COGS 181 - Homework 5

```
from sklearn.datasets import fetch_mldata
mnist = fetch_mldata("MNIST original")
from sklearn.neural_network import MLPClassifier
from sklearn import preprocessing
import numpy as np
import matplotlib.pyplot as plt
import tensorflow as tf
from time import time
from tensorflow.contrib import rnn
```

```
# Concatinate data to shuffle
data = np.column_stack((mnist['data'],mnist['target']))
assert data.shape == (70000, 785)
np.random.shuffle(data)
```

```
# One hot encode labels
y = np.matrix(data[:,-1]).T
oneHotEncoder = preprocessing.OneHotEncoder()
oneHotEncoder.fit(y)
y = oneHotEncoder.transform(y).toarray()
assert y.shape[1] == 10
```

```
x = data[:,:-1]
```

```
# Separate data
train_size = 60000
x_train = x[:train_size, :]
x_test = x[train_size:, :]
y_train = y[0:train_size]
y_test = y[train_size::]
```

Task 1

```
batch_size = 100
training_epochs = 100

display_step = 4
n_hidden_1 = 300 # Nodes in first hidden layer
n_input = 28*28 # Nodes in input layer
n_classes = 10 # Nodes in output layer
learning_rate = 0.1
```

```
X = tf.placeholder(tf.float32, [None, n_input])
Y = tf.placeholder(tf.float32, [None, n_classes])


weights = {
  'h1': tf.Variable(tf.random_normal([n_input, n_hidden_1])),
  'out': tf.Variable(tf.random_normal([n_hidden_1, n_classes]))
}

biases = {
  'b1': tf.Variable(tf.random_normal([n_hidden_1])),
  'out': tf.Variable(tf.random_normal([n_classes]))
}
```

```
def multilayer_perceptron(x):
  hidden_layer = tf.sigmoid(tf.add(tf.matmul(x, weights['h1']), biases['b1']))
  out_layer = tf.matmul(hidden_layer, weights['out']) + biases['out']
  return out_layer

logits = multilayer_perceptron(X)
```

```
loss_op = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(logits=logits, labe

train_step = tf.train.GradientDescentOptimizer(learning_rate).minimize(loss_op)
```

```
def plot_statistics(train_accs, test_accs, losses, batch_size, learning_rate):
  plt.title("batch_size={}\t learning_rate={}".format(batch_size, learning_rate))
  plt.xlabel("iterations")
  plt.ylabel("Accuracy")
  plt.plot(range(len(train_accs)), train_accs, label="Training accuracy")
  plt.plot(range(len(test_accs)), test_accs, label="Testing accuracy")
  plt.legend()
  plt.show()
  plt.xlabel("Iterations")
```

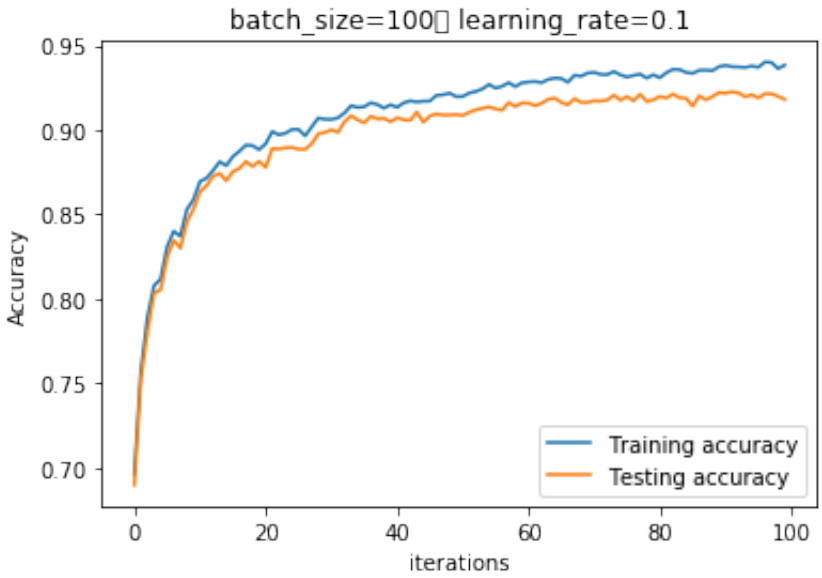
```
  plt.ylabel("Loss function")
  plt.title("batch_size={} \t learning_rate={}".format(batch_size, learning_rate))
  plt.plot(range(len(losses)), losses, label="Loss")
  plt.show()

def task1_sgd(batch_size, learning_rate, max_epochs, x_train, x_test, y_train, y_tes
  # Tracking variables
  t = time()
  train_accs = []
  test_accs = []
  losses = []
  init = tf.global_variables_initializer()
  total_batch_its = len(x_train) / batch_size
  with tf.Session() as sess:
    sess.run(init)

    for epoch in range(max_epochs):
      avg_cost = 0
      for i in range(total_batch_its):
        idx = np.random.choice(range(len(x_train)),batch_size)
        batch_x = x_train[idx]
        batch_y = y_train[idx]

        _, c = sess.run([train_step, loss_op], feed_dict={X:batch_x, Y: batch_y})

        avg_cost += c / total_batch_its

      # Track testing / training accuracy
      pred = tf.nn.softmax(logits)
      correct_pred = tf.equal(tf.argmax(pred,1), tf.argmax(Y,1))
      accuracy = tf.reduce_mean(tf.cast(correct_pred, "float"))
      test_acc = accuracy.eval({X: x_test, Y: y_test})
      train_acc = accuracy.eval({X: x_train, Y: y_train})
      test_accs.append(test_acc)
      train_accs.append(train_acc)
      losses.append(avg_cost)
      if epoch % display_step == 0:
        print "Epoch:", '%04d' % (epoch+1), "cost={:.9f}".format(avg_cost)

        print "Time used: {:10.4f} \t Accuracy test: {:0.4f}".format(time() -t, test
        t = time()


    pred = tf.nn.softmax(logits)
    correct_pred = tf.equal(tf.argmax(pred,1), tf.argmax(Y,1))
    accuracy = tf.reduce_mean(tf.cast(correct_pred, "float"))
    print "Accuracy:", accuracy.eval({X: x_test, Y: y_test})
    print "Optimization finished"
```

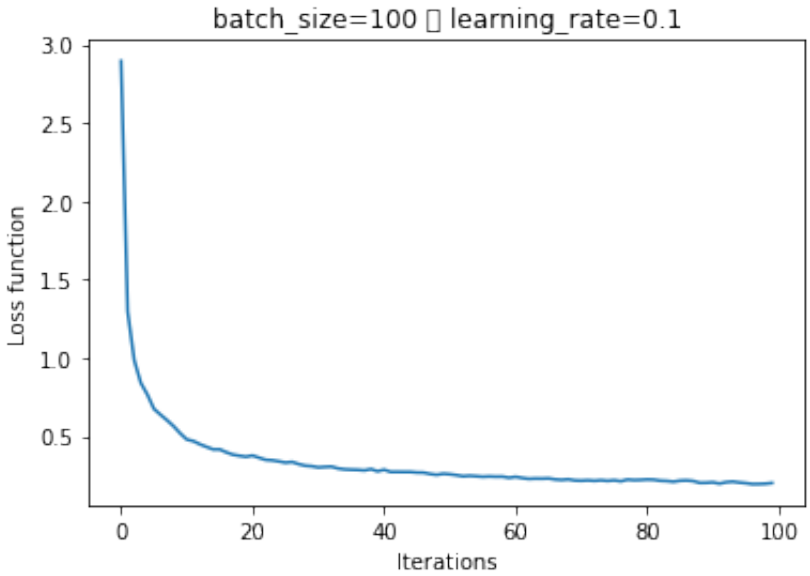
```
    return train_accs, test_accs, losses
```

```
lrate = 0.1
batch_size = 100
train_accs, test_accs, losses = task1_sgd(batch_size, lrate, 100, x_train, x_test, y
```

```
Accuracy: 0.9181
Optimization finished
```

```
plot_statistics(train_accs, test_accs, losses, batch_size, lrate)
```





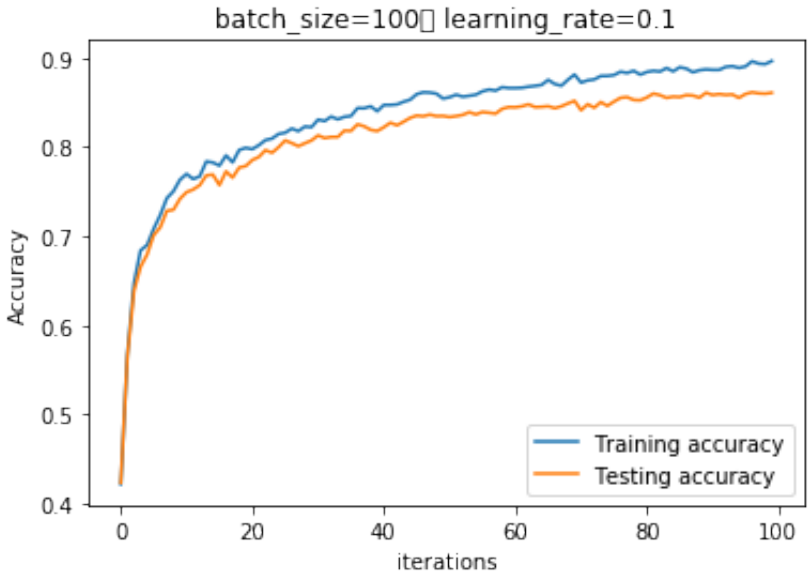
4) Compose training set

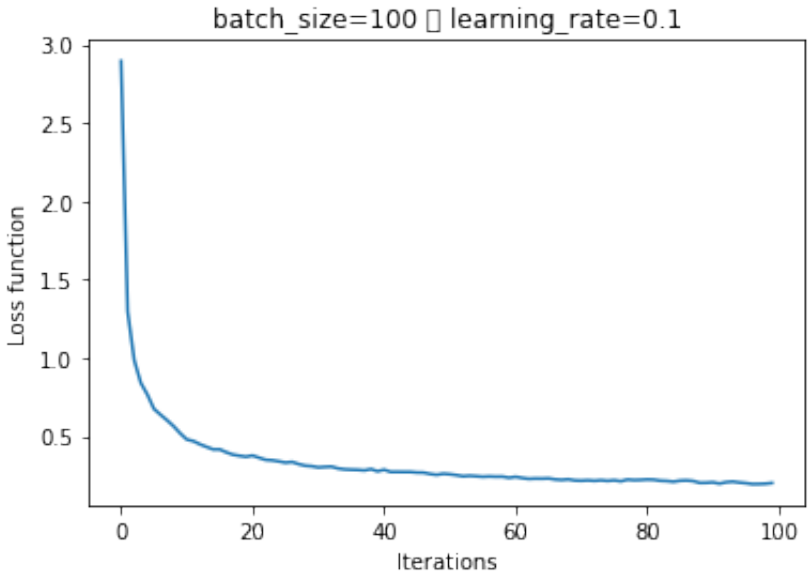
```
def generate_balanced_training_set(x,y,n):
  x_c = np.empty((0,n_input))
  y_c = np.empty((0,n_classes))
  for i in range(0,10):
    inds = np.where(y[:,i] == 1)[0]
    chosen = np.random.choice(inds,n)
    x_c = np.row_stack((x_c,x[chosen]))
    y_c = np.row_stack((y_c,y[chosen]))
  return x_c, y_c

x_t_4, y_t_4 = generate_balanced_training_set(x, y, 1000)
```

5) Training of #4

```
train_accs_4, test_accs_4, losses_4 = task1_sgd(batch_size, lrate, 100, x_t_4, x_tes
plot_statistics(train_accs_4, test_accs_4, losses, batch_size, lrate)
```

```
Accuracy: 0.8607
Optimization finished
```

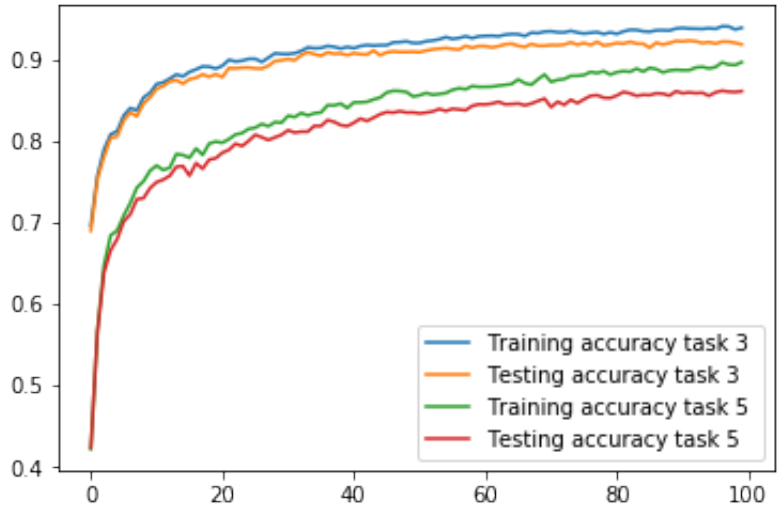




6) Comparison of 3 & 5

- We see that the gap between training and testing accuracy is higher for task 5 than for task 3. This is due that the small image set in task 5 gives us overfitting on the test set.
- The accuracy is overall higher for the larger training set, which makes sence.
- The learning rate is somewhat higher for the larger training set, and it converges faster in terms of epochs.

```
plt.plot(range(len(train_accs)), train_accs, label="Training accuracy task 3")
plt.plot(range(len(test_accs)), test_accs, label="Testing accuracy task 3")
plt.plot(range(len(train_accs_4)), train_accs_4, label="Training accuracy task 5 ")
plt.plot(range(len(test_accs_4)), test_accs_4, label="Testing accuracy task 5")
plt.legend()

plt.show()
```



2 - Convolutional Neural Networks

I mostly followed the tutorial to construct a convolutional neural network.

- The kernel patch is 5x5. For each convolution the resulting feature is half the size of input (28x28 -> 14x14 -> 7x7)
- The stride used is 1 in each direction
- Doing 2x2 max-pooling for each convolutional layer.
- Using an Adam optimizer with learning rate of $1 * 10^{-4}$

Network structure: - First layer is a convolutional network with 32 5x5 kernels, reducing image size to 14x14 - Second layer is a convolutional network with 5x5 kernels, reducing image size to 7x7 - Third layer is a densely connected layer with 1024 perceptrons with ReLU activation. - Fourth layer is a dropout layer to reduce overfitting. The probability to keep is 0.5 in training (1.0 in evaluation) - Last layer consists of 10 perceptrons and is the readout layer. These are fully connected to the droput layer.

Hyperparameters:

- batch_size: 1000
- learning rate: $1 * 10^{-4}$. The adam optimizer has a default value of β_1 which is 0.9 which sets the decay in the learning rate.
- Keeping probability on dropout layer: 0.5

```
# Initialize weight with a bit of noise for symmetry breaking
def weight_variable(shape):
  initial = tf.truncated_normal(shape, stddev=0.1)
  return tf.Variable(initial)

def bias_variable(shape):
  initial = tf.constant(0.1, shape=shape)
  return tf.Variable(initial)
```

```
def conv2d(x, W):
  return tf.nn.conv2d(x, W, strides=[1,1,1,1], padding="SAME")

def max_pool_2x2(x):
  return tf.nn.max_pool(x, ksize=[1,2,2,1],
                        strides = [1,2,2,1], padding="SAME")
```

```
def conv_nn():
    # First convolutional layer
    # 32 features per 5x5 patch, 1 input channel, 32 output
    W_conv1 = weight_variable([5,5,1,32])
    b_conv1 = bias_variable([32])

    # Reshape to a 4d tensor, [, width, height, color_channels]
    x_image = tf.reshape(X, [-1, 28, 28, 1])

    h_conv1 = tf.nn.relu(conv2d(x_image, W_conv1) + b_conv1)
    # Pool reduces image to 14x14
    h_pool1 = max_pool_2x2(h_conv1)


    # Second convolutional network
    # 64 features per 5x5 patch, 32 inputs, 64 outputs
    W_conv2 = weight_variable([5,5,32,64])
    b_conv2 = bias_variable([64])

    h_conv2 = tf.nn.relu(conv2d(h_pool1, W_conv2) + b_conv2)
    # Pool reduces image to 7x7
    h_pool2 = max_pool_2x2(h_conv2)


    # Densely connected layer
    # Layers with 1024 fully connected layers
    W_fc1 = weight_variable([7 * 7 * 64, 1024])
    b_fc1 = bias_variable([1024])
    # Reshape to a into a batch of vectors
    h_pool2_flat = tf.reshape(h_pool2, [-1, 7*7*64])
    # Multiply by a weight matrix and apply ReLU
    h_fc1 = tf.nn.relu(tf.matmul(h_pool2_flat, W_fc1) + b_fc1)


    # DropoutLayer to reduce overfitting
    keep_prob = tf.placeholder(tf.float32) # Add possibility to turn dropout on / of
    h_fc1_drop = tf.nn.dropout(h_fc1, keep_prob)


    # Readout layer
    W_fc2 = weight_variable([1024, 10])
    b_fc2 = bias_variable([10])
    y_conv = tf.matmul(h_fc1_drop, W_fc2) + b_fc2
    return y_conv, keep_prob
```

```
batch_size = 1000
display_step = 10
training_epochs = 250
```

```
def cnn_train(logits, x_train, y_train, x_test, y_test, lrate=1e-4):
  # Tracking variables
  train_accs = []
  test_accs = []
  losses = []
  total_batch_its = len(x_train) / batch_size

  # Setup loss / optmization functions

  cross_entropy = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(labels=Y, l
  train_step = tf.train.AdamOptimizer(lrate).minimize(cross_entropy)
  #
  correct_pred = tf.equal(tf.argmax(logits, 1), tf.argmax(Y, 1))
  accuracy = tf.reduce_mean(tf.cast(correct_pred, tf.float32))

  with tf.Session() as sess:
    sess.run(tf.global_variables_initializer()  )

    for epoch in range(training_epochs):
      avg_cost = 0
      for i in range(total_batch_its):
        # Select batch
        idx = np.random.choice(range(len(x_train)),batch_size)
        batch_x = x_train[idx]
        batch_y = y_train[idx]
        #train_step.run
        _, c = sess.run([train_step, cross_entropy], feed_dict={X: batch_x, Y: batch

        avg_cost += c / total_batch_its

      # Track testing / training accuracy
      test_acc = accuracy.eval({X: x_test, Y: y_test, keep_prob: 1.0})
      train_acc = accuracy.eval({X: batch_x, Y: batch_y, keep_prob: 1.0})
      test_accs.append(test_acc)
      train_accs.append(train_acc)
      losses.append(avg_cost)
      if epoch % display_step == 0:
        print "Epoch:", '%04d' % (epoch+1), "cost={:.9f}".format(avg_cost)

        print "Accuracy test:", test_acc


    pred = tf.nn.softmax(logits)
```

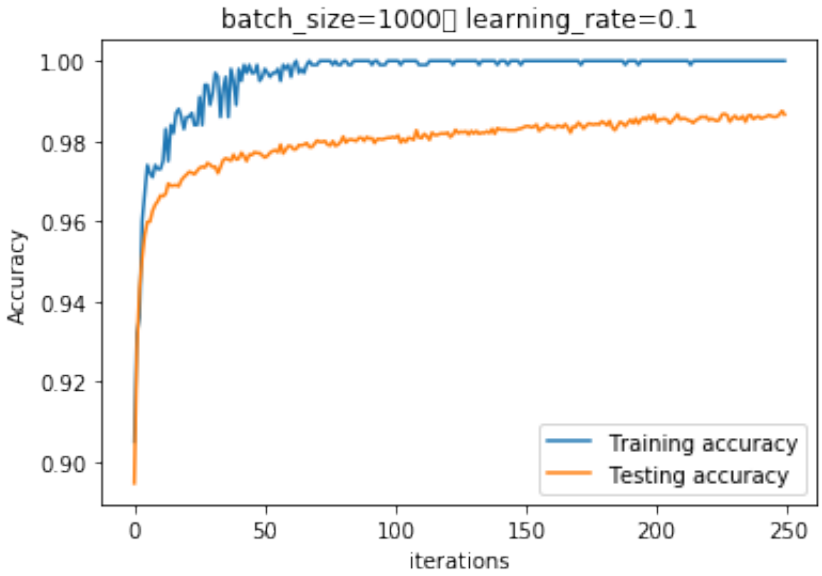
```
    correct_pred = tf.equal(tf.argmax(pred,1), tf.argmax(Y,1))
    accuracy = tf.reduce_mean(tf.cast(correct_pred, "float"))
    print "Accuracy:", accuracy.eval({X: x_test, Y: y_test, keep_prob: 1.0})
    print "Final loss:", losses[-1]
    print "Optimization finished"
  return train_accs, test_accs, losses
```

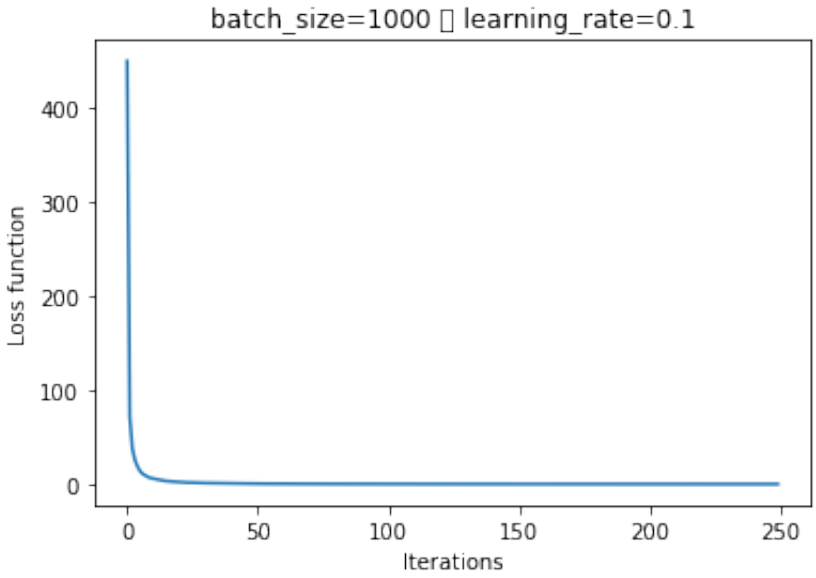
```
logits, keep_prob = conv_nn()
```

```
train_accs2_3, test_accs2_3, losses2_3 = cnn_train(logits, x_train, y_train, x_test,
```

```
Accuracy: 0.9866
Final loss: 0.0229855580967
Optimization finished
```

```
plot_statistics(train_accs2_3, test_accs2_3, losses2_3, batch_size, lrate)
```





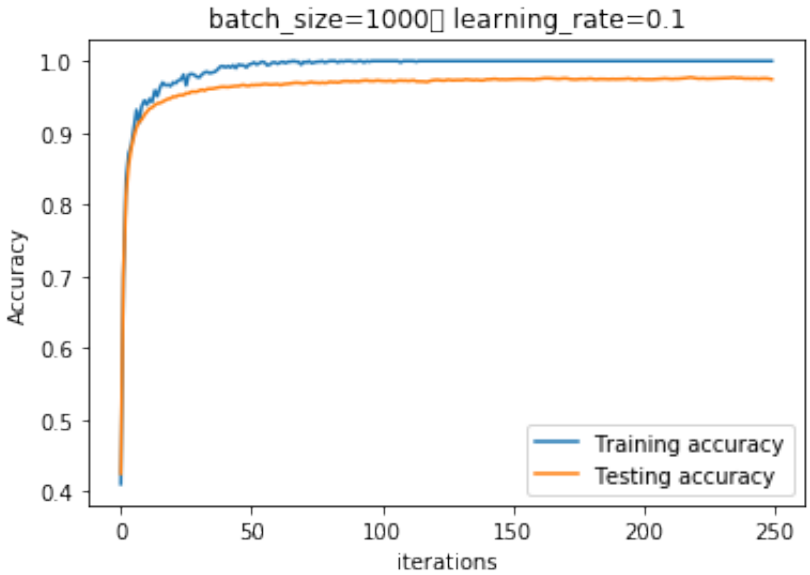
2.4 Training on smaller training set

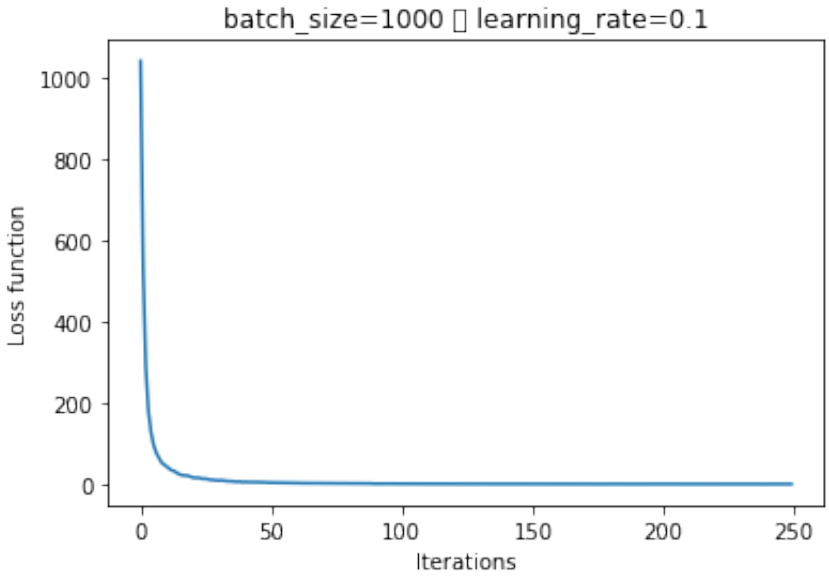
```
x_t_4, y_t_4 = generate_balanced_training_set(x, y, 1000)
```

```
train_accs2_4, test_accs2_4, losses2_4 = cnn_train(logits, x_t_4, y_t_4, x_test, y_t
```

```
Accuracy: 0.9746
Final loss: 0.158231183834
Optimization finished
```

```
plot_statistics(train_accs2_4, test_accs2_4, losses2_4, batch_size, lrate)
```

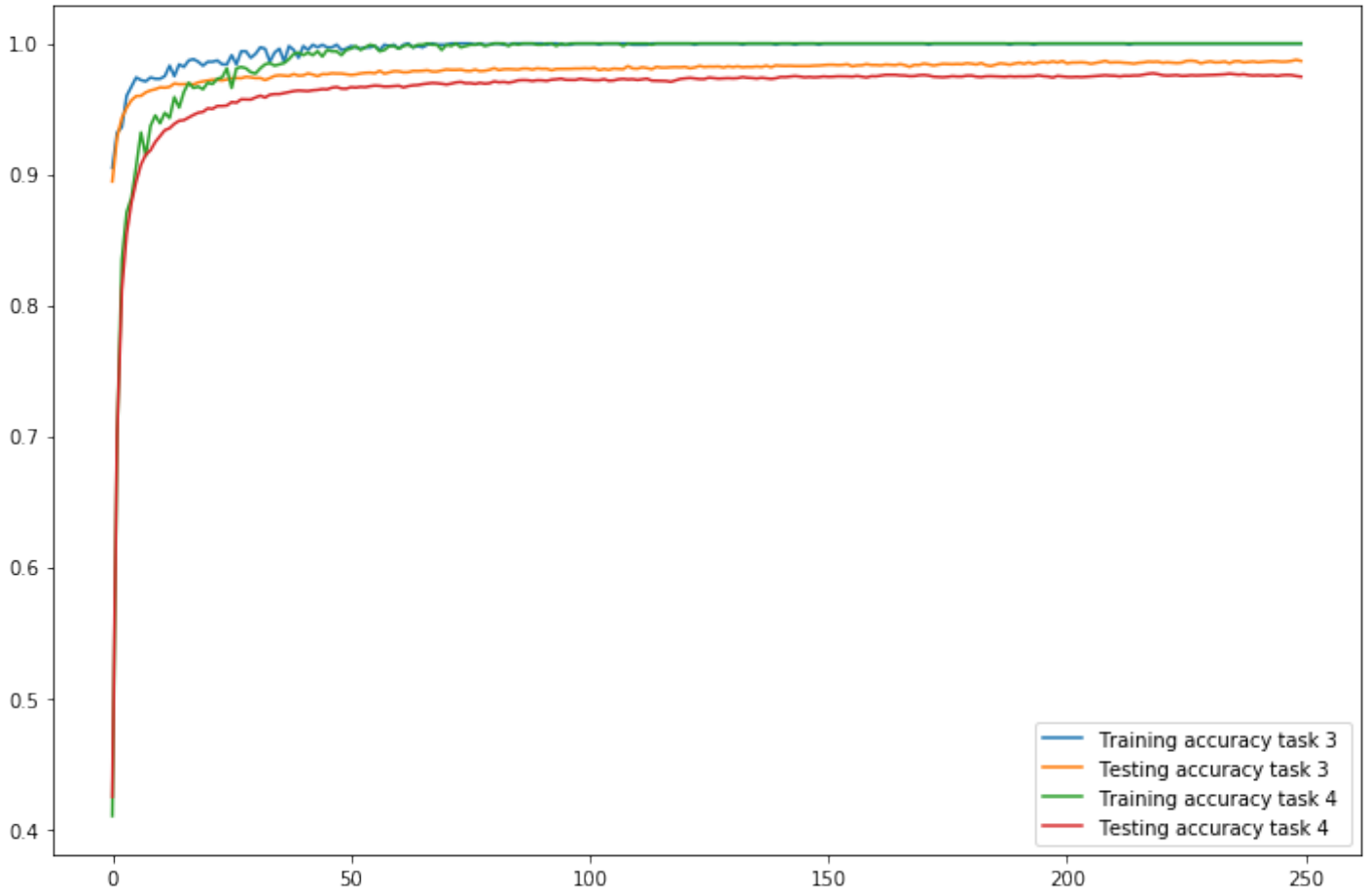




2.5 Comparison of #3 and #4

- We see that the gap between training and testing accuracy is higher for task 4 than for task 3. This is due that the small image set in task 5 gives us overfitting on the test set.
- The accuracy is overall higher for the larger training set, which makes sense.
- The learning rate is somewhat higher for the larger training set, and it converges faster in terms of epochs.
- Both training accuracies reach a perfect classifier for the training set.

```
plt.figure(figsize=(12,8))
plt.plot(range(len(train_accs2_3)), train_accs2_3, label="Training accuracy task 3")
plt.plot(range(len(test_accs2_3)), test_accs2_3, label="Testing accuracy task 3")
plt.plot(range(len(train_accs2_4)), train_accs2_4, label="Training accuracy task 4 "
plt.plot(range(len(test_accs2_4)), test_accs2_4, label="Testing accuracy task 4")
plt.legend()

plt.show()
```


1.0
0.9
0.8
0.7
0.6
0.5
0.4
0
50
100
150
200
250
Training accuracy task 3
Testing accuracy task 3
Training accuracy task 4
Testing accuracy task 4

```
import os
os.environ["CUDA_VISIBLE_DEVICES"]="0"
import sys
import tensorflow as tf
from tensorflow.contrib import rnn
import numpy as np
import matplotlib
import matplotlib.pyplot as plt
```

Generate Samples (to be filled)

This section generates some samples for sine function $y = F(t)$:

$$y = F(t) = \sin(2\pi f(t + t_0))$$

Here, in each batch, given a series of time points $t_1, t_2, \ldots, t_n$ ($n = n_{samples} + n_{predict}$), we want to generate a series of function values $y_1, y_2, \ldots, y_n$. We use t_0 as a randomizer to make each batch start from different time point (different phase). Then, we will collect first $n_{samples}$ function values $y_1, y_2, \ldots, y_{n_{samples}}$ as input vector $\mathbf{y}_{samples}$ and next $n_{predict}$ function values $y_{n_{samples}+1}, y_{n_{samples}+2}, \ldots, y_{n_{samples}+n_{predict}}$ as input vector $\mathbf{y}_{predict}$. Here we use `seq2seq` model, where $n_{predict} = 1$.

```
def generate_sample(f = 1.0, batch_size = 1,
                    predict = 1, samples = 100):
    """
    Generates data samples.
    :param f: The frequency to use for all time series.
    :param batch_size: The number of time series to generate.
    :param predict: The number of future samples to generate.
    :param samples: The number of past (and current) samples to generate.
    :return: Tuple that contains the past times and values as well as the future tim
             each row represents one time series of the batch.
    """
    const = 100.0

    # Empty batch vectors.
    T = np.empty((batch_size, samples))
    Y = np.empty((batch_size, samples))
    FT = np.empty((batch_size, predict))
    FY = np.empty((batch_size, predict))

    for i in range(batch_size):
        # We define the range of t here.
        t = np.arange(0, samples + predict) / const

        # Here we want to sample some points for sine function.
        t0 = np.random.choice(t,1)[0]
        y = np.sin(2*np.pi*f*(t+t0))

        T[i, :] = t[0:samples]+t0       # t_1 ... t_{n_samples}
        Y[i, :] = y[0:samples]        # y_1 ... y_{n_samples}

        FT[i, :] = t[samples:samples+predict] + t0       # t_{n_samples+1} ... t_{n_
        FY[i, :] = y[samples:samples+predict]       # y_{n_samples+1} ... y_{n_sampl

    return T, Y, FT, FY
```

RNN Model (to be filled)

This section builds a RNN model. The model is like:

$$\hat{\mathbf{y}}_{predict} = \tanh(Linear(RNN(\mathbf{y}_{samples})))$$

```
# State size = n_steps
def RNN(x, weights, biases, n_input, n_steps, n_hidden):
  # Prepare data shape to match `rnn` function requirements
  # Current data input shape: (batch_size, n_steps, n_input)
  # Required shape: 'n_steps' tensors list of shape (batch_size, n_input)

  # Permuting batch_size and n_steps
  x = tf.transpose(x, [1, 0, 2])
  # Reshaping to (n_steps*batch_size, n_input).
  x = tf.reshape(x, [-1, n_input])
  # Split to get a list of 'n_steps' tensors of shape (batch_size, n_input).
  x = tf.split(x, n_steps, axis=0)
  # Define a RNN cell with TensorFlow.
  rnn_cell = rnn.BasicLSTMCell(num_units=n_hidden, forget_bias=1.0)

  # Get RNN cell output.
  # Hint: Use rnn.static_rnn()
  outputs, states = rnn.static_rnn(rnn_cell, x, initial_state=None, dtype=tf.float32

  # Linear layer and tanh activation, using RNN inner loop last output.
  # Hint: Use tf.tanh, tf.nn.bias_add(), tf.matmul(), weights, biases.
  final_output = [tf.tanh(tf.nn.bias_add(tf.matmul(out, weights), biases)) for out i

  return final_output[-1]
```

Parameters and Network Building

This section sets the parameters and builds the network.

```
tf.reset_default_graph()
# Parameters
learning_rate = 0.001
training_iters = 50000
batch_size = 50
display_step = 100

# Network Parameters
n_input = 1     # Input is sin(x).
n_steps = 100   # Timesteps.
n_hidden = 100  # Hidden layer num of features.
n_outputs = 1   # Output is sin(x+1).

# tf Graph input
x = tf.placeholder("float", [None, n_steps, n_input])
y = tf.placeholder("float", [None, n_outputs])

# Define weights
weights = tf.Variable(tf.random_normal([n_hidden, n_outputs]))
biases = tf.Variable(tf.random_normal([n_outputs]))

pred = RNN(x, weights, biases, n_input, n_steps, n_hidden)

# Define loss (Euclidean distance) and optimizer.
individual_losses = tf.reduce_sum(tf.squared_difference(pred, y), reduction_indices=
loss = tf.reduce_mean(individual_losses)
optimizer = tf.train.AdamOptimizer(learning_rate=learning_rate).minimize(loss)

# Initializing the variables.
init = tf.global_variables_initializer()

# Set dynamic allocation of GPU memory rather than pre-allocation.
# Also set soft placement, which means when current GPU does not exist,
# it will change into another.
config = tf.ConfigProto(allow_soft_placement = True)
config.gpu_options.allow_growth = True
```

Network Training.

This section trains the network.

```
# Launch the graph.
sess = tf.Session(config=config)
sess.run(init)

step = 1
# Keep training until reach max iterations.
while step * batch_size < training_iters:
    _, batch_x, __, batch_y = generate_sample(f=1.0, batch_size=batch_size, samples=
                                              predict=n_outputs)

    batch_x = batch_x.reshape((batch_size, n_steps, n_input))
    batch_y = batch_y.reshape((batch_size, n_outputs))

    # Run optimization op (backprop).
    sess.run(optimizer, feed_dict={x: batch_x, y: batch_y})
    if step % display_step == 0:
        # Calculate batch loss.
        loss_value = sess.run(loss, feed_dict={x: batch_x, y: batch_y})
        print("Iter " + str(step * batch_size) + ", Minibatch Loss= " +
              "{:.6f}".format(loss_value))
    step += 1
print("Optimization Finished!")
```

```
Iter 5000, Minibatch Loss= 0.005657
Iter 10000, Minibatch Loss= 0.005032
Iter 15000, Minibatch Loss= 0.002268
Iter 20000, Minibatch Loss= 0.000544
Iter 25000, Minibatch Loss= 0.000430
Iter 30000, Minibatch Loss= 0.001009
Iter 35000, Minibatch Loss= 0.000277
Iter 40000, Minibatch Loss= 0.000251
Iter 45000, Minibatch Loss= 0.000253
Optimization Finished!
```

Network Evaluation and Visualization.

This section evaluates the network and gives a visualization of result.

** Attention ** You may be frustrated because your result is not like the given figure (though the given figure is also not that good). Do not worry to much. Your implementation may be correct and this `seq2seq` model is not that strong. You can re-run the notebook for a couple of times (`Kernel` -> `Restart & Run All`) and then select the best result image to report.

```
# Test the prediction.
t, start_y, next_t, expected_y = generate_sample(f=1, samples=n_steps, predict=50)

pred_y = []
y = start_y

for i in range(50):
    test_input = y.reshape((1, n_steps, n_input))
    prediction = sess.run(pred, feed_dict={x: test_input})
    prediction = prediction.squeeze()
    pred_y.append(prediction)
    y = np.append(y.squeeze()[1:], prediction)
```

```
# Remove the batch size dimensions.
t = t.squeeze()
start_y = start_y.squeeze()
next_t = next_t.squeeze()

plt.plot(t, start_y, color='black')
plt.plot(np.append(t[-1], next_t), np.append(start_y[-1], expected_y), color='green'
plt.plot(np.append(t[-1], next_t), np.append(start_y[-1], pred_y), color='red')
plt.ylim([-1, 1])
plt.xlabel('time [t]')
plt.ylabel('signal')

plt.show()
```