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 [0]:

```
# 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 [0]:

```
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 (18724, 28, 28) (18724,)
```

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 [0]:

```
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 (18724, 28, 28, 1) (18724, 10)
```

In [0]:

```
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 [0]:

```
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]))

  # 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=logit
s))

  # 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 [0]:

```
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 : 3.51275
Minibatch accuracy: 6.2%
Validation accuracy: 12.8%
Minibatch loss at step 50 : 1.48703
Minibatch accuracy: 43.8%
Validation accuracy: 50.4%
Minibatch loss at step 100 : 1.04377
Minibatch accuracy: 68.8%
Validation accuracy: 67.4%
Minibatch loss at step 150 : 0.601682
Minibatch accuracy: 68.8%
Validation accuracy: 73.0%
Minibatch loss at step 200 : 0.898649
Minibatch accuracy: 75.0%
Validation accuracy: 77.8%
Minibatch loss at step 250 : 1.3637
Minibatch accuracy: 56.2%
Validation accuracy: 75.4%
Minibatch loss at step 300 : 1.41968
Minibatch accuracy: 62.5%
Validation accuracy: 76.0%
Minibatch loss at step 350 : 0.300648
Minibatch accuracy: 81.2%
Validation accuracy: 80.2%
Minibatch loss at step 400 : 1.32092
Minibatch accuracy: 56.2%
Validation accuracy: 80.4%
Minibatch loss at step 450 : 0.556701
Minibatch accuracy: 81.2%
Validation accuracy: 79.4%
Minibatch loss at step 500 : 1.65595
Minibatch accuracy: 43.8%
Validation accuracy: 79.6%
Minibatch loss at step 550 : 1.06995
Minibatch accuracy: 75.0%
Validation accuracy: 81.2%
Minibatch loss at step 600 : 0.223684
Minibatch accuracy: 100.0%
Validation accuracy: 82.3%
Minibatch loss at step 650 : 0.619602
Minibatch accuracy: 87.5%
Validation accuracy: 81.8%
Minibatch loss at step 700 : 0.812091
Minibatch accuracy: 75.0%
Validation accuracy: 82.4%
Minibatch loss at step 750 : 0.276302
Minibatch accuracy: 87.5%
Validation accuracy: 82.3%
Minibatch loss at step 800 : 0.450241
Minibatch accuracy: 81.2%
Validation accuracy: 82.3%
Minibatch loss at step 850 : 0.137139
Minibatch accuracy: 93.8%
Validation accuracy: 82.3%
Minibatch loss at step 900 : 0.52664
Minibatch accuracy: 75.0%
Validation accuracy: 82.2%
Minibatch loss at step 950 : 0.623835
Minibatch accuracy: 87.5%
Validation accuracy: 82.1%
```

```
Minibatch loss at step 1000 : 0.243114
Minibatch accuracy: 93.8%
Validation accuracy: 82.9%
Test accuracy: 90.0%
```

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.

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/)](http://yann.lecun.com/exdb/lenet/) architecture, adding Dropout, and/or adding learning rate decay.
