UFLDL Tutorial

ver 201406*

1 Introduction

This tutorial will teach you the main ideas of Unsupervised Feature Learning and Deep Learning. By working through it, you will also get to implement several feature learning/deep learning algorithms, get to see them work for yourself, and learn how to apply/adapt these ideas to new problems.

This tutorial assumes a basic knowledge of machine learning (specifically, familiarity with the ideas of supervised learning, logistic regression, gradient descent). If you are not familiar with these ideas, we suggest you go to this Machine Learning course and complete sections II, III, IV (up to Logistic Regression) first.

Material contributed by: Andrew Ng, Jiquan Ngiam, Chuan Yu Foo, Yifan Mai, Caroline Suen

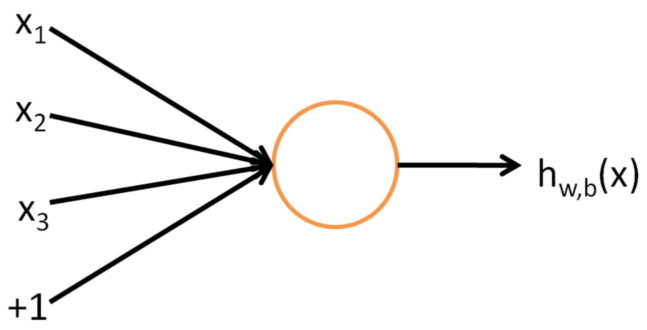
*This document was formated base on the online UFLDL Tutorial 2.15. You may update the latest version from GitHub. Please don't hesitate to leave message in the web site if you have any comments or suggestion. Yes, the format of this document is not perfect, and you may adjust it by yourself if you want. Don't forget to share your version, and notify me to update this document. Thanks, have fun!

2 Sparse Autoencoder

2.1 Neural Networks

Consider a supervised learning problem where we have access to labeled training examples $(x^{(i)}, y^{(i)})$. Neural networks give a way of defining a complex, non-linear form of hypotheses $h_{W,b}(x)$, with parameters W, b that we can fit to our data.

To describe neural networks, we will begin by describing the simplest possible neural network, one which comprises a single "neuron." We will use the following diagram to denote a single neuron:



This "neuron" is a computational unit that takes as input x_1, x_2, x_3 (and a $+1$ intercept term), and outputs $h_{W,b}(x) = f(W^T x) = f(\sum_{i=1}^{3} W_i x_i + b)$, where $f : \Re \mapsto \Re$ is called the activation function. In these notes, we will choose $f(\cdot)$ to be the function:

$$f(z) = \frac{1}{1 + \exp(-z)}.$$

Thus, our single neuron corresponds exactly to the input-output mapping defined by logistic regression. Although these notes will use the sigmoid function, it is worth noting that another common choice for f is the hyperbolic tangent, or tanh, function:

$$f(z) = \tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}},$$

Here are plots of the sigmoid and tanh functions (Figure 1):

The $\tanh(z)$ function is a rescaled version of the sigmoid, and its output range is $[-1, 1]$ instead of $[0, 1]$.

Note that unlike some other venues (including the OpenClassroom videos, and parts of CS229), we are not using the convention here of $x_0 = 1$. Instead, the intercept term is handled separately by the parameter b.

Finally, one identity that'll be useful later: If $f(z) = 1/(1 + \exp(-z))$ is the sigmoid function, then its derivative is given by $f'(z) = f(z)(1 - f(z))$. (If f is the tanh function, then its derivative is given by $f'(z) = 1 - (f(z))^2$.) You can derive this yourself using the definition of the sigmoid (or tanh) function.

2.1.1 Neural Network model

A neural network is put together by hooking together many of our simple "neurons," so that the output of a neuron can be the input of another. For example, here is a small neural network:

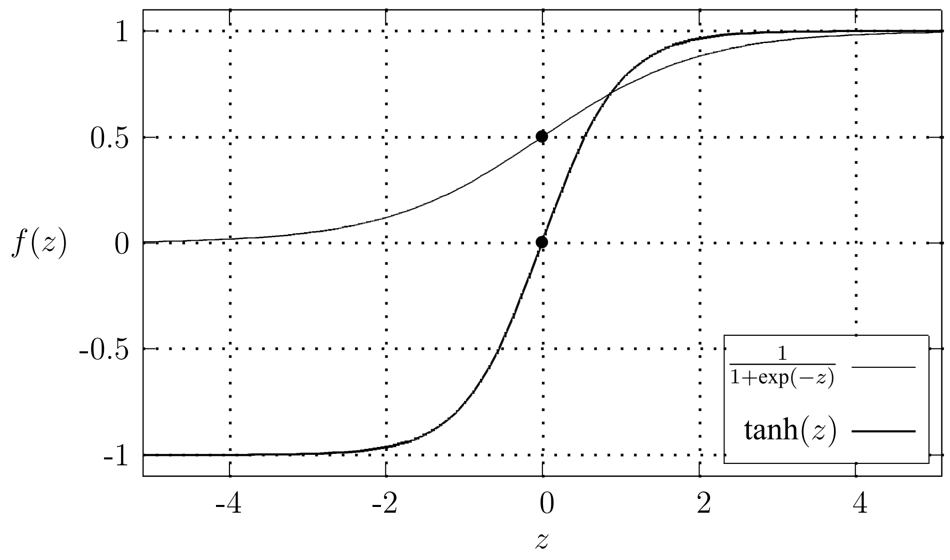


Figure 1: Activation functions

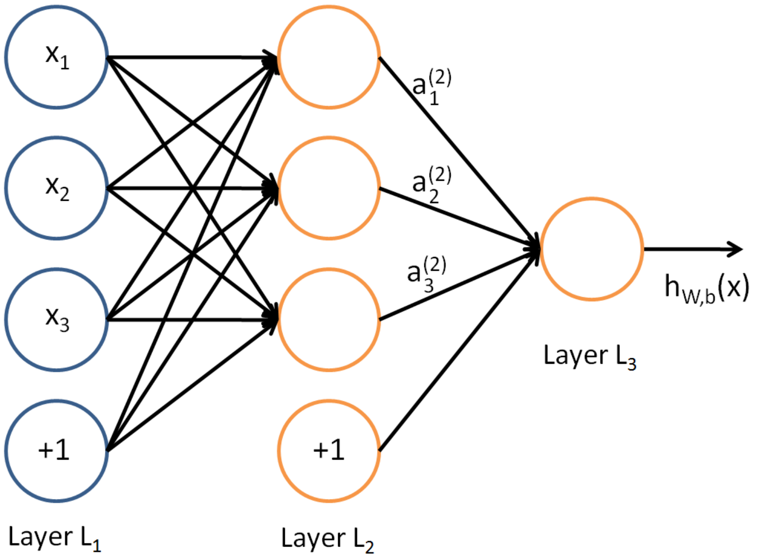
In this figure, we have used circles to also denote the inputs to the network. The circles labeled "+1" are called bias units, and correspond to the intercept term. The leftmost layer of the network is called the input layer, and the rightmost layer the output layer (which, in this example, has only one node). The middle layer of nodes is called the hidden layer, because its values are not observed in the training set. We also say that our example neural network has 3 input units (not counting the bias unit), 3 hidden units, and 1 output unit.

We will let n_l denote the number of layers in our network; thus $n_l = 3$ in our example. We label layer l as L_l, so layer L_1 is the input layer, and layer L_{n_l} the output layer. Our neural network has parameters $(W, b) = (W^{(1)}, b^{(1)}, W^{(2)}, b^{(2)})$, where we write $W_{ij}^{(l)}$ to denote the parameter (or weight) associated with the connection between unit j in layer l, and unit i in layer $l+1$. (Note the order of the indices.) Also, $b_i^{(l)}$ is the bias associated with unit i in layer $l+1$. Thus, in our example, we have $W^{(1)} \in \Re^{3\times 3}$, and $W^{(2)} \in \Re^{1\times 3}$. Note that bias units don't have inputs or connections going into them, since they always output the value +1. We also let s_l denote the number of nodes in layer l (not counting the bias unit).

We will write $a_i^{(l)}$ to denote the activation (meaning output value) of unit i in layer l. For $l = 1$, we also use $a_i^{(1)} = x_i$ to denote the i-th input. Given a fixed setting of the parameters W, b, our neural network defines a hypothesis $h_{W,b}(x)$ that outputs a real number. Specifically, the computation that this neural network represents is given by:

$$\begin{aligned}
a_1^{(2)} &= f(W_{11}^{(1)}x_1 + W_{12}^{(1)}x_2 + W_{13}^{(1)}x_3 + b_1^{(1)}) \\
a_2^{(2)} &= f(W_{21}^{(1)}x_1 + W_{22}^{(1)}x_2 + W_{23}^{(1)}x_3 + b_2^{(1)}) \\
a_3^{(2)} &= f(W_{31}^{(1)}x_1 + W_{32}^{(1)}x_2 + W_{33}^{(1)}x_3 + b_3^{(1)}) \\
h_{W,b}(x) &= a_1^{(3)} = f(W_{11}^{(2)}a_1^{(2)} + W_{12}^{(2)}a_2^{(2)} + W_{13}^{(2)}a_3^{(2)} + b_1^{(2)})
\end{aligned}$$

In the sequel, we also let $z_i^{(l)}$ denote the total weighted sum of inputs to unit i in layer l, including the bias term (e.g., $z_i^{(2)} = \sum_{j=1}^{n} W_{ij}^{(1)}x_j + b_i^{(1)}$), so that $a_i^{(l)} = f(z_i^{(l)})$.



Note that this easily lends itself to a more compact notation. Specifically, if we extend the activation function $f(\cdot)$ to apply to vectors in an element-wise fashion (i.e., $f([z_1, z_2, z_3]) = [f(z_1), f(z_2), f(z_3)]$, then we can write the equations above more compactly as:

$$\begin{aligned} z^{(2)} &= W^{(1)}x + b^{(1)} \\ a^{(2)} &= f(z^{(2)}) \\ z^{(3)} &= W^{(2)}a^{(2)} + b^{(2)} \\ h_{W,b}(x) &= a^{(3)} = f(z^{(3)}) \end{aligned}$$

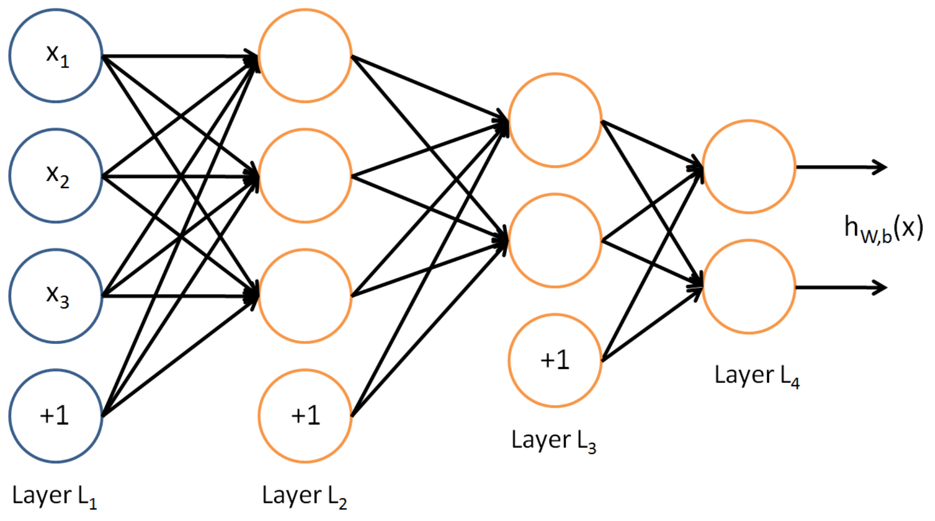
We call this step forward propagation. More generally, recalling that we also use $a^{(1)} = x$ to also denote the values from the input layer, then given layer l's activations $a^{(l)}$, we can compute layer $l+1$'s activations $a^{(l+1)}$ as:

$$\begin{aligned} z^{(l+1)} &= W^{(l)}a^{(l)} + b^{(l)} \\ a^{(l+1)} &= f(z^{(l+1)}) \end{aligned}$$

By organizing our parameters in matrices and using matrix-vector operations, we can take advantage of fast linear algebra routines to quickly perform calculations in our network.

We have so far focused on one example neural network, but one can also build neural networks with other architectures (meaning patterns of connectivity between neurons), including ones with multiple hidden layers. The most common choice is a n_l-layered network where layer 1 is the input layer, layer n_l is the output layer, and each layer l is densely connected to layer $l+1$. In this setting, to compute the output of the network, we can successively compute all the activations in layer L_2, then layer L_3, and so on, up to layer L_{n_l}, using the equations above that describe the forward propagation step. This is one example of a feedforward neural network, since the connectivity graph does not have any directed loops or cycles.

Neural networks can also have multiple output units. For example, here is a network with two hidden layers layers L_2 and L_3 and two output units in layer L_4:



To train this network, we would need training examples $(x^{(i)}, y^{(i)})$ where $y^{(i)} \in \Re^2$. This sort of network is useful if there're multiple outputs that you're interested in predicting. (For example, in a medical diagnosis application, the vector x might give the input features of a patient, and the different outputs y_i's might indicate presence or absence of different diseases.)

2.2 Backpropagation Algorithm

Suppose we have a fixed training set $\{(x^{(1)}, y^{(1)}), \ldots, (x^{(m)}, y^{(m)})\}$ of m training examples. We can train our neural network using batch gradient descent. In detail, for a single training example (x, y), we define the cost function with respect to that single example to be:

$$J(W, b; x, y) = \frac{1}{2} \left\| h_{W,b}(x) - y \right\|^2 . \tag{1}$$

This is a (one-half) squared-error cost function. Given a training set of m examples, we then define the overall cost function to be:

$$\begin{aligned} J(W, b) &= \left[\frac{1}{m} \sum_{i=1}^{m} J(W, b; x^{(i)}, y^{(i)}) \right] + \frac{\lambda}{2} \sum_{l=1}^{n_l - 1} \sum_{i=1}^{s_l} \sum_{j=1}^{s_{l+1}} \left(W_{ji}^{(l)} \right)^2 &(2) \\ &= \left[\frac{1}{m} \sum_{i=1}^{m} \left(\frac{1}{2} \left\| h_{W,b}(x^{(i)}) - y^{(i)} \right\|^2 \right) \right] + \frac{\lambda}{2} \sum_{l=1}^{n_l - 1} \sum_{i=1}^{s_l} \sum_{j=1}^{s_{l+1}} \left(W_{ji}^{(l)} \right)^2 &(3) \end{aligned}$$

The first term in the definition of J(W,b) is an average sum-of-squares error term. The second term is a regularization term (also called a weight decay term) that tends to decrease the magnitude of the weights, and helps prevent overfitting. [1]

[1] Usually weight decay is not applied to the bias terms $b_i^{(l)}$, as reflected in our definition for $J(W, b)$. Applying weight decay to the bias units usually makes only a small difference to the final network, however. If you've taken CS229 (Machine Learning) at Stanford or watched the course's videos on YouTube, you may also recognize this weight decay as essentially a variant of the Bayesian regularization method you saw there, where we placed a Gaussian prior on the parameters and did MAP (instead of maximum likelihood) estimation.

The weight decay parameter λ controls the relative importance of the two terms. Note also the slightly overloaded notation: $J(W, b; x, y)$ is the squared error cost with respect to a single example; $J(W, b)$ is the overall cost function, which includes the weight decay term.

This cost function above is often used both for classification and for regression problems. For classification, we let $y = 0$ or 1 represent the two class labels (recall that the sigmoid activation function outputs values in $[0, 1]$; if we were using a tanh activation function, we would instead use -1 and $+1$ to denote the labels). For regression problems, we first scale our outputs to ensure that they lie in the $[0, 1]$ range (or if we were using a tanh activation function, then the $[-1, 1]$ range).

Our goal is to minimize $J(W, b)$ as a function of W and b. To train our neural network, we will initialize each parameter $W_{ij}^{(l)}$ and each $b_i^{(l)}$ to a small random value near zero (say according to a $Normal(0, \epsilon^2)$ distribution for some small ϵ, say 0.01), and then apply an optimization algorithm such as batch gradient descent. Since $J(W, b)$ is a non-convex function, gradient descent is susceptible to local optima; however, in practice gradient descent usually works fairly well. Finally, note that it is important to initialize the parameters randomly, rather than to all 0's. If all the parameters start off at identical values, then all the hidden layer units will end up learning the same function of the input (more formally, $W_{ij}^{(1)}$ will be the same for all values of i, so that $a_1^{(2)} = a_2^{(2)} = a_3^{(2)} = \ldots$ for any input x). The random initialization serves the purpose of symmetry breaking.

One iteration of gradient descent updates the parameters W, b as follows:

$$W_{ij}^{(l)} = W_{ij}^{(l)} - \alpha \frac{\partial}{\partial W_{ij}^{(l)}} J(W, b) \tag{4}$$

$$b_i^{(l)} = b_i^{(l)} - \alpha \frac{\partial}{\partial b_i^{(l)}} J(W, b) \tag{5}$$

where α is the learning rate. The key step is computing the partial derivatives above. We will now describe the backpropagation algorithm, which gives an efficient way to compute these partial derivatives.

We will first describe how backpropagation can be used to compute $\frac{\partial}{\partial W_{ij}^{(l)}} J(W, b; x, y)$ and $\frac{\partial}{\partial b_i^{(l)}} J(W, b; x, y)$, the partial derivatives of the cost function $J(W, b; x, y)$ defined with respect to a single example (x, y). Once we can compute these, we see that the derivative of the overall cost function $J(W, b)$ can be computed as:

$$\frac{\partial}{\partial W_{ij}^{(l)}} J(W, b) = \left[\frac{1}{m} \sum_{i=1}^{m} \frac{\partial}{\partial W_{ij}^{(l)}} J(W, b; x^{(i)}, y^{(i)}) \right] + \lambda W_{ij}^{(l)} \tag{6}$$

$$\frac{\partial}{\partial b_i^{(l)}} J(W, b) = \frac{1}{m} \sum_{i=1}^{m} \frac{\partial}{\partial b_i^{(l)}} J(W, b; x^{(i)}, y^{(i)}) \tag{7}$$

The two lines above differ slightly because weight decay is applied to W but not b.

The intuition behind the backpropagation algorithm is as follows. Given a training example (x, y), we will first run a "forward pass" to compute all the activations throughout the network, including the output value of the hypothesis $h_{W,b}(x)$. Then, for each node i in layer l, we would like to compute an "error term" $\delta_i^{(l)}$ that measures how much that node was "responsible" for any errors in our output. For an output node, we can directly measure the difference between the network's activation and the true target value, and use that to define $\delta_i^{(n_l)}$ (where layer n_l is the output layer). How about hidden units? For those, we will compute $\delta_i^{(l)}$ based on a weighted average of the error terms of the nodes that uses $a_i^{(l)}$ as an input. In detail, here is the backpropagation algorithm:

1. Perform a feedforward pass, computing the activations for layers $L_2, L_3, \ldots$, and so on up to the output layer L_{n_l}.

2. For each output unit i in layer n_l (the output layer), set

$$\delta_i^{(n_l)} = \frac{\partial}{\partial z_i^{(n_l)}} \frac{1}{2} \|y - h_{W,b}(x)\|^2 = -(y_i - a_i^{(n_l)}) \cdot f'(z_i^{(n_l)}) \tag{8}$$

   2

3. For $l = n_l - 1, n_l - 2, n_l - 3, \ldots, 2$

   For each node i in layer l, set

$$\delta_i^{(l)} = \left(\sum_{j=1}^{s_{l+1}} W_{ji}^{(l)} \delta_j^{(l+1)} \right) f'(z_i^{(l)})$$

   3

4. Compute the desired partial derivatives, which are given as:

$$\frac{\partial}{\partial W_{ij}^{(l)}} J(W, b; x, y) = a_j^{(l)} \delta_i^{(l+1)} \tag{17}$$

$$\frac{\partial}{\partial b_i^{(l)}} J(W, b; x, y) = \delta_i^{(l+1)}. \tag{18}$$

Finally, we can also re-write the algorithm using matrix-vectorial notation. We will use "$\bullet$" to denote the element-wise product operator (denoted "`.*`" in Matlab or Octave, and also called the

2

$$\delta_i^{(n_l)} = \frac{\partial}{\partial z_i^{n_l}} J(W, b; x, y) = \frac{\partial}{\partial z_i^{n_l}} \frac{1}{2} \|y - h_{W,b}(x)\|^2 \tag{9}$$

$$= \frac{\partial}{\partial z_i^{n_l}} \frac{1}{2} \sum_{j=1}^{S_{n_l}} (y_j - a_j^{(n_l)})^2 = \frac{\partial}{\partial z_i^{n_l}} \frac{1}{2} \sum_{j=1}^{S_{n_l}} (y_j - f(z_j^{(n_l)}))^2 \tag{10}$$

$$= -(y_i - f(z_i^{(n_l)})) \cdot f'(z_i^{(n_l)}) = -(y_i - a_i^{(n_l)}) \cdot f'(z_i^{(n_l)}) \tag{11}$$

3

Hadamard product), so that if $a = b \bullet c$, then $a_i = b_i c_i$. Similar to how we extended the definition of $f(\cdot)$ to apply element-wise to vectors, we also do the same for $f'(\cdot)$ (so that $f'([z_1, z_2, z_3]) = [f'(z_1), f'(z_2), f'(z_3)]$).

The algorithm can then be written:

1. Perform a feedforward pass, computing the activations for layers L_2, L_3, up to the output layer L_{n_l}, using the equations defining the forward propagation steps
2. For the output layer (layer n_l), set

$$\delta^{(n_l)} = -(y - a^{(n_l)}) \bullet f'(z^{(n_l)}) \tag{19}$$

3. For $l = n_l - 1, n_l - 2, n_l - 3, \ldots, 2$, Set

$$\delta^{(l)} = \left((W^{(l)})^T \delta^{(l+1)}\right) \bullet f'(z^{(l)}) \tag{20}$$

4. Compute the desired partial derivatives:

$$\nabla_{W^{(l)}} J(W, b; x, y) = \delta^{(l+1)} (a^{(l)})^T, \tag{21}$$
$$\nabla_{b^{(l)}} J(W, b; x, y) = \delta^{(l+1)}. \tag{22}$$

Implementation note: In steps 2 and 3 above, we need to compute $f'(z_i^{(l)})$ for each value of i. Assuming $f(z)$ is the sigmoid activation function, we would already have $a_i^{(l)}$ stored away from

---

$$\delta_i^{(n_l-1)} = \frac{\partial}{\partial z_i^{n_l-1}} J(W, b; x, y) = \frac{\partial}{\partial z_i^{n_l-1}} \frac{1}{2} \left\| y - h_{W,b}(x) \right\|^2 = \frac{\partial}{\partial z_i^{n_l-1}} \frac{1}{2} \sum_{j=1}^{S_{n_l}} (y_j - a_j^{(n_l)})^2 \tag{12}$$

$$= \frac{1}{2} \sum_{j=1}^{S_{n_l}} \frac{\partial}{\partial z_i^{n_l-1}} (y_j - a_j^{(n_l)})^2 = \frac{1}{2} \sum_{j=1}^{S_{n_l}} \frac{\partial}{\partial z_i^{n_l-1}} (y_j - f(z_j^{(n_l)}))^2 \tag{13}$$

$$= \sum_{j=1}^{S_{n_l}} -(y_j - f(z_j^{(n_l)})) \cdot \frac{\partial}{\partial z_i^{(n_l-1)}} f(z_j^{(n_l)}) = \sum_{j=1}^{S_{n_l}} -(y_j - f(z_j^{(n_l)})) \cdot f'(z_j^{(n_l)}) \cdot \frac{\partial z_j^{(n_l)}}{\partial z_i^{(n_l-1)}} \tag{14}$$

$$= \sum_{j=1}^{S_{n_l}} \delta_j^{(n_l)} \cdot \frac{\partial z_j^{(n_l)}}{\partial z_i^{n_l-1}} = \sum_{j=1}^{S_{n_l}} \left(\delta_j^{(n_l)} \cdot \frac{\partial}{\partial z_i^{n_l-1}} \sum_{k=1}^{S_{n_l-1}} f(z_k^{n_l-1}) \cdot W_{jk}^{n_l-1} \right) \tag{15}$$

$$= \sum_{j=1}^{S_{n_l}} \delta_j^{(n_l)} \cdot W_{ji}^{n_l-1} \cdot f'(z_i^{n_l-1}) = \left(\sum_{j=1}^{S_{n_l}} W_{ji}^{n_l-1} \delta_j^{(n_l)} \right) f'(z_i^{n_l-1}) \tag{16}$$

$n_l - 1$ n_l l $l+1$

$$\delta_i^{(l)} = \left(\sum_{j=1}^{s_{l+1}} W_{ji}^{(l)} \delta_j^{(l+1)} \right) f'(z_i^{(l)})$$

the forward pass through the network. Thus, using the expression that we worked out earlier for $f'(z)$, we can compute this as $f'(z_i^{(l)}) = a_i^{(l)}(1 - a_i^{(l)})$.

Finally, we are ready to describe the full gradient descent algorithm. In the pseudo-code below, $\Delta W^{(l)}$ is a matrix (of the same dimension as $W^{(l)}$), and $\Delta b^{(l)}$ is a vector (of the same dimension as $b^{(l)}$). Note that in this notation, "$\Delta W^{(l)}$" is a matrix, and in particular it isn't "Δ times $W^{(l)}$." We implement one iteration of batch gradient descent as follows:

1. Set $\Delta W^{(l)} := 0$, $\Delta b^{(l)} := 0$ (matrix/vector of zeros) for all l.
2. or $i = 1$ to m,
   (a) Use backpropagation to compute $\nabla_{W^{(l)}} J(W, b; x, y)$ and $\nabla_{b^{(l)}} J(W, b; x, y)$.
   (b) Set $\Delta W^{(l)} := \Delta W^{(l)} + \nabla_{W^{(l)}} J(W, b; x, y)$.
   (c) Set $\Delta b^{(l)} := \Delta b^{(l)} + \nabla_{b^{(l)}} J(W, b; x, y)$.
3. Update the parameters:

$$W^{(l)} = W^{(l)} - \alpha \left[\left(\frac{1}{m} \Delta W^{(l)} \right) + \lambda W^{(l)} \right] \tag{23}$$

$$b^{(l)} = b^{(l)} - \alpha \left[\frac{1}{m} \Delta b^{(l)} \right] \tag{24}$$

To train our neural network, we can now repeatedly take steps of gradient descent to reduce our cost function $J(W, b)$.

2.3 Gradient checking and advanced optimization

Backpropagation is a notoriously difficult algorithm to debug and get right, especially since many subtly buggy implementations of it – for example, one that has an off-by-one error in the indices and that thus only trains some of the layers of weights, or an implementation that omits the bias term – will manage to learn something that can look surprisingly reasonable (while performing less well than a correct implementation). Thus, even with a buggy implementation, it may not at all be apparent that anything is amiss. In this section, we describe a method for numerically checking the derivatives computed by your code to make sure that your implementation is correct. Carrying out the derivative checking procedure described here will significantly increase your confidence in the correctness of your code.

Suppose we want to minimize $J(\theta)$ as a function of θ. For this example, suppose $J : \Re \mapsto \Re$, so that $\theta \in \Re$. In this 1-dimensional case, one iteration of gradient descent is given by

$$\theta := \theta - \alpha \frac{d}{d\theta} J(\theta). \tag{25}$$

Suppose also that we have implemented some function $g(\theta)$ that purportedly computes $\frac{d}{d\theta} J(\theta)$, so that we implement gradient descent using the update $\theta := \theta - \alpha g(\theta)$. How can we check if our implementation of g is correct?

Recall the mathematical definition of the derivative as

$$\frac{d}{d\theta}J(\theta) = \lim_{\epsilon \to 0} \frac{J(\theta+\epsilon) - J(\theta-\epsilon)}{2\epsilon}. \tag{26}$$

Thus, at any specific value of θ, we can numerically approximate the derivative as follows:

$$\frac{J(\theta + \text{EPSILON}) - J(\theta - \text{EPSILON})}{2 \times \text{EPSILON}} \tag{27}$$

In practice, we set EPSILON to a small constant, say around 10^{-4}. (There's a large range of values of EPSILON that should work well, but we don't set EPSILON to be "extremely" small, say 10^{-20}, as that would lead to numerical roundoff errors.)

Thus, given a function $g(\theta)$ that is supposedly computing $\frac{d}{d\theta}J(\theta)$, we can now numerically verify its correctness by checking that

$$g(\theta) \approx \frac{J(\theta + \text{EPSILON}) - J(\theta - \text{EPSILON})}{2 \times \text{EPSILON}}. \tag{28}$$

The degree to which these two values should approximate each other will depend on the details of J. But assuming $\text{EPSILON} = 10^{-4}$, you'll usually find that the left- and right-hand sides of the above will agree to at least 4 significant digits (and often many more).

Now, consider the case where $\theta \in \Re^n$ is a vector rather than a single real number (so that we have n parameters that we want to learn), and $J : \Re^n \mapsto \Re$. In our neural network example we used "$J(W, b)$," but one can imagine "unrolling" the parameters W, b into a long vector θ. We now generalize our derivative checking procedure to the case where θ may be a vector.

Suppose we have a function $g_i(\theta)$ that purportedly computes $\frac{\partial}{\partial\theta_i}J(\theta)$; we'd like to check if g_i is outputting correct derivative values. Let $\theta^{(i+)} = \theta + \text{EPSILON} \times \vec{e}_i$, where

$$\vec{e}_i = \begin{bmatrix} 0 \\ 0 \\ \vdots \\ 1 \\ \vdots \\ 0 \end{bmatrix} \tag{29}$$

is the i-th basis vector (a vector of the same dimension as θ, with a "1" in the i-th position and "0"s everywhere else). So, $\theta^{(i+)}$ is the same as θ, except its i-th element has been incremented by EPSILON. Similarly, let $\theta^{(i-)} = \theta - \text{EPSILON} \times \vec{e}_i$ be the corresponding vector with the i-th element decreased by EPSILON. We can now numerically verify $g_i(\theta)$'s correctness by checking, for each i, that:

$$g_i(\theta) \approx \frac{J(\theta^{(i+)}) - J(\theta^{(i-)})}{2 \times \text{EPSILON}}. \tag{30}$$

When implementing backpropagation to train a neural network, in a correct implementation we will have that

$$\nabla_{W^{(l)}} J(W,b) = \left(\frac{1}{m}\Delta W^{(l)}\right) + \lambda W^{(l)} \tag{31}$$

$$\nabla_{b^{(l)}} J(W,b) = \frac{1}{m}\Delta b^{(l)}. \tag{32}$$

This result shows that the final block of psuedo-code in Backpropagation Algorithm 2.2 is indeed implementing gradient descent. To make sure your implementation of gradient descent is correct, it is usually very helpful to use the method described above to numerically compute the derivatives of $J(W,b)$, and thereby verify that your computations of $\left(\frac{1}{m}\Delta W^{(l)}\right) + \lambda W$ and $\frac{1}{m}\Delta b^{(l)}$ are indeed giving the derivatives you want.

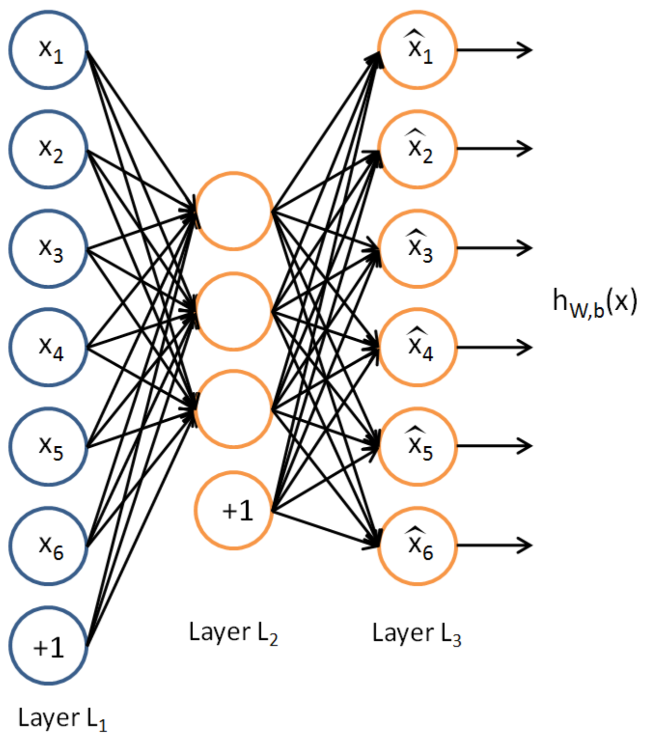
Finally, so far our discussion has centered on using gradient descent to minimize $J(\theta)$. If you have implemented a function that computes $J(\theta)$ and $\nabla_\theta J(\theta)$, it turns out there are more sophisticated algorithms than gradient descent for trying to minimize $J(\theta)$. For example, one can envision an algorithm that uses gradient descent, but automatically tunes the learning rate α so as to try to use a step-size that causes θ to approach a local optimum as quickly as possible. There are other algorithms that are even more sophisticated than this; for example, there are algorithms that try to find an approximation to the Hessian matrix, so that it can take more rapid steps towards a local optimum (similar to Newton's method). A full discussion of these algorithms is beyond the scope of these notes, but one example is the L-BFGS algorithm. (Another example is the conjugate gradient algorithm.) You will use one of these algorithms in the programming exercise. The main thing you need to provide to these advanced optimization algorithms is that for any θ, you have to be able to compute $J(\theta)$ and $\nabla_\theta J(\theta)$. These optimization algorithms will then do their own internal tuning of the learning rate/step-size α (and compute its own approximation to the Hessian, etc.) to automatically search for a value of θ that minimizes $J(\theta)$. Algorithms such as L-BFGS and conjugate gradient can often be much faster than gradient descent.

2.4 Autoencoders and Sparsity

So far, we have described the application of neural networks to supervised learning, in which we have labeled training examples. Now suppose we have only a set of unlabeled training examples $\{x^{(1)}, x^{(2)}, x^{(3)}, \ldots\}$, where $x^{(i)} \in \Re^n$. An autoencoder neural network is an unsupervised learning algorithm that applies backpropagation, setting the target values to be equal to the inputs. I.e., it uses $y^{(i)} = x^{(i)}$.

Here is an autoencoder:

The autoencoder tries to learn a function $h_{W,b}(x) \approx x$. In other words, it is trying to learn an approximation to the identity function, so as to output $\hat{x}$ that is similar to x. The identity function seems a particularly trivial function to be trying to learn; but by placing constraints on the network, such as by limiting the number of hidden units, we can discover interesting structure about the data. As a concrete example, suppose the inputs x are the pixel intensity values from a 10×10 image (100 pixels) so $n = 100$, and there are $s_2 = 50$ hidden units in layer L_2. Note that we also have $y \in \Re^{100}$. Since there are only 50 hidden units, the network is forced to learn a compressed representation of the input. I.e., given only the vector of hidden unit activations $a^{(2)} \in \Re^{50}$, it must try to reconstruct the 100-pixel input x. If the input were completely random—say, each x_i comes from an IID Gaussian independent of the other features—then this compression task would be very difficult. But if there is structure in the data, for example, if some of the input features are correlated, then this algorithm will be able to discover some of those correlations. In



fact, this simple autoencoder often ends up learning a low-dimensional representation very similar to PCAs.

Our argument above relied on the number of hidden units s_2 being small. But even when the number of hidden units is large (perhaps even greater than the number of input pixels), we can still discover interesting structure, by imposing other constraints on the network. In particular, if we impose a sparsity constraint on the hidden units, then the autoencoder will still discover interesting structure in the data, even if the number of hidden units is large.

Informally, we will think of a neuron as being "active" (or as "firing") if its output value is close to 1, or as being "inactive" if its output value is close to 0. We would like to constrain the neurons to be inactive most of the time. This discussion assumes a sigmoid activation function. If you are using a tanh activation function, then we think of a neuron as being inactive when it outputs values close to -1.

Recall that $a_j^{(2)}$ denotes the activation of hidden unit j in the autoencoder. However, this notation doesn't make explicit what was the input x that led to that activation. Thus, we will write $a_j^{(2)}(x)$ to denote the activation of this hidden unit when the network is given a specific input x. Further, let

$$\hat{\rho}_j = \frac{1}{m}\sum_{i=1}^{m}\left[a_j^{(2)}(x^{(i)})\right] \tag{33}$$

be the average activation of hidden unit j (averaged over the training set). We would like to (approximately) enforce the constraint

$$\hat{\rho}_j = \rho, \tag{34}$$

where ρ is a sparsity parameter, typically a small value close to zero (say $\rho = 0.05$). In other words, we would like the average activation of each hidden neuron j to be close to 0.05 (say). To satisfy this constraint, the hidden unit's activations must mostly be near 0.

To achieve this, we will add an extra penalty term to our optimization objective that penalizes $\hat{\rho}_j$ deviating significantly from ρ. Many choices of the penalty term will give reasonable results. We will choose the following:

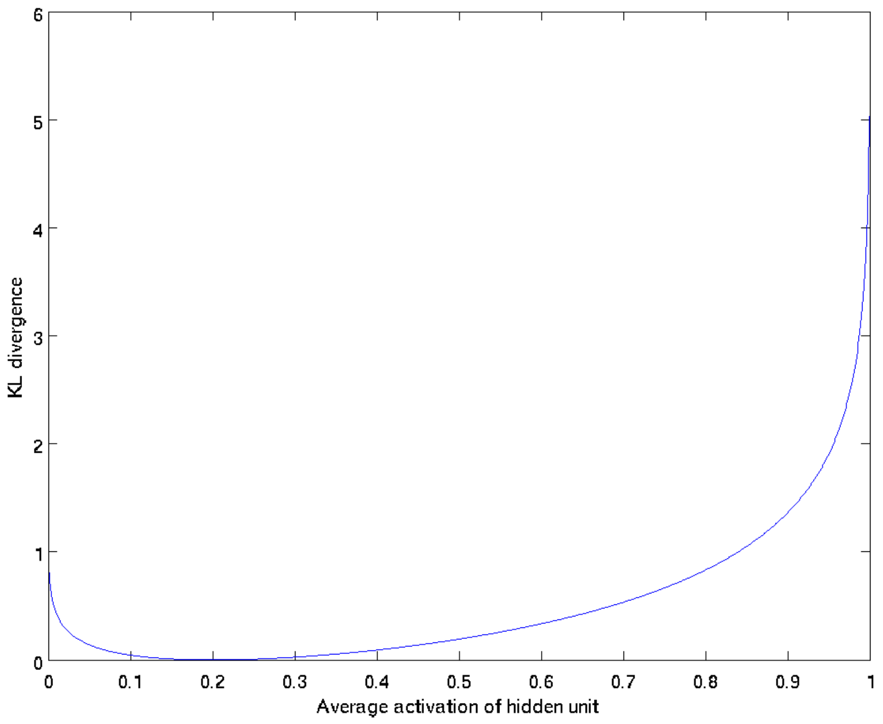
$$\sum_{j=1}^{s_2} \rho \log \frac{\rho}{\hat{\rho}_j} + (1-\rho) \log \frac{1-\rho}{1-\hat{\rho}_j}. \tag{35}$$

Here, s_2 is the number of neurons in the hidden layer, and the index j is summing over the hidden units in our network. If you are familiar with the concept of KL divergence, this penalty term is based on it, and can also be written

$$\sum_{j=1}^{s_2} \mathrm{KL}(\rho || \hat{\rho}_j), \tag{36}$$

where $\mathrm{KL}(\rho||\hat{\rho}_j) = \rho \log \frac{\rho}{\hat{\rho}_j} + (1-\rho) \log \frac{1-\rho}{1-\hat{\rho}_j}$ is the Kullback-Leibler (KL) divergence between a Bernoulli random variable with mean ρ and a Bernoulli random variable with mean $\hat{\rho}_j$. KL-divergence is a standard function for measuring how different two different distributions are. (If you've not seen KL-divergence before, don't worry about it; everything you need to know about it is contained in these notes.)

This penalty function has the property that $\mathrm{KL}(\rho||\hat{\rho}_j) = 0$ if $\hat{\rho}_j = \rho$, and otherwise it increases monotonically as $\hat{\rho}_j$ diverges from ρ. For example, in the figure below, we have set $\rho = 0.2$, and plotted $\mathrm{KL}(\rho||\hat{\rho}_j)$ for a range of values of $\hat{\rho}_j$:



We see that the KL-divergence reaches its minimum of 0 at $\hat{\rho}_j = \rho$, and blows up (it actually approaches ∞) as $\hat{\rho}_j$ approaches 0 or 1. Thus, minimizing this penalty term has the effect of causing $\hat{\rho}_j$ to be close to ρ.

Our overall cost function is now

$$J_{\text{sparse}}(W,b) = J(W,b) + \beta \sum_{j=1}^{s_2} \text{KL}(\rho || \hat{\rho}_j), \tag{37}$$

where $J(W,b)$ is as defined previously, and β controls the weight of the sparsity penalty term. The term $\hat{\rho}_j$ (implicitly) depends on W,b also, because it is the average activation of hidden unit j, and the activation of a hidden unit depends on the parameters W,b.

To incorporate the KL-divergence term into your derivative calculation, there is a simple-to-implement trick involving only a small change to your code. Specifically, where previously for the second layer ($l=2$), during backpropagation you would have computed

$$\delta_i^{(2)} = \left(\sum_{j=1}^{s_2} W_{ji}^{(2)} \delta_j^{(3)} \right) f'(z_i^{(2)}), \tag{38}$$

now instead compute

$$\delta_i^{(2)} = \left(\left(\sum_{j=1}^{s_2} W_{ji}^{(2)} \delta_j^{(3)} \right) + \beta \left(-\frac{\rho}{\hat{\rho}_i} + \frac{1-\rho}{1-\hat{\rho}_i} \right) \right) f'(z_i^{(2)}). \tag{39}$$

One subtlety is that you'll need to know $\hat{\rho}_i$ to compute this term. Thus, you'll need to compute a forward pass on all the training examples first to compute the average activations on the training set, before computing backpropagation on any example. If your training set is small enough to fit comfortably in computer memory (this will be the case for the programming assignment), you can compute forward passes on all your examples and keep the resulting activations in memory and compute the $\hat{\rho}_i s$. Then you can use your precomputed activations to perform backpropagation on all your examples. If your data is too large to fit in memory, you may have to scan through your examples computing a forward pass on each to accumulate (sum up) the activations and compute $\hat{\rho}_i$ (discarding the result of each forward pass after you have taken its activations $a_i^{(2)}$ into account for computing $\hat{\rho}_i$). Then after having computed $\hat{\rho}_i$, you'd have to redo the forward pass for each example so that you can do backpropagation on that example. In this latter case, you would end up computing a forward pass twice on each example in your training set, making it computationally less efficient.

The full derivation showing that the algorithm above results in gradient descent is beyond the scope of these notes. But if you implement the autoencoder using backpropagation modified this way, you will be performing gradient descent exactly on the objective $J_{\text{sparse}}(W,b)$. Using the derivative checking method, you will be able to verify this for yourself as well.

2.5 Visualizing a Trained Autoencoder

Having trained a (sparse) autoencoder, we would now like to visualize the function learned by the algorithm, to try to understand what it has learned. Consider the case of training an autoencoder on 10×10 images, so that $n = 100$. Each hidden unit i computes a function of the input:

$$a_i^{(2)} = f\left(\sum_{j=1}^{100} W_{ij}^{(1)} x_j + b_i^{(1)} \right). \tag{40}$$

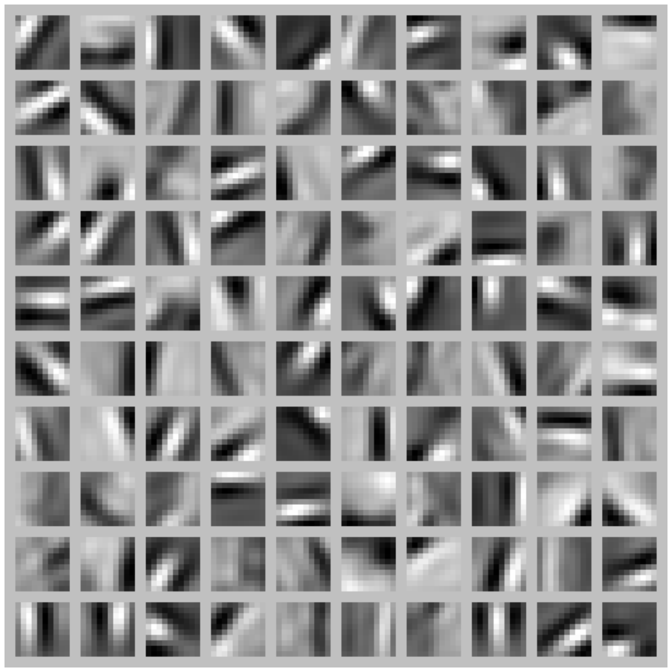
We will visualize the function computed by hidden unit i — which depends on the parameters $W^{(1)}_{ij}$ (ignoring the bias term for now)—using a 2D image. In particular, we think of $a^{(2)}_i$ as some non-linear feature of the input x. We ask: What input image x would cause $a^{(2)}_i$ to be maximally activated? (Less formally, what is the feature that hidden unit i is looking for?) For this question to have a non-trivial answer, we must impose some constraints on x. If we suppose that the input is norm constrained by $||x||^2 = \sum_{i=1}^{100} x_i^2 \leq 1$, then one can show (try doing this yourself) that the input which maximally activates hidden unit i is given by setting pixel x_j (for all 100 pixels, $j = 1, \ldots, 100$) to

$$x_j = \frac{W^{(1)}_{ij}}{\sqrt{\sum_{j=1}^{100} (W^{(1)}_{ij})^2}}. \tag{41}$$

By displaying the image formed by these pixel intensity values, we can begin to understand what feature hidden unit i is looking for.

If we have an autoencoder with 100 hidden units (say), then we our visualization will have 100 such images—one per hidden unit. By examining these 100 images, we can try to understand what the ensemble of hidden units is learning.

When we do this for a sparse autoencoder (trained with 100 hidden units on 10x10 pixel inputs) [4] we get the following result:



Each square in the figure above shows the (norm bounded) input image x that maximally actives one of 100 hidden units. We see that the different hidden units have learned to detect edges at different positions and orientations in the image.

These features are, not surprisingly, useful for such tasks as object recognition and other vision tasks. When applied to other input domains (such as audio), this algorithm also learns useful representations/features for those domains too.

[4] The learned features were obtained by training on whitened natural images. Whitening is a preprocessing step which removes redundancy in the input, by causing adjacent pixels to become less correlated.

2.6 Sparse Autoencoder Notation Summary

Here is a summary of the symbols used in our derivation of the sparse autoencoder:

Table 1: Sparse Autoencoder Notation Summary

Symbol	Meaning
x	Input features for a training example, $x \in \Re^n$.
y	Output/target values. Here, y can be vector valued. In the case of an autoencoder, $y = x$.
$(x^{(i)}, y^{(i)})$	The i-th training example
$h_{W,b}(x)$	Output of our hypothesis on input x, using parameters W, b. This should be a vector of the same dimension as the target value y.
$W_{ij}^{(l)}$	The parameter associated with the connection between unit j in layer l, and unit i in layer $l+1$.
$b_i^{(l)}$	The bias term associated with unit i in layer $l+1$. Can also be thought of as the parameter associated with the connection between the bias unit in layer l and unit i in layer $l+1$.
θ	Our parameter vector. It is useful to think of this as the result of taking the parameters W, b and "unrolling them into a long column vector.
$a_i^{(l)}$	Activation (output) of unit i in layer l of the network. In addition, since layer L_1 is the input layer, we also have $a_i^{(1)} = x_i$.
$f(\cdot)$	The activation function. Throughout these notes, we used $f(z) = \mathsf{tanh}(z)$.
$z_i^{(l)}$	Total weighted sum of inputs to unit i in layer l. Thus, $a_i^{(l)} = f(z_i^{(l)})$.
α	Learning rate parameter
s_l	Number of units in layer l (not counting the bias unit).
n_l	Number layers in the network. Layer L_1 is usually the input layer, and layer L_{n_l} the output layer.
λ	Weight decay parameter.
$\hat{x}$	For an autoencoder, its output; i.e., its reconstruction of the input x. Same meaning as $h_{W,b}(x)$.
ρ	Sparsity parameter, which specifies our desired level of sparsity
$\hat{\rho}_i$	The average activation of hidden unit i (in the sparse autoencoder).
β	Weight of the sparsity penalty term (in the sparse autoencoder objective).

2.7 Exercise: Sparse Autoencoder

You may want to download the related document from http://nlp.stanford.edu/~socherr/sparseAutoencoder_2011new.pdf and http://www.stanford.edu/class/cs294a/cs294a_2011-assignment.pdf.

Sparse autoencoder implementation

In this problem set, you will implement the sparse autoencoder algorithm, and show how it discovers that edges are a good representation for natural images. (Images provided by Bruno Olshausen.) The sparse autoencoder algorithm is described in the lecture notes found on the course website.

In the file http://ufldl.stanford.edu/wiki/resources/sparseae_exercise.zip , we have provided some starter code in Matlab. You should write your code at the places indicated in the files (`"YOUR CODE HERE"`). You have to complete the following files: `sampleIMAGES.m, sparseAutoencoderCost.m, computeNumericalGradient.m`. The starter code in `train.m` shows how these functions are used.

Specifically, in this exercise you will implement a sparse autoencoder, trained with 8×8 image patches using the L-BFGS optimization algorithm.

A note on the software: The provided .zip file includes a subdirectory `minFunc` with 3rd party software implementing L-BFGS, that is licensed under a Creative Commons, Attribute, Non-Commercial license. If you need to use this software for commercial purposes, you can download and use a different function (fminlbfgs) that can serve the same purpose, but runs $3\times$ slower for this exercise (and thus is less recommended). You can read more about this in the http://deeplearning.stanford.edu/wiki/index.php/Fminlbfgs_Details page.

2.7.1 Step 1: Generate training set

The first step is to generate a training set. To get a single training example x, randomly pick one of the 10 images, then randomly sample an 8×8 image patch from the selected image, and convert the image patch (either in row-major order or column-major order; it doesn't matter) into a 64-dimensional vector to get a training example $x \in \Re^{64}$.

Complete the code in `sampleIMAGES.m`. Your code should sample 10000 image patches and concatenate them into a 64×10000 matrix.

To make sure your implementation is working, run the code in "Step 1" of train.m. This should result in a plot of a random sample of 200 patches from the dataset.

Implementational tip: When we run our implemented `sampleImages()`, it takes under 5 seconds. If your implementation takes over 30 seconds, it may be because you are accidentally making a copy of an entire 512×512 image each time you're picking a random image. By copying a 512×512 image 10000 times, this can make your implementation much less efficient. While this doesn't slow down your code significantly for this exercise (because we have only 10000 examples), when we scale to much larger problems later this quarter with 10^6 or more examples, this will significantly slow down your code. Please implement `sampleIMAGES` so that you aren't making a copy of an entire 512×512 image each time you need to cut out an 8×8 image patch.

2.7.2 Step 2: Sparse autoencoder objective

Implement code to compute the sparse autoencoder cost function $J_{sparse}(W, b)$ (Section 3 of the lecture notes) and the corresponding derivatives of J_{sparse} with respect to the different parameters.

Use the sigmoid function for the activation function, $f(z) = \frac{1}{1+e^{-z}}$. In particular, complete the code in `sparseAutoencoderCost.m`.

The sparse autoencoder is parameterized by matrices $W^{(1)} \in \Re^{s_1 \times s_2}, W^{(2)} \in \Re^{s_2 \times s_3}$ vectors $b^{(1)} \in \Re^{s_2}, b^{(2)} \in \Re^{s_3}$. However, for subsequent notational convenience, we will "unroll" all of these parameters into a very long parameter vector θ with $s_1 s_2 + s_2 s_3 + s_2 + s_3$ elements. The code for converting between the $(W^{(1)}, W^{(2)}, b^{(1)}, b^{(2)})$ and the θ parameterization is already provided in the starter code.

Implementational tip: The objective $J_{sparse}(W, b)$ contains 3 terms, corresponding to the squared error term, the weight decay term, and the sparsity penalty. You're welcome to implement this however you want, but for ease of debugging, you might implement the cost function and derivative computation (backpropagation) only for the squared error term first (this corresponds to setting $\lambda = \beta = 0$), and implement the gradient checking method in the next section to first verify that this code is correct. Then only after you have verified that the objective and derivative calculations corresponding to the squared error term are working, add in code to compute the weight decay and sparsity penalty terms and their corresponding derivatives.

2.7.3 Step 3: Gradient checking

Following Section 2.3 of the lecture notes, implement code for gradient checking. Specifically, complete the code in `computeNumericalGradient.m`. Please use EPSILON $= 10^{-4}$ as described in the lecture notes.

We've also provided code in checkNumericalGradient.m for you to test your code. This code defines a simple quadratic function $h : \Re^2 \mapsto \Re$ given by $h(x) = x_1^2 + 3x_1x_2$, and evaluates it at the point $x = (4, 10)^T$. It allows you to verify that your numerically evaluated gradient is very close to the true (analytically computed) gradient.

After using checkNumericalGradient.m to make sure your implementation is correct, next use computeNumericalGradient.m to make sure that your sparseAutoencoderCost.m is computing derivatives correctly. For details, see Steps 3 in train.m. We strongly encourage you not to proceed to the next step until you've verified that your derivative computations are correct.

Implementational tip: If you are debugging your code, performing gradient checking on smaller models and smaller training sets (e.g., using only 10 training examples and 1-2 hidden units) may speed things up.

2.7.4 Step 4: Train the sparse autoencoder

Now that you have code that computes Jsparse and its derivatives, we're ready to minimize Jsparse with respect to its parameters, and thereby train our sparse autoencoder.

We will use the L-BFGS algorithm. This is provided to you in a function called minFunc (code provided by Mark Schmidt) included in the starter code. (For the purpose of this assignment, you only need to call minFunc with the default parameters. You do not need to know how L-BFGS works.) We have already provided code in train.m (Step 4) to call minFunc. The minFunc code assumes that the parameters to be optimized are a long parameter vector; so we will use the "θ" parameterization rather than the "$(W^{(1)}, W^{(2)}, b^{(1)}, b^{(2)})$" parameterization when passing our parameters to it.

Train a sparse autoencoder with 64 input units, 25 hidden units, and 64 output units. In our starter code, we have provided a function for initializing the parameters. We initialize the biases $b_i^{(l)}$ to zero, and the weights $W_{ij}^{(l)}$ to random numbers drawn uniformly from the interval

$\left[-\sqrt{\frac{6}{n_{\text{in}}+n_{\text{out}}+1}}, \sqrt{\frac{6}{n_{\text{in}}+n_{\text{out}}+1}}\right]$, where n_{in} is the fan-in (the number of inputs feeding into a node) and n_{out} is the fan-in (the number of units that a node feeds into).

The values we provided for the various parameters (λ, β, ρ, etc.) should work, but feel free to play with different settings of the parameters as well.

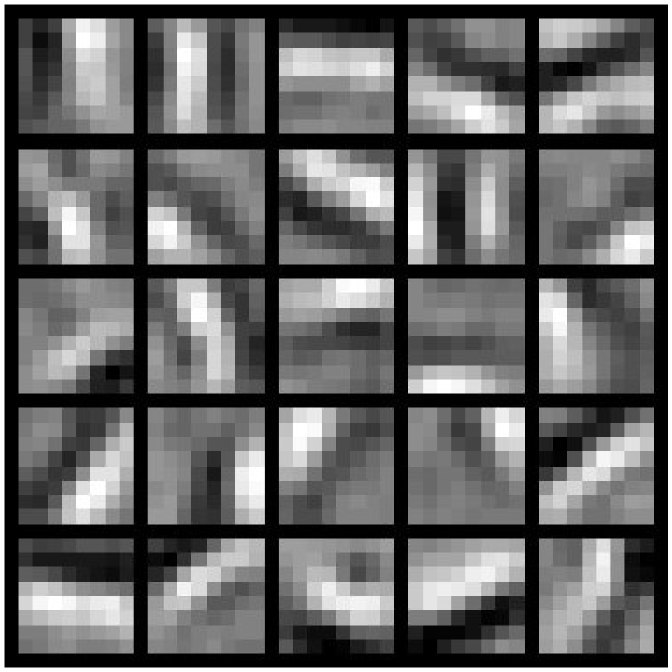
Implementational tip: Once you have your backpropagation implementation correctly computing the derivatives (as verified using gradient checking in Step 3), when you are now using it with L-BFGS to optimize $J_{sparse}(W, b)$, make sure you're not doing gradient-checking on every step. Backpropagation can be used to compute the derivatives of Jsparse(W,b) fairly efficiently, and if you were additionally computing the gradient numerically on every step, this would slow down your program significantly.

2.7.5 Step 5: Visualization

After training the autoencoder, use `display_network.m` to visualize the learned weights. (See `train.m`, Step 5.) Run "`print -djpeg weights.jpg`" to save the visualization to a file "`weights.jpg`" (which you will submit together with your code).

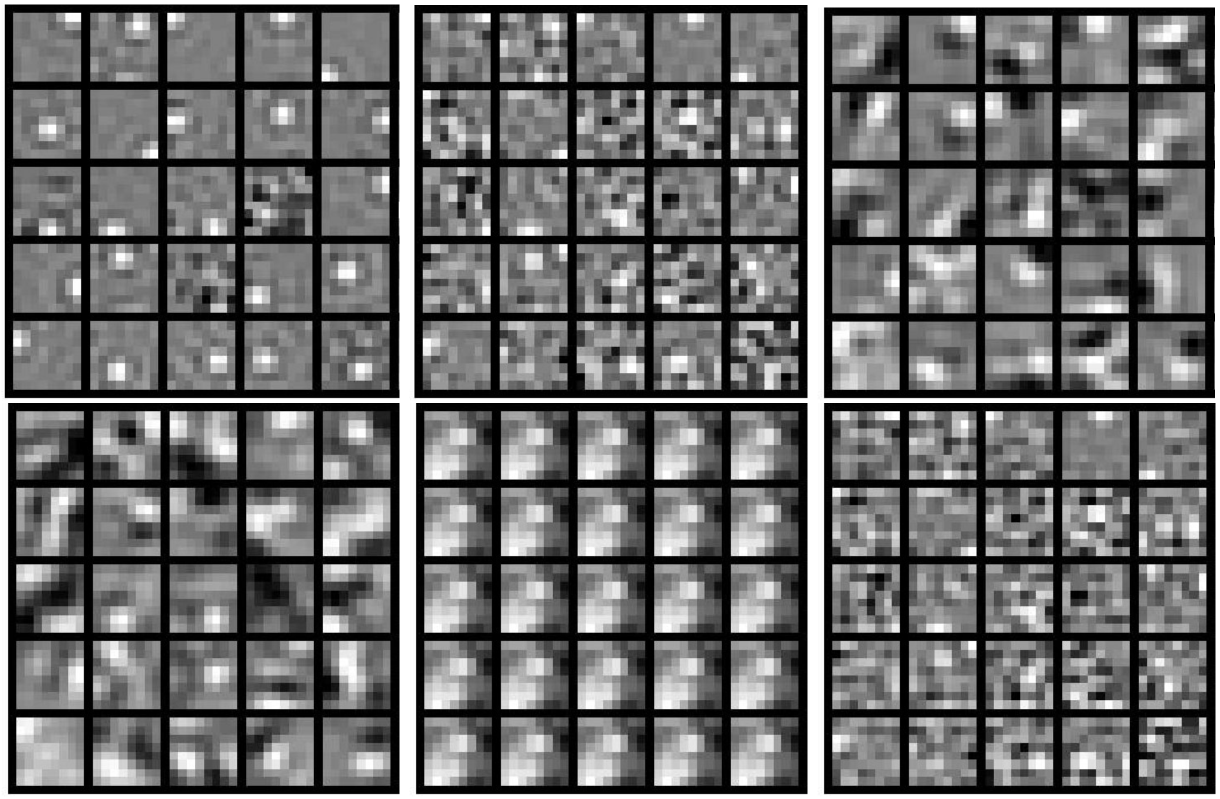
Results

To successfully complete this assignment, you should demonstrate your sparse autoencoder algorithm learning a set of edge detectors. For example, this was the visualization we obtained:



Our implementation took around 5 minutes to run on a fast computer. In case you end up needing to try out multiple implementations or different parameter values, be sure to budget enough time for debugging and to run the experiments you'll need.

Also, by way of comparison, here are some visualizations from implementations that we do not consider successful (either a buggy implementation, or where the parameters were poorly tuned):



3 Vectorized implementation

3.1 Vectorization

When working with learning algorithms, having a faster piece of code often means that you'll make progress faster on your project. For example, if your learning algorithm takes 20 minutes to run to completion, that means you can “try" up to 3 new ideas per hour. But if your code takes 20 hours to run, that means you can “try" only one idea a day, since that's how long you have to wait to get feedback from your program. In this latter case, if you can speed up your code so that it takes only 10 hours to run, that can literally double your personal productivity!

Vectorization refers to a powerful way to speed up your algorithms. Numerical computing and parallel computing researchers have put decades of work into making certain numerical operations (such as matrix-matrix multiplication, matrix-matrix addition, matrix-vector multiplication) fast. The idea of vectorization is that we would like to express our learning algorithms in terms of these highly optimized operations.

For example, if $x \in \Re^{n+1}$ and $\theta \in \Re^{n+1}$ are vectors and you need to compute $z = \theta^T x$, you can implement (in Matlab):

```
z = 0;
for i=1:(n+1),
  z = z + theta(i) * x(i);
end;
```

or you can more simply implement

```
z = theta' * x;
```

The second piece of code is not only simpler, but it will also run much faster.

More generally, a good rule-of-thumb for coding Matlab/Octave is:

Whenever possible, avoid using explicit for-loops in your code.

In particular, the first code example used an explicit `for` loop. By implementing the same functionality without the `for` loop, we sped it up significantly. A large part of vectorizing our Matlab/Octave code will focus on getting rid of `for` loops, since this lets Matlab/Octave extract more parallelism from your code, while also incurring less computational overhead from the interpreter.

In terms of a strategy for writing your code, initially you may find that vectorized code is harder to write, read, and/or debug, and that there may be a tradeoff in ease of programming/debugging vs. running time. Thus, for your first few programs, you might choose to first implement your algorithm without too many vectorization tricks, and verify that it is working correctly (perhaps by running on a small problem). Then only after it is working, you can vectorize your code one piece at a time, pausing after each piece to verify that your code is still computing the same result as before. At the end, you'll then hopefully have a correct, debugged, and vectorized/efficient piece of code.

After you become familiar with the most common vectorization methods and tricks, you'll find that it usually isn't much effort to vectorize your code. Doing so will make your code run much faster and, in some cases, simplify it too.

3.2 Logistic Regression Vectorization Example

Consider training a logistic regression model using batch gradient ascent. Suppose our hypothesis is

$$h_\theta(x) = \frac{1}{1+\exp(-\theta^T x)}, \tag{42}$$

where (following the notational convention from the OpenClassroom videos and from CS229) we let $x_0 = 1$, so that $x \in \Re^{n+1}$ and $\theta \in \Re^{n+1}$, and θ_0 is our intercept term. We have a training set $\{(x^{(1)}, y^{(1)}), \ldots, (x^{(m)}, y^{(m)})\}$ of m examples, and the batch gradient ascent update rule is $\theta := \theta + \alpha \nabla_\theta \ell(\theta)$, where $\ell(\theta)$ is the log likelihood and $\nabla_\theta \ell(\theta)$ is its derivative.

[Note: Most of the notation below follows that defined in the OpenClassroom videos or in the class CS229: Machine Learning. For details, see either the http://openclassroom.stanford.edu/MainFolder/CoursePage.php?course=MachineLearning or Lecture Notes #1 of http://cs229.stanford.edu/ .]

We thus need to compute the gradient:

$$\nabla_\theta \ell(\theta) = \sum_{i=1}^{m} \left(y^{(i)} - h_\theta(x^{(i)})\right) x_j^{(i)}. \tag{43}$$

Suppose that the Matlab/Octave variable x is a matrix containing the training inputs, so that `x(:,i)` is the i-th training example $x^{(i)}$, and `x(j,i)` is $x_j^{(i)}$. Further, suppose the Matlab/Octave variable y is a row vector of the labels in the training set, so that the variable `y(i)` is $y^{(i)} \in \{0, 1\}$. (Here we differ from the OpenClassroom/CS229 notation. Specifically, in the matrix-valued x we stack the training inputs in columns rather than in rows; and y$\in \Re^{1\times m}$ is a row vector rather than a column vector.)

Here's truly horrible, extremely slow, implementation of the gradient computation:

```
% Implementation 1
grad = zeros(n+1,1);
for i=1:m,
  h = sigmoid(theta'*x(:,i));
  temp = y(i) - h;
  for j=1:n+1,
    grad(j) = grad(j) + temp * x(j,i);
  end;
end;
```

The two nested `for`-loops makes this very slow. Here's a more typical implementation, that partially vectorizes the algorithm and gets better performance:

```
% Implementation 2
grad = zeros(n+1,1);
for i=1:m,
  grad = grad + (y(i) - sigmoid(theta'*x(:,i)))* x(:,i);
end;
```

However, it turns out to be possible to even further vectorize this. If we can get rid of the `for`-loop, we can significantly speed up the implementation. In particular, suppose `b` is a column vector, and `A` is a matrix. Consider the following ways of computing `A * b`:

```
% Slow implementation of matrix-vector multiply
grad = zeros(n+1,1);
for i=1:m,
  grad = grad + b(i) * A(:,i);  % more commonly written A(:,i)*b(i)
end;

% Fast implementation of matrix-vector multiply
grad = A*b;
```

We recognize that Implementation 2 of our gradient descent calculation above is using the slow version with a `for`-loop, with `b(i)` playing the role of (`y(i) - sigmoid(theta'*x(:,i))`), and `A` playing the role of `x`. We can derive a fast implementation as follows:

```
% Implementation 3
grad = x * (y- sigmoid(theta'*x))';
```

Here, we assume that the Matlab/Octave `sigmoid(z)` takes as input a vector `z`, applies the sigmoid function component-wise to the input, and returns the result. The output of `sigmoid(z)` is therefore itself also a vector, of the same dimension as the input `z`.

When the training set is large, this final implementation takes the greatest advantage of Matlab/Octave's highly optimized numerical linear algebra libraries to carry out the matrix-vector operations, and so this is far more efficient than the earlier implementations.

Coming up with vectorized implementations isn't always easy, and sometimes requires careful thought. But as you gain familiarity with vectorized operations, you'll find that there are design patterns (i.e., a small number of ways of vectorizing) that apply to many different pieces of code.

3.3 Neural Network Vectorization

In this section, we derive a vectorized version of our neural network. In our earlier description of Neural Networks(2.1), we had already given a partially vectorized implementation, that is quite efficient if we are working with only a single example at a time. We now describe how to implement the algorithm so that it simultaneously processes multiple training examples. Specifically, we will do this for the forward propagation and backpropagation steps, as well as for learning a sparse set of features.

3.3.1 Forward propagation

Consider a 3 layer neural network (with one input, one hidden, and one output layer), and suppose `x` is a column vector containing a single training example $x^{(i)} \in \Re^n$. Then the forward propagation step is given by:

$$\begin{aligned}
z^{(2)} &= W^{(1)}x + b^{(1)} && (44)\\
a^{(2)} &= f(z^{(2)}) && (45)\\
z^{(3)} &= W^{(2)}a^{(2)} + b^{(2)} && (46)\\
h_{W,b}(x) &= a^{(3)} = f(z^{(3)}) && (47)
\end{aligned}$$

This is a fairly efficient implementation for a single example. If we have m examples, then we would wrap a `for` loop around this.

Concretely, following the Logistic Regression Vectorization Example(3.2), let the Matlab/Octave variable `x` be a matrix containing the training inputs, so that `x(:,i)` is the i-th training example. We can then implement forward propagation as:

```
% Unvectorized implementation
for i=1:m,
  z2 = W1 * x(:,i) + b1;
  a2 = f(z2);
  z3 = W2 * a2 + b2;
  h(:,i) = f(z3);
end;
```

Can we get rid of the `for` loop? For many algorithms, we will represent intermediate stages of computation via vectors. For example, `z2, a2`, and `z3` here are all column vectors that're used to compute the activations of the hidden and output layers. In order to take better advantage of parallelism and efficient matrix operations, we would like to have our algorithm operate simultaneously on many training examples. Let us temporarily ignore `b1` and `b2` (say, set them to zero for now). We can then implement the following:

```
% Vectorized implementation (ignoring b1, b2)
z2 = W1 * x;
a2 = f(z2);
z3 = W2 * a2;
h = f(z3)
```

In this implementation, `z2, a2`, and `z3` are all matrices, with one column per training example. A common design pattern in vectorizing across training examples is that whereas previously we had a column vector (such as `z2`) per training example, we can often instead try to compute a matrix so that all of these column vectors are stacked together to form a matrix. Concretely, in this example, `a2` becomes a s_2 by m matrix (where s_2 is the number of units in layer 2 of the network, and m is the number of training examples). And, the i-th column of `a2` contains the activations of the hidden units (layer 2 of the network) when the i-th training example `x(:,i)` is input to the network.

In the implementation above, we have assumed that the activation function `f(z)` takes as input a matrix `z`, and applies the activation function component-wise to the input. Note that your implementation of `f(z)` should use Matlab/Octave's matrix operations as much as possible, and avoid `for` loops as well. We illustrate this below, assuming that `f(z)` is the sigmoid activation function:

```
% Inefficient, unvectorized implementation of the activation function
function output = unvectorized_f(z)
output = zeros(size(z))
for i=1:size(z,1),
  for j=1:size(z,2),
    output(i,j) = 1/(1+exp(-z(i,j)));
  end;
end;
end

% Efficient, vectorized implementation of the activation function
function output = vectorized_f(z)
output = 1./(1+exp(-z));      % "./" is Matlab/Octave's element-wise division
    operator.
```

```
end
```

Finally, our vectorized implementation of forward propagation above had ignored `b1` and `b2`. To incorporate those back in, we will use Matlab/Octave's built-in `repmat` function. We have:

```
% Vectorized implementation of forward propagation
z2 = W1 * x + repmat(b1,1,m);
a2 = f(z2);
z3 = W2 * a2 + repmat(b2,1,m);
h = f(z3)
```

The result of `repmat(b1,1,m)` is a matrix formed by taking the column vector `b1` and stacking m copies of them in columns as follows

$$\begin{bmatrix} | & | & & | \\ \mathrm{b1} & \mathrm{b1} & \cdots & \mathrm{b1} \\ | & | & & | \end{bmatrix}.$$

This forms a s_2 by m matrix. Thus, the result of adding this to `W1 * x` is that each column of the matrix gets `b1` added to it, as desired. See Matlab/Octave's documentation (type "`help repmat`") for more information. As a Matlab/Octave built-in function, `repmat` is very efficient as well, and runs much faster than if you were to implement the same thing yourself using a `for` loop.

3.3.2 Backpropagation

We now describe the main ideas behind vectorizing backpropagation. Before reading this section, we strongly encourage you to carefully step through all the forward propagation code examples above to make sure you fully understand them. In this text, we'll only sketch the details of how to vectorize backpropagation, and leave you to derive the details in the Vectorization exercise(3.4).

We are in a supervised learning setting, so that we have a training set $\{(x^{(1)}, y^{(1)}), \ldots, (x^{(m)}, y^{(m)})\}$ of m training examples. (For the autoencoder, we simply set $y^{(i)} = x^{(i)}$, but our derivation here will consider this more general setting.)

Suppose we have s_3 dimensional outputs, so that our target labels are $y^{(i)} \in \Re^{s_3}$. In our Matlab/Octave datastructure, we will stack these in columns to form a Matlab/Octave variable `y`, so that the i-th column `y(:,i)` is $y^{(i)}$.

We now want to compute the gradient terms $\nabla_{W^{(l)}} J(W,b)$ and $\nabla_{b^{(l)}} J(W,b)$. Consider the first of these terms. Following our earlier description of the Backpropagation Algorithm(2.2), we had that for a single training example (x,y), we can compute the derivatives as

$$\delta^{(3)} = -(y - a^{(3)}) \bullet f'(z^{(3)}), \tag{48}$$

$$\delta^{(2)} = ((W^{(2)})^T \delta^{(3)}) \bullet f'(z^{(2)}), \tag{49}$$

$$\nabla_{W^{(2)}} J(W,b;x,y) = \delta^{(3)} (a^{(2)})^T, \tag{50}$$

$$\nabla_{W^{(1)}} J(W,b;x,y) = \delta^{(2)} (a^{(1)})^T. \tag{51}$$

Here, $\bullet$ denotes element-wise product. For simplicity, our description here will ignore the derivatives with respect to $b^{(l)}$, though your implementation of backpropagation will have to compute those derivatives too.

Suppose we have already implemented the vectorized forward propagation method, so that the matrix-valued `z2, a2, z3` and `h` are computed as described above. We can then implement an unvectorized version of backpropagation as follows:

```
gradW1 = zeros(size(W1));
gradW2 = zeros(size(W2));
for i=1:m,
  delta3 = -(y(:,i) - h(:,i)) .* fprime(z3(:,i));
  delta2 = W2'*delta3(:,i) .* fprime(z2(:,i));

  gradW2 = gradW2 + delta3*a2(:,i)';
  gradW1 = gradW1 + delta2*a1(:,i)';
end;
```

This implementation has a `for` loop. We would like to come up with an implementation that simultaneously performs backpropagation on all the examples, and eliminates this `for` loop.

To do so, we will replace the vectors `delta3` and `delta2` with matrices, where one column of each matrix corresponds to each training example. We will also implement a function `fprime(z)` that takes as input a matrix `z`, and applies $f'(\cdot)$ element-wise. Each of the four lines of Matlab in the for loop above can then be vectorized and replaced with a single line of Matlab code (without a surrounding `for` loop).

In the Vectorization exercise(3.4), we ask you to derive the vectorized version of this algorithm by yourself. If you are able to do it from this description, we strongly encourage you to do so. Here also are some Backpropagation vectorization hints(3.5); however, we encourage you to try to carry out the vectorization yourself without looking at the hints.

3.3.3 Sparse autoencoder

The sparse autoencoder(2.4) neural network has an additional sparsity penalty that constrains neurons' average firing rate to be close to some target activation ρ. When performing backpropagation on a single training example, we had taken into the account the sparsity penalty by computing the following:

$$\delta_i^{(2)} = \left(\left(\sum_{j=1}^{s_2} W_{ji}^{(2)} \delta_j^{(3)} \right) + \beta \left(-\frac{\rho}{\hat{\rho}_i} + \frac{1-\rho}{1-\hat{\rho}_i} \right) \right) f'(z_i^{(2)}). \tag{52}$$

In the unvectorized case, this was computed as:

```
% Sparsity Penalty Delta
sparsity_delta = - rho ./ rho_hat + (1 - rho) ./ (1 - rho_hat);
for i=1:m,
  ...
  delta2 = (W2'*delta3(:,i) + beta*sparsity_delta).* fprime(z2(:,i));
  ...
end;
```

The code above still had a `for` loop over the training set, and `delta2` was a column vector.

In contrast, recall that in the vectorized case, `delta2` is now a matrix with m columns corresponding to the m training examples. Now, notice that the `sparsity_delta` term is the same regardless of what training example we are processing. This suggests that vectorizing the computation above can be done by simply adding the same value to each column when constructing the `delta2` matrix. Thus, to vectorize the above computation, we can simply add `sparsity_delta` (e.g., using `repmat`) to each column of `delta2`.

3.4 Exercise: Vectorization

In the previous problem set, we implemented a sparse autoencoder for patches taken from natural images. In this problem set, you will vectorize your code to make it run much faster, and further adapt your sparse autoencoder to work on images of handwritten digits. Your network for learning from handwritten digits will be much larger than the one you'd trained on the natural images, and so using the original implementation would have been painfully slow. But with a vectorized implementation of the autoencoder, you will be able to get this to run in a reasonable amount of computation time.

3.4.1 Support Code/Data

The following additional files are required for this exercise:

- MNIST Dataset (Training Images) `http://yann.lecun.com/exdb/mnist/train-images-idx` `gz`
- MNIST Dataset (Training Labels) `http://yann.lecun.com/exdb/mnist/train-labels-idx` `gz`
- Support functions for loading MNIST in Matlab A

3.4.2 Step 1: Vectorize your Sparse Autoencoder Implementation

Using the ideas from Vectorization(3.1) and Neural Network Vectorization(3.3), vectorize your implementation of `sparseAutoencoderCost.m`. In our implementation, we were able to remove all for-loops with the use of matrix operations and repmat. (If you want to play with more advanced vectorization ideas, also type `help bsxfun`. The `bsxfun` function provides an alternative to `repmat` for some of the vectorization steps, but is not necessary for this exercise). A vectorized version of our sparse autoencoder code ran in under one minute on a fast computer (for learning 25 features from 10000 8×8 image patches).

(Note that you do not need to vectorize the code in the other files.)

3.4.3 Step 2: Learn features for handwritten digits

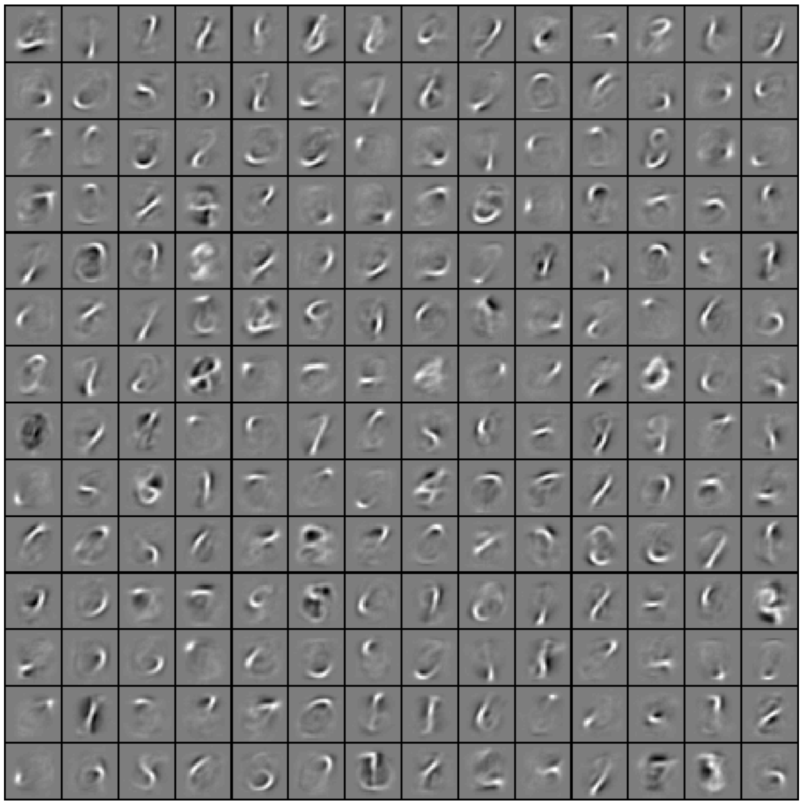
Now that you have vectorized the code, it is easy to learn larger sets of features on medium sized images. In this part of the exercise, you will use your sparse autoencoder to learn features for handwritten digits from the MNIST dataset.

The MNIST data is available at `http://yann.lecun.com/exdb/mnist/`. Download the file train-images-idx3-ubyte.gz and decompress it. After obtaining the source images, you should use helper functions that we provide to load the data into Matlab as matrices. While the helper functions that we provide(A) will load both the input examples x and the class labels y, for this assignment, you will only need the input examples x since the sparse autoencoder is an unsupervised learning algorithm. (In a later assignment, we will use the labels y as well.)

The following set of parameters worked well for us to learn good features on the MNIST dataset:

```
visibleSize = 28*28
hiddenSize = 196
sparsityParam = 0.1
lambda = 3e-3
beta = 3
patches = first 10000 images from the MNIST dataset
```

After 400 iterations of updates using minFunc, your autoencoder should have learned features that resemble pen strokes. In other words, this has learned to represent handwritten characters in terms of what pen strokes appear in an image. Our implementation takes around 15-20 minutes on a fast machine. Visualized, the features should look like the following image:



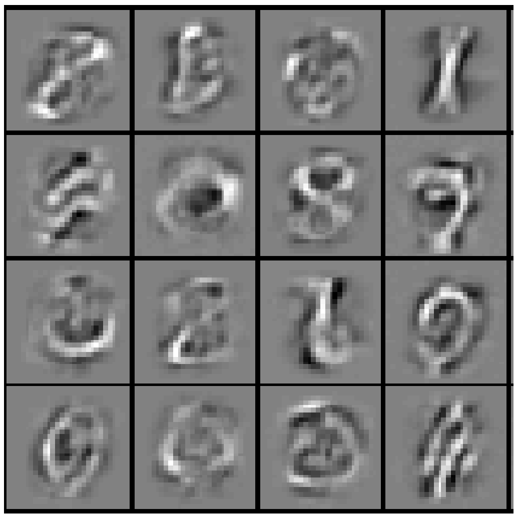
If your parameters are improperly tuned, or if your implementation of the autoencoder is buggy, you may get one of the following images instead:

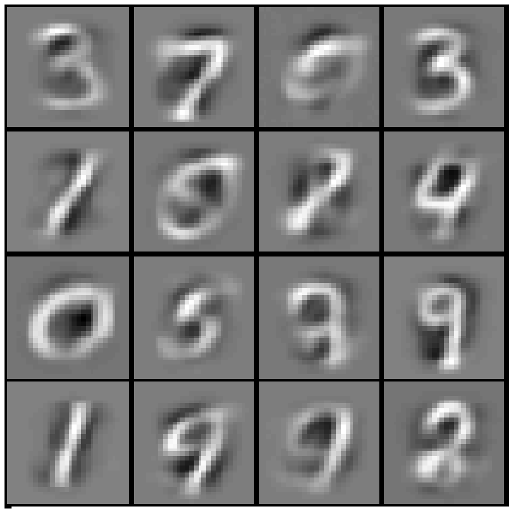
3.5 Backpropagation vectorization hints

Here, we give a few hints on how to vectorize the Backpropagation step. The hints here specifically build on our earlier description of how to vectorize a neural network(3.3).

Assume we have already implemented the vectorized forward propagation steps, so that the matrix-valued `z2, a2, z3` and `h` have already been computed. Here was our unvectorized implementation of `backprop`:

```
gradW1 = zeros(size(W1));
gradW2 = zeros(size(W2));
for i=1:m,
```





```
  delta3 = -(y(:,i) - h(:,i)) .* fprime(z3(:,i));
  delta2 = W2'*delta3(:,i) .* fprime(z2(:,i));

  gradW2 = gradW2 + delta3*a2(:,i)';
  gradW1 = gradW1 + delta2*a1(:,i)';
end;
```

Assume that we have implemented a version of `fprime(z)` that accepts matrix-valued inputs. We will use matrix-valued `delta3, delta2`. Here, `delta3` and `delta2` will have m columns, with one column per training example. We want to compute `delta3, delta2, gradW2` and `gradW1`.

Consider the computation for the matrix `delta3`, which can now be written:

```
for i=1:m,
  delta3(:,i) = -(y(:,i) - h(:,i)) .* fprime(z3(:,i));
end;
```

Each iteration of the for loop computes one column of `delta3`. You should be able to find a single line of Matlab to compute `delta3` as a function of the matrices `y, h` and `z3`. This lets you compute `delta3`. Similarly, you should also be able to find a single line of code to compute the entire matrix `delta2`, as a function of `W2, delta3` (which is now a matrix), and `z2`.

Next, consider the computation for `gradW2`. We can now write this as:

```
gradW2 = zeros(size(W2));
for i=1:m,
  gradW2 = gradW2 + delta3(:,i)*a2(:,i)';
end;
```

You should be able to find a single line of Matlab that replaces this for loop, and computes `gradW2` as a function of the matrices `delta3` and `a2`. If you're having trouble, take another look at the Logistic Regression Vectorization Example(3.2), which uses a related (but slightly different) vectorization step to get to the final implementation. Using a similar method, you will also be able to compute `gradW1` with a single line of code.

When you complete the derivation, you should be able to replace the unvectorized backpropagation code example above with just 4 lines of Matlab/Octave code.

4 Preprocessing: PCA and Whitening

4.1 PCA

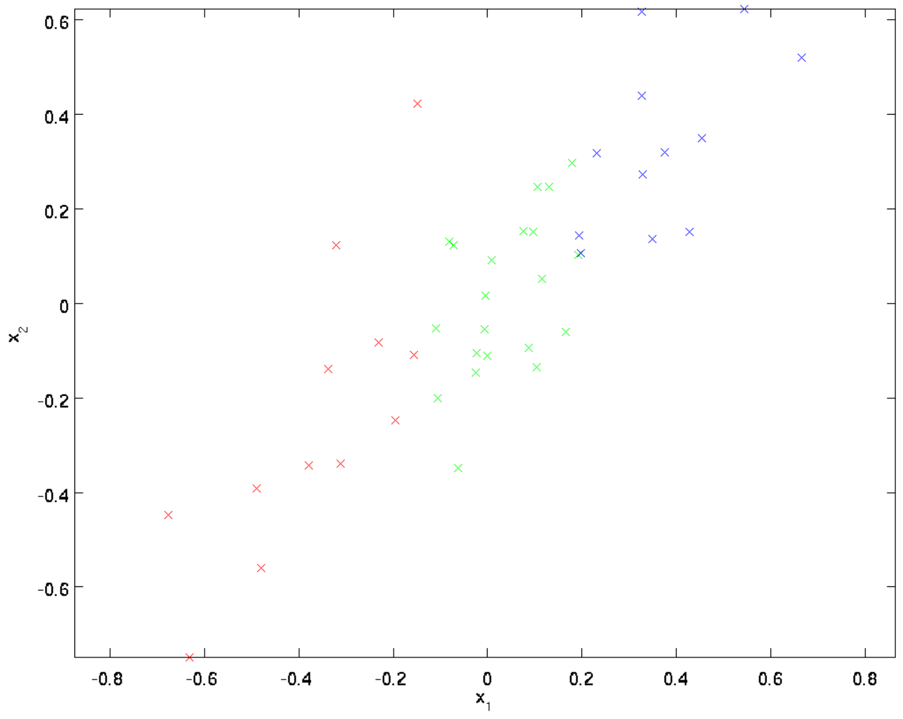
4.1.1 Introduction

Principal Components Analysis (PCA) is a dimensionality reduction algorithm that can be used to significantly speed up your unsupervised feature learning algorithm. More importantly, understanding PCA will enable us to later implement whitening, which is an important pre-processing step for many algorithms.

Suppose you are training your algorithm on images. Then the input will be somewhat redundant, because the values of adjacent pixels in an image are highly correlated. Concretely, suppose we are training on 16×16 grayscale image patches. Then $x \in \Re^{256}$ are 256 dimensional vectors, with one feature x_j corresponding to the intensity of each pixel. Because of the correlation between adjacent pixels, PCA will allow us to approximate the input with a much lower dimensional one, while incurring very little error.

4.1.2 Example and Mathematical Background

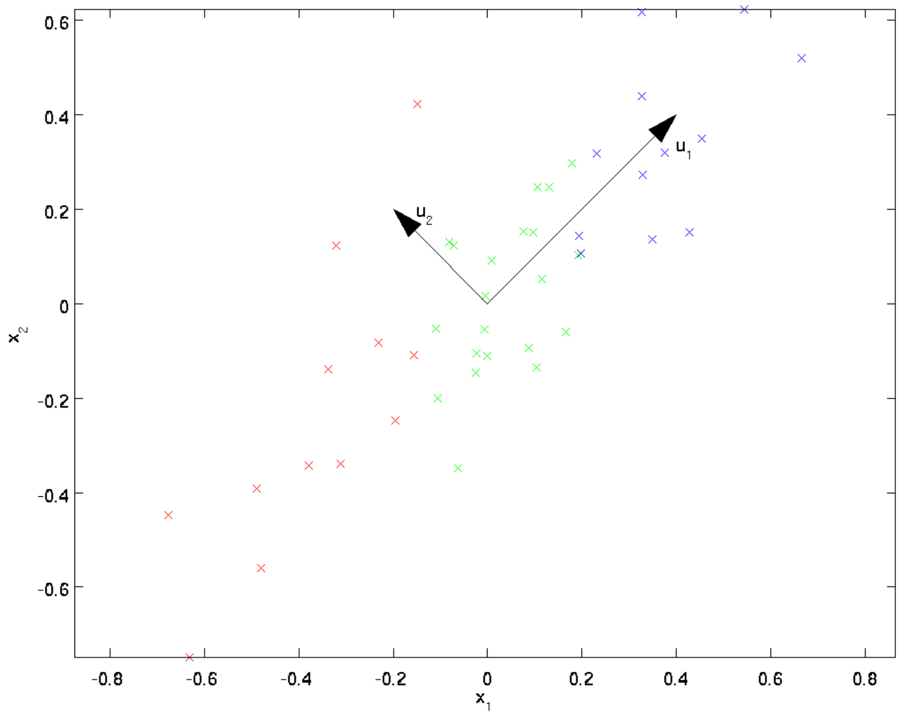
For our running example, we will use a dataset $\{x^{(1)}, x^{(2)}, \ldots, x^{(m)}\}$ with $n = 2$ dimensional inputs, so that $x^{(i)} \in \Re^2$. Suppose we want to reduce the data from 2 dimensions to 1. (In practice, we might want to reduce data from 256 to 50 dimensions, say; but using lower dimensional data in our example allows us to visualize the algorithms better.) Here is our dataset:



This data has already been pre-processed so that each of the features x_1 and x_2 have about the same mean (zero) and variance.

For the purpose of illustration, we have also colored each of the points one of three colors, depending on their x_1 value; these colors are not used by the algorithm, and are for illustration only.

PCA will find a lower-dimensional subspace onto which to project our data. From visually examining the data, it appears that u_1 is the principal direction of variation of the data, and u_2 the secondary direction of variation:



I.e., the data varies much more in the direction u_1 than u_2. To more formally find the directions u_1 and u_2, we first compute the matrix Σ as follows:

$$\Sigma = \frac{1}{m}\sum_{i=1}^{m}(x^{(i)})(x^{(i)})^T. \tag{53}$$

If x has zero mean, then Σ is exactly the covariance matrix of x. (The symbol "Σ", pronounced "Sigma", is the standard notation for denoting the covariance matrix. Unfortunately it looks just like the summation symbol, as in $\sum_{i=1}^{n} i$; but these are two different things.)

It can then be shown that u_1 — the principal direction of variation of the data — is the top (principal) eigenvector of Σ, and u_2 is the second eigenvector.

Note: If you are interested in seeing a more formal mathematical derivation/justification of this result, see the CS229 (Machine Learning) lecture notes on PCA[5] (link at bottom of this page). You won't need to do so to follow along this course, however.

You can use standard numerical linear algebra software to find these eigenvectors (see Implementation Notes). Concretely, let us compute the eigenvectors of Σ, and stack the eigenvectors in columns to form the matrix `U`:

[5]Stanford CS229 Machine Learning http://cs229.stanford.edu

$$U = \begin{bmatrix} | & | & & | \\ u_1 & u_2 & \cdots & u_n \\ | & | & & | \end{bmatrix} \tag{54}$$

Here, u_1 is the principal eigenvector (corresponding to the largest eigenvalue), u_2 is the second eigenvector, and so on. Also, let $\lambda_1, \lambda_2, \ldots, \lambda_n$ be the corresponding eigenvalues.

The vectors u_1 and u_2 in our example form a new basis in which we can represent the data. Concretely, let $x \in \Re^2$ be some training example. Then $u_1^T x$ is the length (magnitude) of the projection of x onto the vector u_1.

Similarly, $u_2^T x$ is the magnitude of x projected onto the vector u_2.

4.1.3 Rotating the Data

Thus, we can represent x in the (u_1, u_2)-basis by computing

$$x_{\text{rot}} = U^T x = \begin{bmatrix} u_1^T x \\ u_2^T x \end{bmatrix} \tag{55}$$

(The subscript "rot" comes from the observation that this corresponds to a rotation (and possibly reflection) of the original data.) Lets take the entire training set, and compute $x_{\text{rot}}^{(i)} = U^T x^{(i)}$ for every i. Plotting this transformed data x_{rot}, we get:

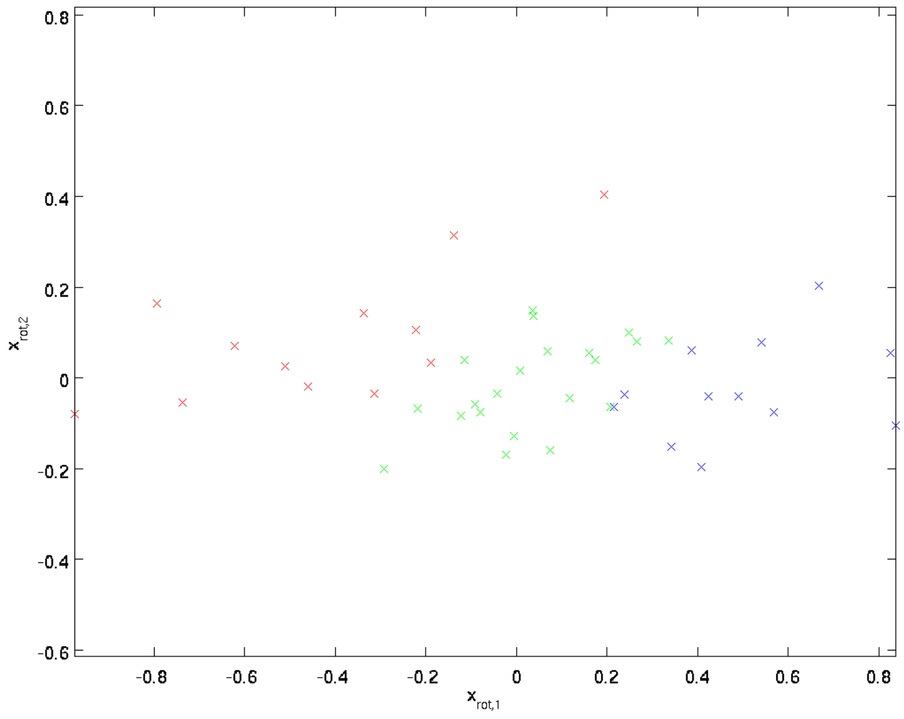


Figure 2:

This is the training set rotated into the u_1, u_2 basis. In the general case, $U^T x$ will be the training set rotated into the basis $u_1, u_2, \ldots, u_n$.

One of the properties of U is that it is an "orthogonal" matrix, which means that it satisfies $U^T U = UU^T = I$. So if you ever need to go from the rotated vectors x_{rot} back to the original data x, you can compute

$$x = Ux_{\text{rot}}, \tag{56}$$

because $Ux_{\text{rot}} = UU^T x = x$.

4.1.4 Reducing the Data Dimension

We see that the principal direction of variation of the data is the first dimension $x_{\text{rot},1}$ of this rotated data. Thus, if we want to reduce this data to one dimension, we can set

$$\tilde{x}^{(i)} = x^{(i)}_{\text{rot},1} = u_1^T x^{(i)} \in \Re. \tag{57}$$

More generally, if $x \in \Re^n$ and we want to reduce it to a k dimensional representation $\tilde{x} \in \Re^k$ (where $k < n$), we would take the first k components of x_{rot}, which correspond to the top k directions of variation.

Another way of explaining PCA is that x_{rot} is an n dimensional vector, where the first few components are likely to be large (e.g., in our example, we saw that $x^{(i)}_{\text{rot},1} = u_1^T x^{(i)}$ takes reasonably large values for most examples i), and the later components are likely to be small (e.g., in our example, $x^{(i)}_{\text{rot},2} = u_2^T x^{(i)}$ was more likely to be small).

What PCA does it it drops the the later (smaller) components of x_{rot}, and just approximates them with 0's. Concretely, our definition of $\tilde{x}$ can also be arrived at by using an approximation to x_{rot} where all but the first k components are zeros. In other words, we have:

$$\tilde{x} = \begin{bmatrix} x_{\text{rot},1} \\ \vdots \\ x_{\text{rot},k} \\ 0 \\ \vdots \\ 0 \end{bmatrix} \approx \begin{bmatrix} x_{\text{rot},1} \\ \vdots \\ x_{\text{rot},k} \\ x_{\text{rot},k+1} \\ \vdots \\ x_{\text{rot},n} \end{bmatrix} = x_{\text{rot}} \tag{58}$$

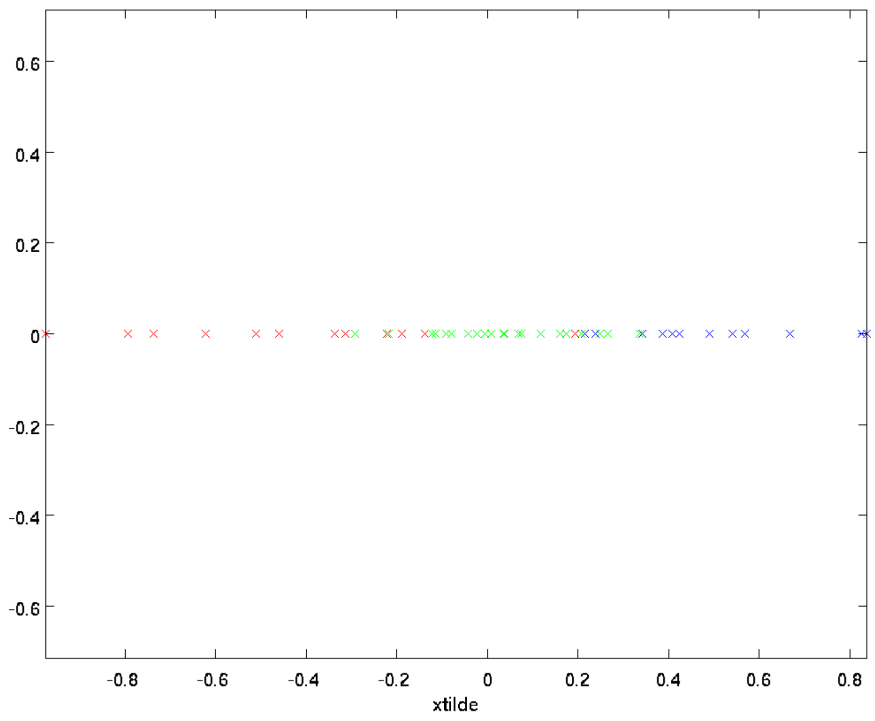
In our example, this gives us the following plot of $\tilde{x}$ (using $n = 2, k = 1$):

However, since the final $n - k$ components of $\tilde{x}$ as defined above would always be zero, there is no need to keep these zeros around, and so we define $\tilde{x}$ as a k-dimensional vector with just the first k (non-zero) components.

This also explains why we wanted to express our data in the $u_1, u_2, \ldots, u_n$ basis: Deciding which components to keep becomes just keeping the top k components. When we do this, we also say that we are "retaining the top k PCA (or principal) components."

4.1.5 Recovering an Approximation of the Data

Now, $\tilde{x} \in \Re^k$ is a lower-dimensional, "compressed" representation of the original $x \in \Re^n$. Given $\tilde{x}$, how can we recover an approximation $\hat{x}$ to the original value of x? From an earlier section(4.1.3),



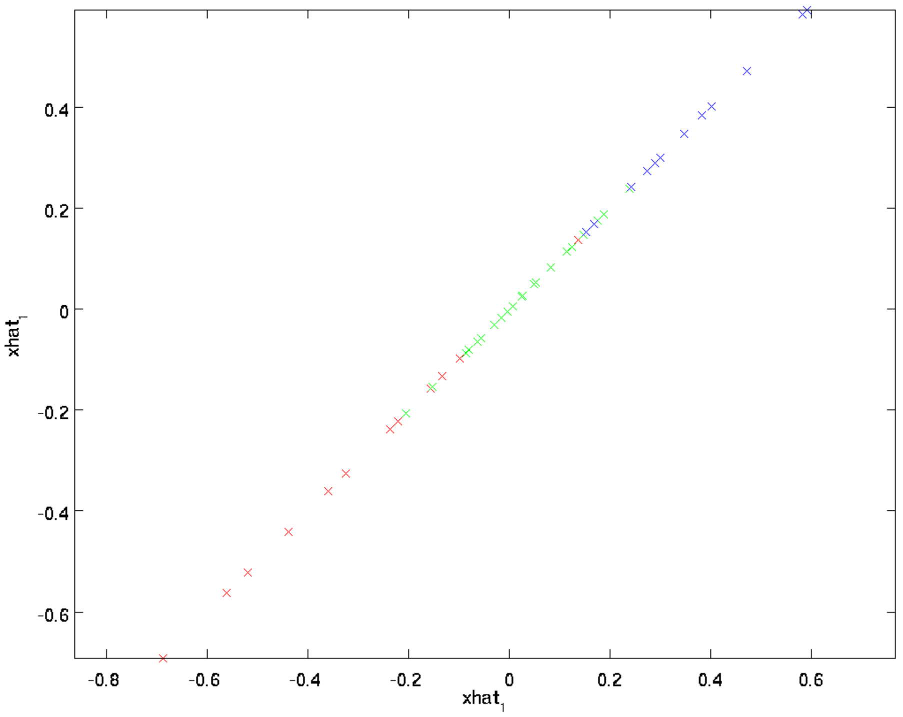
we know that $x = Ux_{\text{rot}}$. Further, we can think of $\tilde{x}$ as an approximation to x_{rot}, where we have set the last $n-k$ components to zeros. Thus, given $\tilde{x} \in \Re^k$, we can pad it out with $n-k$ zeros to get our approximation to $x_{\text{rot}} \in \Re^n$. Finally, we pre-multiply by U to get our approximation to x. Concretely, we get

$$\hat{x} = U \begin{bmatrix} \tilde{x}_1 \\ \vdots \\ \tilde{x}_k \\ 0 \\ \vdots \\ 0 \end{bmatrix} = \sum_{i=1}^{k} u_i \tilde{x}_i. \tag{59}$$

The final equality above comes from the definition of U given earlier(4.1.2). (In a practical implementation, we wouldn't actually zero pad $\tilde{x}$ and then multiply by U, since that would mean multiplying a lot of things by zeros; instead, we'd just multiply $\tilde{x} \in \Re^k$ with the first k columns of U as in the final expression above.) Applying this to our dataset, we get the following plot for $\hat{x}$:

We are thus using a 1 dimensional approximation to the original dataset.

If you are training an autoencoder or other unsupervised feature learning algorithm, the running time of your algorithm will depend on the dimension of the input. If you feed $\tilde{x} \in \Re^k$ into your learning algorithm instead of x, then you'll be training on a lower-dimensional input, and thus your algorithm might run significantly faster. For many datasets, the lower dimensional $\tilde{x}$ representation can be an extremely good approximation to the original, and using PCA this way can significantly speed up your algorithm while introducing very little approximation error.



4.1.6 Number of components to retain

How do we set k; i.e., how many PCA components should we retain? In our simple 2 dimensional example, it seemed natural to retain 1 out of the 2 components, but for higher dimensional data, this decision is less trivial. If k is too large, then we won't be compressing the data much; in the limit of $k = n$, then we're just using the original data (but rotated into a different basis). Conversely, if k is too small, then we might be using a very bad approximation to the data.

To decide how to set k, we will usually look at the percentage of variance retained for different values of k. Concretely, if $k = n$, then we have an exact approximation to the data, and we say that 100% of the variance is retained. I.e., all of the variation of the original data is retained. Conversely, if $k = 0$, then we are approximating all the data with the zero vector, and thus 0% of the variance is retained.

More generally, let $\lambda_1, \lambda_2, \ldots, \lambda_n$ be the eigenvalues of Σ (sorted in decreasing order), so that λ_j is the eigenvalue corresponding to the eigenvector u_j. Then if we retain k principal components, the percentage of variance retained is given by:

$$\frac{\sum_{j=1}^{k} \lambda_j}{\sum_{j=1}^{n} \lambda_j}. \tag{60}$$

In our simple 2D example above, $\lambda_1 = 7.29$, and $\lambda_2 = 0.69$. Thus, by keeping only $k = 1$ principal components, we retained $7.29/(7.29 + 0.69) = 0.913$, or 91.3% of the variance.

A more formal definition of percentage of variance retained is beyond the scope of these notes. However, it is possible to show that $\lambda_j = \sum_{i=1}^{m} x_{\text{rot},j}^2$. Thus, if $\lambda_j \approx 0$, that shows that $x_{\text{rot},j}$ is usually near 0 anyway, and we lose relatively little by approximating it with a constant 0. This also explains why we retain the top principal components (corresponding to the larger values of λ_j) instead of the bottom ones. The top principal components $x_{\text{rot},j}$ are the ones that're more variable and that take on larger values, and for which we would incur a greater approximation error if we were to set them to zero.

In the case of images, one common heuristic is to choose k so as to retain 99% of the variance. In other words, we pick the smallest value of k that satisfies

$$\frac{\sum_{j=1}^{k} \lambda_j}{\sum_{j=1}^{n} \lambda_j} \geq 0.99. \tag{61}$$

Depending on the application, if you are willing to incur some additional error, values in the 90-98% range are also sometimes used. When you describe to others how you applied PCA, saying that you chose k to retain 95% of the variance will also be a much more easily interpretable description than saying that you retained 120 (or whatever other number of) components.

4.1.7 PCA on Images

For PCA to work, usually we want each of the features $x_1, x_2, \ldots, x_n$ to have a similar range of values to the others (and to have a mean close to zero). If you've used PCA on other applications before, you may therefore have separately pre-processed each feature to have zero mean and unit variance, by separately estimating the mean and variance of each feature x_j. However, this isn't the pre-processing that we will apply to most types of images. Specifically, suppose we are training our algorithm on natural images, so that x_j is the value of pixel j. By "natural images," we informally mean the type of image that a typical animal or person might see over their lifetime.

Note: Usually we use images of outdoor scenes with grass, trees, etc., and cut out small (say 16×16) image patches randomly from these to train the algorithm. But in practice most feature learning algorithms are extremely robust to the exact type of image it is trained on, so most images taken with a normal camera, so long as they aren't excessively blurry or have strange artifacts, should work.

When training on natural images, it makes little sense to estimate a separate mean and variance for each pixel, because the statistics in one part of the image should (theoretically) be the same as any other. This property of images is called stationarity.

In detail, in order for PCA to work well, informally we require that (i) The features have approximately zero mean, and (ii) The different features have similar variances to each other. With natural images, (ii) is already satisfied even without variance normalization, and so we won't perform any variance normalization. (If you are training on audio data—say, on spectrograms—or on text data—say, bag-of-word vectors—we will usually not perform variance normalization either.) In fact, PCA is invariant to the scaling of the data, and will return the same eigenvectors regardless of the scaling of the input. More formally, if you multiply each feature vector x by some positive number (thus scaling every feature in every training example by the same number), PCA's output eigenvectors will not change.

So, we won't use variance normalization. The only normalization we need to perform then is mean normalization, to ensure that the features have a mean around zero. Depending on the application, very often we are not interested in how bright the overall input image is. For example, in object recognition tasks, the overall brightness of the image doesn't affect what objects there are in the image. More formally, we are not interested in the mean intensity value of an image patch; thus, we can subtract out this value, as a form of mean normalization.

Concretely, if $x^{(i)} \in \Re^n$ are the (grayscale) intensity values of a 16×16 image patch ($n = 256$), we might normalize the intensity of each image $x^{(i)}$ as follows:

$$\mu^{(i)} := \frac{1}{n}\sum_{j=1}^{n} x_j^{(i)}$$

$$x_j^{(i)} := x_j^{(i)} - \mu^{(i)}, \text{ forall } j$$

Note that the two steps above are done separately for each image $x^{(i)}$, and that $\mu^{(i)}$ here is the mean intensity of the image $x^{(i)}$. In particular, this is not the same thing as estimating a mean value separately for each pixel x_j.

If you are training your algorithm on images other than natural images (for example, images of handwritten characters, or images of single isolated objects centered against a white background), other types of normalization might be worth considering, and the best choice may be application dependent. But when training on natural images, using the per-image mean normalization method as given in the equations above would be a reasonable default.

4.1.8 References

http://cs229.stanford.edu/

4.2 Whitening

4.2.1 Introduction

We have used PCA to reduce the dimension of the data. There is a closely related preprocessing step called whitening (or, in some other literatures, sphering) which is needed for some algorithms. If we are training on images, the raw input is redundant, since adjacent pixel values are highly correlated. The goal of whitening is to make the input less redundant; more formally, our desiderata are that our learning algorithms sees a training input where (i) the features are less correlated with each other, and (ii) the features all have the same variance.

4.2.2 2D example

We will first describe whitening using our previous 2D example. We will then describe how this can be combined with smoothing, and finally how to combine this with PCA.

How can we make our input features uncorrelated with each other? We had already done this when computing $x_{\text{rot}}^{(i)} = U^T x^{(i)}$. Repeating our previous figure, our plot for x_{rot} was:

The covariance matrix of this data is given by:

$$\begin{bmatrix} 7.29 & 0 \\ 0 & 0.69 \end{bmatrix}. \qquad (62)$$

(Note: Technically, many of the statements in this section about the “covariance" will be true only if the data has zero mean. In the rest of this section, we will take this assumption as implicit in our statements. However, even if the data’s mean isn’t exactly zero, the intuitions we’re presenting here still hold true, and so this isn’t something that you should worry about.)

It is no accident that the diagonal values are λ_1 and λ_2. Further, the off-diagonal entries are zero; thus, $x_{\text{rot},1}$ and $x_{\text{rot},2}$ are uncorrelated, satisfying one of our desiderata for whitened data (that the features be less correlated).

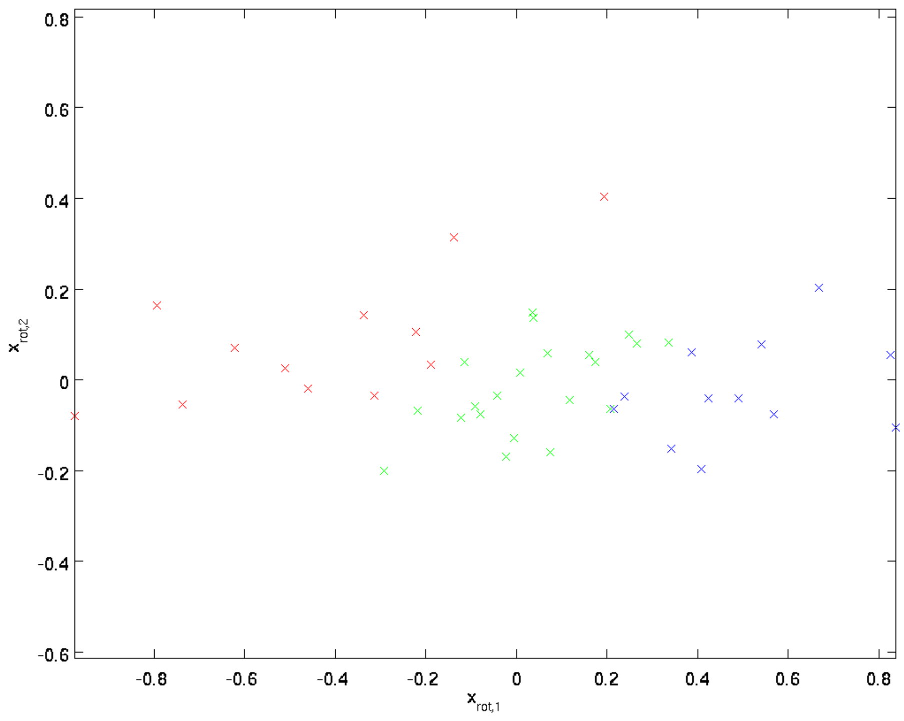


Figure 3:

To make each of our input features have unit variance, we can simply rescale each feature $x_{\text{rot},i}$ by $1/\sqrt{\lambda_i}$. Concretely, we define our whitened data $x_{\text{PCAwhite}} \in \Re^n$ as follows:

$$x_{\text{PCAwhite},i} = \frac{x_{\text{rot},i}}{\sqrt{\lambda_i}}. \tag{63}$$

Plotting x_{PCAwhite}, we get:

This data now has covariance equal to the identity matrix I. We say that x_{PCAwhite} is our PCA whitened version of the data: The different components of x_{PCAwhite} are uncorrelated and have unit variance.

Whitening combined with dimensionality reduction. If you want to have data that is whitened and which is lower dimensional than the original input, you can also optionally keep only the top k components of x_{PCAwhite}. When we combine PCA whitening with regularization (described later), the last few components of x_{PCAwhite} will be nearly zero anyway, and thus can safely be dropped.

4.2.3 ZCA Whitening

Finally, it turns out that this way of getting the data to have covariance identity I isn't unique. Concretely, if R is any orthogonal matrix, so that it satisfies $RR^T = R^TR = I$ (less formally, if R is a rotation/reflection matrix), then $R\,x_{\text{PCAwhite}}$ will also have identity covariance. In ZCA whitening, we choose $R = U$. We define

$$x_{\text{ZCAwhite}} = Ux_{\text{PCAwhite}} \tag{64}$$

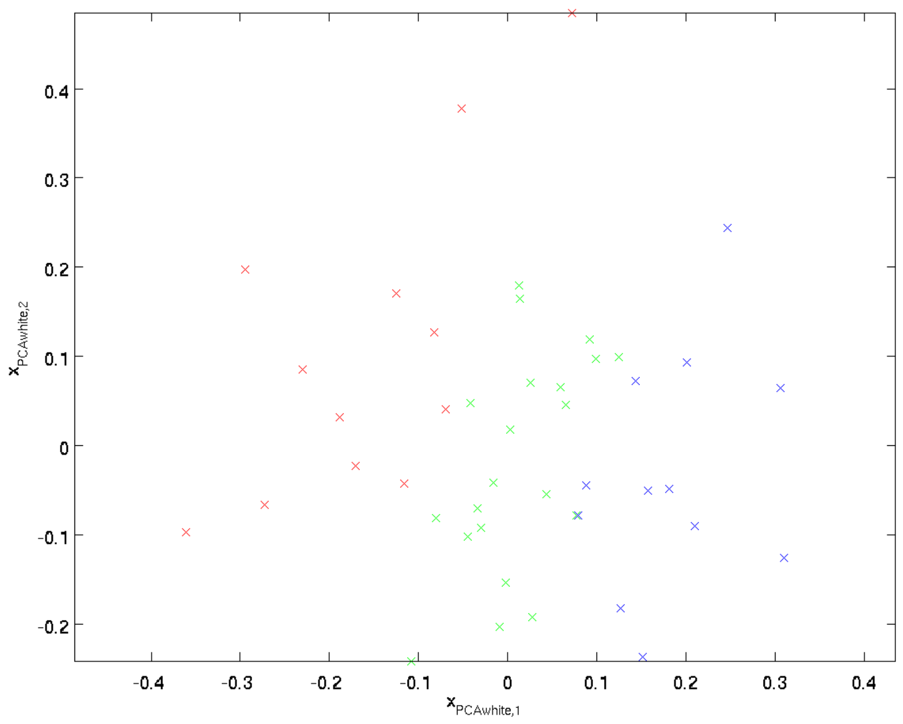


Figure 4:

Plotting x_{ZCAwhite}, we get:

It can be shown that out of all possible choices for R, this choice of rotation causes x_{ZCAwhite} to be as close as possible to the original input data x.

When using ZCA whitening (unlike PCA whitening), we usually keep all n dimensions of the data, and do not try to reduce its dimension.

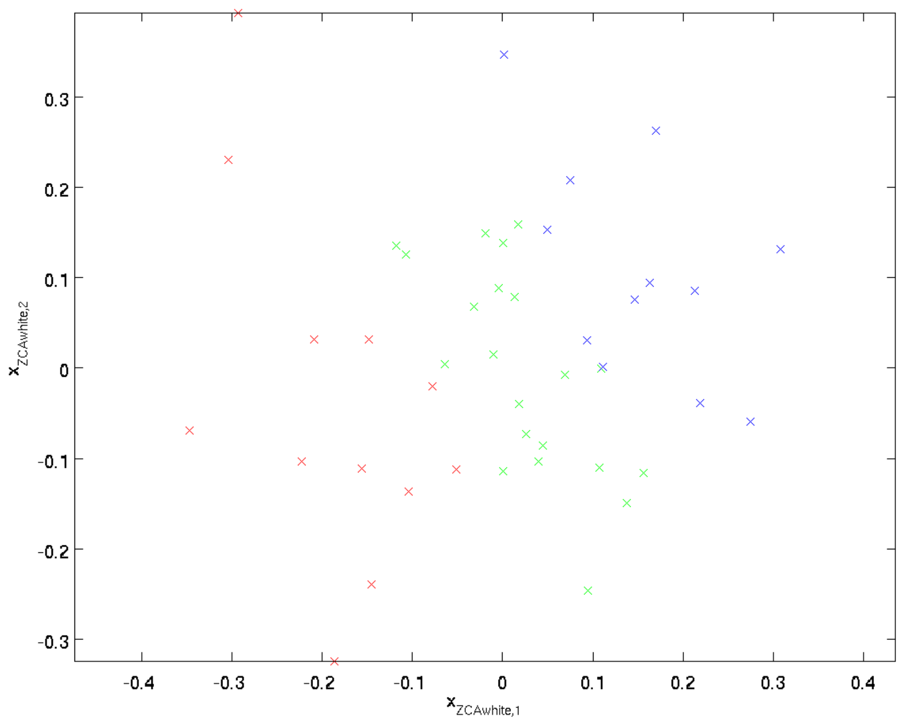
4.2.4 Regularizaton

When implementing PCA whitening or ZCA whitening in practice, sometimes some of the eigenvalues λ_i will be numerically close to 0, and thus the scaling step where we divide by $\sqrt{\lambda_i}$ would involve dividing by a value close to zero; this may cause the data to blow up (take on large values) or otherwise be numerically unstable. In practice, we therefore implement this scaling step using a small amount of regularization, and add a small constant ϵ to the eigenvalues before taking their square root and inverse:

$$x_{\text{PCAwhite},i} = \frac{x_{\text{rot},i}}{\sqrt{\lambda_i + \epsilon}}. \tag{65}$$

When x takes values around $[-1, 1]$, a value of $\epsilon \approx 10^{-5}$ might be typical.

For the case of images, adding ϵ here also has the effect of slightly smoothing (or low-pass filtering) the input image. This also has a desirable effect of removing aliasing artifacts caused by the way pixels are laid out in an image, and can improve the features learned (details are beyond the scope of these notes).



ZCA whitening is a form of pre-processing of the data that maps it from x to x_{ZCAwhite}. It turns out that this is also a rough model of how the biological eye (the retina) processes images. Specifically, as your eye perceives images, most adjacent "pixels" in your eye will perceive very similar values, since adjacent parts of an image tend to be highly correlated in intensity. It is thus wasteful for your eye to have to transmit every pixel separately (via your optic nerve) to your brain. Instead, your retina performs a decorrelation operation (this is done via retinal neurons that compute a function called "on center, off surround/off center, on surround") which is similar to that performed by ZCA. This results in a less redundant representation of the input image, which is then transmitted to your brain.

4.3 Implementing PCA/Whitening

In this section, we summarize the PCA, PCA whitening and ZCA whitening algorithms, and also describe how you can implement them using efficient linear algebra libraries.

First, we need to ensure that the data has (approximately) zero-mean. For natural images, we achieve this (approximately) by subtracting the mean value of each image patch.

We achieve this by computing the mean for each patch and subtracting it for each patch. In Matlab, we can do this by using

```
avg = mean(x, 1);     % Compute the mean pixel intensity value separately for each
    patch.
x = x - repmat(avg, size(x, 1), 1);
```

Next, we need to compute $\Sigma = \frac{1}{m}\sum_{i=1}^{m}(x^{(i)})(x^{(i)})^T$. If you're implementing this in Matlab (or even if you're implementing this in C++, Java, etc., but have access to an efficient linear algebra library), doing it as an explicit sum is inefficient. Instead, we can compute this in one fell swoop as

```
sigma = x * x' / size(x, 2);
```

(Check the math yourself for correctness.) Here, we assume that x is a data structure that contains one training example per column (so, x is a n-by-m matrix).

Next, PCA computes the eigenvectors of Σ. One could do this using the Matlab eig function. However, because Σ is a symmetric positive semi-definite matrix, it is more numerically reliable to do this using the svd function. Concretely, if you implement

```
[U,S,V] = svd(sigma);
```

then the matrix U will contain the eigenvectors of Sigma (one eigenvector per column, sorted in order from top to bottom eigenvector), and the diagonal entries of the matrix S will contain the corresponding eigenvalues (also sorted in decreasing order). The matrix V will be equal to transpose of U, and can be safely ignored.

(Note: The svd function actually computes the singular vectors and singular values of a matrix, which for the special case of a symmetric positive semi-definite matrix—which is all that we're concerned with here—is equal to its eigenvectors and eigenvalues. A full discussion of singular vectors vs. eigenvectors is beyond the scope of these notes.)

Finally, you can compute x_{rot} and $\tilde{x}$ as follows:

```
xRot = U' * x;          % rotated version of the data.
xTilde = U(:,1:k)' * x; % reduced dimension representation of the data,
                        % where k is the number of eigenvectors to keep
```

This gives your PCA representation of the data in terms of $\tilde{x} \in \Re^k$. Incidentally, if x is a n-by-m matrix containing all your training data, this is a vectorized implementation, and the expressions above work too for computing x_{rot} and $\tilde{x}$ for your entire training set all in one go. The resulting x_{rot} and $\tilde{x}$ will have one column corresponding to each training example.

To compute the PCA whitened data x_{PCAwhite}, use

```
xPCAwhite = diag(1./sqrt(diag(S) + epsilon)) * U' * x;
```

Since S's diagonal contains the eigenvalues λ_i, this turns out to be a compact way of computing $x_{\text{PCAwhite},i} = \frac{x_{\text{rot},i}}{\sqrt{\lambda_i}}$ simultaneously for all i.

Finally, you can also compute the ZCA whitened data x_{ZCAwhite} as:

```
xZCAwhite = U * diag(1./sqrt(diag(S) + epsilon)) * U' * x;
```

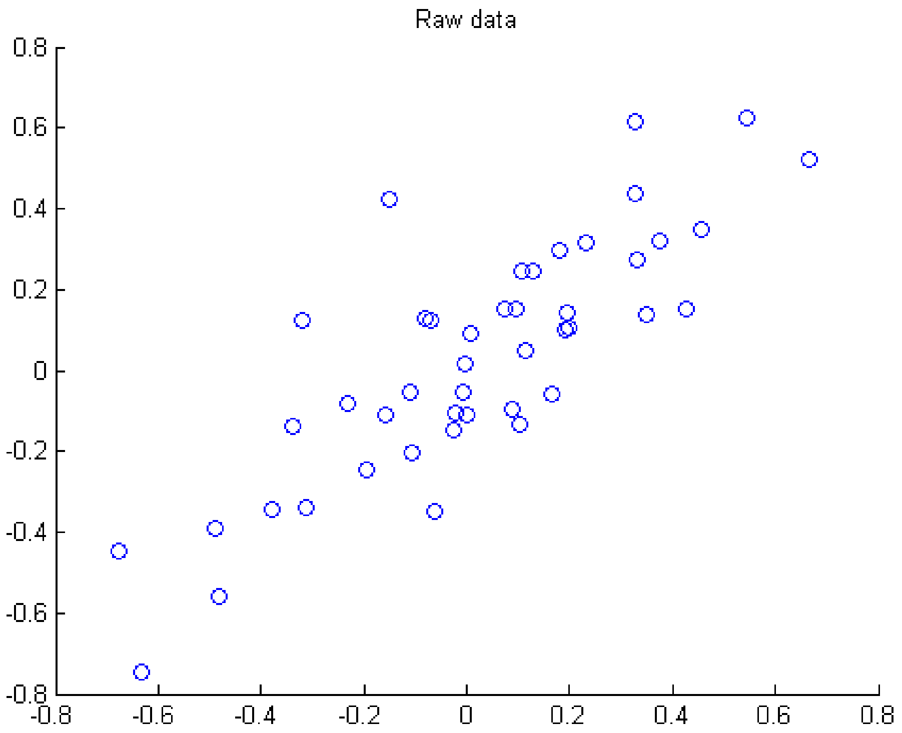
4.4 Exercise:PCA in 2D

PCA, PCA whitening and ZCA whitening in 2D

In this exercise you will implement PCA, PCA whitening and ZCA whitening, as described in the earlier sections of this tutorial, and generate the images shown in the earlier sections yourself. You will build on the starter code that has been provided at http://ufldl.stanford.edu/wiki/resources/pca_2d.zip. You need only write code at the places indicated by "YOUR CODE HERE" in the files. The only file you need to modify is pca_2d.m. Implementing this exercise will make the next exercise significantly easier to understand and complete.

4.4.1 Step 0: Load data

The starter code contains code to load 45 2D data points. When plotted using the scatter function, the results should look like the following:



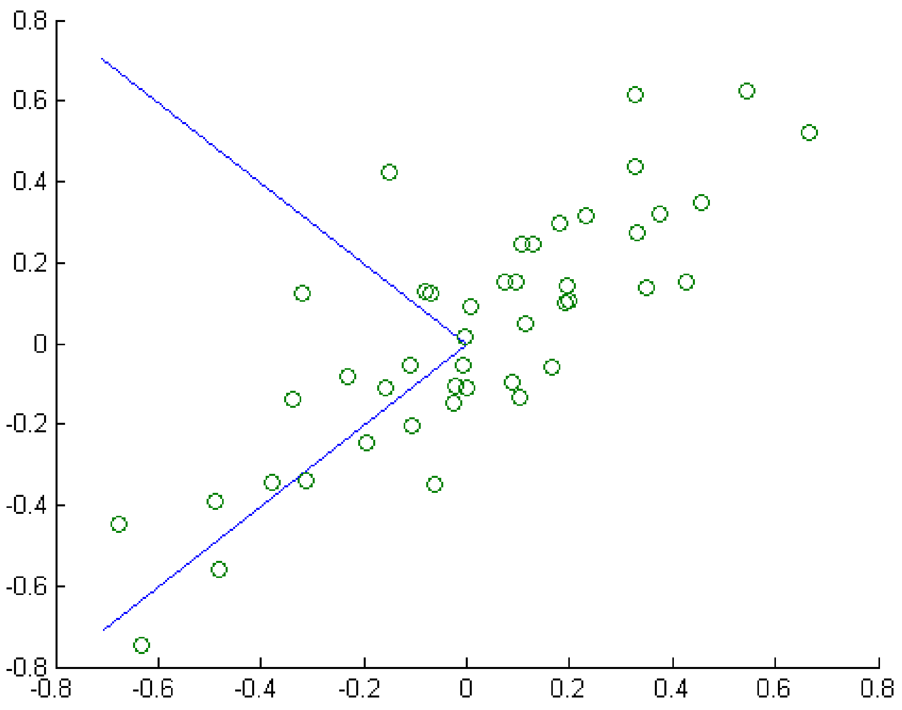
4.4.2 Step 1: Implement PCA

In this step, you will implement PCA to obtain xrot, the matrix in which the data is "rotated" to the basis comprising $u_1, \ldots, u_n$ made up of the principal components. As mentioned in the implementation notes, you should make use of MATLAB's svd function here.

4.4.3 Step 1a: Finding the PCA basis

Find u_1 and u_2, and draw two lines in your figure to show the resulting basis on top of the given data points. You may find it useful to use MATLAB's `hold on` and `hold off` functions. (After calling `hold on`, plotting functions such as `plot` will draw the new data on top of the previously existing figure rather than erasing and replacing it; and `hold off` turns this off.) You can use `plot([x1,x2], [y1,y2], '-')` to draw a line between `(x1,y1)` and `(x2,y2)`. Your figure should look like this:

If you are doing this in Matlab, you will probably get a plot that's identical to ours. However, eigenvectors are defined only up to a sign. I.e., instead of returning u_1 as the first eigenvector, Matlab/Octave could just as easily have returned $-u_1$, and similarly instead of u_2 Matlab/Octave could have returned $-u_2$. So if you wound up with one or both of the eigenvectors pointing in a direction opposite (180 degrees difference) from what's shown above, that's okay too.



4.4.4 Step 1b: Check xRot

Compute `xRot`, and use the `scatter` function to check that `xRot` looks as it should, which should be something like the following:

Because Matlab/Octave could have returned $-u_1$ and/or $-u_2$ instead of u_1 and u_2, it's also possible that you might have gotten a figure which is “flipped" or “reflected" along the x- and/or y-axis; a flipped/reflected version of this figure is also a completely correct result.

4.4.5 Step 2: Dimension reduce and replot

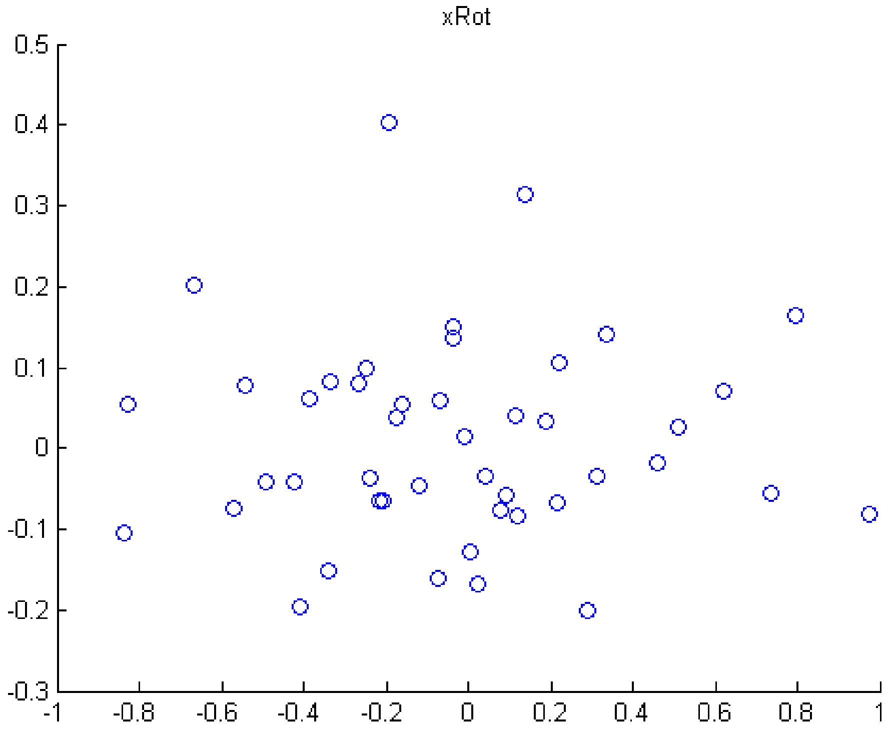
In the next step, set k, the number of components to retain, to be 1 (we have already done this for you). Compute the resulting xHat and plot the results. You should get the following (this figure should not be flipped along the x- or y-axis):

4.4.6 Step 3: PCA Whitening

Implement PCA whitening using the formula from the notes. Plot `xPCAWhite`, and verify that it looks like the following (a figure that is flipped/reflected on either/both axes is also correct):

4.4.7 Step 4: ZCA Whitening

Implement ZCA whitening and plot the results. The results should look like the following (this should not be flipped/reflected along the x- or y-axis):



4.5 Exercise:PCA and Whitening

PCA and Whitening on natural images

In this exercise, you will implement PCA, PCA whitening and ZCA whitening, and apply them to image patches taken from natural images.

You will build on the MATLAB starter code which we have provided in `http://ufldl.stanford.edu/wiki/resources/pca_exercise.zip`pca_exercise.zip. You need only write code at the places indicated by “YOUR CODE HERE" in the files. The only file you need to modify is `pca_gen.m`.

4.5.1 Step 0: Prepare data

Step 0a: Load data

The starter code contains code to load a set of natural images and sample 12×12 patches from them. The raw patches will look something like this:

These patches are stored as column vectors $x^{(i)} \in \mathbb{R}^{144}$ in the 144×10000 matrix x.

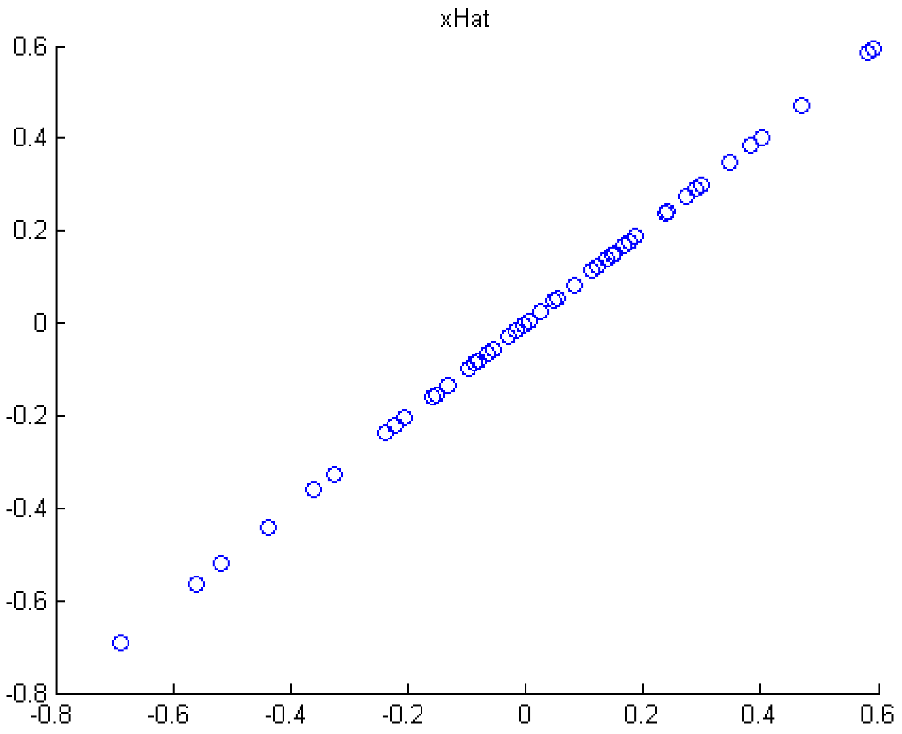
Step 0b: Zero mean the data

First, for each image patch, compute the mean pixel value and subtract it from that image, this centering the image around zero. You should compute a different mean value for each image patch.

4.5.2 Step 1: Implement PCA

Step 1a: Implement PCA

n this step, you will implement PCA to obtain x_{rot}, the matrix in which the data is “rotated" to the basis comprising the principal components (i.e. the eigenvectors of Σ). Note that in this



part of the exercise, you should not whiten the data.

Step 1b: Check covariance

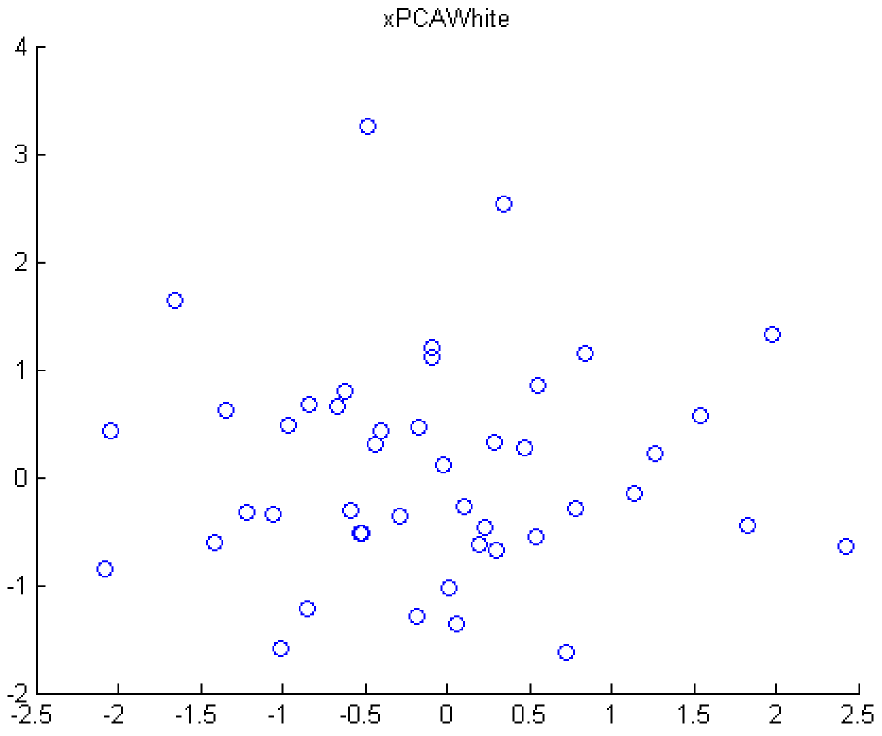
To verify that your implementation of PCA is correct, you should check the covariance matrix for the rotated data x_{rot}. PCA guarantees that the covariance matrix for the rotated data is a diagonal matrix (a matrix with non-zero entries only along the main diagonal). Implement code to compute the covariance matrix and verify this property. One way to do this is to compute the covariance matrix, and visualise it using the MATLAB command imagesc. The image should show a coloured diagonal line against a blue background. For this dataset, because of the range of the diagonal entries, the diagonal line may not be apparent, so you might get a figure like the one show below, but this trick of visualizing using imagesc will come in handy later in this exercise.

4.5.3 Step 2: Find number of components to retain

Next, choose k, the number of principal components to retain. Pick k to be as small as possible, but so that at least 99% of the variance is retained. In the step after this, you will discard all but the top k principal components, reducing the dimension of the original data to k.

4.5.4 Step 3: PCA with dimension reduction

Now that you have found k, compute $\tilde{x}$, the reduced-dimension representation of the data. This gives you a representation of each image patch as a k dimensional vector instead of a 144 dimensional vector. If you are training a sparse autoencoder or other algorithm on this reduced-dimensional data, it will run faster than if you were training on the original 144 dimensional data.



To see the effect of dimension reduction, go back from $\tilde{x}$ to produce the matrix $\hat{x}$, the dimension-reduced data but expressed in the original 144 dimensional space of image patches. Visualise $\hat{x}$ and compare it to the raw data, x. You will observe that there is little loss due to throwing away the principal components that correspond to dimensions with low variation. For comparison, you may also wish to generate and visualise $\hat{x}$ for when only 90% of the variance is retained.

4.5.5 Step 4: PCA with whitening and regularization

Step 4a: Implement PCA with whitening and regularization

Now implement PCA with whitening and regularization to produce the matrix `xPCAWhite`. Use the following parameter value:

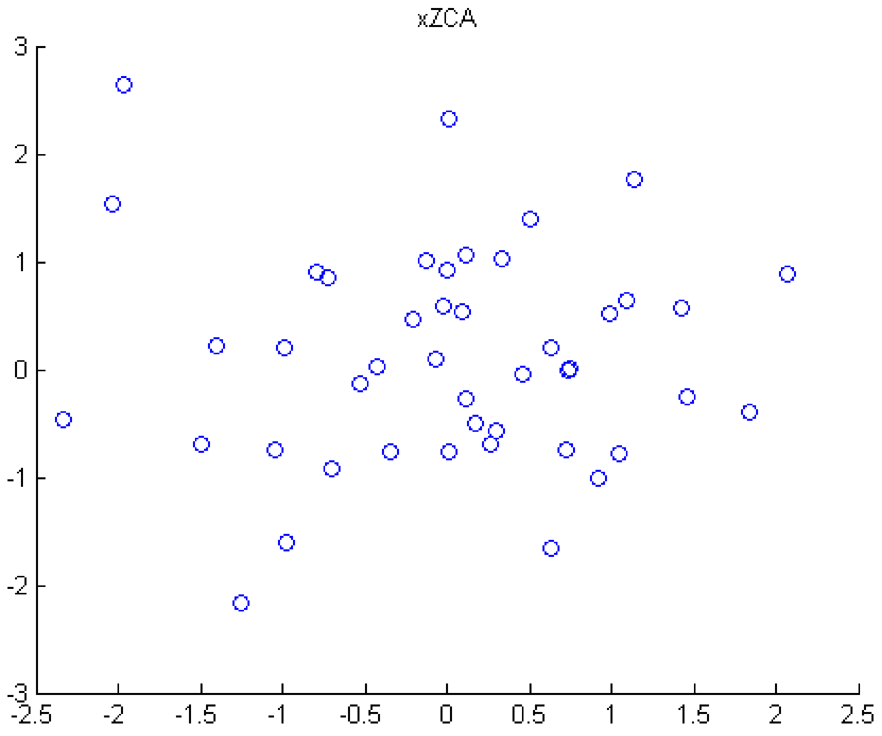
```
epsilon = 0.1
```

Step 4b: Check covariance

Similar to using PCA alone, PCA with whitening also results in processed data that has a diagonal covariance matrix. However, unlike PCA alone, whitening additionally ensures that the diagonal entries are equal to 1, i.e. that the covariance matrix is the identity matrix.

That would be the case if you were doing whitening alone with no regularization. However, in this case you are whitening with regularization, to avoid numerical/etc. problems associated with small eigenvalues. As a result of this, some of the diagonal entries of the covariance of your `xPCAwhite` will be smaller than 1.

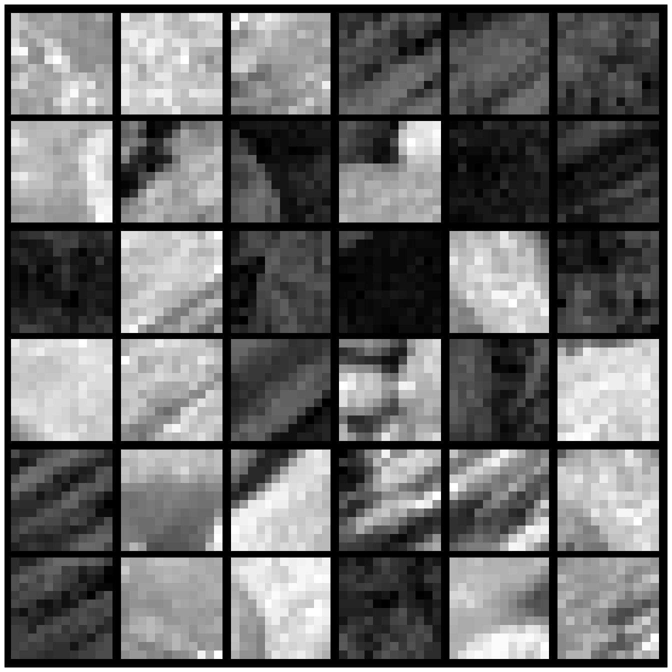
To verify that your implementation of PCA whitening with and without regularization is correct, you can check these properties. Implement code to compute the covariance matrix and verify this property. (To check the result of PCA without whitening, simply set epsilon to 0, or close to 0, say 1e-10). As earlier, you can visualise the covariance matrix with imagesc. When

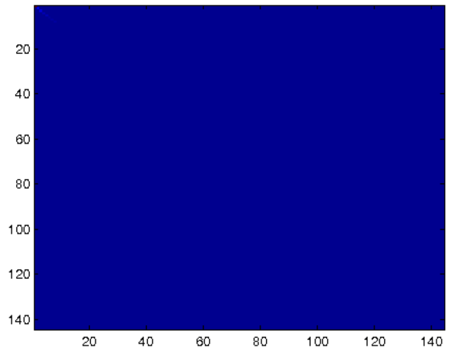


visualised as an image, for PCA whitening without regularization you should see a red line across the diagonal (corresponding to the one entries) against a blue background (corresponding to the zero entries); for PCA whitening with regularization you should see a red line that slowly turns blue across the diagonal (corresponding to the 1 entries slowly becoming smaller).

4.5.6 Step 5: ZCA whitening

Now implement ZCA whitening to produce the matrix `xZCAWhite`. Visualize `xZCAWhite` and compare it to the raw data, x. You should observe that whitening results in, among other things, enhanced edges. Try repeating this with epsilon set to 1, 0.1, and 0.01, and see what you obtain. The example shown below (left image) was obtained with epsilon $= 0.1$.




20
40
60
80
100
120
140
20
40
60
80
100
120
140

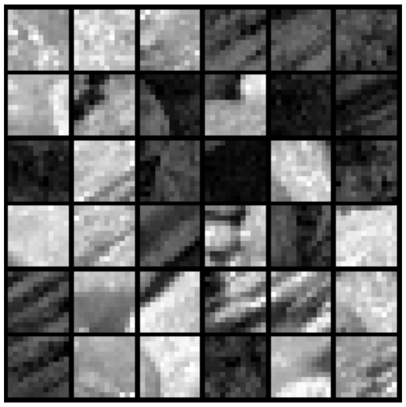


Figure 5: Raw images

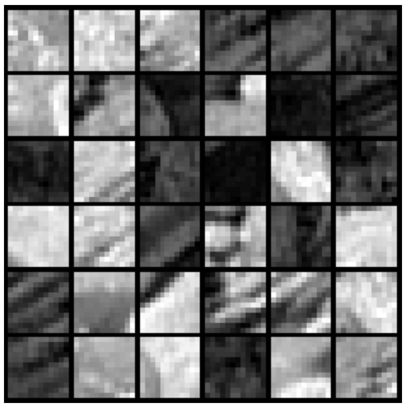


Figure 6: PCA dimension-reduced images (99% variance)

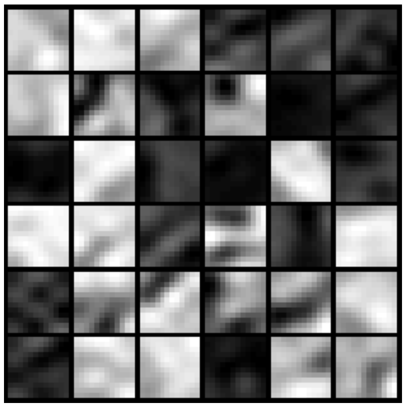


Figure 7: PCA dimension-reduced images (90% variance)

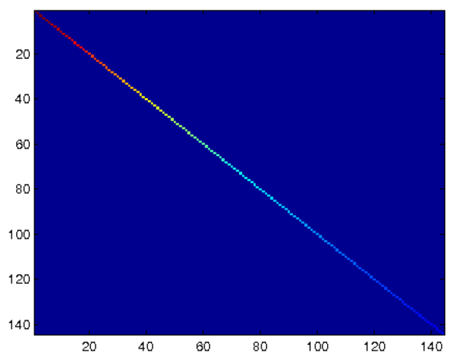


Figure 8: Covariance for PCA whitening with regularization

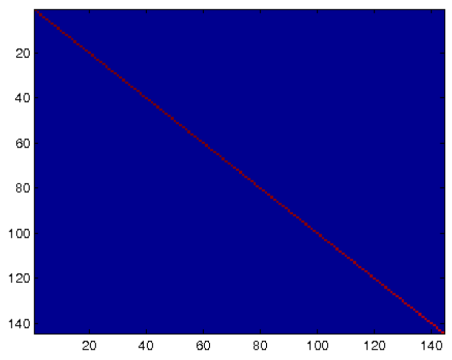


Figure 9: Covariance for PCA whitening without regularization

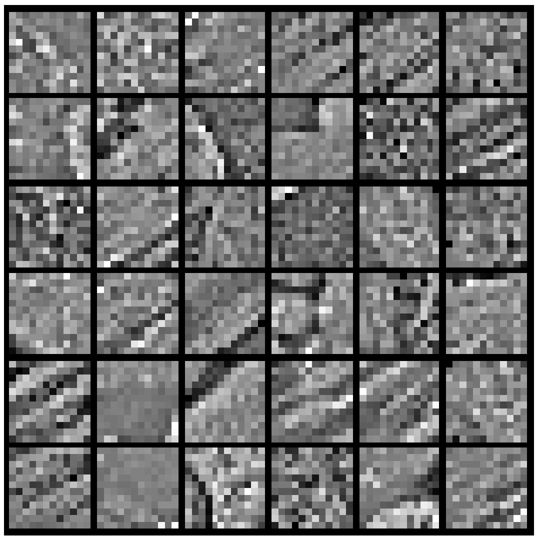


Figure 10: ZCA whitened images

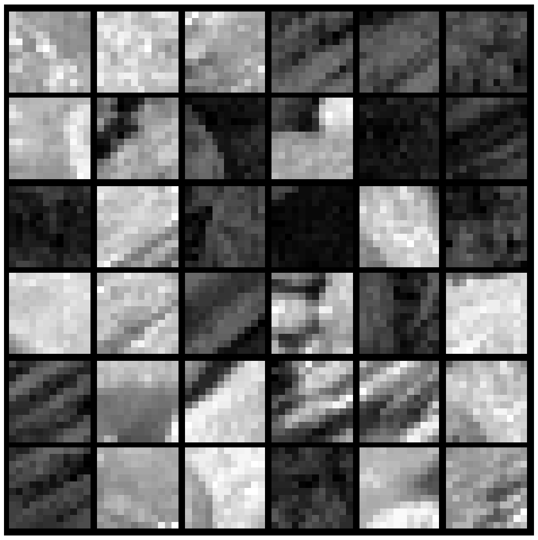


Figure 11: Raw images

5 Softmax Regression

5.1 Softmax Regression

5.1.1 Introduction

In these notes, we describe the Softmax regression model. This model generalizes logistic regression to classification problems where the class label y can take on more than two possible values. This will be useful for such problems as MNIST digit classification, where the goal is to distinguish between 10 different numerical digits. Softmax regression is a supervised learning algorithm, but we will later be using it in conjuction with our deep learning/unsupervised feature learning methods.

Recall that in logistic regression, we had a training set $\{(x^{(1)}, y^{(1)}), \ldots, (x^{(m)}, y^{(m)})\}$ of m labeled examples, where the input features are $x^{(i)} \in \Re^{n+1}$. (In this set of notes, we will use the notational convention of letting the feature vectors x be $n+1$ dimensional, with $x_0 = 1$ corresponding to the intercept term.) With logistic regression, we were in the binary classification setting, so the labels were $y^{(i)} \in \{0, 1\}$. Our hypothesis took the form:

$$h_\theta(x) = \frac{1}{1 + \exp(-\theta^T x)}, \tag{66}$$

and the model parameters θ were trained to minimize the cost function

$$J(\theta) = -\frac{1}{m}\left[\sum_{i=1}^{m} y^{(i)} \log h_\theta(x^{(i)}) + (1 - y^{(i)}) \log(1 - h_\theta(x^{(i)}))\right] \tag{67}$$

In the softmax regression setting, we are interested in multi-class classification (as opposed to only binary classification), and so the label y can take on k different values, rather than only two. Thus, in our training set $\{(x^{(1)}, y^{(1)}), \ldots, (x^{(m)}, y^{(m)})\}$, we now have that $y^{(i)} \in \{1, 2, \ldots, k\}$. (Note that our convention will be to index the classes starting from 1, rather than from 0.) For example, in the MNIST digit recognition task, we would have $k = 10$ different classes.

Given a test input x, we want our hypothesis to estimate the probability that $p(y = j|x)$ for each value of $j = 1, \ldots, k$. I.e., we want to estimate the probability of the class label taking on each of the k different possible values. Thus, our hypothesis will output a k dimensional vector (whose elements sum to 1) giving us our k estimated probabilities. Concretely, our hypothesis $h_\theta(x)$ takes the form:

$$h_\theta(x^{(i)}) = \begin{bmatrix} p(y^{(i)} = 1|x^{(i)};\theta) \\ p(y^{(i)} = 2|x^{(i)};\theta) \\ \vdots \\ p(y^{(i)} = k|x^{(i)};\theta) \end{bmatrix} = \frac{1}{\sum_{j=1}^{k} e^{\theta_j^T x^{(i)}}} \begin{bmatrix} e^{\theta_1^T x^{(i)}} \\ e^{\theta_2^T x^{(i)}} \\ \vdots \\ e^{\theta_k^T x^{(i)}} \end{bmatrix} \tag{68}$$

Here $\theta_1, \theta_2, \ldots, \theta_k \in \Re^{n+1}$ are the parameters of our model. Notice that the term $\frac{1}{\sum_{j=1}^{k} e^{\theta_j^T x^{(i)}}}$ normalizes the distribution, so that it sums to one.

For convenience, we will also write θ to denote all the parameters of our model. When you implement softmax regression, it is usually convenient to represent θ as a k-by-$(n+1)$ matrix obtained by stacking up $\theta_1, \theta_2, \ldots, \theta_k$ in rows, so that

$$\theta = \begin{bmatrix} —\theta_1^T— \\ —\theta_2^T— \\ \vdots \\ —\theta_k^T— \end{bmatrix}$$

5.1.2 Cost Function

We now describe the cost function that we'll use for softmax regression. In the equation below, $1\{\cdot\}$ is the indicator function, so that $1\{$`a true statement`$\} = 1$, and $1\{$`a false statement`$\} = 0$. For example, $1\{2+2 = 4\}$ evaluates to 1; whereas $1\{1+1 = 5\}$ evaluates to 0. Our cost function will be:

$$J(\theta) = -\frac{1}{m}\left[\sum_{i=1}^{m}\sum_{j=1}^{k} 1\left\{y^{(i)} = j\right\} \log \frac{e^{\theta_j^T x^{(i)}}}{\sum_{l=1}^{k} e^{\theta_l^T x^{(i)}}}\right] \tag{69}$$

Notice that this generalizes the logistic regression cost function, which could also have been written:

$$\begin{aligned} J(\theta) &= -\frac{1}{m}\left[\sum_{i=1}^{m}(1-y^{(i)})\log(1-h_\theta(x^{(i)})) + y^{(i)}\log h_\theta(x^{(i)})\right] & (70) \\ &= -\frac{1}{m}\left[\sum_{i=1}^{m}\sum_{j=0}^{1} 1\left\{y^{(i)} = j\right\}\log p(y^{(i)} = j|x^{(i)};\theta)\right] & (71) \end{aligned}$$

The softmax cost function is similar, except that we now sum over the k different possible values of the class label. Note also that in softmax regression, we have that $p(y^{(i)} = j|x^{(i)};\theta) = \frac{e^{\theta_j^T x^{(i)}}}{\sum_{l=1}^{k} e^{\theta_l^T x^{(i)}}}$
.

There is no known closed-form way to solve for the minimum of $J(\theta)$, and thus as usual we'll resort to an iterative optimization algorithm such as gradient descent or L-BFGS. Taking derivatives, one can show that the gradient is:

$$\nabla_{\theta_j} J(\theta) = -\frac{1}{m}\sum_{i=1}^{m}\left[x^{(i)}\left(1\{y^{(i)} = j\} - p(y^{(i)} = j|x^{(i)};\theta)\right)\right] \tag{72}$$

Recall the meaning of the "∇_{θ_j}" notation. In particular, $\nabla_{\theta_j} J(\theta)$ is itself a vector, so that its l-th element is $\frac{\partial J(\theta)}{\partial \theta_{jl}}$ the partial derivative of $J(\theta)$ with respect to the l-th element of θ_j.

Armed with this formula for the derivative, one can then plug it into an algorithm such as gradient descent, and have it minimize $J(\theta)$. For example, with the standard implementation of gradient descent, on each iteration we would perform the update $\theta_j := \theta_j - \alpha\nabla_{\theta_j} J(\theta)$ (for each $j = 1, \ldots, k$).

When implementing softmax regression, we will typically use a modified version of the cost function described above; specifically, one that incorporates weight decay. We describe the motivation and details below.

5.1.3 Properties of softmax regression parameterization

Softmax regression has an unusual property that it has a "redundant" set of parameters. To explain what this means, suppose we take each of our parameter vectors θ_j, and subtract some fixed vector ψ from it, so that every θ_j is now replaced with $\theta_j - \psi$ (for every $j = 1, \dots, k$). Our hypothesis now estimates the class label probabilities as

$$p(y^{(i)} = j | x^{(i)}; \theta) = \frac{e^{(\theta_j - \psi)^T x^{(i)}}}{\sum_{l=1}^{k} e^{(\theta_l - \psi)^T x^{(i)}}} \tag{73}$$

$$= \frac{e^{\theta_j^T x^{(i)}} e^{-\psi^T x^{(i)}}}{\sum_{l=1}^{k} e^{\theta_l^T x^{(i)}} e^{-\psi^T x^{(i)}}} \tag{74}$$

$$= \frac{e^{\theta_j^T x^{(i)}}}{\sum_{l=1}^{k} e^{\theta_l^T x^{(i)}}}. \tag{75}$$

In other words, subtracting ψ from every θ_j does not affect our hypothesis' predictions at all! This shows that softmax regression's parameters are "redundant." More formally, we say that our softmax model is overparameterized, meaning that for any hypothesis we might fit to the data, there are multiple parameter settings that give rise to exactly the same hypothesis function h_θ mapping from inputs x to the predictions.

Further, if the cost function $J(\theta)$ is minimized by some setting of the parameters $(\theta_1, \theta_2, \dots, \theta_k)$, then it is also minimized by $(\theta_1 - \psi, \theta_2 - \psi, \dots, \theta_k - \psi)$ for any value of ψ. Thus, the minimizer of $J(\theta)$ is not unique. (Interestingly, $J(\theta)$ is still convex, and thus gradient descent will not run into a local optima problems. But the Hessian is singular/non-invertible, which causes a straightforward implementation of Newton's method to run into numerical problems.)

Notice also that by setting $\psi = \theta_1$, one can always replace θ_1 with $\theta_1 - \psi = \vec{0}$ (the vector of all 0's), without affecting the hypothesis. Thus, one could "eliminate" the vector of parameters θ_1 (or any other θ_j, for any single value of j), without harming the representational power of our hypothesis. Indeed, rather than optimizing over the $k(n+1)$ parameters $(\theta_1, \theta_2, \dots, \theta_k)$ (where $\theta_j \in \Re^{n+1}$), one could instead set $\theta_1 = \vec{0}$ and optimize only with respect to the $(k-1)(n+1)$ remaining parameters, and this would work fine.

In practice, however, it is often cleaner and simpler to implement the version which keeps all the parameters $(\theta_1, \theta_2, \dots, \theta_n)$, without arbitrarily setting one of them to zero. But we will make one change to the cost function: Adding weight decay. This will take care of the numerical problems associated with softmax regression's overparameterized representation.

5.1.4 Weight Decay

We will modify the cost function by adding a weight decay term $\frac{\lambda}{2} \sum_{i=1}^{k} \sum_{j=0}^{n} \theta_{ij}^2$ which penalizes large values of the parameters. Our cost function is now

$$J(\theta) = -\frac{1}{m} \left[\sum_{i=1}^{m} \sum_{j=1}^{k} 1\left\{y^{(i)} = j\right\} \log \frac{e^{\theta_j^T x^{(i)}}}{\sum_{l=1}^{k} e^{\theta_l^T x^{(i)}}} \right] + \frac{\lambda}{2} \sum_{i=1}^{k} \sum_{j=0}^{n} \theta_{ij}^2 \tag{76}$$

With this weight decay term (for any $\lambda > 0$), the cost function $J(\theta)$ is now strictly convex, and is guaranteed to have a unique solution. The Hessian is now invertible, and because $J(\theta)$ is convex, algorithms such as gradient descent, L-BFGS, etc. are guaranteed to converge to the global minimum.

To apply an optimization algorithm, we also need the derivative of this new definition of $J(\theta)$. One can show that the derivative is:

$$\nabla_{\theta_j} J(\theta) = -\frac{1}{m} \sum_{i=1}^{m} \left[x^{(i)} (1\{y^{(i)} = j\} - p(y^{(i)} = j | x^{(i)}; \theta)) \right] + \lambda \theta_j \tag{77}$$

By minimizing $J(\theta)$ with respect to θ, we will have a working implementation of softmax regression.

5.1.5 Relationship to Logistic Regression

In the special case where $k = 2$, one can show that softmax regression reduces to logistic regression. This shows that softmax regression is a generalization of logistic regression. Concretely, when $k = 2$, the softmax regression hypothesis outputs

$$h_\theta(x) = \frac{1}{e^{\theta_1^T x} + e^{\theta_2^T x^{(i)}}} \begin{bmatrix} e^{\theta_1^T x} \\ e^{\theta_2^T x} \end{bmatrix} \tag{78}$$

Taking advantage of the fact that this hypothesis is overparameterized and setting $\psi = \theta_1$, we can subtract θ_1 from each of the two parameters, giving us

$$h(x) = \frac{1}{e^{\vec{0}^T x} + e^{(\theta_2 - \theta_1)^T x^{(i)}}} \begin{bmatrix} e^{\vec{0}^T x} \\ e^{(\theta_2 - \theta_1)^T x} \end{bmatrix} \tag{79}$$

$$= \begin{bmatrix} \frac{1}{1 + e^{(\theta_2 - \theta_1)^T x^{(i)}}} \\ \frac{e^{(\theta_2 - \theta_1)^T x}}{1 + e^{(\theta_2 - \theta_1)^T x^{(i)}}} \end{bmatrix} \tag{80}$$

$$= \begin{bmatrix} \frac{1}{1 + e^{(\theta_2 - \theta_1)^T x^{(i)}}} \\ 1 - \frac{1}{1 + e^{(\theta_2 - \theta_1)^T x^{(i)}}} \end{bmatrix} \tag{81}$$

Thus, replacing $\theta_2 - \theta_1$ with a single parameter vector θ', we find that softmax regression predicts the probability of one of the classes as $\frac{1}{1 + e^{(\theta')^T x^{(i)}}}$, and that of the other class as $1 - \frac{1}{1 + e^{(\theta')^T x^{(i)}}}$, same as logistic regression.

5.1.6 Softmax Regression vs. k Binary Classifiers

Suppose you are working on a music classification application, and there are k types of music that you are trying to recognize. Should you use a softmax classifier, or should you build k separate binary classifiers using logistic regression?

This will depend on whether the four classes are mutually exclusive. For example, if your four classes are classical, country, rock, and jazz, then assuming each of your training examples is labeled with exactly one of these four class labels, you should build a softmax classifier with $k = 4$. (If there're also some examples that are none of the above four classes, then you can set $k = 5$ in softmax regression, and also have a fifth, "none of the above," class.)

If however your categories are `has_vocals`, dance, soundtrack, pop, then the classes are not mutually exclusive; for example, there can be a piece of pop music that comes from a soundtrack and in addition has vocals. In this case, it would be more appropriate to build 4 binary logistic regression classifiers. This way, for each new musical piece, your algorithm can separately decide whether it falls into each of the four categories.

Now, consider a computer vision example, where you're trying to classify images into three different classes. (i) Suppose that your classes are `indoor_scene, outdoor_urban_scene`, and `outdoor_wilderness_scene`. Would you use sofmax regression or three logistic regression classifiers? (ii) Now suppose your classes are indoor_scene, black_and_white_image, and image_has_people. Would you use softmax regression or multiple logistic regression classifiers?

In the first case, the classes are mutually exclusive, so a softmax regression classifier would be appropriate. In the second case, it would be more appropriate to build three separate logistic regression classifiers.

5.2 Exercise:Softmax Regression

In this problem set, you will use softmax regression(5) to classify MNIST images. The goal of this exercise is to build a softmax classifier that you will be able to reuse in the future exercises and also on other classification problems that you might encounter.

In the file `http://ufldl.stanford.edu/wiki/resources/softmax_exercise.zip`,we have provided some starter code. You should write your code in the places indicated by "YOUR CODE HERE" in the files.

In the starter code, you will need to modify `softmaxCost.m` and `softmaxPredict.m` for this exercise.

We have also provided `softmaxExercise.m` that will help walk you through the steps in this exercise.

5.2.1 Dependencies

The following additional files are required for this exercise:

- MNIST Dataset `http://yann.lecun.com/exdb/mnist/`
- Support functions for loading MNIST in Matlab (A)
- Starter Code `http://ufldl.stanford.edu/wiki/resources/softmax_exercise.zip`

You will also need:

- `computeNumericalGradient.m` from Exercise:Sparse Autoencoder (2.7)

If you have not completed the exercises listed above, we strongly suggest you complete them first.

5.2.2 Step 0: Initialize constants and parameters

We've provided the code for this step in `softmaxExercise.m`.

Two constants, inputSize and numClasses, corresponding to the size of each input vector and the number of class labels have been defined in the starter code. This will allow you to reuse your code on a different data set in a later exercise. We also initialize lambda, the weight decay parameter here.

5.2.3 Step 1: Load data

The starter code loads the MNIST images and labels into inputData and labels respectively. The images are pre-processed to scale the pixel values to the range $[0, 1]$, and the label 0 is remapped to 10 for convenience of implementation, so that the labels take values in $\{1, 2, \ldots, 10\}$. You will not need to change any code in this step for this exercise, but note that your code should be general enough to operate on data of arbitrary size belonging to any number of classes.

5.2.4 Step 2: Implement softmaxCost

In softmaxCost.m, implement code to compute the softmax cost function $J(\theta)$. Remember to include the weight decay term in the cost as well. Your code should also compute the appropriate gradients, as well as the predictions for the input data (which will be used in the cross-validation step later).

It is important to vectorize your code so that it runs quickly. We also provide several implementation tips below:

Note: In the provided starter code, `theta` is a matrix where each the jth row is θ_j^T

Implementation Tip: Computing the ground truth matrix - In your code, you may need to compute the ground truth matrix `M`, such that `M(r, c)` is 1 if $y^{(c)} = r$ and 0 otherwise. This can be done quickly, without a loop, using the MATLAB functions `sparse` and `full`. Specifically, the command `M = sparse(r, c, v)` creates a sparse matrix such that `M(r(i), c(i)) = v(i)` for all `i`. That is, the vectors `r` and `c` give the position of the elements whose values we wish to set, and `v` the corresponding values of the elements. Running full on a sparse matrix gives a "full" representation of the matrix for use (meaning that Matlab will no longer try to represent it as a sparse matrix in memory). The code for using `sparse` and `full` to compute the ground truth matrix has already been included in `softmaxCost.m`.

Implementation Tip: Preventing overflows - in softmax regression, you will have to compute the hypothesis

$$h(x^{(i)}) = \frac{1}{\sum_{j=1}^{k} e^{\theta_j^T x^{(i)}}} \begin{bmatrix} e^{\theta_1^T x^{(i)}} \\ e^{\theta_2^T x^{(i)}} \\ \vdots \\ e^{\theta_k^T x^{(i)}} \end{bmatrix} \tag{82}$$

When the products $\theta_i^T x^{(i)}$ are large, the exponential function $e^{\theta_i^T x^{(i)}}$ will become very large and possibly overflow. When this happens, you will not be able to compute your hypothesis. However, there is an easy solution - observe that we can multiply the top and bottom of the hypothesis by some constant without changing the output:

$$h(x^{(i)}) = \frac{1}{\sum_{j=1}^{k} e^{\theta_j^T x^{(i)}}} \begin{bmatrix} e^{\theta_1^T x^{(i)}} \\ e^{\theta_2^T x^{(i)}} \\ \vdots \\ e^{\theta_k^T x^{(i)}} \end{bmatrix} \tag{83}$$

$$= \frac{e^{-\alpha}}{e^{-\alpha}\sum_{j=1}^{k} e^{\theta_j^T x^{(i)}}} \begin{bmatrix} e^{\theta_1^T x^{(i)}} \\ e^{\theta_2^T x^{(i)}} \\ \vdots \\ e^{\theta_k^T x^{(i)}} \end{bmatrix} \tag{84}$$

$$= \frac{1}{\sum_{j=1}^{k} e^{\theta_j^T x^{(i)}-\alpha}} \begin{bmatrix} e^{\theta_1^T x^{(i)}-\alpha} \\ e^{\theta_2^T x^{(i)}-\alpha} \\ \vdots \\ e^{\theta_k^T x^{(i)}-\alpha} \end{bmatrix} \tag{85}$$

$$\tag{86}$$

Hence, to prevent overflow, simply subtract some large constant value from each of the $\theta_j^T x^{(i)}$ terms before computing the exponential. In practice, for each example, you can use the maximum of the $\theta_j^T x^{(i)}$ terms as the constant. Assuming you have a matrix `M` containing these terms such that `M(r, c)` is $\theta_r^T x^{(c)}$, then you can use the following code to accomplish this:

```
% M is the matrix as described in the text
M = bsxfun(@minus, M, max(M, [], 1));
```

`max(M)` yields a row vector with each element giving the maximum value in that column. `bsxfun` (short for binary singleton expansion function) applies minus along each row of `M`, hence subtracting the maximum of each column from every element in the column.

Implementation Tip: Computing the predictions - you may also find `bsxfun` useful in computing your predictions - if you have a matrix `M` containing the $e^{\theta_j^T x^{(i)}}$ terms, such that `M(r, c)` contains the $e^{\theta_r^T x^{(c)}}$ term, you can use the following code to compute the hypothesis (by dividing all elements in each column by their column sum):

```
% M is the matrix as described in the text
M = bsxfun(@rdivide, M, sum(M))
```

The operation of `bsxfun` in this case is analogous to the earlier example.

5.2.5 Step 3: Gradient checking

Once you have written the softmax cost function, you should check your gradients numerically. In general, whenever implementing any learning algorithm, you should always check your gradients numerically before proceeding to train the model. The norm of the difference between the numerical gradient and your analytical gradient should be small, on the order of 10^{-9}.

Implementation Tip: Faster gradient checking - when debugging, you can speed up gradient checking by reducing the number of parameters your model uses. In this case, we have included code for reducing the size of the input data, using the first 8 pixels of the images instead of the full 28×28 images. This code can be used by setting the variable `DEBUG` to true, as described in step 1 of the code.

5.2.6 Step 4: Learning parameters

Now that you've verified that your gradients are correct, you can train your softmax model using the function `softmaxTrain` in `softmaxTrain.m`. `softmaxTrain` which uses the L-BFGS algorithm, in the function `minFunc`. Training the model on the entire MNIST training set of 60000 28×28 images should be rather quick, and take less than 5 minutes for 100 iterations.

Factoring `softmaxTrain` out as a function means that you will be able to easily reuse it to train softmax models on other data sets in the future by invoking the function with different parameters.

Use the following parameter when training your softmax classifier:

```
lambda = 1e-4
```

5.2.7 Step 5: Testing

Now that you've trained your model, you will test it against the MNIST test set, comprising 10000 28×28 images. However, to do so, you will first need to complete the function `softmaxPredict` in `softmaxPredict.m`, a function which generates predictions for input data under a trained softmax model.

Once that is done, you will be able to compute the accuracy (the proportion of correctly classified images) of your model using the code provided. Our implementation achieved an accuracy of 92.6%. If your model's accuracy is significantly less (less than 91%), check your code, ensure that you are using the trained weights, and that you are training your model on the full 60000 training images. Conversely, if your accuracy is too high (99-100%), ensure that you have not accidentally trained your model on the test set as well.

6 Self-Taught Learning and Unsupervised Feature Learning

6.1 Self-Taught Learning

6.1.1 Overview

Assuming that we have a sufficiently powerful learning algorithm, one of the most reliable ways to get better performance is to give the algorithm more data. This has led to the that aphorism that in machine learning, "sometimes it's not who has the best algorithm that wins; it's who has the most data."

One can always try to get more labeled data, but this can be expensive. In particular, researchers have already gone to extraordinary lengths to use tools such as AMT (Amazon Mechanical Turk) to get large training sets. While having large numbers of people hand-label lots of data is probably a step forward compared to having large numbers of researchers hand-engineer features, it would be nice to do better. In particular, the promise of self-taught learning and unsupervised feature learning is that if we can get our algorithms to learn from unlabeled data, then we can easily obtain and learn from massive amounts of it. Even though a single unlabeled example is less informative than a single labeled example, if we can get tons of the former—for example, by downloading random unlabeled images/audio clips/text documents off the internet—and if our algorithms can exploit this unlabeled data effectively, then we might be able to achieve better performance than the massive hand-engineering and massive hand-labeling approaches.

In Self-taught learning and Unsupervised feature learning, we will give our algorithms a large amount of unlabeled data with which to learn a good feature representation of the input. If we are trying to solve a specific classification task, then we take this learned feature representation and whatever (perhaps small amount of) labeled data we have for that classification task, and apply supervised learning on that labeled data to solve the classification task.

These ideas probably have the most powerful effects in problems where we have a lot of unlabeled data, and a smaller amount of labeled data. However, they typically give good results even if we have only labeled data (in which case we usually perform the feature learning step using the labeled data, but ignoring the labels).

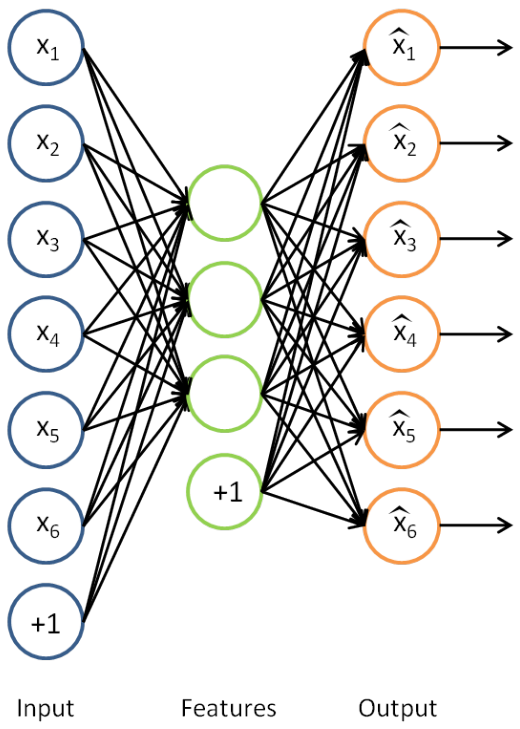
6.1.2 Learning features

We have already seen how an autoencoder can be used to learn features from unlabeled data. Concretely, suppose we have an unlabeled training set $\{x_u^{(1)}, x_u^{(2)}, \ldots, x_u^{(m_u)}\}$ with m_u unlabeled examples. (The subscript "u" stands for "unlabeled.") We can then train a sparse autoencoder on this data (perhaps with appropriate whitening or other pre-processing):

Having trained the parameters $W^{(1)}, b^{(1)}, W^{(2)}, b^{(2)}$ of this model, given any new input x, we can now compute the corresponding vector of activations a of the hidden units. As we saw previously, this often gives a better representation of the input than the original raw input x. We can also visualize the algorithm for computing the features/activations a as the following neural network:

This is just the sparse autoencoder that we previously had, with with the final layer removed.

Now, suppose we have a labeled training set $\{(x_l^{(1)}, y^{(1)}), (x_l^{(2)}, y^{(2)}), \ldots (x_l^{(m_l)}, y^{(m_l)})\}$ of m_l examples. (The subscript "l" stands for "labeled.") We can now find a better representation for the inputs. In particular, rather than representing the first training example as $x_l^{(1)}$, we can feed $x_l^{(1)}$ as the input to our autoencoder, and obtain the corresponding vector of activations $a_l^{(1)}$. To represent this example, we can either just replace the original feature vector with $a_l^{(1)}$. Alternatively, we can



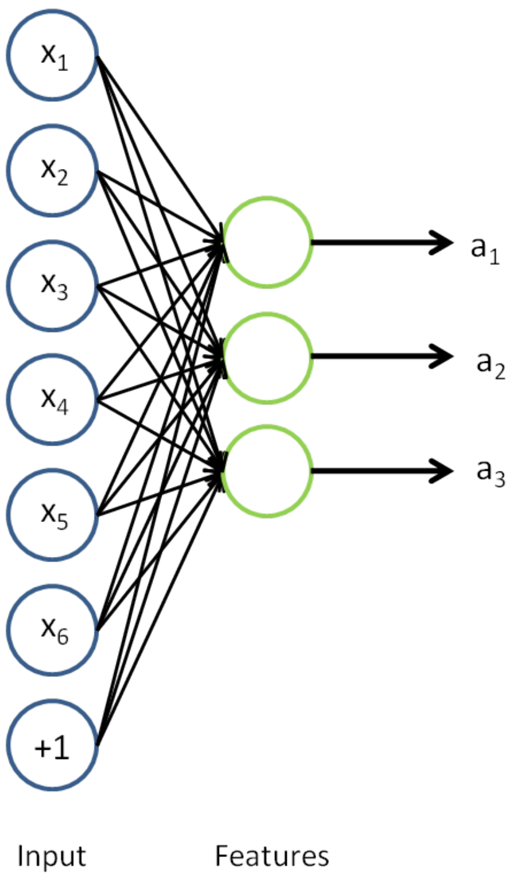
concatenate the two feature vectors together, getting a representation $(x_l^{(1)}, a_l^{(1)})$.

Thus, our training set now becomes $\{(a_l^{(1)}, y^{(1)}), (a_l^{(2)}, y^{(2)}), \dots (a_l^{(m_l)}, y^{(m_l)})\}$ (if we use the replacement representation, and use $a_l^{(i)}$ to represent the i-th training example), or $\{((x_l^{(1)}, a_l^{(1)}), y^{(1)}), ((x_l^{(2)}, a_l^{(1)}),$ (if we use the concatenated representation). In practice, the concatenated representation often works better; but for memory or computation representations, we will sometimes use the replacement representation as well.

Finally, we can train a supervised learning algorithm such as an SVM, logistic regression, etc. to obtain a function that makes predictions on the y values. Given a test example x_{test}, we would then follow the same procedure: For feed it to the autoencoder to get a_{test}. Then, feed either a_{test} or $(x_{\text{test}}, a_{\text{test}})$ to the trained classifier to get a prediction.

6.1.3 On pre-processing the data

During the feature learning stage where we were learning from the unlabeled training set $\{x_u^{(1)}, x_u^{(2)}, \dots, x_u^{(m_u)}\}$, we may have computed various pre-processing parameters. For example, one may have computed a mean value of the data and subtracted off this mean to perform mean normalization, or used PCA to compute a matrix U to represent the data as $U^T x$ (or used PCA whitening or ZCA whitening). If this is the case, then it is important to save away these preprocessing parameters, and to use the same parameters during the labeled training phase and the test phase, so as to make sure we are always transforming the data the same way to feed into the autoencoder. In particular, if we have computed a matrix U using the unlabeled data and PCA, we should keep the same matrix U and use it to preprocess the labeled examples and the test data. We should not re-estimate a different U matrix (or data mean for mean normalization, etc.) using the labeled training set, since that might result in a dramatically different pre-processing transformation, which would make the input distribution to the autoencoder very different from what it was actually trained on.



6.1.4 On the terminology of unsupervised feature learning

There are two common unsupervised feature learning settings, depending on what type of unlabeled data you have. The more general and powerful setting is the self-taught learning setting, which does not assume that your unlabeled data x_u has to be drawn from the same distribution as your labeled data x_l. The more restrictive setting where the unlabeled data comes from exactly the same distribution as the labeled data is sometimes called the semi-supervised learning setting. This distinctions is best explained with an example, which we now give.

Suppose your goal is a computer vision task where you'd like to distinguish between images of cars and images of motorcycles; so, each labeled example in your training set is either an image of a car or an image of a motorcycle. Where can we get lots of unlabeled data? The easiest way would be to obtain some random collection of images, perhaps downloaded off the internet. We could then train the autoencoder on this large collection of images, and obtain useful features from them. Because here the unlabeled data is drawn from a different distribution than the labeled data (i.e., perhaps some of our unlabeled images may contain cars/motorcycles, but not every image downloaded is either a car or a motorcycle), we call this self-taught learning.

In contrast, if we happen to have lots of unlabeled images lying around that are all images of either a car or a motorcycle, but where the data is just missing its label (so you don't know which ones are cars, and which ones are motorcycles), then we could use this form of unlabeled data to learn the features. This setting—where each unlabeled example is drawn from the same distribution as your labeled examples—is sometimes called the semi-supervised setting. In practice, we often do not have this sort of unlabeled data (where would you get a database of images where every image is either a car or a motorcycle, but just missing its label?), and so in the context of learning features from unlabeled data, the self-taught learning setting is more broadly applicable.

6.2 Exercise: Self-Taught Learning

6.2.1 Overview

In this exercise, we will use the self-taught learning paradigm with the sparse autoencoder and softmax classifier to build a classifier for handwritten digits.

You will be building upon your code from the earlier exercises. First, you will train your sparse autoencoder on an "unlabeled" training dataset of handwritten digits. This produces feature that are penstroke-like. We then extract these learned features from a labeled dataset of handwritten digits. These features will then be used as inputs to the softmax classifier that you wrote in the previous exercise.

Concretely, for each example in the the labeled training dataset x_l, we forward propagate the example to obtain the activation of the hidden units $a^{(2)}$. We now represent this example using $a^{(2)}$ (the "replacement" representation), and use this to as the new feature representation with which to train the softmax classifier.

Finally, we also extract the same features from the test data to obtain predictions.

In this exercise, our goal is to distinguish between the digits from 0 to 4. We will use the digits 5 to 9 as our "unlabeled" dataset which which to learn the features; we will then use a labeled dataset with the digits 0 to 4 with which to train the softmax classifier.

In the starter code, we have provided a file `stlExercise.m` that will help walk you through the steps in this exercise.

6.2.2 Dependencies

The following additional files are required for this exercise:

- MNIST Dataset `http://yann.lecun.com/exdb/mnist/`
- Support functions for loading MNIST in Matlab (A)
- Starter Code stl_exercise.zip)

You will also need your code from the following exercises:

- Exercise:Sparse Autoencoder (2.7)
- Exercise:Vectorization (3.4)
- Exercise:Softmax Regression (5.2)

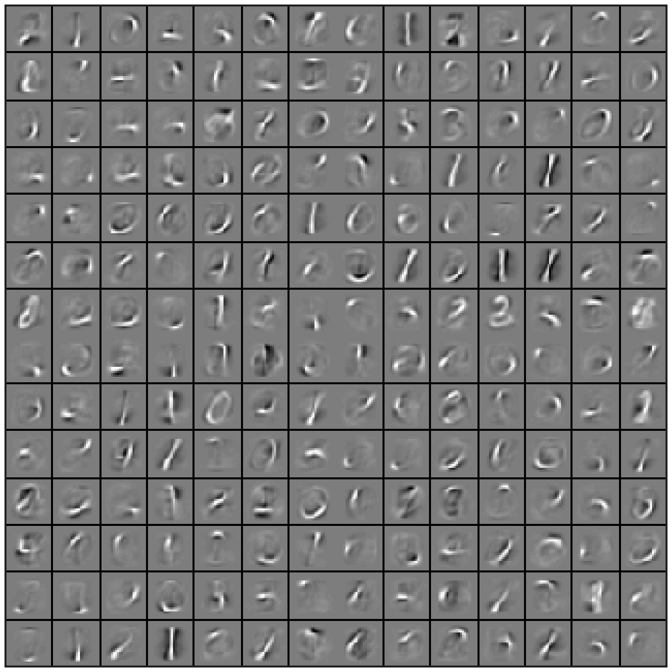
If you have not completed the exercises listed above, we strongly suggest you complete them first.

6.2.3 Step 1: Generate the input and test data sets

Download and decompress stl_exercise.zip, which contains starter code for this exercise. Additionally, you will need to download the datasets from the MNIST Handwritten Digit Database for this project.

6.2.4 Step 2: Train the sparse autoencoder

Next, use the unlabeled data (the digits from 5 to 9) to train a sparse autoencoder, using the same `sparseAutoencoderCost.m` function as you had written in the previous exercise. (From the earlier exercise, you should have a working and vectorized implementation of the sparse autoencoder.) For us, the training step took less than 25 minutes on a fast desktop. When training is complete, you should get a visualization of pen strokes like the image shown below:



Informally, the features learned by the sparse autoencoder should correspond to penstrokes.

6.2.5 Step 3: Extracting features

After the sparse autoencoder is trained, you will use it to extract features from the handwritten digit images.

Complete `feedForwardAutoencoder.m` to produce a matrix whose columns correspond to activations of the hidden layer for each example, i.e., the vector $a^{(2)}$ corresponding to activation of layer 2. (Recall that we treat the inputs as layer 1).

After completing this step, calling `feedForwardAutoencoder.m` should convert the raw image data to hidden unit activations $a^{(2)}$.

6.2.6 Step 4: Training and testing the logistic regression model

Use your code from the softmax exercise (`softmaxTrain.m`) to train a softmax classifier using the training set features (`trainFeatures`) and labels (`trainLabels`).

6.2.7 Step 5: Classifying on the test set

Finally, complete the code to make predictions on the test set (`testFeatures`) and see how your learned features perform! If you've done all the steps correctly, you should get an accuracy of about 98% percent.

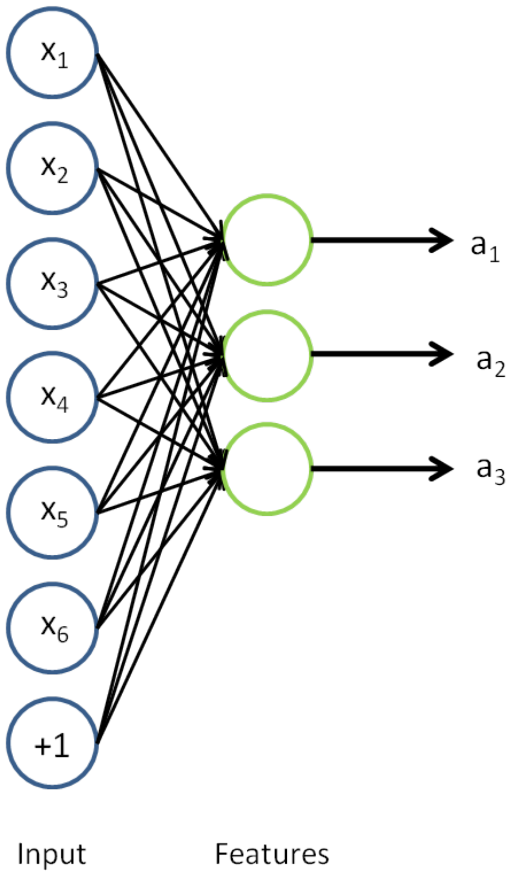
As a comparison, when raw pixels are used (instead of the learned features), we obtained a test accuracy of only around 96% (for the same train and test sets).

7 Building Deep Networks for Classification

7.1 From Self-Taught Learning to Deep Networks

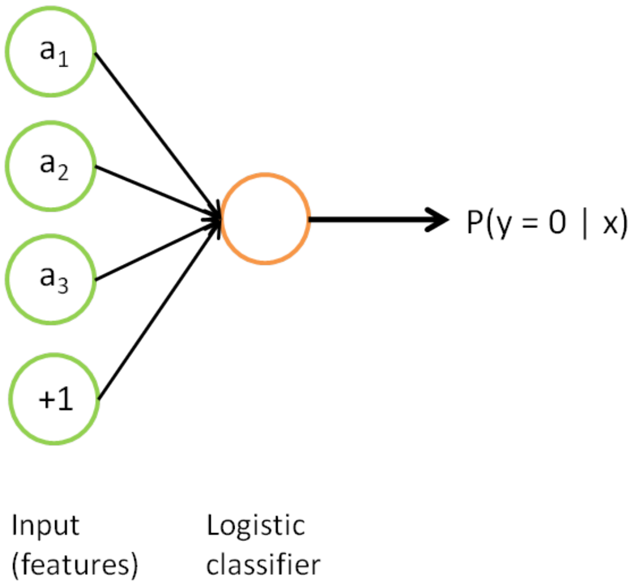
In the previous section(6), you used an autoencoder to learn features that were then fed as input to a softmax or logistic regression classifier. In that method, the features were learned using only unlabeled data. In this section, we describe how you can fine-tune and further improve the learned features using labeled data. When you have a large amount of labeled training data, this can significantly improve your classifier's performance.

In self-taught learning, we first trained a sparse autoencoder on the unlabeled data. Then, given a new example x, we used the hidden layer to extract features a. This is illustrated in the following diagram:



We are interested in solving a classification task, where our goal is to predict labels y. We have a labeled training set $\{(x_l^{(1)}, y^{(1)}), (x_l^{(2)}, y^{(2)}), \ldots (x_l^{(m_l)}, y^{(m_l)})\}$ of m_l labeled examples. We showed previously that we can replace the original features $x^{(i)}$ with features $a^{(l)}$ computed by the sparse autoencoder (the"replacement" representation). This gives us a training set $\{(a^{(1)}, y^{(1)}), \ldots (a^{(m_l)}, y^{(m_l)})\}$. Finally, we train a logistic classifier to map from the features $a^{(i)}$ to the classification label $y^{(i)}$. To illustrate this step, similar to our earlier notes(2.1), we can draw our logistic regression unit (shown in orange) as follows:

Now, consider the overall classifier (i.e., the input-output mapping) that we have learned using this method. In particular, let us examine the function that our classifier uses to map from from a new test example x to a new prediction $p(y = 1|x)$. We can draw a representation of this function by putting together the two pictures from above. In particular, the final classifier looks like this:



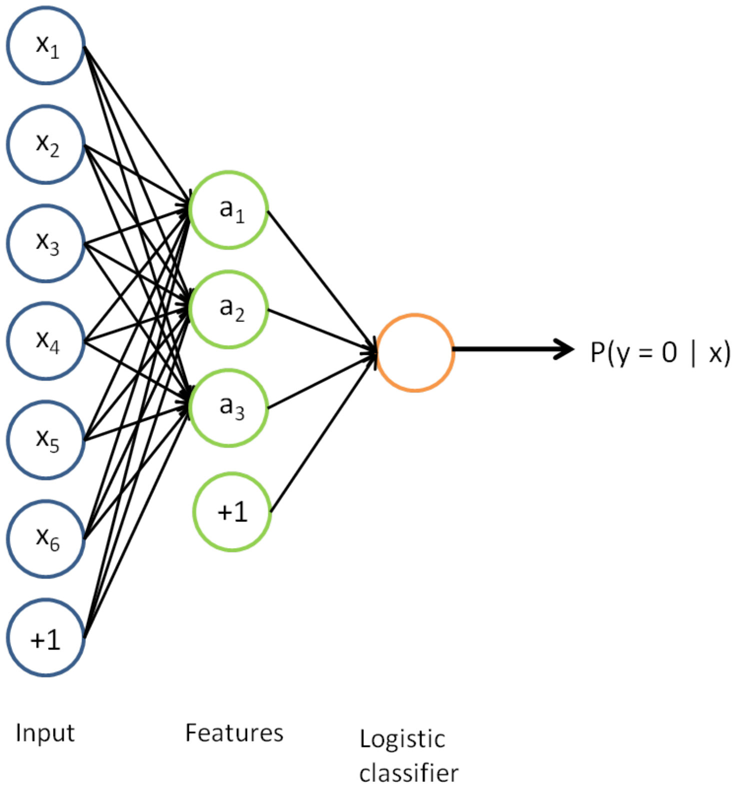
The parameters of this model were trained in two stages: The first layer of weights $W^{(1)}$ mapping from the input x to the hidden unit activations a were trained as part of the sparse autoencoder training process. The second layer of weights $W^{(2)}$ mapping from the activations a to the output y was trained using logistic regression (or softmax regression).

But the form of our overall/final classifier is clearly just a whole big neural network. So, having trained up an initial set of parameters for our model (training the first layer using an autoencoder, and the second layer via logistic/softmax regression), we can further modify all the parameters in our model to try to further reduce the training error. In particular, we can fine-tune the parameters, meaning perform gradient descent (or use L-BFGS) from the current setting of the parameters to try to reduce the training error on our labeled training set $\{(x_l^{(1)}, y^{(1)}), (x_l^{(2)}, y^{(2)}), \ldots (x_l^{(m_l)}, y^{(m_l)})\}$.

When fine-tuning is used, sometimes the original unsupervised feature learning steps (i.e., training the autoencoder and the logistic classifier) are called pre-training. The effect of fine-tuning is that the labeled data can be used to modify the weights $W^{(1)}$ as well, so that adjustments can be made to the features a extracted by the layer of hidden units.

So far, we have described this process assuming that you used the "replacement" representation, where the training examples seen by the logistic classifier are of the form $(a^{(i)}, y^{(i)})$, rather than the "concatenation" representation, where the examples are of the form $((x^{(i)}, a^{(i)}), y^{(i)})$. It is also possible to perform fine-tuning too using the "concatenation" representation. (This corresponds to a neural network where the input units x_i also feed directly to the logistic classifier in the output layer. You can draw this using a slightly different type of neural network diagram than the ones we have seen so far; in particular, you would have edges that go directly from the first layer input nodes to the third layer output node, "skipping over" the hidden layer.) However, so long as we are using finetuning, usually the "concatenation" representation has little advantage over the "replacement" representation. Thus, if we are using fine-tuning usually we will do so with a network built using the replacement representation. (If you are not using fine-tuning however, then sometimes the concatenation representation can give much better performance.)

When should we use fine-tuning? It is typically used only if you have a large labeled training set; in this setting, fine-tuning can significantly improve the performance of your classifier. However, if you have a large unlabeled dataset (for unsupervised feature learning/pre-training) and only a



relatively small labeled training set, then fine-tuning is significantly less likely to help.

7.2 Deep Networks: Overview

7.2.1 Overview

In the previous sections, you constructed a 3-layer neural network comprising an input, hidden and output layer. While fairly effective for MNIST, this 3-layer model is a fairly shallow network; by this, we mean that the features (hidden layer activations $a^{(2)}$) are computed using only “one layer" of computation (the hidden layer).

In this section, we begin to discuss deep neural networks, meaning ones in which we have multiple hidden layers; this will allow us to compute much more complex features of the input. Because each hidden layer computes a non-linear transformation of the previous layer, a deep network can have significantly greater representational power (i.e., can learn significantly more complex functions) than a shallow one.

Note that when training a deep network, it is important to use a non-linear activation function $f(\cdot)$ in each hidden layer. This is because multiple layers of linear functions would itself compute only a linear function of the input (i.e., composing multiple linear functions together results in just another linear function), and thus be no more expressive than using just a single layer of hidden units.

7.2.2 Advantages of deep networks

Why do we want to use a deep network? The primary advantage is that it can compactly represent a significantly larger set of fuctions than shallow networks. Formally, one can show that there are functions which a k-layer network can represent compactly (with a number of hidden units that is polynomial in the number of inputs), that a $(k-1)$-layer network cannot represent unless it has an exponentially large number of hidden units.

To take a simple example, consider building a boolean circuit/network to compute the parity (or XOR) of n input bits. Suppose each node in the network can compute either the logical OR of its inputs (or the OR of the negation of the inputs), or compute the logical AND. If we have a network with only one input, one hidden, and one output layer, the parity function would require a number of nodes that is exponential in the input size n. If however we are allowed a deeper network, then the network/circuit size can be only polynomial in n.

By using a deep network, in the case of images, one can also start to learn part-whole decompositions. For example, the first layer might learn to group together pixels in an image in order to detect edges (as seen in the earlier exercises). The second layer might then group together edges to detect longer contours, or perhaps detect simple "parts of objects." An even deeper layer might then group together these contours or detect even more complex features.

Finally, cortical computations (in the brain) also have multiple layers of processing. For example, visual images are processed in multiple stages by the brain, by cortical area "V1", followed by cortical area "V2" (a different part of the brain), and so on.

7.2.3 Difficulty of training deep architectures

While the theoretical benefits of deep networks in terms of their compactness and expressive power have been appreciated for many decades, until recently researchers had little success training deep architectures.

The main learning algorithm that researchers were using was to randomly initialize the weights of a deep network, and then train it using a labeled training set $\{(x_l^{(1)}, y^{(1)}), \ldots, (x_l^{(m_l)}, y^{(m_l)})\}$ using a supervised learning objective, for example by applying gradient descent to try to drive down the training error. However, this usually did not work well. There were several reasons for this.

Availability of data

With the method described above, one relies only on labeled data for training. However, labeled data is often scarce, and thus for many problems it is difficult to get enough examples to fit the parameters of a complex model. For example, given the high degree of expressive power of deep networks, training on insufficient data would also result in overfitting.

Local optima

Training a shallow network (with 1 hidden layer) using supervised learning usually resulted in the parameters converging to reasonable values; but when we are training a deep network, this works much less well. In particular, training a neural network using supervised learning involves solving a highly non-convex optimization problem (say, minimizing the training error $\sum_i ||h_W(x^{(i)}) - y^{(i)}||^2$ as a function of the network parameters W). In a deep network, this problem turns out to be rife with bad local optima, and training with gradient descent (or methods like conjugate gradient and L-BFGS) no longer work well.

Diffusion of gradients

There is an additional technical reason, pertaining to the gradients becoming very small, that explains why gradient descent (and related algorithms like L-BFGS) do not work well on a deep networks with randomly initialized weights. Specifically, when using backpropagation to compute the derivatives, the gradients that are propagated backwards (from the output layer to the earlier layers of the network) rapidly diminish in magnitude as the depth of the network increases. As a result, the derivative of the overall cost with respect to the weights in the earlier layers is very small. Thus, when using gradient descent, the weights of the earlier layers change slowly, and the earlier layers fail to learn much. This problem is often called the “diffusion of gradients."

A closely related problem to the diffusion of gradients is that if the last few layers in a neural network have a large enough number of neurons, it may be possible for them to model the labeled data alone without the help of the earlier layers. Hence, training the entire network at once with all the layers randomly initialized ends up giving similar performance to training a shallow network (the last few layers) on corrupted input (the result of the processing done by the earlier layers).

7.2.4 Greedy layer-wise training

How can we train a deep network? One method that has seen some success is the greedy layer-wise training method. We describe this method in detail in later sections, but briefly, the main idea is to train the layers of the network one at a time, so that we first train a network with 1 hidden layer, and only after that is done, train a network with 2 hidden layers, and so on. At each step, we take the old network with $k-1$ hidden layers, and add an additional k-th hidden layer (that takes as input the previous hidden layer $k-1$ that we had just trained). Training can either be supervised (say, with classification error as the objective function on each step), but more frequently it is unsupervised (as in an autoencoder; details to provided later). The weights from training the layers individually are then used to initialize the weights in the final/overall deep network, and only then is the entire architecture “fine-tuned" (i.e., trained together to optimize the labeled training set error).

The success of greedy layer-wise training has been attributed to a number of factors:

Availability of data

While labeled data can be expensive to obtain, unlabeled data is cheap and plentiful. The promise of self-taught learning is that by exploiting the massive amount of unlabeled data, we can learn much better models. By using unlabeled data to learn a good initial value for the weights in all the layers $W^{(l)}$ (except for the final classification layer that maps to the outputs/predictions), our algorithm is able to learn and discover patterns from massively more amounts of data than purely supervised approaches. This often results in much better classifiers being learned.

Better local optima

After having trained the network on the unlabeled data, the weights are now starting at a better location in parameter space than if they had been randomly initialized. We can then further fine-tune the weights starting from this location. Empirically, it turns out that gradient descent from this location is much more likely to lead to a good local minimum, because the unlabeled data has already provided a significant amount of “prior" information about what patterns there are in the input data.

In the next section, we will describe the specific details of how to go about implementing greedy layer-wise training.

7.3 Stacked Autoencoders

7.3.1 Overview

The greedy layerwise approach for pretraining a deep network works by training each layer in turn. In this page, you will find out how autoencoders can be “stacked" in a greedy layerwise fashion for pretraining (initializing) the weights of a deep network.

A stacked autoencoder is a neural network consisting of multiple layers of sparse autoencoders in which the outputs of each layer is wired to the inputs of the successive layer. Formally, consider a stacked autoencoder with n layers. Using notation from the autoencoder section, let $W^{(k,1)}, W^{(k,2)}, b^{(k,1)}, b^{(k,2)}$ denote the parameters $W^{(1)}, W^{(2)}, b^{(1)}, b^{(2)}$ for kth autoencoder. Then the encoding step for the stacked autoencoder is given by running the encoding step of each layer in forward order:

$$a^{(l)} = f(z^{(l)}) \tag{87}$$
$$z^{(l+1)} = W^{(l,1)}a^{(l)} + b^{(l,1)} \tag{88}$$

The decoding step is given by running the decoding stack of each autoencoder in reverse order:

$$a^{(n+l)} = f(z^{(n+l)}) \tag{89}$$
$$z^{(n+l+1)} = W^{(n-l,2)}a^{(n+l)} + b^{(n-l,2)} \tag{90}$$

The information of interest is contained within $a^{(n)}$, which is the activation of the deepest layer of hidden units. This vector gives us a representation of the input in terms of higher-order features.

The features from the stacked autoencoder can be used for classification problems by feeding $a^{(n)}$ to a softmax classifier.

Training

A good way to obtain good parameters for a stacked autoencoder is to use greedy layer-wise training. To do this, first train the first layer on raw input to obtain parameters $W^{(1,1)}, W^{(1,2)}, b^{(1,1)}, b^{(1,2)}$. Use the first layer to transform the raw input into a vector consisting of activation of the hidden units, A. Train the second layer on this vector to obtain parameters $W^{(2,1)}, W^{(2,2)}, b^{(2,1)}, b^{(2,2)}$. Repeat for subsequent layers, using the output of each layer as input for the subsequent layer.

This method trains the parameters of each layer individually while freezing parameters for the remainder of the model. To produce better results, after this phase of training is complete, fine-tuning(Fine-tuning Stacked AEs 7.4) using backpropagation can be used to improve the results by tuning the parameters of all layers are changed at the same time.

If one is only interested in finetuning for the purposes of classification, the common practice is to then discard the “decoding" layers of the stacked autoencoder and link the last hidden layer $a^{(n)}$ to the softmax classifier. The gradients from the (softmax) classification error will then be backpropagated into the encoding layers.

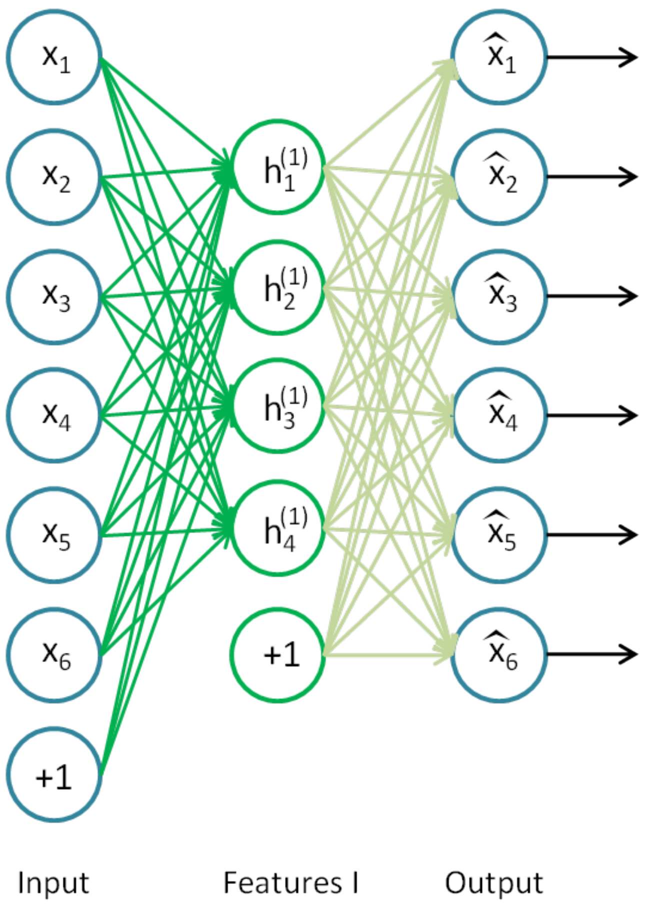
Concrete example

To give a concrete example, suppose you wished to train a stacked autoencoder with 2 hidden layers for classification of MNIST digits, as you will be doing in the next exercise(7.5).

First, you would train a sparse autoencoder on the raw inputs $x^{(k)}$ to learn primary features $h^{(1)(k)}$ on the raw input.

Next, you would feed the raw input into this trained sparse autoencoder, obtaining the primary feature activations $h^{(1)(k)}$ for each of the inputs $x^{(k)}$. You would then use these primary features as the “raw input" to another sparse autoencoder to learn secondary features $h^{(2)(k)}$ on these primary features.

Following this, you would feed the primary features into the second sparse autoencoder to obtain the secondary feature activations $h^{(2)(k)}$ for each of the primary features $h^{(1)(k)}$ (which correspond to the primary features of the corresponding inputs $x^{(k)}$). You would then treat these secondary features as “raw input" to a softmax classifier, training it to map secondary features to digit labels.



Finally, you would combine all three layers together to form a stacked autoencoder with 2 hidden layers and a final softmax classifier layer capable of classifying the MNIST digits as desired.

Discussion

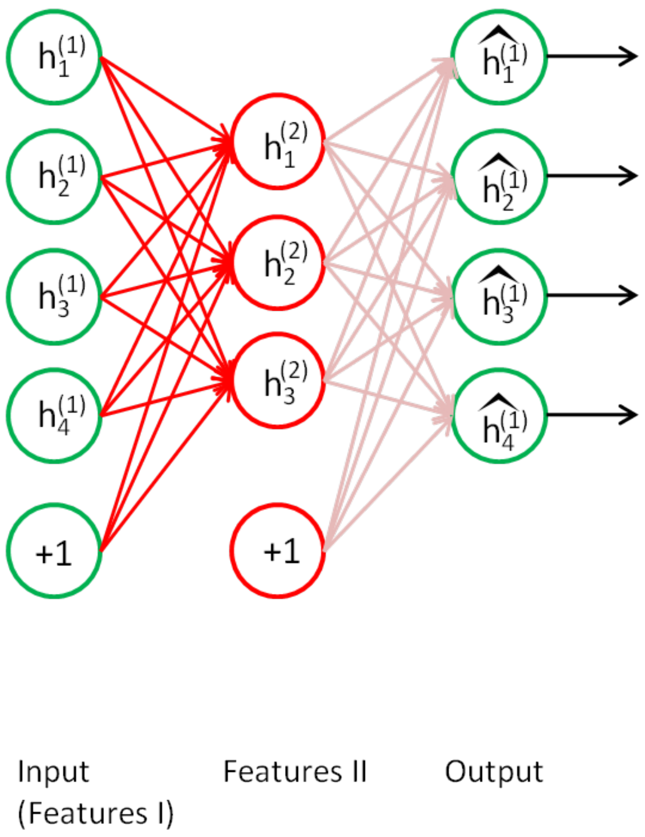
A stacked autoencoder enjoys all the benefits of any deep network of greater expressive power.

Further, it often captures a useful “hierarchical grouping" or “part-whole decomposition" of the input. To see this, recall that an autoencoder tends to learn features that form a good representation of its input. The first layer of a stacked autoencoder tends to learn first-order features in the raw input (such as edges in an image). The second layer of a stacked autoencoder tends to learn second-order features corresponding to patterns in the appearance of first-order features (e.g., in terms of what edges tend to occur together–for example, to form contour or corner detectors). Higher layers of the stacked autoencoder tend to learn even higher-order features.

7.4 Fine-tuning Stacked AEs

7.4.1 Introduction

Fine tuning is a strategy that is commonly found in deep learning. As such, it can also be used to greatly improve the performance of a stacked autoencoder. From a high level perspective, fine tuning treats all layers of a stacked autoencoder as a single model, so that in one iteration, we are improving upon all the weights in the stacked autoencoder.



7.4.2 General Strategy

Fortunately, we already have all the tools necessary to implement fine tuning for stacked autoencoders! In order to compute the gradients for all the layers of the stacked autoencoder in each iteration, we use the Backpropagation Algorithm(2.2), as discussed in the sparse autoencoder section. As the backpropagation algorithm can be extended to apply for an arbitrary number of layers, we can actually use this algorithm on a stacked autoencoder of arbitrary depth.

7.4.3 Finetuning with Backpropagation

For your convenience, the summary of the backpropagation algorithm using element wise notation is below:

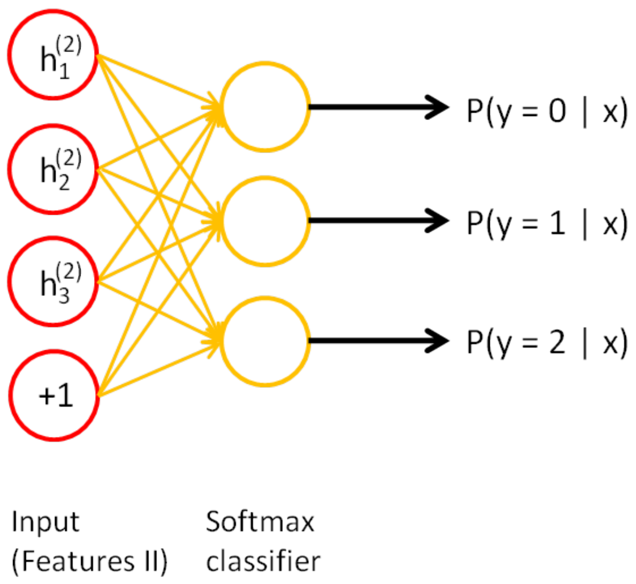
1. Perform a feedforward pass, computing the activations for layers L_2, L_3, up to the output layer L_{n_l}, using the equations defining the forward propagation steps.

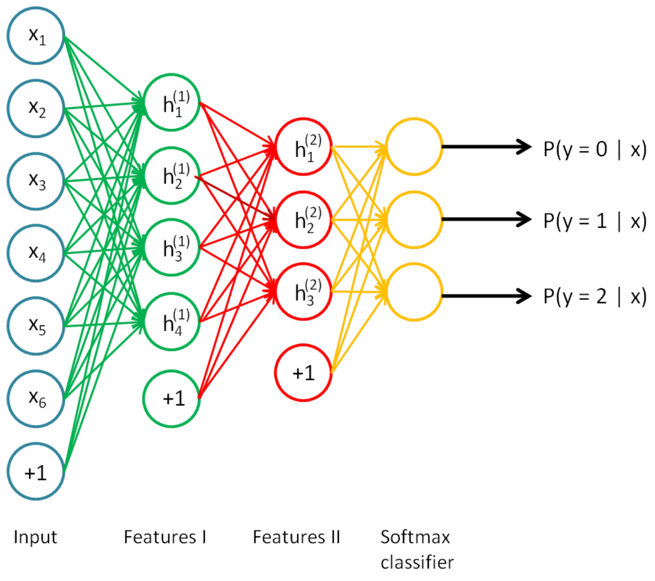
2. For the output layer (layer n_l), set

$$\delta^{(n_l)} = -(\nabla_{a^{n_l}} J) \bullet f'(z^{(n_l)}) \tag{91}$$

   (When using softmax regression, the softmax layer has $\nabla J = \theta^T(I - P)$ where I is the input labels and P is the vector of conditional probabilities.)

3. For $l = n_l - 1, n_l - 2, n_l - 3, \ldots, 2$; Set





$$\delta^{(l)} = \left((W^{(l)})^T \delta^{(l+1)}\right) \bullet f'(z^{(l)}) \tag{92}$$

4. Compute the desired partial derivatives:

$$\nabla_{W^{(l)}} J(W, b; x, y) = \delta^{(l+1)} (a^{(l)})^T, \tag{93}$$
$$\nabla_{b^{(l)}} J(W, b; x, y) = \delta^{(l+1)}. \tag{94}$$

$$J(W, b) = \left[\frac{1}{m} \sum_{i=1}^{m} J(W, b; x^{(i)}, y^{(i)})\right] \tag{95}$$

Note: While one could consider the softmax classifier as an additional layer, the derivation above does not. Specifically, we consider the "last layer" of the network to be the features that goes into the softmax classifier. Therefore, the derivatives (in Step 2) are computed using $\delta^{(n_l)} = -(\nabla_{a^{n_l}} J) \bullet f'(z^{(n_l)})$, where $\nabla J = \theta^T (I - P)$.

7.5 Exercise: Implement deep networks for digit classification

7.5.1 Overview

In this exercise, you will use a stacked autoencoder for digit classification. This exercise is very similar to the self-taught learning exercise, in which we trained a digit classifier using a autoencoder layer followed by a softmax layer. The only difference in this exercise is that we will be using two autoencoder layers instead of one and further finetune the two layers.

The code you have already implemented will allow you to stack various layers and perform layer-wise training. However, to perform fine-tuning, you will need to implement backpropogation through both layers. We will see that fine-tuning significantly improves the model's performance.

In the file stackedae_exercise.zip, we have provided some starter code. You will need to complete the code in `stackedAECost.m`, `stackedAEPredict.m` and `stackedAEExercise.m`. We have also provided `params2stack.m` and `stack2params.m` which you might find helpful in constructing deep networks.

7.5.2 Dependencies

The following additional files are required for this exercise:

- MNIST Dataset `http://yann.lecun.com/exdb/mnist/`
- Support functions for loading MNIST in Matlab (A)
- Starter Code (stackedae_exercise.zip)

You will also need your code from the following exercises:

- Exercise:Sparse Autoencoder (2.7)
- Exercise:Vectorization (3.4)
- Exercise:Softmax Regression (5.2)
- Exercise:Self-Taught Learning (6.2)

If you have not completed the exercises listed above, we strongly suggest you complete them first.

7.5.3 Step 0: Initialize constants and parameters

Open `stackedAEExercise.m`. In this step, we set meta-parameters to the same values that were used in previous exercise, which should produce reasonable results. You may to modify the meta-parameters if you wish.

7.5.4 Step 1: Train the data on the first stacked autoencoder

Train the first autoencoder on the training images to obtain its parameters. This step is identical to the corresponding step in the sparse autoencoder and STL assignments, complete this part of the code so as to learn a first layer of features using your `sparseAutoencoderCost.m` and `minFunc`.

7.5.5 Step 2: Train the data on the second stacked autoencoder

We first forward propagate the training set through the first autoencoder (using `feedForwardAutoencoder` that you completed in Exercise:Self-Taught Learning(6.2)) to obtain hidden unit activations. These activations are then used to train the second sparse autoencoder. Since this is just an adapted application of a standard autoencoder, it should run similarly with the first. Complete this part of the code so as to learn a first layer of features using your `sparseAutoencoderCost.m` and `minFunc`.

This part of the exercise demonstrates the idea of greedy layerwise training with the same learning algorithm reapplied multiple times.

7.5.6 Step 3: Train the softmax classifier on the L2 features

Next, continue to forward propagate the L1 features through the second autoencoder (using `feedForwardAutoencoder.m`) to obtain the L2 hidden unit activations. These activations are then used to train the softmax classifier. You can either use `softmaxTrain.m` or directly use `softmaxCost.m` that you completed in Exercise:Softmax Regression(5.2) to complete this part of the assignment.

7.5.7 Step 4: Implement fine-tuning

To implement fine tuning, we need to consider all three layers as a single model. Implement `stackedAECost.m` to return the cost and gradient of the model. The cost function should be as defined as the log likelihood and a gradient decay term. The gradient should be computed using back-propagation as discussed earlier(2.2). The predictions should consist of the activations of the output layer of the softmax model.

To help you check that your implementation is correct, you should also check your gradients on a synthetic small dataset. We have implemented `checkStackedAECost.m` to help you check your gradients. If this checks passes, you will have implemented fine-tuning correctly.

Note: When adding the weight decay term to the cost, you should regularize only the softmax weights (do not regularize the weights that compute the hidden layer activations).

Implementation Tip: It is always a good idea to implement the code modularly and check (the gradient of) each part of the code before writing the more complicated parts.

7.5.8 Step 5: Test the model

Finally, you will need to classify with this model; complete the code in `stackedAEPredict.m` to classify using the stacked autoencoder with a classification layer.

After completing these steps, running the entire script in stackedAETrain.m will perform layer-wise training of the stacked autoencoder, finetune the model, and measure its performance on the test set. If you've done all the steps correctly, you should get an accuracy of about 87.7% before finetuning and 97.6% after finetuning (for the 10-way classification problem).

8 Linear Decoders with Autoencoders

8.1 Linear Decoders

8.1.1 Sparse Autoencoder Recap

In the sparse autoencoder, we had 3 layers of neurons: an input layer, a hidden layer and an output layer. In our previous description of autoencoders (and of neural networks), every neuron in the neural network used the same activation function. In these notes, we describe a modified version of the autoencoder in which some of the neurons use a different activation function. This will result in a model that is sometimes simpler to apply, and can also be more robust to variations in the parameters.

Recall that each neuron (in the output layer) computed the following:

$$z^{(3)} = W^{(2)}a^{(2)} + b^{(2)} \tag{96}$$

$$a^{(3)} = f(z^{(3)}) \tag{97}$$

where $a^{(3)}$ is the output. In the autoencoder, $a^{(3)}$ is our approximate reconstruction of the input $x = a^{(1)}$.

Because we used a sigmoid activation function for $f(z^{(3)})$, we needed to constrain or scale the inputs to be in the range $[0, 1]$, since the sigmoid function outputs numbers in the range $[0, 1]$. While some datasets like MNIST fit well with this scaling of the output, this can sometimes be awkward to satisfy. For example, if one uses PCA whitening, the input is no longer constrained to $[0, 1]$ and it's not clear what the best way is to scale the data to ensure it fits into the constrained range.

8.1.2 Linear Decoder

One easy fix for this problem is to set $a^{(3)} = z^{(3)}$. Formally, this is achieved by having the output nodes use an activation function that's the identity function $f(z) = z$, so that $a^{(3)} = f(z^{(3)}) = z^{(3)}$. This particular activation function $f(\cdot)$ is called the linear activation function (though perhaps "identity activation function" would have been a better name). Note however that in the hidden layer of the network, we still use a sigmoid (or tanh) activation function, so that the hidden unit activations are given by (say) $a^{(2)} = \sigma(W^{(1)}x + b^{(1)})$, where $\sigma(\cdot)$ is the sigmoid function, x is the input, and $W^{(1)}$ and $b^{(1)}$ are the weight and bias terms for the hidden units. It is only in the output layer that we use the linear activation function.

An autoencoder in this configuration–with a sigmoid (or tanh) hidden layer and a linear output layer–is called a linear decoder. In this model, we have $\hat{x} = a^{(3)} = z^{(3)} = W^{(2)}a + b^{(2)}$. Because the output $\hat{x}$ is a now linear function of the hidden unit activations, by varying $W^{(2)}$, each output unit $a^{(3)}$ can be made to produce values greater than 1 or less than 0 as well. This allows us to train the sparse autoencoder real-valued inputs without needing to pre-scale every example to a specific range.

Since we have changed the activation function of the output units, the gradients of the output units also change. Recall that for each output unit, we had set set the error terms as follows:

$$\delta_i^{(3)} = \frac{\partial}{\partial z_i} \frac{1}{2} \|y - \hat{x}\|^2 = -(y_i - \hat{x}_i) \cdot f'(z_i^{(3)}) \tag{98}$$

where $y = x$ is the desired output, $\hat{x}$ is the output of our autoencoder, and $f(\cdot)$ is our activation function. Because in the output layer we now have $f(z) = z$, that implies $f'(z) = 1$ and thus the above now simplifies to:

$$\delta_i^{(3)} = -(y_i - \hat{x}_i) \tag{99}$$

Of course, when using backpropagation to compute the error terms for the hidden layer:

$$\delta^{(2)} = \left((W^{(2)})^T \delta^{(3)}\right) \bullet f'(z^{(2)}) \tag{100}$$

Because the hidden layer is using a sigmoid (or **tanh**) activation f, in the equation above $f'(\cdot)$ should still be the derivative of the sigmoid (or **tanh**) function.

8.2 Exercise:Learning color features with Sparse Autoencoders

In this exercise, you will implement a linear decoder (8.1.1, a sparse autoencoder whose output layer uses a linear activation function). You will then apply it to learn features on color images from the STL-10 dataset. These features will be used in an later exercise on convolution and pooling for classifying STL-10 images.

In the file linear_decoder_exercise.zip we have provided some starter code. You should write your code at the places indicated “YOUR CODE HERE" in the files.

For this exercise, you will need to copy and modify `sparseAutoencoderCost.m` from the sparse autoencoder exercise(2.7).

8.2.1 Dependencies

You will need:

`sparseAutoencoderCost.m` (and related functions) from Exercise:Sparse Autoencoder(2.7)

The following additional file is also required for this exercise:

Sampled 8×8 patches from the STL-10 dataset (stl10_patches_100k.zip)

If you have not completed the exercise listed above, we strongly suggest you complete it first.

8.2.2 Learning from color image patches

In all the exercises so far, you have been working only with grayscale images. In this exercise, you will get to work with RGB color images for the first time.

Conveniently, the fact that an image has three color channels (RGB), rather than a single gray channel, presents little difficulty for the sparse autoencoder. You can just combine the intensities from all the color channels for the pixels into one long vector, as if you were working with a grayscale image with $3\times$ the number of pixels as the original image.

8.2.3 Step 0: Initialization

In this step, we initialize some parameters used in the exercise (see starter code for details).

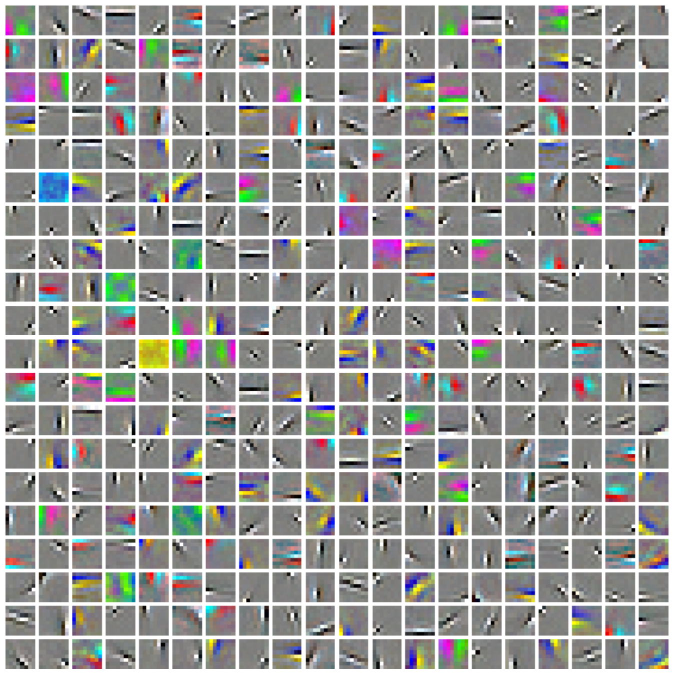
8.2.4 Step 1: Modify your sparse autoencoder to use a linear decoder

Copy `sparseAutoencoderCost.m` to the directory for this exercise and rename it to `sparseAutoencode`
Rename the function sparseAutoencoderCost in the file to sparseAutoencoderLinearCost, and modify it to use a linear decoder((8.1.1)). In particular, you should change the cost and gradients returned to reflect the change from a sigmoid to a linear decoder. After making this change, check your gradients to ensure that they are correct.

8.2.5 Step 2: Learn features on small patches

You will now use your sparse autoencoder to learn features on a set of 100,000 small 8×8 patches sampled from the larger 96×96 STL-10 images (The STL-10 dataset comprises 5000 training and 8000 test examples, with each example being a 96×96 labelled color image belonging to one of ten classes: airplane, bird, car, cat, deer, dog, horse, monkey, ship, truck.)

The code provided in this step trains your sparse autoencoder for 400 iterations with the default parameters initialized in step 0. This should take around 45 minutes. Your sparse autoencoder should learn features which when visualized, look like edges and “opponent colors," as in the figure below.



If your parameters are improperly tuned (the default parameters should work), or if your implementation of the autoencoder is buggy, you might instead get images that look like one of the following:

The learned features will be saved to `STL10Features.mat`, which will be used in the later exercise on convolution and pooling(9.3).

9 Working with Large Images

9.1 Feature extraction using convolution

9.1.1 Overview

In the previous exercises, you worked through problems which involved images that were relatively low in resolution, such as small image patches and small images of hand-written digits. In this section, we will develop methods which will allow us to scale up these methods to more realistic datasets that have larger images.

9.1.2 Fully Connected Networks

In the sparse autoencoder, one design choice that we had made was to “fully connect" all the hidden units to all the input units. On the relatively small images that we were working with (e.g., 8×8 patches for the sparse autoencoder assignment, 28×28 images for the MNIST dataset), it was computationally feasible to learn features on the entire image. However, with larger images (e.g., 96×96 images) learning features that span the entire image (fully connected networks) is very computationally expensive–you would have about 10^4 input units, and assuming you want to learn 100 features, you would have on the order of 10^6 parameters to learn. The feedforward and backpropagation computations would also be about 102 times slower, compared to 28×28 images.

9.1.3 Locally Connected Networks

One simple solution to this problem is to restrict the connections between the hidden units and the input units, allowing each hidden unit to connect to only a small subset of the input units. Specifically, each hidden unit will connect to only a small contiguous region of pixels in the input. (For input modalities different than images, there is often also a natural way to select “contiguous groups" of input units to connect to a single hidden unit as well; for example, for audio, a hidden unit might be connected to only the input units corresponding to a certain time span of the input audio clip.)

This idea of having locally connected networks also draws inspiration from how the early visual system is wired up in biology. Specifically, neurons in the visual cortex have localized receptive fields (i.e., they respond only to stimuli in a certain location).

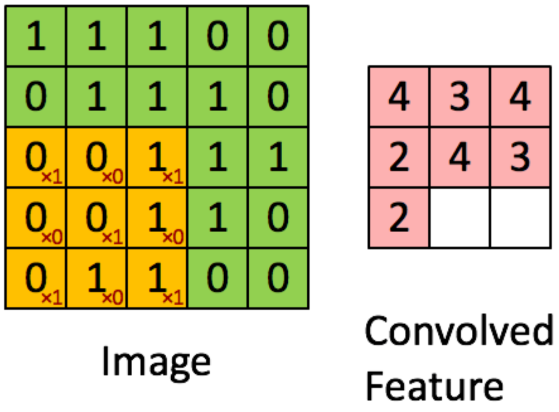
9.1.4 Convolutions

Natural images have the property of being stationary, meaning that the statistics of one part of the image are the same as any other part. This suggests that the features that we learn at one part of the image can also be applied to other parts of the image, and we can use the same features at all locations.

More precisely, having learned features over small (say 8×8) patches sampled randomly from the larger image, we can then apply this learned 8×8 feature detector anywhere in the image. Specifically, we can take the learned 8×8 features and convolve them with the larger image, thus obtaining a different feature activation value at each location in the image.

To give a concrete example, suppose you have learned features on 8×8 patches sampled from a 96×96 image. Suppose further this was done with an autoencoder that has 100 hidden units.

To get the convolved features, for every 8×8 region of the 96×96 image, that is, the 8×8 regions starting at $(1, 1), (1, 2), \ldots (89, 89)$, you would extract the 8×8 patch, and run it through your trained sparse autoencoder to get the feature activations. This would result in 100 sets 89×89 convolved features.



Formally, given some large $r \times c$ images x_{large}, we first train a sparse autoencoder on small $a \times b$ patches xsmall sampled from these images, learning k features $f = \sigma(W^{(1)}x_{small} + b^{(1)})$ (where σ is the sigmoid function), given by the weights $W^{(1)}$ and biases $b^{(1)}$ from the visible units to the hidden units. For every $a \times b$ patch xs in the large image, we compute $f_s = \sigma(W^{(1)}x_s + b^{(1)})$, giving us $f_{convolved}$, a $k \times (r - a + 1) \times (c - b + 1)$ array of convolved features.

In the next section, we further describe how to "pool" these features together to get even better features for classification.

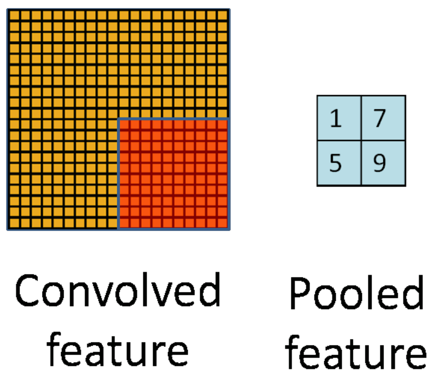
9.2 Pooling

9.2.1 Pooling: Overview

After obtaining features using convolution, we would next like to use them for classification. In theory, one could use all the extracted features with a classifier such as a softmax classifier, but this can be computationally challenging. Consider for instance images of size 96×96 pixels, and suppose we have learned 400 features over 8×8 inputs. Each convolution results in an output of size $(96 - 8 + 1) * (96 - 8 + 1) = 7921$, and since we have 400 features, this results in a vector of $89^2 * 400 = 3,168,400$ features per example. Learning a classifier with inputs having 3+ million features can be unwieldy, and can also be prone to over-fitting.

To address this, first recall that we decided to obtain convolved features because images have the "stationarity" property, which implies that features that are useful in one region are also likely to be useful for other regions. Thus, to describe a large image, one natural approach is to aggregate statistics of these features at various locations. For example, one could compute the mean (or max) value of a particular feature over a region of the image. These summary statistics are much lower in dimension (compared to using all of the extracted features) and can also improve results (less over-fitting). We aggregation operation is called this operation pooling, or sometimes mean pooling or max pooling (depending on the pooling operation applied).

The following image shows how pooling is done over 4 non-overlapping regions of the image.



9.2.2 Pooling for Invariance

If one chooses the pooling regions to be contiguous areas in the image and only pools features generated from the same (replicated) hidden units. Then, these pooling units will then be translation invariant. This means that the same (pooled) feature will be active even when the image undergoes (small) translations. Translation-invariant features are often desirable; in many tasks (e.g., object detection, audio recognition), the label of the example (image) is the same even when the image is translated. For example, if you were to take an MNIST digit and translate it left or right, you would want your classifier to still accurately classify it as the same digit regardless of its final position.

9.2.3 Formal description

Formally, after obtaining our convolved features as described earlier, we decide the size of the region, say $m \times n$ to pool our convolved features over. Then, we divide our convolved features into disjoint $m \times n$ regions, and take the mean (or maximum) feature activation over these regions to obtain the pooled convolved features. These pooled features can then be used for classification.

9.3 Exercise:Convolution and Pooling

In this exercise you will use the features you learned on 8×8 patches sampled from images from the STL-10 dataset in the earlier exercise on linear decoders(8.2) for classifying images from a reduced STL-10 dataset applying convolution(9.1) and pooling(9.2). The reduced STL-10 dataset comprises 64×64 images from 4 classes (airplane, car, cat, dog).

In the file cnn_exercise.zip we have provided some starter code. You should write your code at the places indicated "YOUR CODE HERE" in the files.

For this exercise, you will need to modify cnnConvolve.m and cnnPool.m.

9.3.1 Dependencies

The following additional files are required for this exercise:

- A subset of the STL10 Dataset (stlSubset.zip)
- Starter Code (cnn_exercise.zip)

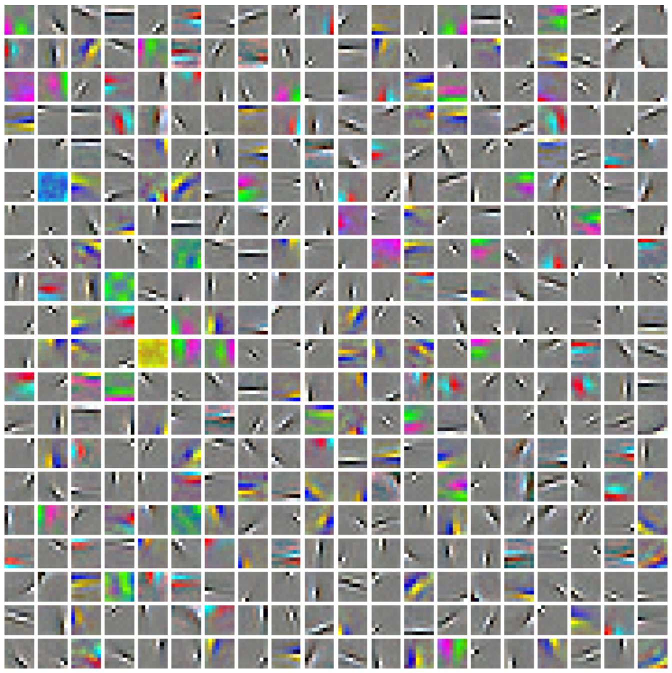
You will also need:

- `sparseAutoencoderLinear.m` or your saved features from Exercise:Learning color features with Sparse Autoencoders (8.2)
- `feedForwardAutoencoder.m` (and related functions) from Exercise:Self-Taught Learning (6.2)
- `softmaxTrain.m` (and related functions) from Exercise:Softmax Regression (5.2)

If you have not completed the exercises listed above, we strongly suggest you complete them first.

9.3.2 Step 1: Load learned features

In this step, you will use the features from Exercise:Learning color features with Sparse Autoencoders(8.2). If you have completed that exercise, you can load the color features that were previously saved. To verify that the features are good, the visualized features should look like the following:



9.3.3 Step 2: Implement and test convolution and pooling

In this step, you will implement convolution and pooling, and test them on a small part of the data set to ensure that you have implemented these two functions correctly. In the next step, you will actually convolve and pool the features with the STL-10 images.

9.3.4 Step 2a: Implement convolution

Implement convolution, as described in feature extraction using convolution(9.1), in the function cnnConvolve in cnnConvolve.m. Implementing convolution is somewhat involved, so we will guide you through the process below.

First, we want to compute $\sigma(Wx_{(r,c)} + b)$ for all valid (r,c) (valid meaning that the entire 8×8 patch is contained within the image; this is as opposed to a full convolution, which allows the patch to extend outside the image, with the area outside the image assumed to be 0), where W and b are the learned weights and biases from the input layer to the hidden layer, and $x_{(r,c)}$ is the 8×8 patch with the upper left corner at (r,c). To accomplish this, one naive method is to loop over all such patches and compute $\sigma(Wx_{(r,c)} + b)$ for each of them; while this is fine in theory, it can very slow. Hence, we usually use Matlab's built in convolution functions, which are well optimized.

Observe that the convolution above can be broken down into the following three small steps. First, compute $Wx_{(r,c)}$ for all (r,c). Next, add b to all the computed values. Finally, apply the sigmoid function to the resulting values. This doesn't seem to buy you anything, since the first step still requires a loop. However, you can replace the loop in the first step with one of MATLAB's optimized convolution functions, `conv2`, speeding up the process significantly.

However, there are two important points to note in using `conv2`.

First, `conv2` performs a 2-D convolution, but you have 5 “dimensions" - image number, feature number, row of image, column of image, and (color) channel of image - that you want to convolve over. Because of this, you will have to convolve each feature and image channel separately for each image, using the row and column of the image as the 2 dimensions you convolve over. This means that you will need three outer loops over the image number `imageNum`, feature number `featureNum`, and the channel number of the image channel. Inside the three nested for-loops, you will perform a `conv2` 2-D convolution, using the weight matrix for the `featureNum`-th feature and `channel`-th channel, and the image matrix for the `imageNum`-th image.

Second, because of the mathematical definition of convolution, the feature matrix must be “flipped" before passing it to `conv2`. The following implementat

Implementation tip: Using `conv2` and `convn`

Because the mathematical definition of convolution involves “flipping" the matrix to convolve with (reversing its rows and its columns), to use MATLAB's convolution functions, you must first “flip" the weight matrix so that when MATLAB “flips" it according to the mathematical definition the entries will be at the correct place. For example, suppose you wanted to convolve two matrices `image` (a large image) and `W` (the feature) using `conv2(image, W)`, and `W` is a 3×3 matrix as below:

$$W = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix}$$

If you use `conv2(image, W)`, MATLAB will first “flip" `W`, reversing its rows and columns, before convolving `W` with `image`, as below:

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix} \xrightarrow{flip} \begin{pmatrix} 9 & 8 & 7 \\ 6 & 5 & 4 \\ 3 & 2 & 1 \end{pmatrix}$$

If the original layout of `W` was correct, after flipping, it would be incorrect. For the layout to be correct after flipping, you will have to flip W before passing it into `conv2`, so that after MATLAB flips `W` in `conv2`, the layout will be correct. For `conv2`, this means reversing the rows and columns, which can be done with flipud and fliplr, as shown below:

```
% Flip W for use in conv2
W = flipud(fliplr(W));
```

Next, to each of the `convolvedFeatures`, you should then add `b`, the corresponding bias for the `featureNum`-th feature.

However, there is one additional complication. If we had not done any preprocessing of the input patches, you could just follow the procedure as described above, and apply the sigmoid function to obtain the convolved features, and we'd be done. However, because you preprocessed the patches before learning features on them, you must also apply the same preprocessing steps to the convolved patches to get the correct feature activations.

In particular, you did the following to the patches:

- subtract the mean patch, `meanPatch` to zero the mean of the patches
- ZCA whiten using the whitening matrix `ZCAWhite`.

These same three steps must also be applied to the input image patches.

Taking the preprocessing steps into account, the feature activations that you should compute is $\sigma(W(T(x-\bar{x}))+b)$, where T is the whitening matrix and $\bar{x}$ is the mean patch. Expanding this, you obtain $\sigma(WTx - WT\bar{x} + b)$, which suggests that you should convolve the images with WT rather than W as earlier, and you should add $(b - WT\bar{x})$, rather than just b to `convolvedFeatures`, before finally applying the sigmoid function.

9.3.5 Step 2b: Check your convolution

We have provided some code for you to check that you have done the convolution correctly. The code randomly checks the convolved values for a number of (feature, row, column) tuples by computing the feature activations using `feedForwardAutoencoder` for the selected features and patches directly using the sparse autoencoder.

9.3.6 Step 2c: Pooling

Implement pooling in the function `cnnPool` in `cnnPool.m`. You should implement mean pooling (i.e., averaging over feature responses) for this part.

9.3.7 Step 2d: Check your pooling

We have provided some code for you to check that you have done the pooling correctly. The code runs `cnnPool` against a test matrix to see if it produces the expected result.

9.3.8 Step 3: Convolve and pool with the dataset

In this step, you will convolve each of the features you learned with the full 64×64 images from the STL-10 dataset to obtain the convolved features for both the training and test sets. You will then pool the convolved features to obtain the pooled features for both training and test sets. The pooled features for the training set will be used to train your classifier, which you can then test on the test set.

Because the convolved features matrix is very large, the code provided does the convolution and pooling 50 features at a time to avoid running out of memory.

9.3.9 Step 4: Use pooled features for classification

In this step, you will use the pooled features to train a softmax classifier to map the pooled features to the class labels. The code in this section uses `softmaxTrain` from the softmax exercise to train a softmax classifier on the pooled features for 500 iterations, which should take around a few minutes.

9.3.10 Step 5: Test classifier

Now that you have a trained softmax classifier, you can see how well it performs on the test set. These pooled features for the test set will be run through the softmax classifier, and the accuracy of the predictions will be computed. You should expect to get an accuracy of around 80%.

10 Sparse Coding

10.1 Sparse Coding

Sparse coding is a class of unsupervised methods for learning sets of over-complete bases to represent data efficiently. The aim of sparse coding is to find a set of basis vectors ϕ_i such that we can represent an input vector x as a linear combination of these basis vectors:

$$\mathsf{x} = \sum_{i=1}^{k} a_i \phi_i \tag{101}$$

While techniques such as Principal Component Analysis (PCA) allow us to learn a complete set of basis vectors efficiently, we wish to learn an over-complete set of basis vectors to represent input vectors $\mathsf{x} \in \mathbb{R}^n$ (i.e. such that $k > n$). The advantage of having an over-complete basis is that our basis vectors are better able to capture structures and patterns inherent in the input data. However, with an over-complete basis, the coefficients ai are no longer uniquely determined by the input vector x. Therefore, in sparse coding, we introduce the additional criterion of sparsity to resolve the degeneracy introduced by over-completeness.

Here, we define sparsity as having few non-zero components or having few components not close to zero. The requirement that our coefficients a_i be sparse means that given a input vector, we would like as few of our coefficients to be far from zero as possible. The choice of sparsity as a desired characteristic of our representation of the input data can be motivated by the observation that most sensory data such as natural images may be described as the superposition of a small number of atomic elements such as surfaces or edges. Other justifications such as comparisons to the properties of the primary visual cortex have also been advanced.

We define the sparse coding cost function on a set of m input vectors as

$$\text{minimize}_{a_i^{(j)}, \phi_i} \sum_{j=1}^{m} \left|\left| \mathsf{x}^{(j)} - \sum_{i=1}^{k} a_i^{(j)} \phi_i \right|\right|^2 + \lambda \sum_{i=1}^{k} S(a_i^{(j)}) \tag{102}$$

where $S(\cdot)$ is a sparsity cost function which penalizes ai for being far from zero. We can interpret the first term of the sparse coding objective as a reconstruction term which tries to force the algorithm to provide a good representation of x and the second term as a sparsity penalty which forces our representation of x to be sparse. The constant λ is a scaling constant to determine the relative importance of these two contributions.

Although the most direct measure of sparsity is the "L_0" norm ($S(a_i) = \mathbf{1}(|a_i| > 0)$), it is non-differentiable and difficult to optimize in general. In practice, common choices for the sparsity cost $S(\cdot)$ are the L_1 penalty $S(a_i) = |a_i|_1$ and the log penalty $S(a_i) = \log(1 + a_i^2)$.

In addition, it is also possible to make the sparsity penalty arbitrarily small by scaling down a_i and scaling ϕ_i up by some large constant. To prevent this from happening, we will constrain $||\phi||^2$ to be less than some constant C. The full sparse coding cost function including our constraint on ϕ is

$$\begin{array}{rc} \text{minimize}_{a_i^{(j)}, \phi_i} & \sum_{j=1}^{m} \left|\left| \mathsf{x}^{(j)} - \sum_{i=1}^{k} a_i^{(j)} \phi_i \right|\right|^2 + \lambda \sum_{i=1}^{k} S(a_i^{(j)}) \\ \text{subject to} & ||\phi_i||^2 \leq C, \forall i = 1, ..., k \end{array}$$

10.1.1 Probabilistic Interpretation

[Based on Olshausen and Field 1996]

So far, we have considered sparse coding in the context of finding a sparse, over-complete set of basis vectors to span our input space. Alternatively, we may also approach sparse coding from a probabilistic perspective as a generative model.

Consider the problem of modelling natural images as the linear superposition of k independent source features ϕ_i with some additive noise ν:

$$\mathsf{x} = \sum_{i=1}^{k} a_i \phi_i + \nu(\mathsf{x}) \tag{103}$$

Our goal is to find a set of basis feature vectors ϕ such that the distribution of images $P(\mathsf{x} \mid \phi)$ is as close as possible to the empirical distribution of our input data $P^*(\mathsf{x})$. One method of doing so is to minimize the KL divergence between $P^*(\mathsf{x})$ and $P(\mathsf{x} \mid \phi)$ where the KL divergence is defined as:

$$D(P^*(\mathsf{x})||P(\mathsf{x} \mid \phi)) = \int P^*(\mathsf{x}) \log \left(\frac{P^*(\mathsf{x})}{P(\mathsf{x} \mid \phi)} \right) d\mathsf{x} \tag{104}$$

Since the empirical distribution $P^*(\mathsf{x})$ is constant across our choice of ϕ, this is equivalent to maximizing the log-likelihood of $P(\mathsf{x} \mid \phi)$.

Assuming ν is Gaussian white noise with variance σ^2, we have that

$$P(\mathsf{x} \mid \mathsf{a}, \phi) = \frac{1}{Z} \exp \left(- \frac{(\mathsf{x} - \sum_{i=1}^{k} a_i \phi_i)^2}{2\sigma^2} \right) \tag{105}$$

In order to determine the distribution $P(\mathsf{x} \mid \phi)$, we also need to specify the prior distribution $P(\mathsf{a})$. Assuming the independence of our source features, we can factorize our prior probability as

$$P(\mathsf{a}) = \prod_{i=1}^{k} P(a_i) \tag{106}$$

At this point, we would like to incorporate our sparsity assumption – the assumption that any single image is likely to be the product of relatively few source features. Therefore, we would like the probability distribution of a_i to be peaked at zero and have high kurtosis. A convenient parameterization of the prior distribution is

$$P(a_i) = \frac{1}{Z} \exp(-\beta S(a_i)) \tag{107}$$

Where $S(a_i)$ is a function determining the shape of the prior distribution.

Having defined $P(\mathsf{x} \mid \mathsf{a}, \phi)$ and $P(\mathsf{a})$, we can write the probability of the data x under the model defined by ϕ as

$$P(\mathsf{x} \mid \phi) = \int P(\mathsf{x} \mid \mathsf{a}, \phi) P(\mathsf{a}) d\mathsf{a} \tag{108}$$

and our problem reduces to finding

$$\phi^* = \text{argmax}_\phi < \log(P(\mathsf{x} \mid \phi)) > \tag{109}$$

Where $< \cdot >$ denotes expectation over our input data.

Unfortunately, the integral over a to obtain $P(\mathsf{x} \mid \phi)$ is generally intractable. We note though that if the distribution of $P(\mathsf{x} \mid \phi)$ is sufficiently peaked (w.r.t. a), we can approximate its integral with the maximum value of $P(\mathsf{x} \mid \phi)$ and obtain a approximate solution

$$\phi^{*'} = \text{argmax}_\phi < \max_{\mathsf{a}} \log(P(\mathsf{x} \mid \phi)) > \tag{110}$$

As before, we may increase the estimated probability by scaling down a_i and scaling up ϕ (since $P(a_i)$ peaks about zero) , we therefore impose a norm constraint on our features ϕ to prevent this.

Finally, we can recover our original cost function by defining the energy function of this linear generative model

$$\begin{aligned} E\left(\mathsf{x}, \mathsf{a} \mid \phi\right) &:= -\log\left(P(\mathsf{x} \mid \phi, \mathsf{a})\, P(\mathsf{a})\right) \\ &= \sum_{j=1}^{m} \left|\left| \mathsf{x}^{(j)} - \sum_{i=1}^{k} a_i^{(j)} \phi_i \right|\right|^2 + \lambda \sum_{i=1}^{k} S(a_i^{(j)}) \end{aligned}$$

where $\lambda = 2\sigma^2\beta$ and irrelevant constants have been hidden. Since maximizing the log-likelihood is equivalent to minimizing the energy function, we recover the original optimization problem:

$$\phi^*, \mathsf{a}^* = \text{argmin}_{\phi, \mathsf{a}} \sum_{j=1}^{m} \left|\left| \mathsf{x}^{(j)} - \sum_{i=1}^{k} a_i^{(j)} \phi_i \right|\right|^2 + \lambda \sum_{i=1}^{k} S(a_i^{(j)}) \tag{111}$$

Using a probabilistic approach, it can also be seen that the choices of the L_1 penalty $|a_i|_1$ and the log penalty $\log(1 + a_i^2)$ for $S(\cdot)$ correspond to the use of the Laplacian $P(a_i) \propto \exp\left(-\beta |a_i|\right)$ and the Cauchy prior $P(a_i) \propto \frac{\beta}{1+a_i^2}$ respectively.

10.1.2 Learning

Learning a set of basis vectors ϕ using sparse coding consists of performing two separate optimizations, the first being an optimization over coefficients a_i for each training example x and the second an optimization over basis vectors ϕ across many training examples at once.

Assuming an L_1 sparsity penalty, learning $a_i^{(j)}$ reduces to solving a L_1 regularized least squares problem which is convex in $a_i^{(j)}$ for which several techniques have been developed (convex optimization software such as CVX can also be used to perform L_1 regularized least squares). Assuming a differentiable $S(\cdot)$ such as the log penalty, gradient-based methods such as conjugate gradient methods can also be used.

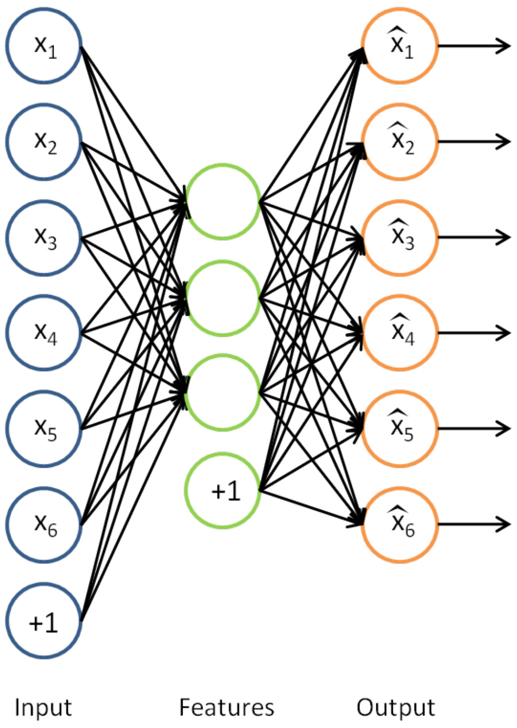
Learning a set of basis vectors with a L_2 norm constraint also reduces to a least squares problem with quadratic constraints which is convex in ϕ. Standard convex optimization software (e.g. CVX) or other iterative methods can be used to solve for ϕ although significantly more efficient methods such as solving the Lagrange dual have also been developed.

As described above, a significant limitation of sparse coding is that even after a set of basis vectors have been learnt, in order to "encode" a new data example, optimization must be performed to obtain the required coefficients. This significant "runtime" cost means that sparse coding is computationally expensive to implement even at test time especially compared to typical feedforward architectures.

10.2 Sparse Coding: Autoencoder Interpretation

10.2.1 Sparse Coding

In the sparse autoencoder, we tried to learn a set of weights W (and associated biases b) that would give us sparse features $\sigma(Wx + b)$ useful in reconstructing the input x.



Sparse coding can be seen as a modification of the sparse autoencoder method in which we try to learn the set of features for some data "directly". Together with an associated basis for transforming the learned features from the feature space to the data space, we can then reconstruct the data from the learned features.

Formally, in sparse coding, we have some data x we would like to learn features on. In particular, we would like to learn s, a set of sparse features useful for representing the data, and A, a basis for transforming the features from the feature space to the data space. Our objective function is hence:

$$J(A, s) = \|As - x\|_2^2 + \lambda\|s\|_1$$

(If you are unfamiliar with the notation, $\|x\|_k$ refers to the L_k norm of the x which is equal to $\left(\sum \left|x_i^k\right|\right)^{\frac{1}{k}}$. The L_2 norm is the familiar Euclidean norm, while the L_1 norm is the sum of absolute values of the elements of the vector)

The first term is the error in reconstructing the data from the features using the basis, and the second term is a sparsity penalty term to encourage the learned features to be sparse.

However, the objective function as it stands is not properly constrained - it is possible to reduce the sparsity cost (the second term) by scaling A by some constant and scaling s by the inverse of the same constant, without changing the error. Hence, we include the additional constraint that that for every column A_j of A, $A_j^T A_j \leq 1$. Our problem is thus:

$$\begin{array}{ll} \text{minimize} & \|As - x\|_2^2 + \lambda\|s\|_1 \\ \text{s.t.} & A_j^T A_j \leq 1 \; \forall j \end{array}$$

Unfortunately, the objective function is non-convex, and hence impossible to optimize well using gradient-based methods. However, given A, the problem of finding s that minimizes $J(A, s)$ is convex. Similarly, given s, the problem of finding A that minimizes $J(A, s)$ is also convex. This suggests that we might try alternately optimizing for A for a fixed s, and then optimizing for s given a fixed A. It turns out that this works quite well in practice.

However, the form of our problem presents another difficulty - the constraint that $A_j^T A_j \leq 1 \; \forall j$ cannot be enforced using simple gradient-based methods. Hence, in practice, this constraint is weakened to a “weight decay" term designed to keep the entries of A small. This gives us a new objective function:

$$J(A, s) = \|As - x\|_2^2 + \lambda\|s\|_1 + \gamma\|A\|_2^2$$

(note that the third term, $\|A\|_2^2$ is simply the sum of squares of the entries of A, or $\sum_r \sum_c A_{rc}^2$)

This objective function presents one last problem - the L_1 norm is not differentiable at 0, and hence poses a problem for gradient-based methods. While the problem can be solved using other non-gradient descent-based methods, we will “smooth out" the L_1 norm using an approximation which will allow us to use gradient descent. To “smooth out" the L_1 norm, we use $\sqrt{x^2 + \epsilon}$ in place of $|x|$, where ϵ is a “smoothing parameter" which can also be interpreted as a sort of “sparsity parameter" (to see this, observe that when ϵ is large compared to x, the $x + \epsilon$ is dominated by ϵ, and taking the square root yields approximately $\sqrt{\epsilon}$). This “smoothing" will come in handy later when considering topographic sparse coding below.

Our final objective function is hence:

$$J(A, s) = \|As - x\|_2^2 + \lambda\sqrt{s^2 + \epsilon} + \gamma\|A\|_2^2$$

(where $\sqrt{s^2 + \epsilon}$ is shorthand for $\sum_k \sqrt{s_k^2 + \epsilon}$)

This objective function can then be optimized iteratively, using the following procedure:

1. Initialize A randomly
2. Repeat until convergence
   (a) Find the s that minimizes $J(A, s)$ for the A found in the previous step
   (b) Solve for the A that minimizes $J(A, s)$ for the s found in the previous step

Observe that with our modified objective function, the objective function $J(A, s)$ given s, that is $J(A; s) = \|As - x\|_2^2 + \gamma\|A\|_2^2$ (the L_1 term in s can be omitted since it is not a function of A) is simply a quadratic term in A, and hence has an easily derivable analytic solution in A. A quick way to derive this solution would be to use matrix calculus - some pages about matrix calculus can be found in the useful links section(C.3). Unfortunately, the objective function given A does not have a similarly nice analytic solution, so that minimization step will have to be carried out using gradient descent or similar optimization methods.

In theory, optimizing for this objective function using the iterative method as above should (eventually) yield features (the basis vectors of A) similar to those learned using the sparse autoencoder. However, in practice, there are quite a few tricks required for better convergence of the algorithm, and these tricks are described in greater detail in the later section on sparse coding in practice(10.2.3). Deriving the gradients for the objective function may be slightly tricky as well, and using matrix calculus or using the backpropagation intuition(B.2) can be helpful.

10.2.2 Topographic sparse coding

With sparse coding, we can learn a set of features useful for representing the data. However, drawing inspiration from the brain, we would like to learn a set of features that are “orderly" in some manner. For instance, consider visual features. As suggested earlier, the V1 cortex of the brain contains neurons which detect edges at particular orientations. However, these neurons are also organized into hypercolumns in which adjacent neurons detect edges at similar orientations. One neuron could detect a horizontal edge, its neighbors edges oriented slightly off the horizontal, and moving further along the hypercolumn, the neurons detect edges oriented further off the horizontal.

Inspired by this example, we would like to learn features which are similarly “topographically ordered". What does this imply for our learned features? Intuitively, if “adjacent" features are “similar", we would expect that if one feature is activated, its neighbors will also be activated to a lesser extent.

Concretely, suppose we (arbitrarily) organized our features into a square matrix. We would then like adjacent features in the matrix to be similar. The way this is accomplished is to group these adjacent features together in the smoothed L1 penalty, so that instead of say $\sqrt{s_{1,1}^2 + \epsilon}$, we use say $\sqrt{s_{1,1}^2 + s_{1,2}^2 + s_{1,3}^2 + s_{2,1}^2 + s_{2,2}^2 + s_{3,2}^2 + s_{3,1}^2 + s_{3,2}^2 + s_{3,3}^2 + \epsilon}$ instead, if we group in 3×3 regions. The grouping is usually overlapping, so that the 3×3 region starting at the 1st row and 1st column is one group, the 3×3 region starting at the 1st row and 2nd column is another group, and so on. Further, the grouping is also usually done wrapping around, as if the matrix were a torus, so that every feature is counted an equal number of times.

Hence, in place of the smoothed L1 penalty, we use the sum of smoothed L1 penalties over all the groups, so our new objective function is:

$$J(A, s) = \|As - x\|_2^2 + \lambda \sum_{\text{all groups} g} \sqrt{\left(\sum_{\text{all} s \in g} s^2 \right) + \epsilon} + \gamma \|A\|_2^2$$

In practice, the “grouping" can be accomplished using a “grouping matrix" V, such that the rth row of V indicates which features are grouped in the rth group, so $V_{r,c} = 1$ if group r contains feature c. Thinking of the grouping as being achieved by a grouping matrix makes the computation of the gradients more intuitive. Using this grouping matrix, the objective function can be rewritten as:

$$J(A, s) = \|As - x\|_2^2 + \lambda \sum \sqrt{Vss^T + \epsilon} + \gamma \|A\|_2^2$$

(where $\sum \sqrt{Vss^T + \epsilon}$ is $\sum_r \sum_c D_{r,c}$ if we let $D = \sqrt{Vss^T + \epsilon}$)

This objective function can be optimized using the iterated method described in the earlier section. Topographic sparse coding will learn features similar to those learned by sparse coding, except that the features will now be “ordered" in some way.

10.2.3 Sparse coding in practice

As suggested in the earlier sections, while the theory behind sparse coding is quite simple, writing a good implementation that actually works and converges reasonably quickly to good optima requires a bit of finesse.

Recall the simple iterative algorithm proposed earlier:

1. Initialize A randomly
2. Repeat until convergence
   (a) Find the s that minimizes $J(A, s)$ for the A found in the previous step
   (b) Solve for the A that minimizes $J(A, s)$ for the s found in the previous step

It turns out that running this algorithm out of the box will not produce very good results, if any results are produced at all. There are two main tricks to achieve faster and better convergence:

1. Batching examples into “mini-batches"
2. Good initialization of s

Batching examples into mini-batches

If you try running the simple iterative algorithm on a large dataset of say 10 000 patches at one go, you will find that each iteration takes a long time, and the algorithm may hence take a long time to converge. To increase the rate of convergence, you can instead run the algorithm on mini-batches instead. To do this, instead of running the algorithm on all 10 000 patches, in each iteration, select a mini-batch - a (different) random subset of say 2000 patches from the 10 000 patches - and run the algorithm on that mini-batch for the iteration instead. This accomplishes two things - firstly, it speeds up each iteration, since now each iteration is operating on 2000 rather than 10 000 patches; secondly, and more importantly, it increases the rate of convergence (TODO: explain why).

Good initialization of s

Another important trick in obtaining faster and better convergence is good initialization of the feature matrix s before using gradient descent (or other methods) to optimize for the objective function for s given A. In practice, initializing s randomly at each iteration can result in poor convergence unless a good optima is found for s before moving on to optimize for A. A better way to initialize s is the following:

1. Set $s \leftarrow W^T x$ (where x is the matrix of patches in the mini-batch)
2. For each feature in s (i.e. each column of s), divide the feature by the norm of the corresponding basis vector in A. That is, if $s_{r,c}$ is the rth feature for the cth example, and A_c is the cth basis vector in A, then set $s_{r,c} \leftarrow \frac{s_{r,c}}{\|A_c\|}$.

Very roughly and informally speaking, this initialization helps because the first step is an attempt to find a good s such that $Ws \approx x$, and the second step “normalizes" s in an attempt to keep the sparsity penalty small. It turns out that initializing s using only one but not both steps results in poor performance in practice. (TODO: a better explanation for why this initialization helps?)

The practical algorithm

With the above two tricks, the algorithm for sparse coding then becomes:

1. Initialize A randomly
2. Repeat until convergence
   (a) Select a random mini-batch of 2000 patches

(b) Initialize s as described above

(c) Find the s that minimizes $J(A, s)$ for the A found in the previous step

(d) Solve for the A that minimizes $J(A, s)$ for the s found in the previous step

With this method, you should be able to reach a good local optima relatively quickly.

10.3 Exercise:Sparse Coding

In this exercise, you will implement sparse coding(??) and topographic sparse coding(10.2.2) on black-and-white natural images.

In the file sparse_coding_exercise.zip we have provided some starter code. You should write your code at the places indicated "YOUR CODE HERE" in the files.

For this exercise, you will need to modify sparseCodingWeightCost.m, sparseCodingFeatureCost.m and sparseCodingExercise.m.

10.3.1 Dependencies

You will need:

- `computeNumericalGradient.m` from Exercise:Sparse Autoencoder (2.7)
- `display_network.m` from Exercise:Sparse Autoencoder (2.7)

If you have not completed the exercise listed above, we strongly suggest you complete it first.

10.3.2 Step 0: Initialization

In this step, we initialize some parameters used for the exercise.

10.3.3 Step 1: Sample patches

In this step, we sample some patches from the `IMAGES.mat` dataset comprising 10 black-and-white pre-whitened natural images.

10.3.4 Step 2: Implement and check sparse coding cost functions

In this step, you should implement the two sparse coding cost functions:

1. `sparseCodingWeightCost` in `sparseCodingWeightCost.m`, which is used for optimizing the weight cost given the features
2. `sparseCodingFeatureCost` in `sparseCodingFeatureCost.m`, which is used for optimizing the feature cost given the weights

Each of these functions should compute the appropriate cost and gradient. You may wish to implement the non-topographic version of sparseCodingFeatureCost first, ignoring the grouping matrix and assuming that none of the features are grouped. You can then extend this to the topographic version later. Alternatively, you may implement the topographic version directly - using the non-topographic version will then involve setting the grouping matrix to the identity matrix.

Once you have implemented these functions, you should check the gradients numerically.

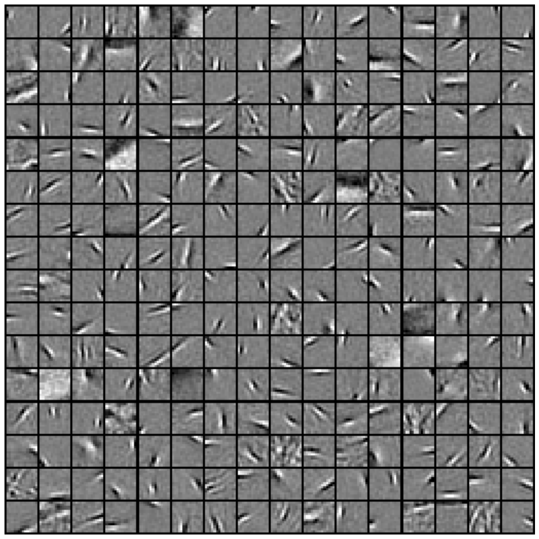
Implementation tip - gradient checking the feature cost. One particular point to note is that when checking the gradient for the feature cost, `epsilon` should be set to a larger value, for instance `1e-2` (as has been done for you in the checking code provided), to ensure that checking the gradient numerically makes sense. This is necessary because as `epsilon` becomes smaller, the function `sqrt(x + epsilon)` becomes “sharper" and more “pointed", making the numerical gradient computed near 0 less and less accurate. To see this, consider what would happen if the numerical gradient was computed by using a point with x less than 0 and a point with x greater than 0 - the computed numerical slope would be wildly inaccurate.

10.3.5 Step 3: Iterative optimization

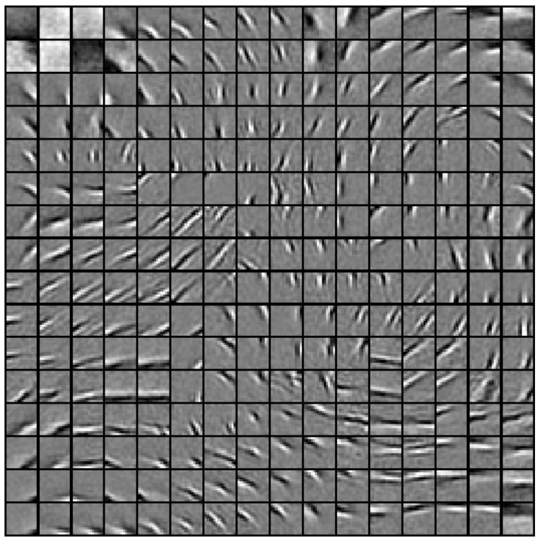
In this step, you will iteratively optimize for the weights and features to learn a basis for the data, as described in the section on sparse coding(10.2). Mini-batching and initialization of the features s has already been done for you. However, you need to still need to fill in the analytic solution to the the optimization problem with respect to the weight matrix, given the feature matrix.

Once that is done, you should check that your solution is correct using the given checking code, which checks that the gradient at the point determined by your analytic solution is close to 0. Once your solution has been verified, comment out the checking code, and run the iterative optimization code. 200 iterations should take less than 45 minutes to run, and by 100 iterations you should be able to see bases that look like edges, similar to those you learned in the sparse autoencoder exercise(2.7).

For the non-topographic case, these features will not be “ordered", and will look something like the following:



For the topographic case, the features will be “ordered topographically", and will look something like the following:



11 ICA Style Models

11.1 Independent Component Analysis

11.1.1 Introduction

If you recall, in sparse coding(10), we wanted to learn an over-complete basis for the data. In particular, this implies that the basis vectors that we learn in sparse coding will not be linearly independent. While this may be desirable in certain situations, sometimes we want to learn a linearly independent basis for the data. In independent component analysis (ICA), this is exactly what we want to do. Further, in ICA, we want to learn not just any linearly independent basis, but an orthonormal basis for the data. (An orthonormal basis is a basis $(\phi_1, \dots \phi_n)$ such that $\phi_i \cdot \phi_j = 0$ if $i \neq j$ and 1 if $i = j$).

Like sparse coding, independent component analysis has a simple mathematical formulation. Given some data x, we would like to learn a set of basis vectors which we represent in the columns of a matrix W, such that, firstly, as in sparse coding, our features are sparse; and secondly, our basis is an orthonormal basis. (Note that while in sparse coding, our matrix A was for mapping features s to raw data, in independent component analysis, our matrix W works in the opposite direction, mapping raw data x to features instead). This gives us the following objective function:

$$J(W) = \|Wx\|_1$$

This objective function is equivalent to the sparsity penalty on the features s in sparse coding, since Wx is precisely the features that represent the data. Adding in the orthonormality constraint gives us the full optimization problem for independent component analysis:

$$\begin{array}{rl} \text{minimize} & \|Wx\|_1 \\ \text{s.t.} & WW^T = I \end{array}$$

As is usually the case in deep learning, this problem has no simple analytic solution, and to make matters worse, the orthonormality constraint makes it slightly more difficult to optimize for the objective using gradient descent – every iteration of gradient descent must be followed by a step that maps the new basis back to the space of orthonormal bases (hence enforcing the constraint).

In practice, optimizing for the objective function while enforcing the orthonormality constraint (as described in Orthonormal ICA 11.1.2 section below) is feasible but slow. Hence, the use of orthonormal ICA is limited to situations where it is important to obtain an orthonormal basis (TODO: what situations).

11.1.2 Orthonormal ICA

The orthonormal ICA objective is:

$$\begin{array}{rl} \text{minimize} & \|Wx\|_1 \\ \text{s.t.} & WW^T = I \end{array}$$

Observe that the constraint $WW^T = I$ implies two other constraints.

Firstly, since we are learning an orthonormal basis, the number of basis vectors we learn must be less than the dimension of the input. In particular, this means that we cannot learn over-complete bases as we usually do in sparse coding(10.2).

Secondly, the data must be ZCA whitened(4.2) with no regularization (that is, with ϵ set to 0). (TODO Why must this be so?)

Hence, before we even begin to optimize for the orthonormal ICA objective, we must ensure that our data has been whitened, and that we are learning an under-complete basis.

Following that, to optimize for the objective, we can use gradient descent, interspersing gradient descent steps with projection steps to enforce the orthonormality constraint. Hence, the procedure will be as follows:

Repeat until done:
$W \leftarrow W - \alpha \nabla_W \|Wx\|_1$
$W \leftarrow \text{proj}_U W$ where U is the space of matrices satisfying $WW^T = I$

In practice, the learning rate α is varied using a line-search algorithm to speed up the descent, and the projection step is achieved by setting $W \leftarrow (WW^T)^{-\frac{1}{2}} W$, which can actually be seen as ZCA whitening (TODO explain how it is like ZCA whitening).

11.1.3 Topographic ICA

Just like sparse coding(10.2), independent component analysis can be modified to give a topographic variant by adding a topographic cost term.

11.2 Exercise:Independent Component Analysis

In this exercise, you will implement Independent Component Analysis(11.1) on color images from the STL-10 dataset.

In the file independent_component_analysis_exercise.zip we have provided some starter code. You should write your code at the places indicated "YOUR CODE HERE" in the files.

For this exercise, you will need to modify `OrthonormalICACost.m` and `ICAExercise.m`.

11.2.1 Dependencies

You will need:

- `computeNumericalGradient.m` from Exercise:Sparse Autoencoder(2.7)
- `displayColorNetwork.m` from Exercise:Learning color features with Sparse Autoencoders(8.2)

The following additional file is also required for this exercise:

- Sampled 8×8 patches from the STL-10 dataset (stl10_patches_100k.zip)

If you have not completed the exercises listed above, we strongly suggest you complete them first.

11.2.2 Step 0: Initialization

In this step, we initialize some parameters used for the exercise.

11.2.3 Step 1: Sample patches

In this step, we load and use a portion of the 8×8 patches from the STL-10 dataset (which you first saw in the exercise on linear decoders 8.2).

11.2.4 Step 2: ZCA whiten patches

In this step, we ZCA whiten the patches as required by orthonormal ICA.

11.2.5 Step 3: Implement and check ICA cost functions

In this step, you should implement the ICA cost function: `orthonormalICACost` in `orthonormalICACos`
which computes the cost and gradient for the orthonormal ICA objective. Note that the orthonormality constraint is not enforced in the cost function. It will be enforced by a projection in the gradient descent step, which you will have to complete in step 4.

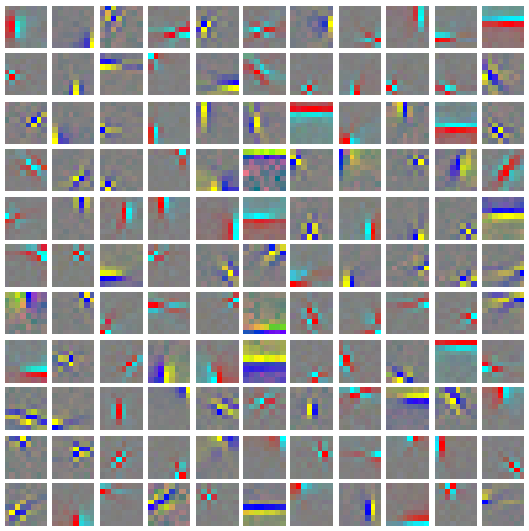
When you have implemented the cost function, you should check the gradients numerically.

Hint - if you are having difficulties deriving the gradients, you may wish to consult the page on deriving gradients using the backpropagation idea (B.2).

11.2.6 Step 4: Optimization

In step 4, you will optimize for the orthonormal ICA objective using gradient descent with backtracking line search (the code for which has already been provided for you. For more details on the backtracking line search, you may wish to consult the appendix(11.2.7) of this exercise). The orthonormality constraint should be enforced with a projection, which you should fill in.

Once you have filled in the code for the projection, check that it is correct by using the verification code provided. Once you have verified that your projection is correct, comment out the verification code and run the optimization. 1000 iterations of gradient descent should take less than 15 minutes, and produce a basis which looks like the following:



It is comparatively difficult to optimize for the objective while enforcing the orthonormality constraint using gradient descent, and convergence can be slow. Hence, in situations where an orthonormal basis is not required, other faster methods of learning bases (such as sparse coding 10.2) may be preferable.

11.2.7 Appendix

Backtracking line search

The backtracking line search used in the exercise is based off that in Convex Optimization by Boyd and Vandenbergh(http://www.stanford.edu/~boyd/cvxbook/). In the backtracking line search, given a descent direction $\vec{u}$ (in this exercise we use $\vec{u} = -\nabla f(\vec{x})$), we want to find a good step size t that gives us a steep descent. The general idea is to use a linear approximation (the first order Taylor approximation) to the function f at the current point $\vec{x}$, and to search for a step size t such that we can decrease the function's value by more than α times the decrease predicted by the linear approximation ($\alpha \in (0, 0.5)$. For more details, you may wish to consult the book.

However, it is not necessary to use the backtracking line search here. Gradient descent with a small step size, or backtracking to a step size so that the objective decreases is sufficient for this exercise.

A Using the MNIST Dataset

A.1 Introduction

The MNIST dataset is a dataset of handwritten digits, comprising 60 000 training examples and 10 000 test examples. The dataset can be downloaded from http://yann.lecun.com/exdb/mnist/.

A.2 Usage

The image and label data is stored in a binary format described on the website. For your convenience, we have provided two MATLAB helper functions for extracting the data. These functions are available at http://ufldl.stanford.edu/wiki/resources/mnistHelper.zip.

As an example of how to use these functions, you can check the images and labels using the following code:

```
% Change the filenames if you've saved the files under different names
% On some platforms, the files might be saved as
% train-images.idx3-ubyte / train-labels.idx1-ubyte
images = loadMNISTImages('train-images-idx3-ubyte');
labels = loadMNISTLabels('train-labels-idx1-ubyte');

% We are using display_network from the autoencoder code
display_network(images(:,1:100)); % Show the first 100 images
disp(labels(1:10));
```

B Miscellaneous Topics

B.1 Data Preprocessing

B.1.1 Overview

Data preprocessing plays a very important in many deep learning algorithms. In practice, many methods work best after the data has been normalized and whitened. However, the exact parameters for data preprocessing are usually not immediately apparent unless one has much experience working with the algorithms. In this page, we hope to demystify some of the preprocessing methods and also provide tips (and a “standard pipeline") for preprocessing data.

Tip: When approaching a dataset, the first thing to do is to look at the data itself and observe its properties. While the techniques here apply generally, you might want to opt to do certain things differently given your dataset. For example, one standard preprocessing trick is to subtract the mean of each data point from itself (also known as remove DC, local mean subtraction, subtractive normalization). While this makes sense for data such as natural images, it is less obvious for data where stationarity does not hold.

B.1.2 Data Normalization

A standard first step to data preprocessing is data normalization. While there are a few possible approaches, this step is usually clear depending on the data. The common methods for feature normalization are:

- Simple Rescaling
- Per-example mean subtraction (a.k.a. remove DC)
- Feature Standardization (zero-mean and unit variance for each feature across the dataset)

Simple Rescaling

In simple rescaling, our goal is to rescale the data along each data dimension (possibly independently) so that the final data vectors lie in the range $[0, 1]$ or $[-1, 1]$ (depending on your dataset). This is useful for later processing as many default parameters (e.g., `epsilon` in PCA-whitening) treat the data as if it has been scaled to a reasonable range.

Example: When processing natural images, we often obtain pixel values in the range [0,255]. It is a common operation to rescale these values to [0,1] by dividing the data by 255.

Per-example mean subtraction

If your data is stationary (i.e., the statistics for each data dimension follow the same distribution), then you might want to consider subtracting the mean-value for each example (computed per-example).

Example: In images, this normalization has the property of removing the average brightness (intensity) of the data point. In many cases, we are not interested in the illumination conditions of the image, but more so in the content; removing the average pixel value per data point makes sense here. Note: While this method is generally used for images, one might want to take more care when applying this to color images. In particular, the stationarity property does not generally apply across pixels in different color channels.

Feature Standardization

Feature standardization refers to (independently) setting each dimension of the data to have zero-mean and unit-variance. This is the most common method for normalization and is generally used widely (e.g., when working with SVMs, feature standardization is often recommended as a preprocessing step). In practice, one achieves this by first computing the mean of each dimension (across the dataset) and subtracts this from each dimension. Next, each dimension is divided by its standard deviation.

Example: When working with audio data, it is common to use MFCCs as the data representation. However, the first component (representing the DC) of the MFCC features often overshadow the other components. Thus, one method to restore balance to the components is to standardize the values in each component independently.

B.1.3 PCA/ZCA Whitening

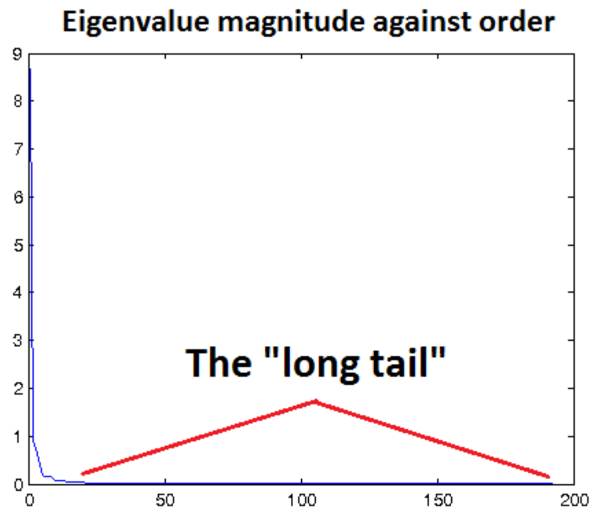
After doing the simple normalizations, whitening is often the next preprocessing step employed that helps make our algorithms work better. In practice, many deep learning algorithms rely on whitening to learn good features.

In performing PCA/ZCA whitening, it is pertinent to first zero-mean the features (across the dataset) to ensure that $\frac{1}{m}\sum_i x^{(i)} = 0$. Specifically, this should be done before computing the covariance matrix. (The only exception is when per-example mean subtraction is performed and the data is stationary across dimensions/pixels.)

Next, one needs to select the value of `epsilon` to use when performing PCA/ZCA whitening(4.2) (recall that this was the regularization term that has an effect of low-pass filtering the data). It turns out that selecting this value can also play an important role for feature learning, we discuss two cases for selecting `epsilon`:

Reconstruction Based Models

In models based on reconstruction (including Autoencoders, Sparse Coding, RBMs, k-Means), it is often preferable to set `epsilon` to a value such that low-pass filtering is achieved. One way to check this is to set a value for `epsilon`, run ZCA whitening, and thereafter visualize the data before and after whitening. If the value of `epsilon` is set too low, the data will look very noisy; conversely, if `epsilon` is set too high, you will see a “blurred" version of the original data. A good way to get a feel for the magnitude of `epsilon` to try is to plot the eigenvalues on a graph. As visible in the example graph below, you may get a “long tail" corresponding to the high frequency noise components. You will want to choose `epsilon` such that most of the “long tail" is filtered out, i.e. choose `epsilon` such that it is greater than most of the small eigenvalues corresponding to the noise.



In reconstruction based models, the loss function includes a term that penalizes reconstructions that are far from the original inputs. Then, if `epsilon` is set too low, the data will contain a lot of noise which the model will need to reconstruct well. As a result, it is very important for reconstruction based models to have data that has been low-pass filtered.

Tip: If your data has been scaled reasonably (e.g., to [0,1]), start with `epsilon = 0.01` or `epsilon = 0.1`.

ICA-based Models (with orthogonalization)

For ICA-based models with orthogonalization, it is very important for the data to be as close to white (identity covariance) as possible. This is a side-effect of using orthogonalization to decorrelate the features learned (more details in ICA 11.1). Hence, in this case, you will want to use an `epsilon` that is as small as possible (e.g., `epsilon = 1e - 6`).

Tip: In PCA whitening, one also has the option of performing dimension reduction while whitening the data. This is usually an excellent idea since it can greatly speed up the algorithms (less computation and less parameters). A simple rule of thumb to choose how many principle components to retain is to keep enough components to have 99% of the variance retained (more details at PCA 4.1.6)

Note: When working in a classification framework, one should compute the PCA/ZCA whitening matrices based only on the training set. The following parameters used be saved for use with the test set: (a) average vector that was used to zero-mean the data, (b) whitening matrices. The test set should undergo the same preprocessing steps using these saved values.

B.1.4 Large Images

For large images, PCA/ZCA based whitening methods are impractical as the covariance matrix is too large. For these cases, we defer to 1/f-whitening methods. (more details to come)

B.1.5 Standard Pipelines

In this section, we describe several “standard pipelines" that have worked well for some datasets:

Natural Grey-scale Images

Since grey-scale images have the stationarity property, we usually first remove the mean-component from each data example separately (remove DC). After this step, PCA/ZCA whitening is often employed with a value of `epsilon` set large enough to low-pass filter the data.

Color Images

For color images, the stationarity property does not hold across color channels. Hence, we usually start by rescaling the data (making sure it is in [0,1]) ad then applying PCA/ZCA with a sufficiently large `epsilon`. Note that it is important to perform feature mean-normalization before computing the PCA transformation.

Audio (MFCC/Spectrograms)

For audio data (MFCC and Spectrograms), each dimension usually have different scales (variances); the first component of MFCCs, for example, is the DC component and usually has a larger magnitude than the other components. This is especially so when one includes the temporal derivatives (a common practice in audio processing). As a result, the preprocessing usually starts with simple data standardization (zero-mean, unit-variance per data dimension), followed by PCA/ZCA whitening (with an appropriate `epsilon`).

MNIST Handwritten Digits

The MNIST dataset has pixel values in the range [0,255]. We thus start with simple rescaling to shift the data into the range [0,1]. In practice, removing the mean-value per example can also help feature learning. Note: While one could also elect to use PCA/ZCA whitening on MNIST if desired, this is not often done in practice.

B.2 Deriving gradients using the backpropagation idea

B.2.1 Introduction

In the section on the backpropagation algorithm (2.2), you were briefly introduced to backpropagation as a means of deriving gradients for learning in the sparse autoencoder. It turns out that together with matrix calculus, this provides a powerful method and intuition for deriving gradients for more complex matrix functions (functions from matrices to the reals, or symbolically, from $\mathbb{R}^{r \times c} \to \mathbb{R}$).

First, recall the backpropagation idea, which we present in a modified form appropriate for our purposes below:

1. For each output unit i in layer n_l (the final layer), set
$$\delta_i^{(n_l)} = \frac{\partial}{\partial z_i^{(n_l)}} \; J(z^{(n_l)})$$
where $J(z)$ is our "objective function" (explained below).

2. For $l = n_l - 1, n_l - 2, n_l - 3, \ldots, 2$
For each node i in layer l, set
$$\delta_i^{(l)} = \left(\sum_{j=1}^{s_{l+1}} W_{ji}^{(l)} \delta_j^{(l+1)} \right) \bullet \frac{\partial}{\partial z_i^{(l)}} f^{(l)}(z_i^{(l)})$$

3. Compute the desired partial derivatives,
$$\nabla_{W^{(l)}} J(W, b; x, y) = \delta^{(l+1)} (a^{(l)})^T, \tag{112}$$
$$\tag{113}$$

Quick notation recap:

- l is the number of layers in the neural network
- n_l is the number of neurons in the lth layer
- $W_{ji}^{(l)}$ is the weight from the ith unit in the lth layer to the jth unit in the $(l+1)$th layer
- $z_i^{(l)}$ is the input to the ith unit in the lth layer
- $a_i^{(l)}$ is the activation of the ith unit in the lth layer
- $A \bullet B$ is the Hadamard or element-wise product, which for $r \times c$ matrices A and B yields the $r \times c$ matrix $C = A \bullet B$ such that $C_{r,c} = A_{r,c} \cdot B_{r,c}$
- $f(l)$ is the activation function for units in the lth layer

Let's say we have a function F that takes a matrix X and yields a real number. We would like to use the backpropagation idea to compute the gradient with respect to X of F, that is $\nabla_X F$. The general idea is to see the function F as a multi-layer neural network, and to derive the gradients using the backpropagation idea.

To do this, we will set our "objective function" to be the function $J(z)$ that when applied to the outputs of the neurons in the last layer yields the value $F(X)$. For the intermediate layers, we will also choose our activation functions $f^{(l)}$ to this end.

Using this method, we can easily compute derivatives with respect to the inputs X, as well as derivatives with respect to any of the weights in the network, as we shall see later.

B.2.2 Examples

To illustrate the use of the backpropagation idea to compute derivatives with respect to the inputs, we will use two functions from the section on sparse coding(10.2), in examples 1 and 2. In example 3, we use a function from independent component analysis(11.1) to illustrate the use of this idea to compute derivates with respect to weights, and in this specific case, what to do in the case of tied or repeated weights.

Example 1: Objective for weight matrix in sparse coding

Recall for sparse coding(10.2), the objective function for the weight matrix A, given the feature matrix s:

$$F(A; s) = \|As - x\|_2^2 + \gamma \|A\|_2^2$$

We would like to find the gradient of F with respect to A, or in symbols, $\nabla_A F(A)$. Since the objective function is a sum of two terms in A, the gradient is the sum of gradients of each of the individual terms. The gradient of the second term is trivial, so we will consider the gradient of the first term instead.

The first term, $\|As - x\|_2^2$, can be seen as an instantiation of neural network taking s as an input, and proceeding in four steps, as described and illustrated in the paragraph and diagram below:

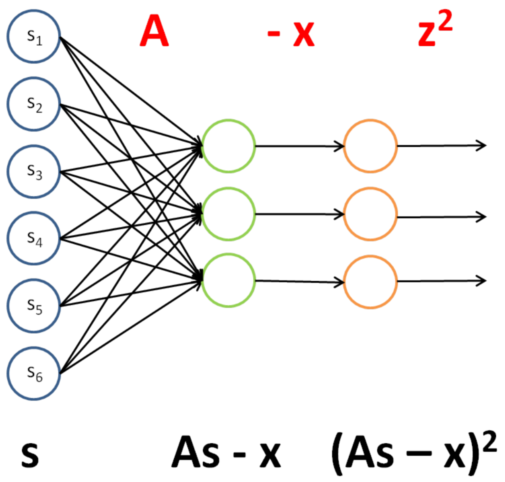
1. Apply A as the weights from the first layer to the second layer.
2. Subtract x from the activation of the second layer, which uses the identity activation function.
3. Pass this unchanged to the third layer, via identity weights. Use the square function as the activation function for the third layer.
4. Sum all the activations of the third layer.

The weights and activation functions of this network are as follows:

Layer	Weight	Activation function f
1	A	$f(z_i) = z_i$ (identity)
2	I (identity)	$f(z_i) = z_i - x_i$
3	N/A	$f(z_i) = z_i^2$

To have $J(z(3)) = F(x)$, we can set $J(z^{(3)}) = \sum_k J(z_k^{(3)})$.

Once we see F as a neural network, the gradient $\nabla_X F$ becomes easy to compute - applying backpropagation yields:



Layer	Derivative of activation function f'	Delta	Input z to this layer
3	$f'(z_i) = 2z_i$	$f'(z_i) = 2z_i$	$As - x$
2	$f'(z_i) = 1$	$\left(I^T\delta^{(3)}\right) \bullet 1$	As
1	$f'(z_i) = 1$	$\left(A^T\delta^{(2)}\right) \bullet 1$	s

Hence,

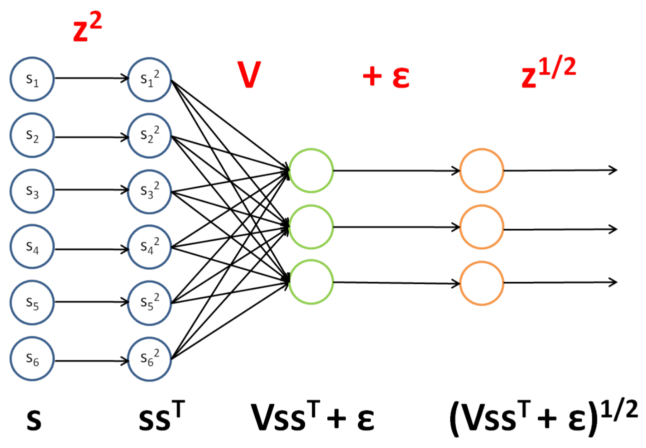
$$\nabla_X F = A^T I^T 2(As - x) \tag{114}$$
$$= A^T 2(As - x) \tag{115}$$

Example 2: Smoothed topographic L1 sparsity penalty in sparse coding
Recall the smoothed topographic L1 sparsity penalty on s in sparse coding(10.2):

$$\sum \sqrt{Vss^T + \epsilon}$$

where V is the grouping matrix, s is the feature matrix and ϵ is a constant.

We would like to find $\nabla_s \sum \sqrt{Vss^T + \epsilon}$. As above, let's see this term as an instantiation of a neural network:



Layer	Weight	Activation function f
1	I	$f(z_i) = z_i^2$
2	V	$f(z_i) = z_i$
3	I	$f(z_i) = z_i + \epsilon$
4	N/A	$f(z_i) = z_i^{\frac{1}{2}}$

The weights and activation functions of this network are as follows:

To have $J(z(4)) = F(x)$, we can set $J(z^{(4)}) = \sum_k J(z_k^{(4)})$.

Once we see F as a neural network, the gradient $\nabla_X F$ becomes easy to compute - applying backpropagation yields:

Layer	Derivative of activation function f'	Delta	Input z to this layer
4	$f'(z_i) = \frac{1}{2} z_i^{-\frac{1}{2}}$	$f'(z_i) = \frac{1}{2} z_i^{-\frac{1}{2}}$	$(Vss^T + \epsilon)$
3	$f'(z_i) = 1$	$\left(I^T \delta^{(4)}\right) \bullet 1$	Vss^T
2	$f'(z_i) = 1$	$\left(V^T \delta^{(3)}\right) \bullet 1$	ss^T
1	$f'(z_i) = 2z_i$	$\left(I^T \delta^{(2)}\right) \bullet 2s$	s

Hence,

$$\nabla_X F = I^T V^T I^T \frac{1}{2} (Vss^T + \epsilon)^{-\frac{1}{2}} \bullet 2s \tag{116}$$

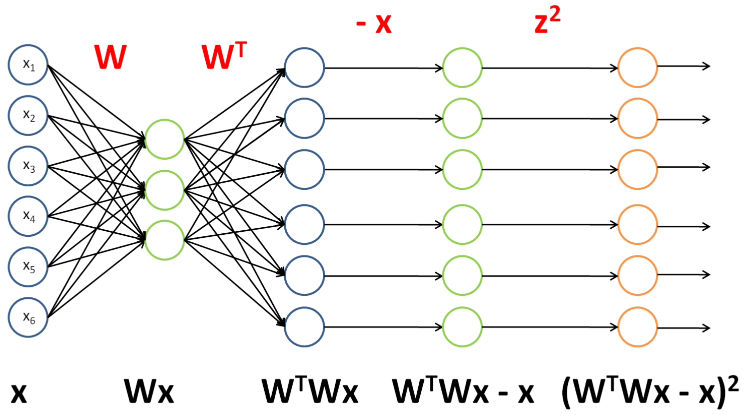
$$= V^T \frac{1}{2} (Vss^T + \epsilon)^{-\frac{1}{2}} \bullet 2s \tag{117}$$

$$= V^T (Vss^T + \epsilon)^{-\frac{1}{2}} \bullet s \tag{118}$$

Example 3: ICA reconstruction cost

Recall the independent component analysis (ICA 11.1) reconstruction cost term: $\|W^T Wx - x\|_2^2$ where W is the weight matrix and x is the input.

We would like to find $\nabla_W \|W^T Wx - x\|_2^2$ – the derivative of the term with respect to the weight matrix, rather than the input as in the earlier two examples. We will still proceed similarly though, seeing this term as an instantiation of a neural network:



Layer	Weight	Activation function f
1	W	$f(z_i) = z_i$
2	W^T	$f(z_i) = z_i$
3	I	$f(z_i) = z_i - x_i$
4	N/A	$f(z_i) = z_i^2$

The weights and activation functions of this network are as follows:
To have $J(z(4)) = F(x)$, we can set $J(z^{(4)}) = \sum_k J(z_k^{(4)})$.

Now that we can see F as a neural network, we can try to compute the gradient $\nabla_W F$. However, we now face the difficulty that W appears twice in the network. Fortunately, it turns out that if W appears multiple times in the network, the gradient with respect to W is simply the sum of gradients for each instance of W in the network (you may wish to work out a formal proof of this fact to convince yourself). With this in mind, we will proceed to work out the deltas first:

Layer	Derivative of activation function f'	Delta	Input z to this layer
4	$f'(z_i) = 2z_i$	$f'(z_i) = 2z_i$	$(W^TWx - x)$
3	$f'(z_i) = 1$	$\left(I^T\delta^{(4)}\right) \bullet 1$	W^TWx
2	$f'(z_i) = 1$	$\left((W^T)^T\delta^{(3)}\right) \bullet 1$	Wx
1	$f'(z_i) = 1$	$\left(W^T\delta^{(2)}\right) \bullet 1$	x

To find the gradients with respect to W, first we find the gradients with respect to each instance of W in the network.

With respect to W^T:

$$\nabla_{W^T} F = \delta^{(3)} a^{(2)T} \tag{119}$$
$$= 2(W^TWx - x)(Wx)^T \tag{120}$$

With respect to W:

$$\nabla_W F = \delta^{(2)} a^{(1)T} \tag{121}$$
$$= (W^T)(2(W^TWx - x))x^T \tag{122}$$

Taking sums, noting that we need to transpose the gradient with respect to W^T to get the gradient with respect to W, yields the final gradient with respect to W (pardon the slight abuse of notation here):

$$\nabla_W F = \nabla_W F + (\nabla_{W^T} F)^T \tag{123}$$
$$= (W^T)(2(W^TWx - x))x^T + 2(Wx)(W^TWx - x)^T \tag{124}$$

C Miscellaneous

C.1 MATLAB Modules

Sparse autoencoder | sparseae_exercise.zip

- checkNumericalGradient.m - Makes sure that computeNumericalGradient is implmented correctly
- computeNumericalGradient.m - Computes numerical gradient of a function (to be filled in)
- display_network.m - Visualizes images or filters for autoencoders as a grid
- initializeParameters.m - Initializes parameters for sparse autoencoder randomly
- sampleIMAGES.m - Samples 8×8 patches from an image matrix (to be filled in)
- sparseAutoencoderCost.m - Calculates cost and gradient of cost function of sparse autoencoder
- train.m - Framework for training and testing sparse autoencoder

Using the MNIST Dataset | mnistHelper.zip

- loadMNISTImages.m - Returns a matrix containing raw MNIST images
- loadMNISTLabels.m - Returns a matrix containing MNIST labels

PCA and Whitening | pca_exercise.zip

- display_network.m - Visualizes images or filters for autoencoders as a grid
- pca_gen.m - Framework for whitening exercise
- sampleIMAGESRAW.m - Returns 8×8 raw unwhitened patches

Softmax Regression | softmax_exercise.zip

- checkNumericalGradient.m - Makes sure that computeNumericalGradient is implmented correctly
- display_network.m - Visualizes images or filters for autoencoders as a grid
- loadMNISTImages.m - Returns a matrix containing raw MNIST images
- loadMNISTLabels.m - Returns a matrix containing MNIST labels
- softmaxCost.m - Computes cost and gradient of cost function of softmax
- softmaxTrain.m - Trains a softmax model with the given parameters
- train.m - Framework for this exercise

C.2 Style Guide

File / Function Names

Functions and file names should be alphanumeric, with the first letter of the first word in lowercase, and the first letter in the remaining words in uppercase. E.g.

Variable Names

Variable names should follow the same convention as the style guide.

C.3 Useful Links

Matlab Guide

- Writing Fast MATLAB Code (by Pascal Getreuer)
- Matrix Calculus Reference
- The Matrix Cookbook
- Notes on Convolutional Neural Networks

Index