Contents of this part:

- loading and visualizing the MNIST dataset (training and test subsets)
- build and evaluate a simple CNN model
  - Experiments:
    - changing network structure/adding more layers: conv, pool, dropout, or fully connected
    - changing meta parameters
    - reduce the size of the training set to see how many samples are required for a good model
- Beyond accuracy: some additional metrics of predictive performance
- visualize network architecture (does not work yet; requires graphviz and Pydot libraries)
- Inspecting a learned network: visualizing learned network weights, visualizing network activations, ...
- What are the important features in a given input image: systematic occlusion experiment

Now, let's train an extremely simple CNN: First, we have to load the keras libraries and prepare the MNIST dataset (in this version it is already split into a training and a test set)

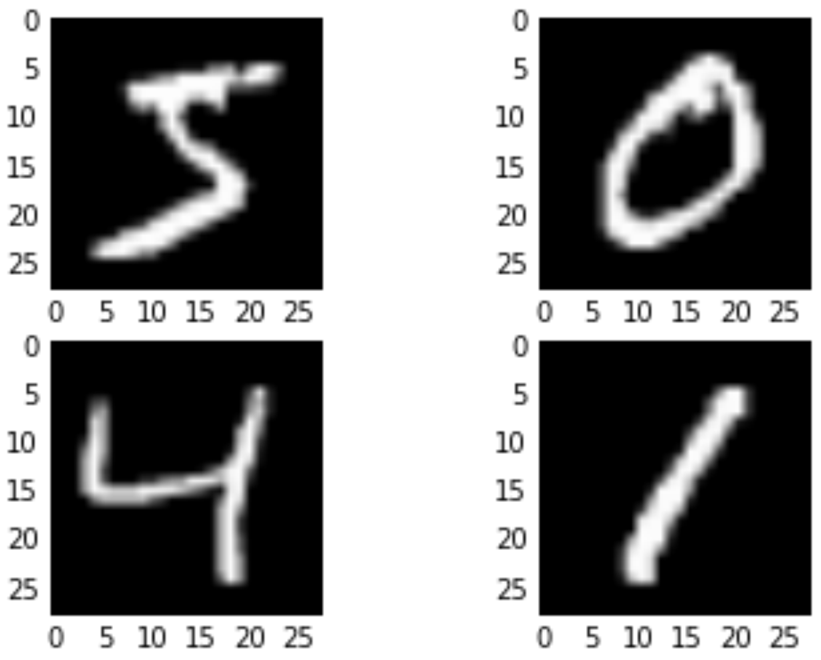
```
In [134]:
```

```
import numpy as np
from keras.models import Sequential
from keras.layers import Dense
from keras.datasets import mnist
from keras.layers import Dropout
from keras.layers import Flatten
from keras.layers.convolutional import Convolution2D
from keras.layers.convolutional import MaxPooling2D
from keras.utils import np_utils
from keras import backend as K
K.set_image_dim_ordering('th')
import matplotlib.pyplot as plt
%matplotlib inline
# todo: replace link to MNIST dataset:
(X_train, y_train), (X_test, y_test) = mnist.load_data("/home/singto1/deeplear
ning/mnist.pkl.gz")
```

Let's look at some of the MNIST samples:

In [135]:

```
plt.subplot(221)
plt.imshow(X_train[0], cmap=plt.get_cmap('gray'))
plt.subplot(222)
plt.imshow(X_train[1], cmap=plt.get_cmap('gray'))
plt.subplot(223)
plt.imshow(X_train[2], cmap=plt.get_cmap('gray'))
plt.subplot(224)
plt.imshow(X_train[3], cmap=plt.get_cmap('gray'))
# show the plot
plt.show()
```



In [136]:

```
# take one channel only:
X_train = X_train.reshape(X_train.shape[0], 1, 28, 28).astype('float32')
X_test = X_test.reshape(X_test.shape[0], 1, 28, 28).astype('float32')
# normalize inputs from 0-255 to 0-1
X_train = X_train / 255
X_test = X_test / 255
# one hot encode outputs
y_train = np_utils.to_categorical(y_train)
y_test = np_utils.to_categorical(y_test)
num_classes = y_test.shape[1]
```

Time to define and train our first (not-so)-deep neural network!

Suggestions for experimenting:

You can experiment a bit with the number of epochs, batch size, or even with the *number of training samples* (can we still train a good model with, say 10000 instead of 60000 training samples?) Try what happens with predictive performance if you add a dropout layer, additional Conv/Pool layers, and/or another fully connected layer

In [137]:

```
def model1():
    model = Sequential()
    model.add(Convolution2D(32, 5, 5, border_mode='valid', input_shape=(1, 28,
28), activation='relu'))
    model.add(MaxPooling2D(pool_size=(2, 2)))
    # model.add(Dropout(0.2))
    model.add(Flatten())
    model.add(Dense(128, activation='relu'))
    model.add(Dense(num_classes, activation='softmax'))
    # Compile model
    model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=[
'accuracy'])
    return model
```

In [6]:

```
# build the model
model = model1()
# Fit the model
model.fit(X_train, y_train, validation_data=(X_test, y_test), nb_epoch=10, bat
ch_size=200, verbose=2)
# Final evaluation of the model
scores = model.evaluate(X_test, y_test, verbose=0)
print("Baseline Error: %.2f%%" % (100-scores[1]*100))
```

```
WARNING (theano.gof.cmodule): WARNING: your Theano flags `gcc.cxxf
lags` specify an `-march=X` flags.
         It is better to let Theano/g++ find it automatically, but
we don't do it now
```

```
Train on 60000 samples, validate on 10000 samples
Epoch 1/10
3s - loss: 0.2474 - acc: 0.9295 - val_loss: 0.0810 - val_acc: 0.97
45
Epoch 2/10
3s - loss: 0.0680 - acc: 0.9803 - val_loss: 0.0514 - val_acc: 0.98
40
Epoch 3/10
3s - loss: 0.0468 - acc: 0.9860 - val_loss: 0.0380 - val_acc: 0.98
84
Epoch 4/10
3s - loss: 0.0351 - acc: 0.9890 - val_loss: 0.0395 - val_acc: 0.98
69
Epoch 5/10
3s - loss: 0.0275 - acc: 0.9918 - val_loss: 0.0401 - val_acc: 0.98
85
Epoch 6/10
3s - loss: 0.0216 - acc: 0.9933 - val_loss: 0.0347 - val_acc: 0.98
88
Epoch 7/10
3s - loss: 0.0166 - acc: 0.9947 - val_loss: 0.0353 - val_acc: 0.98
90
Epoch 8/10
3s - loss: 0.0134 - acc: 0.9958 - val_loss: 0.0374 - val_acc: 0.98
88
Epoch 9/10
3s - loss: 0.0115 - acc: 0.9965 - val_loss: 0.0368 - val_acc: 0.98
85
Epoch 10/10
3s - loss: 0.0085 - acc: 0.9976 - val_loss: 0.0388 - val_acc: 0.98
91
Baseline Error: 1.09%
```

It's always good to inspect some additional metrics of predictive performance. If you're not familiar with them you could check their definition:

In [19]:

```
Y_pred = model.predict(X_test)
# Convert one-hot to index
y_pred = np.argmax(Y_pred, axis=1)
yt = np.argmax(y_test, axis=1)
from sklearn.metrics import classification_report
print(classification_report(yt, y_pred))
```

```
             precision    recall  f1-score   support

          0       0.98      0.99      0.99       980
          1       1.00      0.99      1.00      1135
          2       0.98      0.99      0.99      1032
          3       0.98      0.99      0.98      1010
          4       0.99      0.99      0.99       982
          5       0.99      0.98      0.99       892
          6       0.99      0.99      0.99       958
          7       0.99      0.99      0.99      1028
          8       0.99      0.98      0.98       974
          9       0.98      0.99      0.99      1009

avg / total       0.99      0.99      0.99     10000
```

We can visualize the architecture of the neural network, but currently the graphviz back-end and the pydot wrapper is not available in our environment. Anyways, this is how it would work:

In [22]:

```
from keras.utils.visualize_util import plot
plot(model) # , to_file='model.png')
```

```
---------------------------------------------------------------------------
ImportError                               Traceback (most recent call last)
<ipython-input-22-be8c52e570b4> in <module>()
----> 1 from keras.utils.visualize_util import plot
      2 plot(model) # , to_file='model.png')

/usr/prog/pythonML/3.2-goolf-1.5.14-NX-python-2.7.11/lib/python2.7/site-packages/Keras-1.1.2-py2.7.egg/keras/utils/visualize_util.py in <module>()
      9 except ImportError:
     10     # fall back on pydot if necessary
---> 11     import pydot
     12 if not pydot.find_graphviz():
     13     raise RuntimeError('Failed to import pydot. You must install pydot'

ImportError: No module named pydot
```

Visualize weights and convolutions:

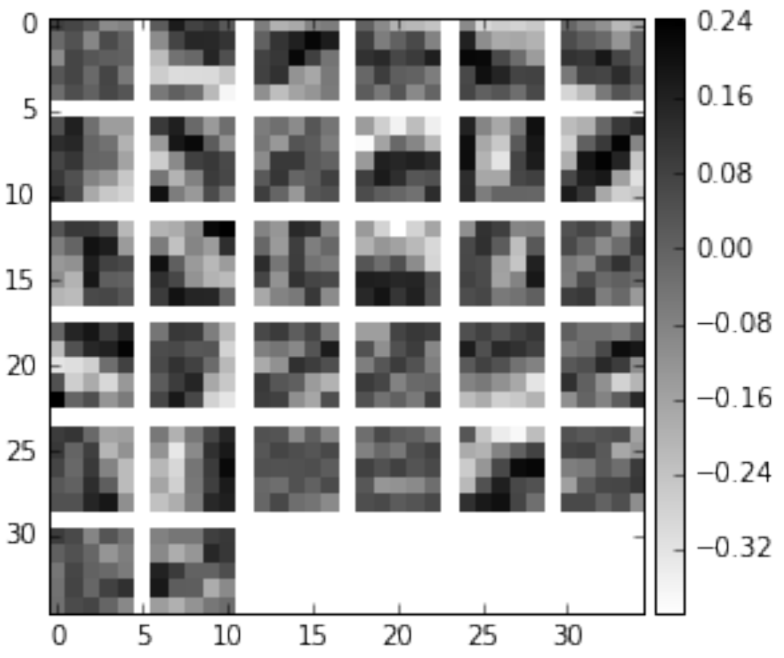
In [20]:

```
import numpy.ma as ma
from mpl_toolkits.axes_grid1 import make_axes_locatable

def nice_imshow(ax, data, vmin=None, vmax=None, cmap=None):
    """Wrapper around pl.imshow"""
    if cmap is None:
        cmap = cm.jet
    if vmin is None:
        vmin = data.min()
    if vmax is None:
        vmax = data.max()
    divider = make_axes_locatable(ax)
    cax = divider.append_axes("right", size="5%", pad=0.05)
    im = ax.imshow(data, vmin=vmin, vmax=vmax, interpolation='nearest', cmap=c
map)
    plt.colorbar(im, cax=cax)
def make_mosaic(imgs, nrows, ncols, border=1):
    """
    Given a set of images with all the same shape, makes a
    mosaic with nrows and ncols
    """
    nimgs = imgs.shape[0]
    imshape = imgs.shape[1:]

    mosaic = ma.masked_all((nrows * imshape[0] + (nrows - 1) * border,
                            ncols * imshape[1] + (ncols - 1) * border),
                            dtype=np.float32)

    paddedh = imshape[0] + border
    paddedw = imshape[1] + border
    for i in xrange(nimgs):
        row = int(np.floor(i / ncols))
        col = i % ncols

        mosaic[row * paddedh:row * paddedh + imshape[0],
               col * paddedw:col * paddedw + imshape[1]] = imgs[i]
    return mosaic
```

In [21]:

```
# Visualize weights
W = model.layers[0].W.get_value(borrow=True)
W = np.squeeze(W)
print("W shape : ", W.shape)

nice_imshow(plt.gca(), make_mosaic(W, 6, 6), cmap=plt.cm.binary)
```

```
('W shape : ', (32, 5, 5))
```



It's very useful to be able to save or load module weights, for later model inspection, for continuing to train, or for transfer learning (transferring a learned model to a different dataset). Question: What is the difference between model.save and model.save_weights?

In [24]:

```
from keras.models import load_model

model.save('my_model.h5')  # creates a HDF5 file 'my_model.h5'
del model  # deletes the existing model

# returns a compiled model
# identical to the previous one
model = load_model('my_model.h5')
model.save_weights('my_model_weights.h5')
model.load_weights('my_model_weights.h5')
```

It may also be instructive to look at the activations in a specific network layer, given a specific input sample. If you are interested in how this is done you can google "visualizing network activations keras"....

What is the classifier "looking at" in a given image? Systematic occlusion experiment:

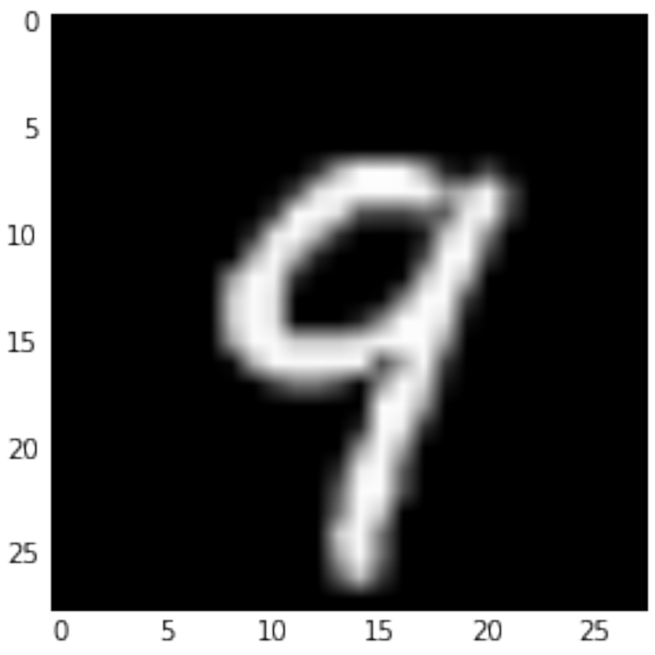
So what are the key regions in the following digit for the classifier?

In [104]:

```
plt.imshow(X_test[12,0,:,:], cmap=plt.get_cmap('gray'))
```

Out[104]:

```
<matplotlib.image.AxesImage at 0x2aab2db8e750>
```



In [105]:

```
model.predict(X_test[12:13,:,:,:])
```

Out[105]:

```
array([[  1.48771967e-12,   1.37436639e-13,   1.83610613e-10,
          2.35146214e-08,   1.73420290e-06,   1.93743110e-09,
          6.20960870e-15,   2.31211729e-07,   2.06545359e-08,
          9.99997973e-01]], dtype=float32)
```

Let's take a part away from that digit....

In [106]:

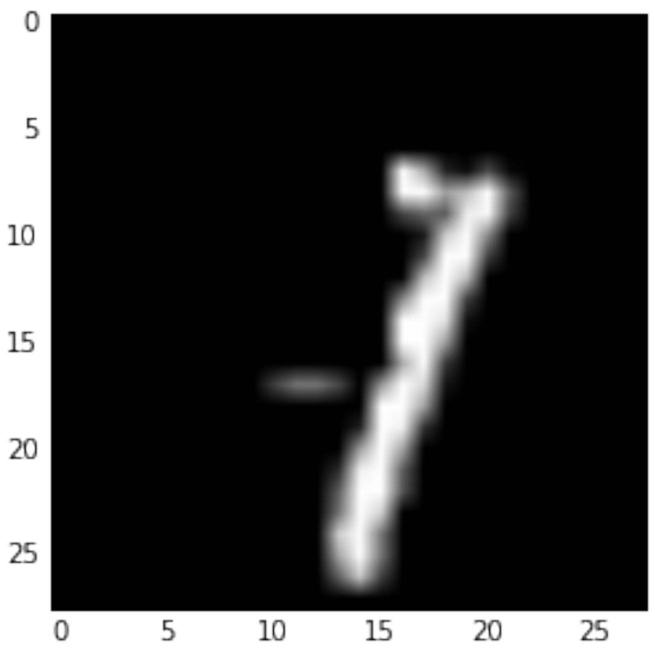
```
import copy
xx = copy.copy(X_test[12:13,:,:,:])
xx[0,0,5:17,5:16] = 0
```

In [107]:

```
plt.imshow(xx[0,0,:,:], cmap=plt.get_cmap('gray'))
```

Out[107]:

```
<matplotlib.image.AxesImage at 0x2aab2dc3cd10>
```



Now the classifier thinks it's most likely a 7, as we can see when predicting again (question: what do you think would happen if we had "European" 1s in the dataset, too?!):

In [108]:

```
model.predict(xx[0:1,:,:,:])
```

Out[108]:

```
array([[  1.60252142e-07,   4.82410287e-05,   2.81161811e-05,
          5.03473848e-06,   1.17908203e-04,   2.40964084e-08,
          2.09353507e-10,   9.93810534e-01,   2.07447702e-05,
          5.96931158e-03]], dtype=float32)
```

Now let's try to do the occlusion systematically, sliding a black square across the image:

In [114]:

```
def occlude(X, size=10):
    Y = np.zeros(((X.shape[1]-size)*(X.shape[0]-size), 1, 28, 28))
    for i in xrange(0, X.shape[0]-size):
        for j in xrange(0, X.shape[1]-size):
            XX = X.copy()
            XX[i:(i+size), j:(j+size)] = 0
            Y[j+i*(X.shape[1]-size),0,:,:] = XX
    return Y
```

In [115]:

```
xx = copy.copy(X_test[12:13,:,:,:])
```

In [116]:

```
Y = occlude(xx[0,0,:,:])
```

We now have 324 partially occluded images generated from our original '9' digit. Have a look at some of them if you like:

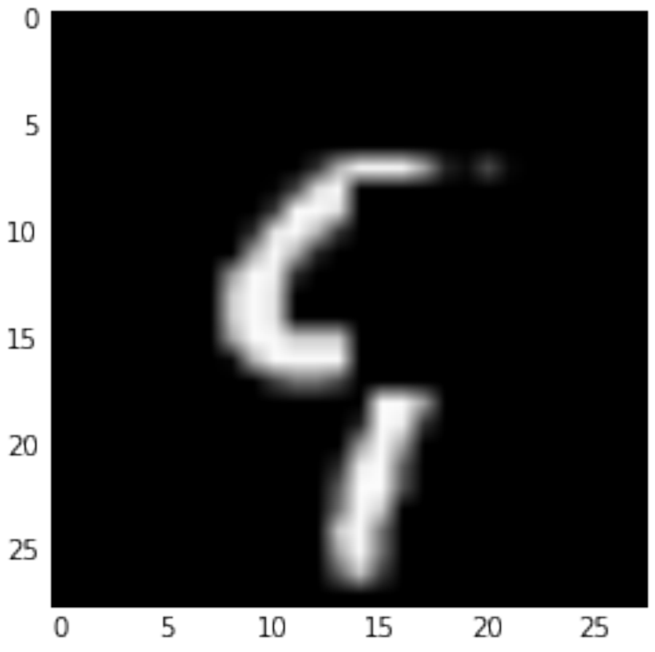
In [132]:

```
print(Y.shape)
plt.imshow(Y[158,0,:,:], cmap=plt.get_cmap('gray'))
```

```
(324, 1, 28, 28)
```

Out[132]:

```
<matplotlib.image.AxesImage at 0x2aab2e077d10>
```



Now, let's make the predictions for all of them using our classifier:

In [124]:

```
pp = model.predict(Y)
```

In [125]:

```
pred = model.predict(X_test[12:13,:,:,:])
pred
```

Out[125]:

```
array([[  1.48771967e-12,   1.37436639e-13,   1.83610613e-10,
          2.35146214e-08,   1.73420290e-06,   1.93743110e-09,
          6.20960870e-15,   2.31211729e-07,   2.06545359e-08,
          9.99997973e-01]], dtype=float32)
```

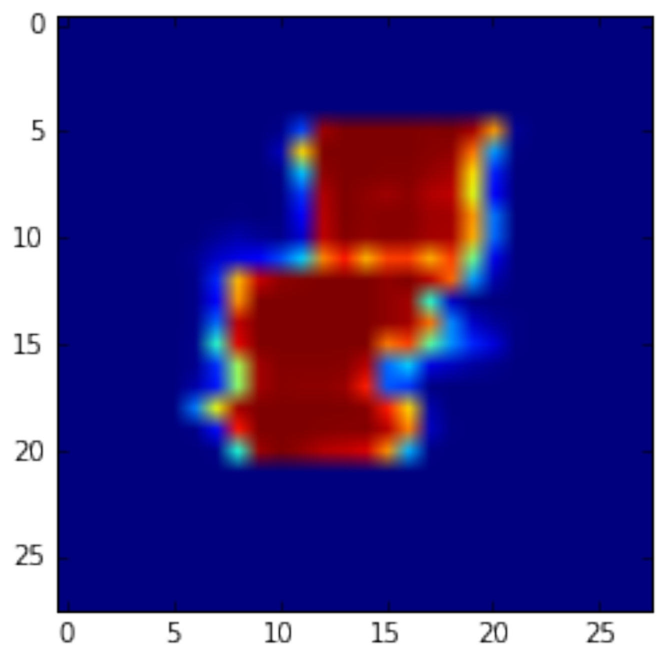
In [130]:

```
myDiff = ([pred[0,9]]*len(pp[:,9])) -pp[:,9]
diffMatrix = np.reshape(myDiff, (18,18), order='C')
```

```
In [131]:
```

```
DD = np.zeros((28,28))
DD[5:23,5:23] = diffMatrix
plt.imshow(DD)
```

```
Out[131]:

<matplotlib.image.AxesImage at 0x2aab2e009850>
```



....so in this case, the importance map is not too surprising or enlightening. If you want you can experiment with the size of the sliding black square. We used 10x10 which is quite big for the 28x28 thumbnails.

You could also experiment with some other digits from the MNIST test dataset....

```
In [ ]:
```