Deep Learning

Assignment 4

Previously in `2_fullyconnected.ipynb` and `3_regularization.ipynb`, we trained fully connected networks to classify notMNIST (http://yaroslavvb.blogspot.com/2011/09/notmnist-dataset.html) characters.

The goal of this assignment is make the neural network convolutional.

In [1]:
```
# These are all the modules we'll be using later. Make sure you can import them
# before proceeding further.
from __future__ import print_function
import numpy as np
import tensorflow as tf
from six.moves import cPickle as pickle
from six.moves import range
```

In [2]:
```
pickle_file = 'notMNIST.pickle'

with open(pickle_file, 'rb') as f:
  save = pickle.load(f)
  train_dataset = save['train_dataset']
  train_labels = save['train_labels']
  valid_dataset = save['valid_dataset']
  valid_labels = save['valid_labels']
  test_dataset = save['test_dataset']
  test_labels = save['test_labels']
  del save  # hint to help gc free up memory
  print('Training set', train_dataset.shape, train_labels.shape)
  print('Validation set', valid_dataset.shape, valid_labels.shape)
  print('Test set', test_dataset.shape, test_labels.shape)
```

```
Training set (200000, 28, 28) (200000,)
Validation set (10000, 28, 28) (10000,)
Test set (10000, 28, 28) (10000,)
```

Reformat into a TensorFlow-friendly shape:

- convolutions need the image data formatted as a cube (width by height by #channels)
- labels as float 1-hot encodings.

In [3]:
```
image_size = 28
num_labels = 10
num_channels = 1 # grayscale

import numpy as np

def reformat(dataset, labels):
  dataset = dataset.reshape(
    (-1, image_size, image_size, num_channels)).astype(np.float32)
  labels = (np.arange(num_labels) == labels[:,None]).astype(np.float32)
  return dataset, labels
train_dataset, train_labels = reformat(train_dataset, train_labels)
valid_dataset, valid_labels = reformat(valid_dataset, valid_labels)
test_dataset, test_labels = reformat(test_dataset, test_labels)
print('Training set', train_dataset.shape, train_labels.shape)
print('Validation set', valid_dataset.shape, valid_labels.shape)
print('Test set', test_dataset.shape, test_labels.shape)
```

```
Training set (200000, 28, 28, 1) (200000, 10)
Validation set (10000, 28, 28, 1) (10000, 10)
Test set (10000, 28, 28, 1) (10000, 10)
```

In [4]:
```
def accuracy(predictions, labels):
  return (100.0 * np.sum(np.argmax(predictions, 1) == np.argmax(labels, 1))
          / predictions.shape[0])
```

Let's build a small network with two convolutional layers, followed by one fully connected layer. Convolutional networks are more expensive computationally, so we'll limit its depth and number of fully connected nodes.

In [5]:
```
batch_size = 16
patch_size = 5
depth = 16
num_hidden = 64

graph = tf.Graph()

with graph.as_default():

  # Input data.
  tf_train_dataset = tf.placeholder(
    tf.float32, shape=(batch_size, image_size, image_size, num_channels))
  tf_train_labels = tf.placeholder(tf.float32, shape=(batch_size, num_labels))
  tf_valid_dataset = tf.constant(valid_dataset)
  tf_test_dataset = tf.constant(test_dataset)

  # Variables.
  layer1_weights = tf.Variable(tf.truncated_normal(
      [patch_size, patch_size, num_channels, depth], stddev=0.1))
  layer1_biases = tf.Variable(tf.zeros([depth]))
  layer2_weights = tf.Variable(tf.truncated_normal(
      [patch_size, patch_size, depth, depth], stddev=0.1))
  layer2_biases = tf.Variable(tf.constant(1.0, shape=[depth]))
  layer3_weights = tf.Variable(tf.truncated_normal(
      [image_size // 4 * image_size // 4 * depth, num_hidden], stddev=0.1))
  layer3_biases = tf.Variable(tf.constant(1.0, shape=[num_hidden]))
  layer4_weights = tf.Variable(tf.truncated_normal(
      [num_hidden, num_labels], stddev=0.1))
  layer4_biases = tf.Variable(tf.constant(1.0, shape=[num_labels]))

  print(layer1_weights.get_shape())
  print(layer2_weights.get_shape())
  print(layer3_weights.get_shape())
  print(layer4_weights.get_shape())

  # Model.
  def model(data):
    conv = tf.nn.conv2d(data, layer1_weights, [1, 2, 2, 1], padding='SAME')
    # print("model: conv shape: %s" % conv.get_shape())
    hidden = tf.nn.relu(conv + layer1_biases)
    # print("model: hidden shape: %s" % hidden.get_shape())
```

```
    conv = tf.nn.conv2d(hidden, layer2_weights, [1, 2, 2, 1], padding='SAME')
    # print("model: conv shape: %s" % conv.get_shape())
    hidden = tf.nn.relu(conv + layer2_biases)
    # print("model: hidden shape: %s" % hidden.get_shape())
    shape = hidden.get_shape().as_list()
    reshape = tf.reshape(hidden, [shape[0], shape[1] * shape[2] * shape[3]])
    hidden = tf.nn.relu(tf.matmul(reshape, layer3_weights) + layer3_biases)
    return tf.matmul(hidden, layer4_weights) + layer4_biases

  # Training computation.
  logits = model(tf_train_dataset)
  loss = tf.reduce_mean(
    tf.nn.softmax_cross_entropy_with_logits(labels=tf_train_labels, logits=logits))

  # Optimizer.
  optimizer = tf.train.GradientDescentOptimizer(0.05).minimize(loss)

  # Predictions for the training, validation, and test data.
  train_prediction = tf.nn.softmax(logits)
  valid_prediction = tf.nn.softmax(model(tf_valid_dataset))
  test_prediction = tf.nn.softmax(model(tf_test_dataset))
```

```
(5, 5, 1, 16)
(5, 5, 16, 16)
(784, 64)
(64, 10)
```

In [6]:
```
num_steps = 1001

with tf.Session(graph=graph) as session:
  tf.global_variables_initializer().run()
  print('Initialized')
  for step in range(num_steps):
    offset = (step * batch_size) % (train_labels.shape[0] - batch_size)
    batch_data = train_dataset[offset:(offset + batch_size), :, :, :]
    batch_labels = train_labels[offset:(offset + batch_size), :]
    feed_dict = {tf_train_dataset : batch_data, tf_train_labels : batch_labels}
    _, l, predictions = session.run(
      [optimizer, loss, train_prediction], feed_dict=feed_dict)
    if (step % 50 == 0):
      print('Minibatch loss at step %d: %f' % (step, l))
      print('Minibatch accuracy: %.1f%%' % accuracy(predictions, batch_labels))
      print('Validation accuracy: %.1f%%' % accuracy(
        valid_prediction.eval(), valid_labels))
  print('Test accuracy: %.1f%%' % accuracy(test_prediction.eval(), test_labels))
```

```
Initialized
Minibatch loss at step 0: 2.675679
Minibatch accuracy: 12.5%
Validation accuracy: 10.0%
Minibatch loss at step 50: 1.455389
Minibatch accuracy: 50.0%
Validation accuracy: 50.0%
Minibatch loss at step 100: 0.369873
Minibatch accuracy: 100.0%
Validation accuracy: 74.8%
Minibatch loss at step 150: 0.495924
Minibatch accuracy: 93.8%
Validation accuracy: 76.1%
Minibatch loss at step 200: 0.417204
Minibatch accuracy: 87.5%
Validation accuracy: 76.6%
Minibatch loss at step 250: 1.547781
Minibatch accuracy: 62.5%
Validation accuracy: 75.8%
Minibatch loss at step 300: 0.860488
Minibatch accuracy: 81.2%
Validation accuracy: 80.2%
```

```
Minibatch loss at step 350: 0.970358
Minibatch accuracy: 62.5%
Validation accuracy: 79.3%
Minibatch loss at step 400: 0.904927
Minibatch accuracy: 87.5%
Validation accuracy: 79.6%
Minibatch loss at step 450: 0.541103
Minibatch accuracy: 81.2%
Validation accuracy: 80.3%
Minibatch loss at step 500: 0.533026
Minibatch accuracy: 93.8%
Validation accuracy: 81.0%
Minibatch loss at step 550: 0.212027
Minibatch accuracy: 93.8%
Validation accuracy: 82.0%
Minibatch loss at step 600: 0.444889
Minibatch accuracy: 87.5%
Validation accuracy: 81.4%
Minibatch loss at step 650: 0.429304
Minibatch accuracy: 87.5%
Validation accuracy: 81.1%
Minibatch loss at step 700: 0.522365
Minibatch accuracy: 81.2%
Validation accuracy: 80.9%
Minibatch loss at step 750: 0.200323
Minibatch accuracy: 100.0%
Validation accuracy: 82.3%
Minibatch loss at step 800: 0.813128
Minibatch accuracy: 87.5%
Validation accuracy: 82.1%
Minibatch loss at step 850: 0.709090
Minibatch accuracy: 81.2%
Validation accuracy: 82.4%
Minibatch loss at step 900: 0.430374
Minibatch accuracy: 87.5%
Validation accuracy: 81.2%
Minibatch loss at step 950: 0.146930
Minibatch accuracy: 100.0%
Validation accuracy: 83.3%
Minibatch loss at step 1000: 0.265595
Minibatch accuracy: 87.5%
```

```
Validation accuracy: 82.0%
Test accuracy: 89.2%
```

Problem 1

The convolutional model above uses convolutions with stride 2 to reduce the dimensionality. Replace the strides by a max pooling operation (`nn.max_pool()`) of stride 2 and kernel size 2.

In [8]:
```
batch_size = 16
patch_size = 5
depth = 16
num_hidden = 64

beta = 0.001

graph = tf.Graph()

with graph.as_default():

  # Input data.
  tf_train_dataset = tf.placeholder(
    tf.float32, shape=(batch_size, image_size, image_size, num_channels))
  tf_train_labels = tf.placeholder(tf.float32, shape=(batch_size, num_labels))
  tf_valid_dataset = tf.constant(valid_dataset)
  tf_test_dataset = tf.constant(test_dataset)

  # Variables.
  layer1_weights = tf.Variable(tf.truncated_normal(
      [patch_size, patch_size, num_channels, depth], stddev=0.1))
  layer1_biases = tf.Variable(tf.zeros([depth]))
  layer2_weights = tf.Variable(tf.truncated_normal(
      [patch_size, patch_size, depth, depth], stddev=0.1))
  layer2_biases = tf.Variable(tf.constant(1.0, shape=[depth]))
  layer3_weights = tf.Variable(tf.truncated_normal(
      [image_size // 4 * image_size // 4 * depth, num_hidden], stddev=0.1))
  layer3_biases = tf.Variable(tf.constant(1.0, shape=[num_hidden]))
  layer4_weights = tf.Variable(tf.truncated_normal(
      [num_hidden, num_labels], stddev=0.1))
  layer4_biases = tf.Variable(tf.constant(1.0, shape=[num_labels]))

  print(layer1_weights.get_shape())
  print(layer2_weights.get_shape())
  print(layer3_weights.get_shape())
  print(layer4_weights.get_shape())

  # Model.
  def model(data):
    conv = tf.nn.conv2d(data, layer1_weights, [1, 1, 1, 1], padding='SAME')
    # print("model: conv shape: %s" % conv.get_shape())
```

```
    pool1 = tf.nn.max_pool(conv, ksize=[1, 2, 2, 1], strides=[1, 2, 2, 1], padding='SAME')
    # print("model: pool shape: %s" % pool1.get_shape())
    hidden = tf.nn.relu(pool1 + layer1_biases)
    # print("model: hidden shape: %s" % hidden.get_shape())

    conv = tf.nn.conv2d(hidden, layer2_weights, [1, 1, 1, 1], padding='SAME')
    # print("model: conv shape: %s" % conv.get_shape())
    pool2 = tf.nn.max_pool(conv, ksize=[1, 2, 2, 1], strides=[1, 2, 2, 1], padding='SAME')
    # print("model: pool shape: %s" % pool1.get_shape())
    hidden = tf.nn.relu(pool2 + layer2_biases)
    # print("model: hidden shape: %s" % hidden.get_shape())

    shape = hidden.get_shape().as_list()
    reshape = tf.reshape(hidden, [shape[0], shape[1] * shape[2] * shape[3]])
    hidden = tf.nn.relu(tf.matmul(reshape, layer3_weights) + layer3_biases)
    return tf.matmul(hidden, layer4_weights) + layer4_biases

  # Training computation.
  logits = model(tf_train_dataset)
  loss = tf.reduce_mean(
    tf.nn.softmax_cross_entropy_with_logits(labels=tf_train_labels, logits=logits))

  # Optimizer.
  optimizer = tf.train.GradientDescentOptimizer(0.05).minimize(loss)

  # Predictions for the training, validation, and test data.
  train_prediction = tf.nn.softmax(logits)
  valid_prediction = tf.nn.softmax(model(tf_valid_dataset))
  test_prediction = tf.nn.softmax(model(tf_test_dataset))
```

```
(5, 5, 1, 16)
(5, 5, 16, 16)
(784, 64)
(64, 10)
```

In [9]:
```
num_steps = 1001


with tf.Session(graph=graph) as session:
  tf.global_variables_initializer().run()
  print('Initialized')
  for step in range(num_steps):
    offset = (step * batch_size) % (train_labels.shape[0] - batch_size)
    batch_data = train_dataset[offset:(offset + batch_size), :, :, :]
    batch_labels = train_labels[offset:(offset + batch_size), :]
    feed_dict = {tf_train_dataset : batch_data, tf_train_labels : batch_labels}
    _, l, predictions = session.run(
      [optimizer, loss, train_prediction], feed_dict=feed_dict)
    if (step % 50 == 0):
      print('Minibatch loss at step %d: %f' % (step, l))
      print('Minibatch accuracy: %.1f%%' % accuracy(predictions, batch_labels))
      print('Validation accuracy: %.1f%%' % accuracy(
        valid_prediction.eval(), valid_labels))
  print('Test accuracy: %.1f%%' % accuracy(test_prediction.eval(), test_labels))
```

```
Initialized
Minibatch loss at step 0: 2.769767
Minibatch accuracy: 12.5%
Validation accuracy: 10.0%
Minibatch loss at step 50: 1.557950
Minibatch accuracy: 31.2%
Validation accuracy: 50.6%
Minibatch loss at step 100: 0.632284
Minibatch accuracy: 81.2%
Validation accuracy: 72.7%
Minibatch loss at step 150: 0.465463
Minibatch accuracy: 93.8%
Validation accuracy: 76.6%
Minibatch loss at step 200: 0.394912
Minibatch accuracy: 93.8%
Validation accuracy: 77.7%
Minibatch loss at step 250: 1.370968
Minibatch accuracy: 62.5%
Validation accuracy: 76.7%
Minibatch loss at step 300: 0.739579
Minibatch accuracy: 81.2%
```

```
Validation accuracy: 79.8%
Minibatch loss at step 350: 0.864967
Minibatch accuracy: 68.8%
Validation accuracy: 78.8%
Minibatch loss at step 400: 1.093237
Minibatch accuracy: 75.0%
Validation accuracy: 78.7%
Minibatch loss at step 450: 0.614690
Minibatch accuracy: 81.2%
Validation accuracy: 80.4%
Minibatch loss at step 500: 0.382195
Minibatch accuracy: 87.5%
Validation accuracy: 81.2%
Minibatch loss at step 550: 0.194540
Minibatch accuracy: 100.0%
Validation accuracy: 82.2%
Minibatch loss at step 600: 0.495355
Minibatch accuracy: 81.2%
Validation accuracy: 81.6%
Minibatch loss at step 650: 0.460583
Minibatch accuracy: 87.5%
Validation accuracy: 82.2%
Minibatch loss at step 700: 0.581673
Minibatch accuracy: 75.0%
Validation accuracy: 81.9%
Minibatch loss at step 750: 0.219646
Minibatch accuracy: 93.8%
Validation accuracy: 82.7%
Minibatch loss at step 800: 0.698183
Minibatch accuracy: 87.5%
Validation accuracy: 82.4%
Minibatch loss at step 850: 0.661875
Minibatch accuracy: 87.5%
Validation accuracy: 82.5%
Minibatch loss at step 900: 0.350851
Minibatch accuracy: 87.5%
Validation accuracy: 81.8%
Minibatch loss at step 950: 0.106321
Minibatch accuracy: 100.0%
Validation accuracy: 83.8%
Minibatch loss at step 1000: 0.217801
Minibatch accuracy: 93.8%
```

```
Validation accuracy: 83.2%
Test accuracy: 90.3%
```

In [10]:

```
# at first, it failed because of memory limitation issue. Migrate it to another machine
#
```

Problem 2

Try to get the best performance you can using a convolutional net. Look for example at the classic LeNet5 (http://yann.lecun.com/exdb/lenet/) architecture, adding Dropout, and/or adding learning rate decay.

In [24]:
```
batch_size = 16
patch_size = 5
depth = 16
num_hidden = 64

beta = 0.001

graph = tf.Graph()

with graph.as_default():

  # Input data.
  tf_train_dataset = tf.placeholder(
    tf.float32, shape=(batch_size, image_size, image_size, num_channels))
  tf_train_labels = tf.placeholder(tf.float32, shape=(batch_size, num_labels))
  tf_valid_dataset = tf.constant(valid_dataset)
  tf_test_dataset = tf.constant(test_dataset)

  # Variables.
  layer1_weights = tf.Variable(tf.truncated_normal(
      [patch_size, patch_size, num_channels, depth], stddev=0.1))
  layer1_biases = tf.Variable(tf.zeros([depth]))
  layer2_weights = tf.Variable(tf.truncated_normal(
      [patch_size, patch_size, depth, depth], stddev=0.1))
  layer2_biases = tf.Variable(tf.constant(1.0, shape=[depth]))
  layer3_weights = tf.Variable(tf.truncated_normal(
      [image_size // 4 * image_size // 4 * depth, num_hidden], stddev=0.1))
  layer3_biases = tf.Variable(tf.constant(1.0, shape=[num_hidden]))
  layer4_weights = tf.Variable(tf.truncated_normal(
      [num_hidden, num_labels], stddev=0.1))
  layer4_biases = tf.Variable(tf.constant(1.0, shape=[num_labels]))

  print(layer1_weights.get_shape())
  print(layer2_weights.get_shape())
  print(layer3_weights.get_shape())
  print(layer4_weights.get_shape())

  keep_prob = tf.placeholder("float")

  # Model.
  def model(data, keep_prob=None):
```

```
    conv = tf.nn.conv2d(data, layer1_weights, [1, 1, 1, 1], padding='SAME')
    # print("model: conv shape: %s" % conv.get_shape())
    pool1 = tf.nn.max_pool(conv, ksize=[1, 2, 2, 1], strides=[1, 2, 2, 1], padding='SAME')
    # print("model: pool shape: %s" % pool1.get_shape())
    hidden = tf.nn.relu(pool1 + layer1_biases)
    # print("model: hidden shape: %s" % hidden.get_shape())

    conv = tf.nn.conv2d(hidden, layer2_weights, [1, 1, 1, 1], padding='SAME')
    # print("model: conv shape: %s" % conv.get_shape())
    pool2 = tf.nn.max_pool(conv, ksize=[1, 2, 2, 1], strides=[1, 2, 2, 1], padding='SAME')
    # print("model: pool shape: %s" % pool1.get_shape())
    hidden = tf.nn.relu(pool2 + layer2_biases)
    # print("model: hidden shape: %s" % hidden.get_shape())

    shape = hidden.get_shape().as_list()
    reshape = tf.reshape(hidden, [shape[0], shape[1] * shape[2] * shape[3]])
    hidden = tf.nn.relu(tf.matmul(reshape, layer3_weights) + layer3_biases)
    if keep_prob is not None:
        drop = tf.nn.dropout(hidden, keep_prob)
        return tf.matmul(drop, layer4_weights) + layer4_biases
    else:
        return tf.matmul(hidden, layer4_weights) + layer4_biases

  # Training computation.
  logits = model(tf_train_dataset, keep_prob)
  loss = tf.reduce_mean(
    tf.nn.softmax_cross_entropy_with_logits(labels=tf_train_labels, logits=logits))

  # Optimizer.
  optimizer = tf.train.GradientDescentOptimizer(0.05).minimize(loss)

  # Predictions for the training, validation, and test data.
  train_prediction = tf.nn.softmax(logits)
  valid_prediction = tf.nn.softmax(model(tf_valid_dataset))
  test_prediction = tf.nn.softmax(model(tf_test_dataset))
```

```
(5, 5, 1, 16)
(5, 5, 16, 16)
(784, 64)
(64, 10)
```

In [25]:
```
num_steps = 1001


with tf.Session(graph=graph) as session:
  tf.global_variables_initializer().run()
  print('Initialized')
  for step in range(num_steps):
    offset = (step * batch_size) % (train_labels.shape[0] - batch_size)
    batch_data = train_dataset[offset:(offset + batch_size), :, :, :]
    batch_labels = train_labels[offset:(offset + batch_size), :]
    feed_dict = {tf_train_dataset : batch_data, tf_train_labels : batch_labels, keep_prob: 0.9}
    _, l, predictions = session.run(
      [optimizer, loss, train_prediction], feed_dict=feed_dict)
    if (step % 50 == 0):
      print('Minibatch loss at step %d: %f' % (step, l))
      print('Minibatch accuracy: %.1f%%' % accuracy(predictions, batch_labels))
      print('Validation accuracy: %.1f%%' % accuracy(
        valid_prediction.eval(), valid_labels))
  print('Test accuracy: %.1f%%' % accuracy(test_prediction.eval(), test_labels))
```

```
Initialized
Minibatch loss at step 0: 4.257010
Minibatch accuracy: 6.2%
Validation accuracy: 10.0%
Minibatch loss at step 50: 2.059375
Minibatch accuracy: 18.8%
Validation accuracy: 32.8%
Minibatch loss at step 100: 1.121794
Minibatch accuracy: 62.5%
Validation accuracy: 63.2%
Minibatch loss at step 150: 0.840861
Minibatch accuracy: 68.8%
Validation accuracy: 72.6%
Minibatch loss at step 200: 0.615932
Minibatch accuracy: 87.5%
Validation accuracy: 76.9%
Minibatch loss at step 250: 1.432374
Minibatch accuracy: 68.8%
Validation accuracy: 75.2%
Minibatch loss at step 300: 0.574929
Minibatch accuracy: 87.5%
```

```
Validation accuracy: 79.8%
Minibatch loss at step 350: 1.029322
Minibatch accuracy: 56.2%
Validation accuracy: 79.7%
Minibatch loss at step 400: 1.073759
Minibatch accuracy: 81.2%
Validation accuracy: 79.4%
Minibatch loss at step 450: 0.679190
Minibatch accuracy: 81.2%
Validation accuracy: 81.0%
Minibatch loss at step 500: 0.542995
Minibatch accuracy: 81.2%
Validation accuracy: 82.2%
Minibatch loss at step 550: 0.264706
Minibatch accuracy: 93.8%
Validation accuracy: 81.9%
Minibatch loss at step 600: 0.479310
Minibatch accuracy: 87.5%
Validation accuracy: 82.4%
Minibatch loss at step 650: 0.530982
Minibatch accuracy: 87.5%
Validation accuracy: 82.4%
Minibatch loss at step 700: 0.582734
Minibatch accuracy: 81.2%
Validation accuracy: 82.5%
Minibatch loss at step 750: 0.252468
Minibatch accuracy: 100.0%
Validation accuracy: 83.0%
Minibatch loss at step 800: 0.857202
Minibatch accuracy: 87.5%
Validation accuracy: 83.1%
Minibatch loss at step 850: 0.679520
Minibatch accuracy: 75.0%
Validation accuracy: 82.4%
Minibatch loss at step 900: 0.457159
Minibatch accuracy: 81.2%
Validation accuracy: 82.2%
Minibatch loss at step 950: 0.201450
Minibatch accuracy: 100.0%
Validation accuracy: 83.8%
Minibatch loss at step 1000: 0.284401
Minibatch accuracy: 87.5%
```

```
Validation accuracy: 82.8%
Test accuracy: 89.2%
```

In [26]:
```
# it seems the starter learning rate is a little high. Then at the end, it cannot get a better result.
```

In [29]:
```
batch_size = 16
patch_size = 5
depth = 16
num_hidden = 64
beta = 0.01

graph = tf.Graph()

with graph.as_default():

  # Input data.
  tf_train_dataset = tf.placeholder(
    tf.float32, shape=(batch_size, image_size, image_size, num_channels))
  tf_train_labels = tf.placeholder(tf.float32, shape=(batch_size, num_labels))
  tf_valid_dataset = tf.constant(valid_dataset)
  tf_test_dataset = tf.constant(test_dataset)

  # Variables.
  layer1_weights = tf.Variable(tf.truncated_normal(
      [patch_size, patch_size, num_channels, depth], stddev=0.1))
  layer1_biases = tf.Variable(tf.zeros([depth]))
  layer2_weights = tf.Variable(tf.truncated_normal(
      [patch_size, patch_size, depth, depth], stddev=0.1))
  layer2_biases = tf.Variable(tf.constant(1.0, shape=[depth]))
  layer3_weights = tf.Variable(tf.truncated_normal(
      [image_size // 4 * image_size // 4 * depth, num_hidden], stddev=0.1))
  layer3_biases = tf.Variable(tf.constant(1.0, shape=[num_hidden]))
  layer4_weights = tf.Variable(tf.truncated_normal(
      [num_hidden, num_labels], stddev=0.1))
  layer4_biases = tf.Variable(tf.constant(1.0, shape=[num_labels]))

  print(layer1_weights.get_shape())
  print(layer2_weights.get_shape())
  print(layer3_weights.get_shape())
  print(layer4_weights.get_shape())

  # Model.
  def model(data, keep_prob=None):
    conv = tf.nn.conv2d(data, layer1_weights, [1, 1, 1, 1], padding='SAME')
    # print("model: conv shape: %s" % conv.get_shape())
    pool1 = tf.nn.max_pool(conv, ksize=[1, 2, 2, 1], strides=[1, 2, 2, 1], padding='SAME')
```

```
    # print("model: pool shape: %s" % pool1.get_shape())
    hidden = tf.nn.relu(pool1 + layer1_biases)
    # print("model: hidden shape: %s" % hidden.get_shape())

    conv = tf.nn.conv2d(hidden, layer2_weights, [1, 1, 1, 1], padding='SAME')
    # print("model: conv shape: %s" % conv.get_shape())
    pool2 = tf.nn.max_pool(conv, ksize=[1, 2, 2, 1], strides=[1, 2, 2, 1], padding='SAME')
    # print("model: pool shape: %s" % pool1.get_shape())
    hidden = tf.nn.relu(pool2 + layer2_biases)
    # print("model: hidden shape: %s" % hidden.get_shape())

    shape = hidden.get_shape().as_list()
    reshape = tf.reshape(hidden, [shape[0], shape[1] * shape[2] * shape[3]])
    hidden = tf.nn.relu(tf.matmul(reshape, layer3_weights) + layer3_biases)
    if keep_prob is not None:
        drop = tf.nn.dropout(hidden, keep_prob)
        return tf.matmul(drop, layer4_weights) + layer4_biases
    else:
        return tf.matmul(hidden, layer4_weights) + layer4_biases

  keep_prob = tf.placeholder("float")
  # Training computation.
  logits = model(tf_train_dataset, keep_prob)
  loss = tf.reduce_mean(
    tf.nn.softmax_cross_entropy_with_logits(labels=tf_train_labels, logits=logits)\
    + beta * tf.nn.l2_loss(layer4_weights)\
    + beta * tf.nn.l2_loss(layer3_weights)\
    + beta * tf.nn.l2_loss(layer2_weights)\
    + beta * tf.nn.l2_loss(layer1_weights))

  # Optimizer.
  #optimizer = tf.train.GradientDescentOptimizer(0.05).minimize(loss)
  global_step = tf.Variable(0)  # count the number of steps taken.
  starter_learning_rate = tf.placeholder("float")
  learning_rate = tf.train.exponential_decay(starter_learning_rate, global_step, 10000, 0.96, staircase=True)
  optimizer = tf.train.GradientDescentOptimizer(learning_rate).minimize(loss, global_step=global_step)

  # Predictions for the training, validation, and test data.
  train_prediction = tf.nn.softmax(logits)
  valid_prediction = tf.nn.softmax(model(tf_valid_dataset))
  test_prediction = tf.nn.softmax(model(tf_test_dataset))
```

```
(5, 5, 1, 16)
(5, 5, 16, 16)
(784, 64)
(64, 10)
```

In [34]:
```
num_steps = 3001
keep_prob_rate = 0.9
starter_learning = 0.00001

with tf.Session(graph=graph) as session:
  tf.global_variables_initializer().run()
  print('Initialized')
  for step in range(num_steps):
    offset = (step * batch_size) % (train_labels.shape[0] - batch_size)
    batch_data = train_dataset[offset:(offset + batch_size), :, :, :]
    batch_labels = train_labels[offset:(offset + batch_size), :]
    feed_dict = {tf_train_dataset : batch_data, tf_train_labels : batch_labels, keep_prob: 0.9, starter_learning_rate: 0.
    _, l, predictions = session.run(
      [optimizer, loss, train_prediction], feed_dict=feed_dict)
    if (step % 50 == 0):
      print('Minibatch loss at step %d: %f' % (step, l))
      print('Minibatch accuracy: %.1f%%' % accuracy(predictions, batch_labels))
      print('Validation accuracy: %.1f%%' % accuracy(
        valid_prediction.eval(), valid_labels))
  print('Test accuracy: %.1f%%' % accuracy(test_prediction.eval(), test_labels))
```

```
Minibatch accuracy: 68.8%
Validation accuracy: 80.6%
Minibatch loss at step 2750: 2.721976
Minibatch accuracy: 75.0%
Validation accuracy: 80.5%
Minibatch loss at step 2800: 2.636328
Minibatch accuracy: 81.2%
Validation accuracy: 80.6%
Minibatch loss at step 2850: 2.731177
Minibatch accuracy: 81.2%
Validation accuracy: 80.7%
Minibatch loss at step 2900: 2.495913
Minibatch accuracy: 87.5%
Validation accuracy: 80.8%
Minibatch loss at step 2950: 2.431013
Minibatch accuracy: 81.2%
Validation accuracy: 80.7%
Minibatch loss at step 3000: 2.240144
Minibatch accuracy: 87.5%
Validation accuracy: 80.7%
Test accuracy: 87.0%
```

In [ ]:
```

```