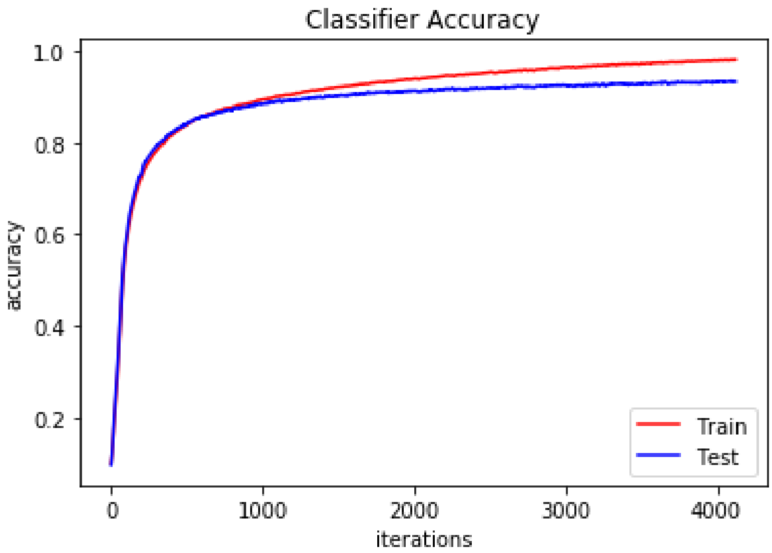
In [13]:

```
# Problem 1.1-1.3
# part of the code for Problem 1 is referenced from aymericdamien on Github
:
# https://github.com/aymericdamien/TensorFlow-Examples/blob/master/
# examples/3_NeuralNetworks/multilayer_perceptron.py

import tensorflow as tf

# Import MNIST data
from tensorflow.examples.tutorials.mnist import input_data
mnist = input_data.read_data_sets("/tmp/data/", one_hot=True)

# Parameters
learning_rate = 0.001
training_epochs = 15
batch_size = 200
display_step = 1

# Network Parameters
n_hidden = 300 # number of hidden layer units
n_input = 784 # MNIST image data input, 28*28
n_classes = 10 # MNIST total classes, 0-9 digits

# tf Graph input
X = tf.placeholder("float", [None, n_input])
Y = tf.placeholder("float", [None, n_classes])

# layers weight & bias
weights = {
    'h1': tf.Variable(tf.random_normal([n_input, n_hidden])),
    'out': tf.Variable(tf.random_normal([n_hidden, n_classes]))
}
biases = {
    'b1': tf.Variable(tf.random_normal([n_hidden])),
    'out': tf.Variable(tf.random_normal([n_classes]))
}

# Create model
def multilayer_perceptron(x):
    # Hidden fully connected layer with sigmoid activation function
    hidden_layer = tf.nn.sigmoid(tf.matmul(x, weights['h1']) + biases['b1']
)

    # Output fully connected layer
    out_layer = tf.matmul(hidden_layer, weights['out']) + biases['out']
    return out_layer

# Construct logits with model
logits = multilayer_perceptron(X)

# Define loss and optimizer
loss_op = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(logits=log
its, labels=Y))
train_op = tf.train.AdamOptimizer(learning_rate=learning_rate).minimize(los
s_op)
```

```
# prediction and accuracy nodes
pred = tf.nn.softmax(logits)
correct_prediction = tf.equal(tf.argmax(pred, 1), tf.argmax(Y, 1))
accuracy = tf.reduce_mean(tf.cast(correct_prediction, "float"))

# Initializing the variables
init = tf.global_variables_initializer()

# Create session for running nodes
sess = tf.Session()

sess.run(init)

train_accuracy = []
test_accuracy = []
#with tf.Session() as sess:
for epoch in range(training_epochs):

    # calculate total number of batches
    total_batch = int(mnist.train.num_examples/batch_size)

    # Loop over all batches
    for i in range(total_batch):
        batch_x, batch_y = mnist.train.next_batch(batch_size)

        # Run optimization op (backprop) and cost op (to get loss value)
        _, c = sess.run([train_op, loss_op], feed_dict={X: batch_x,
                                                        Y: batch_y})
        # Compute train accuracy
        train_accuracy.append(sess.run(accuracy, feed_dict={X: mnist.train.i
mages, Y: mnist.train.labels}))
        test_accuracy.append(sess.run(accuracy, feed_dict={X: mnist.test.ima
ges, Y: mnist.test.labels}))

print "Optimization Finished!"

# plot the accuracy
import matplotlib.pyplot as plt

plt.plot(train_accuracy, color = 'r', label = 'Train')
plt.plot(test_accuracy, color = 'b', label = 'Test')
plt.title('Classifier Accuracy')
plt.xlabel('iterations')
plt.ylabel('accuracy')
plt.legend(loc='lower right')
plt.show()

# model parameters and logistics
print "training_epochs: " + str(training_epochs)
print "batch_size: " + str(batch_size)
print "learning_rate: " + str(learning_rate)
print "Final Train Accuracy:" + str(sess.run(accuracy, feed_dict={X: mnist.
train.images, Y: mnist.train.labels}))
print "Final Test Accuracy:" + str(sess.run(accuracy, feed_dict={X: mnist.t
est.images, Y: mnist.test.labels}))
```

```
Extracting /tmp/data/train-images-idx3-ubyte.gz
Extracting /tmp/data/train-labels-idx1-ubyte.gz
Extracting /tmp/data/t10k-images-idx3-ubyte.gz
```

```
Extracting /tmp/data/t10k-labels-idx1-ubyte.gz
Optimization Finished!
```




```
training_epochs: 15
batch_size: 200
learning_rate: 0.001
Final Train Accuracy:0.980964
Final Test Accuracy:0.9331
```

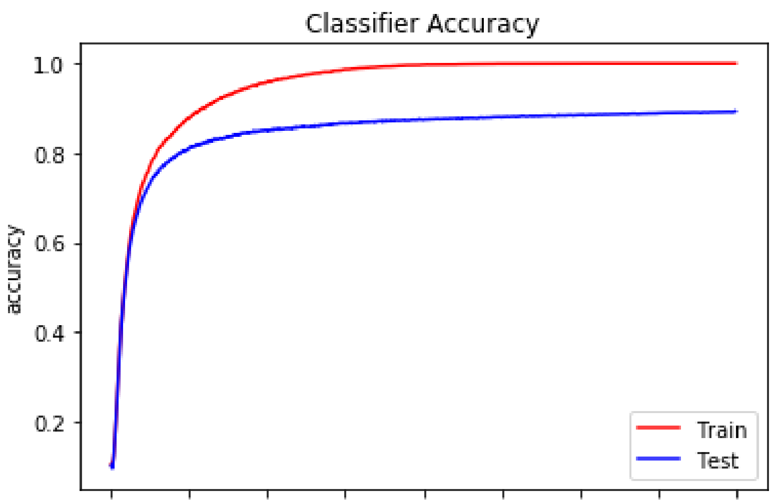
In [2]:

```python
# overriding mnist.train

import tensorflow as tf

# Import MNIST data
from tensorflow.examples.tutorials.mnist import input_data
mnist = input_data.read_data_sets("/tmp/data/", one_hot=True)

import numpy as np
sample_count = np.zeros(10)
TRAIN_SIZE = 1000

training_images_list = []
training_labels_list = []

for i in range(mnist.train.labels.shape[0]):
    label = np.argmax(mnist.train.labels[i])
    if sample_count[label] < TRAIN_SIZE:

        # append training images and labels to the set
        training_images_list.append(mnist.train.images[i])
        training_labels_list.append(mnist.train.labels[i])

        sample_count[label] += 1 # increase sample count for the label

training_images = np.array(training_images_list)
training_labels = np.array(training_labels_list)

mnist.train._num_examples = 10*TRAIN_SIZE
mnist.train._images = training_images
mnist.train._labels = training_labels
```

```
# Parameters
learning_rate = 0.001
training_epochs = 80
batch_size = 200
display_step = 1

# Network Parameters
n_hidden = 300 # number of hidden layer units
n_input = 784 # MNIST image data input, 28*28
n_classes = 10 # MNIST total classes, 0-9 digits

# tf Graph input
X = tf.placeholder("float", [None, n_input])
Y = tf.placeholder("float", [None, n_classes])

# layers weight & bias
weights = {
    'h1': tf.Variable(tf.random_normal([n_input, n_hidden])),
    'out': tf.Variable(tf.random_normal([n_hidden, n_classes]))
}
biases = {
    'b1': tf.Variable(tf.random_normal([n_hidden])),
    'out': tf.Variable(tf.random_normal([n_classes]))
}

# Create model
def multilayer_perceptron(x):
    # Hidden fully connected layer with sigmoid activation function
    hidden_layer = tf.nn.sigmoid(tf.matmul(x, weights['h1']) + biases['b1']
)

    # Output fully connected layer
    out_layer = tf.matmul(hidden_layer, weights['out']) + biases['out']
    return out_layer

# Construct logits with model
logits = multilayer_perceptron(X)

# Define loss and optimizer
loss_op = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(logits=log
its, labels=Y))
train_op = tf.train.AdamOptimizer(learning_rate=learning_rate).minimize(los
s_op)

# prediction and accuracy nodes
pred = tf.nn.softmax(logits)
correct_prediction = tf.equal(tf.argmax(pred, 1), tf.argmax(Y, 1))
accuracy = tf.reduce_mean(tf.cast(correct_prediction, "float"))

# Initializing the variables
init = tf.global_variables_initializer()

# Create session for running nodes
sess = tf.Session()

sess.run(init)

train_accuracy = []
test_accuracy = []

for epoch in range(training_epochs):
```

```
for epoch in range(training_epochs):

    # calculate total number of batches
    total_batch = int(10*TRAIN_SIZE/batch_size)

    # Loop over all batches
    for i in range(total_batch):
        batch_x, batch_y = mnist.train.next_batch(batch_size)

        # Run train op (backprop)
        sess.run(train_op, feed_dict={X: batch_x, Y: batch_y})

        # Compute train accuracy
        train_accuracy.append(sess.run(accuracy, feed_dict={X: mnist.train.i
mages, Y: mnist.train.labels}))
        test_accuracy.append(sess.run(accuracy, feed_dict={X: mnist.test.ima
ges, Y: mnist.test.labels}))

print "Optimization Finished!"

# plot the accuracy
import matplotlib.pyplot as plt

plt.plot(train_accuracy, color = 'r', label = 'Train')
plt.plot(test_accuracy, color = 'b', label = 'Test')
plt.title('Classifier Accuracy')
plt.xlabel('iterations')
plt.ylabel('accuracy')
plt.legend(loc='lower right')
plt.show()

# model parameters and logistics
print "training_epochs: " + str(training_epochs)
print "batch_size: " + str(batch_size)
print "learning_rate: " + str(learning_rate)
print "Final Train Accuracy:" + str(sess.run(accuracy, feed_dict={X: mnist.
train.images, Y: mnist.train.labels}))
print "Final Test Accuracy:" + str(sess.run(accuracy, feed_dict={X: mnist.t
est.images, Y: mnist.test.labels}))
```

```
Extracting /tmp/data/train-images-idx3-ubyte.gz
Extracting /tmp/data/train-labels-idx1-ubyte.gz
Extracting /tmp/data/t10k-images-idx3-ubyte.gz
Extracting /tmp/data/t10k-labels-idx1-ubyte.gz
Optimization Finished!
```




```
training_epochs: 80
batch_size: 200
learning_rate: 0.001
Final Train Accuracy:1.0
Final Test Accuracy:0.8931
```

Problem 1.6: Compare plots from 1.3 and 1.5

For training with the entire dataset in 1.3, the accuracy difference between train and test accuracy gradually increases as the iterations increase. The final difference is about 5% accuracy, which means the train and test accuracy are fairly close to each other.

For training with the smaller dataset in 1.5, the accuracy different between train and test accuracy is approximately constant as iterations increase as they converge. The model is able to overfit the training set and achieved a 100% accuracy on the smaller training set. As expected, since our model is overfitted on this smaller dataset, its classification accuracy (89%) for the test set is lower than the model trained on the entire dataset (93%).

Problem 2.1, 2.2

stride is [1,1,1,1] for convolution, [1,2,2,1] for maxpooling for non-overlapping receptive field

Network Structure:

1) First convolution layer, kernal size of 5x5 and depth 1, 32 output units (filters), ReLU activation and followed by maxpooling

2) Second convolution layer, kernal size of 5x5 and depth 32, 64 output units (filters), ReLU activation and followed by maxpooling

3) Fully connected layer with 1024 output units, ReLU activation

4) Output layer, 1024 input units and 10 logits as output

Training Parameters:

learning_rate: 0.0001, training_iterations: 1500, batch_size: 100

In [5]:

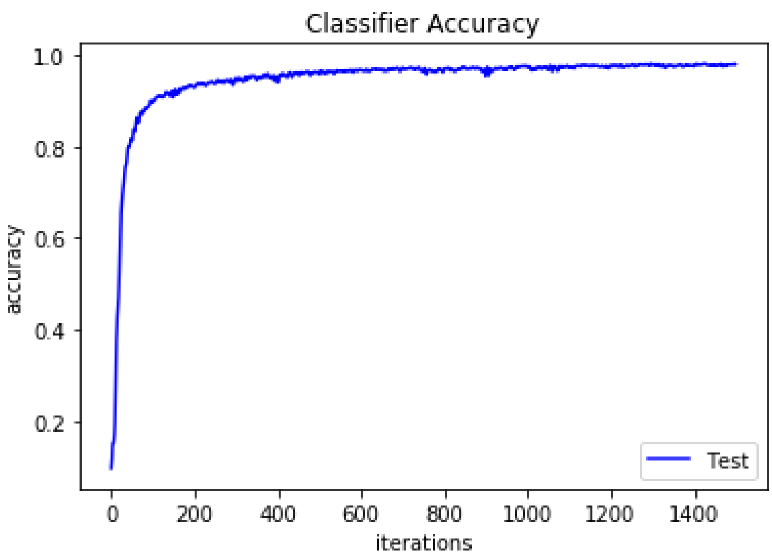
```python
# Problem 2.1-2.3
# some part of the code below is referenced from the Tensorflow tutorial:
# https://www.tensorflow.org/get_started/mnist/pros

import tensorflow as tf

# Import MNIST data
from tensorflow.examples.tutorials.mnist import input_data
mnist = input_data.read_data_sets("/tmp/data/", one_hot=True)

# tf Graph input
n_input = 784 # MNIST image data input, 28*28
n_classes = 10 # MNIST total classes, 0-9 digits
```

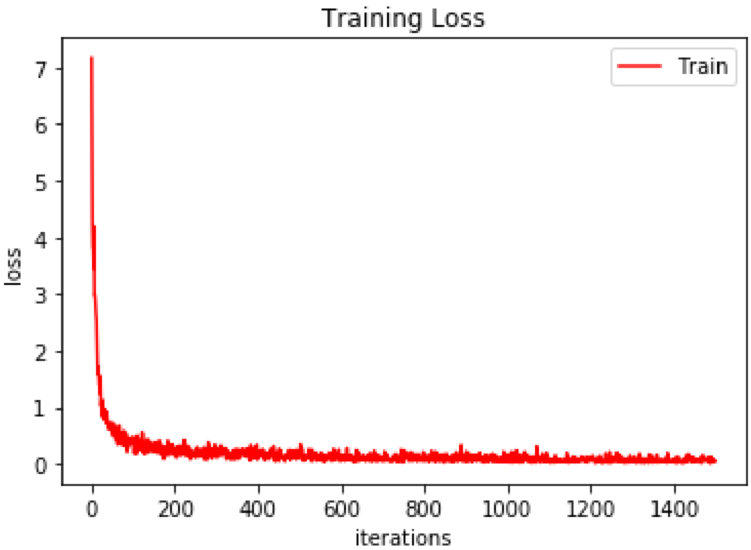
```
n_classes = 10 # MNIST total classes, 0-9 digits
TRAIN_SIZE = 1000

X = tf.placeholder("float", [None, n_input])
Y = tf.placeholder("float", [None, n_classes])

# functions for initializing weights
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)


# functions for convolution layer and max pooling
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')

# CNN model

# Parameters
learning_rate = 0.0001
batch_size = 100

# first conv layer
conv1_size = 32
kernal1_size = 5
W_conv1 = weight_variable([kernal1_size, kernal1_size, 1, conv1_size])
b_conv1 = bias_variable([conv1_size])

# convert to image shape
X_image = tf.reshape(X, [-1, 28, 28, 1])

# convolve images with the kernal W, add bias to it and pass thru a ReLU
activation
h_conv1 = tf.nn.relu(conv2d(X_image, W_conv1) + b_conv1)
h_pool1 = max_pool_2x2(h_conv1) # max pooling


# second conv layer, similar to first layer but with different units #
kernal2_size = 5
conv2_size = 64
W_conv2 = weight_variable([kernal2_size, kernal2_size, conv1_size, conv2_si
ze])
b_conv2 = bias_variable([conv2_size])

h_conv2 = tf.nn.relu(conv2d(h_pool1, W_conv2) + b_conv2)
h_pool2 = max_pool_2x2(h_conv2)


# Fully connected layer, feature map size is now 7*7 after previous operati
ons, reduced from 28*28
fc1_size = 1024
W_fc1 = weight_variable([7 * 7 * conv2_size, fc1_size])
b_fc1 = bias_variable([fc1_size])
```

```
b_fc1 = bias_variable([fc1_size])

h_pool2_flat = tf.reshape(h_pool2, [-1, 7 * 7 * conv2_size])
h_fc1 = tf.nn.relu(tf.matmul(h_pool2_flat, W_fc1) + b_fc1)


# output layer
fc2_size = 10 # same size as number of classes
W_fc2 = weight_variable([fc1_size, fc2_size])
b_fc2 = bias_variable([fc2_size])

y_conv = tf.matmul(h_fc1, W_fc2) + b_fc2


# Define loss and optimizer
loss_op = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(logits = y
_conv, labels = Y))
train_op = tf.train.AdamOptimizer(learning_rate=learning_rate).minimize(los
s_op)

# prediction and accuracy nodes
pred = tf.nn.softmax(y_conv)
correct_prediction = tf.equal(tf.argmax(pred, 1), tf.argmax(Y, 1))
accuracy = tf.reduce_mean(tf.cast(correct_prediction, "float"))

# Initializing the variables
init = tf.global_variables_initializer()

# Create session for running nodes
sess = tf.Session()

sess.run(init)

train_loss = []
test_accuracy = []
#with tf.Session() as sess:

iteration_size = 1500
for epoch in range(iteration_size):

    # Loop over batches
    batch_x, batch_y = mnist.train.next_batch(batch_size)

    # Run train op (backprop)
    _, c = sess.run([train_op, loss_op], feed_dict={X: batch_x, Y: batch_y})

    # Compute train accuracy
    train_loss.append(c)
    test_accuracy.append(sess.run(accuracy, feed_dict={X: mnist.test.images
, Y: mnist.test.labels}))

print "Optimization Finished!"

# plot the accuracy
import matplotlib.pyplot as plt

plt.plot(test_accuracy, color = 'b', label = 'Test')
plt.title('Classifier Accuracy')
plt.xlabel('iterations')
plt.ylabel('accuracy')
plt.legend(loc='lower right')
```

```
plt.show()

plt.plot(train_loss, color = 'r', label = 'Train')
plt.title('Training Loss')
plt.xlabel('iterations')
plt.ylabel('loss')
plt.legend(loc='upper right')
plt.show()

print "iteration_size: " + str(iteration_size)
print "batch_size: " + str(batch_size)
print "learning_rate: " + str(learning_rate)
print "Final Test Accuracy:" + str(sess.run(accuracy, feed_dict={X: mnist.test.images, Y: mnist.test.labels}))
```

```
Extracting /tmp/data/train-images-idx3-ubyte.gz
Extracting /tmp/data/train-labels-idx1-ubyte.gz
Extracting /tmp/data/t10k-images-idx3-ubyte.gz
Extracting /tmp/data/t10k-labels-idx1-ubyte.gz
Optimization Finished!
```






```
iteration_size: 1500
batch_size: 100
learning_rate: 0.0001
Final Test Accuracy:0.9805
```

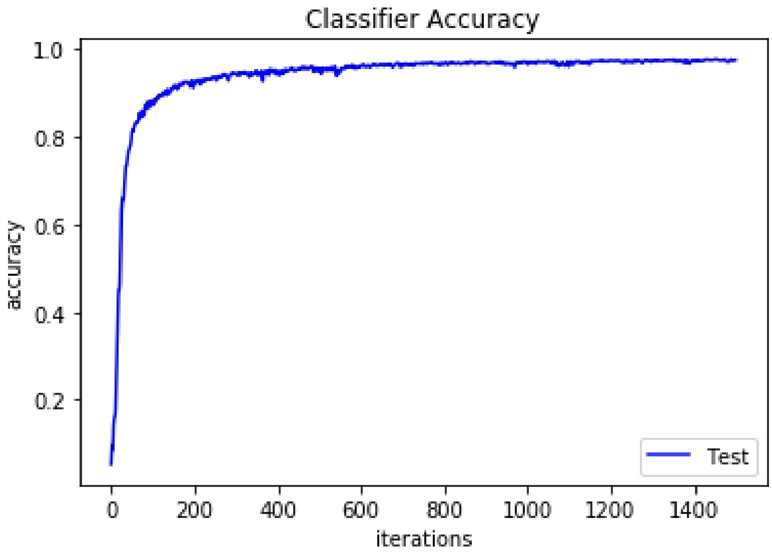
In [4]:

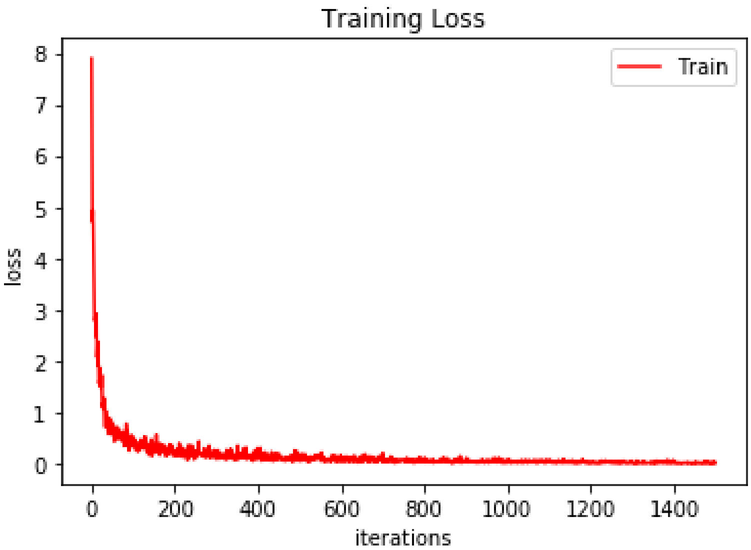
```
# Problem 2.4
# some part of the code below is referenced from the Tensorflow tutorial:
# https://www.tensorflow.org/get_started/mnist/pros

import tensorflow as tf

# Import MNIST data
from tensorflow.examples.tutorials.mnist import input_data
mnist = input_data.read_data_sets("/tmp/data/", one_hot=True)

import numpy as np
sample_count = np.zeros(10)
TRAIN_SIZE = 1000

training_images_list = []
training_labels_list = []

for i in range(mnist.train.labels.shape[0]):
    label = np.argmax(mnist.train.labels[i])
    if sample_count[label] < TRAIN_SIZE:

        # append training images and labels to the set
        training_images_list.append(mnist.train.images[i])
        training_labels_list.append(mnist.train.labels[i])

        sample_count[label] += 1 # increase sample count for the label

training_images = np.array(training_images_list)
training_labels = np.array(training_labels_list)

mnist.train._num_examples = 10*TRAIN_SIZE
mnist.train._images = training_images
mnist.train._labels = training_labels


# tf Graph input
n_input = 784 # MNIST image data input, 28*28
n_classes = 10 # MNIST total classes, 0-9 digits

X = tf.placeholder("float", [None, n_input])
Y = tf.placeholder("float", [None, n_classes])

# functions for initializing weights
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)


# functions for convolution layer and max pooling
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')

# CNN model

# Parameters
learning_rate = 0.0001
batch_size = 100

# first conv layer
conv1_size = 32
kernal1_size = 5
W_conv1 = weight_variable([kernal1_size, kernal1_size, 1, conv1_size])
b_conv1 = bias_variable([conv1_size])

# convert to image shape
X_image = tf.reshape(X, [-1, 28, 28, 1])

# convolve images with the kernal W, add bias to it and pass thru a ReLU
activation
h_conv1 = tf.nn.relu(conv2d(X_image, W_conv1) + b_conv1)
h_pool1 = max_pool_2x2(h_conv1) # max pooling


# second conv layer, similar to first layer but with different units #
kernal2_size = 5
conv2_size = 64
W_conv2 = weight_variable([kernal2_size, kernal2_size, conv1_size, conv2_si
ze])
b_conv2 = bias_variable([conv2_size])

h_conv2 = tf.nn.relu(conv2d(h_pool1, W_conv2) + b_conv2)
h_pool2 = max_pool_2x2(h_conv2)


# Fully connected layer, feature map size is now 7*7 after previous operati
ons, reduced from 28*28
fc1_size = 1024
W_fc1 = weight_variable([7 * 7 * conv2_size, fc1_size])
b_fc1 = bias_variable([fc1_size])

h_pool2_flat = tf.reshape(h_pool2, [-1, 7 * 7 * conv2_size])
h_fc1 = tf.nn.relu(tf.matmul(h_pool2_flat, W_fc1) + b_fc1)


# output layer
fc2_size = 10 # same size as number of classes
W_fc2 = weight_variable([fc1_size, fc2_size])
b_fc2 = bias_variable([fc2_size])

y_conv = tf.matmul(h_fc1, W_fc2) + b_fc2


# Define loss and optimizer
loss_op = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(logits = y
_conv, labels = Y))
train_op = tf.train.AdamOptimizer(learning_rate=learning_rate).minimize(los
s_op)

# prediction and accuracy nodes
pred = tf.nn.softmax(y_conv)
```

```
correct_prediction = tf.equal(tf.argmax(pred, 1), tf.argmax(Y, 1))
accuracy = tf.reduce_mean(tf.cast(correct_prediction, "float"))

# Initializing the variables
init = tf.global_variables_initializer()

# Create session for running nodes
sess = tf.Session()

sess.run(init)

train_loss = []
test_accuracy = []
#with tf.Session() as sess:

iteration_size = 1500
for epoch in range(iteration_size):

    # Loop over batches
    batch_x, batch_y = mnist.train.next_batch(batch_size)

    # Run train op (backprop)
    _, c = sess.run([train_op, loss_op], feed_dict={X: batch_x, Y: batch_y})

    # Compute train accuracy
    train_loss.append(c)
    test_accuracy.append(sess.run(accuracy, feed_dict={X: mnist.test.images
, Y: mnist.test.labels}))

print "Optimization Finished!"

# plot the accuracy
import matplotlib.pyplot as plt

plt.plot(test_accuracy, color = 'b', label = 'Test')
plt.title('Classifier Accuracy')
plt.xlabel('iterations')
plt.ylabel('accuracy')
plt.legend(loc='lower right')
plt.show()

plt.plot(train_loss, color = 'r', label = 'Train')
plt.title('Training Loss')
plt.xlabel('iterations')
plt.ylabel('loss')
plt.legend(loc='upper right')
plt.show()

print "iteration_size: " + str(iteration_size)
print "batch_size: " + str(batch_size)
print "learning_rate: " + str(learning_rate)
print "Final Test Accuracy:" + str(sess.run(accuracy, feed_dict={X: mnist.t
est.images, Y: mnist.test.labels}))
```

```
Extracting /tmp/data/train-images-idx3-ubyte.gz
Extracting /tmp/data/train-labels-idx1-ubyte.gz
Extracting /tmp/data/t10k-images-idx3-ubyte.gz
Extracting /tmp/data/t10k-labels-idx1-ubyte.gz
Optimization Finished!
```






```
iteration_size: 1500
batch_size: 100
learning_rate: 0.0001
Final Test Accuracy:0.9752
```

Problem 2.5: Compare plots from 3 and 4

The resulting plots for accuracy and loss and their convergences are very similar, except that the training on entire dataset has more noise in its loss curve. This makes sense because the batches sampled from the entire data set have higher variance, which causes the parameter updates and loss to fluctuate more, resulting in higher noise.

Recurrent Neural Network in Sine Reconstruction

In this notebook, you will learn how to use RNN to reconstruct sine funcion.

Attention Every time when you do some modification, if you find the code cannot run correctly, please use `Restart` under `Kernel` menu in Jupyter Notebook to re-run the code.

Import Library

This section imports all needed libraries. All libraries are either built-in or from PyPI. You may need to use `pip` to install missing libraries.

In [1]:

```
import os
# Set GPU number.
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

# Display the versions for libraries.
print('Python version: {}'.format(sys.version))
print('NumPy version: {}'.format(np.__version__))
print('TensorFlow version: {}'.format(tf.__version__))
print('Matplotlib version: {}'.format(matplotlib.__version__))

# In my environment, they are
#    Python version: 2.7.13 |Anaconda, Inc.| (default, Sep 30 2017, 18:12:4
3)
#    [GCC 7.2.0]
#    NumPy version: 1.13.1
#    TensorFlow version: 1.3.0
#    Matplotlib version: 2.1.0
```

```
Python version: 2.7.14 |Anaconda, Inc.| (default, Oct  5 2017, 07:26:46)
[GCC 7.2.0]
NumPy version: 1.13.3
TensorFlow version: 1.4.0-dev20171012
Matplotlib version: 2.1.0
```

Generate Samples (to be filled)

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

$$y = F(t) = \sin(2\pi f(t + t_{rand}))$$

where f is the frenquency. In this problem, it is fixed at 1 where f is the frequency. In this problem, it is fixed at 1.

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_{rand} 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 $n_{predict} = 1$.

In [2]:

```
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 times and values. In all outputs,
             each row represents one time series of the batch.
    """
    const = 100.0

    # Empty batch vectors.
    samples_T = np.empty((batch_size, samples))
    samples_Y = np.empty((batch_size, samples))
    predict_T = np.empty((batch_size, predict))
    predict_Y = np.empty((batch_size, predict))

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

        # Here we want to sample some points for sine function.
        t_rand = np.random.rand()*5 #_                # t_rand is a random single value.
        y = np.sin(2*np.pi*f*(t+t_rand)) # y = F(t) = sin(2*pi*f*(t+t_rand)), e.g. [F(0),F(1),..., F(100)]
                                         # f is the frequency from parameter.
        # Use the slice of y to fill the blanks below:
        # samples_T is the batches of sample time points.
        # samples_Y is the batches of sample function values corresponding to samples_T.
        samples_T[i, :] =  t[:samples]# t_1 ... t_{n_samples} e.g. [0, 1, ..., 99]
        samples_Y[i, :] =  y[:samples] # y_1 ... y_{n_samples} e.g. [F(0), F(1), ..., F(99)]
        # samples_T is the batches of sample time points.
        # samples_Y is the batches of sample function values corresponding to samples_T.
        predict_T[i, :] =  t[samples:] # t_{n_samples+1} ... t_{n_samples+n_predict} e.g. [100]
        predict_Y[i, :] =  y[samples:] # y_{n_samples+1} ... y_{n_samples+n_predict} e.g. [F(100)]

    return samples_T, samples_Y, predict_T, predict_Y
```

RNN Model (to be filled)

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

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

In [3]:

```
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_inpu
t).
    x = tf.split(x, n_steps, axis=0)
    # You can use x as input data below.

    # Define a RNN cell with TensorFlow.
    rnn_cell = rnn.BasicRNNCell(n_hidden) #_

    # Get RNN cell output.
    # Hint: Use rnn.static_rnn() and x as the input.
    outputs, states = rnn.static_rnn(rnn_cell, x, dtype=tf.float32)

    # Linear layer and tanh activation, using RNN last output.
    # Hint: Use tf.tanh, tf.nn.bias_add(), tf.matmul(), weights, biases.
    final_output = tf.tanh(tf.nn.bias_add(tf.matmul(outputs[-1], weights),
biases))

    return final_output
```

Parameters and Network Building

This section sets the parameters and builds the network.

In [4]:

```
# 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=1)
loss = tf.reduce_mean(individual_losses)

optimizer = tf.train.AdamOptimizer(learning_rate=learning_rate).minimize(lo
ss)

# 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.

In [5]:

```
# 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=n_steps,
                                              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= 0.001725
Iter 10000, Minibatch Loss= 0.000326
Iter 15000, Minibatch Loss= 0.000397
Iter 20000, Minibatch Loss= 0.001413
Iter 25000, Minibatch Loss= 0.000442
Iter 30000, Minibatch Loss= 0.015467
Iter 35000, Minibatch Loss= 0.000126
Iter 40000, Minibatch Loss= 0.000112
Iter 45000, Minibatch Loss= 0.005627
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.

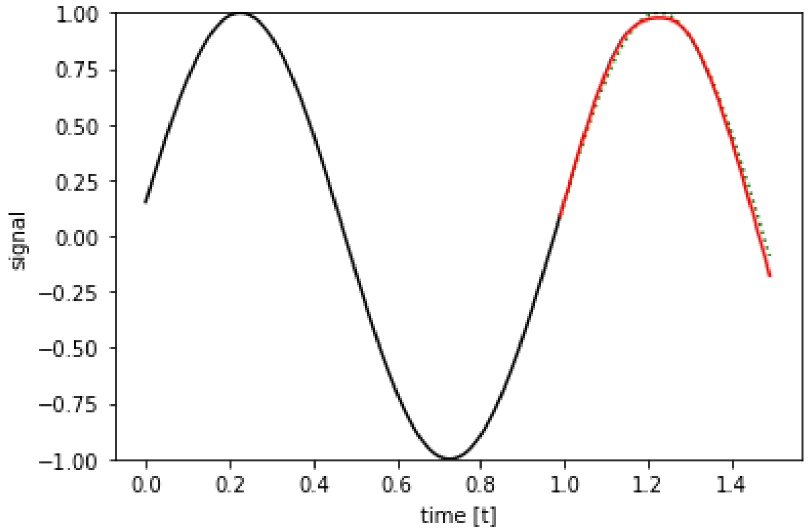
In [6]:

```
# 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)
```

In [10]:

```
# 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', linestyle=':')
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()
```


1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
signal
0.0
0.2
0.4
0.6
0.8
1.0
1.2
1.4
time [t]