Deep Learning

Assignment 3

Previously in `2_fullyconnected.ipynb`, you trained a logistic regression and a neural network model.

The goal of this assignment is to explore regularization techniques.

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
```

First reload the data we generated in `1_notmnist.ipynb`.

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 shape that's more adapted to the models we're going to train:

- data as a flat matrix,
- labels as float 1-hot encodings.

In [3]:
```
image_size = 28
num_labels = 10

def reformat(dataset, labels):
  dataset = dataset.reshape((-1, image_size * image_size)).astype(np.float32)
  # Map 1 to [0.0, 1.0, 0.0 ...], 2 to [0.0, 0.0, 1.0 ...]
  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, 784) (200000, 10)
Validation set (10000, 784) (10000, 10)
Test set (10000, 784) (10000, 10)
```

In [6]:
```
def accuracy(predictions, labels):
  return (100.0 * np.sum(np.argmax(predictions, 1) == np.argmax(labels, 1))
          / predictions.shape[0])
```

Problem 1

Introduce and tune L2 regularization for both logistic and neural network models. Remember that L2 amounts to adding a penalty on the norm of the weights to the loss. In TensorFlow, you can compute the L2 loss for a tensor `t` using `nn.l2_loss(t)`. The right amount of regularization should improve your validation / test accuracy.

In [4]:
```
# add l2 to logistic model

batch_size = 128

beta = 0.01

graph = tf.Graph()
with graph.as_default():

  # Input data. For the training data, we use a placeholder that will be fed
  # at run time with a training minibatch.
  tf_train_dataset = tf.placeholder(tf.float32,
                                    shape=(batch_size, image_size * image_size))
  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.
  weights = tf.Variable(
    tf.truncated_normal([image_size * image_size, num_labels]))
  biases = tf.Variable(tf.zeros([num_labels]))

  # Training computation.
  logits = tf.matmul(tf_train_dataset, weights) + biases
  loss = tf.reduce_mean(
    tf.nn.softmax_cross_entropy_with_logits(labels=tf_train_labels, logits=logits) + beta * tf.nn.l2_loss(weights))

  # Optimizer.
  optimizer = tf.train.GradientDescentOptimizer(0.5).minimize(loss)

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

In [7]:

```
num_steps = 3001

with tf.Session(graph=graph) as session:
  tf.global_variables_initializer().run()
  print("Initialized")
  for step in range(num_steps):
    # Pick an offset within the training data, which has been randomized.
    # Note: we could use better randomization across epochs.
    offset = (step * batch_size) % (train_labels.shape[0] - batch_size)
    # Generate a minibatch.
    batch_data = train_dataset[offset:(offset + batch_size), :]
    batch_labels = train_labels[offset:(offset + batch_size), :]
    # Prepare a dictionary telling the session where to feed the minibatch.
    # The key of the dictionary is the placeholder node of the graph to be fed,
    # and the value is the numpy array to feed to it.
    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 % 500 == 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: 47.156769
Minibatch accuracy: 10.9%
Validation accuracy: 13.7%
Minibatch loss at step 500: 0.910600
Minibatch accuracy: 85.9%
Validation accuracy: 79.9%
Minibatch loss at step 1000: 0.694983
Minibatch accuracy: 81.2%
Validation accuracy: 80.6%
Minibatch loss at step 1500: 0.740580
Minibatch accuracy: 81.2%
Validation accuracy: 81.0%
Minibatch loss at step 2000: 0.620068
Minibatch accuracy: 85.2%
Validation accuracy: 80.3%
```

```
Minibatch loss at step 2500: 0.693578
Minibatch accuracy: 82.8%
Validation accuracy: 81.1%
Minibatch loss at step 3000: 0.664391
Minibatch accuracy: 83.6%
Validation accuracy: 81.2%
Test accuracy: 87.9%
```

In [12]:
```
# add l2 to the neural network

batch_size = 128
hidden_nodes = 1024

beta1 = 0.01
beta2 = 0.01

graph = tf.Graph()
with graph.as_default():

  # Input data. For the training data, we use a placeholder that will be fed
  # at run time with a training minibatch.
  tf_train_dataset = tf.placeholder(tf.float32,
                                    shape=(batch_size, image_size * image_size))
  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)

  # new hidden layer
  hidden_weights = tf.Variable( tf.truncated_normal([image_size * image_size, hidden_nodes]) )
  hidden_biases = tf.Variable( tf.zeros([hidden_nodes]))
  hidden_layer = tf.nn.relu( tf.matmul( tf_train_dataset, hidden_weights) + hidden_biases)

  # Variables.
  weights = tf.Variable(
    tf.truncated_normal([hidden_nodes, num_labels]))
  biases = tf.Variable(tf.zeros([num_labels]))

  # Training computation.
  logits = tf.matmul(hidden_layer, weights) + biases
  loss = tf.reduce_mean(
    tf.nn.softmax_cross_entropy_with_logits(labels=tf_train_labels, logits=logits)\
    + beta1 * tf.nn.l2_loss(hidden_weights)\
    + beta2 * tf.nn.l2_loss(weights))

  # Optimizer.
  optimizer = tf.train.GradientDescentOptimizer(0.5).minimize(loss)

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

```
  valid_relu = tf.nn.relu(  tf.matmul(tf_valid_dataset, hidden_weights) + hidden_biases)
  valid_prediction = tf.nn.softmax(
    tf.matmul(valid_relu, weights) + biases)
  test_relu = tf.nn.relu( tf.matmul( tf_test_dataset, hidden_weights) + hidden_biases)
  test_prediction = tf.nn.softmax(tf.matmul(test_relu, weights) + biases)
```

In [13]:
```
num_steps = 3001

with tf.Session(graph=graph) as session:
  tf.global_variables_initializer().run()
  print("Initialized")
  for step in range(num_steps):
    # Pick an offset within the training data, which has been randomized.
    # Note: we could use better randomization across epochs.
    offset = (step * batch_size) % (train_labels.shape[0] - batch_size)
    # Generate a minibatch.
    batch_data = train_dataset[offset:(offset + batch_size), :]
    batch_labels = train_labels[offset:(offset + batch_size), :]
    # Prepare a dictionary telling the session where to feed the minibatch.
    # The key of the dictionary is the placeholder node of the graph to be fed,
    # and the value is the numpy array to feed to it.
    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 % 500 == 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: 3494.061523
Minibatch accuracy: 4.7%
Validation accuracy: 32.8%
Minibatch loss at step 500: 21.304165
Minibatch accuracy: 87.5%
Validation accuracy: 83.8%
Minibatch loss at step 1000: 0.808530
Minibatch accuracy: 84.4%
Validation accuracy: 82.3%
Minibatch loss at step 1500: 0.733221
Minibatch accuracy: 82.8%
Validation accuracy: 83.6%
Minibatch loss at step 2000: 0.645359
Minibatch accuracy: 85.9%
Validation accuracy: 82.1%
```

```
Minibatch loss at step 2500: 0.701366
Minibatch accuracy: 85.2%
Validation accuracy: 83.2%
Minibatch loss at step 3000: 0.645480
Minibatch accuracy: 85.2%
Validation accuracy: 83.2%
Test accuracy: 89.6%
```

Problem 2

Let's demonstrate an extreme case of overfitting. Restrict your training data to just a few batches. What happens?

In [14]:
```
train_subset = 256
small_train_dataset = train_dataset[:train_subset, :]
small_train_labels = train_labels[:train_subset]
```

In [15]:
```
# add l2 to logistic model

batch_size = 128

beta = 0.01

graph = tf.Graph()
with graph.as_default():

  # Input data. For the training data, we use a placeholder that will be fed
  # at run time with a training minibatch.
  tf_train_dataset = tf.placeholder(tf.float32,
                                    shape=(batch_size, image_size * image_size))
  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.
  weights = tf.Variable(
    tf.truncated_normal([image_size * image_size, num_labels]))
  biases = tf.Variable(tf.zeros([num_labels]))

  # Training computation.
  logits = tf.matmul(tf_train_dataset, weights) + biases
  loss = tf.reduce_mean(
    tf.nn.softmax_cross_entropy_with_logits(labels=tf_train_labels, logits=logits) + beta * tf.nn.l2_loss(weights))

  # Optimizer.
  optimizer = tf.train.GradientDescentOptimizer(0.5).minimize(loss)

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

In [17]:
```
num_steps = 3001

with tf.Session(graph=graph) as session:
  tf.global_variables_initializer().run()
  print("Initialized")
  for step in range(num_steps):
    # Pick an offset within the training data, which has been randomized.
    # Note: we could use better randomization across epochs.
    offset = (step * batch_size) % (small_train_labels.shape[0] - batch_size)
    # Generate a minibatch.
    batch_data = small_train_dataset[offset:(offset + batch_size), :]
    batch_labels = small_train_labels[offset:(offset + batch_size), :]
    # Prepare a dictionary telling the session where to feed the minibatch.
    # The key of the dictionary is the placeholder node of the graph to be fed,
    # and the value is the numpy array to feed to it.
    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 % 500 == 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: 48.186317
Minibatch accuracy: 8.6%
Validation accuracy: 10.0%
Minibatch loss at step 500: 0.319718
Minibatch accuracy: 100.0%
Validation accuracy: 71.2%
Minibatch loss at step 1000: 0.128471
Minibatch accuracy: 100.0%
Validation accuracy: 72.5%
Minibatch loss at step 1500: 0.126468
Minibatch accuracy: 100.0%
Validation accuracy: 72.5%
Minibatch loss at step 2000: 0.126113
Minibatch accuracy: 100.0%
Validation accuracy: 72.5%
```

```
Minibatch loss at step 2500: 0.125942
Minibatch accuracy: 100.0%
Validation accuracy: 72.4%
Minibatch loss at step 3000: 0.125852
Minibatch accuracy: 100.0%
Validation accuracy: 72.4%
Test accuracy: 79.8%
```

In [26]:
```
# add l2 to the neural network

batch_size = 128
hidden_nodes = 1024

beta = 0.01

graph = tf.Graph()
with graph.as_default():

  # Input data. For the training data, we use a placeholder that will be fed
  # at run time with a training minibatch.
  tf_train_dataset = tf.placeholder(tf.float32,
                                    shape=(batch_size, image_size * image_size))
  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)

  # new hidden layer
  hidden_weights = tf.Variable( tf.truncated_normal([image_size * image_size, hidden_nodes]) )
  hidden_biases = tf.Variable( tf.zeros([hidden_nodes]))
  hidden_layer = tf.nn.relu( tf.matmul( tf_train_dataset, hidden_weights) + hidden_biases)

  # Variables.
  weights = tf.Variable(
    tf.truncated_normal([hidden_nodes, num_labels]))
  biases = tf.Variable(tf.zeros([num_labels]))

  # Training computation.
  logits = tf.matmul(hidden_layer, weights) + biases
  loss = tf.reduce_mean(
    tf.nn.softmax_cross_entropy_with_logits(labels=tf_train_labels, logits=logits)\
    + beta * tf.nn.l2_loss(weights))

  # Optimizer.
  optimizer = tf.train.GradientDescentOptimizer(0.5).minimize(loss)

  # Predictions for the training, validation, and test data.
  train_prediction = tf.nn.softmax(logits)
  valid_relu = tf.nn.relu(  tf.matmul(tf_valid_dataset, hidden_weights) + hidden_biases)
  valid_prediction = tf.nn.softmax(
```

```
    tf.matmul(valid_relu, weights) + biases)
  test_relu = tf.nn.relu( tf.matmul( tf_test_dataset, hidden_weights) + hidden_biases)
  test_prediction = tf.nn.softmax(tf.matmul(test_relu, weights) + biases)
```

In [27]:
```
num_steps = 3001

with tf.Session(graph=graph) as session:
  tf.global_variables_initializer().run()
  print("Initialized")
  for step in range(num_steps):
    # Pick an offset within the training data, which has been randomized.
    # Note: we could use better randomization across epochs.
    offset = (step * batch_size) % (small_train_labels.shape[0] - batch_size)
    # Generate a minibatch.
    batch_data = small_train_dataset[offset:(offset + batch_size), :]
    batch_labels = small_train_labels[offset:(offset + batch_size), :]
    # Prepare a dictionary telling the session where to feed the minibatch.
    # The key of the dictionary is the placeholder node of the graph to be fed,
    # and the value is the numpy array to feed to it.
    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 % 500 == 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: 355.508209
Minibatch accuracy: 10.2%
Validation accuracy: 36.7%
Minibatch loss at step 500: 0.278965
Minibatch accuracy: 100.0%
Validation accuracy: 68.4%
Minibatch loss at step 1000: 0.003259
Minibatch accuracy: 100.0%
Validation accuracy: 71.0%
Minibatch loss at step 1500: 0.001980
Minibatch accuracy: 100.0%
Validation accuracy: 71.5%
Minibatch loss at step 2000: 0.001970
Minibatch accuracy: 100.0%
Validation accuracy: 71.5%
```

```
Minibatch loss at step 2500: 0.001968
Minibatch accuracy: 100.0%
Validation accuracy: 71.5%
Minibatch loss at step 3000: 0.001965
Minibatch accuracy: 100.0%
Validation accuracy: 71.5%
Test accuracy: 79.2%
```

Problem 3

Introduce Dropout on the hidden layer of the neural network. Remember: Dropout should only be introduced during training, not evaluation, otherwise your evaluation results would be stochastic as well. TensorFlow provides `nn.dropout()` for that, but you have to make sure it's only inserted during training.

What happens to our extreme overfitting case?

In [28]:
```
# add l2 to the neural network
# add dropout

batch_size = 128
hidden_nodes = 1024

beta = 0.01

graph = tf.Graph()
with graph.as_default():

  # Input data. For the training data, we use a placeholder that will be fed
  # at run time with a training minibatch.
  tf_train_dataset = tf.placeholder(tf.float32,
                                    shape=(batch_size, image_size * image_size))
  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)

  # new hidden layer
  hidden_weights = tf.Variable( tf.truncated_normal([image_size * image_size, hidden_nodes]) )
  hidden_biases = tf.Variable( tf.zeros([hidden_nodes]))
  hidden_layer = tf.nn.relu( tf.matmul( tf_train_dataset, hidden_weights) + hidden_biases)

  # add dropout on hidden layer
  dropout = tf.placeholder("float")
  hidden_layer_drop = tf.nn.dropout(hidden_layer, dropout)


  # Variables.
  weights = tf.Variable(
    tf.truncated_normal([hidden_nodes, num_labels]))
  biases = tf.Variable(tf.zeros([num_labels]))

  # Training computation.
  logits = tf.matmul(hidden_layer_drop, weights) + biases
  loss = tf.reduce_mean(
    tf.nn.softmax_cross_entropy_with_logits(labels=tf_train_labels, logits=logits)\
    + beta * tf.nn.l2_loss(weights))

  # Optimizer.
```

```
  optimizer = tf.train.GradientDescentOptimizer(0.5).minimize(loss)

  # Predictions for the training, validation, and test data.
  train_prediction = tf.nn.softmax(logits)
  valid_relu = tf.nn.relu(  tf.matmul(tf_valid_dataset, hidden_weights) + hidden_biases)
  valid_prediction = tf.nn.softmax(
    tf.matmul(valid_relu, weights) + biases)
  test_relu = tf.nn.relu( tf.matmul( tf_test_dataset, hidden_weights) + hidden_biases)
  test_prediction = tf.nn.softmax(tf.matmul(test_relu, weights) + biases)
```

In [29]:
```
num_steps = 3001
dropout_rate = 0.5

with tf.Session(graph=graph) as session:
  tf.global_variables_initializer().run()
  print("Initialized")
  for step in range(num_steps):
    # Pick an offset within the training data, which has been randomized.
    # Note: we could use better randomization across epochs.
    offset = (step * batch_size) % (small_train_labels.shape[0] - batch_size)
    # Generate a minibatch.
    batch_data = small_train_dataset[offset:(offset + batch_size), :]
    batch_labels = small_train_labels[offset:(offset + batch_size), :]
    # Prepare a dictionary telling the session where to feed the minibatch.
    # The key of the dictionary is the placeholder node of the graph to be fed,
    # and the value is the numpy array to feed to it.
    feed_dict = {tf_train_dataset : batch_data, tf_train_labels : batch_labels, dropout: dropout_rate}
    _, l, predictions = session.run(
      [optimizer, loss, train_prediction], feed_dict=feed_dict)
    if (step % 500 == 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: 535.049316
Minibatch accuracy: 7.8%
Validation accuracy: 22.7%
Minibatch loss at step 500: 2.117288
Minibatch accuracy: 99.2%
Validation accuracy: 70.5%
Minibatch loss at step 1000: 0.407852
Minibatch accuracy: 100.0%
Validation accuracy: 69.5%
Minibatch loss at step 1500: 0.360406
Minibatch accuracy: 100.0%
Validation accuracy: 68.3%
Minibatch loss at step 2000: 1.595005
Minibatch accuracy: 100.0%
```

```
Validation accuracy: 70.5%
Minibatch loss at step 2500: 0.789690
Minibatch accuracy: 100.0%
Validation accuracy: 71.5%
Minibatch loss at step 3000: 0.173409
Minibatch accuracy: 100.0%
Validation accuracy: 68.5%
Test accuracy: 75.9%
```

Problem 4

Try to get the best performance you can using a multi-layer model! The best reported test accuracy using a deep network is 97.1% (http://yaroslavvb.blogspot.com/2011/09/notmnist-dataset.html?showComment=1391023266211#c8758720086795711595).

One avenue you can explore is to add multiple layers.

Another one is to use learning rate decay:

```
global_step = tf.Variable(0)  # count the number of steps taken.
learning_rate = tf.train.exponential_decay(0.5, global_step, ...)
optimizer = tf.train.GradientDescentOptimizer(learning_rate).minimize(loss, global_step=global_step)
```

In [30]:
```
# add l2 to the neural network
# add dropout
# to use full dataset

batch_size = 128
hidden_nodes = 1024

beta = 0.01

graph = tf.Graph()
with graph.as_default():

  # Input data. For the training data, we use a placeholder that will be fed
  # at run time with a training minibatch.
  tf_train_dataset = tf.placeholder(tf.float32,
                                    shape=(batch_size, image_size * image_size))
  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)

  # new hidden layer
  hidden_weights = tf.Variable( tf.truncated_normal([image_size * image_size, hidden_nodes]) )
  hidden_biases = tf.Variable( tf.zeros([hidden_nodes]))
  hidden_layer = tf.nn.relu( tf.matmul( tf_train_dataset, hidden_weights) + hidden_biases)

  # add dropout on hidden layer
  dropout = tf.placeholder("float")
  hidden_layer_drop = tf.nn.dropout(hidden_layer, dropout)

  # Variables.
  weights = tf.Variable(
    tf.truncated_normal([hidden_nodes, num_labels]))
  biases = tf.Variable(tf.zeros([num_labels]))

  # Training computation.
  logits = tf.matmul(hidden_layer_drop, weights) + biases
  loss = tf.reduce_mean(
    tf.nn.softmax_cross_entropy_with_logits(labels=tf_train_labels, logits=logits)\
    + beta * tf.nn.l2_loss(weights))

  # Optimizer.
```

```
  optimizer = tf.train.GradientDescentOptimizer(0.5).minimize(loss)

  # Predictions for the training, validation, and test data.
  train_prediction = tf.nn.softmax(logits)
  valid_relu = tf.nn.relu(  tf.matmul(tf_valid_dataset, hidden_weights) + hidden_biases)
  valid_prediction = tf.nn.softmax(
    tf.matmul(valid_relu, weights) + biases)
  test_relu = tf.nn.relu( tf.matmul( tf_test_dataset, hidden_weights) + hidden_biases)
  test_prediction = tf.nn.softmax(tf.matmul(test_relu, weights) + biases)
```

In [31]:
```
num_steps = 3001
dropout_rate = 0.5

with tf.Session(graph=graph) as session:
  tf.global_variables_initializer().run()
  print("Initialized")
  for step in range(num_steps):
    # Pick an offset within the training data, which has been randomized.
    # Note: we could use better randomization across epochs.
    offset = (step * batch_size) % (train_labels.shape[0] - batch_size)
    # Generate a minibatch.
    batch_data = train_dataset[offset:(offset + batch_size), :]
    batch_labels = train_labels[offset:(offset + batch_size), :]
    # Prepare a dictionary telling the session where to feed the minibatch.
    # The key of the dictionary is the placeholder node of the graph to be fed,
    # and the value is the numpy array to feed to it.
    feed_dict = {tf_train_dataset : batch_data, tf_train_labels : batch_labels, dropout: dropout_rate}
    _, l, predictions = session.run(
      [optimizer, loss, train_prediction], feed_dict=feed_dict)
    if (step % 500 == 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: 529.353088
Minibatch accuracy: 16.4%
Validation accuracy: 33.1%
Minibatch loss at step 500: 22.772110
Minibatch accuracy: 71.9%
Validation accuracy: 77.2%
Minibatch loss at step 1000: 16.299805
Minibatch accuracy: 68.8%
Validation accuracy: 71.2%
Minibatch loss at step 1500: 11.092232
Minibatch accuracy: 70.3%
Validation accuracy: 77.4%
Minibatch loss at step 2000: 10.009911
Minibatch accuracy: 65.6%
```

```
Validation accuracy: 74.8%
Minibatch loss at step 2500: 7.858951
Minibatch accuracy: 75.0%
Validation accuracy: 77.4%
Minibatch loss at step 3000: 6.109713
Minibatch accuracy: 74.2%
Validation accuracy: 77.4%
Test accuracy: 85.4%
```

In [34]:
```
# tune keep prod
num_steps = 3001
dropout_rate = 0.8

with tf.Session(graph=graph) as session:
  tf.global_variables_initializer().run()
  print("Initialized")
  for step in range(num_steps):
    # Pick an offset within the training data, which has been randomized.
    # Note: we could use better randomization across epochs.
    offset = (step * batch_size) % (train_labels.shape[0] - batch_size)
    # Generate a minibatch.
    batch_data = train_dataset[offset:(offset + batch_size), :]
    batch_labels = train_labels[offset:(offset + batch_size), :]
    # Prepare a dictionary telling the session where to feed the minibatch.
    # The key of the dictionary is the placeholder node of the graph to be fed,
    # and the value is the numpy array to feed to it.
    feed_dict = {tf_train_dataset : batch_data, tf_train_labels : batch_labels, dropout: dropout_rate}
    _, l, predictions = session.run(
      [optimizer, loss, train_prediction], feed_dict=feed_dict)
    if (step % 500 == 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: 387.364868
Minibatch accuracy: 8.6%
Validation accuracy: 29.6%
Minibatch loss at step 500: 15.623976
Minibatch accuracy: 81.2%
Validation accuracy: 77.2%
Minibatch loss at step 1000: 6.056636
Minibatch accuracy: 84.4%
Validation accuracy: 76.6%
Minibatch loss at step 1500: 9.579308
Minibatch accuracy: 69.5%
Validation accuracy: 75.9%
Minibatch loss at step 2000: 7.590422
```

```
Minibatch accuracy: 72.7%
Validation accuracy: 77.9%
Minibatch loss at step 2500: 6.492410
Minibatch accuracy: 82.8%
Validation accuracy: 77.5%
Minibatch loss at step 3000: 3.533623
Minibatch accuracy: 80.5%
Validation accuracy: 78.2%
Test accuracy: 85.3%
```

In [35]:

```
# through testing, dropout_rate should be keep rate
# definition mistake
```

In [59]:
```
# add l2 to the neural network
# add dropout
# to use full dataset
# add to use learning rate

batch_size = 128
hidden_nodes = 1024
hidden_nodes1 = 512
hidden_nodes2 = 256

beta = 0.01

graph = tf.Graph()
with graph.as_default():

  # Input data. For the training data, we use a placeholder that will be fed
  # at run time with a training minibatch.
  tf_train_dataset = tf.placeholder(tf.float32,
                                    shape=(batch_size, image_size * image_size))
  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)

  # new hidden layer
  hidden_weights = tf.Variable( tf.truncated_normal([image_size * image_size, hidden_nodes]) )
  hidden_biases = tf.Variable( tf.zeros([hidden_nodes]))
  hidden_layer = tf.nn.relu( tf.matmul( tf_train_dataset, hidden_weights) + hidden_biases)

  # add dropout on hidden layer
  dropout = tf.placeholder("float")
  hidden_layer_drop = tf.nn.dropout(hidden_layer, dropout)

  # Variables.
  weights = tf.Variable(tf.truncated_normal([hidden_nodes, num_labels]))
  biases = tf.Variable(tf.zeros([num_labels]))

  # Training computation.
  logits = tf.matmul(hidden_layer_drop, weights) + biases
  loss = tf.reduce_mean(
    tf.nn.softmax_cross_entropy_with_logits(labels=tf_train_labels, logits=logits)\
    + beta * tf.nn.l2_loss(weights))
```

```
# Optimizer.
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)
#optimizer = tf.train.GradientDescentOptimizer(0.5).minimize(loss)

# Predictions for the training, validation, and test data.
train_prediction = tf.nn.softmax(logits)
valid_relu = tf.nn.relu(  tf.matmul(tf_valid_dataset, hidden_weights) + hidden_biases)
#valid_relu = tf.nn.relu(  tf.matmul(valid_relu, hidden_weights1) + hidden_biases1)
valid_prediction = tf.nn.softmax(
  tf.matmul(valid_relu, weights) + biases)

test_relu = tf.nn.relu( tf.matmul( tf_test_dataset, hidden_weights) + hidden_biases)
#test_relu = tf.nn.relu( tf.matmul( test_relu, hidden_weights1) + hidden_biases1)
test_prediction = tf.nn.softmax(tf.matmul(test_relu, weights) + biases)
```

In [60]:
```
# tune keep prod
num_steps = 3001
dropout_rate = 0.8
dropout_rate1 = 0.8

with tf.Session(graph=graph) as session:
  tf.global_variables_initializer().run()
  print("Initialized")
  for step in range(num_steps):
    # Pick an offset within the training data, which has been randomized.
    # Note: we could use better randomization across epochs.
    offset = (step * batch_size) % (train_labels.shape[0] - batch_size)
    # Generate a minibatch.
    batch_data = train_dataset[offset:(offset + batch_size), :]
    batch_labels = train_labels[offset:(offset + batch_size), :]
    # Prepare a dictionary telling the session where to feed the minibatch.
    # The key of the dictionary is the placeholder node of the graph to be fed,
    # and the value is the numpy array to feed to it.
    feed_dict = {tf_train_dataset : batch_data, tf_train_labels : batch_labels,
                 dropout: dropout_rate,
                 starter_learning_rate: 0.5}
    _, l, predictions = session.run(
      [optimizer, loss, train_prediction], feed_dict=feed_dict)
    if (step % 500 == 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: 482.271332
Minibatch accuracy: 11.7%
Validation accuracy: 35.1%
Minibatch loss at step 500: 13.050676
Minibatch accuracy: 77.3%
Validation accuracy: 78.5%
Minibatch loss at step 1000: 7.028480
Minibatch accuracy: 84.4%
Validation accuracy: 77.0%
Minibatch loss at step 1500: 15.434306
```

```
Minibatch accuracy: 71.1%
Validation accuracy: 77.3%
Minibatch loss at step 2000: 9.080334
Minibatch accuracy: 79.7%
Validation accuracy: 79.6%
Minibatch loss at step 2500: 10.153558
Minibatch accuracy: 75.8%
Validation accuracy: 74.4%
Minibatch loss at step 3000: 6.091805
Minibatch accuracy: 78.1%
Validation accuracy: 76.7%
Test accuracy: 84.4%
```

In [102]:

```
# Starter_learning_rate should not be too big.
# at the beginning, I set it to 0.5. I got a lot of nan errors
```

In [103]:
```
# add l2 to the neural network
# add dropout
# to use full dataset
# add to use learning rate
# add more layers

batch_size = 128
hidden_nodes1 = 1024
hidden_nodes2 = 512

beta = 0.001

graph = tf.Graph()
with graph.as_default():

  # Input data. For the training data, we use a placeholder that will be fed
  # at run time with a training minibatch.
  tf_train_dataset = tf.placeholder(tf.float32, shape=(batch_size, image_size * image_size))
  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)

  # new hidden layer
  hidden_weights = tf.Variable( tf.truncated_normal([image_size * image_size, hidden_nodes1]) )
  hidden_biases = tf.Variable( tf.zeros([hidden_nodes1]))
  hidden_layer = tf.nn.relu( tf.matmul( tf_train_dataset, hidden_weights) + hidden_biases)


  # add dropout on hidden layer
  keep_prob = tf.placeholder("float")
  hidden_layer_drop = tf.nn.dropout(hidden_layer, keep_prob)

  # new layer
  hidden_weights2 = tf.Variable( tf.truncated_normal([hidden_nodes1, hidden_nodes2]) )
  hidden_biases2 = tf.Variable( tf.zeros([hidden_nodes2]))
  hidden_layer2 = tf.nn.relu( tf.matmul(hidden_layer_drop, hidden_weights2) + hidden_biases2)

  hidden_layer_drop2 = tf.nn.dropout(hidden_layer2, keep_prob)

  # Variables.
  weights = tf.Variable(tf.truncated_normal([hidden_nodes2, num_labels]))
```

```
biases = tf.Variable(tf.zeros([num_labels]))

# Training computation.
logits = tf.matmul(hidden_layer_drop2, weights) + biases

loss = tf.reduce_mean(
  tf.nn.softmax_cross_entropy_with_logits(labels=tf_train_labels, logits=logits))
loss = tf.reduce_mean(loss + beta * tf.nn.l2_loss(weights))

# Optimizer.
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)
#optimizer = tf.train.GradientDescentOptimizer(0.5).minimize(loss)

# Predictions for the training, validation, and test data.
train_prediction = tf.nn.softmax(logits)

valid_relu1 = tf.nn.relu(  tf.matmul(tf_valid_dataset, hidden_weights) + hidden_biases)
valid_relu = tf.nn.relu(  tf.matmul(valid_relu1, hidden_weights2) + hidden_biases2)
valid_prediction = tf.nn.softmax(tf.matmul(valid_relu, weights) + biases)

test_relu1 = tf.nn.relu( tf.matmul(tf_test_dataset, hidden_weights) + hidden_biases)
test_relu = tf.nn.relu( tf.matmul(test_relu1, hidden_weights2) + hidden_biases2)
test_prediction = tf.nn.softmax(tf.matmul(test_relu, weights) + biases)
```

In [104]:
```
# tune keep prod
num_steps = 3001
keep_prob_rate = 1.


with tf.Session(graph=graph) as session:
  tf.global_variables_initializer().run()
  print("Initialized")
  for step in range(num_steps):
    # Pick an offset within the training data, which has been randomized.
    # Note: we could use better randomization across epochs.
    offset = (step * batch_size) % (train_labels.shape[0] - batch_size)
    # Generate a minibatch.
    batch_data = train_dataset[offset:(offset + batch_size), :]
    batch_labels = train_labels[offset:(offset + batch_size), :]
    # Prepare a dictionary telling the session where to feed the minibatch.
    # The key of the dictionary is the placeholder node of the graph to be fed,
    # and the value is the numpy array to feed to it.
    feed_dict = {tf_train_dataset : batch_data, tf_train_labels : batch_labels,
                 keep_prob: keep_prob_rate,
                 starter_learning_rate: 0.001}
    _, l, predictions = session.run(
      [optimizer, loss, train_prediction], feed_dict=feed_dict)
    if (step % 500 == 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: 6457.143066
Minibatch accuracy: 13.3%
Validation accuracy: 10.2%
Minibatch loss at step 500: 386.485168
Minibatch accuracy: 78.1%
Validation accuracy: 76.5%
Minibatch loss at step 1000: 140.711548
Minibatch accuracy: 82.8%
Validation accuracy: 77.2%
Minibatch loss at step 1500: 226.031113
```

```
Minibatch accuracy: 81.2%
Validation accuracy: 79.0%
Minibatch loss at step 2000: 126.234245
Minibatch accuracy: 82.8%
Validation accuracy: 79.1%
Minibatch loss at step 2500: 151.252487
Minibatch accuracy: 80.5%
Validation accuracy: 79.5%
Minibatch loss at step 3000: 82.338501
Minibatch accuracy: 85.2%
Validation accuracy: 79.9%
Test accuracy: 87.0%
```

In [109]:
```
# add l2 to the neural network
# add dropout
# to use full dataset
# add to use learning rate
# add more layers

batch_size = 128
hidden_nodes1 = 1024
hidden_nodes2 = 512
hidden_nodes3 = 256

beta = 0.01

graph = tf.Graph()
with graph.as_default():

  # Input data. For the training data, we use a placeholder that will be fed
  # at run time with a training minibatch.
  tf_train_dataset = tf.placeholder(tf.float32,
                                    shape=(batch_size, image_size * image_size))
  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)

  # new hidden layer
  hidden_weights = tf.Variable( tf.truncated_normal([image_size * image_size, hidden_nodes]) )
  hidden_biases = tf.Variable( tf.zeros([hidden_nodes]))
  hidden_layer = tf.nn.relu( tf.matmul( tf_train_dataset, hidden_weights) + hidden_biases)

  # add dropout on hidden layer
  keep_prob = tf.placeholder("float")
  hidden_layer_drop = tf.nn.dropout(hidden_layer, keep_prob)

  # new layer
  hidden_weights1 = tf.Variable( tf.truncated_normal([hidden_nodes, hidden_nodes1]) )
  hidden_biases1 = tf.Variable( tf.zeros([hidden_nodes1]))
  hidden_layer1 = tf.nn.relu( tf.matmul(hidden_layer_drop, hidden_weights1) + hidden_biases1)

  # add a new dropout on hidden layer
  hidden_layer_drop1 = tf.nn.dropout(hidden_layer1, keep_prob)
```

```
    # another new layer
    hidden_weights2 = tf.Variable( tf.truncated_normal([hidden_nodes1, hidden_nodes2]) )
    hidden_biases2 = tf.Variable( tf.zeros([hidden_nodes2]))
    hidden_layer2 = tf.nn.relu( tf.matmul(hidden_layer_drop1, hidden_weights2) + hidden_biases2)

    # add another new dropout on hidden layer
    hidden_layer_drop2 = tf.nn.dropout(hidden_layer2, keep_prob)

    # Variables.
    weights = tf.Variable(tf.truncated_normal([hidden_nodes2, num_labels]))
    biases = tf.Variable(tf.zeros([num_labels]))

    # Training computation.
    logits = tf.matmul(hidden_layer_drop2, weights) + biases
    loss = tf.reduce_mean(
      tf.nn.softmax_cross_entropy_with_logits(labels=tf_train_labels, logits=logits)\
      + beta * tf.nn.l2_loss(weights))

    # Optimizer.
    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)
    #optimizer = tf.train.GradientDescentOptimizer(0.5).minimize(loss)

    # Predictions for the training, validation, and test data.
    train_prediction = tf.nn.softmax(logits)
    valid_relu = tf.nn.relu(  tf.matmul(tf_valid_dataset, hidden_weights) + hidden_biases)
    valid_relu = tf.nn.relu(  tf.matmul(valid_relu, hidden_weights1) + hidden_biases1)
    valid_relu = tf.nn.relu(  tf.matmul(valid_relu, hidden_weights2) + hidden_biases2)
    valid_prediction = tf.nn.softmax(
      tf.matmul(valid_relu, weights) + biases)
    test_relu = tf.nn.relu( tf.matmul( tf_test_dataset, hidden_weights) + hidden_biases)
    test_relu = tf.nn.relu( tf.matmul( test_relu, hidden_weights1) + hidden_biases1)
    test_relu = tf.nn.relu( tf.matmul( test_relu, hidden_weights2) + hidden_biases2)
    test_prediction = tf.nn.softmax(tf.matmul(test_relu, weights) + biases)
```

In [112]:
```
# tune keep prod
num_steps = 6001
keep_prob_rate = 1.

with tf.Session(graph=graph) as session:
  tf.global_variables_initializer().run()
  print("Initialized")
  for step in range(num_steps):
    # Pick an offset within the training data, which has been randomized.
    # Note: we could use better randomization across epochs.
    offset = (step * batch_size) % (train_labels.shape[0] - batch_size)
    # Generate a minibatch.
    batch_data = train_dataset[offset:(offset + batch_size), :]
    batch_labels = train_labels[offset:(offset + batch_size), :]
    # Prepare a dictionary telling the session where to feed the minibatch.
    # The key of the dictionary is the placeholder node of the graph to be fed,
    # and the value is the numpy array to feed to it.
    feed_dict = {tf_train_dataset : batch_data, tf_train_labels : batch_labels,
                 keep_prob: keep_prob_rate,
                starter_learning_rate: 0.0001}
    _, l, predictions = session.run(
      [optimizer, loss, train_prediction], feed_dict=feed_dict)
    if (step % 500 == 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: 83174.703125
Minibatch accuracy: 12.5%
Validation accuracy: 14.2%
Minibatch loss at step 500: 2997.090820
Minibatch accuracy: 82.0%
Validation accuracy: 77.7%
Minibatch loss at step 1000: 2294.321045
Minibatch accuracy: 80.5%
Validation accuracy: 78.1%
Minibatch loss at step 1500: 2070.109131
Minibatch accuracy: 79.7%
```

```
Validation accuracy: 78.0%
Minibatch loss at step 2000: 978.247681
Minibatch accuracy: 78.1%
Validation accuracy: 78.1%
Minibatch loss at step 2500: 709.178284
Minibatch accuracy: 75.8%
Validation accuracy: 76.9%
Minibatch loss at step 3000: 328.203857
Minibatch accuracy: 78.9%
Validation accuracy: 75.0%
Minibatch loss at step 3500: 367.792694
Minibatch accuracy: 76.6%
Validation accuracy: 74.0%
Minibatch loss at step 4000: 269.953064
Minibatch accuracy: 71.1%
Validation accuracy: 72.4%
Minibatch loss at step 4500: 240.408539
Minibatch accuracy: 69.5%
Validation accuracy: 72.1%
Minibatch loss at step 5000: 83.369095
Minibatch accuracy: 75.8%
Validation accuracy: 71.5%
Minibatch loss at step 5500: 106.328583
Minibatch accuracy: 67.2%
Validation accuracy: 69.9%
Minibatch loss at step 6000: 84.193954
Minibatch accuracy: 75.0%
Validation accuracy: 67.4%
Test accuracy: 74.3%
```

In [ ]:

```
# with more layers, the starter learning rate should be more smaller.
# otherwise, at the beginning the learning rate will be fast. but then the learning rate will be very slow.
```