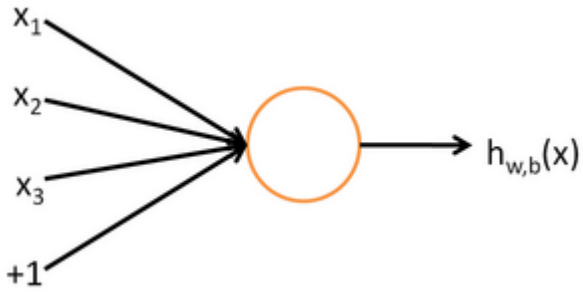
Neural Networks

From Ufldl

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 sigmoid function:

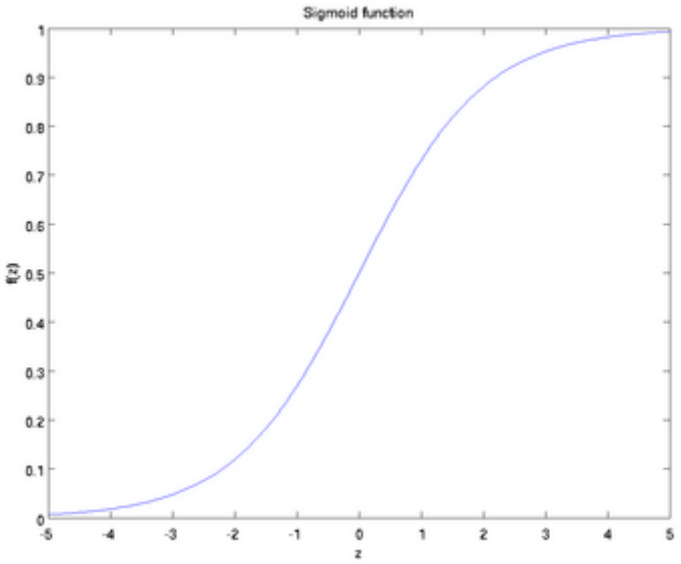
$$f(z) = \frac{1}{1+\exp(-z)}.$$

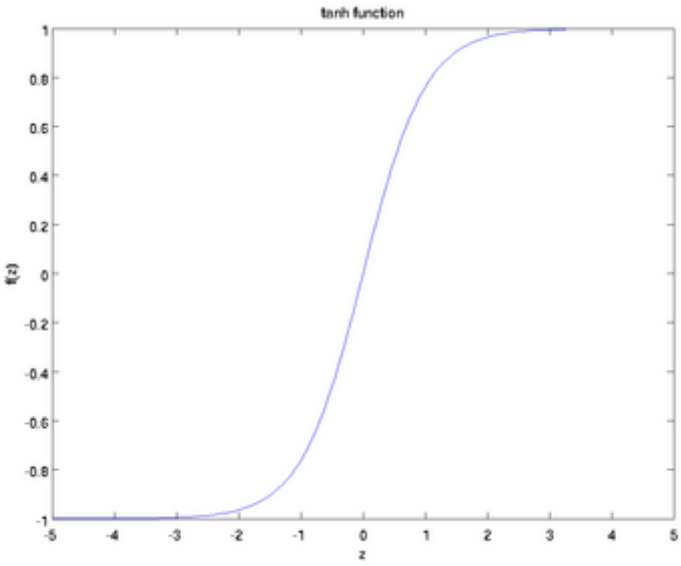
Thus, our single neuron corresponds exactly to the input-output mapping defined by logistic regression.

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:





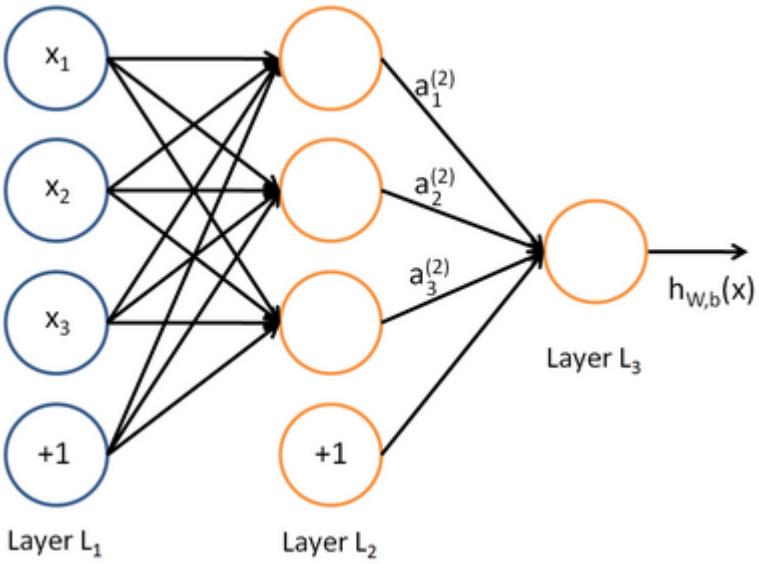
The tanh(z) function is a rescaled version of the sigmoid, and its output range is $[-1,1]$ instead of $[0,1]$.

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.

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:



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:

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

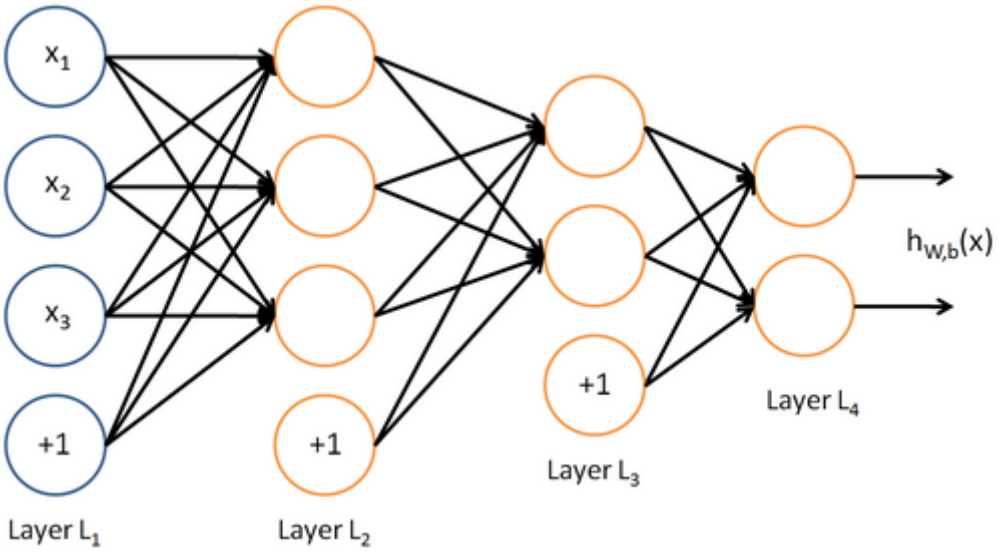
We call this step **forward propagation.** More generally, recalling that we also use $a^{(1)} = x$ to also denote the values from the input layer, then given layer l's activations $a^{(l)}$, we can compute layer $l + 1$'s activations $a^{(l + 1)}$ as:

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

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.)

Language : 中文

Backpropagation Algorithm

From Ufldl

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:

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

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:

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

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.

[Note: 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,ε^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:

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

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

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:

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

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

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, 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

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

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

   For each node *i* in layer *l*, set

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

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

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

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 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
$$\delta^{(n_l)} = -(y - a^{(n_l)}) \bullet f'(z^{(n_l)})$$
3. For $l = n_l - 1, n_l - 2, n_l - 3, \ldots, 2$

   Set
$$\delta^{(l)} = \left((W^{(l)})^T \delta^{(l+1)}\right) \bullet f'(z^{(l)})$$
4. Compute the desired partial derivatives:
$$\nabla_{W^{(l)}} J(W,b;x,y) = \delta^{(l+1)} (a^{(l)})^T,$$
$$\nabla_{b^{(l)}} J(W,b;x,y) = \delta^{(l+1)}.$$

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 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. For $i = 1$ 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:

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

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

Neural Networks | **Backpropagation Algorithm** | Gradient checking and advanced optimization | Autoencoders and Sparsity | Visualizing a Trained Autoencoder | Sparse Autoencoder Notation Summary | Exercise:Sparse Autoencoder

Language : 中文

Gradient checking and advanced optimization

From Ufldl

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

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

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

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

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

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

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

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

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

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

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:

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

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

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

This result shows that the final block of psuedo-code in Backpropagation Algorithm 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.

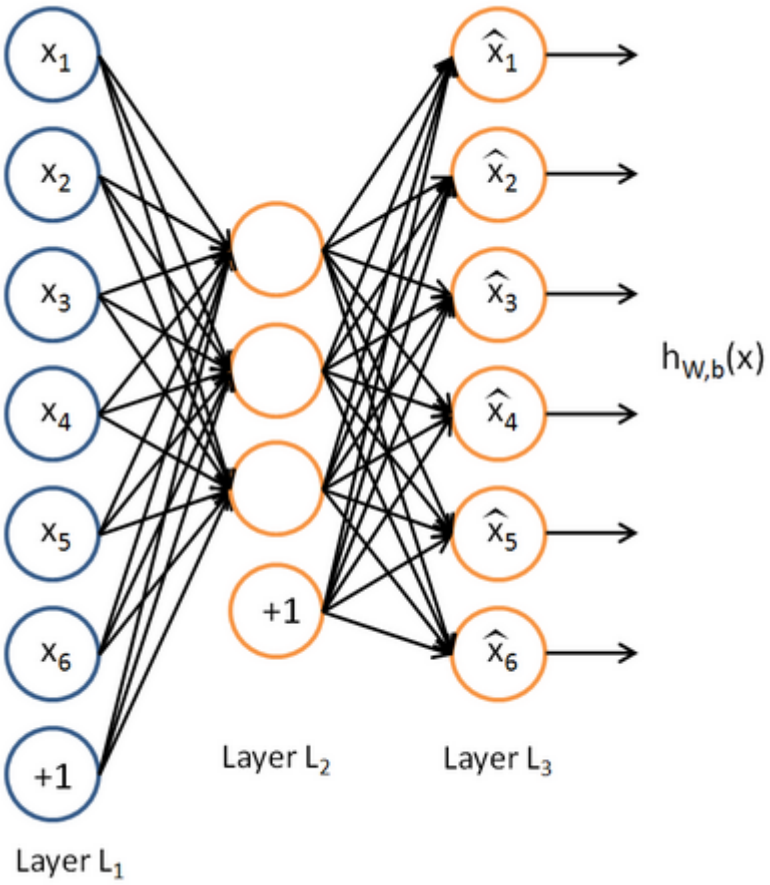
Language : 中文

Autoencoders and Sparsity

From Ufldl

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

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

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

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

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:

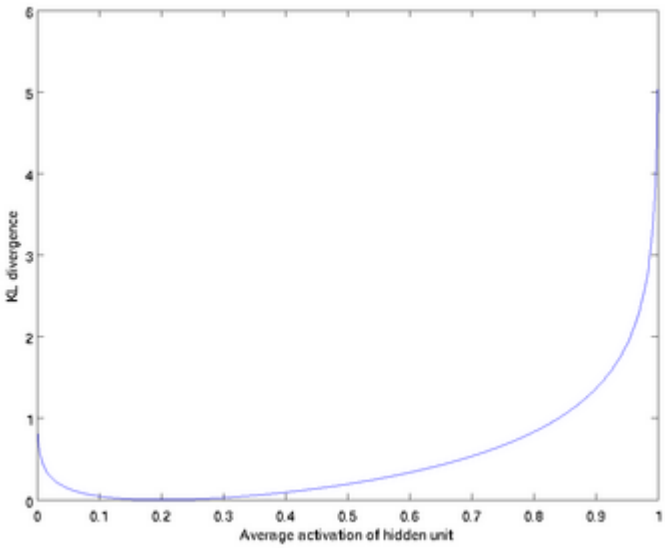
$$\sum_{j=1}^{s_2} \rho \log \frac{\rho}{\hat{\rho}_j} + (1-\rho) \log \frac{1-\rho}{1-\hat{\rho}_j}.$$

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

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

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

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

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

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

now instead compute

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

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.

Neural Networks | Backpropagation Algorithm | Gradient checking and advanced optimization | **Autoencoders and Sparsity** | Visualizing a Trained Autoencoder | Sparse Autoencoder Notation Summary | Exercise:Sparse Autoencoder

Language : 中文

Visualizing a Trained Autoencoder

From Ufldl

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:

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

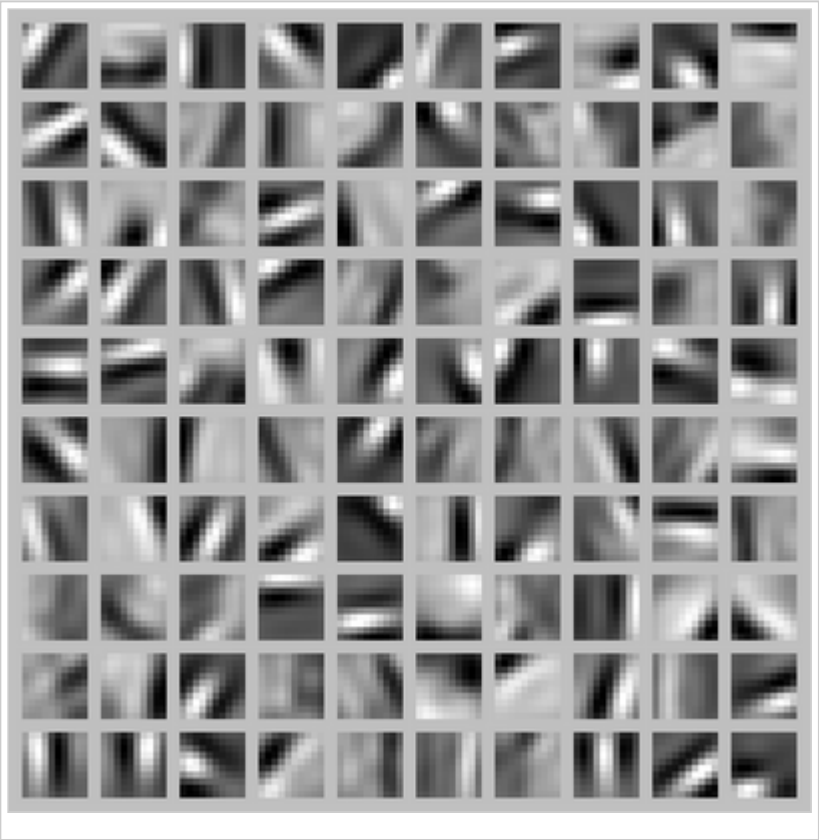
We will visualize the function computed by hidden unit i---which depends on the parameters $W_{ij}^{(1)}$ (ignoring the bias term for now)---using a 2D image. In particular, we think of $a_i^{(2)}$ as some non-linear feature of the input x. We ask: What input image x would cause $a_i^{(2)}$ to be maximally 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

$$x_j = \frac{W_{ij}^{(1)}}{\sqrt{\sum_{j=1}^{100} (W_{ij}^{(1)})^2}}.$$

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[1] 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.

---

[1] *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.*

Neural Networks | Backpropagation Algorithm | Gradient checking and advanced optimization | Autoencoders and Sparsity | **Visualizing a Trained Autoencoder** | Sparse Autoencoder Notation Summary | Exercise:Sparse Autoencoder

Language : 中文

Sparse Autoencoder Notation Summary

From Ufldl

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

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 $f(z) = \tanh(z)$.
$z_i^{(l)}$	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).

Neural Networks | Backpropagation Algorithm | Gradient checking and advanced optimization | Autoencoders and Sparsity | Visualizing a Trained Autoencoder | **Sparse Autoencoder Notation Summary** | Exercise:Sparse Autoencoder

Language : 中文

Exercise:Sparse Autoencoder

From Ufldl

Contents

Download Related Reading

- sparseae_reading.pdf (http://nlp.stanford.edu/~socherr/sparseAutoencoder_2011new.pdf)
- sparseae_exercise.pdf (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 sparseae_exercise.zip (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 ~3x slower for this exercise (and thus is less recommended). You can read more about this in the Fminlbfgs_Details page.

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 8x8 image patch.

Step 2: Sparse autoencoder objective

Implement code to compute the sparse autoencoder cost function $J_{\text{sparse}}(W,b)$ (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 $J_{\text{sparse}}(W,b)$ 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.

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.

Step 4: Train the sparse autoencoder

Now that you have code that computes J_{sparse} and its derivatives, we're ready to minimize J_{sparse} 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.

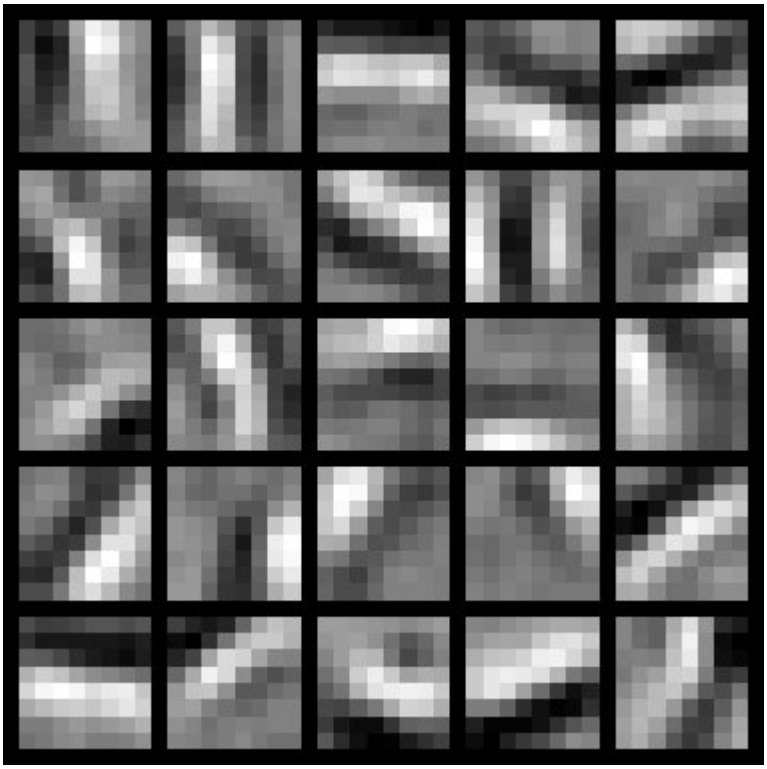
Implementational tip: Once you have your backpropagation implementation correctly computing the derivatives (as verified using gradient checking in Step 3), when you are now using it with L-BFGS to optimize $J_{\text{sparse}}(W,b)$, make sure you're not doing gradient-checking on every step. Backpropagation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

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).

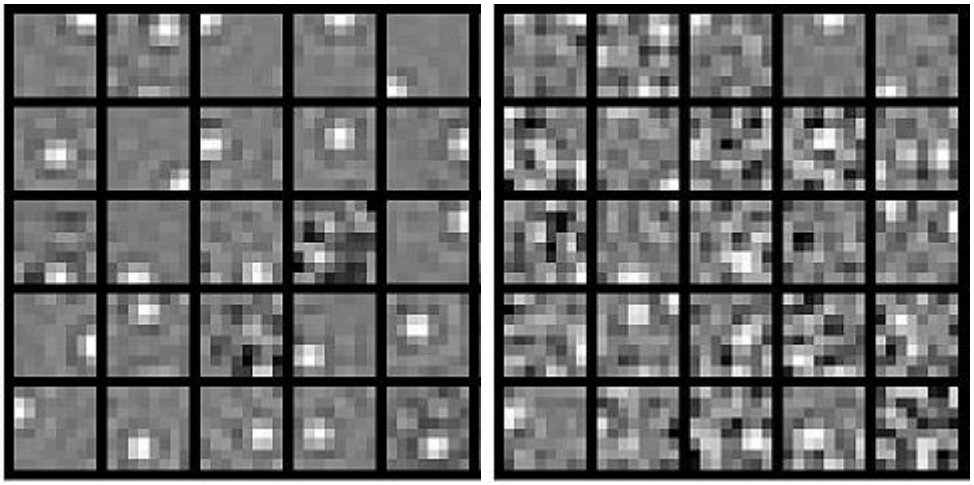
Results

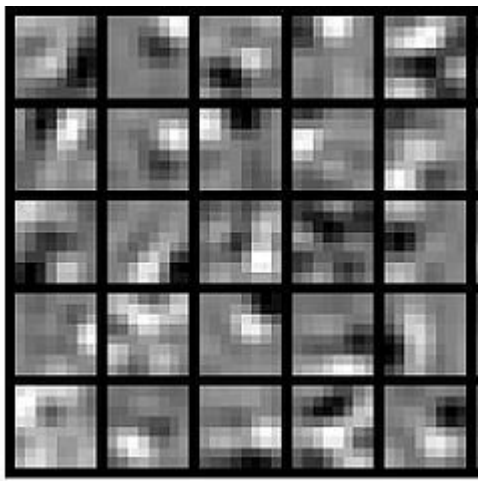
To successfully complete this assignment, you should demonstrate your sparse autoencoder algorithm learning a set of edge detectors. For example, this was the visualization we obtained:

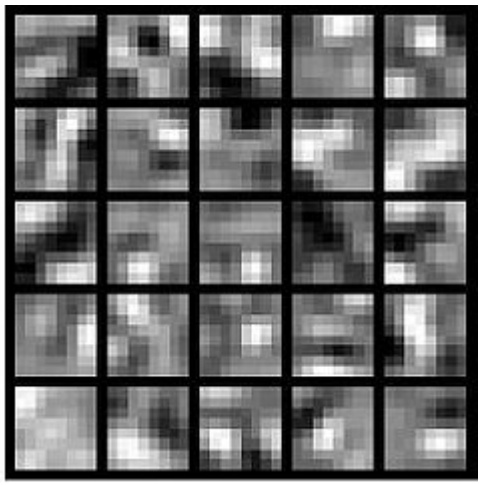


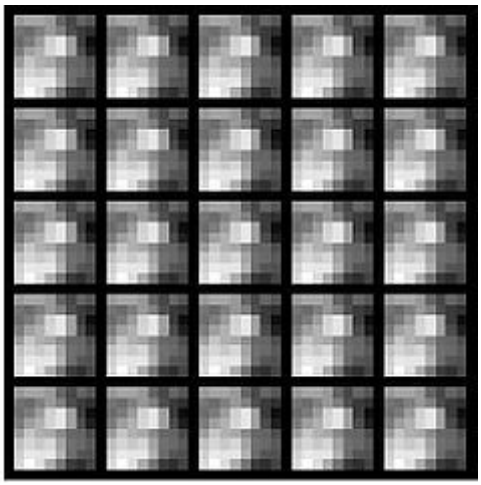
Our implementation took around 5 minutes to run on a fast computer. In case you end up needing to try out multiple implementations or different parameter values, be sure to budget enough time for debugging and to run the experiments you'll need.

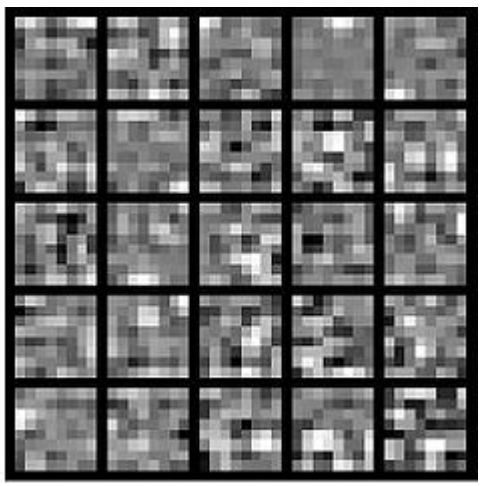
Also, by way of comparison, here are some visualizations from implementations that we do not consider successful (either a buggy implementation, or where the parameters were poorly tuned):











Neural Networks | Backpropagation Algorithm | Gradient checking and advanced optimization | Autoencoders and Sparsity | Visualizing a Trained Autoencoder | Sparse Autoencoder Notation Summary | **Exercise:Sparse Autoencoder**

Vectorization

From Ufldl

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.

Language : 中文

Logistic Regression Vectorization Example

From Ufldl

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

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

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 OpenClassroom videos (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:

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

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.

Vectorization | **Logistic Regression Vectorization Example** | Neural Network Vectorization | Exercise:Vectorization

Neural Network Vectorization

From Ufldl

In this section, we derive a vectorized version of our neural network. In our earlier description of Neural Networks, 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.

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:

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

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, 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

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

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.

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.

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 $y^{(i)} \in \Re^{s3}$. 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, we had that for a single training example (*x*,*y*), we can compute the derivatives as

$$\begin{aligned}
\delta^{(3)} &= -(y - a^{(3)}) \bullet f'(z^{(3)}), \\
\delta^{(2)} &= ((W^{(2)})^T \delta^{(3)}) \bullet f'(z^{(2)}), \\
\nabla_{W^{(2)}} J(W,b;x,y) &= \delta^{(3)} (a^{(2)})^T, \\
\nabla_{W^{(1)}} J(W,b;x,y) &= \delta^{(2)} (a^{(1)})^T.
\end{aligned}$$

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, 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; however, we encourage you to try to carry out the vectorization yourself without looking at the hints.

Sparse autoencoder

The sparse autoencoder 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:

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

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`.

Vectorization | Logistic Regression Vectorization Example | **Neural Network Vectorization** | Exercise:Vectorization

Language : 中文

Exercise:Vectorization

From Ufldl

Contents

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.

Support Code/Data

The following additional files are required for this exercise:

- MNIST Dataset (Training Images) (http://yann.lecun.com/exdb/mnist/train-images-idx3-ubyte.gz)
- MNIST Dataset (Training Labels) (http://yann.lecun.com/exdb/mnist/train-labels-idx1-ubyte.gz)
- Support functions for loading MNIST in Matlab

Step 1: Vectorize your Sparse Autoencoder Implementation

Using the ideas from Vectorization and Neural Network Vectorization, 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 8x8 image patches).

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

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.

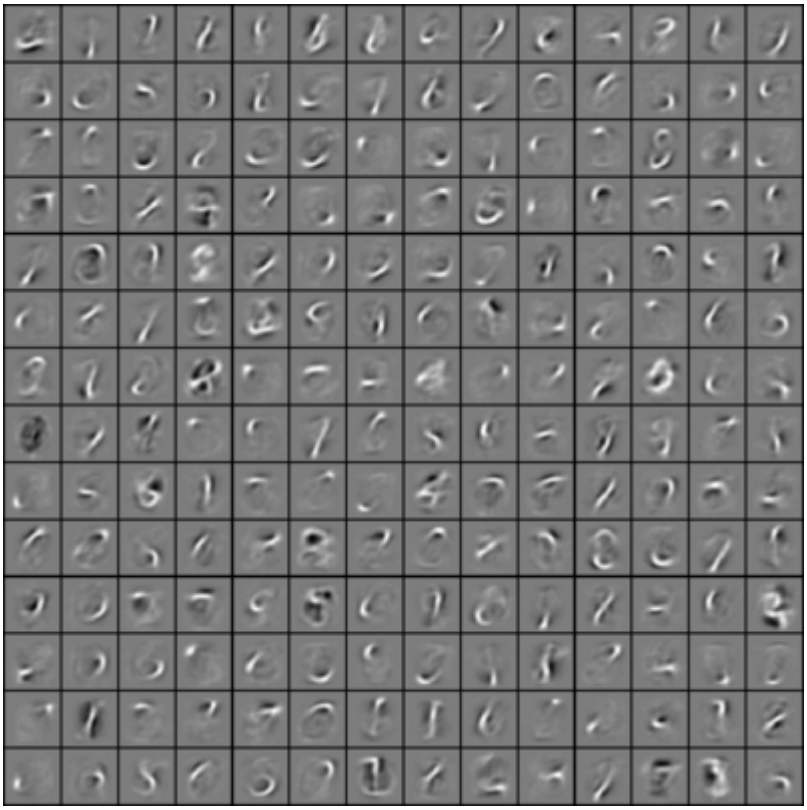
The MNIST data is available at [1] (http://yann.lecun.com/exdb/mnist/) . Download the file `train-images-idx3-ubyte.gz` and decompress it. After obtaining the source images, you should use helper functions that we provide to load the data into Matlab as matrices. While the helper functions that we provide 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.)

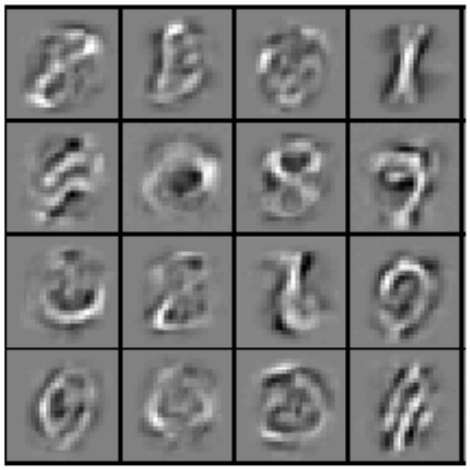
The following set of parameters worked well for us to learn good features on the MNIST dataset:

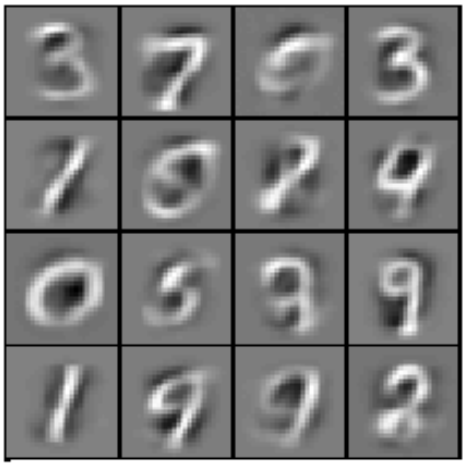
```
visibleSize = 28*28
hiddenSize = 196
sparsityParam = 0.1
lambda = 3e-3
beta = 3
patches = first 10000 images from the MNIST dataset
```

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:





If your image looks like one of the above images, check your code and parameters again. Learning these features are a prelude to the later exercises, where we shall see how they will be useful for classification.