Building Deep Convolutional Networks with TensorFlow

The basic idea of TensorFlow is that you first build up a model with Variables and Constants connected together into a "flow" of functions. Then you run a session which process the whole flow to give a result. You can also use "placeholders" for "input constants"

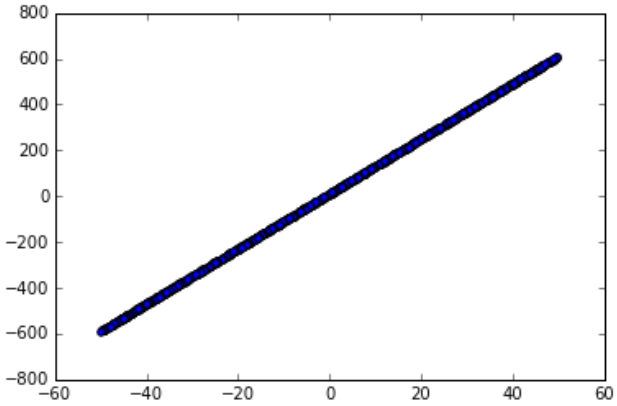
In [6]:
```
import tensorflow as tf

x = tf.placeholder(tf.float32, (2,1))
W = tf.constant([[3., 5.]])
y = tf.matmul(x,W)

with tf.Session() as sess:
    print y.eval(feed_dict={x:[[2.],[4.]]})
```

```
[[  6.  10.]
 [ 12.  20.]]
```

There are many methods defined to help create neural networks following the same principles. When you build the "graph" with all the variables, there are also methods defined to optimize them according to a certain goal.

Let's say we want to make a regression of the type y = ax+b with some data points

In [54]:
```
%matplotlib inline
import numpy as np
from matplotlib import pyplot as plt

# y = 12x + 5 + random noise
data_x = np.random.random((1000,1))*100 - 50
data_y = data_x*12 + 5 + np.random.random((1000,1))*2-1

plt.scatter( data_x, data_y )
plt.show()
```

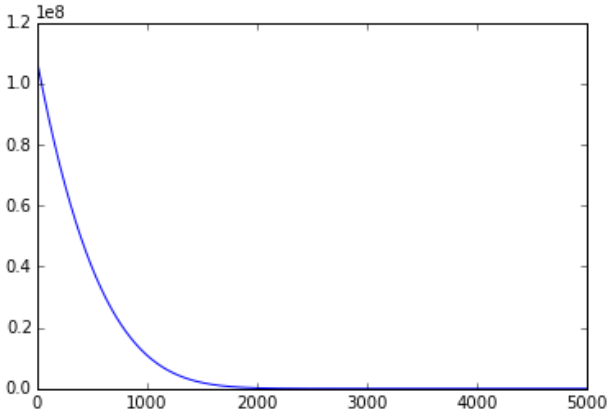


In [65]:
```
import tensorflow as tf

x = tf.placeholder(tf.float32, [None, 1]) # None means that it can take any nu
mber of input points

a = tf.Variable(tf.random_normal([1,1]))
b = tf.Variable(tf.random_normal([1]))

y = tf.matmul(x, a)+b # the "model" : y = ax + b
print y

target = tf.placeholder(tf.float32, [None,1])
cost = tf.reduce_sum(tf.square(target-y)) # cost function : sum on all trainin
g set of (target-y)²

opt = tf.train.AdamOptimizer(1e-2).minimize(cost) # objective : minimize the c
ost
print opt
```

```
Tensor("add_20:0", shape=TensorShape([Dimension(None), Dimension(1)]), dtype=fl
oat32)
name: "Adam_2"
op: "NoOp"
input: "^Adam_2/update_Variable_42/ApplyAdam"
input: "^Adam_2/update_Variable_43/ApplyAdam"
input: "^Adam_2/Assign"
input: "^Adam_2/Assign_1"
```

In [70]:
```
init = tf.initialize_all_variables()

with tf.Session() as sess:
    sess.run(init)
    print "Initial values of model : y = %fx + %f"%(a.eval(), b.eval())

    # iterate the optimizer
    cost_over_time = []
    for i in xrange(5000):
        opt.run(feed_dict={x:data_x, target:data_y})
        cost_over_time.append(cost.eval(feed_dict={x:data_x, target:data_y}))

    print "Final values of model : y = %fx + %f"%(a.eval(), b.eval())
    plt.plot(cost_over_time)
    plt.show()
```

```
Initial values of model : y = 0.577805x + 0.083649
Final values of model : y = 11.998857x + 4.982445
```



After that, it's just a matter of putting it all together to create the model we want. To make it all easier, we define the helper functions to create layers of a deep model. Each layer will take an input *x*, and produce an output *h*. It may have variable weights and biases. We also have helper functions to initialize the weights. We have a very simple Network class putting everything together.

In [75]:
```
from NetworkLayer import *
from WeightInit import WeightInit
```

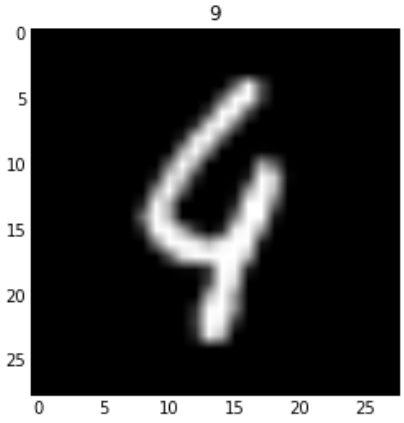
In [77]:
```
class Network():
    def __init__(self, inputLayer, hiddenLayers, outputLayer):
        self.layers = []
        self.x = inputLayer.x

        inputLayer.link()
        self.layers.append(inputLayer)

        curx = inputLayer.h
        for hl in hiddenLayers:
            hl.link(curx)
            self.layers.append(hl)
            curx = hl.h
            print curx

        outputLayer.link(curx)
        self.layers.append(outputLayer)
        self.y = outputLayer.h

    def setupTrainingForClassification(self, costFunction="cross-entropy", opt
imizer="Adam"):
        self.target = tf.placeholder(tf.float32, self.y.get_shape(), "Target")
        if costFunction == "cross-entropy":
            self.cost = -tf.reduce_sum(self.target*tf.log(self.y))
        elif costFunction == "squared-diff":
            self.cost = tf.reduce_mean(tf.square(self.target-self.y))

        if optimizer == "Adam":
            self.trainingStep = tf.train.AdamOptimizer(1e-4).minimize(self.cos
t)
        elif optimizer == "GradientDescent":
            self.trainingStep = tf.train.GradientDescentOptimizer(0.01).minimi
ze(self.cost)
        self.correct_prediction = tf.equal(tf.argmax(self.y,1), tf.argmax(self
.target,1))
        self.accuracy = tf.reduce_mean(tf.cast(self.correct_prediction, "float
"))

    def train(self, x, y):
        self.trainingStep.run(feed_dict={self.x: x, self.target: y})

    def evaluate(self, x, y):
        return self.accuracy.eval(feed_dict={self.x: x, self.target: y})

    def predict(self, x):
        return self.y.eval(feed_dict={self.x: x})
```

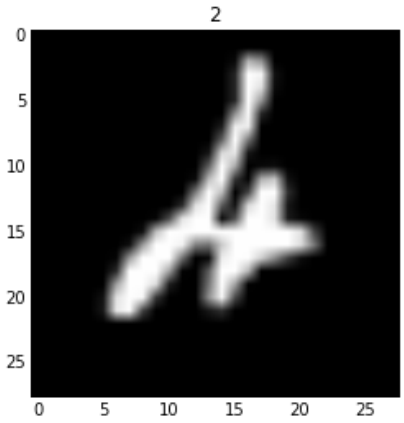
With that, we can create complex models in a few lines of code. For instance, we define here a model taking as input a flattened vector from the 28x28 images of the MNIST digits database. It then has a 1000 neurons hidden layer, and another 200 neurons hidden layers, both fully connected with a tanh activation function. The last layer is a softmax classification layer with ten classes (0-9)

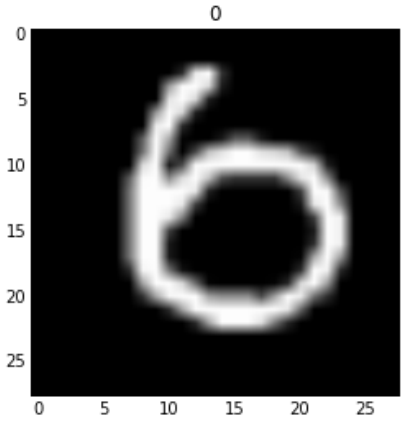
In [78]:
```
FullyConnectedNetworkDefinition = {
    'inputLayer' : FlatInputLayer(784),
    'hiddenLayers' : [
        FullyConnectedLayer(784,1000,WeightInit.truncatedNormal,WeightInit.pos
itive,tf.nn.tanh),
        FullyConnectedLayer(1000,200,WeightInit.truncatedNormal,WeightInit.pos
itive,tf.nn.tanh)
    ],
    'outputLayer' : FullyConnectedLayer(200,10,WeightInit.truncatedNormal,Weig
htInit.positive,tf.nn.softmax)
}
```

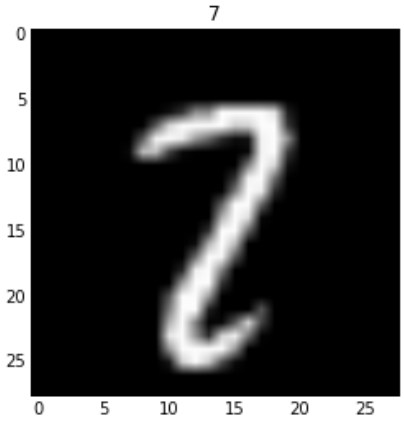
In [89]:
```
import MNISTData # Helper functions to read the MNIST dataset

model = FullyConnectedNetworkDefinition
net = Network(model['inputLayer'], model['hiddenLayers'], model['outputLayer']
)
net.setupTrainingForClassification("cross-entropy", "Adam")

init = tf.initialize_all_variables()
mnist = MNISTData.read_data_sets('MNIST_data', one_hot=True)

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

    # Training
    for i in range(20000):
        batch_xs, batch_ys = mnist.train.next_batch(50)
        if i%1000 == 0:
            acc = net.evaluate(batch_xs, batch_ys)
            print "step %d, training accuracy %g"%(i,acc)

        net.train(batch_xs, batch_ys)

    print "Test accuracy %g"%net.evaluate(mnist.test.images, mnist.test.labels
)

    # Showing a few example of misclassified digits
    for i in range(500):
        y = net.predict([mnist.test.images[i]]).argmax()
        if y != mnist.test.labels[i].argmax():
            plt.figure()
            plt.gray()
            plt.imshow(mnist.test.images[i].reshape([28,28]))
            plt.title(str(y))
    plt.show()
```

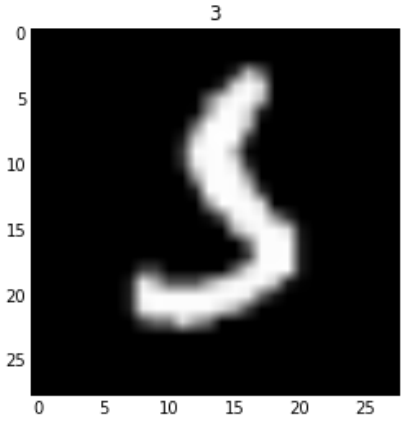
```
Tensor("Tanh_12:0", shape=TensorShape([Dimension(None), Dimension(1000)]), dtyp
e=float32)
Tensor("Tanh_13:0", shape=TensorShape([Dimension(None), Dimension(200)]), dtype
=float32)
Extracting MNIST_data/train-images-idx3-ubyte.gz
Extracting MNIST_data/train-labels-idx1-ubyte.gz
Extracting MNIST_data/t10k-images-idx3-ubyte.gz
Extracting MNIST_data/t10k-labels-idx1-ubyte.gz
step 0, training accuracy 0.08
step 1000, training accuracy 0.92
step 2000, training accuracy 0.96
step 3000, training accuracy 0.98
step 4000, training accuracy 1
step 5000, training accuracy 0.94
step 6000, training accuracy 0.98
step 7000, training accuracy 1
step 8000, training accuracy 1
step 9000, training accuracy 0.98
step 10000, training accuracy 1
step 11000, training accuracy 1
step 12000, training accuracy 0.98
step 13000, training accuracy 1
step 14000, training accuracy 1
step 15000, training accuracy 1
step 16000, training accuracy 1
step 17000, training accuracy 1
step 18000, training accuracy 1
step 19000, training accuracy 1
Test accuracy 0.9801
```

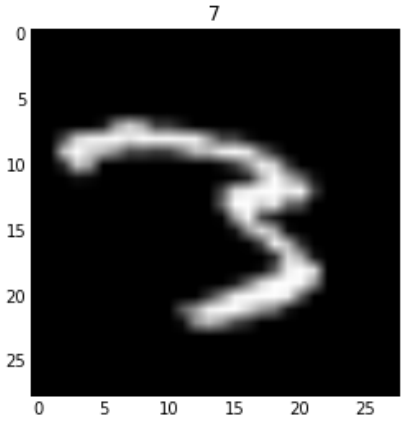


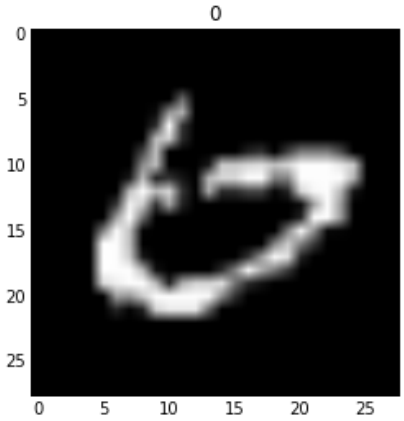


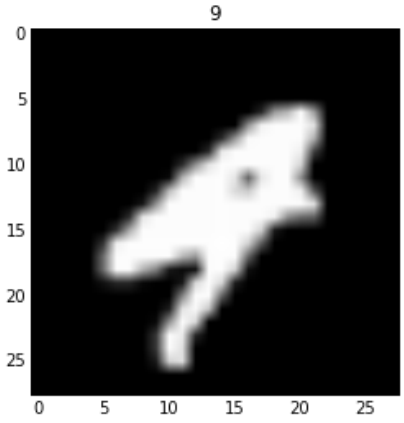

0
0
5
10
15
20
25
0
5
10
15
20
25

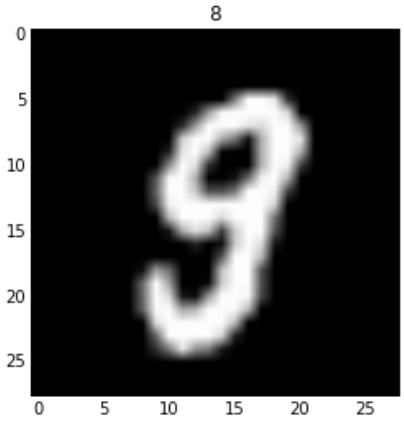

7
0
5
10
15
20
25
0
5
10
15
20
25

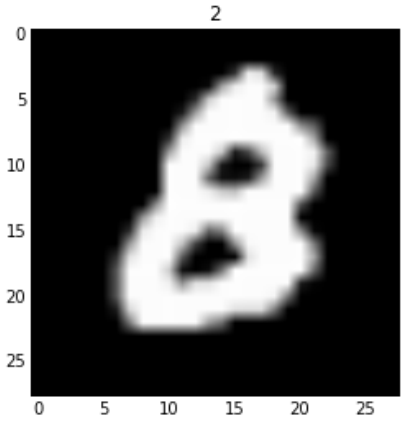

3
0
5
10
15
20
25
0
5
10
15
20
25


7
0
5
10
15
20
25
0
5
10
15
20
25


0
0
5
10
15
20
25
0
5
10
15
20
25


9
0
5
10
15
20
25
0
5
10
15
20
25


8
0
5
10
15
20
25
0
5
10
15
20
25


2
0
5
10
15
20
25
0
5
10
15
20
25