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 [2]:

```
# 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 *notmist.ipynb*.

In [3]:

```
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 [4]:

```
image_size = 28
num_labels = 10

def reformat(dataset, labels):
  dataset = dataset.reshape((-1, image_size * image_size)).astype(np.float32)
  # Map 2 to [0.0, 1.0, 0.0 ...], 3 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 [5]:

```
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 [13]:

```
#for logistic regression model code is below
#We will use SGD for training to save our time. Code is from Assignment 2
#beta is the new parameter - controls level of regularization. Default is 0.01
#but feel free to play with it
#notice, we introduce L2 for both biases and weights

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
  #HERE IS THE MOMENT
  #where actually add the L2 loss to our dataset
  #we first compute the loss as before and than we add the l2 norm
  loss = tf.reduce_mean(
    tf.nn.softmax_cross_entropy_with_logits(logits, tf_train_labels) + beta *
tf.nn.l2_loss(weights) + beta * tf.nn.l2_loss(biases))
  # 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)

#now run it and check the results (probably compare it to assignment2 results?)
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: 48.056805
Minibatch accuracy: 11.7%
Validation accuracy: 11.5%
Test accuracy: 11.5%
Minibatch loss at step 500: 1.106451
Minibatch accuracy: 76.6%
Validation accuracy: 80.4%
Test accuracy: 87.7%
……
Test accuracy: 87.4%
Minibatch loss at step 3000: 0.893538
Minibatch accuracy: 82.0%
Validation accuracy: 80.1%
Test accuracy: 87.3%
```

In [20]:

```
#for NeuralNetwork model code is below
```

```
#We will use SGD for training to save our time. Code is from Assignment 2
#beta is the new parameter - controls level of regularization.
#Feel free to play with it - the best one I found is 0.001
#notice, we introduce L2 for both biases and weights of all layers

batch_size = 128
beta = 0.001

#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
  #Notice - we introduce our L2 here
  loss = (tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(
    out_layer, tf_train_labels) +
    beta*tf.nn.l2_loss(hidden_weights) +
    beta*tf.nn.l2_loss(hidden_biases) +
    beta*tf.nn.l2_loss(out_weights) +
```

```
    beta*tf.nn.l2_loss(out_biases)))

  #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

#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))
```

```
Initialized
Minibatch loss at step 0: 582.600708
```

```
Minibatch accuracy: 14.1%
Validation accuracy: 30.5%
Test accuracy: 33.2%
Minibatch loss at step 500: 194.931396
Minibatch accuracy: 78.9%
Validation accuracy: 80.3%
Test accuracy: 87.9%
……
Minibatch loss at step 3000: 15.478810
Minibatch accuracy: 89.8%
Validation accuracy: 86.0%
Test accuracy: 92.4%
```

Problem 2

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

In [22]:

```
#ANN with same architecture as above
#This time we still use the L2 but restrict training dataset
#to be extremely small

#notice that despite L2 regularization performance on validation dataset
#is still not good

#get just first 500 of examples, so that our ANN can memorize whole dataset
train_dataset_2 = train_dataset[:500, :]
train_labels_2 = train_labels[:500]

batch_size = 128
beta = 0.001

#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
  #Notice - we introduce our L2 here
  loss = (tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(
    out_layer, tf_train_labels) +
    beta*tf.nn.l2_loss(hidden_weights) +
    beta*tf.nn.l2_loss(hidden_biases) +
    beta*tf.nn.l2_loss(out_weights) +
    beta*tf.nn.l2_loss(out_biases)))

  #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

#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_2.shape[0] - batch_size)
    # Generate a minibatch.
    batch_data = train_dataset_2[offset:(offset + batch_size), :]
    batch_labels = train_labels_2[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: 718.793457
Minibatch accuracy: 6.2%
Validation accuracy: 37.6%
Test accuracy: 40.6%
Minibatch loss at step 500: 191.865036
Minibatch accuracy: 99.2%
Validation accuracy: 75.1%
Test accuracy: 81.5%
……
Minibatch loss at step 3000: 15.596663
Minibatch accuracy: 100.0%
Validation accuracy: 76.7%
Test accuracy: 83.6%
```

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 [6]:

```
#ANN with introduced dropout
#This time we still use the L2 but restrict training dataset
#to be extremely small

#notice that despite L2 regularization performance on validation dataset
#is still not good

#get just first 500 of examples, so that our ANN can memorize whole dataset
train_dataset_2 = train_dataset[:500, :]
train_labels_2 = train_labels[:500]
```

```
#batch size for SGD and beta parameter for L2 loss
batch_size = 128
beta = 0.001

#that's how many hidden neurons we want
num_hidden_neurons = 1024

#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
  #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)

  #add dropout on hidden layer
  #we pick up the probabylity of switching off the activation
  #and perform the switch off of the activations
  keep_prob = tf.placeholder("float")
  hidden_layer_drop = tf.nn.dropout(hidden_layer, keep_prob)

  #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
  #notice that upon training we use the switched off activations
  #i.e. the variaction of hidden_layer with the dropout active
  out_layer = tf.matmul(hidden_layer_drop,out_weights) + out_biases
  #our real output is a softmax of prior result
  #and we also compute its cross-entropy to get our loss
  #Notice - we introduce our L2 here
  loss = (tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(
    out_layer, tf_train_labels) +
```

```
    beta*tf.nn.l2_loss(hidden_weights) +
    beta*tf.nn.l2_loss(hidden_biases) +
    beta*tf.nn.l2_loss(out_weights) +
    beta*tf.nn.l2_loss(out_biases)))

  #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

#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_2.shape[0] - batch_size)
    # Generate a minibatch.
    batch_data = train_dataset_2[offset:(offset + batch_size), :]
    batch_labels = train_labels_2[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 : 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: 857.903442
Minibatch accuracy: 4.7%
Validation accuracy: 33.2%
Test accuracy: 36.0%
Minibatch loss at step 500: 196.597656
Minibatch accuracy: 99.2%
Validation accuracy: 76.8%
Test accuracy: 83.3%
……
Minibatch loss at step 3000: 15.772614
Minibatch accuracy: 100.0%
Validation accuracy: 77.8%
Test accuracy: 85.1%
```

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%.

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(lo
ss, global_step=global_step)
```

In [6]:

```
#We try the following - 2 ReLU layers
#Dropout on both of them
#Also L2 regularization on them
#and learning rate decay also

#batch size for SGD
batch_size = 128
#beta parameter for L2 loss
beta = 0.001

#that's how many hidden neurons we want
num_hidden_neurons = 1024

#learning rate decay
#starting value, number of steps decay is performed,
#size of the decay
```

```
start_learning_rate = 0.005
decay_steps = 1000
decay_size = 0.95

#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 first hidden layer
  #its weights
  hidden_weights_1 = tf.Variable(
    tf.truncated_normal([image_size * image_size, num_hidden_neurons]))
  hidden_biases_1 = tf.Variable(tf.zeros([num_hidden_neurons]))

  #now the layer 1 itself. It multiplies data by weights, adds biases
  #and takes ReLU over result
  hidden_layer_1 = tf.nn.relu(tf.matmul(tf_train_dataset, hidden_weights_1) +
hidden_biases_1)

  #add dropout on hidden layer 1
  #we pick up the probabylity of switching off the activation
  #and perform the switch off of the activations
  keep_prob = tf.placeholder("float")
  hidden_layer_drop_1 = tf.nn.dropout(hidden_layer_1, keep_prob)

  #now let's build our second hidden layer
  #its weights
  hidden_weights_2 = tf.Variable(
    tf.truncated_normal([num_hidden_neurons, num_hidden_neurons]))
  hidden_biases_2 = tf.Variable(tf.zeros([num_hidden_neurons]))

  #now the layer 2 itself. It multiplies data by weights, adds biases
  #and takes ReLU over result
  hidden_layer_2 = tf.nn.relu(tf.matmul(hidden_layer_drop_1, hidden_weights_2)
+ hidden_biases_2)

  #add dropout on hidden layer 2
  #we pick up the probabylity of switching off the activation
  #and perform the switch off of the activations
  hidden_layer_drop_2 = tf.nn.dropout(hidden_layer_2, keep_prob)

  #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
  #notice that upon training we use the switched off activations
  #i.e. the variaction of hidden_layer with the dropout active
  out_layer = tf.matmul(hidden_layer_drop_2,out_weights) + out_biases
  #our real output is a softmax of prior result
  #and we also compute its cross-entropy to get our loss
  #Notice - we introduce our L2 here
  loss = (tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(
    out_layer, tf_train_labels) +
    beta*tf.nn.l2_loss(hidden_weights_1) +
    beta*tf.nn.l2_loss(hidden_biases_1) +
    beta*tf.nn.l2_loss(hidden_weights_2) +
    beta*tf.nn.l2_loss(hidden_biases_2) +
    beta*tf.nn.l2_loss(out_weights) +
    beta*tf.nn.l2_loss(out_biases)))

  #variable to count number of steps taken
  global_step = tf.Variable(0)

  #compute current learning rate
  learning_rate = tf.train.exponential_decay(start_learning_rate, global_step,
decay_steps, decay_size)
  #use it in optimizer
  optimizer = tf.train.GradientDescentOptimizer(learning_rate).minimize(loss,
global_step=global_step)

  #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_1 = tf.nn.relu(  tf.matmul(tf_valid_dataset, hidden_weights_1) +
hidden_biases_1)
  valid_relu_2 = tf.nn.relu(  tf.matmul(valid_relu_1, hidden_weights_2) +
hidden_biases_2)
  valid_prediction = tf.nn.softmax( tf.matmul(valid_relu_2, out_weights) +
out_biases)

  test_relu_1 = tf.nn.relu( tf.matmul( tf_test_dataset, hidden_weights_1) +
hidden_biases_1)
  test_relu_2 = tf.nn.relu( tf.matmul( test_relu_1, hidden_weights_2) +
hidden_biases_2)
  test_prediction = tf.nn.softmax(tf.matmul(test_relu_2, 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
```

```
#number of steps we will train our ANN
num_steps = 6001

#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, 
keep_prob : 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: 13478.936523
Minibatch accuracy: 10.9%
Validation accuracy: 23.6%
Test accuracy: 25.4%
Minibatch loss at step 500: 1335.539062
Minibatch accuracy: 65.6%
Validation accuracy: 82.2%
Test accuracy: 88.7%
…..
Minibatch loss at step 6000: 705.214050
Minibatch accuracy: 60.9%
Validation accuracy: 79.7%
Test accuracy: 86.6%
```

In [ ]: