Deep Learning for Computer Vision: Assignment 4

Computer Science: COMS W 4995 006

Due: March 20, 2018

Problem

In this notebook we provide three networks for classifying handwritten digits from the MNIST dataset. The networks are implemented and tested using the Tensorflow framework. The third and final network is a convolutional neural network (CNN aka ConvNet) which achieves 99.25% accuracy on this dataset.

Your task is to re-implement all three networks using the Keras wrapper around Tensorflow OR re-implement using Pytorch. You will likely find several Keras or Pytorch implementations on the internet. It is ok to study these. However, you must not cut and paste this code into your assignment--you must write this yourself. Furthermore, you need to comment every line of code and succintly explain what it is doing!

Here is what is required:

a) A FULLY commented re-implementation of the ConvNet below using the Keras wrapper on Tensorflow OR Pytorch.

b) your network trained on the same MNIST data as used here.

c) an evaluation of the accuracy on the MNIST test set.

d) plots of 10 randomly selected digits from the test set along with the correct label and the assigned label.

e) have your training record a log of the data using the Keras API and then use Tensorboard (a command line tool) to display plots of the validation loss and validation accuracy. you can zip up a screenshot of this with your notebook before submission.

f) have your training continually save the best model so far (as determined by the validation loss) using the Keras API or Pytorch.

g) after training, load the saved weights using the best model so far. re-run you accuracy evaluation using these saved weights.

Below we include the Tensorflow examples shown in class.

A Simple Convolutional Neural Network in Tensorflow

This notebook covers a python and tensorflow-based solution to the handwritten digits recognition problem. It is based on tensorflow tutorials and Yann LeCun's early work on CNN's. This toturial compares a simple softmax regressor, a multi-layer perceptron (MLP), and a simple convolutional neural network (CNN).

Load in the MNIST digit dataset directly from tensorflow examples.

In [2]:
```
from tensorflow.examples.tutorials.mnist import input_data
mnist = input_data.read_data_sets('MNIST_data', one_hot=True)
```

```
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
```

The MNIST data is split into three parts: 55,000 data points of training data (mnist.train), 10,000 points of test data (mnist.test), and 5,000 points of validation data (mnist.validation).

Let's import tensorflow and begin an interactive session.

In [34]:
```
import tensorflow as tf
sess = tf.InteractiveSession()
```

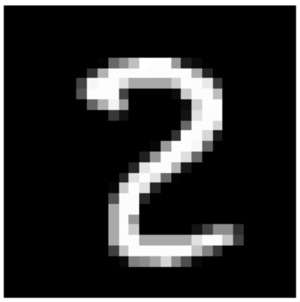
Softmax Regression Model on the MNIST Digits Data

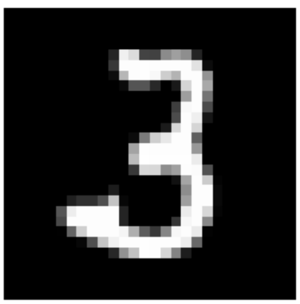
We need to create placeholders for the data. Data will be dumped here when it is batched from the MNIST dataset.

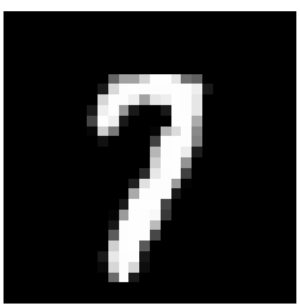
In [88]:
```
x = tf.placeholder(tf.float32, shape=[None, 784])
y_ = tf.placeholder(tf.float32, shape=[None, 10])
```

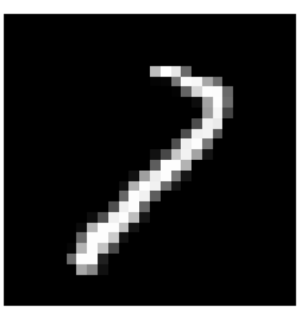
Now let's see what this data looks like.

In [89]:
```
import matplotlib.pyplot as plt
import numpy as np

for i in range(4):
    batch = mnist.test.next_batch(1)
    image = np.asarray(batch[0]).reshape((28, 28))
    label = batch[1]

    plt.imshow(image, cmap='gray')
    plt.axis("off")
    plt.show()
```









We are first going to do softmax logistic regression. This is a linear layer followed by softmax. Note there are NO hidden layers here. Also note that the digit images (28x28 grayscale images) are reshaped into a 784 element vector.

Below we create the parameters (weights) for our linear layer.

In [90]:
```
W = tf.Variable(tf.zeros([784,10]))
b = tf.Variable(tf.zeros([10]))
```

We then use tensorflows initializer to initialize these weights.

In [91]:
```
sess.run(tf.global_variables_initializer())
```

We create our linear layer as a function of the input and the weights.

In [92]:
```
y_regressor = tf.matmul(x,W) + b
```

Below we create our loss function. Note that the cross entropy is $H_{\hat{y}}(y) = -\sum_i \hat{y}_i \ \log(y_i)$ where $\hat{y}$ is the true probability distribution and is expressed as a one-hot vector, y is the estimated probability distribution, and i indexes elements of these two vectors. Also note that this reduces to $H_{\hat{y}}(y) = -\log(y_{i^*})$ where i^* is the correct label. And if we sum this over all of our samples indexed by j, then $H_{\hat{y}}(y) = -\sum_j \log(y_{i^*}^{(j)})$. This is precisely the same loss function as we used before, but we called the MLE loss. They are one and the same.

In [93]:
```
cross_entropy = tf.reduce_mean(
    tf.nn.softmax_cross_entropy_with_logits(labels=y_, logits=y_regressor))
```

Now we tell tf to use gradient descent with a step size of 0.5 and to minimize the cross entropy.

In [94]:
```
train_step = tf.train.GradientDescentOptimizer(0.5).minimize(cross_entropy)
```

We train by grabbing mini-batches with 100 samples each and pushing these through the network to update our weights (W and b).

In [95]:
```
for _ in range(1000):
  batch = mnist.train.next_batch(100)
  train_step.run(feed_dict={x: batch[0], y_: batch[1]})
```

We define how to compute correct predicitions.

In [96]:
```
correct_prediction = tf.equal(tf.argmax(y_regressor,1), tf.argmax(y_,1))
```

And from these correct predictions how to compute the accuracy.

In [97]:
```
accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))
```

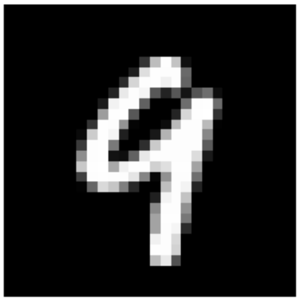
In [98]:
```
print(accuracy.eval(feed_dict={x: mnist.test.images, y_: mnist.test.labels}))
```

```
0.9168
```

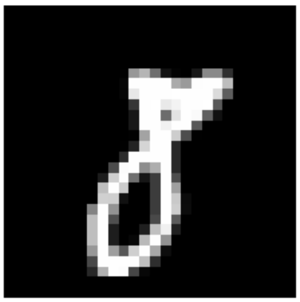
Let's print out some test images and the corresponsing predictions made by the network. But first, let's add an output to the computation graph that computes the softmax probabilities.

In [99]:
```
y_probs_regressor = tf.nn.softmax(logits=y_regressor, name=None)
```

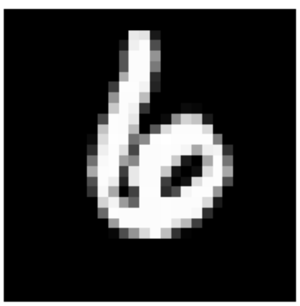
In [100]:
```
for i in range(5):
    batch = mnist.test.next_batch(1)
    image = np.asarray(batch[0]).reshape((28, 28))
    label = batch[1]

    plt.imshow(image, cmap='gray')
    plt.axis("off")
    plt.show()
    print ("Label = ", label)
    print ("Class probabilities = ", y_probs_regressor.eval(feed_dict={
        x: batch[0], y_: batch[1]}))
```



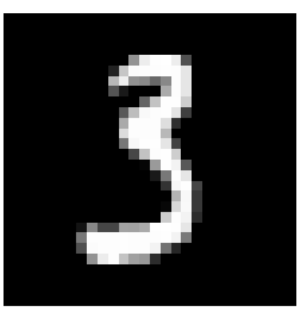
```
Label =  [[0. 0. 0. 0. 0. 0. 0. 0. 0. 1.]]
Class probabilities =  [[5.7868356e-06 1.0960512e-06 7.0933005e-05 1.3369437e-04 2.2966379e-01
  4.3973927e-05 8.0909129e-05 2.7946601e-04 5.5398531e-03 7.6418048e-01]]
```



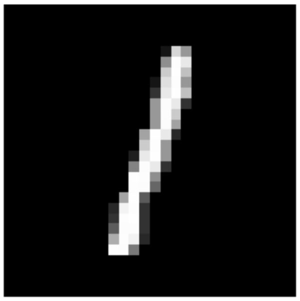
```
Label =  [[0. 0. 0. 0. 0. 0. 0. 0. 1. 0.]]
Class probabilities =  [[1.07988366e-04 1.85040496e-02 6.85910694e-04 9.52854156e-01
  3.68666690e-07 1.11012615e-03 3.56482616e-07 1.94580108e-03
  1.34969251e-02 1.12942373e-02]]
```



```
Label =  [[0. 0. 0. 0. 0. 0. 1. 0. 0. 0.]]
Class probabilities =  [[7.7000050e-07 1.9392106e-04 2.6541899e-03 1.9969192e-04 4.3661382e-02
  3.7085495e-04 9.4621152e-01 2.3503811e-05 1.2821722e-03 5.4019629e-03]]
```



```
Label =  [[0. 0. 0. 1. 0. 0. 0. 0. 0. 0.]]
Class probabilities =  [[4.0822830e-03 6.9544464e-02 1.0719517e-02 6.8291938e-01 1.6180762e-06
  1.9808555e-01 2.7856557e-04 2.1188654e-04 3.3590816e-02 5.6585693e-04]]
```



```
Label =  [[0. 1. 0. 0. 0. 0. 0. 0. 0. 0.]]
Class probabilities =  [[2.2713131e-05 9.7108227e-01 1.3474120e-02 5.9603201e-03 4.4256221e-06
  7.7636359e-04 2.2026870e-04 1.5071974e-04 8.0492878e-03 2.5937715e-04]]
```

Softmax regression model

a) - b) Re-implement softmax regression model and train it using keras

In [184]:
```
from keras.models import Sequential
from keras.layers import Dense, Activation
from keras.optimizers import SGD
from keras.callbacks import ModelCheckpoint, TensorBoard

import matplotlib.pyplot as plt
import numpy as np
```

In [185]:
```
# initialize a sequential keras model for softmax regression
model = Sequential([
    # a single fully connected layer without activation function
    # initialize weight and bias to be zero
    Dense(10, input_dim=784, kernel_initializer='zeros', bias_initializer='zeros'),
    # use softmax to compute 10 class possibilities
    Activation('softmax'),
])

# define the SGD optimizaer for this model, set learning rate to be 0.5
sgd = SGD(lr=0.5)

# configure the training process, using SGD with cross entropy loss
model.compile(optimizer=sgd,
              loss='categorical_crossentropy',
              metrics=['accuracy'])

# checkpoint to save the model with best validation accuracy
checkpointer = ModelCheckpoint(filepath='weights/weights_softmax_regression.hdf5', monitor='val_acc', verbose=1, sav
e_best_only=True, mode='max')
# another callback function to record a log of the data for tensorboard visualization
tensorboard = TensorBoard(log_dir='tensorboard/', histogram_freq=0, write_graph=True, write_images=True)

# start traing 30 epochs
# batch size is 100
model.fit(x=mnist.train.images,
          y=mnist.train.labels,
          validation_data=(mnist.validation.images, mnist.validation.labels),
          callbacks=[checkpointer, tensorboard],
          verbose=1,
          epochs=30,
          batch_size=100)
```

```
Train on 55000 samples, validate on 5000 samples
Epoch 1/30
55000/55000 [==============================] - 1s 23us/step - loss: 0.3985 - acc: 0.8869 - val_loss: 0.2940 - val_a
cc: 0.9178

Epoch 00001: val_acc improved from -inf to 0.91780, saving model to weights/weights_softmax_regression.hdf5
Epoch 2/30
55000/55000 [==============================] - 1s 17us/step - loss: 0.3104 - acc: 0.9117 - val_loss: 0.2797 - val_a
cc: 0.9226

Epoch 00002: val_acc improved from 0.91780 to 0.92260, saving model to weights/weights_softmax_regression.hdf5
Epoch 3/30
55000/55000 [==============================] - 1s 17us/step - loss: 0.2947 - acc: 0.9167 - val_loss: 0.2784 - val_a
cc: 0.9226

Epoch 00003: val_acc did not improve
Epoch 4/30
55000/55000 [==============================] - 1s 17us/step - loss: 0.2868 - acc: 0.9192 - val_loss: 0.2837 - val_a
cc: 0.9214

Epoch 00004: val_acc did not improve
Epoch 5/30
55000/55000 [==============================] - 1s 17us/step - loss: 0.2814 - acc: 0.9209 - val_loss: 0.2726 - val_a
cc: 0.9254

Epoch 00005: val_acc improved from 0.92260 to 0.92540, saving model to weights/weights_softmax_regression.hdf5
Epoch 6/30
55000/55000 [==============================] - 1s 17us/step - loss: 0.2776 - acc: 0.9222 - val_loss: 0.2708 - val_a
cc: 0.9240

Epoch 00006: val_acc did not improve
Epoch 7/30
55000/55000 [==============================] - 1s 17us/step - loss: 0.2742 - acc: 0.9235 - val_loss: 0.2668 - val_a
cc: 0.9266

Epoch 00007: val_acc improved from 0.92540 to 0.92660, saving model to weights/weights_softmax_regression.hdf5
Epoch 8/30
55000/55000 [==============================] - 1s 17us/step - loss: 0.2721 - acc: 0.9232 - val_loss: 0.2694 - val_a
cc: 0.9266

Epoch 00008: val_acc did not improve
Epoch 9/30
55000/55000 [==============================] - 1s 17us/step - loss: 0.2690 - acc: 0.9244 - val_loss: 0.2663 - val_a
cc: 0.9280

Epoch 00009: val_acc improved from 0.92660 to 0.92800, saving model to weights/weights_softmax_regression.hdf5
Epoch 10/30
55000/55000 [==============================] - 1s 18us/step - loss: 0.2669 - acc: 0.9265 - val_loss: 0.2641 - val_a
cc: 0.9262

Epoch 00010: val_acc did not improve
Epoch 11/30
55000/55000 [==============================] - 1s 17us/step - loss: 0.2660 - acc: 0.9259 - val_loss: 0.2761 - val_a
cc: 0.9210

Epoch 00011: val_acc did not improve
Epoch 12/30
55000/55000 [==============================] - 1s 17us/step - loss: 0.2643 - acc: 0.9255 - val_loss: 0.2649 - val_a
cc: 0.9280

Epoch 00012: val_acc did not improve
Epoch 13/30
55000/55000 [==============================] - 1s 17us/step - loss: 0.2632 - acc: 0.9271 - val_loss: 0.2664 - val_a
cc: 0.9266

Epoch 00013: val_acc did not improve
Epoch 14/30
55000/55000 [==============================] - 1s 17us/step - loss: 0.2619 - acc: 0.9277 - val_loss: 0.2648 - val_a
cc: 0.9270

Epoch 00014: val_acc did not improve
Epoch 15/30
55000/55000 [==============================] - 1s 18us/step - loss: 0.2603 - acc: 0.9273 - val_loss: 0.2722 - val_a
cc: 0.9236

Epoch 00015: val_acc did not improve
Epoch 16/30
55000/55000 [==============================] - 1s 17us/step - loss: 0.2591 - acc: 0.9277 - val_loss: 0.2634 - val_a
cc: 0.9294

Epoch 00016: val_acc improved from 0.92800 to 0.92940, saving model to weights/weights_softmax_regression.hdf5
Epoch 17/30
55000/55000 [==============================] - 1s 17us/step - loss: 0.2583 - acc: 0.9281 - val_loss: 0.2677 - val_a
cc: 0.9278
```

```
Epoch 00017: val_acc did not improve
Epoch 18/30
55000/55000 [==============================] - 1s 17us/step - loss: 0.2577 - acc: 0.9283 - val_loss: 0.2633 - val_a
cc: 0.9282

Epoch 00018: val_acc did not improve
Epoch 19/30
55000/55000 [==============================] - 1s 17us/step - loss: 0.2565 - acc: 0.9286 - val_loss: 0.2655 - val_a
cc: 0.9282

Epoch 00019: val_acc did not improve
Epoch 20/30
55000/55000 [==============================] - 1s 17us/step - loss: 0.2566 - acc: 0.9287 - val_loss: 0.2708 - val_a
cc: 0.9252

Epoch 00020: val_acc did not improve
Epoch 21/30
55000/55000 [==============================] - 1s 17us/step - loss: 0.2554 - acc: 0.9287 - val_loss: 0.2744 - val_a
cc: 0.9226

Epoch 00021: val_acc did not improve
Epoch 22/30
55000/55000 [==============================] - 1s 17us/step - loss: 0.2543 - acc: 0.9290 - val_loss: 0.2656 - val_a
cc: 0.9278

Epoch 00022: val_acc did not improve
Epoch 23/30
55000/55000 [==============================] - 1s 17us/step - loss: 0.2538 - acc: 0.9292 - val_loss: 0.2721 - val_a
cc: 0.9242

Epoch 00023: val_acc did not improve
Epoch 24/30
55000/55000 [==============================] - 1s 17us/step - loss: 0.2534 - acc: 0.9292 - val_loss: 0.2712 - val_a
cc: 0.9264

Epoch 00024: val_acc did not improve
Epoch 25/30
55000/55000 [==============================] - 1s 17us/step - loss: 0.2519 - acc: 0.9295 - val_loss: 0.2719 - val_a
cc: 0.9240

Epoch 00025: val_acc did not improve
Epoch 26/30
55000/55000 [==============================] - 1s 17us/step - loss: 0.2516 - acc: 0.9297 - val_loss: 0.2732 - val_a
cc: 0.9270

Epoch 00026: val_acc did not improve
Epoch 27/30
55000/55000 [==============================] - 1s 17us/step - loss: 0.2523 - acc: 0.9291 - val_loss: 0.2770 - val_a
cc: 0.9258

Epoch 00027: val_acc did not improve
Epoch 28/30
55000/55000 [==============================] - 1s 17us/step - loss: 0.2513 - acc: 0.9295 - val_loss: 0.2827 - val_a
cc: 0.9222

Epoch 00028: val_acc did not improve
Epoch 29/30
55000/55000 [==============================] - 1s 18us/step - loss: 0.2499 - acc: 0.9303 - val_loss: 0.2675 - val_a
cc: 0.9276

Epoch 00029: val_acc did not improve
Epoch 30/30
55000/55000 [==============================] - 1s 17us/step - loss: 0.2505 - acc: 0.9299 - val_loss: 0.2700 - val_a
cc: 0.9280

Epoch 00030: val_acc did not improve
```

Out[185]:
```
<keras.callbacks.History at 0x13a5ef2b0>
```

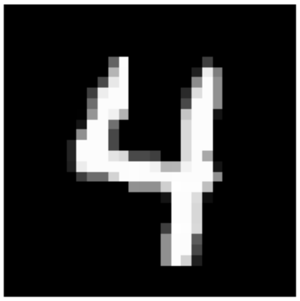
c) Evaluate performance on mnist test data

In [186]:
```
# accuracy on test data
test_accuracy = model.evaluate(mnist.test.images, mnist.test.labels, verbose=0)[1]
# accuracy on validation data
validation_accuracy = model.evaluate(mnist.validation.images, mnist.validation.labels, verbose=0)[1]

print("Accuracy on the MNIST test set: {}, validation accuracy: {}".format(test_accuracy, validation_accuracy))
```

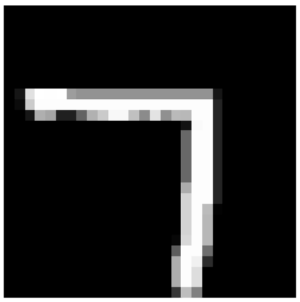
```
Accuracy on the MNIST test set: 0.9258, validation accuracy: 0.928
```

d) Plot 10 random images with true and predicted labels

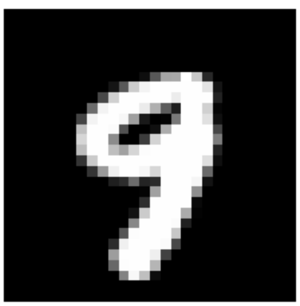
In [187]:
```
for i in range(10):
    batch = mnist.test.next_batch(1)
    image = np.asarray(batch[0]).reshape((28, 28))
    label = batch[1]

    plt.imshow(image, cmap='gray')
    plt.axis("off")
    plt.show()
    print("Correct label: {}".format(np.argmax(label)))
    print("Assigned label: {}".format(np.argmax(model.predict(image.reshape(1, 784)))))
```



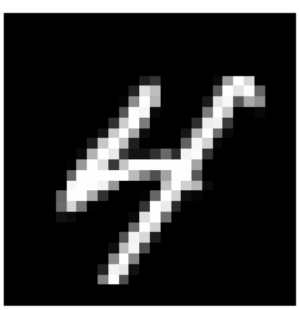
```
Correct label: 4
Assigned label: 4
```



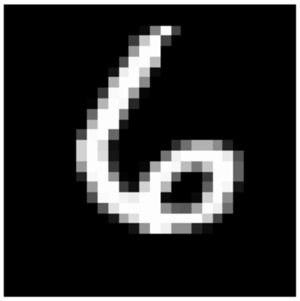
```
Correct label: 7
Assigned label: 7
```



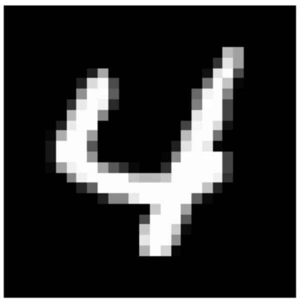
```
Correct label: 9
Assigned label: 9
```



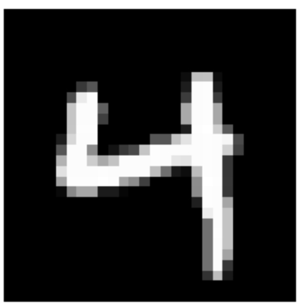
```
Correct label: 4
Assigned label: 4
```



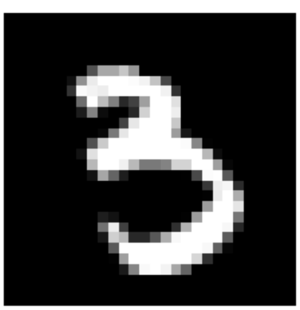
Correct label: 6
Assigned label: 6



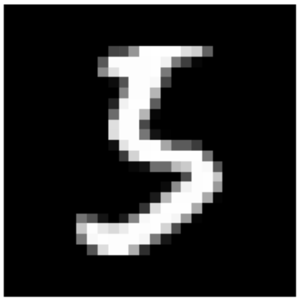
Correct label: 4
Assigned label: 4



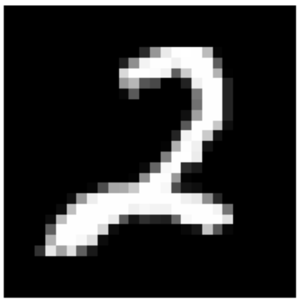
Correct label: 4
Assigned label: 4



Correct label: 3
Assigned label: 3

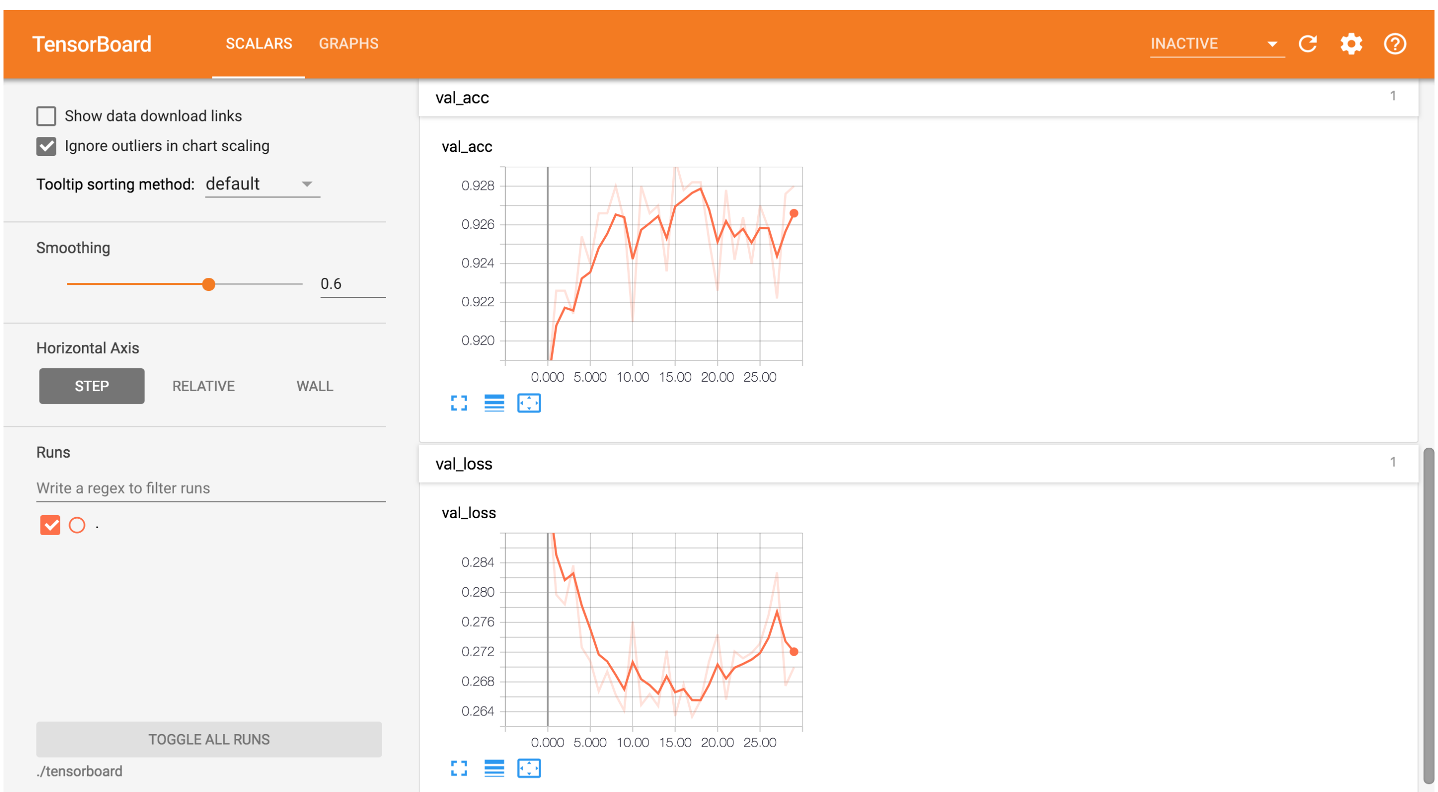


```
Correct label: 5
Assigned label: 5
```



```
Correct label: 2
Assigned label: 2
```

e) See the submitted screen shot - softmax_regression.png



f) - g) Load the best model saved by the program

In [188]:
```
# load the weights
model.load_weights(filepath='weights/weights_softmax_regression.hdf5')
# accuracy of best model on test data
test_accuracy_best = model.evaluate(mnist.test.images, mnist.test.labels, verbose=0)[1]
# accuracy of best model on validation data
validation_accuracy_best = model.evaluate(mnist.validation.images, mnist.validation.labels, verbose=0)[1]

print("Model with best validation accuracy\nAccuracy on the MNIST test set: {}, validation accuracy: {}".format(test
_accuracy_best, validation_accuracy_best))
```

```
Model with best validation accuracy
Accuracy on the MNIST test set: 0.9239, validation accuracy: 0.9294
```

Softmax Multi-Layer Perceptron on the MNIST Digits Data

Here we define both weight and bias variables and how they are to be initialized. Note that the weights are are distributed according to a standard normal distribution (mean = 0, std = 0.1). This random initialization helps avoid hidden units get stuck together, as units that start with the same value will be updated identically in the non-convolutional layers. In contrast, the bias variables are set to a small positive number--this is help prevent hidden units from starting out and getting stuck in the zero part of the ReLU.

In [32]:
```
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)
```

Next we create placeholders for the training data.

In [35]:
```
x = tf.placeholder(tf.float32, shape=[None, 784])
y_ = tf.placeholder(tf.float32, shape=[None, 10])
```

We create the first and only fully connected hidden layer.

In [36]:
```
W_h = weight_variable([784, 512])
b_h = bias_variable([512])
h = tf.nn.relu(tf.matmul(x, W_h) + b_h)
```

We create the output layer.

In [37]:
```
W_out = weight_variable([512, 10])
b_out = bias_variable([10])
y_MLP = tf.matmul(h, W_out) + b_out
```

We again use cross entropy loss on a softmax distribution on the outputs.

In [39]:
```
cross_entropy = tf.reduce_mean(
    tf.nn.softmax_cross_entropy_with_logits(labels=y_, logits=y_MLP))
```

For training we choose an Adam learning rate and update rule. We then run this for 20,000 iterations and evaluate our accuracy after training. Note this softmax MLP network does quite a bit bettter than our softmax regressor. The non-linear layer really helps makes sense of the data! But we can do better still...

In [40]:
```
train_step = tf.train.AdamOptimizer(1e-4).minimize(cross_entropy)
correct_prediction = tf.equal(tf.argmax(y_MLP,1), tf.argmax(y_,1))
accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))
sess.run(tf.global_variables_initializer())
for i in range(20000):
  batch = mnist.train.next_batch(50)
  if i%1000 == 0:
    train_accuracy = accuracy.eval(feed_dict={
        x:batch[0], y_: batch[1]})
    print("step %d, training accuracy %g"%(i, train_accuracy))
  train_step.run(feed_dict={x: batch[0], y_: batch[1]})

print("test accuracy %g"%accuracy.eval(feed_dict={
    x: mnist.test.images, y_: mnist.test.labels}))
```

```
step 0, training accuracy 0.14
step 1000, training accuracy 0.88
step 2000, training accuracy 0.92
step 3000, training accuracy 0.92
step 4000, training accuracy 1
step 5000, training accuracy 1
step 6000, training accuracy 0.98
step 7000, training accuracy 1
step 8000, training accuracy 0.96
step 9000, training accuracy 1
step 10000, training accuracy 0.92
step 11000, training accuracy 1
step 12000, training accuracy 1
step 13000, training accuracy 1
step 14000, training accuracy 1
step 15000, training accuracy 1
step 16000, training accuracy 1
step 17000, training accuracy 0.96
step 18000, training accuracy 1
step 19000, training accuracy 1
test accuracy 0.9787
```

MLP

a) - b) Re-implement MLP model and train it using adam optimizer

In [189]:
```
from keras.optimizers import Adam
from keras.initializers import RandomNormal, Constant
```

In [190]:
```
# initialize weight as values under normal distribution with mean=0, standard deviation=0.1
weight_initializer = RandomNormal(mean=0.0, stddev=0.1, seed=None)
# initialize bias as small constants
bias_initializer = Constant(value=0.1)

# declare a keras sequential model
model = Sequential([
    # hidden layer
    Dense(512, input_dim=784, kernel_initializer=weight_initializer, bias_initializer=bias_initializer),
    # relu activation function for outputs of hidden layer
    Activation('relu'),
    # output layer
    Dense(10, input_dim=512, kernel_initializer=weight_initializer, bias_initializer=bias_initializer),
    # softmax function to get the 10 probabilities
    Activation('softmax')
])

# define the optimizer as adam optimizer
# the model implemented in tensorflow has a learning rate of 1e-4, and other parameters are as default
adam = Adam(lr=1e-4, beta_1=0.9, beta_2=0.999, epsilon=None, decay=0.0, amsgrad=False)

# configure the training process
# use adam optimizer with cross entropy loss
model.compile(optimizer=adam,
              loss='categorical_crossentropy',
              metrics=['accuracy'])

# checkpoint to save the best model
checkpointer = ModelCheckpoint(filepath='weights/weights_mlp.hdf5', monitor='val_acc', verbose=1, save_best_only=Tru
e, mode='max')
# another call back function to write a log of the data for tensorboard visualization
tensorboard = TensorBoard(log_dir='tensorboard/', histogram_freq=0, write_graph=True, write_images=True)

# start training for 30 epochs
# batch size is 50
model.fit(x=mnist.train.images,
          y=mnist.train.labels,
          validation_data=(mnist.validation.images, mnist.validation.labels),
          callbacks=[checkpointer, tensorboard],
          verbose=1,
          epochs=30,
          batch_size=50)
```

```
Train on 55000 samples, validate on 5000 samples
Epoch 1/30
55000/55000 [==============================] - 5s 87us/step - loss: 0.5562 - acc: 0.8361 - val_loss: 0.2686 - val_a
cc: 0.9288

Epoch 00001: val_acc improved from -inf to 0.92880, saving model to weights/weights_mlp.hdf5
Epoch 2/30
55000/55000 [==============================] - 5s 83us/step - loss: 0.2412 - acc: 0.9315 - val_loss: 0.1975 - val_a
cc: 0.9472

Epoch 00002: val_acc improved from 0.92880 to 0.94720, saving model to weights/weights_mlp.hdf5
Epoch 3/30
55000/55000 [==============================] - 5s 85us/step - loss: 0.1862 - acc: 0.9485 - val_loss: 0.1634 - val_a
cc: 0.9562

Epoch 00003: val_acc improved from 0.94720 to 0.95620, saving model to weights/weights_mlp.hdf5
Epoch 4/30
55000/55000 [==============================] - 4s 79us/step - loss: 0.1523 - acc: 0.9582 - val_loss: 0.1428 - val_a
cc: 0.9592

Epoch 00004: val_acc improved from 0.95620 to 0.95920, saving model to weights/weights_mlp.hdf5
Epoch 5/30
55000/55000 [==============================] - 4s 79us/step - loss: 0.1284 - acc: 0.9651 - val_loss: 0.1239 - val_a
cc: 0.9652

Epoch 00005: val_acc improved from 0.95920 to 0.96520, saving model to weights/weights_mlp.hdf5
Epoch 6/30
55000/55000 [==============================] - 5s 84us/step - loss: 0.1101 - acc: 0.9703 - val_loss: 0.1139 - val_a
cc: 0.9678

Epoch 00006: val_acc improved from 0.96520 to 0.96780, saving model to weights/weights_mlp.hdf5
Epoch 7/30
55000/55000 [==============================] - 5s 88us/step - loss: 0.0960 - acc: 0.9745 - val_loss: 0.1031 - val_a
cc: 0.9704

Epoch 00007: val_acc improved from 0.96780 to 0.97040, saving model to weights/weights_mlp.hdf5
Epoch 8/30
55000/55000 [==============================] - 5s 86us/step - loss: 0.0841 - acc: 0.9779 - val_loss: 0.0934 - val_a
cc: 0.9728

Epoch 00008: val_acc improved from 0.97040 to 0.97280, saving model to weights/weights_mlp.hdf5
Epoch 9/30
55000/55000 [==============================] - 5s 87us/step - loss: 0.0744 - acc: 0.9808 - val_loss: 0.0865 - val_a
cc: 0.9744

Epoch 00009: val_acc improved from 0.97280 to 0.97440, saving model to weights/weights_mlp.hdf5
Epoch 10/30
55000/55000 [==============================] - 5s 89us/step - loss: 0.0660 - acc: 0.9833 - val_loss: 0.0843 - val_a
cc: 0.9752

Epoch 00010: val_acc improved from 0.97440 to 0.97520, saving model to weights/weights_mlp.hdf5
Epoch 11/30
55000/55000 [==============================] - 5s 93us/step - loss: 0.0588 - acc: 0.9848 - val_loss: 0.0786 - val_a
cc: 0.9750

Epoch 00011: val_acc did not improve
Epoch 12/30
55000/55000 [==============================] - 5s 93us/step - loss: 0.0522 - acc: 0.9869 - val_loss: 0.0767 - val_a
cc: 0.9758

Epoch 00012: val_acc improved from 0.97520 to 0.97580, saving model to weights/weights_mlp.hdf5
Epoch 13/30
55000/55000 [==============================] - 5s 92us/step - loss: 0.0468 - acc: 0.9886 - val_loss: 0.0740 - val_a
cc: 0.9774

Epoch 00013: val_acc improved from 0.97580 to 0.97740, saving model to weights/weights_mlp.hdf5
Epoch 14/30
55000/55000 [==============================] - 5s 86us/step - loss: 0.0418 - acc: 0.9901 - val_loss: 0.0717 - val_a
cc: 0.9784

Epoch 00014: val_acc improved from 0.97740 to 0.97840, saving model to weights/weights_mlp.hdf5
Epoch 15/30
55000/55000 [==============================] - 4s 80us/step - loss: 0.0379 - acc: 0.9911 - val_loss: 0.0687 - val_a
cc: 0.9788

Epoch 00015: val_acc improved from 0.97840 to 0.97880, saving model to weights/weights_mlp.hdf5
Epoch 16/30
55000/55000 [==============================] - 4s 80us/step - loss: 0.0337 - acc: 0.9925 - val_loss: 0.0681 - val_a
cc: 0.9792

Epoch 00016: val_acc improved from 0.97880 to 0.97920, saving model to weights/weights_mlp.hdf5
Epoch 17/30
55000/55000 [==============================] - 4s 79us/step - loss: 0.0300 - acc: 0.9937 - val_loss: 0.0659 - val_a
cc: 0.9798
```

```
Epoch 00017: val_acc improved from 0.97920 to 0.97980, saving model to weights/weights_mlp.hdf5
Epoch 18/30
55000/55000 [==============================] - 4s 78us/step - loss: 0.0272 - acc: 0.9948 - val_loss: 0.0632 - val_a
cc: 0.9808

Epoch 00018: val_acc improved from 0.97980 to 0.98080, saving model to weights/weights_mlp.hdf5
Epoch 19/30
55000/55000 [==============================] - 4s 77us/step - loss: 0.0243 - acc: 0.9957 - val_loss: 0.0629 - val_a
cc: 0.9812

Epoch 00019: val_acc improved from 0.98080 to 0.98120, saving model to weights/weights_mlp.hdf5
Epoch 20/30
55000/55000 [==============================] - 4s 77us/step - loss: 0.0217 - acc: 0.9963 - val_loss: 0.0626 - val_a
cc: 0.9816

Epoch 00020: val_acc improved from 0.98120 to 0.98160, saving model to weights/weights_mlp.hdf5
Epoch 21/30
55000/55000 [==============================] - 4s 77us/step - loss: 0.0194 - acc: 0.9973 - val_loss: 0.0640 - val_a
cc: 0.9792

Epoch 00021: val_acc did not improve
Epoch 22/30
55000/55000 [==============================] - 4s 77us/step - loss: 0.0176 - acc: 0.9974 - val_loss: 0.0623 - val_a
cc: 0.9794

Epoch 00022: val_acc did not improve
Epoch 23/30
55000/55000 [==============================] - 4s 76us/step - loss: 0.0156 - acc: 0.9982 - val_loss: 0.0599 - val_a
cc: 0.9818

Epoch 00023: val_acc improved from 0.98160 to 0.98180, saving model to weights/weights_mlp.hdf5
Epoch 24/30
55000/55000 [==============================] - 4s 76us/step - loss: 0.0140 - acc: 0.9986 - val_loss: 0.0597 - val_a
cc: 0.9812

Epoch 00024: val_acc did not improve
Epoch 25/30
55000/55000 [==============================] - 4s 80us/step - loss: 0.0124 - acc: 0.9988 - val_loss: 0.0589 - val_a
cc: 0.9828

Epoch 00025: val_acc improved from 0.98180 to 0.98280, saving model to weights/weights_mlp.hdf5
Epoch 26/30
55000/55000 [==============================] - 5s 91us/step - loss: 0.0111 - acc: 0.9993 - val_loss: 0.0600 - val_a
cc: 0.9814

Epoch 00026: val_acc did not improve
Epoch 27/30
55000/55000 [==============================] - 5s 100us/step - loss: 0.0099 - acc: 0.9993 - val_loss: 0.0587 - val_
acc: 0.9832

Epoch 00027: val_acc improved from 0.98280 to 0.98320, saving model to weights/weights_mlp.hdf5
Epoch 28/30
55000/55000 [==============================] - 6s 101us/step - loss: 0.0088 - acc: 0.9994 - val_loss: 0.0589 - val_
acc: 0.9824

Epoch 00028: val_acc did not improve
Epoch 29/30
55000/55000 [==============================] - 6s 108us/step - loss: 0.0077 - acc: 0.9994 - val_loss: 0.0593 - val_
acc: 0.9824

Epoch 00029: val_acc did not improve
Epoch 30/30
55000/55000 [==============================] - 6s 103us/step - loss: 0.0070 - acc: 0.9996 - val_loss: 0.0596 - val_
acc: 0.9828

Epoch 00030: val_acc did not improve
```

Out[190]: <keras.callbacks.History at 0x14040ae80>

c) Evaluate performance on mnist test data

In [191]:

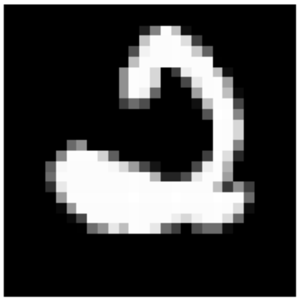
```
# accuracy on test data
test_accuracy = model.evaluate(mnist.test.images, mnist.test.labels, verbose=0)[1]
# accuracy on validation data
validation_accuracy = model.evaluate(mnist.validation.images, mnist.validation.labels, verbose=0)[1]

print("Accuracy on the MNIST test set: {}, validation accuracy: {}".format(test_accuracy, validation_accuracy))
```

```
Accuracy on the MNIST test set: 0.9803, validation accuracy: 0.9828
```

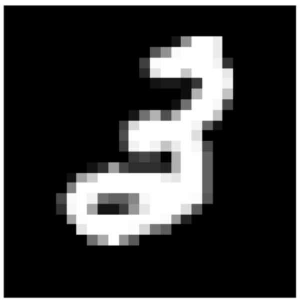
d) Plot 10 random images with true and predicted labels

In [192]:

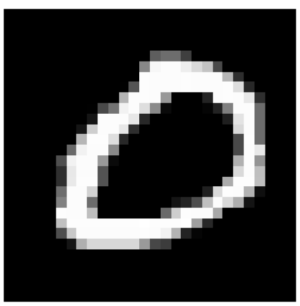
```
for i in range(10):
    batch = mnist.test.next_batch(1)
    image = np.asarray(batch[0]).reshape((28, 28))
    label = batch[1]

    plt.imshow(image, cmap='gray')
    plt.axis("off")
    plt.show()
    print("Correct label: {}".format(np.argmax(label)))
    print("Assigned label: {}".format(np.argmax(model.predict(image.reshape(1, 784)))))
```



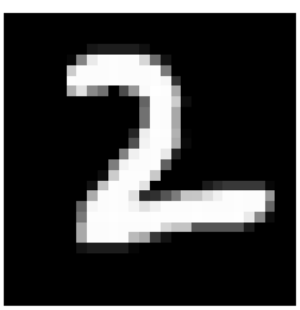
```
Correct label: 2
Assigned label: 2
```



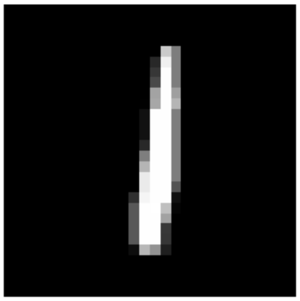
```
Correct label: 3
Assigned label: 3
```



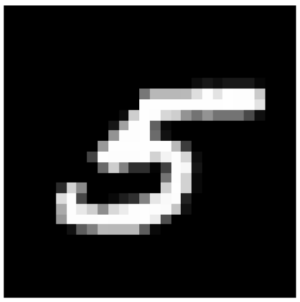
```
Correct label: 0
Assigned label: 0
```



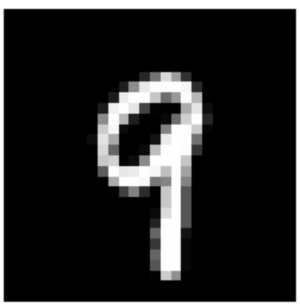
```
Correct label: 2
Assigned label: 2
```



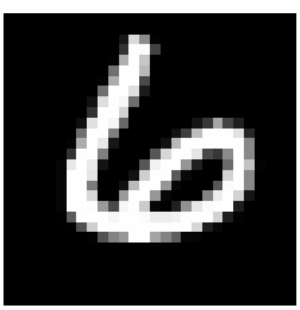
```
Correct label: 1
Assigned label: 1
```



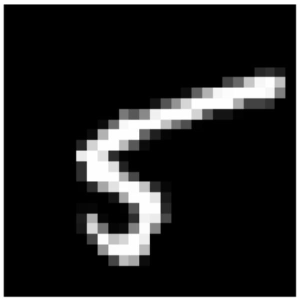
```
Correct label: 5
Assigned label: 5
```



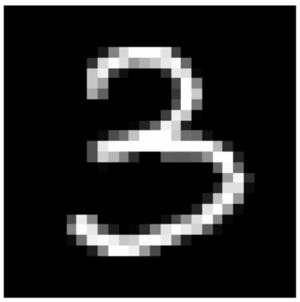
```
Correct label: 9
Assigned label: 9
```



```
Correct label: 6
Assigned label: 6
```

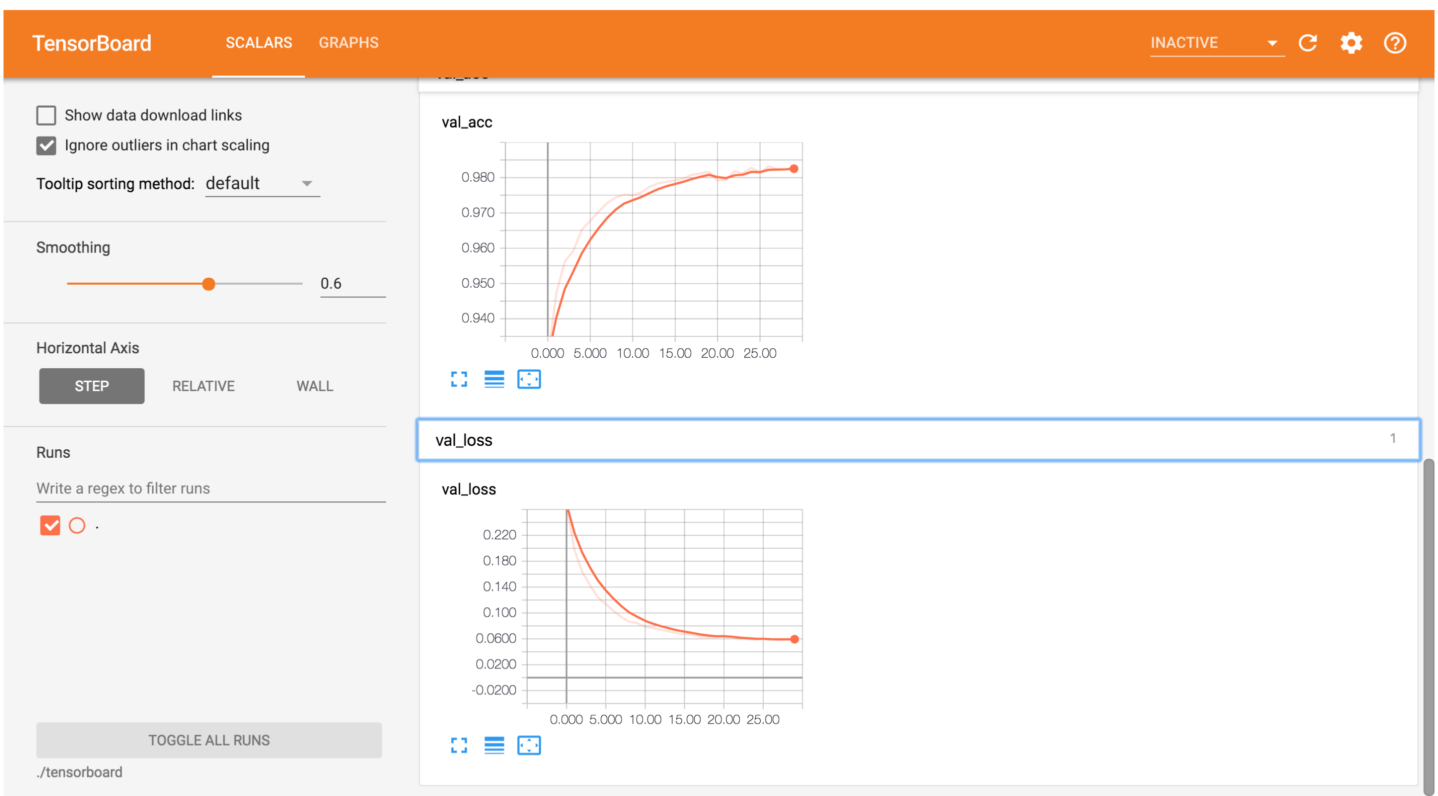


```
Correct label: 5
Assigned label: 5
```



```
Correct label: 3
Assigned label: 3
```

e) See the submitted screen shot - mlp.png



f) - g) Load the best model saved by the program

In [193]:
```
# load weights of best model
model.load_weights(filepath='weights/weights_mlp.hdf5')
# accuracy of best model on test data
test_accuracy_best = model.evaluate(mnist.test.images, mnist.test.labels, verbose=0)[1]
# accuracy of best model on validation data
validation_accuracy_best = model.evaluate(mnist.validation.images, mnist.validation.labels, verbose=0)[1]
print("Model with best validation accuracy\nAccuracy on the MNIST test set: {}, validation accuracy: {}".format(test
_accuracy_best, validation_accuracy_best))
```

```
Model with best validation accuracy
Accuracy on the MNIST test set: 0.9797, validation accuracy: 0.9832
```

A Simple Convolutional Neural Network: LeNet

Here we make our first CNN. It's quite simple network, but it's surprisingly good at this handwritten digit recognition task. This a variant on Yann LeCun's CNN network that really helped to move deep learning forward.

We define both weight and bias variables and how they are to be initialized. Note that the weights are are distributed according to a standard normal distribution (mean = 0, std = 0.1). This random initialization helps avoid hidden units get stuck together, as units that start with the same value will be updated identically in the non-convolutional layers. In contrast, the bias variables are set to a small positive number--this is help prevent hidden units from starting out and getting stuck in the zero part of the ReLu.

In [63]:
```
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)
```

Next we define how the convolution is to be computed and the extent and type of pooling. The convolution will use a 5x5 kernel and will pad the image with zeros around the edges and use a stride of 1 pixel so that the resulting image (after convolution) has the same size as the original input image. The network will learn the weights for a stack of 32 separate kernels along with 32 bias variables. Finally, after the ReLu is performed the result will be under go 2x2 max pooling, thus halfing both dimensions of the image. The choices for the stride, padding, and pooling are not parameters that the network needs to estimate. Rather these are termed "hyperparamters" that are usually set by the network designer.

In [64]:
```
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')
```

This creates the weight and bias variables for the first convolutional layer as described above. Note the output has depth 32, so there will be 32 feature images after this layer.

In [65]:
```
W_conv1 = weight_variable([5, 5, 1, 32])
b_conv1 = bias_variable([32])
```

Unlike for our softmax regressor above, here we need keep the images as images and not collapse these into vectors; this allows us to perform the 2D convolution.

In [66]:
```
x_image = tf.reshape(x, [-1,28,28,1])
```

Finally, we define are first layer of our CNN!

In [67]:
```
h_conv1 = tf.nn.relu(conv2d(x_image, W_conv1) + b_conv1)
h_pool1 = max_pool_2x2(h_conv1)
```

And wasting no time, we define are second layer. The second layer will have to process 32 feature images coming out of the first layer. Note that the images input to this layer have $\frac{1}{4}$ the number of pixels as the original input images due to the 2x2 pooling in the previous layer. Note that convolution layer NOT fully connected as our previous hidden layers have been. A unit in the output layer has a limited "receptive field." Its connections to the input layer are spatially limited by the kernel (or filter) size. Also, because of weight sharing in convolutional layers, the number of parameters for a convolutional is the size of the kernel x the depth of the input layer x depth of the output layer + depth of the output layer. So for the second layer of our ConvNet, we have 5 x 5 x 32 x 64 + 64 = 51,264 parameters.

In [68]:
```
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)
h_pool2 = max_pool_2x2(h_conv2)
```

After the pooling stage of our second convolutional layer, we have 64 7x7 "feature" images. In one penultimate fully connected hidden layer, we are going to map these feature imges to a 1024 dimensional feature space. Note we need to flatten these feature images to do this.

In [69]:
```
W_fc1 = weight_variable([7 * 7 * 64, 1024])
b_fc1 = bias_variable([1024])

h_pool2_flat = tf.reshape(h_pool2, [-1, 7*7*64])
h_fc1 = tf.nn.relu(tf.matmul(h_pool2_flat, W_fc1) + b_fc1)
```

Dropout is added here, although it is not really needed for such small network.

In [70]:
```
keep_prob = tf.placeholder(tf.float32)
h_fc1_drop = tf.nn.dropout(h_fc1, keep_prob)
```

We have a final linear output layer mapping features to scores topped off with a softmax cross entropy loss function, as explained earlier.

In [71]:
```
W_fc2 = weight_variable([1024, 10])
b_fc2 = bias_variable([10])

y_conv = tf.matmul(h_fc1_drop, W_fc2) + b_fc2
```

In [72]:
```
cross_entropy = tf.reduce_mean(
    tf.nn.softmax_cross_entropy_with_logits(labels=y_, logits=y_conv))
```

For training we choose an Adam learning rate and update rule. We then run this for 20,000 iterations and evaluate our accuracy after training.

In [73]:
```
train_step = tf.train.AdamOptimizer(1e-4).minimize(cross_entropy)
correct_prediction = tf.equal(tf.argmax(y_conv,1), tf.argmax(y_,1))
accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))
sess.run(tf.global_variables_initializer())
for i in range(20000):
  batch = mnist.train.next_batch(50)
  if i%1000 == 0:
    train_accuracy = accuracy.eval(feed_dict={
        x:batch[0], y_: batch[1], keep_prob: 1.0})
    print("step %d, training accuracy %g"%(i, train_accuracy))
  train_step.run(feed_dict={x: batch[0], y_: batch[1], keep_prob: 0.5})

print("test accuracy %g"%accuracy.eval(feed_dict={
    x: mnist.test.images, y_: mnist.test.labels, keep_prob: 1.0}))
```

```
step 0, training accuracy 0.12
step 1000, training accuracy 1
step 2000, training accuracy 0.96
step 3000, training accuracy 0.96
step 4000, training accuracy 1
step 5000, training accuracy 0.98
step 6000, training accuracy 0.96
step 7000, training accuracy 1
step 8000, training accuracy 1
step 9000, training accuracy 1
step 10000, training accuracy 1
step 11000, training accuracy 1
step 12000, training accuracy 1
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 0.9922
```

We add an output to compuational graph that computes the label probabilities.

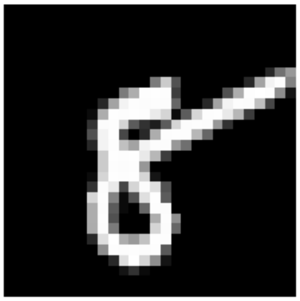
In [74]:
```
y_probs = tf.nn.softmax(logits=y_conv, name=None)
```

In [75]:
```
print("test accuracy %g"%accuracy.eval(feed_dict={
    x: mnist.test.images, y_: mnist.test.labels, keep_prob: 1.0}))
```

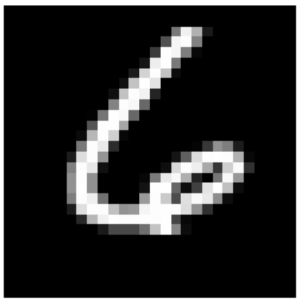
```
test accuracy 0.9922
```

Next we step through some test examples and see how well the network is doing.

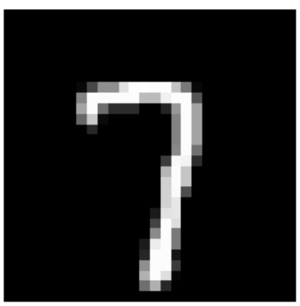
In [76]:
```
for i in range(5):
    batch = mnist.test.next_batch(1)
    image = np.asarray(batch[0]).reshape((28, 28))
    label = batch[1]

    plt.imshow(image, cmap='gray')
    plt.axis("off")
    plt.show()
    print("Label = ", label)
    print("Class probabilities = ", y_probs.eval(feed_dict={
        x: batch[0], y_: batch[1], keep_prob: 1.0}))
```



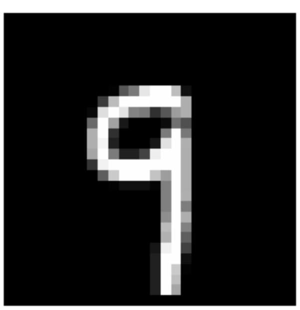
```
Label =  [[0. 0. 0. 0. 0. 0. 0. 0. 1. 0.]]
Class probabilities =  [[5.5720966e-11 2.1611019e-15 2.7565496e-13 1.0915743e-12 1.5609882e-09
  1.1711752e-05 4.4562470e-10 6.7399352e-15 9.9998832e-01 8.4068614e-13]]
```



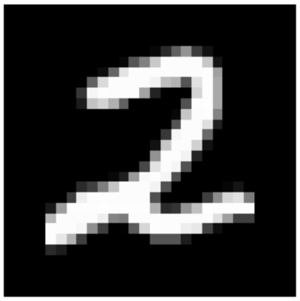
```
Label =  [[0. 0. 0. 0. 0. 0. 1. 0. 0. 0.]]
Class probabilities =  [[1.2043487e-09 1.5353710e-11 4.5305044e-12 1.6923188e-12 1.5912740e-07
  4.1741839e-11 9.9999988e-01 2.8309913e-14 5.4157507e-12 4.8473680e-11]]
```



```
Label =  [[0. 0. 0. 0. 0. 0. 0. 1. 0. 0.]]
Class probabilities =  [[5.4806799e-09 4.8692396e-08 1.2130279e-06 6.6668832e-08 3.8204265e-10
  6.9664642e-11 1.5919462e-11 9.9999321e-01 9.7107633e-10 5.5070896e-06]]
```



```
Label =  [[0. 0. 0. 0. 0. 0. 0. 0. 0. 1.]]
Class probabilities =  [[8.2692798e-14 2.1852979e-14 2.8715599e-15 1.4701707e-10 4.1126889e-09
  2.1203433e-10 6.4222917e-16 6.7994235e-11 7.9277740e-10 1.0000000e+00]]
```



```
Label =  [[0. 0. 1. 0. 0. 0. 0. 0. 0. 0.]]
Class probabilities =  [[1.8989047e-15 8.6311562e-11 1.0000000e+00 1.1673494e-11 4.5530964e-16
  5.2712527e-18 2.1630724e-16 1.9241346e-11 1.3348978e-11 1.1909466e-13]]
```

CNN

a) -b) Re-implement the cnn and train cnn using keras

In [176]:
```python
from keras.layers import Conv2D, MaxPooling2D, Flatten, Dropout
```

In [177]:
```python
# initialize weight as values under normal distribution with mean=0, standard deviation=0.1
weight_initializer = RandomNormal(mean=0.0, stddev=0.1, seed=None)
# initialize bias as small constants 0.1
bias_initializer = Constant(value=0.1)

# declare a sequential model
model = Sequential()

# the first hidden layer, 32 kernels
model.add(Conv2D(filters=32, kernel_size=(5, 5),
                 activation='relu',
                 padding='same',
                 strides=(1, 1),
                 kernel_initializer=weight_initializer,
                 bias_initializer=bias_initializer,
                 input_shape=(28, 28, 1)))

# max pooling layer, pool size 2*2, stride 2
model.add(MaxPooling2D(pool_size=(2, 2)))

# the second hidden layer, 64 kernels
model.add(Conv2D(filters=64, kernel_size=(5, 5),
                activation='relu',
                padding='same',
                kernel_initializer=weight_initializer,
                bias_initializer=bias_initializer,
                strides=(1, 1)))

# another max pooling layer, pool size 2*2, stride 2
model.add(MaxPooling2D(pool_size=(2, 2)))

# flat the output from last pooling layer for the last dense hidden layer
model.add(Flatten())

# a fully connected hidden layer
model.add(Dense(units=1024,
               activation='relu',
               kernel_initializer=weight_initializer,
               bias_initializer=bias_initializer))

# add dropout layer
model.add(Dropout(0.5))

# another fully connected layer  and then use softmax to compute the 10 class probabilities
model.add(Dense(units=10,
               activation='softmax',
               kernel_initializer=weight_initializer,
               bias_initializer=bias_initializer))
```

In [178]:
```
# summarise the model
model.summary()
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
conv2d_27 (Conv2D)           (None, 28, 28, 32)        832       
_________________________________________________________________
max_pooling2d_19 (MaxPooling (None, 14, 14, 32)        0         
_________________________________________________________________
conv2d_28 (Conv2D)           (None, 14, 14, 64)        51264     
_________________________________________________________________
max_pooling2d_20 (MaxPooling (None, 7, 7, 64)          0         
_________________________________________________________________
flatten_9 (Flatten)          (None, 3136)              0         
_________________________________________________________________
dense_31 (Dense)             (None, 1024)              3212288   
_________________________________________________________________
dropout_2 (Dropout)          (None, 1024)              0         
_________________________________________________________________
dense_32 (Dense)             (None, 10)                10250     
=================================================================
Total params: 3,274,634
Trainable params: 3,274,634
Non-trainable params: 0
_________________________________________________________________
```

In [179]:
```
# define the adam optimizer
# the model implemented in tensorflow has a learning rate of 1e-4, and other parameters are as default
adam = Adam(lr=1e-4, beta_1=0.9, beta_2=0.999, epsilon=None, decay=0.0, amsgrad=False)

# configure the training process, using adam optimizer with cross entropy loss
model.compile(optimizer=adam,
              loss='categorical_crossentropy',
              metrics=['accuracy'])

# checkpoint to save the model with best validation accuracy
checkpointer = ModelCheckpoint(filepath='weights/weights_cnn.hdf5', monitor='val_acc', verbose=1, save_best_only=Tru
e, mode='max')
# another callback function to record a log of the process for tensorflow visualization
tensorboard = TensorBoard(log_dir='tensorboard/', histogram_freq=0, write_graph=False, write_images=True)

# start training for 15 epochs, needs to reshape training data to fit the network input shape
# as implemented in tensorflow: batch size is 50
model.fit(x=(mnist.train.images.reshape(55000, 28, 28, 1)),
          y=mnist.train.labels,
          validation_data=(mnist.validation.images.reshape(5000, 28, 28, 1), mnist.validation.labels),
          callbacks=[checkpointer, tensorboard],
          verbose=1,
          epochs=15,
          batch_size=50)
```

```
Train on 55000 samples, validate on 5000 samples
Epoch 1/15
55000/55000 [==============================] - 97s 2ms/step - loss: 0.8692 - acc: 0.8286 - val_loss: 0.1305 - val_a
cc: 0.9612

Epoch 00001: val_acc improved from -inf to 0.96120, saving model to weights/weights_cnn.hdf5
Epoch 2/15
55000/55000 [==============================] - 107s 2ms/step - loss: 0.1670 - acc: 0.9475 - val_loss: 0.0805 - val_
acc: 0.9774

Epoch 00002: val_acc improved from 0.96120 to 0.97740, saving model to weights/weights_cnn.hdf5
Epoch 3/15
55000/55000 [==============================] - 112s 2ms/step - loss: 0.1102 - acc: 0.9659 - val_loss: 0.0582 - val_
acc: 0.9814

Epoch 00003: val_acc improved from 0.97740 to 0.98140, saving model to weights/weights_cnn.hdf5
Epoch 4/15
55000/55000 [==============================] - 110s 2ms/step - loss: 0.0818 - acc: 0.9738 - val_loss: 0.0553 - val_
acc: 0.9832

Epoch 00004: val_acc improved from 0.98140 to 0.98320, saving model to weights/weights_cnn.hdf5
Epoch 5/15
55000/55000 [==============================] - 110s 2ms/step - loss: 0.0663 - acc: 0.9788 - val_loss: 0.0452 - val_
acc: 0.9866

Epoch 00005: val_acc improved from 0.98320 to 0.98660, saving model to weights/weights_cnn.hdf5
Epoch 6/15
55000/55000 [==============================] - 112s 2ms/step - loss: 0.0532 - acc: 0.9830 - val_loss: 0.0433 - val_
acc: 0.9868

Epoch 00006: val_acc improved from 0.98660 to 0.98680, saving model to weights/weights_cnn.hdf5
Epoch 7/15
55000/55000 [==============================] - 112s 2ms/step - loss: 0.0456 - acc: 0.9849 - val_loss: 0.0402 - val_
acc: 0.9876

Epoch 00007: val_acc improved from 0.98680 to 0.98760, saving model to weights/weights_cnn.hdf5
Epoch 8/15
55000/55000 [==============================] - 112s 2ms/step - loss: 0.0372 - acc: 0.9881 - val_loss: 0.0406 - val_
acc: 0.9880

Epoch 00008: val_acc improved from 0.98760 to 0.98800, saving model to weights/weights_cnn.hdf5
Epoch 9/15
55000/55000 [==============================] - 114s 2ms/step - loss: 0.0325 - acc: 0.9899 - val_loss: 0.0401 - val_
acc: 0.9888

Epoch 00009: val_acc improved from 0.98800 to 0.98880, saving model to weights/weights_cnn.hdf5
Epoch 10/15
55000/55000 [==============================] - 112s 2ms/step - loss: 0.0275 - acc: 0.9910 - val_loss: 0.0387 - val_
acc: 0.9890

Epoch 00010: val_acc improved from 0.98880 to 0.98900, saving model to weights/weights_cnn.hdf5
Epoch 11/15
55000/55000 [==============================] - 111s 2ms/step - loss: 0.0234 - acc: 0.9923 - val_loss: 0.0363 - val_
acc: 0.9908

Epoch 00011: val_acc improved from 0.98900 to 0.99080, saving model to weights/weights_cnn.hdf5
Epoch 12/15
55000/55000 [==============================] - 113s 2ms/step - loss: 0.0192 - acc: 0.9937 - val_loss: 0.0339 - val_
acc: 0.9908

Epoch 00012: val_acc improved from 0.99080 to 0.99080, saving model to weights/weights_cnn.hdf5
Epoch 13/15
55000/55000 [==============================] - 114s 2ms/step - loss: 0.0191 - acc: 0.9938 - val_loss: 0.0347 - val_
acc: 0.9904

Epoch 00013: val_acc did not improve
Epoch 14/15
55000/55000 [==============================] - 108s 2ms/step - loss: 0.0154 - acc: 0.9949 - val_loss: 0.0359 - val_
acc: 0.9906

Epoch 00014: val_acc did not improve
Epoch 15/15
55000/55000 [==============================] - 97s 2ms/step - loss: 0.0147 - acc: 0.9953 - val_loss: 0.0348 - val_a
cc: 0.9910

Epoch 00015: val_acc improved from 0.99080 to 0.99100, saving model to weights/weights_cnn.hdf5
```

```
Out[179]: <keras.callbacks.History at 0x13acd5dd8>
```

c) Evaluate the model on MNIST test data

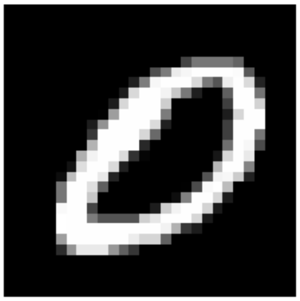
In [180]:
```
# keras automatically turns off dropout under test mode
# accuracy on test data
test_accuracy = model.evaluate(mnist.test.images.reshape(10000, 28, 28, 1), mnist.test.labels, verbose=0)[1]
# accuracy on validation data
validation_accuracy = model.evaluate(mnist.validation.images.reshape(5000, 28, 28, 1), mnist.validation.labels, verb
ose=0)[1]
print("Accuracy on the MNIST test set: {}, validation accuracy: {}".format(test_accuracy, validation_accuracy))
```

```
Accuracy on the MNIST test set: 0.9915, validation accuracy: 0.991
```

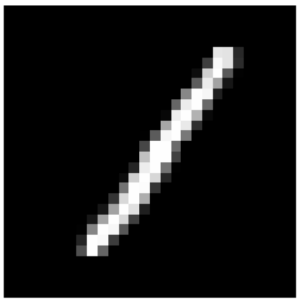
d) Plot 10 random images with true and predicted labels

In [181]:

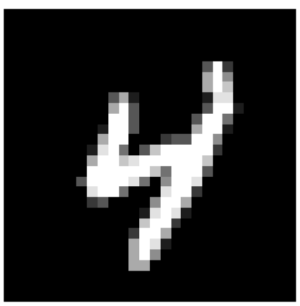
```
for i in range(10):
    batch = mnist.test.next_batch(1)
    image = np.asarray(batch[0]).reshape((28, 28))
    label = batch[1]

    plt.imshow(image, cmap='gray')
    plt.axis("off")
    plt.show()
    print("Correct label: {}".format(np.argmax(label)))
    print("Assigned label: {}".format(np.argmax(model.predict(image.reshape(1, 28, 28, 1)))))
```



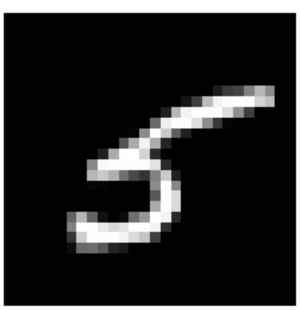
Correct label: 0
Assigned label: 0



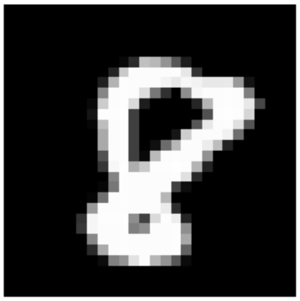
Correct label: 1
Assigned label: 1



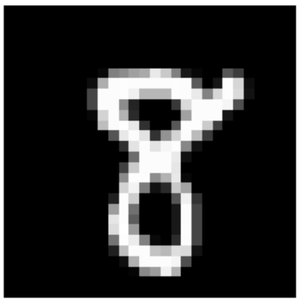
Correct label: 4
Assigned label: 4



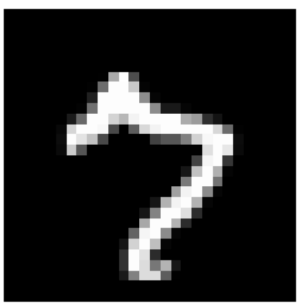
Correct label: 5
Assigned label: 5



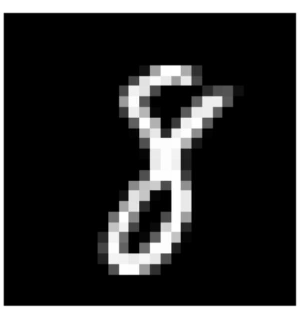
Correct label: 8
Assigned label: 8



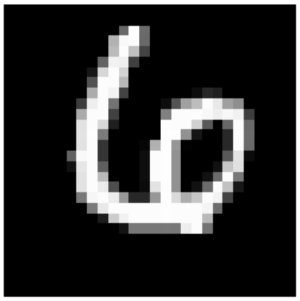
Correct label: 8
Assigned label: 8



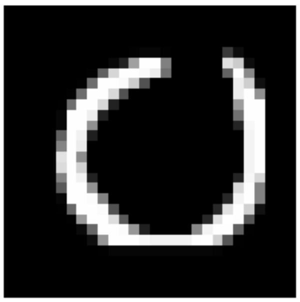
Correct label: 7
Assigned label: 7



Correct label: 8
Assigned label: 8

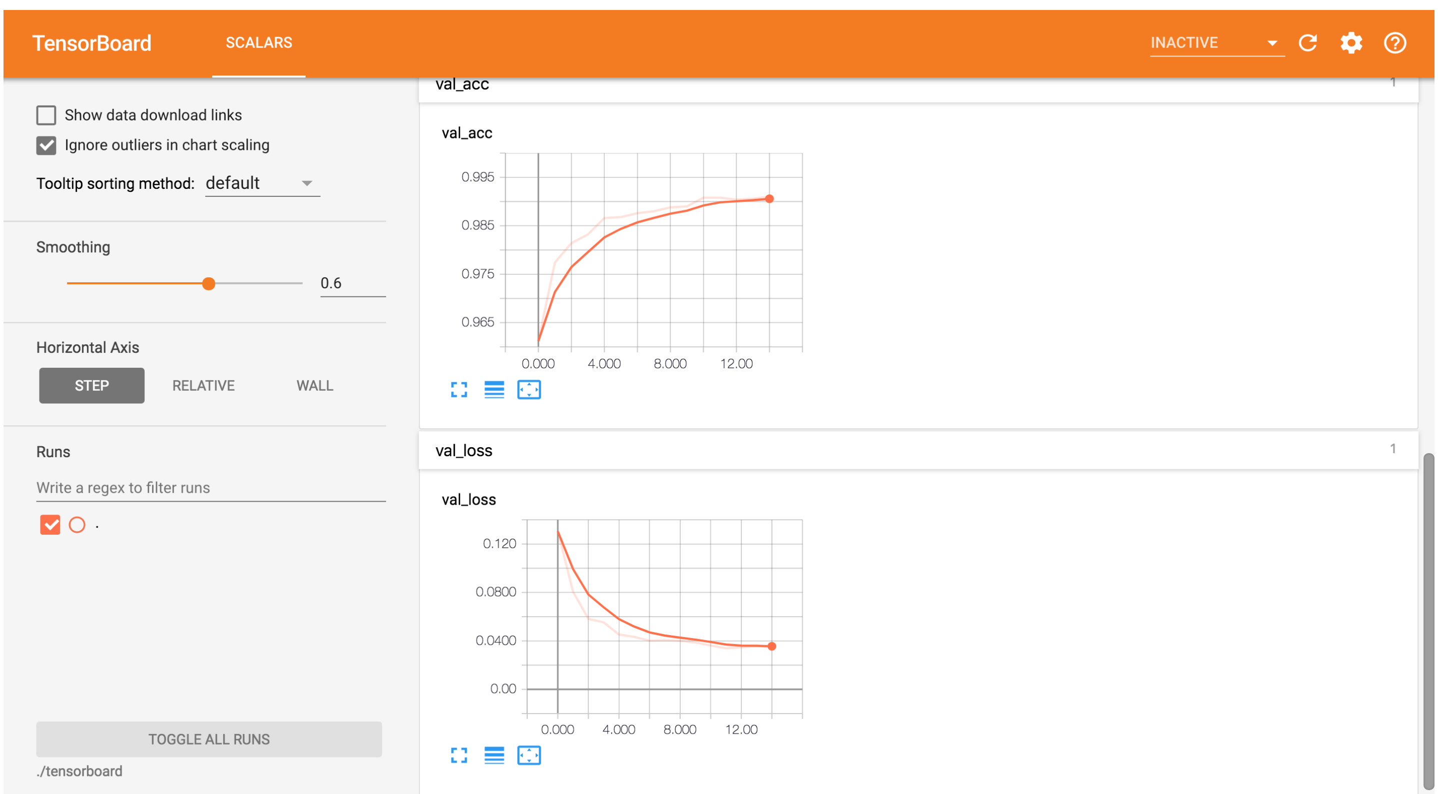


```
Correct label: 6
Assigned label: 6
```



```
Correct label: 0
Assigned label: 0
```

e) See the submitted screen shot - cnn.png



f) - g) Load the best model saved by the program

In [183]:
```
# load best model.
model.load_weights(filepath='weights/weights_cnn.hdf5')
# accuracy of best model on test data
test_accuracy_best = model.evaluate(mnist.test.images.reshape(10000, 28, 28, 1), mnist.test.labels, verbose=0)[1]
# accuracy of best model on validation data
validation_accuracy_best = model.evaluate(mnist.validation.images.reshape(5000, 28, 28, 1), mnist.validation.labels,
 verbose=0)[1]
print("Model with best validation accuracy\nAccuracy on the MNIST test set: {}, validation accuracy: {}".format(test
_accuracy_best, validation_accuracy_best))
```

```
Model with best validation accuracy
Accuracy on the MNIST test set: 0.9915, validation accuracy: 0.991
```