Convolutional Neural Networks

November 1, 2018

1 Lab 11

2 Convolutional neural networks

2.1 Submitted to: Prof. Sweetlin Hemlatha

2.2 Submitted by: Prateek Singh (15BCE1091)

```
In [1]: 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]: url = 'https://commondatastorage.googleapis.com/books1000/'
        last_percent_reported = None
        data_root = '.' # Change me to store data elsewhere

        def download_progress_hook(count, blockSize, totalSize):
          global last_percent_reported
          percent = int(count * blockSize * 100 / totalSize)

          if last_percent_reported != percent:
            if percent % 5 == 0:
              sys.stdout.write("%s%%" % percent)
              sys.stdout.flush()
            else:
              sys.stdout.write(".")
              sys.stdout.flush()

            last_percent_reported = percent

        def maybe_download(filename, expected_bytes, force=False):
          """Download a file if not present, and make sure it's the right size."""
          dest_filename = os.path.join(data_root, filename)
          if force or not os.path.exists(dest_filename):
            print('Attempting to download:', filename)
            filename, _ = urlretrieve(url + filename, dest_filename, reporthook=download_progre
```

```
            print('\nDownload Complete!')
          statinfo = os.stat(dest_filename)
          if statinfo.st_size == expected_bytes:
            print('Found and verified', dest_filename)
          else:
            raise Exception(
              'Failed to verify ' + dest_filename + '. Can you get to it with a browser?')
          return dest_filename

        train_filename = maybe_download('notMNIST_large.tar.gz', 247336696)
        test_filename = maybe_download('notMNIST_small.tar.gz', 8458043)
```

```
Found and verified ./notMNIST_large.tar.gz
Found and verified ./notMNIST_small.tar.gz
```

Extract the dataset from the compressed .tar.gz file. This would give us a set of directories, labeled A through J.

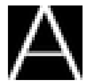
```
In [3]: num_classes = 10
        np.random.seed(133)

        def maybe_extract(filename, force=False):
          root = os.path.splitext(os.path.splitext(filename)[0])[0]  # remove .tar.gz
          if os.path.isdir(root) and not force:
            # You may override by setting force=True.
            print('%s already present - Skipping extraction of %s.' % (root, filename))
          else:
            print('Extracting data for %s. This may take a while. Please wait.' % root)
            tar = tarfile.open(filename)
            sys.stdout.flush()
            tar.extractall(data_root)
            tar.close()
          data_folders = [
            os.path.join(root, d) for d in sorted(os.listdir(root))
            if os.path.isdir(os.path.join(root, d))]
          if len(data_folders) != num_classes:
            raise Exception(
              'Expected %d folders, one per class. Found %d instead.' % (
                num_classes, len(data_folders)))
          print(data_folders)
          return data_folders

        train_folders = maybe_extract(train_filename)
        test_folders = maybe_extract(test_filename)
```

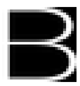
```
./notMNIST_large already present - Skipping extraction of ./notMNIST_large.tar.gz.
['./notMNIST_large/A', './notMNIST_large/B', './notMNIST_large/C', './notMNIST_large/D', './no
./notMNIST_small already present - Skipping extraction of ./notMNIST_small.tar.gz.
```

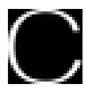
```
['./notMNIST_small/A', './notMNIST_small/B', './notMNIST_small/C', './notMNIST_small/D', './no
```

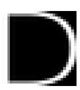
Looking at the dataset to ensure it's is sensible

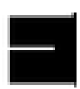
```
In [4]: for folders in train_folders:
            a = os.listdir(folders)
            display(Image(filename = folders + '/' + a[0]))
```

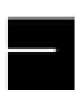


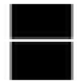


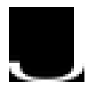












Loading the data in a more manageable format

We'll convert the entire dataset into a 3D array (image index, x, y) of floating point values, normalized to have approximately zero mean and standard deviation ~0.5 to make training easier down the road.

A few images might not be readable, we'll just skip them.

In [5]:
```
image_size = 28  # Pixel width and height.
pixel_depth = 255.0  # Number of levels per pixel.

def load_letter(folder, min_num_images):
  """Load the data for a single letter label."""
  image_files = os.listdir(folder)
  dataset = np.ndarray(shape=(len(image_files), image_size, image_size),
                         dtype=np.float32)
  print(folder)
  num_images = 0
  for image in image_files:
    image_file = os.path.join(folder, image)
    try:
      image_data = (imageio.imread(image_file).astype(float) -
                    pixel_depth / 2) / pixel_depth
      if image_data.shape != (image_size, image_size):
        raise Exception('Unexpected image shape: %s' % str(image_data.shape))
      dataset[num_images, :, :] = image_data
      num_images = num_images + 1
    except (IOError, ValueError) as e:
      print('Could not read:', image_file, ':', e, '- it\'s ok, skipping.')

  dataset = dataset[0:num_images, :, :]
  if num_images < min_num_images:
    raise Exception('Many fewer images than expected: %d < %d' %
```

```python
                    (num_images, min_num_images))

  print('Full dataset tensor:', dataset.shape)
  print('Mean:', np.mean(dataset))
  print('Standard deviation:', np.std(dataset))
  return dataset

def maybe_pickle(data_folders, min_num_images_per_class, force=False):
  dataset_names = []
  for folder in data_folders:
    set_filename = folder + '.pickle'
    dataset_names.append(set_filename)
    if os.path.exists(set_filename) and not force:
      # You may override by setting force=True.
      print('%s already present - Skipping pickling.' % set_filename)
    else:
      print('Pickling %s.' % set_filename)
      dataset = load_letter(folder, min_num_images_per_class)
      try:
        with open(set_filename, 'wb') as f:
          pickle.dump(dataset, f, pickle.HIGHEST_PROTOCOL)
      except Exception as e:
        print('Unable to save data to', set_filename, ':', e)

  return dataset_names

train_datasets = maybe_pickle(train_folders, 45000)
test_datasets = maybe_pickle(test_folders, 1800)
```

```
./notMNIST_large/A.pickle already present - Skipping pickling.
./notMNIST_large/B.pickle already present - Skipping pickling.
./notMNIST_large/C.pickle already present - Skipping pickling.
./notMNIST_large/D.pickle already present - Skipping pickling.
./notMNIST_large/E.pickle already present - Skipping pickling.
./notMNIST_large/F.pickle already present - Skipping pickling.
./notMNIST_large/G.pickle already present - Skipping pickling.
./notMNIST_large/H.pickle already present - Skipping pickling.
./notMNIST_large/I.pickle already present - Skipping pickling.
./notMNIST_large/J.pickle already present - Skipping pickling.
./notMNIST_small/A.pickle already present - Skipping pickling.
./notMNIST_small/B.pickle already present - Skipping pickling.
./notMNIST_small/C.pickle already present - Skipping pickling.
./notMNIST_small/D.pickle already present - Skipping pickling.
./notMNIST_small/E.pickle already present - Skipping pickling.
./notMNIST_small/F.pickle already present - Skipping pickling.
./notMNIST_small/G.pickle already present - Skipping pickling.
./notMNIST_small/H.pickle already present - Skipping pickling.
./notMNIST_small/I.pickle already present - Skipping pickling.
```

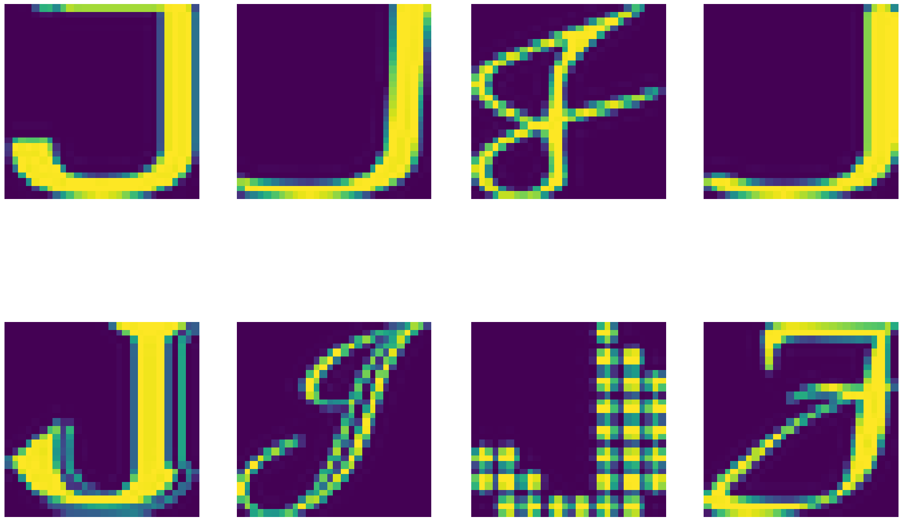
```
./notMNIST_small/J.pickle already present - Skipping pickling.
```

Verifying that the data still looks good. Displaying a sample of the labels and images from the ndarray.

```
In [6]: import random
        def disp_8_img(imgs, titles):
          """Display subplot with 8 images or less"""
          for i, img in enumerate(imgs):
            plt.subplot(2, 4, i+1)
            plt.title(titles[i])
            plt.axis('off')
            plt.imshow(img)

        def disp_sample_pickles(data_folders):
          folder = random.sample(data_folders, 1)
          pickle_filename = ''.join(folder) + '.pickle'
          print(folder)
          print(data_folders)
          try:
            with open(pickle_filename, 'rb') as f:
              dataset = pickle.load(f)
          except Exception as e:
            print('Unable to read data from', pickle_filename, ':', e)
            return
          # display
          plt.suptitle(''.join(folder)[-1])
          for i, img in enumerate(random.sample(list(dataset), 8)):
            plt.subplot(2, 4, i+1)
            plt.axis('off')
            plt.imshow(img)

        disp_sample_pickles(train_folders)
```

```
['./notMNIST_large/J']
['./notMNIST_large/A', './notMNIST_large/B', './notMNIST_large/C', './notMNIST_large/D', './no
```

J



checking for the data to be balanced across classes.

In [7]:
```python
def disp_number_images(data_folders):
  for folder in data_folders:
    pickle_filename = ''.join(folder) + '.pickle'
    try:
      with open(pickle_filename, 'rb') as f:
        dataset = pickle.load(f)
    except Exception as e:
      print('Unable to read data from', pickle_filename, ':', e)
      return
    print('Number of images in ', folder, ' : ', len(dataset))

disp_number_images(train_folders)
disp_number_images(test_folders)
```

```
Number of images in  ./notMNIST_large/A  :  52909
Number of images in  ./notMNIST_large/B  :  52911
Number of images in  ./notMNIST_large/C  :  52912
Number of images in  ./notMNIST_large/D  :  52911
Number of images in  ./notMNIST_large/E  :  52912
Number of images in  ./notMNIST_large/F  :  52912
Number of images in  ./notMNIST_large/G  :  52912
Number of images in  ./notMNIST_large/H  :  52912
Number of images in  ./notMNIST_large/I  :  52912
Number of images in  ./notMNIST_large/J  :  52911
```

```
Number of images in  ./notMNIST_small/A  :  1872
Number of images in  ./notMNIST_small/B  :  1873
Number of images in  ./notMNIST_small/C  :  1873
Number of images in  ./notMNIST_small/D  :  1873
Number of images in  ./notMNIST_small/E  :  1873
Number of images in  ./notMNIST_small/F  :  1872
Number of images in  ./notMNIST_small/G  :  1872
Number of images in  ./notMNIST_small/H  :  1872
Number of images in  ./notMNIST_small/I  :  1872
Number of images in  ./notMNIST_small/J  :  1872
```

In [8]:
```
def make_arrays(nb_rows, img_size):
  if nb_rows:
    dataset = np.ndarray((nb_rows, img_size, img_size), dtype=np.float32)
    labels = np.ndarray(nb_rows, dtype=np.int32)
  else:
    dataset, labels = None, None
  return dataset, labels

def merge_datasets(pickle_files, train_size, valid_size=0):
  num_classes = len(pickle_files)
  print(pickle_files)
  valid_dataset, valid_labels = make_arrays(valid_size, image_size)
  train_dataset, train_labels = make_arrays(train_size, image_size)
  vsize_per_class = valid_size // num_classes
  tsize_per_class = train_size // num_classes

  start_v, start_t = 0, 0
  end_v, end_t = vsize_per_class, tsize_per_class
  end_l = vsize_per_class+tsize_per_class
  for label, pickle_file in enumerate(pickle_files):
    try:
      with open(pickle_file, 'rb') as f:
        letter_set = pickle.load(f)
        # let's shuffle the letters to have random validation and training set
        np.random.shuffle(letter_set)
        if valid_dataset is not None:
          valid_letter = letter_set[:vsize_per_class, :, :]
          valid_dataset[start_v:end_v, :, :] = valid_letter
          valid_labels[start_v:end_v] = label
          start_v += vsize_per_class
          end_v += vsize_per_class

        train_letter = letter_set[vsize_per_class:end_l, :, :]
        train_dataset[start_t:end_t, :, :] = train_letter
        train_labels[start_t:end_t] = label
        start_t += tsize_per_class
```

```
            end_t += tsize_per_class
        except Exception as e:
          print('Unable to process data from', pickle_file, ':', e)
          raise

      return valid_dataset, valid_labels, train_dataset, train_labels


    train_size = 200000
    valid_size = 10000
    test_size = 10000

    valid_dataset, valid_labels, train_dataset, train_labels = merge_datasets(
      train_datasets, train_size, valid_size)
    _, _, test_dataset, test_labels = merge_datasets(test_datasets, test_size)

    print('Training:', train_dataset.shape, train_labels.shape)
    print('Validation:', valid_dataset.shape, valid_labels.shape)
    print('Testing:', test_dataset.shape, test_labels.shape)
```

```
['./notMNIST_large/A.pickle', './notMNIST_large/B.pickle', './notMNIST_large/C.pickle', './notM
['./notMNIST_small/A.pickle', './notMNIST_small/B.pickle', './notMNIST_small/C.pickle', './notM
Training: (200000, 28, 28) (200000,)
Validation: (10000, 28, 28) (10000,)
Testing: (10000, 28, 28) (10000,)
```

Next, we'll randomize the data. It's important to have the labels well shuffled for the training and test distributions to match.

```
In [9]: def randomize(dataset, labels):
          permutation = np.random.permutation(labels.shape[0])
          shuffled_dataset = dataset[permutation,:,:]
          shuffled_labels = labels[permutation]
          return shuffled_dataset, shuffled_labels
        train_dataset, train_labels = randomize(train_dataset, train_labels)
        test_dataset, test_labels = randomize(test_dataset, test_labels)
        valid_dataset, valid_labels = randomize(valid_dataset, valid_labels)
```

```
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])
```

building 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 [6]: batch_size = 16
        patch_size = 5
        depth = 16
        num_hidden = 64
```

```python
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]))

  # Model.
  def model(data):
    conv = tf.nn.conv2d(data, layer1_weights, [1, 2, 2, 1], padding='SAME')
    hidden = tf.nn.relu(conv + layer1_biases)
    conv = tf.nn.conv2d(hidden, layer2_weights, [1, 2, 2, 1], padding='SAME')
    hidden = tf.nn.relu(conv + layer2_biases)
    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))
```

```
In [7]: 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: 3.252700
Minibatch accuracy: 0.0%
Validation accuracy: 9.9%
Minibatch loss at step 50: 1.850138
Minibatch accuracy: 31.2%
Validation accuracy: 48.7%
Minibatch loss at step 100: 1.010035
Minibatch accuracy: 62.5%
Validation accuracy: 64.9%
Minibatch loss at step 150: 1.289849
Minibatch accuracy: 68.8%
Validation accuracy: 72.0%
Minibatch loss at step 200: 0.505446
Minibatch accuracy: 81.2%
Validation accuracy: 75.6%
Minibatch loss at step 250: 1.345587
Minibatch accuracy: 56.2%
Validation accuracy: 69.6%
Minibatch loss at step 300: 0.695081
Minibatch accuracy: 75.0%
Validation accuracy: 77.0%
Minibatch loss at step 350: 0.638521
Minibatch accuracy: 75.0%
Validation accuracy: 78.8%
Minibatch loss at step 400: 0.511915
Minibatch accuracy: 81.2%
Validation accuracy: 78.8%
Minibatch loss at step 450: 0.779062
```

```
Minibatch accuracy: 75.0%
Validation accuracy: 78.7%
Minibatch loss at step 500: 0.602857
Minibatch accuracy: 81.2%
Validation accuracy: 78.8%
Minibatch loss at step 550: 1.071783
Minibatch accuracy: 75.0%
Validation accuracy: 80.2%
Minibatch loss at step 600: 0.966325
Minibatch accuracy: 56.2%
Validation accuracy: 80.5%
Minibatch loss at step 650: 0.564390
Minibatch accuracy: 81.2%
Validation accuracy: 81.5%
Minibatch loss at step 700: 0.549062
Minibatch accuracy: 81.2%
Validation accuracy: 81.0%
Minibatch loss at step 750: 0.473424
Minibatch accuracy: 87.5%
Validation accuracy: 81.4%
Minibatch loss at step 800: 0.421251
Minibatch accuracy: 87.5%
Validation accuracy: 80.3%
Minibatch loss at step 850: 0.261832
Minibatch accuracy: 93.8%
Validation accuracy: 81.6%
Minibatch loss at step 900: 0.716957
Minibatch accuracy: 68.8%
Validation accuracy: 82.2%
Minibatch loss at step 950: 0.280484
Minibatch accuracy: 93.8%
Validation accuracy: 82.5%
Minibatch loss at step 1000: 0.360719
Minibatch accuracy: 93.8%
Validation accuracy: 81.6%
Test accuracy: 88.5%
```

The convolutional model above uses convolutions with stride 2 to reduce the dimensionality. Replacing 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

graph = tf.Graph()
```

```python
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]))

  # Model.
  def model(data):
    conv1 = tf.nn.conv2d(data, layer1_weights, [1, 1, 1, 1], padding='SAME')
    bias1 = tf.nn.relu(conv1 + layer1_biases)
    pool1 = tf.nn.max_pool(bias1, [1, 2, 2, 1], [1, 2, 2, 1], padding='SAME')
    conv2 = tf.nn.conv2d(pool1, layer2_weights, [1, 1, 1, 1], padding='SAME')
    bias2 = tf.nn.relu(conv2 + layer2_biases)
    pool2 = tf.nn.max_pool(bias2, [1, 2, 2, 1], [1, 2, 2, 1], padding='SAME')
    shape = pool2.get_shape().as_list()
    reshape = tf.reshape(pool2, [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))
```

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
    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: 3.045946
Minibatch accuracy: 12.5%
Validation accuracy: 10.0%
Minibatch loss at step 50: 1.833991
Minibatch accuracy: 31.2%
Validation accuracy: 46.6%
Minibatch loss at step 100: 1.044222
Minibatch accuracy: 56.2%
Validation accuracy: 62.4%
Minibatch loss at step 150: 1.086824
Minibatch accuracy: 68.8%
Validation accuracy: 70.6%
Minibatch loss at step 200: 0.626718
Minibatch accuracy: 81.2%
Validation accuracy: 73.7%
Minibatch loss at step 250: 1.310024
Minibatch accuracy: 62.5%
Validation accuracy: 74.4%
Minibatch loss at step 300: 0.477789
Minibatch accuracy: 75.0%
Validation accuracy: 77.4%
Minibatch loss at step 350: 0.495264
Minibatch accuracy: 81.2%
Validation accuracy: 79.5%
Minibatch loss at step 400: 0.597861
Minibatch accuracy: 75.0%
Validation accuracy: 79.2%
Minibatch loss at step 450: 0.637514
Minibatch accuracy: 81.2%
```

```
Validation accuracy: 80.6%
Minibatch loss at step 500: 0.441908
Minibatch accuracy: 87.5%
Validation accuracy: 80.2%
Minibatch loss at step 550: 0.707909
Minibatch accuracy: 75.0%
Validation accuracy: 81.6%
Minibatch loss at step 600: 1.103696
Minibatch accuracy: 75.0%
Validation accuracy: 82.0%
Minibatch loss at step 650: 0.399387
Minibatch accuracy: 87.5%
Validation accuracy: 83.0%
Minibatch loss at step 700: 0.450146
Minibatch accuracy: 87.5%
Validation accuracy: 82.2%
Minibatch loss at step 750: 0.412422
Minibatch accuracy: 87.5%
Validation accuracy: 82.6%
Minibatch loss at step 800: 0.354055
Minibatch accuracy: 87.5%
Validation accuracy: 83.0%
Minibatch loss at step 850: 0.240893
Minibatch accuracy: 93.8%
Validation accuracy: 83.3%
Minibatch loss at step 900: 0.460379
Minibatch accuracy: 93.8%
Validation accuracy: 83.8%
Minibatch loss at step 950: 0.301897
Minibatch accuracy: 87.5%
Validation accuracy: 84.6%
Minibatch loss at step 1000: 0.424297
Minibatch accuracy: 87.5%
Validation accuracy: 83.5%
Test accuracy: 89.8%
```