CS294A Lecture notes

Andrew Ng

Sparse autoencoder

1 Introduction

Supervised learning is one of the most powerful tools of AI, and has led to automatic zip code recognition, speech recognition, self-driving cars, and a continually improving understanding of the human genome. Despite its significant successes, supervised learning today is still severely limited. Specifically, most applications of it still require that we manually specify the input features x given to the algorithm. Once a good feature representation is given, a supervised learning algorithm can do well. But in such domains as computer vision, audio processing, and natural language processing, there're now hundreds or perhaps thousands of researchers who've spent years of their lives slowly and laboriously hand-engineering vision, audio or text features. While much of this feature-engineering work is extremely clever, one has to wonder if we can do better. Certainly this labor-intensive hand-engineering approach does not scale well to new problems; further, ideally we'd like to have algorithms that can automatically learn even better feature representations than the hand-engineered ones.

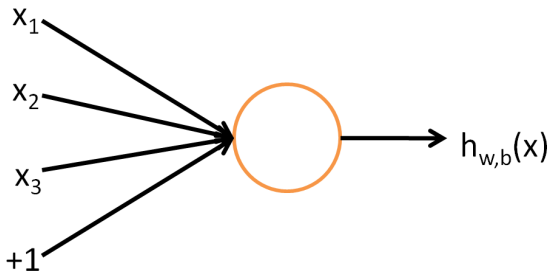
These notes describe the **sparse autoencoder** learning algorithm, which is one approach to automatically learn features from unlabeled data. In some domains, such as computer vision, this approach is not by itself competitive with the best hand-engineered features, but the features it can learn do turn out to be useful for a range of problems (including ones in audio, text, etc). Further, there're more sophisticated versions of the sparse autoencoder (not described in these notes, but that you'll hear more about later in the class) that do surprisingly well, and in many cases are competitive with or superior to even the best hand-engineered representations.

These notes are organized as follows. We will first describe feedforward neural networks and the backpropagation algorithm for supervised learning. Then, we show how this is used to construct an autoencoder, which is an unsupervised learning algorithm. Finally, we build on this to derive a sparse autoencoder. Because these notes are fairly notation-heavy, the last page also contains a summary of the symbols used.

2 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 $f : \mathbb{R} \mapsto \mathbb{R}$ is called the **activation function**. In these notes, we will choose $f(\cdot)$ to be the sigmoid function:

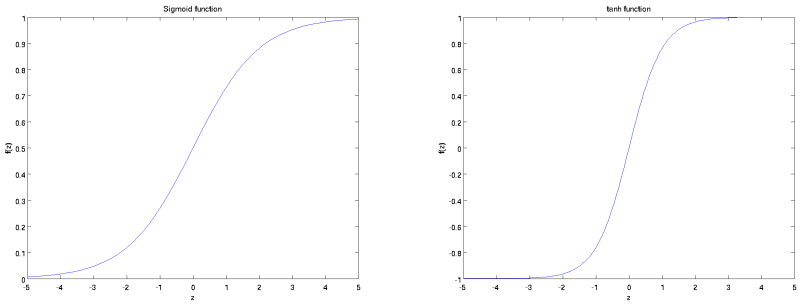
$$f(z) = \frac{1}{1 + \exp(-z)}.$$

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:

$$f(z) = \tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}, \tag{1}$$

Here are plots of the sigmoid and tanh functions:



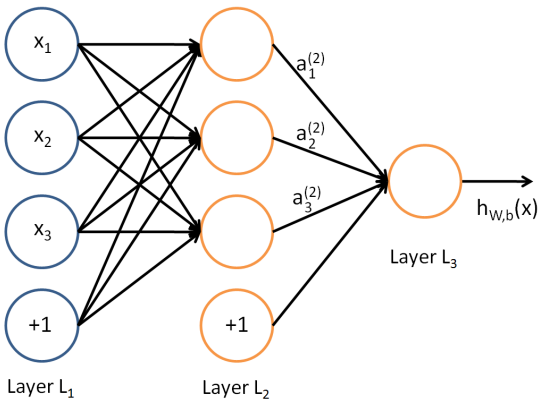
The $\tanh(z)$ function is a rescaled version of the sigmoid, and its output range is $[-1, 1]$ instead of $[0, 1]$.

Note that unlike CS221 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 Neural network formulation

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 $W^{(1)} \in \mathbb{R}^{3\times 3}$, and $W^{(2)} \in \mathbb{R}^{1\times 3}$. 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:

$$
\begin{aligned}
a_1^{(2)} &= f(W_{11}^{(1)}x_1 + W_{12}^{(1)}x_2 + W_{13}^{(1)}x_3 + b_1^{(1)}) && (2)\\
a_2^{(2)} &= f(W_{21}^{(1)}x_1 + W_{22}^{(1)}x_2 + W_{23}^{(1)}x_3 + b_2^{(1)}) && (3)\\
a_3^{(2)} &= f(W_{31}^{(1)}x_1 + W_{32}^{(1)}x_2 + W_{33}^{(1)}x_3 + b_3^{(1)}) && (4)\\
h_{W,b}(x) &= a_1^{(3)} = f(W_{11}^{(2)}a_1^{(2)} + W_{12}^{(2)}a_2^{(2)} + W_{13}^{(2)}a_3^{(2)} + b_1^{(2)}) && (5)
\end{aligned}
$$

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

Equations (2-5) more compactly as:

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

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:

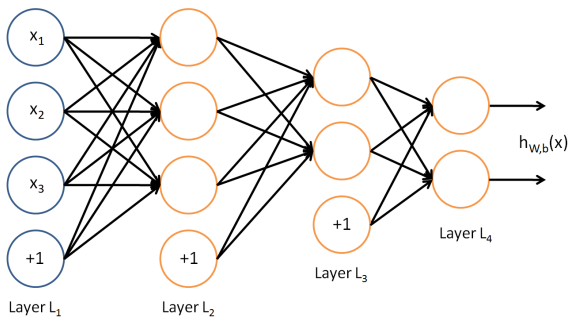
$$z^{(l+1)} = W^{(l)}a^{(l)} + b^{(l)} \tag{6}$$

$$a^{(l+1)} = f(z^{(l+1)}) \tag{7}$$

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 Equations (6-7). 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 $y^{(i)} \in \mathbb{R}^2$. 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

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

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

$$\begin{aligned} J(W, b) &= \left[\frac{1}{m} \sum_{i=1}^{m} J(W, b; x^{(i)}, y^{(i)}) \right] + \frac{\lambda}{2} \sum_{l=1}^{n_l - 1} \sum_{i=1}^{s_l} \sum_{j=1}^{s_{l+1}} \left(W_{ji}^{(l)} \right)^2 \qquad (8) \\ &= \left[\frac{1}{m} \sum_{i=1}^{m} \left(\frac{1}{2} \left\| h_{W,b}(x^{(i)}) - y^{(i)} \right\|^2 \right) \right] + \frac{\lambda}{2} \sum_{l=1}^{n_l - 1} \sum_{i=1}^{s_l} \sum_{j=1}^{s_{l+1}} \left(W_{ji}^{(l)} \right)^2 \end{aligned}$$

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.[1] 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

[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 different to the final network, however. If you took CS229, you may also recognize weight decay this 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.

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 $\mathcal{N}(0, \epsilon^2)$ 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:

$$\begin{aligned} W_{ij}^{(l)} &:= W_{ij}^{(l)} - \alpha \frac{\partial}{\partial W_{ij}^{(l)}} J(W, b) \\ b_i^{(l)} &:= b_i^{(l)} - \alpha \frac{\partial}{\partial b_i^{(l)}} J(W, b) \end{aligned}$$

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, then by referring to Equation (8), we see that the derivative of the overall cost function $J(W, b)$ can be computed as

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

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:

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

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 $f'([z_1, z_2, z_3]) = [\frac{\partial}{\partial z_1} f(z_1), \frac{\partial}{\partial z_2} f(z_2), \frac{\partial}{\partial z_3} f(z_3)]$). 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 Equations (6-7).

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

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

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:

$$\begin{aligned} \nabla_{W^{(l)}} J(W, b; x, y) &= \delta^{(l+1)} (a^{(l)})^T, \\ \nabla_{b^{(l)}} J(W, b; x, y) &= \delta^{(l+1)}. \end{aligned}$$

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,

   2a. Use backpropagation to compute $\nabla_{W^{(l)}} J(W, b; x, y)$ and $\nabla_{b^{(l)}} J(W, b; x, y)$.

   2b. Set $\Delta W^{(l)} := \Delta W^{(l)} + \nabla_{W^{(l)}} J(W, b; x, y)$.

   2c. Set $\Delta b^{(l)} := \Delta b^{(l)} + \nabla_{b^{(l)}} J(W, b; x, y)$.

3. Update the parameters:

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

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 $J : \mathbb{R} \mapsto \mathbb{R}$, so that $\theta \in \mathbb{R}$. 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:

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

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

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

The degree to which these two values should approximate each other will depend on the details of J. But assuming $\texttt{EPSILON} = 10^{-4}$, 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 $\theta \in \mathbb{R}^n$ is a vector rather than a single real number (so that we have n parameters that we want to learn), and $J : \mathbb{R}^n \mapsto \mathbb{R}$. 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 $\theta^{(i+)} = \theta + \texttt{EPSILON} \times \vec{e}_i$, 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 $\theta^{(i-)} = \theta - \texttt{EPSILON} \times \vec{e}_i$ 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:

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

When implementing backpropagation to train a neural network, in a cor-

rect implementation we will have that

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

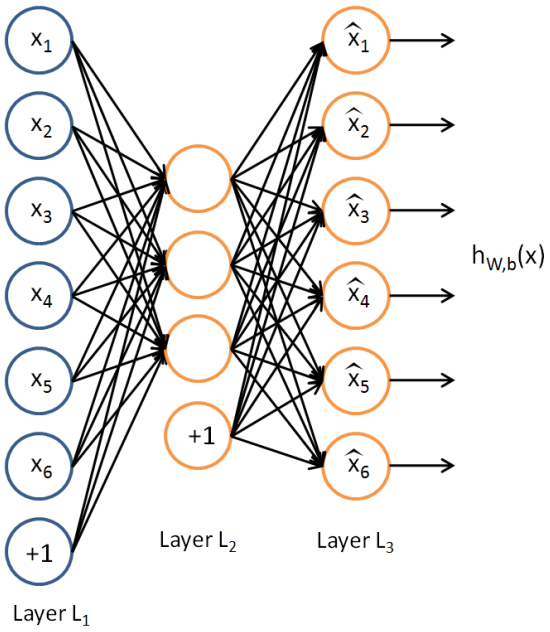
This result shows that the final block of psuedo-code in Section 2.2 is indeed implementing gradient descent. To make sure your implementation of gradient descent is correct, it is usually very helpful to use the method described above to numerically compute the derivatives of $J(W,b)$, and thereby verify 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 **conjugate gradient**.) 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.

3 Autoencoders and sparsity

So far, we have described the application of neural networks to supervised learning, in which we are have labeled training examples. Now suppose we have only unlabeled training examples set $\{x^{(1)}, x^{(2)}, x^{(3)}, \ldots\}$, where $x^{(i)} \in \mathbb{R}^n$. 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 $y \in \mathbb{R}^{100}$. 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 $a^{(2)} \in \mathbb{R}^{50}$, 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.[2]

[2]In fact, this simple autoencoder often ends up learning a low-dimensional representation very similar to PCA's.

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.[3]

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

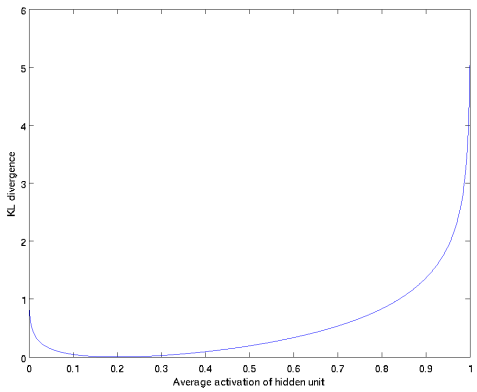
[3]This discussion assumes a sigmoid activation function. If you are using a tanh activation function, then we think of a neuron as being inactive when it outputs values close to -1.

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

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

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.

4 Visualization

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 $a_i^{(1)}$ as some non-linear feature of the input x. We ask: What input image x would cause $a_i^{(1)}$ to be maximally activated? 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

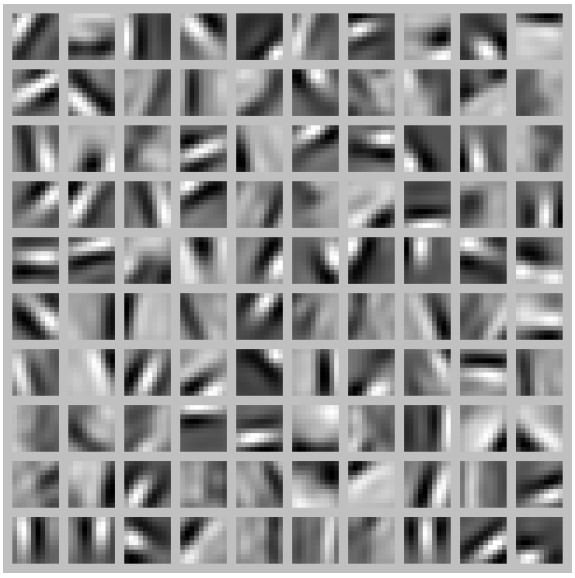
$$x_j = \frac{W_{ij}^{(1)}}{\sqrt{\sum_{j=1}^{100} (W_{ij}^{(1)})^2}}.$$

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[4]) we get the following result:

[4]The results below 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.



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.

5 Summary of notation

x	Input features for a training example, $x \in \mathbb{R}^n$.
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).

§1.1 神经网络

§1.1.1 概述

以监督学习为例, 假设我们有训练样本集 $(\mathbf{x}^{(i)}, \mathbf{y}^{(i)})$, 那么神经网络算法能够提供一种复杂且**非线性**的假设模型 $h_{W,\mathrm{b}}(\mathbf{x})$, 它具有参数 W, b, 可以以此参数来拟合我们的数据.

为了描述神经网络, 我们先从最简单的神经网络讲起, 这个神经网络仅由一个“**神经元**”构成, 图 1.1 给出了这个“神经元”的图示:

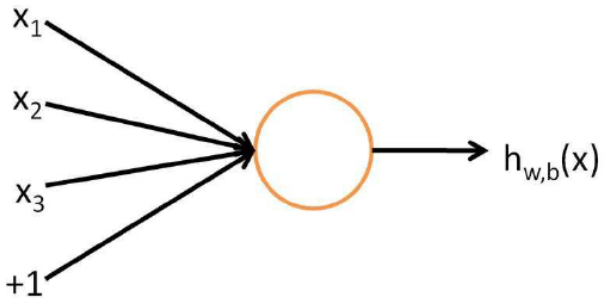


图 1.1 单个神经元结构

这个“神经元”是一个以 x_1, x_2, x_3 及**截距** $+1$ 为输入值的运算单元, 其输出为 $h_{W,b}(\mathbf{x}) = f(W^T\mathbf{x}) = f(\sum_{i=1}^{3} W_i x_i + b)$ (注意对于图 1.1 对应的单个神经元结构, b 为标量), 其中函数 $f : R \mapsto R$ 被称为“**激活函数**”. 在本教程中, 我们选用 **sigmoid 函数**作为激活函数

$$f(z) = sigmoid(z) = \frac{1}{1+e^{-z}}. \tag{1.1.1}$$

可以看出, 这个单一“神经元”的输入 – 输出映射关系其实就是一个**逻辑回归**(logistic regression).

注 1.1.1 *与其它地方 (包括 OpenClassroom 公开课以及斯坦福大学 CS229 课程) 不同的是, 这里我们不再令 $x_0 = 1$. 取而代之, 我们用单独的参数 b 来表示截距.*

注 1.1.2 *如果不引入偏置向量 $\mathbf{b}$, 则可将其包含在 W 中. 即令 $W_0 = b$, $x_0 = 1$, 此时 $h_{W,b}(\mathbf{x}) = f(W^T\mathbf{x}) = f(\sum_{i=0}^{3} W_i x_i)$.*

虽然本教程采用 sigmoid 函数, 但你也可以选择**双曲正切函数** (tanh):

$$f(z) = tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}, \tag{1.1.2}$$

双曲正切函数是 sigmoid 函数的一种变体. 容易验证它们满足:

$$sigmoid(z) = \frac{1}{2}\left[1 + tanh(\frac{z}{2})\right]. \tag{1.1.3}$$

其中 sigmoid 函数的取值范围为 $[0, 1]$, 双曲正切函数的取值范围为 $[-1, 1]$. 图 1.2 和图 1.3 分别给出了 sigmoid 和 tanh 的函数图像.

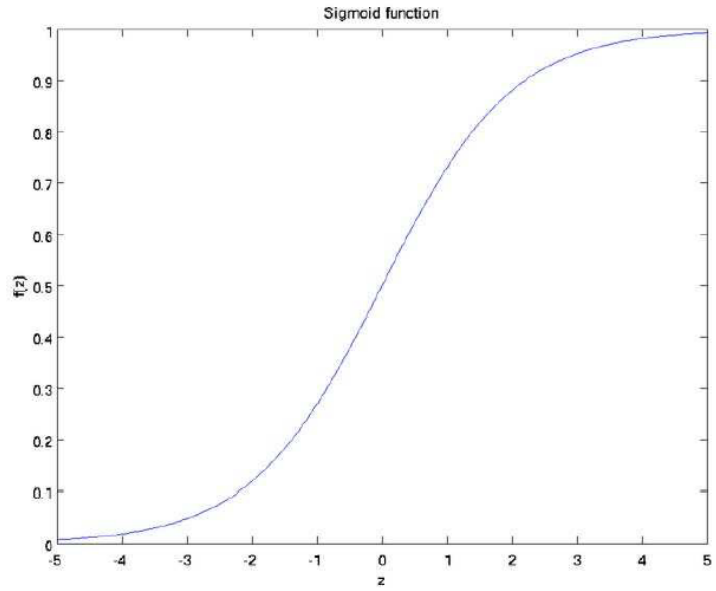


图 1.2 Sigmoid 函数

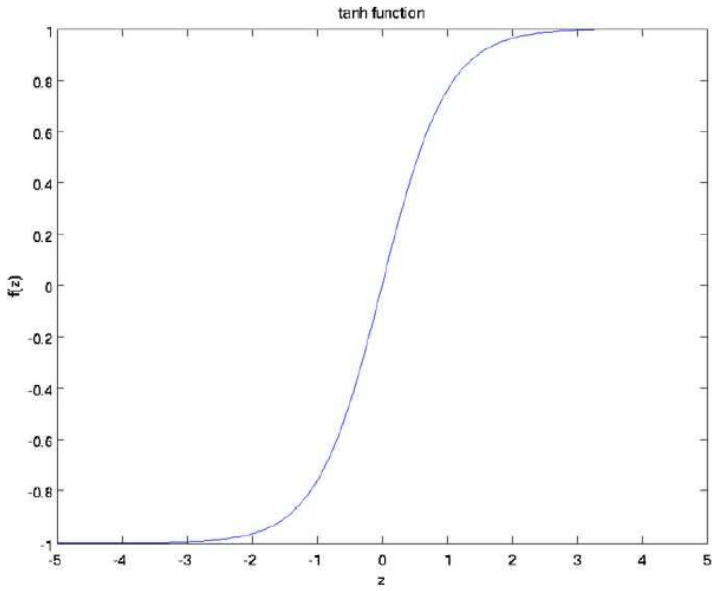


图 1.3 双曲正切函数 tanh

在算法推导中, 激活函数的**导数**会经常用到, 其计算公式分别为

- Sigmoid 函数

$$f'(z) = f(z)(1 - f(z)), \tag{1.1.4}$$

- 双曲正切函数

$$f'(z) = 1 - (f(z))^2. \tag{1.1.5}$$

§1.1.2 神经网络模型

所谓神经网络就是将许多个单一“神经元”联结在一起, 这样, 一个“神经元”的输出就可以是另一个“神经元”的输入. 例如, 图 1.4 就是一个简单的神经网络:

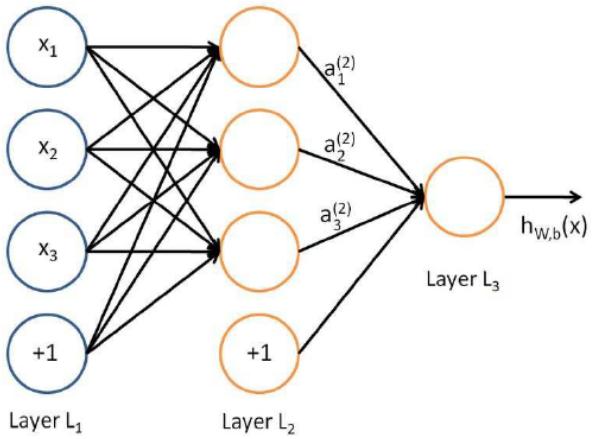


图 1.4 简单的神经网络

我们使用圆圈来表示神经网络的输入, 标上“+1”的圆圈被称为**偏置节点**, 也就是**截距项**. 神经网络最左边的一层叫做**输入层**, 最右边的一层叫做**输出层**(本例中, 输出层只有一个节点). 中间所有节点组成的一层叫做**隐藏层**(因为我们不能在训练样本集中观测到它们的值). 同时可以看到, 以上神经网络的例子中有 3 个输入单元 (偏置单元不计在内), 3 个隐藏单元及 1 个输出单元.

我们用 n_l 来表示**网络的层数**, 本例中 $n_l = 3$, 我们将**第 l 层**记为 L_l, 于是 L_1 是输入层, L_{n_l} 是输出层. 本例神经网络有参数 $(W, \mathbf{b}) = (W^{(1)}, \mathbf{b}^{(1)}, W^{(2)}, \mathbf{b}^{(2)})$, 其中 $W_{ij}^{(l)}$ (下面的式子中将用到) 是第 l 层第 j 单元与第 $l+1$ 层第 i 单元之间的**联接参数**(其实就是连接线上的权重, 注意标号顺序), $b_i^{(l)}$ 是第 $l+1$ 层第 i 单元的**偏置项**. 因此在本例中, $W^{(1)} \in R^{3\times 3}$, $W^{(2)} \in R^{1\times 3}$. 注意, 没有其他单元连向偏置单元 (即偏置单元没有输入), 因为它们总是输出 +1. 同时, 我们用 s_l 表示**第 l 层的节点数**(偏置单元不计在内).

我们用 $a_i^{(l)}$ 表示第 l 层第 i 单元的**激活值**(输出值). 当 $l = 1$ 时, $a_i^{(1)} = x_i$, 也就是第 i 个输入值 (输入值的第 i 个特征). 对于给定参数集合 $W, \mathbf{b}$, 我们的神经网络就可以按照函数

$h_{W,\mathbf{b}}(\mathbf{x})$ 来计算输出结果. 本例神经网络的计算步骤如下:

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

其中的 f 已扩展为**向量表示**, 即 $f((z_1, z_2, z_3)^T) = (f(z_1), f(z_2), f(z_3))^T$.

我们将上面的计算步骤叫作**前向传播**. 回想一下, 之前我们用 $\mathbf{a}^{(1)} = \mathbf{x}$ 表示输入层的激活值, 那么给定第 l 层的激活值 $\mathbf{a}^{(l)}$ 后, 第 $l+1$ 层的激活值 $\mathbf{a}^{(l+1)}$ 就可以按照下面步骤计算得到:

$$\mathbf{z}^{(l+1)} = W^{(l)}\mathbf{a}^{(l)} + \mathbf{b}^{(l)}, \tag{1.1.6}$$

$$\mathbf{a}^{(l+1)} = f(\mathbf{z}^{(l+1)}). \tag{1.1.7}$$

将参数矩阵化, 使用**矩阵 - 向量**运算方式, 我们就可以利用**线性代数**的优势对神经网络进行快速求解.

到目前为止, 我们讨论了一种神经网络, 我们也可以构建另一种结构的神经网络 (这里结构指的是神经元之间的联接模式), 也就是包含**多个隐藏层**的神经网络. 最常见的一个例子是 n_l 层的神经网络, 第 1 层是输入层, 第 n_l 层是输出层, 中间的每个层 l 与层 $l+1$ 紧密相联. 这种模式下, 要计算神经网络的输出结果, 我们可以按照之前描述的等式, 按部就班, 进行前向传播, 逐一计算第 L_2 层的所有激活值, 然后是第 L_3 层的激活值, 以此类推, 直到第 L_{n_l} 层. 这是一个**前馈神经网络** (Feedforward Neural Network, FNN) 的例子, 因为这种联接图没有**闭环**或**回路**.

注 1.1.3 *前馈神经网络是人工神经网络设计的最初的也是可论证的最简单的一种. 在这种网络中, 信息只会在一个方向流动 — 向前, 从输入单元通过隐藏单元 (如果有的话) 然后到达输出单元, 在网络中没有周期或者循环.*

神经网络也可以有**多个输出单元**. 比如, 下面的神经网络有两层隐藏层: L_2 及 L_3, 输出层 L_4 有两个输出单元.

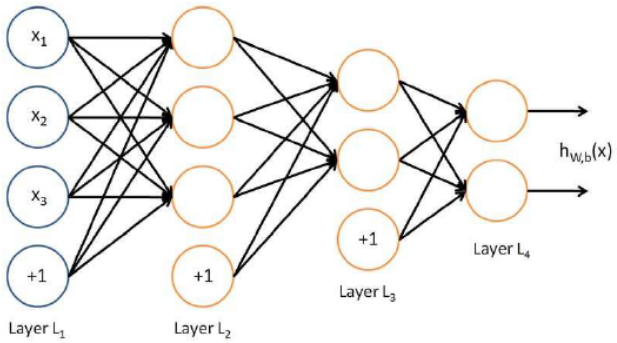


图 1.5 包含两个输出单元的神经网络

要求解这样的神经网络, 需要样本集 $(\mathbf{x}^{(i)}, \mathbf{y}^{(i)})$, 其中 $\mathbf{y}^{(i)} \in R^2$. 如果你想预测的输出是多个的, 那这种神经网络很适用. 比如, 在医疗诊断应用中, 患者的体征指标就可以作为向量的输入值, 而不同的输出值 y_j 可以表示不同的疾病存在与否.

§1.2 反向传导算法

假设我们有一个固定样本集 $\{(\mathbf{x}^{(1)},\mathbf{y}^{(1)}),(\mathbf{x}^{(2)},\mathbf{y}^{(2)}),\cdots,(\mathbf{x}^{(m)},\mathbf{y}^{(m)})\}$, 它包含 m 个样例. 我们可以用**批量梯度下降法**来求解神经网络. 具体来讲, 对于单个样例 $(\mathbf{x},\mathbf{y})$, 定义其**代价函数**为:

$$J(W,\mathrm{b};\mathbf{x},\mathbf{y})=\frac{1}{2}\|h_{W,\mathrm{b}}(\mathbf{x})-\mathbf{y}\|^2, \tag{1.2.8}$$

这是一个 (二分之一的)**方差代价函数**.

给定一个包含 m 个样例的数据集, 我们可以定义**整体代价函数**为:

$$\begin{aligned} J(W,\mathrm{b}) &= \left[\frac{1}{m}\sum_{i=1}^{m}J(W,\mathrm{b};\mathbf{x}^{(i)},\mathbf{y}^{(i)})\right]+\frac{\lambda}{2}\sum_{l=1}^{n_l-1}\sum_{i=1}^{s_l}\sum_{j=1}^{s_{l+1}}(W_{ji}^{(l)})^2 \\ &= \left[\frac{1}{m}\sum_{i=1}^{m}\Big(\frac{1}{2}\|h_{W,\mathrm{b}}(\mathbf{x}^{(i)})-\mathbf{y}^{(i)}\|^2\Big)\right]+\frac{\lambda}{2}\sum_{l=1}^{n_l-1}\sum_{i=1}^{s_l}\sum_{j=1}^{s_{l+1}}(W_{ji}^{(l)})^2. \end{aligned} \tag{1.2.9}$$

以上公式中的第一项是一个**均方差项**. 第二项是一个**规则化项**(也叫**权重衰减项**), 其目的是减小权重的幅度, **防止过度拟合**.

注 1.2.1 通常权重衰减的计算并不使用偏置项 $b_i^{(l)}$, 比如我们在 $J(W,\mathrm{b})$ 的定义中就没有使用. 一般来说, 将偏置项包含在权重衰减项中只会对最终的神经网络产生很小的影响. 如果你在斯坦福选修过 *CS229 (机器学习)* 课程, 或者在 *YouTube* 上看过课程视频, 你会发现这个权重衰减实际上是课上提到的**贝叶斯规则化方法**的变种. 在贝叶斯规则化方法中, 我们将**高斯先验概率**引入到参数中计算 *MAP (极大后验)* 估计 *(而不是极大似然估计)*.

注 1.2.2 文献 *[1]* 中有提到: *Instead, the magnitude of the weights in the network is more important. The smaller the weights are, the better generalization performance the network tends to have.* 因此, 在代价函数中增加衰减项可以防止过拟合, 提高**泛化能力**.

权重衰减参数 λ 用于控制公式中两项的**相对重要性**. 在此重申一下这两个复杂函数的含义: $J(W,\mathrm{b};\mathbf{x},\mathbf{y})$ 是针对单个样例计算得到的方差代价函数; $J(W,\mathrm{b})$ 是整体样本代价函数, 它包含权重衰减项.

以上的代价函数经常被用于**分类**和**回归**问题. 在分类问题中, 我们用 $y=0$ 或 1, 来代表两种类型的标签 (回想一下, 这是因为 sigmoid 激活函数的值域为 $[0,1]$; 如果我们使用双曲正切型激活函数, 那么应该选用 -1 和 $+1$ 作为标签). 对于回归问题, 我们首先要变换输出

值域 (译者注: 也就是 y), 以保证其范围为 $[0,1]$. 同样地, 如果我们使用双曲正切型激活函数, 要使输出值域为 $[-1,1]$.

我们的目标是针对参数 W 和 $\mathbf{b}$ 来求其函数 $J(W,\mathbf{b})$ 的最小值. 为了求解神经网络, 我们需要将每一个参数 $W_{ij}^{(l)}$ 和 $b_i^{(l)}$ 初始化为一个很小的、接近零的随机值 (比如说, 使用正态分布 $Normal(0,\epsilon^2)$ 生成的随机值, 其中 ϵ 设置为 0.01), 之后对目标函数使用诸如批量梯度下降法的最优化算法. 因为 $J(W,\mathbf{b})$ 是一个**非凸函数**, 梯度下降法很可能会收敛到**局部最优解**; 但是在实际应用中, 梯度下降法通常能得到令人满意的结果.

最后, 需要再次强调的是, 要将参数进行随机初始化, 而不是全部置为 0. 如果所有参数都用相同的值作为初始值, 那么所有隐藏层单元最终会得到与输入值有关的、相同的函数 (也就是说, 对于所有 i, $W_{ij}^{(1)}$ 都会取相同的值, 那么对于任何输入 $\mathbf{x}$ 都会有: $a_1^{(2)}=a_2^{(2)}=a_3^{(2)}=\cdots$). 随机初始化的目的是**使对称失效** (symmetry breaking).

梯度下降法中每一次迭代都按照如下公式对参数 W 和 $\mathbf{b}$ 进行更新:

$$W_{ij}^{(l)} = W_{ij}^{(l)} - \alpha\frac{\partial J(W,\mathbf{b})}{\partial W_{ij}^{(l)}}; \tag{1.2.10}$$

$$b_i^{(l)} = b_i^{(l)} - \alpha\frac{\partial J(W,\mathbf{b})}{\partial b_i^{(l)}}, \tag{1.2.11}$$

其中 α 是**学习速率**, $i=1,2,\cdots,s_{l+1}$, $j=1,2,\cdots,s_l$, $l=1,2,\cdots,n_l-1$. 这里关键步骤是计算偏导数. 我们现在来讲一下**反向传播算法**, 它是计算偏导数的一种有效方法.

首先, 我们利用反向传播算法来计算 $\frac{\partial J(W,\mathbf{b};\mathbf{x},\mathbf{y})}{\partial W_{ij}^{(l)}}$ 和 $\frac{\partial J(W,\mathbf{b};\mathbf{x},\mathbf{y})}{\partial b_i^{(l)}}$, 这两项是单个样例 $(\mathbf{x},\mathbf{y})$ 的代价函数 $J(W,\mathbf{b};\mathbf{x},\mathbf{y})$ 的偏导数. 一旦我们求出该偏导数, 就可以推导出整体代价函数 $J(W,\mathbf{b})$ 的偏导数:

$$\frac{\partial J(W,\mathbf{b})}{\partial W_{ij}^{(l)}} = \left[\frac{1}{m}\sum_{k=1}^{m}\frac{\partial J(W,\mathbf{b};\mathbf{x}^{(k)},\mathbf{y}^{(k)})}{\partial W_{ij}^{(l)}}\right] + \lambda W_{ij}^{(l)}; \tag{1.2.12}$$

$$\frac{\partial J(W,\mathbf{b})}{\partial b_i^{(l)}} = \frac{1}{m}\sum_{k=1}^{m}\frac{\partial J(W,\mathbf{b};\mathbf{x}^{(k)},\mathbf{y}^{(k)})}{\partial b_i^{(l)}} \tag{1.2.13}$$

以上两行公式稍有不同, 第一行比第二行多出一项, 是因为权重衰减是作用于 W 而不是 $\mathbf{b}$.

反向传播算法的思路如下: 给定一个样例 $(\mathbf{x},\mathbf{y})$, 我们首先进行“前向传导”运算, 计算出网络中所有的激活值, 包括 $h_{W,\mathbf{b}}(\mathbf{x})$ 的输出值. 之后, 针对第 l 层的每一个节点 i, 我们计算出其“残差” $\delta_i^{(l)}$, 该残差表明了该节点对最终输出值的残差产生了多少影响. 对于最终的输出节点, 可以直接算出网络产生的激活值与实际值之间的差距, 我们将这个差距定义为 $\delta_i^{(n_l)}$ (第 n_l 层表示输出层). 对于隐藏单元, 我们如何处理呢? 我们将基于第 $l+1$ 层节点残差的加权平均值来计算 $\delta_i^{(l)}$, 这些节点以 $a_i^{(l)}$ 作为输入.

注 1.2.3 *关于为什么引入“残差” $\delta_i^{(l)}$ 的理解:*

利用求导链式法则, 我们有

$$\begin{aligned}\frac{\partial J(W,\mathrm{b};\mathrm{x},\mathrm{y})}{\partial W_{ij}^{(l)}} &= \frac{\partial J(W,\mathrm{b};\mathrm{x},\mathrm{y})}{\partial z_i^{(l+1)}}\frac{\partial z_i^{(l+1)}}{\partial W_{ij}^{(l)}},\\ \frac{\partial J(W,\mathrm{b};\mathrm{x},\mathrm{y})}{\partial b_i^{(l)}} &= \frac{\partial J(W,\mathrm{b};\mathrm{x},\mathrm{y})}{\partial z_i^{(l+1)}}\frac{\partial z_i^{(l+1)}}{\partial b_i^{(l)}},\end{aligned}$$

根据公式 *(1.1.6)*, $z_i^{(l+1)}$ 可表示为

$$z_i^{(l+1)} = \sum_{k=1}^{s_l} W_{ik}^{(l)} a_k^{(l)} + b_i^{(l)},$$

故有

$$\frac{\partial z_i^{(l+1)}}{\partial W_{ij}^{(l)}} = a_j^{(l)}, \quad \frac{\partial z_i^{(l+1)}}{\partial b_i^{(l)}} = 1.$$

若令 $\delta_i^{(l)} := \frac{\partial J(W,\mathrm{b};\mathrm{x},\mathrm{y})}{\partial z_i^{(l)}}$, 则有

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

由此知, 要计算 $\frac{\partial J(W,\mathrm{b};\mathrm{x},\mathrm{y})}{\partial W_{ij}^{(l)}}$ 和 $\frac{\partial J(W,\mathrm{b};\mathrm{x},\mathrm{y})}{\partial b_i^{(l)}}$, 关键是计算 $\delta_i^{(l+1)}$.

下面将给出反向传导算法的细节.

1. 进行前馈传导计算, 利用前向传导公式, 得到 $L_2, L_3, \cdots$ 直到输出层 L_{n_l} 的激活值 $a_i^{(l)}$.
2. 对于第 n_l 层 (输出层) 的每个输出单元 i, 根据以下公式计算残差:

$$\delta_i^{(n_l)} = \frac{\partial J(W,\mathrm{b};\mathrm{x},\mathrm{y})}{\partial z_i^{(n_l)}} = -(y_i - a_i^{(n_l)})\cdot f'(z_i^{(n_l)}). \tag{1.2.14}$$

推导过程如下:

$$\begin{aligned}\delta_i^{(n_l)} &= \frac{\partial J(W,\mathrm{b};\mathrm{x},\mathrm{y})}{\partial z_i^{(n_l)}} = \frac{\partial}{\partial z_i^{(n_l)}}\left(\frac{1}{2}\|\mathrm{y} - h_{W,\mathrm{b}}(\mathrm{x})\|^2\right) \quad (J(W,\mathrm{b};\mathrm{x},\mathrm{y}) \text{ 的定义})\\ &= \frac{1}{2}\frac{\partial}{\partial z_i^{(n_l)}}\left(\sum_{j=1}^{s_{n_l}}(y_j - a_j^{(n_l)})^2\right) \quad (\text{向量范数 } \|\cdot\| \text{ 的定义})\\ &= \frac{1}{2}\frac{\partial}{\partial z_i^{(n_l)}}\left(\sum_{j=1}^{s_{n_l}}(y_j - f(z_j^{(n_l)}))^2\right) \quad (\text{利用 } a_j^{(n_l)} = f(z_j^{(n_l)}))\\ &= -(y_i - f(z_i^{(n_l)}))\cdot f'(z_i^{(n_l)}) \quad (\text{求和项对 } z_i^{(n_l)} \text{ 求偏导})\\ &= -(y_i - a_i^{(n_l)})\cdot f'(z_i^{(n_l)}) \quad (\text{利用 } a_i^{(n_l)} = f(z_i^{(n_l)}))\end{aligned}$$

3. 对 $l = n_l - 1, n_l - 2, \cdots, 2$ 的各个层, 第 l 层的第 i 个节点的残差计算方法如下:

$$\delta_i^{(l)} = \left(\sum_{j=1}^{s_{l+1}} W_{ji}^{(l)} \delta_j^{(l+1)}\right) \cdot f'(z_i^{(l)}). \tag{1.2.15}$$

推导过程如下:

$$\begin{aligned}
\delta_i^{(n_l-1)} &= \frac{\partial J(W,\mathbf{b};\mathbf{x},\mathbf{y})}{\partial z_i^{(n_l-1)}} = \frac{\partial}{\partial z_i^{(n_l-1)}}\left(\frac{1}{2}\|\mathbf{y} - h_{W,\mathbf{b}}(\mathbf{x})\|^2\right) \quad (J(W,\mathbf{b};\mathbf{x},\mathbf{y}) \text{ 的定义}) \\
&= \frac{1}{2}\frac{\partial}{\partial z_i^{(n_l-1)}}\left(\sum_{j=1}^{s_{n_l}}(y_j - a_j^{(n_l)})^2\right) \quad (\text{向量范数 } \|\cdot\| \text{ 的定义}) \\
&= \frac{1}{2}\sum_{j=1}^{s_{n_l}}\frac{\partial}{\partial z_i^{(n_l-1)}}(y_j - a_j^{(n_l)})^2 \quad (\text{将求偏导数移到求和号里面}) \\
&= \frac{1}{2}\sum_{j=1}^{s_{n_l}}\frac{\partial}{\partial z_i^{(n_l-1)}}(y_j - f(z_j^{(n_l)}))^2 \quad (\text{利用 } a_j^{(n_l)} = f(z_j^{(n_l)})) \\
&= \sum_{j=1}^{s_{n_l}} -(y_j - f(z_j^{(n_l)}))\frac{\partial f(z_j^{(n_l)})}{\partial z_i^{(n_l-1)}} \quad (\text{求偏导数}) \\
&= \sum_{j=1}^{s_{n_l}} -(y_j - f(z_j^{(n_l)})) \cdot f'(z_j^{(n_l)}) \cdot \frac{\partial z_j^{(n_l)}}{\partial z_i^{(n_l-1)}} \quad (\text{利用链式法则求偏导数 } \tfrac{\partial f(z_j^{(n_l)})}{\partial z_i^{(n_l-1)}}) \\
&= \sum_{j=1}^{s_{n_l}} \delta_j^{(n_l)} \cdot \frac{\partial z_j^{(n_l)}}{\partial z_i^{(n_l-1)}} \quad (\text{利用 (1.2.14) 式}) \\
&= \sum_{j=1}^{s_{n_l}} \delta_j^{(n_l)} \cdot \frac{\partial}{\partial z_i^{(n_l-1)}}\left(\sum_{k=1}^{s_{n_l-1}} f(z_k^{(n_l-1)}) \cdot W_{jk}^{(n_l-1)} + b_j^{(n_l-1)}\right) \quad (\text{将 } z_j^{(n_l)} \text{ 的定义代入}) \\
&= \sum_{j=1}^{s_{n_l}} \delta_j^{(n_l)} \cdot W_{ji}^{(n_l-1)} \cdot f'(z_i^{(n_l-1)}) \quad (\text{对 } z_i^{(n_l-1)} \text{ 求偏导数}) \\
&= \left(\sum_{j=1}^{s_{n_l}} W_{ji}^{(n_l-1)} \cdot \delta_j^{(n_l)}\right) \cdot f'(z_i^{(n_l-1)})
\end{aligned}$$

将上式中 $n_l - 1$ 与 n_l 的关系替换为 l 与 $l+1$ 的关系, 就可以得到 (1.2.15).

以上逐次从后向前求导的过程即为 “反向传导” 的本意所在.

4. 计算我们需要的偏导数, 计算方法如下:

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

为更好地理解上述几个步骤, 我们以一个样本 $(\mathbf{x},\mathbf{y})$ 为例, 图 1.6 用箭头给出相应计算流程的示意图.

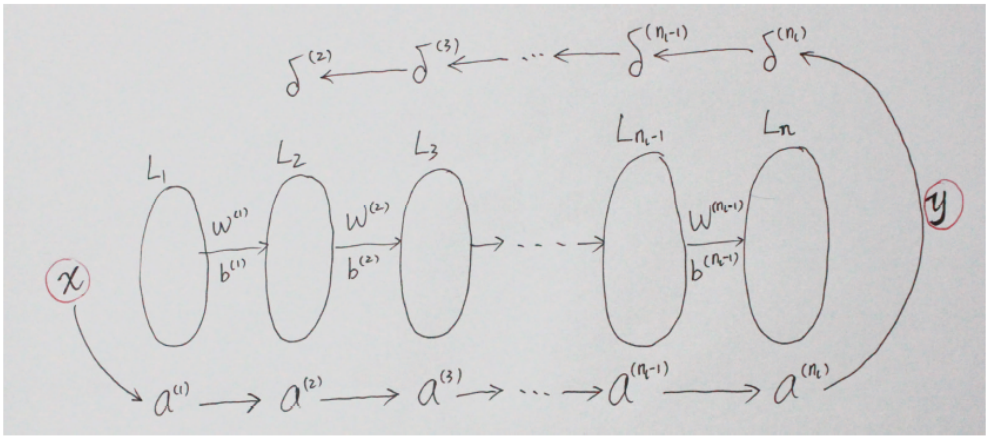


图 1.6 BP 算法计算流程示意图

接下来, 我们利用**矩阵 - 向量表示法**重写以上算法.

记 "$\circ$" 表示向量乘积运算符 (在 Matlab 或 Octave 里用 ".$*$" 表示, 也称作阿达马乘积 (Hadamard product)). 若 $\mathbf{a} = \mathbf{b} \circ \mathbf{c}$, 则 $a_i = b_i c_i$. 在上一节中我们扩展了 $f(\cdot)$ 的定义, 使其包含向量运算, 这里对偏导数 $f'(\cdot)$ 也作同样的扩展, 即有 $f'((z_1, z_2, z_3)^T) = (f'(z_1), f'(z_2), f'(z_3))^T$.

利用上述符号, 反向传播算法可表示为以下几个步骤:

1. 进行前馈传导计算, 利用前向传导公式, 得到 $L_2, L_3, \cdots$ 直到输出层 L_{n_l} 的激活值.

2. 对输出层 (第 n_l 层), 计算:

$$\delta^{(n_l)} = -(\mathbf{y} - \mathbf{a}^{(n_l)}) \circ f'(\mathbf{z}^{(n_l)}).$$

3. 对 $l = n_l - 1, n_l - 2, \cdots, 2$ 的各层, 计算:

$$\delta^{(l)} = \left((W^{(l)})^T \delta^{(l+1)}\right) \circ f'(\mathbf{z}^{(l)}).$$

4. 计算最终需要的偏导数值:

$$\begin{aligned} \nabla_{W^{(l)}} J(W, \mathbf{b}; \mathbf{x}, \mathbf{y}) &= \delta^{(l+1)} (\mathbf{a}^{(l)})^T; \\ \nabla_{\mathbf{b}^{(l)}} J(W, \mathbf{b}; \mathbf{x}, \mathbf{y}) &= \delta^{(l+1)}. \end{aligned}$$

实现中应注意: 在以上的第 2 步和第 3 步中, 我们需要为每一个 i 计算其 $f'(z_i^{(l)})$. 假设 $f(z)$ 是 sigmoid 函数, 并且我们已经在前向传导运算中得到了 $a_i^{(l)}$, 那么, 使用我们早先推导出的 $f'(z)$ 表达式 (1.1.4), 就可以计算得到

$$f'(z_i^{(l)}) = f(z_i^{(l)})(1 - f(z_i^{(l)})) = a_i^{(l)}(1 - a_i^{(l)}).$$

最后, 我们对**梯度下降算法**做个全面总结.

在下面的伪代码中, $\triangle W^{(l)}$ 是一个与矩阵 $W^{(l)}$ 维度相同的矩阵, $\triangle \mathbf{b}^{(l)}$ 是一个与 $\mathbf{b}^{(l)}$ 维度相同的向量. 注意这里 "$\triangle W^{(l)}$" 是一个矩阵, 而不是 "$\triangle$ 与 $W^{(l)}$ 相乘". 下面, 我们实现**批量梯度下降法**中的一次迭代:

算法 1.2.1 （批量梯度下降法的一次迭代）

1. 令 $\triangle W^{(l)} := 0$, $\triangle \mathbf{b}^{(l)} := 0$, $l = 1, 2, \cdots, n_l - 1$.

2. 对 $i = 1, 2, \cdots, m$ 循环, 执行

(a) 令 $\mathbf{x} := \mathbf{x}^{(i)}$, $\mathbf{y} := \mathbf{y}^{(i)}$.

(b) 使用反向传播算法计算 $\nabla_{W^{(l)}} J(W, \mathbf{b}; \mathbf{x}, \mathbf{y})$ 和 $\nabla_{\mathbf{b}^{(l)}} J(W, \mathbf{b}; \mathbf{x}, \mathbf{y})$, $l = 1, 2, \cdots, n_l - 1$.

(c) 对 $l = 1, 2, \cdots, n_l - 1$ 循环, 执行

(c1) $\triangle W^{(l)} := \triangle W^{(l)} + \nabla_{W^{(l)}} J(W, \mathbf{b}; \mathbf{x}, \mathbf{y})$.

(c2) $\triangle \mathbf{b}^{(l)} := \triangle \mathbf{b}^{(l)} + \nabla_{\mathbf{b}^{(l)}} J(W, \mathbf{b}; \mathbf{x}, \mathbf{y})$.

3. 更新权重参数: 对 $l = 1, 2, \cdots, n_l - 1$ 循环, 执行

(a) $W^{(l)} := W^{(l)} - \alpha \left[\frac{1}{m} \triangle W^{(l)} + \lambda W^{(l)} \right]$;

(b) $\mathbf{b}^{(l)} := \mathbf{b}^{(l)} - \alpha \left[\frac{1}{m} \triangle \mathbf{b}^{(l)} \right]$.

现在, 我们可以重复梯度下降法的迭代步骤来减小代价函数 $J(W, \mathbf{b})$ 的值, 进而求解我们的神经网络.

§1.3 自编码算法与稀疏性

目前为止, 我们已经讨论了神经网络在**有监督学习**中的应用. 在有监督学习中, 训练样本是有**类别标签**的. 现在假设我们只有一个没有带类别标签的训练样本集合 $\{\mathbf{x}^{(1)}, \mathbf{x}^{(2)}, \cdots\}$, 其中 $\mathbf{x}^{(i)} \in R^n$. **自编码神经网络**是一种**无监督学习算法**, 它使用了反向传播算法, 并让目标值等于输入值, 即 $\mathbf{y}^{(i)} = \mathbf{x}^{(i)}$. 图 1.7 是一个自编码神经网络的示例.

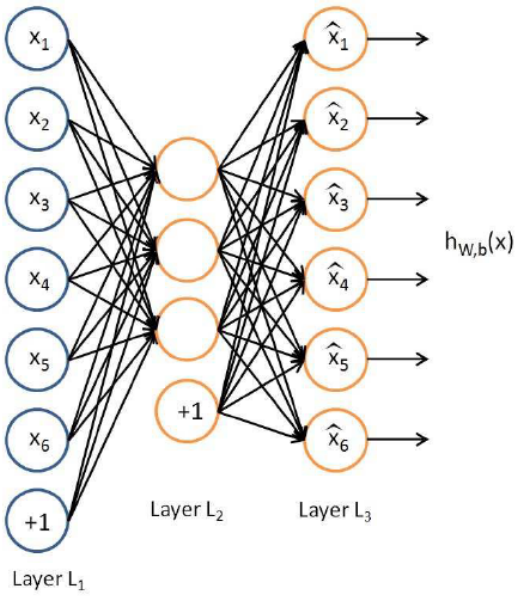


图 1.7 自编码神经网络的示例

自编码神经网络尝试学习一个 $h_{W,b}(\mathbf{x}) \approx \mathbf{x}$ 的函数. 换句话说, 它尝试逼近一个恒等函数, 从而使得输出 $\hat{\mathbf{x}}$ 接近于输入 $\mathbf{x}$. 恒等函数虽然看上去不太有学习的意义, 但是当我们为自编码神经网络**加入某些限制**, 比如**限定隐藏神经元的数量**, 我们就可以**从输入数据中发现一些有趣的结构**.

举例来说, 假设某个自编码神经网络的输入 $\mathbf{x}$ 是一张 10×10 图像 (共 100 个像素) 的像素灰度值, 于是 $n = 100$, 其隐藏层 L_2 中有 50 个隐藏神经元. 注意, 输出也是 100 维的, $\mathbf{y} \in R^{100}$. 由于只有 50 个隐藏神经元, 我们迫使自编码神经网络去学习**输入数据的压缩表示**, 也就是说, 它必须从 50 维的隐藏神经元激活度向量 $\mathbf{a}^{(2)}$ 中重构出 100 维的像素灰度值输入 $\mathbf{x}$.

如果网络的输入数据是完全随机的, 比如每一个输入 x_i 都是一个跟其它特征完全无关的独立同分布 (independently and identically distribute, IID) 高斯随机变量, 那么这一压缩表示将会非常难学习. 但是, 如果输入数据中隐含着一些特定的结构, 比如某些输入特征是彼此相关的, 那么, 这一算法就可以发现输入数据中的这些相关性. 事实上, 这一简单的自编码神经网络通常可以学习出一个跟**主元分析** (PCA) 结果非常相似的输入数据的**低维表示**.

我们刚才的论述是基于隐藏神经元数量较小的假设. 然而, 即使隐藏神经元的数量较大 (可能比输入像素的个数还要多), 我们仍然可以通过给自编码神经网络施加一些其他的限制条件来发现输入数据中的结构. 具体来说, 如果我们给隐藏神经元加入**稀疏性限制**, 那么自编码神经网络即使在隐藏神经元数量较多的情况下仍然可以发现输入数据中一些有趣的结构.

稀疏性可以被简单地解释如下. 如果当神经元的输出接近于 1 的时候我们认为它被激活, 而输出接近于 0 的时候认为它被抑制, 那么, **使得神经元大部分的时间都是被抑制的限制**则被称作**稀疏性限制**. 这里, 我们假设的神经元的激活函数是 sigmoid 函数. 如果你使用 tanh 作为激活函数的话, 当神经元输出为 -1 的时候, 我们认为神经元是被抑制的.

注意到 $a_j^{(2)}$ 表示隐藏神经元 j 的激活度, 但是这一表示方法中并未明确指出哪一个输入 $\mathbf{x}$ 带来了这一激活度. 所以, 我们将使用 $a_j^{(2)}(\mathbf{x})$ 来表示在给定输入为 $\mathbf{x}$ 的情况下, 自编码神经网络隐藏神经元 j 的激活度. 进一步, 让

$$\hat{\rho}_j = \frac{1}{m} \sum_{i=1}^{m} \left[a_j^{(2)}(\mathbf{x}^{(i)}) \right] \tag{1.3.16}$$

表示隐藏神经元 j 的**平均活跃度**(在训练集上取平均). 我们可以近似地加入一条限制

$$\hat{\rho}_j = \rho, \tag{1.3.17}$$

其中, ρ 是**稀疏性参数**, 通常是一个接近于 0 的较小的值 (比如 $\rho = 0.05$). 换句话说, 我们想要让隐藏神经元 j 的平均活跃度接近 0.05. 为了满足这一条件, 隐藏神经元的活跃度必须接近于 0.

为了实现这一限制, 我们将会在**优化目标函数**中加入一个额外的**惩罚因子**, 而这一惩罚因子将惩罚那些 $\hat{\rho}_j$ 和 ρ 有显著不同的情况, 从而使得隐藏神经元的平均活跃度保持在较小

范围内. 惩罚因子的具体形式有很多种合理的选择, 我们将会选择以下这一种:

$$\sum_{j=1}^{s_2}\left[\rho * \log\frac{\rho}{\hat{\rho}_j} + (1-\rho) * \log\frac{1-\rho}{1-\hat{\rho}_j}\right], \tag{1.3.18}$$

这里, s_2 是隐藏层中隐藏神经元的数量, 而索引 j 依次代表隐藏层中的每一个神经元. 如果你对**相对熵** (KL divergence) 比较熟悉, 这一惩罚因子实际上是基于它的. 于是, 惩罚因子也可以被表示为

$$\sum_{j=1}^{s_2} KL(\rho||\hat{\rho}_j), \tag{1.3.19}$$

其中 $KL(\rho||\hat{\rho}_j) = \left[\rho * \log\frac{\rho}{\hat{\rho}_j} + (1-\rho) * \log\frac{1-\rho}{1-\hat{\rho}_j}\right]$ 是一个以 ρ 为均值和一个以 $\hat{\rho}_j$ 为均值的两个**伯努利随机变量**之间的相对熵. **相对熵是一种标准的用来测量两个分布之间差异的方法**. (如果你没有见过相对熵, 不用担心, 所有你需要知道的内容都会被包含在这份笔记之中.)

这一惩罚因子有如下性质, 当 $\hat{\rho}_j = \rho$ 时 $KL(\rho||\hat{\rho}_j) = 0$, 并且随着 $\hat{\rho}_j$ 与 ρ 之间的差异增大而单调递增. 举例来说, 在下图中, 我们设定 $\rho = 0.2$, 并且画出了相对熵值 $KL(\rho||\hat{\rho}_j)$ 随着 $\hat{\rho}_j$ 变化的变化.

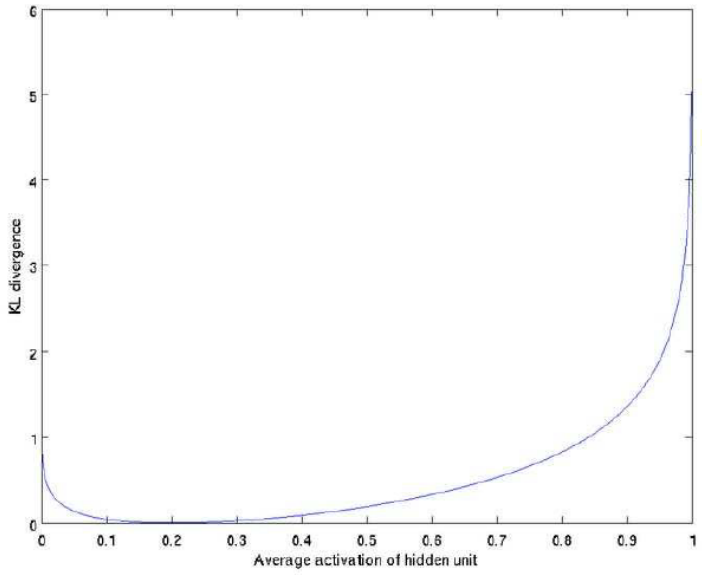


图 1.8 $KL(\rho||\hat{\rho}_j)$ 随 $\hat{\rho}_j$ 变化的曲线图

可以看出, 相对熵在 $\hat{\rho}_j = \rho$ 时达到它的最小值 0, 而当 $\hat{\rho}_j$ 靠近 0 或者 1 的时候, 相对熵则变得非常大 (其实是趋向于 ∞). 所以, **最小化这一惩罚因子具有使得 $\hat{\rho}_j$ 靠近 ρ 的效果**.

现在, 我们的总体代价函数可以表示为

$$J_{sparse}(W, \mathbf{b}) = J(W, \mathbf{b}) + \beta\sum_{j=1}^{s_2} KL(\rho||\hat{\rho}_j), \tag{1.3.20}$$

其中 $J(W,\mathbf{b})$ 如之前所定义, 而 β **控制稀疏性惩罚因子的权重**. $\hat{\rho}_j$ 项则也 (间接地) 取决于 $W,\mathbf{b}$, 因为它是隐藏神经元 j 的平均激活度, 而隐藏层神经元的激活度取决于 $W,\mathbf{b}$.

为了对相对熵进行导数计算, 我们可以使用一个易于实现的技巧, 这只需要在你的程序中稍作改动即可. 具体来说, 前面在后向传播算法中计算第二层 ($l=2$) 更新的时候我们已经计算了

$$\delta_i^{(2)}=\left(\sum_{j=1}^{s_3}W_{ji}^{(2)}\delta_j^{(3)}\right)f'(z_i^{(2)}), \tag{1.3.21}$$

现在我们将其换成

$$\delta_i^{(2)}=\left\{\left(\sum_{j=1}^{s_3}W_{ji}^{(2)}\delta_j^{(3)}\right)+\beta\left(-\frac{\rho}{\hat{\rho}_i}+\frac{1-\rho}{1-\hat{\rho}_i}\right)\right\}f'(z_i^{(2)}) \tag{1.3.22}$$

就可以了.

注 1.3.1 *补充关于 $\frac{\partial J_{sparse}(W,\mathbf{b})}{\partial W_{i,j}^{(l)}}$ 和 $\frac{\partial J_{sparse}(W,\mathbf{b})}{\partial b_i^{(l)}}$ 的推导*

记 $S(W,\mathbf{b})=\sum_{j=1}^{s_2}KL(\rho||\hat{\rho}_j)$, 则 (1.3.20) 可写为

$$J_{sparse}(W,\mathbf{b})=J(W,\mathbf{b})+\beta S(W,\mathbf{b}), \tag{1.3.23}$$

从而梯度下降法中涉及的两个偏导数分别为

$$\begin{cases}\frac{\partial J_{sparse}(W,\mathbf{b})}{\partial W_{i,j}^{(l)}} &= \frac{\partial J(W,\mathbf{b})}{\partial W_{i,j}^{(l)}}+\beta\frac{\partial S(W,\mathbf{b})}{\partial W_{i,j}^{(l)}};\\ \frac{\partial J_{sparse}(W,\mathbf{b})}{\partial b_i^{(l)}} &= \frac{\partial J(W,\mathbf{b})}{\partial b_i^{(l)}}+\beta\frac{\partial S(W,\mathbf{b})}{\partial b_i^{(l)}}.\end{cases} \tag{1.3.24}$$

关于 $\frac{\partial J(W,\mathbf{b})}{\partial W_{i,j}^{(l)}}$ 和 $\frac{\partial J(W,\mathbf{b})}{\partial b_i^{(l)}}$ 的计算, 上一节已经讨论过了, 这里仅讨论 $\frac{\partial S(W,\mathbf{b})}{\partial W_{i,j}^{(l)}}$ 和 $\frac{\partial S(W,\mathbf{b})}{\partial b_i^{(l)}}$ 的计算.

首先展开 $S(W,\mathbf{b})$, 有

$$S(W,\mathbf{b})=\sum_{t=1}^{s_2}KL(\rho||\hat{\rho}_t)=\sum_{t=1}^{s_2}\left[\rho*\log\frac{\rho}{\hat{\rho}_t}+(1-\rho)*\log\frac{1-\rho}{1-\hat{\rho}_t}\right], \tag{1.3.25}$$

其中

$$\hat{\rho}_t=\frac{1}{m}\sum_{k=1}^{m}a_t^{(2)}(\mathbf{x}^{(k)})=\frac{1}{m}\sum_{k=1}^{m}f(z_t^{(2)}), \tag{1.3.26}$$

这里

$$z_t^{(2)}=z_t^{(2)}(\mathbf{x}^{(k)})=\left(\sum_{s=1}^{s_1}W_{t,s}^{(1)}x_s^{(k)}\right)+b_t^{(1)}. \tag{1.3.27}$$

由 (1.3.26) 和 (1.3.27) 可知以下两点:

1. 当 $l \neq 1$ 时，$\frac{\partial S(W,\mathbf{b})}{\partial W_{i,j}^{(l)}} = 0$，$\frac{\partial S(W,\mathbf{b})}{\partial b_i^{(l)}} = 0$.

2. $\frac{\partial S(W,\mathbf{b})}{\partial W_{i,j}^{(l)}} = \frac{\partial \sum_{t=1}^{s_2} KL(\rho||\hat{\rho}_t)}{\partial W_{i,j}^{(l)}} = \frac{\partial KL(\rho||\hat{\rho}_i)}{\partial W_{i,j}^{(l)}}$；$\frac{\partial S(W,\mathbf{b})}{\partial b_i^{(l)}} = \frac{\partial \sum_{t=1}^{s_2} KL(\rho||\hat{\rho}_t)}{\partial b_i^{(l)}} = \frac{\partial KL(\rho||\hat{\rho}_i)}{\partial b_i^{(l)}}$.

根据上述两条，有

$$\begin{aligned}
&\frac{\partial S(W,\mathbf{b})}{\partial W_{i,j}^{(1)}} = \frac{\partial}{\partial W_{i,j}^{(1)}}\left[\rho * \log\frac{\rho}{\hat{\rho}_i} + (1-\rho) * \log\frac{1-\rho}{1-\hat{\rho}_i}\right] \\
= &\frac{\partial}{\partial W_{i,j}^{(1)}}\{\rho(\log\rho - \log\hat{\rho}_i) + (1-\rho)[\log(1-\rho) - \log(1-\hat{\rho}_i)]\} \quad (\log\frac{A}{B} = \log A - \log B) \\
= &\rho(0 - \frac{1}{\hat{\rho}_i}\frac{\partial\hat{\rho}_i}{\partial W_{i,j}^{(1)}}) + (1-\rho)(0 + \frac{1}{1-\hat{\rho}_i}\frac{\partial\hat{\rho}_i}{\partial W_{i,j}^{(1)}}) \quad (\rho \text{ 为常数}) \\
= &(-\frac{\rho}{\hat{\rho}_i} + \frac{1-\rho}{1-\hat{\rho}_i})\frac{\partial\hat{\rho}_i}{\partial W_{i,j}^{(1)}}
\end{aligned}$$

类似地，有

$$\frac{\partial S(W,\mathbf{b})}{\partial b_i^{(1)}} = (-\frac{\rho}{\hat{\rho}_i} + \frac{1-\rho}{1-\hat{\rho}_i})\frac{\partial\hat{\rho}_i}{\partial b_i^{(1)}}.$$

接下来，我们只需求出 $\frac{\partial\hat{\rho}_i}{\partial W_{i,j}^{(1)}}$ 和 $\frac{\partial\hat{\rho}_i}{\partial b_i^{(1)}}$ 即可. 由 *(1.3.26)*, 有

$$\begin{aligned}
\frac{\partial\hat{\rho}_i}{\partial W_{i,j}^{(1)}} &= \frac{1}{m}\sum_{k=1}^{m} f'(z_i^{(2)}) \cdot \frac{\partial z_i^{(2)}}{\partial W_{i,j}^{(1)}} \\
&= \frac{1}{m}\sum_{k=1}^{m} f'(z_i^{(2)}) \cdot x_j^{(k)} \quad (\text{由 (1.3.27) 有 } \frac{\partial z_i^{(2)}}{\partial W_{i,j}^{(1)}} = x_j^{(k)})
\end{aligned}$$

类似地，有

$$\begin{aligned}
\frac{\partial\hat{\rho}_i}{\partial b_i^{(1)}} &= \frac{1}{m}\sum_{k=1}^{m} f'(z_i^{(2)}) \cdot \frac{\partial z_i^{(2)}}{\partial b_i^{(1)}} \\
&= \frac{1}{m}\sum_{k=1}^{m} f'(z_i^{(2)}) \quad (\text{由 (1.3.27) 有 } \frac{\partial z_i^{(2)}}{\partial b_i^{(1)}} = 1)
\end{aligned}$$

综上，我们求得

$$\begin{cases}
\frac{\partial S(W,\mathbf{b})}{\partial W_{i,j}^{(1)}} = (-\frac{\rho}{\hat{\rho}_i} + \frac{1-\rho}{1-\hat{\rho}_i}) \cdot \frac{1}{m}\sum_{k=1}^{m} f'(z_i^{(2)}) \cdot x_j^{(k)}; \\
\frac{\partial S(W,\mathbf{b})}{\partial b_i^{(1)}} = (-\frac{\rho}{\hat{\rho}_i} + \frac{1-\rho}{1-\hat{\rho}_i}) \cdot \frac{1}{m}\sum_{k=1}^{m} f'(z_i^{(2)}).
\end{cases} \tag{1.3.28}$$

在上一节中，我们已经算得

$$\begin{cases}
\frac{\partial J(W,\mathbf{b})}{\partial W_{ij}^{(1)}} &= \left[\frac{1}{m}\sum_{k=1}^{m} a_j^{(1)}\delta_i^{(2)}\right] + \lambda W_{ij}^{(1)} \\
\frac{\partial J(W,\mathbf{b})}{\partial b_i^{(1)}} &= \frac{1}{m}\sum_{k=1}^{m}\delta_i^{(2)}.
\end{cases} \tag{1.3.29}$$

由此并结合 *(1.3.24), (1.3.28)*, 可得

$$\begin{cases} \frac{\partial J_{sparse}(W,\mathrm{b})}{\partial W_{i,j}^{(1)}} &= \left[\frac{1}{m}\sum_{k=1}^{m} a_j^{(1)}\delta_i^{(2)}\right] + \lambda W_{ij}^{(1)} + \beta(-\frac{\rho}{\hat{\rho}_i} + \frac{1-\rho}{1-\hat{\rho}_i}) \cdot \frac{1}{m}\sum_{k=1}^{m} f'(z_i^{(2)}) \cdot x_j^{(k)} \\ \frac{\partial J_{sparse}(W,\mathrm{b})}{\partial b_i^{(1)}} &= \frac{1}{m}\sum_{k=1}^{m} \delta_i^{(2)} + \beta(-\frac{\rho}{\hat{\rho}_i} + \frac{1-\rho}{1-\hat{\rho}_i}) \cdot \frac{1}{m}\sum_{k=1}^{m} f'(z_i^{(2)}). \end{cases} \tag{1.3.30}$$

需要注意的是, 求和号中的 $a_j^{(1)}$, $\delta_i^{(2)}$ 和 $z_i^{(2)}$ 均与 $\mathbf{x}^{(k)}$ (或 $\mathbf{y}^{(k)}$) 有关, 特别地, 有 $a_j^{(1)} = x_j^{(k)}$. 上述推导中将其省略主要是为了简单起见.

对 *(1.3.30)* 中的两个式子作进一步整理, 可得

$$\begin{aligned} & \frac{\partial J_{sparse}(W,\mathrm{b})}{\partial W_{i,j}^{(1)}} \\ = & \frac{1}{m}\sum_{k=1}^{m}\left[a_j^{(1)}\delta_i^{(2)} + x_j^{(k)}\beta(-\frac{\rho}{\hat{\rho}_i} + \frac{1-\rho}{1-\hat{\rho}_i})f'(z_i^{(2)})\right] + \lambda W_{ij}^{(1)} \\ = & \frac{1}{m}\sum_{k=1}^{m}\left\{a_j^{(1)}\left[\delta_i^{(2)} + \beta(-\frac{\rho}{\hat{\rho}_i} + \frac{1-\rho}{1-\hat{\rho}_i})f'(z_i^{(2)})\right]\right\} + \lambda W_{ij}^{(1)} \text{ (利用 } a_j^{(1)} = x_j^{(k)} \text{)} \end{aligned}$$

$$\frac{\partial J_{sparse}(W,\mathrm{b})}{\partial b_i^{(1)}} = \frac{1}{m}\sum_{k=1}^{m}\left[\delta_i^{(2)} + \beta(-\frac{\rho}{\hat{\rho}_i} + \frac{1-\rho}{1-\hat{\rho}_i})f'(z_i^{(2)})\right]$$

至此, 我们已经证明了正文中将 *(1.3.21)* 换为 *(1.3.22)* 的实现技巧.

有一个需要注意的地方是, 我们需要知道 $\hat{\rho}_i$ 来计算这一项更新. 所以, 在计算任何神经元的后向传播之前, 你需要对所有的训练样本计算一遍前向传播, 从而获取平均激活度. 如果你的训练样本可以小到被整个存到内存之中 (对于编程作业来说, 通常如此), 你可以方便地在你所有的样本上计算前向传播, 将得到的激活度存入内存并且计算平均激活度 $\hat{\rho}_i$. 然后, 你就可以使用事先计算好的激活度来对所有的训练样本进行后向传播的计算. 如果你的**数据量太大, 无法全部存入内存**, 你就可以扫过你的训练样本并计算一次前向传播, 然后将获得的结果累积起来并计算平均激活度 $\hat{\rho}_i$ (当某一个前向传播的结果中的激活度 $a_i^{(2)}$ 被用于计算平均激活度 $\hat{\rho}_i$ 之后就可以将此结果删除). 然后, 当你完成平均激活度 $\hat{\rho}_i$ 的计算之后, 你需要重新对每一个训练样本做一次前向传播从而可以对其进行后向传播的计算. **对于后一种情况, 你对每一个训练样本需要计算两次前向传播, 所以在计算上的效率会稍低一些**.

证明上面算法能达到梯度下降效果的完整推导过程不在本教程的范围之内. 不过, 如果你想要使用经过以上修改的后向传播来实现自编码神经网络, 那么你就会对目标函数 (1.3.20) 做梯度下降. 使用梯度验证方法, 你可以自己来验证梯度下降算法是否正确.

最后, 我们列出稀疏自编码器推导过程中用到的符号一览表.

- $\mathbf{x}$: 训练样本的输入特征, $\mathbf{x} \in R^n$.
- $\mathbf{y}$: 输出值/目标值, 这里 $\mathbf{y}$ 可以为标量也可以为向量, 在 autoencoder 中 $\mathbf{y} = \mathbf{x}$.

- $(\mathbf{x}^{(i)}, \mathbf{y}^{(i)})$: 第 i 个训练样本.
- $h_{W,\mathbf{b}}(\mathbf{x})$: 输入为 $\mathbf{x}$ 的假设输出, 其中包含参数 $W, \mathbf{b}$, 该输出应当与目标值 $\mathbf{y}$ 具有相同的维数.
- $W_{ij}^{(l)}$: 连接第 l 层 j 单元和第 $l+1$ 层 i 单元的参数.
- $b_i^{(l)}$: 第 $l+1$ 层 i 单元的偏置项, 也可看做是连接第 l 层偏置单元和第 $l+1$ 层 i 单元的参数.
- θ: 参数向量, 可以认为该向量是通过将参数 $W, \mathbf{b}$ 组合展开为一个长的列向量而得到.
- $a_i^{(l)}$: 第 l 层 i 单元的激活 (输出) 值. 另外, 由于 L_1 是输入层, 所有 $a_i^{(1)} = x_i$.
- $f(\cdot)$: 激活函数, 本文中我们使用 sigmoid 函数.
- $z_i^{(l)}$: 第 l 层 i 单元所有输入的加权和, 因此有 $a_i^{(l)} = f(z_i^{(l)})$.
- α: 学习率.
- s_l: 第 l 层的单元数目 (不包括偏置单元).
- n_l: 网络中的层数, 通常 L_1 层是输入层, L_{n_l} 层是输出层.
- λ: 权重衰减系数.
- $\hat{\mathbf{x}}$: 对于一个 autoencoder, 该符号表示其输出值; 亦即输入值的重构值, 与 $h_{W,\mathbf{b}}(\mathbf{x})$ 涵义相同.
- ρ: 稀疏值, 可以用它指定我们所需的稀疏程度.
- $\hat{\rho}_i$: (sparse autoencoder 中) 隐藏单元 i 的平均激活值.
- β: (sparse autoencoder 目标函数中) 稀疏值惩罚项的权重.