else in the notebook.

In [1]:

```
import tensorflow as tf
from tensorflow.examples.tutorials.mnist import input_data
mnist = input_data.read_data_sets("MNIST_data/", one_hot=True, reshape=False)
```

```
Extracting MNIST_data/train-images-idx3-ubyte.gz
Extracting MNIST_data/train-labels-idx1-ubyte.gz
Extracting MNIST_data/t10k-images-idx3-ubyte.gz
Extracting MNIST_data/t10k-labels-idx1-ubyte.gz
```

Batch Normalization using `tf.layers.batch_normalization`

This version of the network uses `tf.layers` for almost everything, and expects you to implement batch normalization using `tf.layers.batch_normalization`
We'll use the following function to create fully connected layers in our network. We'll create them with the specified number of neurons and a ReLU activation function.

This version of the function does not include batch normalization.

In [2]:

```
"""
DO NOT MODIFY THIS CELL
"""
def fully_connected(prev_layer, num_units):
    """
    Create a fully connectd layer with the given layer as input and the given number of neurons.

    :param prev_layer: Tensor
        The Tensor that acts as input into this layer
    :param num_units: int
        The size of the layer. That is, the number of units, nodes, or neurons.
    :returns Tensor
        A new fully connected layer    """
```

```
    layer = tf.layers.dense(prev_layer, num_units, activation=tf.nn.relu)
    return layer
```

We'll use the following function to create convolutional layers in our network. They are very basic: we're always using a 3x3 kernel, ReLU activation functions, strides of 1x1 on layers with odd depths, and strides of 2x2 on layers with even depths. We aren't bothering with pooling layers at all in this network.

This version of the function does not include batch normalization.

In [3]:

```
def conv_layer(prev_layer, layer_depth):
    """
    Create a convolutional layer with the given layer as input.

    :param prev_layer: Tensor
        The Tensor that acts as input into this layer
    :param layer_depth: int
        We'll set the strides and number of feature maps based on the layer's
depth in the network.
        This is *not* a good way to make a CNN, but it helps us create this
example with very little code.
    :returns Tensor
        A new convolutional layer
    """
    strides = 2 if layer_depth % 3 == 0 else 1
    conv_layer = tf.layers.conv2d(prev_layer, layer_depth*4, 3, strides,
'same', activation=tf.nn.relu)
    return conv_layer
```

Run the following cell, along with the earlier cells (to load the dataset and define the necessary functions).

This cell builds the network **without** batch normalization, then trains it on the MNIST dataset. It displays loss and accuracy data periodically while training.

In [4]:

```
def train(num_batches, batch_size, learning_rate):
    # Build placeholders for the input samples and labels
    inputs = tf.placeholder(tf.float32, [None, 28, 28, 1])
    labels = tf.placeholder(tf.float32, [None, 10])

    # Feed the inputs into a series of 20 convolutional layers
    layer = inputs
    for layer_i in range(1, 20):
        layer = conv_layer(layer, layer_i)

    # Flatten the output from the convolutional layers
    orig_shape = layer.get_shape().as_list()
    layer = tf.reshape(layer, shape=[-1, orig_shape[1] * orig_shape[2] * orig_shape[3]])

    # Add one fully connected layer
    layer = fully_connected(layer, 100)

    # Create the output layer with 1 node for each
    logits = tf.layers.dense(layer, 10)
```

```
    # Define
    model_loss =
tf.reduce_mean(tf.nn.sigmoid_cross_entropy_with_logits(logits=logits,
labels=labels))

    train_opt = tf.train.AdamOptimizer(learning_rate).minimize(model_loss)

    correct_prediction = tf.equal(tf.argmax(logits,1), tf.argmax(labels,1))
    accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))

    # Train and test the network
    with tf.Session() as sess:
        sess.run(tf.global_variables_initializer())
        for batch_i in range(num_batches):
            batch_xs, batch_ys = mnist.train.next_batch(batch_size)

            # train this batch
            sess.run(train_opt, {inputs: batch_xs,
                                 labels: batch_ys})

            # Periodically check the validation or training loss and accuracy
            if batch_i % 100 == 0:
                loss, acc = sess.run([model_loss, accuracy], {inputs: mnist.validation.images,
                                                              labels: mnist.validation.labels})
                print('Batch: {:>2}: Validation loss: {:>3.5f}, Validation
accuracy: {:>3.5f}'.format(batch_i, loss, acc))
            elif batch_i % 25 == 0:
                loss, acc = sess.run([model_loss, accuracy], {inputs: batch_xs,
labels: batch_ys})
                print('Batch: {:>2}: Training loss: {:>3.5f}, Training
accuracy: {:>3.5f}'.format(batch_i, loss, acc))

        # At the end, score the final accuracy for both the validation and test sets
        acc = sess.run(accuracy, {inputs: mnist.validation.images,
                                  labels: mnist.validation.labels})
        print('Final validation accuracy: {:>3.5f}'.format(acc))
        acc = sess.run(accuracy, {inputs: mnist.test.images,
                                  labels: mnist.test.labels})
        print('Final test accuracy: {:>3.5f}'.format(acc))

        # Score the first 100 test images individually, just to make sure batch normalization really worked
        correct = 0
        for i in range(100):
            correct += sess.run(accuracy,feed_dict={inputs: [mnist.test.images[i]],
                                                    labels: [mnist.test.labels[i]]})

        print("Accuracy on 100 samples:", correct/100)

num_batches = 800
batch_size = 64
learning_rate = 0.002
```

```
tf.reset_default_graph()
with tf.Graph().as_default():
    train(num_batches, batch_size, learning_rate)
```

```
Batch:  0: Validation loss: 0.69066, Validation accuracy: 0.09580
Batch: 25: Training loss: 0.33632, Training accuracy: 0.07812
Batch: 50: Training loss: 0.32540, Training accuracy: 0.10938
.....
Batch: 775: Training loss: 0.32533, Training accuracy: 0.12500
Final validation accuracy: 0.11420
Final test accuracy: 0.12190
Accuracy on 100 samples: 0.13
```

With this many layers, it's going to take a lot of iterations for this network to learn. By the time you're done training these 800 batches, your final test and validation accuracies probably won't be much better than 10%. (It will be different each time, but will most likely be less than 15%.)

Using batch normalization, you'll be able to train this same network to over 90% in that same number of batches.

Add batch normalization

To add batch normalization to the layers created by `fully_connected`, we did the following:

1. Added the `is_training` parameter to the function signature so we can pass that information to the batch normalization layer.
2. Removed the bias and activation function from the `dense` layer.
3. Used `tf.layers.batch_normalization` to normalize the layer's output. Notice we pass `is_training` to this layer to ensure the network updates its population statistics appropriately.
4. Passed the normalized values into a ReLU activation function.

In [5]:

```
def fully_connected(prev_layer, num_units, is_training):
    """
    Create a fully connectd layer with the given layer as input and the given number of neurons.

    :param prev_layer: Tensor
        The Tensor that acts as input into this layer
    :param num_units: int
        The size of the layer. That is, the number of units, nodes, or neurons.
    :param is_training: bool or Tensor
        Indicates whether or not the network is currently training, which tells the batch
normalization
        layer whether or not it should update or use its population statistics.
    :returns Tensor
        A new fully connected layer
    """
    layer = tf.layers.dense(prev_layer, num_units, use_bias=False,
activation=None)
    layer = tf.layers.batch_normalization(layer, training=is_training)
    layer = tf.nn.relu(layer)
    return layer
```

To add batch normalization to the layers created by `conv_layer`, we did the following:

1. Added the `is_training` parameter to the function signature so we can pass that information to the batch normalization layer.
2. Removed the bias and activation function from the `conv2d` layer.
3. Used `tf.layers.batch_normalization` to normalize the convolutional layer's output. Notice we pass `is_training`to this layer to ensure the network updates its population statistics appropriately.
4. Passed the normalized values into a ReLU activation function.

If you compare this function to `fully_connected`, you'll see that – when using `tf.layers` – there really isn't any difference between normalizing a fully connected layer and a convolutional layer. However, if you look at the second example in this notebook, where we restrict ourselves to the `tf.nn` package, you'll see a small difference.

In [6]:

```
def conv_layer(prev_layer, layer_depth, is_training):
    """
    Create a convolutional layer with the given layer as input.

    :param prev_layer: Tensor
        The Tensor that acts as input into this layer
    :param layer_depth: int
        We'll set the strides and number of feature maps based on the layer's
depth in the network.
        This is *not* a good way to make a CNN, but it helps us create this
example with very little code.
    :param is_training: bool or Tensor
        Indicates whether or not the network is currently training, which tells
the batch normalization
        layer whether or not it should update or use its population statistics.
    :returns Tensor
        A new convolutional layer
    """
    strides = 2 if layer_depth % 3 == 0 else 1
    conv_layer = tf.layers.conv2d(prev_layer, layer_depth*4, 3, strides,
'same', use_bias=False, activation=None)
    conv_layer = tf.layers.batch_normalization(conv_layer,
training=is_training)
    conv_layer = tf.nn.relu(conv_layer)

    return conv_layer
```

Batch normalization is still a new enough idea that researchers are still discovering how best to use it. In general, people seem to agree to remove the layer's bias (because the batch normalization already has terms for scaling and shifting) and add batch normalization *before* the layer's non-linear activation function. However, for some networks it will work well in other ways, too.

Just to demonstrate this point, the following three versions of `conv_layer` show other ways to implement batch normalization. If you try running with any of these versions of the function, they should all still work fine (although some versions may still work better than others).

Alternate solution that uses bias in the convolutional layer but still adds batch normalization before the ReLU activation function.

In [ ]:

```
def conv_layer(prev_layer, layer_num, is_training):
    strides = 2 if layer_num % 3 == 0 else 1
    conv_layer = tf.layers.conv2d(prev_layer, layer_num*4, 3, strides, 'same',
use_bias=True, activation=None)
    conv_layer = tf.layers.batch_normalization(conv_layer,
training=is_training)
    conv_layer = tf.nn.relu(conv_layer)
    return conv_layer
```

Alternate solution that uses a bias and ReLU activation function *before* batch normalization.

In [ ]:

```
def conv_layer(prev_layer, layer_num, is_training):
    strides = 2 if layer_num % 3 == 0 else 1
    conv_layer = tf.layers.conv2d(prev_layer, layer_num*4, 3, strides, 'same',
use_bias=True, activation=tf.nn.relu)
    conv_layer = tf.layers.batch_normalization(conv_layer,
training=is_training)
    return conv_layer
```

Alternate solution that uses a ReLU activation function *before* normalization, but no bias.

In [ ]:

```
def conv_layer(prev_layer, layer_num, is_training):
    strides = 2 if layer_num % 3 == 0 else 1
    conv_layer = tf.layers.conv2d(prev_layer, layer_num*4, 3, strides, 'same',
use_bias=False, activation=tf.nn.relu)
    conv_layer = tf.layers.batch_normalization(conv_layer,
training=is_training)
    return conv_layer
```

To modify `train`, we did the following:

1. Added `is_training`, a placeholder to store a boolean value indicating whether or not the network is training.
2. Passed `is_training` to the `conv_layer` and `fully_connected` functions.
3. Each time we call `run` on the session, we added to `feed_dict` the appropriate value for `is_training`.
4. Moved the creation of `train_opt` inside a `with tf.control_dependencies...` statement. This is necessary to get the normalization layers created with `tf.layers.batch_normalization` to update their population statistics, which we need when performing inference.

In [7]:

```
def train(num_batches, batch_size, learning_rate):
    # Build placeholders for the input samples and labels
    inputs = tf.placeholder(tf.float32, [None, 28, 28, 1])
    labels = tf.placeholder(tf.float32, [None, 10])

    # Add placeholder to indicate whether or not we're training the model
    is_training = tf.placeholder(tf.bool)

    # Feed the inputs into a series of 20 convolutional layers
```

```
    layer = inputs
    for layer_i in range(1, 20):
        layer = conv_layer(layer, layer_i, is_training)

    # Flatten the output from the convolutional layers
    orig_shape = layer.get_shape().as_list()
    layer = tf.reshape(layer, shape=[-1, orig_shape[1] * orig_shape[2] * orig_shape[3]])

    # Add one fully connected layer
    layer = fully_connected(layer, 100, is_training)

    # Create the output layer with 1 node for each
    logits = tf.layers.dense(layer, 10)

    # Define loss and training operations
    model_loss =
tf.reduce_mean(tf.nn.sigmoid_cross_entropy_with_logits(logits=logits,
labels=labels))

    # Tell TensorFlow to update the population statistics while training
    with tf.control_dependencies(tf.get_collection(tf.GraphKeys.UPDATE_OPS)):
        train_opt = tf.train.AdamOptimizer(learning_rate).minimize(model_loss)

    # Create operations to test accuracy
    correct_prediction = tf.equal(tf.argmax(logits,1), tf.argmax(labels,1))
    accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))

    # Train and test the network
    with tf.Session() as sess:
        sess.run(tf.global_variables_initializer())
        for batch_i in range(num_batches):
            batch_xs, batch_ys = mnist.train.next_batch(batch_size)

            # train this batch
            sess.run(train_opt, {inputs: batch_xs, labels: batch_ys,
is_training: True})

            # Periodically check the validation or training loss and accuracy
            if batch_i % 100 == 0:
                loss, acc = sess.run([model_loss, accuracy], {inputs: mnist.validation.images,
                                                              labels: mnist.validation.labels,
                                                              is_training: False})
                print('Batch: {:>2}: Validation loss: {:>3.5f}, Validation
accuracy: {:>3.5f}'.format(batch_i, loss, acc))
            elif batch_i % 25 == 0:
                loss, acc = sess.run([model_loss, accuracy], {inputs: batch_xs,
labels: batch_ys, is_training: False})
                print('Batch: {:>2}: Training loss: {:>3.5f}, Training
accuracy: {:>3.5f}'.format(batch_i, loss, acc))
```

```
        # At the end, score the final accuracy for both the validation and test sets
        acc = sess.run(accuracy, {inputs: mnist.validation.images,
                                  labels: mnist.validation.labels,
                                  is_training: False})
        print('Final validation accuracy: {:>3.5f}'.format(acc))
        acc = sess.run(accuracy, {inputs: mnist.test.images,
                                  labels: mnist.test.labels,
                                  is_training: False})
        print('Final test accuracy: {:>3.5f}'.format(acc))

        # Score the first 100 test images individually, just to make sure batch normalization really worked
        correct = 0
        for i in range(100):
            correct += sess.run(accuracy,feed_dict={inputs: [mnist.test.images[i]],
                                                    labels: [mnist.test.labels[i]],
                                                    is_training: False})

        print("Accuracy on 100 samples:", correct/100)

num_batches = 800
batch_size = 64
learning_rate = 0.002

tf.reset_default_graph()
with tf.Graph().as_default():
    train(num_batches, batch_size, learning_rate)
```

```
Batch:  0: Validation loss: 0.69135, Validation accuracy: 0.09860
Batch: 25: Training loss: 0.58727, Training accuracy: 0.10938
Batch: 50: Training loss: 0.47633, Training accuracy: 0.09375
Batch: 75: Training loss: 0.40283, Training accuracy: 0.17188


Batch: 775: Training loss: 0.02039, Training accuracy: 0.96875
Final validation accuracy: 0.97460
Final test accuracy: 0.97450
Accuracy on 100 samples: 0.98
```

With batch normalization, we now get excellent performance. In fact, validation accuracy is almost 94% after only 500 batches. Notice also the last line of the output: `Accuracy on 100 samples`. If this value is low while everything else looks good, that means you did not implement batch normalization correctly. Specifically, it means you either did not calculate the population mean and variance while training, or you are not using those values during inference.

Batch Normalization using `tf.nn.batch_normalization`

Most of the time you will be able to use higher level functions exclusively, but sometimes you may want to work at a lower level. For example, if you ever want to implement a new feature – something new enough that TensorFlow does not already include a high-level implementation of it, like batch normalization in an LSTM – then you may need to know these sorts of things.

This version of the network uses `tf.nn` for almost everything, and expects you to implement batch normalization using `tf.nn.batch_normalization`.

This implementation of `fully_connected` is much more involved than the one that uses `tf.layers`. However, if you went through the `Batch_Normalization_Lesson` notebook, things should look pretty familiar. To add batch normalization, we did the following:

1. Added the `is_training` parameter to the function signature so we can pass that information to the batch normalization layer.
2. Removed the bias and activation function from the `dense` layer.
3. Added `gamma`, `beta`, `pop_mean`, and `pop_variance` variables.
4. Used `tf.cond` to make handle training and inference differently.
5. When training, we use `tf.nn.moments` to calculate the batch mean and variance. Then we update the population statistics and use `tf.nn.batch_normalization` to normalize the layer's output using the batch statistics. Notice the `with tf.control_dependencies...` statement - this is required to force TensorFlow to run the operations that update the population statistics.
6. During inference (i.e. when not training), we use `tf.nn.batch_normalization` to normalize the layer's output using the population statistics we calculated during training.
7. Passed the normalized values into a ReLU activation function.

If any of thise code is unclear, it is almost identical to what we showed in the `fully_connected` function in the `Batch_Normalization_Lesson` notebook. Please see that for extensive comments.

In [1]:

```
def fully_connected(prev_layer, num_units, is_training):
    """
    Create a fully connectd layer with the given layer as input and the given number of neurons.
    :param prev_layer: Tensor
        The Tensor that acts as input into this layer
    :param num_units: int
        The size of the layer. That is, the number of units, nodes, or neurons.
    :param is_training: bool or Tensor
        Indicates whether or not the network is currently training, which tells the batch normalization
        layer whether or not it should update or use its population statistics.
    :returns Tensor
        A new fully connected layer
    """
    layer = tf.layers.dense(prev_layer, num_units, use_bias=False, activation=None)

    gamma = tf.Variable(tf.ones([num_units]))
    beta = tf.Variable(tf.zeros([num_units]))

    pop_mean = tf.Variable(tf.zeros([num_units]), trainable=False)
    pop_variance = tf.Variable(tf.ones([num_units]), trainable=False)

    epsilon = 1e-3

    def batch_norm_training():
        batch_mean, batch_variance = tf.nn.moments(layer, [0])

        decay = 0.99
        train_mean = tf.assign(pop_mean, pop_mean * decay + batch_mean * (1 - decay))
        train_variance = tf.assign(pop_variance, pop_variance * decay + batch_variance * (1 - decay))
```

```
        with tf.control_dependencies([train_mean, train_variance]):
            return tf.nn.batch_normalization(layer, batch_mean, batch_variance,
beta, gamma, epsilon)

    def batch_norm_inference():
        return tf.nn.batch_normalization(layer, pop_mean, pop_variance, beta,
gamma, epsilon)

    batch_normalized_output = tf.cond(is_training, batch_norm_training,
batch_norm_inference)
    return tf.nn.relu(batch_normalized_output)
```

The changes we made to `conv_layer` are *almost* exactly the same as the ones we made to `fully_connected`. However, there is an important difference. Convolutional layers have multiple feature maps, and each feature map uses shared weights. So we need to make sure we calculate our batch and population statistics **per feature map** instead of per node in the layer.

To accomplish this, we do **the same things** that we did in `fully_connected`, with two exceptions:

1. The sizes of `gamma`, `beta`, `pop_mean` and `pop_variance` are set to the number of feature maps (output channels) instead of the number of output nodes.
2. We change the parameters we pass to `tf.nn.moments` to make sure it calculates the mean and variance for the correct dimensions.

In [9]:

```
def conv_layer(prev_layer, layer_depth, is_training):
    """
    Create a convolutional layer with the given layer as input.

    :param prev_layer: Tensor
        The Tensor that acts as input into this layer
    :param layer_depth: int
        We'll set the strides and number of feature maps based on the layer's
depth in the network.
        This is *not* a good way to make a CNN, but it helps us create this
example with very little code.
    :param is_training: bool or Tensor
        Indicates whether or not the network is currently training, which tells
the batch normalization
        layer whether or not it should update or use its population statistics.
    :returns Tensor
        A new convolutional layer
    """
    strides = 2 if layer_depth % 3 == 0 else 1

    in_channels = prev_layer.get_shape().as_list()[3]
    out_channels = layer_depth*4

    weights = tf.Variable(
        tf.truncated_normal([3, 3, in_channels, out_channels], stddev=0.05))
```

```
    layer = tf.nn.conv2d(prev_layer, weights, strides=[1,strides, strides, 1],
padding='SAME')

    gamma = tf.Variable(tf.ones([out_channels]))
    beta = tf.Variable(tf.zeros([out_channels]))

    pop_mean = tf.Variable(tf.zeros([out_channels]), trainable=False)
    pop_variance = tf.Variable(tf.ones([out_channels]), trainable=False)

    epsilon = 1e-3

    def batch_norm_training():
        # Important to use the correct dimensions here to ensure the mean and
variance are calculated
        # per feature map instead of for the entire layer
        batch_mean, batch_variance = tf.nn.moments(layer, [0,1,2],
keep_dims=False)

        decay = 0.99
        train_mean = tf.assign(pop_mean, pop_mean * decay + batch_mean * (1 -
decay))
        train_variance = tf.assign(pop_variance, pop_variance * decay +
batch_variance * (1 - decay))

        with tf.control_dependencies([train_mean, train_variance]):
            return tf.nn.batch_normalization(layer, batch_mean, batch_variance,
beta, gamma, epsilon)

    def batch_norm_inference():
        return tf.nn.batch_normalization(layer, pop_mean, pop_variance, beta,
gamma, epsilon)

    batch_normalized_output = tf.cond(is_training, batch_norm_training,
batch_norm_inference)
    return tf.nn.relu(batch_normalized_output)
```

To modify `train`, we did the following:

1. Added `is_training`, a placeholder to store a boolean value indicating whether or not the network is training.
2. Each time we call `run` on the session, we added to `feed_dict` the appropriate value for `is_training`.
3. We did **not** need to add the `with tf.control_dependencies...` statement that we added in the network that used `tf.layers.batch_normalization` because we handled updating the population statistics ourselves in `conv_layer`and `fully_connected`.

In [10]:

```
def train(num_batches, batch_size, learning_rate):
    # Build placeholders for the input samples and labels
    inputs = tf.placeholder(tf.float32, [None, 28, 28, 1])
    labels = tf.placeholder(tf.float32, [None, 10])
```

```
    # Add placeholder to indicate whether or not we're training the model
    is_training = tf.placeholder(tf.bool)

    # Feed the inputs into a series of 20 convolutional layers
    layer = inputs
    for layer_i in range(1, 20):
        layer = conv_layer(layer, layer_i, is_training)

    # Flatten the output from the convolutional layers
    orig_shape = layer.get_shape().as_list()
    layer = tf.reshape(layer, shape=[-1, orig_shape[1] * orig_shape[2] *
orig_shape[3]])

    # Add one fully connected layer
    layer = fully_connected(layer, 100, is_training)

    # Create the output layer with 1 node for each
    logits = tf.layers.dense(layer, 10)

    # Define loss and training operations
    model_loss =
tf.reduce_mean(tf.nn.sigmoid_cross_entropy_with_logits(logits=logits,
labels=labels))
    train_opt = tf.train.AdamOptimizer(learning_rate).minimize(model_loss)

    # Create operations to test accuracy
    correct_prediction = tf.equal(tf.argmax(logits,1), tf.argmax(labels,1))
    accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))

    # Train and test the network
    with tf.Session() as sess:
        sess.run(tf.global_variables_initializer())
        for batch_i in range(num_batches):
            batch_xs, batch_ys = mnist.train.next_batch(batch_size)

            # train this batch
            sess.run(train_opt, {inputs: batch_xs, labels: batch_ys,
is_training: True})

            # Periodically check the validation or training loss and accuracy
            if batch_i % 100 == 0:
                loss, acc = sess.run([model_loss, accuracy], {inputs:
mnist.validation.images,
                                                              labels:
mnist.validation.labels,
                                                              is_training:
False})
                print('Batch: {:>2}: Validation loss: {:>3.5f}, Validation
accuracy: {:>3.5f}'.format(batch_i, loss, acc))
            elif batch_i % 25 == 0:
```

```
                loss, acc = sess.run([model_loss, accuracy], {inputs: batch_xs,
labels: batch_ys, is_training: False})
                print('Batch: {:>2}: Training loss: {:>3.5f}, Training
accuracy: {:>3.5f}'.format(batch_i, loss, acc))

        # At the end, score the final accuracy for both the validation and test
sets
        acc = sess.run(accuracy, {inputs: mnist.validation.images,
                                  labels: mnist.validation.labels,
                                  is_training: False})
        print('Final validation accuracy: {:>3.5f}'.format(acc))
        acc = sess.run(accuracy, {inputs: mnist.test.images,
                                  labels: mnist.test.labels,
                                  is_training: False})
        print('Final test accuracy: {:>3.5f}'.format(acc))

        # Score the first 100 test images individually, just to make sure batch
normalization really worked
        correct = 0
        for i in range(100):
            correct += sess.run(accuracy,feed_dict={inputs:
[mnist.test.images[i]],
                                                    labels:
[mnist.test.labels[i]],
                                                    is_training: False})

        print("Accuracy on 100 samples:", correct/100)


num_batches = 800
batch_size = 64
learning_rate = 0.002

tf.reset_default_graph()
with tf.Graph().as_default():
    train(num_batches, batch_size, learning_rate)
```

```
Batch:  0: Validation loss: 0.68918, Validation accuracy: 0.09860
Batch: 25: Training loss: 0.53546, Training accuracy: 0.07812
Batch: 50: Training loss: 0.41245, Training accuracy: 0.09375
Batch: 75: Training loss: 0.36121, Training accuracy: 0.07812
Batch: 100: Validation loss: 0.33833, Validation accuracy: 0.09900
……
Batch: 775: Training loss: 0.01844, Training accuracy: 0.96875
Final validation accuracy: 0.95780
Final test accuracy: 0.96050
Accuracy on 100 samples: 0.98
```

Once again, the model with batch normalization quickly reaches a high accuracy. But in our run, notice that it doesn't seem to learn anything for the first 250 batches, then the accuracy starts to climb. That just goes to show - even with batch normalization, it's important to give your network a bit of time to learn before you decide it isn't working.