In [15]:
```
%pylab inline
import gzip, cPickle
```

```
Welcome to pylab, a matplotlib-based Python environment [backend:
module://IPython.zmq.pylab.backend_inline].
For more information, type 'help(pylab)'.
```

Lab 2: Classification

Machine Learning and Pattern Recognition, September 2013

- The lab exercises should be made in groups of three people, or at least two people.
- The deadline is october 6th (sunday) 23:59.
- Assignment should be sent to N.Hu@uva.nl (Ninhang Hu). The subject line of your email should be "lab#_lastname1_lastname2_lastname3".
- Put your and your teammates' names in the body of the email
- Attach the .IPYNB (IPython Notebook) file containing your code and answers. Naming of the file follows the same rule as the subject line. For example, if the subject line is "lab01_Kingma_Hu", the attached file should be "lab01_Kingma_Hu.ipynb". Only use underscores ("_") to connect names, otherwise the files cannot be parsed.

Notes on implementation:

- You should write your code and answers in an IPython Notebook: http://ipython.org/notebook.html. If you have problems, please contact us.
- Among the first lines of your notebook should be "%pylab inline". This imports all required modules, and your plots will appear inline.
- For this lab, your regression solutions should be in closed form, i.e., should not perform iterative gradient-based optimization but find the exact optimum directly.
- NOTE: Make sure we can run your notebook / scripts!

Part 1. Multiclass logistic regression

Scenario: you have a friend with one big problem: she's completely blind. You decided to help her: she has a special smartphone for blind people, and you are going to develop a mobile phone app that can do *machine vision* using the mobile camera: converting a picture (from the camera) to the meaning of the image. You decide to start with an app that can read handwritten digits, i.e. convert an image of handwritten digits to text (e.g. it would enable her to read precious handwritten phone numbers).

A key building block for such an app would be a function `predict_digit(x)` that returns the digit class of an image patch $\mathbf{x}$. Since hand-coding this function is highly non-trivial, you decide to solve this problem using machine learning, such that the internal parameters of this function are automatically learned using machine learning techniques.

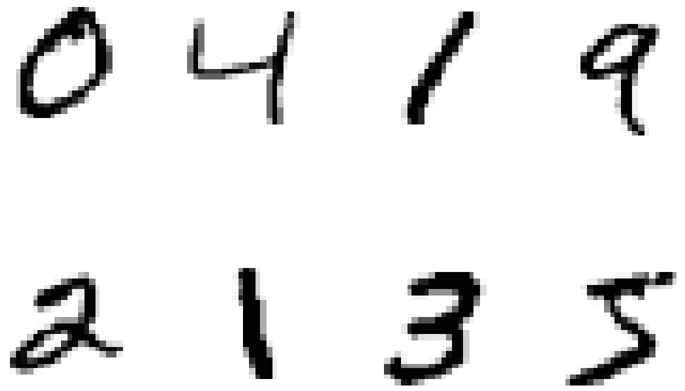
The dataset you're going to use for this is the MNIST handwritten digits dataset (`http://yann.lecun.com/exdb/mnist/`). You can load the data from `mnist.pkl.gz` we provided, using:

In [13]:
```
def load_mnist():
    f = gzip.open('mnist.pkl.gz', 'rb')
    data = cPickle.load(f)
    f.close()
    return data

(x_train, t_train), (x_valid, t_valid), (x_test, t_test) = load_mnist()
```

The tuples represent train, validation and test sets. The first element (`x_train`, `x_valid`, `x_test`) of each tuple is a $N \times M$ matrix, where N is the number of datapoints and $M = 28^2 = 784$ is the dimensionality of the data. The second element (`t_train`, `t_valid`, `t_test`) of each tuple is the corresponding N-dimensional vector of integers, containing the true class labels.

Here's a visualisation of the first 8 digits of the trainingset:

In [14]:
```
def plot_digits(data, numcols, shape=(28,28)):
    numdigits = data.shape[0]
    numrows = int(numdigits/numcols)
    for i in range(numdigits):
        plt.subplot(numrows, numcols, i)
        plt.axis('off')
        plt.imshow(data[i].reshape(shape), interpolation='nearest', cmap='Greys')
    plt.show()

plot_digits(x_train[0:8], numcols=4)
```



In *multiclass* logistic regression, the conditional probability of class label j given the image $\mathbf{x}$ for some datapoint is given by:

$$\log p(t = j \mid \mathbf{x}, \mathbf{b}, \mathbf{W}) = \log q_j - \log Z$$

where $\log q_j = \mathbf{w}_j^T \mathbf{x} + b_j$ (the unnormalized probability of the class j), and $Z = \sum_k q_k$ is the normalizing factor. $\mathbf{w}_j$ is the j-th column of $\mathbf{W}$ (a matrix of size 784×10) corresponding to the class label, b_j is the j-th element of $\mathbf{b}$.

Given an input image, the multiclass logistic regression model first computes the intermediate vector $\log \mathbf{q}$ (of size 10×1), using $\log q_j = \mathbf{w}_j^T \mathbf{x} + b_j$, containing the unnormalized log-probabilities per class.

The unnormalized probabilities are then normalized by Z such that $\sum_j p_j = \sum_j \exp(\log p_j) = 1$. This is done by $\log p_j = \log q_j - \log Z$ where $Z = \sum_j \exp(\log q_j)$. This is known as the *softmax* transformation, and is also used as a last layer of many classifcation neural network models, to ensure that the output of the network is a normalized distribution, regardless of the values of second-to-last layer ($\log \mathbf{q}$)

The network's output $\log \mathbf{p}$ of size 10×1 then contains the conditional log-probabilities $\log p(t = j \mid \mathbf{x}, \mathbf{b}, \mathbf{W})$ for each digit class j. In summary, the computations are done in this order:

$$\mathbf{x} \rightarrow \log \mathbf{q} \rightarrow Z \rightarrow \log \mathbf{p}$$

Given some dataset with N datapoints, the log-likelihood is given by:

$$\mathcal{L}(\mathbf{b}, \mathbf{W}) = \sum_{i=1}^{N} \mathcal{L}^{(i)}$$

where we use $\mathcal{L}^{(i)}$ to denote the partial log-likelihood evaluated over a single datapoint. It is important to see that the log-probability of the class label $t^{(i)}$ given the image, is given by the $t^{(i)}$-th element of the network's output $\log \mathbf{p}$, denoted by $\log p_{t^{(i)}}$:

$$\mathcal{L}^{(i)} = \log p(t = t^{(i)} \mid \mathbf{x} = \mathbf{x}^{(i)}, \mathbf{b}, \mathbf{W}) = \log p_{t^{(i)}} = \log q_{t^{(i)}} - \log Z$$

where $\mathbf{x}^{(i)}$ and $t^{(i)}$ are the input (image) and class label (integer) of the i-th datapoint, and Z is of course different per datapoint.

1.1 Gradient-based stochastic optimization

1.1.1 Derive gradient equations

Derive the equations for computing the (first) partial derivatives of log-likelihood w.r.t. all the parameters, evaluated at a *single* datapoint i.

You should start deriving the equations for $\frac{\partial \mathcal{L}^{(i)}}{\partial \log q_j}$ for each j. For clarity, we'll use the shorthand $\delta^q_j = \frac{\partial \mathcal{L}^{(i)}}{\partial \log q_j}$.

For $j = t^{(i)}$: $\delta^q_j = \frac{\partial \mathcal{L}^{(i)}}{\partial \log p_j} \frac{\partial \log p_j}{\partial \log q_j} + \frac{\partial \mathcal{L}^{(i)}}{\partial \log Z} \frac{\partial \log Z}{\partial Z} \frac{\partial Z}{\partial \log q_j} = 1 \cdot 1 - \frac{\partial \log Z}{\partial Z} \frac{\partial Z}{\partial \log q_j} = 1 - \frac{\partial \log Z}{\partial Z} \frac{\partial Z}{\partial \log q_j}$

For $j \neq t^{(i)}$: $\delta^q_j = \frac{\partial \mathcal{L}^{(i)}}{\partial \log Z} \frac{\partial \log Z}{\partial Z} \frac{\partial Z}{\partial \log q_j} = - \frac{\partial \log Z}{\partial Z} \frac{\partial Z}{\partial \log q_j}$

Complete the above derivations for δ^q_j by furtherly developing $\frac{\partial \log Z}{\partial Z}$ and $\frac{\partial Z}{\partial \log q_j}$. Both are quite simple. For these it doesn't matter whether $j = t^{(i)}$ or not.

Given your equations for computing the gradients δ^q_j it should be quite straightforward to derive the equations for the gradients of the parameters of the model, $\frac{\partial \mathcal{L}^{(i)}}{\partial \log W_{ij}}$ and $\frac{\partial \mathcal{L}^{(i)}}{\partial b_j}$. The gradients for the biases $\mathbf{b}$ are given by:

$$\frac{\partial \mathcal{L}^{(i)}}{\partial b_j} = \frac{\partial \mathcal{L}^{(i)}}{\partial \log q_j} \frac{\partial \log q_j}{\partial b_j} = \delta^q_j \cdot 1 = \delta^q_j$$

The equation above gives the derivative of $\mathcal{L}^{(i)}$ w.r.t. a single element of $\mathbf{b}$, so the vector $\nabla_{\mathbf{b}} \mathcal{L}^{(i)}$ with all derivatives of $\mathcal{L}^{(i)}$ w.r.t. the bias parameters $\mathbf{b}$ is:

$$\nabla_{\mathbf{b}}\mathcal{L}^{(i)} = \delta^q$$

where δ^q denotes the vector of size 10×1 with elements δ^q_j.

The (not fully developed) equation for computing the derivative of $\mathcal{L}^{(i)}$ w.r.t. a single element W_{ij} of $\mathbf{W}$ is:

$$\frac{\partial \mathcal{L}^{(i)}}{\partial W_{ij}} = \frac{\partial \mathcal{L}^{(i)}}{\partial \log q_j} \frac{\partial \log q_j}{\partial W_{ij}} = \delta^q_j \frac{\partial \log q_j}{\partial W_{ij}}$$

What is $\frac{\partial \log q_j}{\partial W_{ij}}$? Complete the equation above.

If you want, you can give the resulting equation in vector format ($\nabla_{\mathbf{w}_j}\mathcal{L}^{(i)} = \ldots$), like we did for $\nabla_{\mathbf{b}}\mathcal{L}^{(i)}$.

1.1.2 Implement gradient computations

Implement the gradient calculations you derived in the previous question. Write a function `logreg_gradient(x, t, w, b)` that returns the gradients $\nabla_{\mathbf{w}_j}\mathcal{L}^{(i)}$ (for each j) and $\nabla_{\mathbf{b}}\mathcal{L}^{(i)}$, i.e. the first partial derivatives of the log-likelihood w.r.t. the parameters $\mathbf{W}$ and $\mathbf{b}$, evaluated at a single datapoint (`x`, `t`). The computation will contain roughly the following intermediate variables:

$$\log \mathbf{q} \rightarrow Z \rightarrow \log \mathbf{p} \rightarrow \delta^q$$

followed by computation of the gradient vectors $\nabla_{\mathbf{w}_j}\mathcal{L}^{(i)}$ (contained in a 784×10 matrix) and $\nabla_{\mathbf{b}}\mathcal{L}^{(i)}$ (a 10×1 vector).

1.1.3 Stochastic gradient descent

Write a function `sgd_iter(x_train, t_train, w, b)` that performs an iteration of stochastic gradient descent (SGD), and returns the new weights. It should go through the trainingset once in randomized order, call `logreg_gradient(x, t, w, b)` for each datapoint to get the gradients, and update the parameters using a small learning rate (e.g. `1E-4`). Note that in this case we're maximizing the likelihood function, so we should actually performing gradient *ascent*..

1.2. Train

1.2.1 Train

Perform a handful of training iterations through the trainingset. Plot (in one graph) the conditional log-probability of the trainingset and validation set after each iteration.

1.2.2 Visualize weights

Visualize the resulting parameters $\mathbf{W}$ after a few iterations through the training set, by treating each column of $\mathbf{W}$ as an image. If you want, you can use or edit the `plot_digits(...)` above.

1.2.3. Visualize the 8 hardest and 8 easiest digits

Visualize the 8 digits in the validation set with the highest probability of the true class label under the model. Also plot the 8 digits that were assigned the lowest probability. Ask yourself if these results make sense.

Part 2. Multilayer perceptron

(In this part you don't have to write any code, except for bonus points).

You discover that the predictions by the logistic regression classifier are not good enough for your application: the model is too simple. You want to increase the accuracy of your predictions by using a better model. For this purpose, you're going to use a multilayer perceptron (MLP), a simple kind of neural network. The perceptron wil have a single hidden $\mathbf{h}$ layer with L elements. The parameters of the model are $\mathbf{V}$ (connections between input $\mathbf{x}$ and hidden layer $\mathbf{h}$), $\mathbf{a}$ (the biases/intercepts $\mathbf{h}$), $\mathbf{W}$ (connections between $\mathbf{h}$ and $\log q$) and $\mathbf{b}$ (the biases/intercepts of $\log q$.

The conditional probability of the class label j is given by:

$$\log p(t = j \mid \mathbf{x}, \mathbf{b}, \mathbf{W}) = \log q_j - \log Z$$

where q_j are again the unnormalized probabilities per class, and $Z = \sum_j q_j$ is again the probability normalizing factor. Each q_j is computed using:

$$\log q_j = \mathbf{w}_j^T \mathbf{h} + b_j$$

where $\mathbf{h}$ is a $L \times 1$ vector with the hidden layer activations (of a hidden layer with size L), and $\mathbf{w}_j$ is the j-th column of $\mathbf{W}$ (a $L \times 10$ matrix). Each element of the hidden layer is computed from the input vector $\mathbf{x}$ using:

$$h_j = \sigma(\mathbf{v}_j^T \mathbf{x} + a_j)$$

where $\mathbf{v}_j$ is the j-th column of $\mathbf{V}$ (a $784 \times L$ matrix), a_j is the j-th element of $\mathbf{a}$, and $\sigma(.)$ is the so-called sigmoid activation function, defined by:

$$\sigma(x) = \frac{1}{1+\exp(-x)}$$

Note that this model is almost equal to the multiclass logistic regression model, but with an extra 'hidden layer' $\mathbf{h}$. The activations of this hidden layer can be viewed as features computed from the input, where the feature transformation ($\mathbf{V}$ and $\mathbf{a}$) is learned.

2.1 Derive gradient equations

State (shortly) why $\nabla_{\mathbf{b}} \mathcal{L}^{(i)}$ is equal to the earlier (multiclass logistic regression) case, and why $\nabla_{\mathbf{w}_j} \mathcal{L}^{(i)}$ is almost equal to the earlier case.

Like in multiclass logistic regression, you should use intermediate variables δ_j^q. In addition, you should use intermediate variables $\delta_j^h = \frac{\partial \mathcal{L}^{(i)}}{\partial \log h_j}$.

Given an input image, roughly the following intermediate variables should be computed:

$$\log \mathbf{q} \rightarrow Z \rightarrow \log \mathbf{p} \rightarrow \delta^q \rightarrow \delta^h$$

where $\delta_j^h = \frac{\partial \mathcal{L}^{(i)}}{\partial \mathbf{h}_j}$.

Give the equations for computing δ^h, and for computing the derivatives of $\mathcal{L}^{(i)}$ w.r.t. $\mathbf{W}$, $\mathbf{b}$, $\mathbf{V}$ and $\mathbf{a}$.

You can use the convenient fact that $\frac{\partial}{\partial x} \sigma(x) = \sigma(x)(1 - \sigma(x))$.

2.2 MAP optimization

You derived equations for finding the *maximum likelihood* solution of the parameters. Explain, in a few sentences, how you could extend this approach so that it optimizes towards a *maximum a posteriori* (MAP) solution of the parameters, with a Gaussian prior on the parameters.

(Like in 2.1, you *don't* need to write any code)

2.3. Extra

2.3.1. 10% bonus points: Implement and train a MLP

You receive 10% bonus points if you succesfully implement a MLP model with a single hidden layer. Your code should include code to learn the parameters.

2.3.2. 10% bonus points: Less than 150 misclassifications on the test set

You receive an additional 10% bonus points if you manage to train a model with very high accuracy: at most 1.5% misclasified digits on the test set. Note that the test set contains 10000 digits, so you model should misclassify at most 150 digits. This should be achievable with a MLP model with one hidden layer. See results of various models at : `http://yann.lecun.com/exdb/mnist/index.html`. To reach such a low accuracy, you probably need to have a very high L (many hidden units), probably $L > 200$, and apply a strong Gaussian prior on the weights. In this case you are allowed to use the validation set for training. You are allowed to add additional layers, and use convolutional networks, although that is probably not required to reach 1.5% misclassifications.