Assignment 2

Previously in `1_notmnist.ipynb`, we created a pickle with formatted datasets for training, development and testing on the notMNIST dataset.

The goal of this assignment is to progressively train deeper and more accurate models using TensorFlow.

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
```

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 0 to [1.0, 0.0, 0.0 ...], 1 to [0.0, 1.0, 0.0 ...]
  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)
```

We're first going to train a multinomial logistic regression using simple gradient descent.

TensorFlow works like this:

- First you describe the computation that you want to see performed: what the inputs, the variables, and the operations look like. These get created as nodes over a computation graph. This description is all contained within the block below:
  - `with graph.as_default():...`
- Then you can run the operations on this graph as many times as you want by calling `session.run()`, providing it outputs to fetch from the graph that get returned. This runtime operation is all contained in the block below:
  - `with tf.Session(graph=graph) as session: ...`

Let's load all the data into TensorFlow and build the computation graph corresponding to our training:

In [4]:

```
# With gradient descent training, even this much data is prohibitive.
# Subset the training data for faster turnaround.
train_subset = 10000

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

  # Input data.
  # Load the training, validation and test data into constants that are
  # attached to the graph.
  tf_train_dataset = tf.constant(train_dataset[:train_subset, :])
  tf_train_labels = tf.constant(train_labels[:train_subset])
  tf_valid_dataset = tf.constant(valid_dataset)
  tf_test_dataset = tf.constant(test_dataset)

  # Variables.
  # These are the parameters that we are going to be training. The weight
  # matrix will be initialized using random values following a (truncated)
  # normal distribution. The biases get initialized to zero.
  weights = tf.Variable(
    tf.truncated_normal([image_size * image_size, num_labels]))
  biases = tf.Variable(tf.zeros([num_labels]))

  # Training computation.
  # We multiply the inputs with the weight matrix, and add biases. We compute
  # the softmax and cross-entropy (it's one operation in TensorFlow, because
  # it's very common, and it can be optimized). We take the average of this
  # cross-entropy across all training examples: that's our loss.
  logits = tf.matmul(tf_train_dataset, weights) + biases
  loss = tf.reduce_mean(
    tf.nn.softmax_cross_entropy_with_logits(logits, tf_train_labels))
```

```
  # Optimizer.
  # We are going to find the minimum of this loss using gradient descent.
  optimizer = tf.train.GradientDescentOptimizer(0.5).minimize(loss)

  # Predictions for the training, validation, and test data.
  # These are not part of training, but merely here so that we can report
  # accuracy figures as we train.
  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)
```

Let's run this computation and iterate:

In [5]:

```
num_steps = 801

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

with tf.Session(graph=graph) as session:
  # This is a one-time operation which ensures the parameters get initialized
as
  # we described in the graph: random weights for the matrix, zeros for the
  # biases.
  tf.initialize_all_variables().run()
  print('Initialized')
  for step in range(num_steps):
    # Run the computations. We tell .run() that we want to run the optimizer,
    # and get the loss value and the training predictions returned as numpy
    # arrays.
    _, l, predictions = session.run([optimizer, loss, train_prediction])
    if (step % 100 == 0):
      print('Loss at step %d: %f' % (step, l))
      print('Training accuracy: %.1f%%' % accuracy(
        predictions, train_labels[:train_subset, :]))
      # Calling .eval() on valid_prediction is basically like calling run(),
but
      # just to get that one numpy array. Note that it recomputes all its graph
      # dependencies.
      print('Validation accuracy: %.1f%%' % accuracy(
        valid_prediction.eval(), valid_labels))
  print('Test accuracy: %.1f%%' % accuracy(test_prediction.eval(),
test_labels))
```

```
Initialized
Loss at step 0: 16.899242
Training accuracy: 9.2%
Validation accuracy: 13.2%
Loss at step 100: 2.236322  ...  ...
Training accuracy: 79.0%
Validation accuracy: 75.2%
Test accuracy: 82.3%
```

Let's now switch to stochastic gradient descent training instead, which is much faster.

The graph will be similar, except that instead of holding all the training data into a constant node, we create a `Placeholder`node which will be fed actual data at every call of `session.run()`.

In [7]:

```
batch_size = 128

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(logits, tf_train_labels))

  # 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)
```

Let's run it:

In [8]:

```
num_steps = 3001

with tf.Session(graph=graph) as session:
  tf.initialize_all_variables().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: 16.206078
Minibatch accuracy: 9.4%
Validation accuracy: 15.2%
Minibatch loss at step 500: 1.855653
……
Minibatch loss at step 3000: 1.316785
Minibatch accuracy: 75.8%
Validation accuracy: 78.1%
Test accuracy: 85.4%
```

Problem

Turn the logistic regression example with SGD into a 1-hidden layer neural network with rectified linear units nn.relu() and 1024 hidden nodes. This model should improve your validation / test accuracy.

In [1]:

```
#for usual dataset
#some importing
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

#loading data
pickle_file =
'/home/maxkhk/Documents/Udacity/DeepLearningCourse/SourceCode/tensorflow/exampl
es/udacity/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)

#prepare data to have right format for tensorflow
#i.e. data is flat matrix, labels are onehot

image_size = 28
num_labels = 10

def reformat(dataset, labels):
  dataset = dataset.reshape((-1, image_size * image_size)).astype(np.float32)
  # Map 0 to [1.0, 0.0, 0.0 ...], 1 to [0.0, 1.0, 0.0 ...]
  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)

#now is the interesting part - we are building a network with
#one hidden ReLU layer and out usual output linear layer

#we are going to use SGD so here is our size of batch
batch_size = 128

#building tensorflow graph
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)

  #now let's build our new hidden layer
  #that's how many hidden neurons we want
  num_hidden_neurons = 1024
  #its weights
  hidden_weights = tf.Variable(
    tf.truncated_normal([image_size * image_size, num_hidden_neurons]))
  hidden_biases = tf.Variable(tf.zeros([num_hidden_neurons]))

  #now the layer itself. It multiplies data by weights, adds biases
  #and takes ReLU over result
  hidden_layer = tf.nn.relu(tf.matmul(tf_train_dataset, hidden_weights) +
hidden_biases)
```

```
  #time to go for output linear layer
  #out weights connect hidden neurons to output labels
  #biases are added to output labels
  out_weights = tf.Variable(
    tf.truncated_normal([num_hidden_neurons, num_labels]))

  out_biases = tf.Variable(tf.zeros([num_labels]))

  #compute output
  out_layer = tf.matmul(hidden_layer,out_weights) + out_biases
  #our real output is a softmax of prior result
  #and we also compute its cross-entropy to get our loss
  loss = tf.reduce_mean( tf.nn.softmax_cross_entropy_with_logits(out_layer,
tf_train_labels))

  #now we just minimize this loss to actually train the network
  optimizer = tf.train.GradientDescentOptimizer(0.5).minimize(loss)

  #nice, now let's calculate the predictions on each dataset for evaluating the
  #performance so far
  # Predictions for the training, validation, and test data.
  train_prediction = tf.nn.softmax(out_layer)
  valid_relu = tf.nn.relu(  tf.matmul(tf_valid_dataset, hidden_weights) +
hidden_biases)
  valid_prediction = tf.nn.softmax( tf.matmul(valid_relu, out_weights) +
out_biases)

  test_relu = tf.nn.relu( tf.matmul( tf_test_dataset, hidden_weights) +
hidden_biases)
  test_prediction = tf.nn.softmax(tf.matmul(test_relu, out_weights) +
out_biases)

#now is the actual training on the ANN we built
#we will run it for some number of steps and evaluate the progress after
#every 500 steps

#function to evaluate accuracy of ANN
def accuracy(predictions, labels):
  return (100.0 * np.sum(np.argmax(predictions, 1) == np.argmax(labels, 1))
          / predictions.shape[0])

#number of steps we will train our ANN
num_steps = 3001

#actual training
with tf.Session(graph=graph) as session:
  tf.initialize_all_variables().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))
```

```
Training set (200000, 28, 28) (200000,)
Validation set (10000, 28, 28) (10000,)
Test set (10000, 28, 28) (10000,)
Training set (200000, 784) (200000, 10)
Validation set (10000, 784) (10000, 10)
Test set (10000, 784) (10000, 10)
Initialized
Minibatch loss at step 0: 332.658447
Minibatch accuracy: 7.0%
Validation accuracy: 30.2%
Test accuracy: 31.8%
Minibatch loss at step 500: 24.117279
Minibatch accuracy: 77.3%
Validation accuracy: 80.7%
Test accuracy: 87.7%
…….
Minibatch loss at step 3000: 7.710226
Minibatch accuracy: 82.0%
Validation accuracy: 81.2%
Test accuracy: 88.4%
```

In [ ]:

```
#for sanizied dataset

#some importing
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

#loading data
pickle_file =
'/home/maxkhk/Documents/Udacity/DeepLearningCourse/SourceCode/tensorflow/exampl
es/udacity/notMNIST_Sanitized.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)

#prepare data to have right format for tensorflow
#i.e. data is flat matrix, labels are onehot

image_size = 28
num_labels = 10

def reformat(dataset, labels):
  dataset = dataset.reshape((-1, image_size * image_size)).astype(np.float32)
  # Map 0 to [1.0, 0.0, 0.0 ...], 1 to [0.0, 1.0, 0.0 ...]
  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)

#now is the interesting part - we are building a network with
#one hidden ReLU layer and out usual output linear layer

#we are going to use SGD so here is our size of batch
batch_size = 128

#building tensorflow graph
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)

  #now let's build our new hidden layer
  #that's how many hidden neurons we want
```

```
  num_hidden_neurons = 1024
  #its weights
  hidden_weights = tf.Variable(
    tf.truncated_normal([image_size * image_size, num_hidden_neurons]))
  hidden_biases = tf.Variable(tf.zeros([num_hidden_neurons]))

  #now the layer itself. It multiplies data by weights, adds biases
  #and takes ReLU over result
  hidden_layer = tf.nn.relu(tf.matmul(tf_train_dataset, hidden_weights) +
hidden_biases)

  #time to go for output linear layer
  #out weights connect hidden neurons to output labels
  #biases are added to output labels
  out_weights = tf.Variable(
    tf.truncated_normal([num_hidden_neurons, num_labels]))

  out_biases = tf.Variable(tf.zeros([num_labels]))

  #compute output
  out_layer = tf.matmul(hidden_layer,out_weights) + out_biases
  #our real output is a softmax of prior result
  #and we also compute its cross-entropy to get our loss
  loss = tf.reduce_mean( tf.nn.softmax_cross_entropy_with_logits(out_layer,
tf_train_labels))

  #now we just minimize this loss to actually train the network
  optimizer = tf.train.GradientDescentOptimizer(0.5).minimize(loss)

  #nice, now let's calculate the predictions on each dataset for evaluating the
  #performance so far
  # Predictions for the training, validation, and test data.
  train_prediction = tf.nn.softmax(out_layer)
  valid_relu = tf.nn.relu(  tf.matmul(tf_valid_dataset, hidden_weights) +
hidden_biases)
  valid_prediction = tf.nn.softmax( tf.matmul(valid_relu, out_weights) +
out_biases)

  test_relu = tf.nn.relu( tf.matmul( tf_test_dataset, hidden_weights) +
hidden_biases)
  test_prediction = tf.nn.softmax(tf.matmul(test_relu, out_weights) +
out_biases)

#now is the actual training on the ANN we built
#we will run it for some number of steps and evaluate the progress after
#every 500 steps

#function to evaluate accuracy of ANN
def accuracy(predictions, labels):
  return (100.0 * np.sum(np.argmax(predictions, 1) == np.argmax(labels, 1))
          / predictions.shape[0])
```

```
#number of steps we will train our ANN
num_steps = 3001

#actual training
with tf.Session(graph=graph) as session:
  tf.initialize_all_variables().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))
```

```
Training set (184692, 28, 28) (184692,)
Validation set (10000, 28, 28) (10000,)
Test set (10000, 28, 28) (10000,)
Training set (184692, 784) (184692, 10)
Validation set (10000, 784) (10000, 10)
Test set (10000, 784) (10000, 10)
Initialized
Minibatch loss at step 0: 352.903442
Minibatch accuracy: 12.5%
Validation accuracy: 31.7%
Test accuracy: 33.1%
Minibatch loss at step 500: 21.723553
Minibatch accuracy: 77.3%
Validation accuracy: 79.6%
Test accuracy: 79.7%
……
Test accuracy: 82.0%
Minibatch loss at step 2000: 1.554712
Minibatch accuracy: 85.9%
Validation accuracy: 81.6%
Test accuracy: 81.8%
Minibatch loss at step 2500: 3.339332
Minibatch accuracy: 84.4%
Validation accuracy: 81.9%
Test accuracy: 81.9%
```