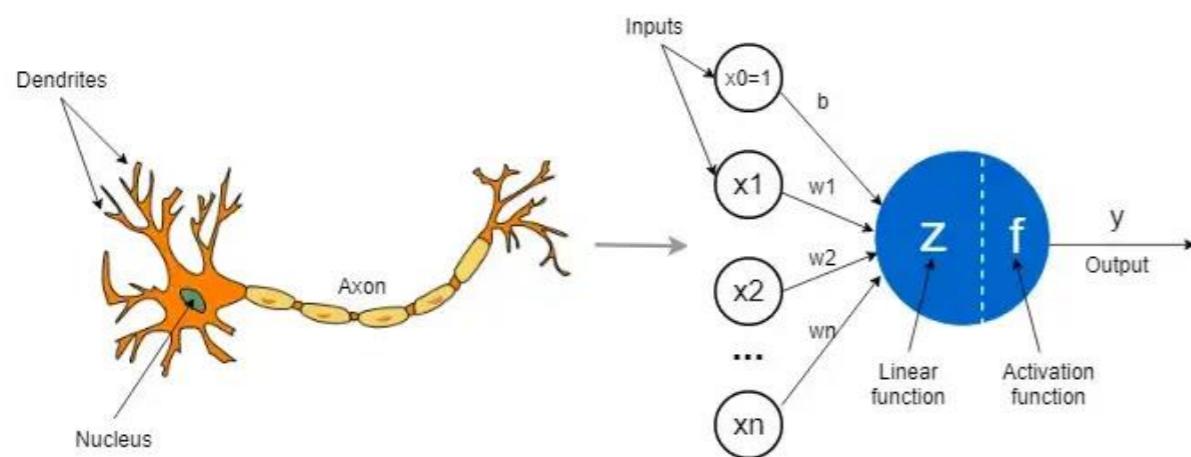


شبکه عصبی

Neural Networks

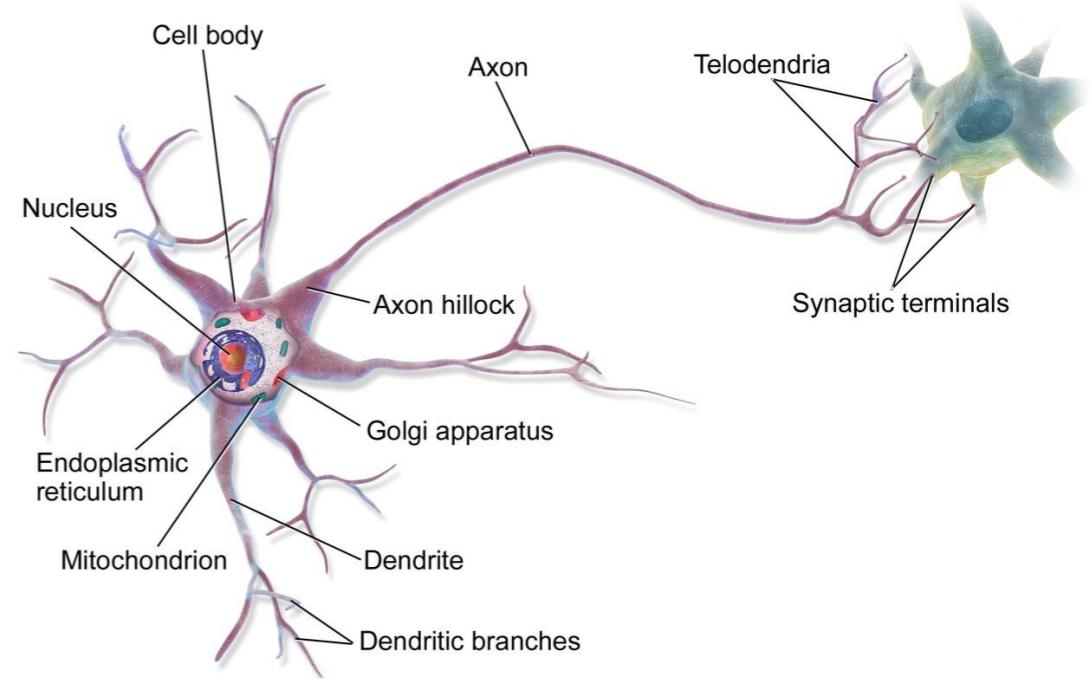
مغز انسان حدود 100 میلیارد نورون دارد که هر نورون بین 1000 تا 10000 اتصال به نورون های دیگر دارد. بنابراین، مغز انسان یک شبکه عصبی با 100 تریلیون تا یک کوادریلیون (10^{15}) اتصالات است. تا آنجا که می دانیم، یادگیری با تنظیم نقاط قوت سیناپسی اتفاق می افتد، زیرا سیگنال های سیناپسی می توانند تحريك کننده یا بازدارنده باشند، که باعث می شود نورون پس سیناپسی به ترتیب احتمال ایجاد پتانسیل عمل را کم یا زیاد کند.



شبکه های عصبی مصنوعی از نورون های انتزاعی تشکیل شده اند که سعی می کنند نورون های واقعی را در سطح بسیار بالایی تقليد کنند. آنها را می توان از طریق یک گراف جهت دار وزنی $G = (V, E)$ توصیف کرد که هر گره $v_i \in V$ یک نورون را نشان می دهد و هر یال جهت دار $(v_i, v_j) \in E$ نشان دهنده ارتباط سیناپسی از v_i به v_j است. وزن یا w_{ij} نشان دهنده قدرت سیناپسی است.

شبکه های عصبی با نوع تابع فعال ساز مورد استفاده برای تولید خروجی و معماری شبکه از نظر نحوه اتصال گره ها مشخص می شوند. توجه به این نکته ضروری است که یک شبکه عصبی برای نمایش و یادگیری اطلاعات با تنظیم وزن سیناپسی طراحی می شود.

یک نورون بیولوژیکی واقعی یا یک سلول عصبی شامل دندrit ها، یک جسم سلولی و یک آکسون است که به پایانه های سیناپسی منتهی می شود.

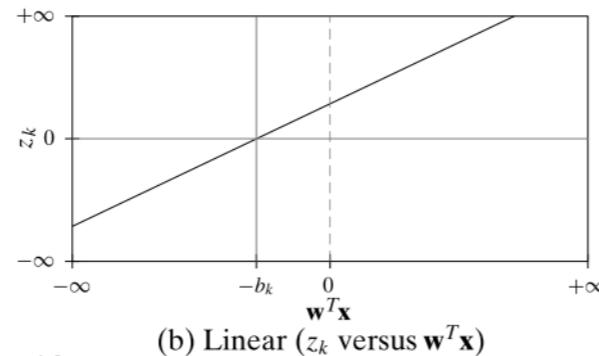
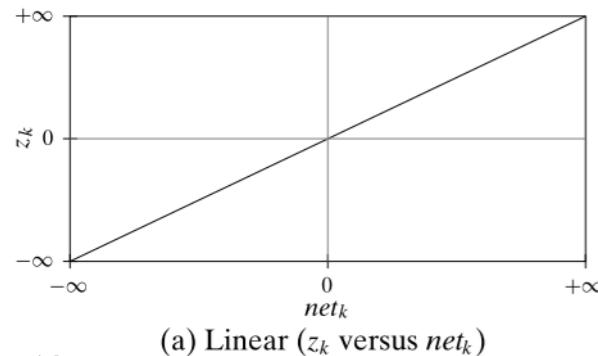


یک نورون اطلاعات را از طریق سیگنال های الکتروشیمیایی منتقل می کند. هنگامی که غلظت کافی یون در دندrit یک نورون وجود داشته باشد، یک پالس الکتریکی در امتداد آکسون خود به نام پتانسیل عمل (Action Potential) تولید می کند که به نوبه خود پایانه های سیناپسی را فعال می کند و یون های بیشتری آزاد می کند و در نتیجه باعث می شود که اطلاعات به دندrit های نورون های دیگر جریان یابد.

Identity/Linear Function

$$f(\text{net}_k) = \text{net}_k$$

$$\frac{\partial f(\text{net}_j)}{\partial \text{net}_j} = 1$$



Step Function

$$f(\text{net}_k) = \begin{cases} 0 & \text{if } \text{net}_k \leq 0 \\ 1 & \text{if } \text{net}_k > 0 \end{cases}$$

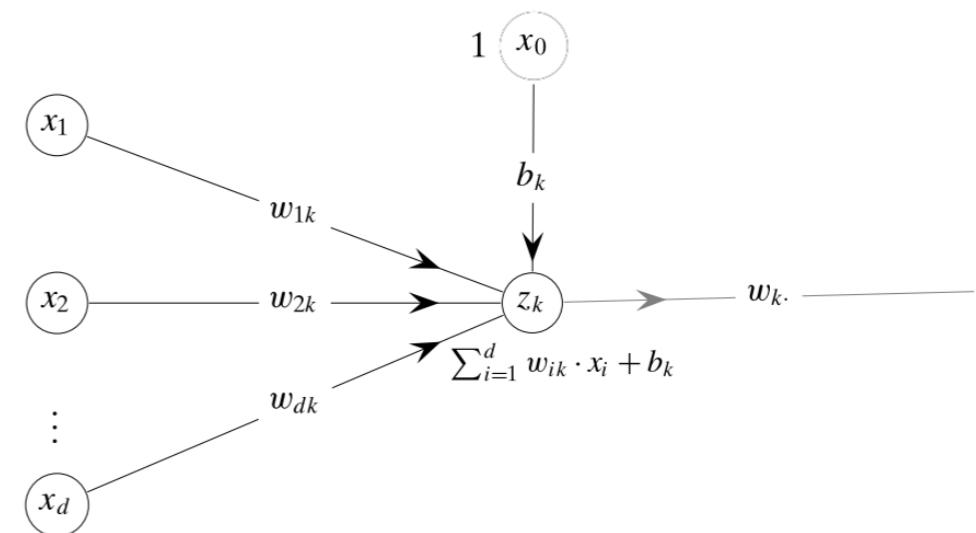
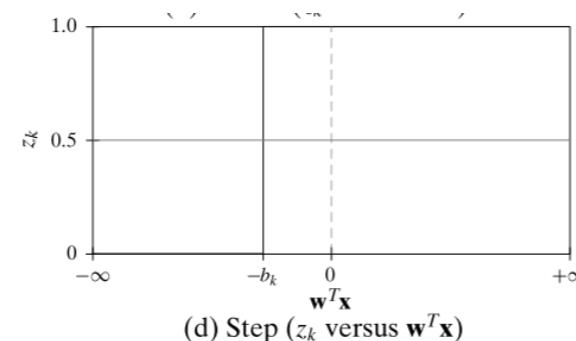
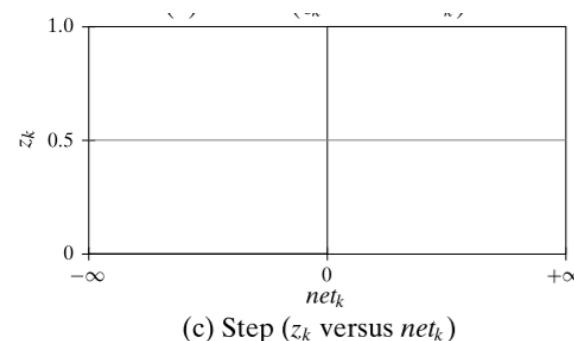


Figure 25.1. Artificial neuron: aggregation and activation.

یک نورون مصنوعی به عنوان یک واحد پردازش عمل می‌کند که ابتدا سیگنال‌های دریافتی را از طریق یک مجموع وزنی جمع می‌کند و سپس برخی از عملکردها را برای تولید یک خروجی اعمال می‌کند.

$$\text{net}_k = b_k + \sum_{i=1}^d w_{ik} \cdot x_i = b_k + \mathbf{w}^T \mathbf{x}$$

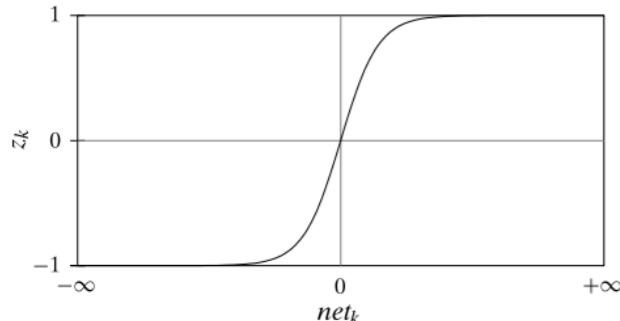
Activation Function

$$z_k = f(\text{net}_k)$$

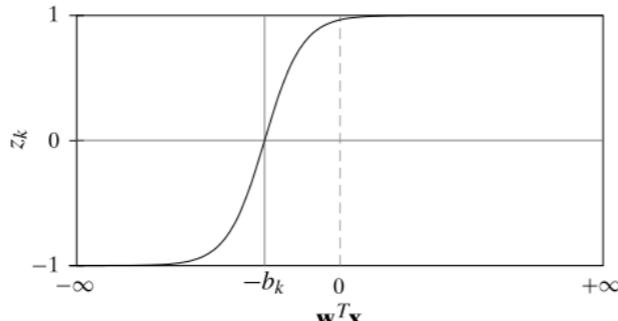
Hyperbolic Tangent (\tanh)

$$f(\text{net}_k) = \frac{\exp\{\text{net}_k\} - \exp\{-\text{net}_k\}}{\exp\{\text{net}_k\} + \exp\{-\text{net}_k\}} = \frac{\exp\{2 \cdot \text{net}_k\} - 1}{\exp\{2 \cdot \text{net}_k\} + 1}$$

$$\frac{\partial f(\text{net}_j)}{\partial \text{net}_j} = 1 - f(\text{net}_j)^2$$



(i) Hyperbolic Tangent (z_k versus net_k)

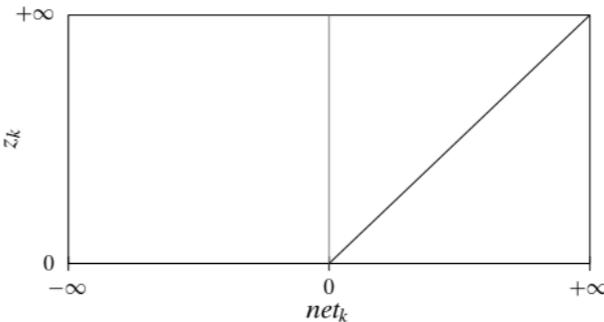


(j) Hyperbolic Tangent (z_k versus $w^T \mathbf{x}$)

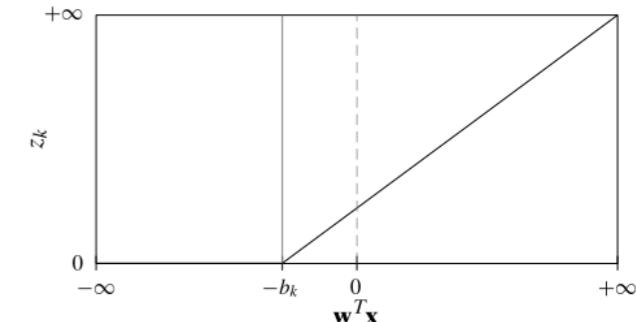
Rectified Linear Unit (ReLU)

$$f(\text{net}_k) = \max\{0, \text{net}_k\}$$

$$\frac{\partial f(\text{net}_j)}{\partial \text{net}_j} = \begin{cases} 0 & \text{if } \text{net}_j \leq 0 \\ 1 & \text{if } \text{net}_j > 0 \end{cases}$$



(e) Rectified Linear (z_k versus net_k)



(f) Rectified Linear (z_k versus $w^T \mathbf{x}$)

Sigmoid

$$f(\text{net}_k) = \frac{1}{1 + \exp\{-\text{net}_k\}}$$

$$\frac{\partial f(\text{net}_j)}{\partial \text{net}_j} = f(\text{net}_j) \cdot (1 - f(\text{net}_j))$$

: تعمیم تابع فعال سازی سیگموئید یا لجستیک است. عمدتاً در لایه خروجی در یک شبکه عصبی استفاده می شود و بخلاف سایر توابع، نه تنها به ورودی کل در نورون k بستگی دارد، بلکه به سیگنال کل در تمام نورون های دیگر در لایه خروجی نیز بستگی دارد.

$$\mathbf{net} = (\text{net}_1, \text{net}_2, \dots, \text{net}_p)^T$$

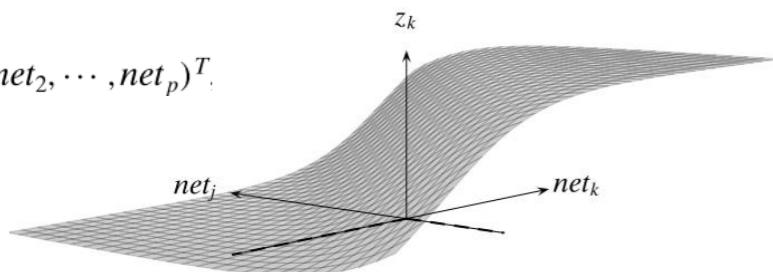
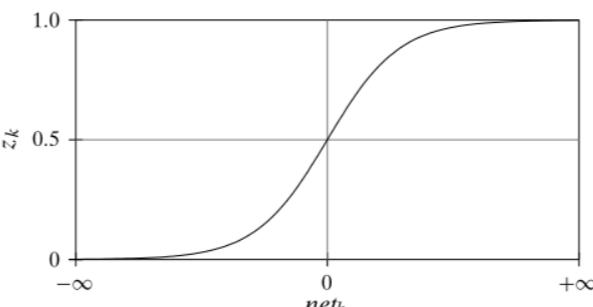


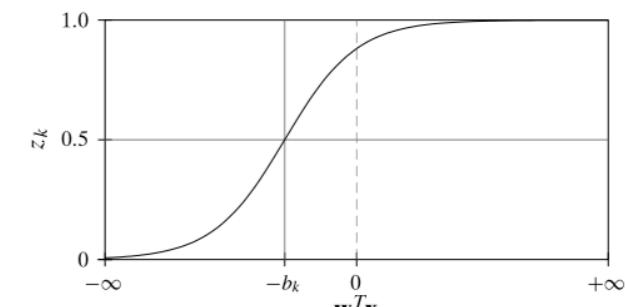
Figure 25.3. Softmax (net_k versus net_j).

$$f(\text{net}_k | \mathbf{net}) = \frac{\exp\{\text{net}_k\}}{\sum_{i=1}^p \exp\{\text{net}_i\}}$$

$$\frac{\partial f(\text{net}_j | \mathbf{net})}{\partial \text{net}_k} = \frac{\partial o_j}{\partial \text{net}_k} = \begin{cases} o_j \cdot (1 - o_j) & \text{if } k = j \\ -o_k \cdot o_j & \text{if } k \neq j \end{cases}$$



(g) Sigmoid (z_k versus net_k)



(h) Sigmoid (z_k versus $w^T \mathbf{x}$)

رگرسیون: Regression

Example 25.1 (Neural Networks for Multiple and Multivariate Regression).

Consider the multiple regression of sepal length and petal length on the dependent attribute petal width for the Iris dataset with $n = 150$ points. From Example 23.3 we find that the solution is given as

$$\hat{y} = -0.014 - 0.082 \cdot x_1 + 0.45 \cdot x_2$$

The squared error for this optimal solution is 6.179 on the training data.

Using the neural network in Figure 25.4(a), with linear activation for the output and minimizing the squared error via gradient descent, results in the following learned parameters, $b = 0.0096$, $w_1 = -0.087$ and $w_2 = 0.452$, yielding the regression model

$$o = 0.0096 - 0.087 \cdot x_1 + 0.452 \cdot x_2$$

with a squared error of 6.18, which is very close to the optimal solution.

Multivariate Linear Regression For multivariate regression, we use the neural network architecture in Figure 25.4(b) to learn the weights and bias for the Iris dataset, where we use sepal length and sepal width as the independent attributes, and petal length and petal width as the response or dependent attributes. Therefore, each input point \mathbf{x}_i is 2-dimensional, and the true response vector \mathbf{y}_i is also 2-dimensional. That is, $d = 2$ and $p = 2$ specify the size of the input and output layers. Minimizing the squared error via gradient descent, yields the following parameters:

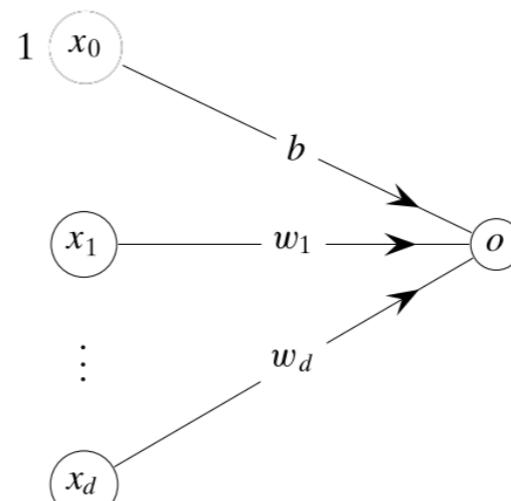
$$\begin{pmatrix} b_1 & b_2 \\ w_{11} & w_{12} \\ w_{21} & w_{22} \end{pmatrix} = \begin{pmatrix} -1.83 & -1.47 \\ 1.72 & 0.72 \\ -1.46 & -0.50 \end{pmatrix} \quad \begin{pmatrix} o_1 \\ o_2 \end{pmatrix} = \begin{pmatrix} -1.83 + 1.72 \cdot x_1 - 1.46 \cdot x_2 \\ -1.47 + 0.72 \cdot x_1 - 0.50 \cdot x_2 \end{pmatrix}$$

The squared error on the training set is 84.9. Optimal least squared multivariate regression yields a squared error of 84.16 with the following parameters

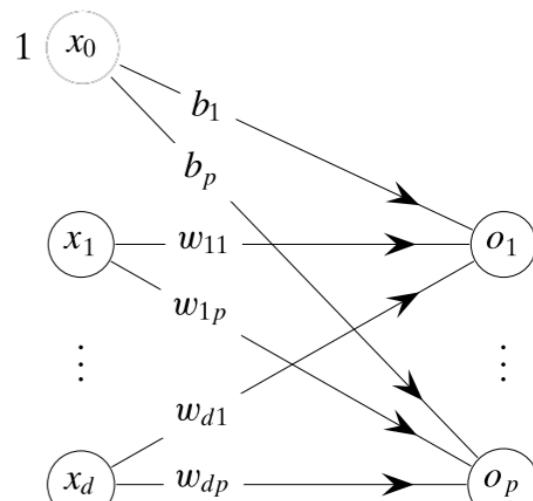
$$\begin{pmatrix} \hat{y}_1 \\ \hat{y}_2 \end{pmatrix} = \begin{pmatrix} -2.56 + 1.78 \cdot x_1 - 1.34 \cdot x_2 \\ -1.59 + 0.73 \cdot x_1 - 0.48 \cdot x_2 \end{pmatrix}$$

$$\hat{y}_i = b + w_1 x_{i1} + w_2 x_{i2} + \cdots + w_d x_{id}$$

$$SSE = \sum_{i=1}^n (y_i - \hat{y}_i)^2$$



(a) Single Output



(b) Multiple Outputs

Figure 25.4. Linear and logistic regression via neural networks.

برای محاسبهٔ خطابازی هر داده، خروجی پیش‌بینی شده $\hat{\mathbf{y}} = \mathbf{o} = (o_1, o_2, \dots, o_p)^T$ با بردار پاسخ واقعی $\mathbf{y} = (y_1, y_2, \dots, y_p)^T$ مقایسه می‌شود و سپس روی تمام داده‌ها جمع زده می‌شود.

$$\mathcal{E}_{\mathbf{x}} = \frac{1}{2} \|\mathbf{y} - \mathbf{o}\|^2 = \frac{1}{2} \sum_{j=1}^p (y_j - o_j)^2$$

$$\mathcal{E} = \sum_{i=1}^n \mathcal{E}_{\mathbf{x}_i} = \frac{1}{2} \cdot \sum_{i=1}^n \|\mathbf{y}_i - \mathbf{o}_i\|^2$$

کلاس‌بندی (Classification)

از تابع فعال‌ساز Softmax برای گرهی خروجی و برای خطا از تابع خطای K-way Cross-Entropy استفاده می‌کنیم.

$$\mathcal{E}_{\mathbf{x}} = - \left(y_1 \cdot \ln(o_1) + \cdots + y_K \cdot \ln(o_K) \right) \quad \mathbf{y} \in \{\mathbf{e}_1, \mathbf{e}_2, \dots, \mathbf{e}_K\}$$

$$o_i = P(\mathbf{y} = \mathbf{e}_i | \mathbf{x}) = f(\text{net}_i | \mathbf{net}) = \frac{\exp\{\text{net}_i\}}{\sum_{j=1}^p \exp\{\text{net}_j\}} = \pi_i(\mathbf{x})$$

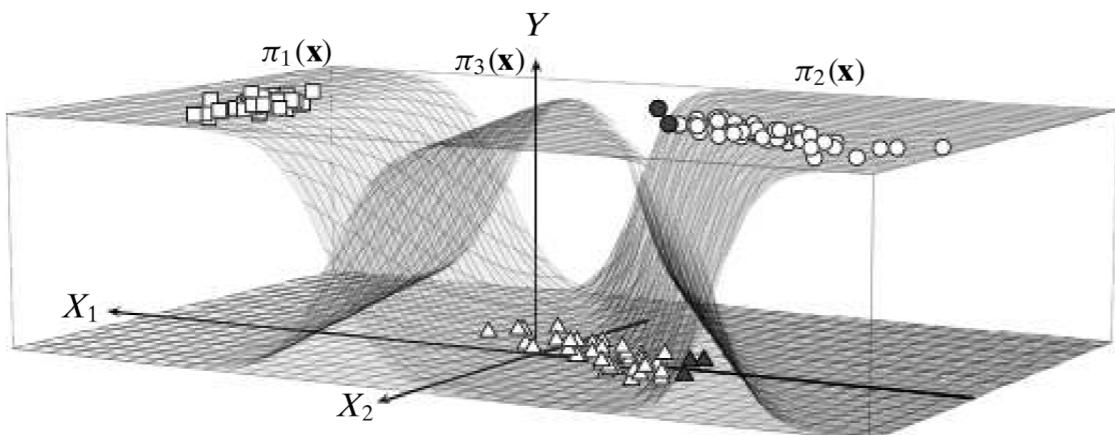


Figure 25.5. Neural networks for multiclass logistic regression: Iris principal components data. Misclassified point are shown in dark gray color. Points in class c_1 and c_2 are shown displaced with respect to the base class c_3 only for illustration.

احتمال پیشامد کلاس $y = 1$ برای داده \mathbf{x} را با تابع سیگموید مدل می‌کنیم و پیشامد $0 = 1$ متمم آن خواهد بود.

$$\pi(\mathbf{x}) = P(y = 1 | \mathbf{x}) = \frac{1}{1 + \exp\{-(b + \mathbf{w}^T \mathbf{x})\}}$$

$$P(y = 0 | \mathbf{x}) = 1 - P(y = 1 | \mathbf{x}) = 1 - \pi(\mathbf{x})$$

از تابع فعال‌ساز سیگموید برای گرهی خروجی و بجای میانگین مربع خطا از خطای Cross-Entropy استفاده می‌کنیم.

$$\mathcal{E}_{\mathbf{x}} = - \left(y \cdot \ln(o) + (1 - y) \cdot \ln(1 - o) \right)$$

$$o = f(\text{net}_o) = \text{sigmoid}(b + \mathbf{w}^T \mathbf{x}) = \frac{1}{1 + \exp\{-(b + \mathbf{w}^T \mathbf{x})\}} = \pi(\mathbf{x})$$

رگرسیون منطقی چندکلاسه (Multiclass Logistic Regression)

احتمال پیشامد بردار تمام K کلاس از طریق تابع Softmax مدل می‌شود.

$$\pi_i(\mathbf{x}) = \frac{\exp\{b_i + \mathbf{w}_i^T \mathbf{x}\}}{\sum_{j=1}^K \exp\{b_j + \mathbf{w}_j^T \mathbf{x}\}}, \quad \mathbf{w}_i \in \mathbb{R}^d$$

توابع خطا

(Squared Error) خطای مربعی

$$\mathcal{E}_x = \frac{1}{2} \|\mathbf{y} - \mathbf{o}\|^2 = \frac{1}{2} \sum_{j=1}^p (y_j - o_j)^2$$

$$\frac{\partial \mathcal{E}_x}{\partial o_j} = \frac{1}{2} \cdot 2 \cdot (y_j - o_j) \cdot -1 = o_j - y_j$$

$$\frac{\partial \mathcal{E}_x}{\partial \mathbf{o}} = \mathbf{o} - \mathbf{y}$$

Cross-Entropy Error

$$\mathcal{E}_x = - \sum_{i=1}^K y_i \cdot \ln(o_i) = - \left(y_1 \cdot \ln(o_1) + \dots + y_K \cdot \ln(o_K) \right)$$

$$\frac{\partial \mathcal{E}_x}{\partial o_j} = -\frac{y_j}{o_j} \quad \frac{\partial \mathcal{E}_x}{\partial \mathbf{o}} = \left(\frac{\partial \mathcal{E}_x}{\partial o_1}, \frac{\partial \mathcal{E}_x}{\partial o_2}, \dots, \frac{\partial \mathcal{E}_x}{\partial o_K} \right)^T = \left(-\frac{y_1}{o_1}, -\frac{y_2}{o_2}, \dots, -\frac{y_K}{o_K} \right)^T$$

Binary Cross-Entropy Error

$$\mathcal{E}_x = -(y \cdot \ln(o) + (1 - y) \cdot \ln(1 - o))$$

$$y \in \{0, 1\}$$

$$\frac{\partial \mathcal{E}_x}{\partial o} = \frac{o - y}{o \cdot (1 - o)}$$

Example 25.2 (Logistic Regression: Binary and Multiclass). We applied the neural network in Figure 25.4(a), with logistic activation at the output neuron and cross-entropy error function, on the Iris principal components dataset. The output is a binary response indicating Iris-virginica ($Y=1$) or one of the other Iris types ($Y=0$). As expected, the neural network learns an identical set of weights and bias as shown for the logistic regression model in Example 24.2, namely:

$$o = -6.79 - 5.07 \cdot x_1 - 3.29 \cdot x_2$$

Next, we applied the neural network in Figure 25.4(b), using a softmax activation and cross-entropy error function, to the Iris principal components data with three classes: Iris-setosa ($Y=1$), Iris-versicolor ($Y=2$) and Iris-virginica ($Y=3$). Thus, we need $K=3$ output neurons, o_1, o_2 , and o_3 . Further, to obtain the same model as in the multiclass logistic regression from Example 24.3, we fix the incoming weights and bias for output neuron o_3 to be zero. The model is given as

$$o_1 = -3.49 + 3.61 \cdot x_1 + 2.65 \cdot x_2$$

$$o_2 = -6.95 - 5.18 \cdot x_1 - 3.40 \cdot x_2$$

$$o_3 = 0 + 0 \cdot x_1 + 0 \cdot x_2$$

which is essentially the same as in Example 24.3.

If we do not constrain the weights and bias for o_3 we obtain the following model:

$$o_1 = -0.89 + 4.54 \cdot x_1 + 1.96 \cdot x_2$$

$$o_2 = -3.38 - 5.11 \cdot x_1 - 2.88 \cdot x_2$$

$$o_3 = 4.24 + 0.52 \cdot x_1 + 0.92 \cdot x_2$$

The classification decision surface for each class is illustrated in Figure 25.5. The points in class c_1 are shown as squares, c_2 as circles, and c_3 as triangles. This figure should be contrasted with the decision boundaries shown for multiclass logistic regression in Figure 24.3, which has the weights and bias set to 0 for the base class c_3 .

$$\mathbf{net}_h = \mathbf{b}_h + \mathbf{W}_h^T \mathbf{x}$$

$$\mathbf{z} = f(\mathbf{net}_h) = f(\mathbf{b}_h + \mathbf{W}_h^T \mathbf{x})$$

$$\mathbf{W}_h = \begin{pmatrix} w_{11} & w_{12} & \cdots & w_{1m} \\ w_{21} & w_{22} & \cdots & w_{2m} \\ \vdots & \vdots & \cdots & \vdots \\ w_{d1} & w_{d2} & \cdots & w_{dm} \end{pmatrix}$$

$$\mathbf{b}_h = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_m \end{pmatrix}$$

$$\mathbf{net}_o = \mathbf{b}_o + \mathbf{W}_o^T \mathbf{z}$$

$$\mathbf{o} = f(\mathbf{net}_o) = f(\mathbf{b}_o + \mathbf{W}_o^T \mathbf{z})$$

$$\mathbf{W}_o = \begin{pmatrix} w_{11} & w_{12} & \cdots & w_{1p} \\ w_{21} & w_{22} & \cdots & w_{2p} \\ \vdots & \vdots & \cdots & \vdots \\ w_{m1} & w_{m2} & \cdots & w_{mp} \end{pmatrix}$$

$$\mathbf{b}_o = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_p \end{pmatrix}$$

$$\mathbf{o} = f(\mathbf{b}_o + \mathbf{W}_o^T \mathbf{z}) = f(\mathbf{b}_o + \mathbf{W}_o^T \cdot f(\mathbf{b}_h + \mathbf{W}_h^T \mathbf{x}))$$

Feed-forward Phase

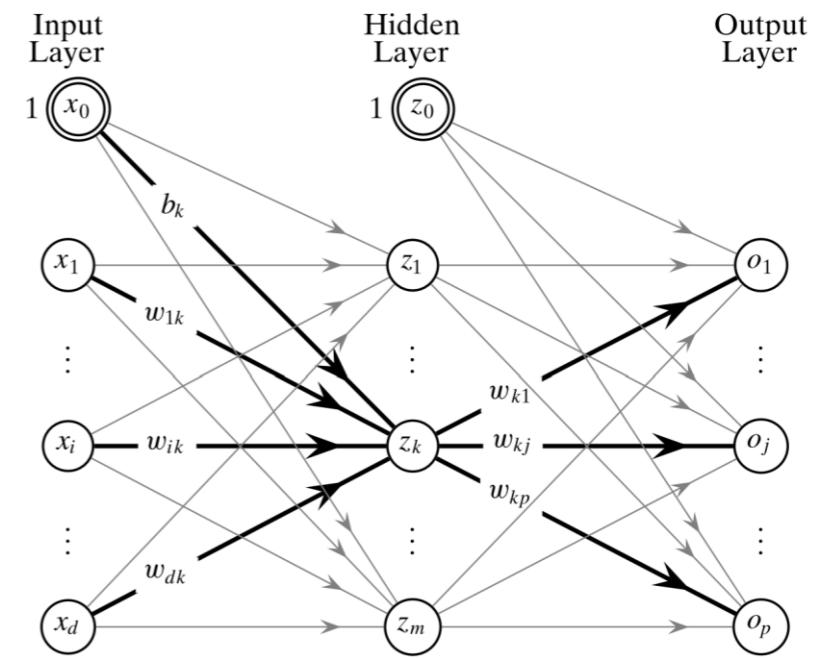


Figure 25.6. Multilayer perceptron with one hidden layer. Input and output links for neuron z_k are shown in bold. Neurons x_0 and z_0 are bias neurons.

$$\mathbf{x} = (x_1, x_2, \dots, x_d)^T \in \mathbb{R}^d$$

$$\mathbf{y}_i \in \mathbb{R}^p$$

$$\mathbf{z} = (z_1, z_2, \dots, z_m)^T$$

$$\mathbf{net}_h = (net_1, \dots, net_m)^T$$

$$z_k = f(net_k) = f\left(b_k + \sum_{i=1}^d w_{ik} \cdot x_i\right)$$

$$o_j = f(net_j) = f\left(b_j + \sum_{i=1}^m w_{ij} \cdot z_i\right)$$

$$\mathbf{W}_o = \mathbf{W}_o - \eta \cdot \nabla_{\mathbf{W}_o}$$

$$\mathbf{b}_o = \mathbf{b}_o - \eta \cdot \nabla_{\mathbf{b}_o}$$

$$\nabla_{\mathbf{W}_o} = \begin{pmatrix} \nabla_{w_{11}} & \nabla_{w_{12}} & \dots \nabla & w_{1p} \\ \nabla_{w_{21}} & \nabla_{w_{22}} & \dots \nabla & w_{2p} \\ \vdots & \vdots & \dots & \vdots \\ \nabla_{w_{m1}} & \nabla_{w_{m2}} & \dots \nabla & w_{mp} \end{pmatrix} = \mathbf{z} \cdot \delta_o^T$$

$$\nabla_{\mathbf{b}_o} = (\nabla_{b_1}, \nabla_{b_2}, \dots, \nabla_{b_p})^T = \boldsymbol{\delta}_o$$

$$\boldsymbol{\delta}_o = \mathbf{o} \odot (\mathbf{1} - \mathbf{o}) \odot (\mathbf{o} - \mathbf{y})$$

Hadamard product \odot

Backpropagation Phase

$$\mathcal{E}_{\mathbf{x}} = \frac{1}{2} \|\mathbf{y} - \mathbf{o}\|^2 = \frac{1}{2} \sum_{j=1}^p (y_j - o_j)^2$$

$$w_{ij} = w_{ij} - \eta \cdot \nabla_{w_{ij}} \quad b_j = b_j - \eta \cdot \nabla_{b_j}$$

Updating Parameters Between Hidden and Output Layer

$$\nabla_{w_{ij}} = \frac{\partial \mathcal{E}_{\mathbf{x}}}{\partial w_{ij}} = \frac{\partial \mathcal{E}_{\mathbf{x}}}{\partial \text{net}_j} \cdot \frac{\partial \text{net}_j}{\partial w_{ij}} = \delta_j \cdot z_i$$

$$\nabla_{b_j} = \frac{\partial \mathcal{E}_{\mathbf{x}}}{\partial b_j} = \frac{\partial \mathcal{E}_{\mathbf{x}}}{\partial \text{net}_j} \cdot \frac{\partial \text{net}_j}{\partial b_j} = \delta_j$$

$$\delta_j = \frac{\partial \mathcal{E}_{\mathbf{x}}}{\partial \text{net}_j} = \frac{\partial \mathcal{E}_{\mathbf{x}}}{\partial f(\text{net}_j)} \cdot \frac{\partial f(\text{net}_j)}{\partial \text{net}_j}$$

$$\frac{\partial \mathcal{E}_{\mathbf{x}}}{\partial f(\text{net}_j)} = \frac{\partial \mathcal{E}_{\mathbf{x}}}{\partial o_j} = \frac{\partial}{\partial o_j} \left\{ \frac{1}{2} \sum_{k=1}^p (y_k - o_k)^2 \right\} = (o_j - y_j)$$

$$\frac{\partial f(\text{net}_j)}{\partial \text{net}_j} = o_j \cdot (1 - o_j) \quad \delta_j = (o_j - y_j) \cdot o_j \cdot (1 - o_j)$$

$$\begin{aligned}\delta_j &= \frac{\partial \mathcal{E}_{\mathbf{x}}}{\partial net_j} = \sum_{k=1}^p \frac{\partial \mathcal{E}_{\mathbf{x}}}{\partial net_k} \cdot \frac{\partial net_k}{\partial z_j} \cdot \frac{\partial z_j}{\partial net_j} = \frac{\partial z_j}{\partial net_j} \cdot \sum_{k=1}^p \frac{\partial \mathcal{E}_{\mathbf{x}}}{\partial net_k} \cdot \frac{\partial net_k}{\partial z_j} \\ &= z_j \cdot (1 - z_j) \cdot \sum_{k=1}^p \delta_k \cdot w_{jk} \quad \frac{\partial z_j}{\partial net_j} = z_j \cdot (1 - z_j)\end{aligned}$$

$$\delta_h = \mathbf{z} \odot (\mathbf{1} - \mathbf{z}) \odot (\mathbf{W}_o \cdot \delta_o)$$

$$\mathbf{W}_o \cdot \delta_o = \left(\sum_{k=1}^p \delta_k \cdot w_{1k}, \sum_{k=1}^p \delta_k \cdot w_{2k}, \dots, \sum_{k=1}^p \delta_k \cdot w_{mk} \right)^T$$

$$\nabla_{\mathbf{W}_h} = \begin{pmatrix} \nabla_{w_{11}} & \dots & \nabla_{w_{1m}} \\ \nabla_{w_{21}} & \dots & \nabla_{w_{2m}} \\ \vdots & \dots & \vdots \\ \nabla_{w_{d1}} & \dots & \nabla_{w_{dm}} \end{pmatrix} = \mathbf{x} \cdot \delta_h^T$$

$$\nabla_{\mathbf{b}_h} = (\nabla_{b_1}, \nabla_{b_2}, \dots, \nabla_{b_m})^T = \delta_h$$

$$\begin{aligned}\mathbf{W}_h &= \mathbf{W}_h - \eta \cdot \nabla_{\mathbf{W}_h} \\ \mathbf{b}_h &= \mathbf{b}_h - \eta \cdot \nabla_{\mathbf{b}_h}\end{aligned}$$

Updating Parameters Between Input and Hidden Layer

$$\nabla_{w_{ij}} = \frac{\partial \mathcal{E}_{\mathbf{x}}}{\partial w_{ij}} = \frac{\partial \mathcal{E}_{\mathbf{x}}}{\partial net_j} \cdot \frac{\partial net_j}{\partial w_{ij}} = \delta_j \cdot x_i$$

$$\nabla_{b_j} = \frac{\partial \mathcal{E}_{\mathbf{x}}}{\partial b_j} = \frac{\partial \mathcal{E}_{\mathbf{x}}}{\partial net_j} \cdot \frac{\partial net_j}{\partial b_j} = \delta_j$$

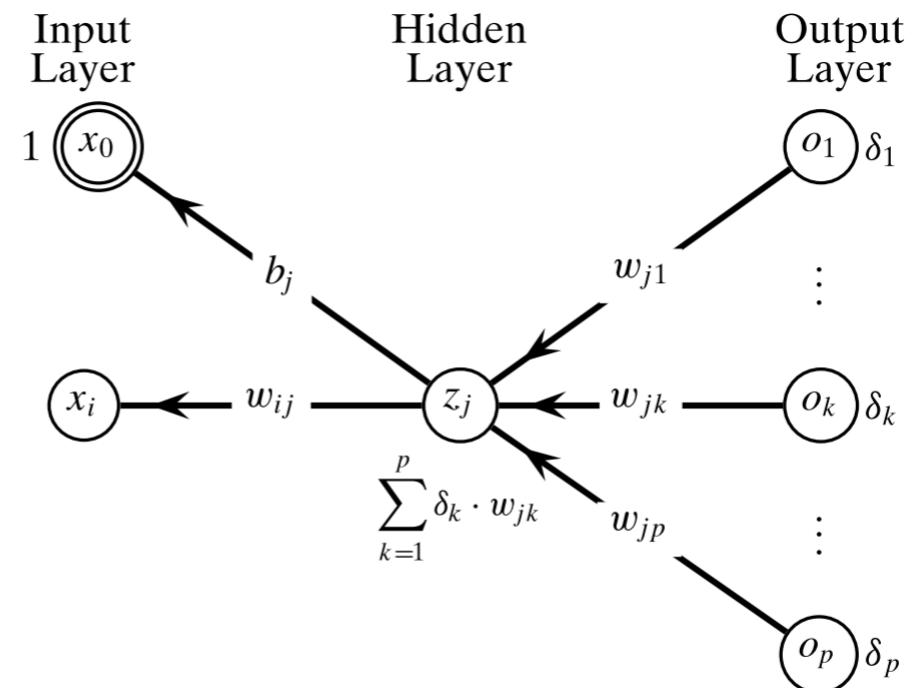


Figure 25.7. Backpropagation of gradients from output to hidden layer.

از نظر پیچیدگی محاسباتی، هر تکرار از الگوریتم آموزشی MLP برای فاز پیش‌رو به اندازه‌ی $O(dm+mp)$ و برای فاز انتشار پسرو خطای $O(dm+mp)$ (Feed-forward) به اندازه‌ی $O(p+mp+m)=O(mp)$ (Error Backpropagation) بروزرسانی ماتریس‌های وزن و بردار اریبی به اندازه‌ی $O(dm+mp)$ زمان صرف می‌شود. پس در کل برای هر تکرار $O(dm+mp)$ زمان صرف می‌شود.

فرآیند یادگیری MLP با دادن مقادیر اولیه‌ی اندک (بطور مثال عددی در بازه‌ی $[-0.01, 0.01]$) به ماتریس‌های ضرایب ورودی به لایه‌ی پنهان (\mathbf{W}_h) و لایه‌ی پنهان به لایه‌ی خروجی (\mathbf{W}_o) و بردارهای اریبی در لایه‌ی ورودی (\mathbf{b}_o) و پنهان (\mathbf{b}_h) شروع می‌شود. باید دقت کرد این مقادیر نباید صفر اختیار شوند.

MLP Training

MLP-TRAINING ($\mathbf{D}, m, \eta, \text{maxiter}$):

```

// Initialize bias vectors
1  $\mathbf{b}_h \leftarrow$  random  $m$ -dimensional vector with small values
2  $\mathbf{b}_o \leftarrow$  random  $p$ -dimensional vector with small values
// Initialize weight matrices
3  $\mathbf{W}_h \leftarrow$  random  $d \times m$  matrix with small values
4  $\mathbf{W}_o \leftarrow$  random  $m \times p$  matrix with small values
5  $t \leftarrow 0$  // iteration counter
6 repeat
7   foreach  $(\mathbf{x}_i, \mathbf{y}_i) \in \mathbf{D}$  in random order do
        // Feed-forward phase
8      $\mathbf{z}_i \leftarrow f(\mathbf{b}_h + \mathbf{W}_h^T \mathbf{x}_i)$ 
9      $\mathbf{o}_i \leftarrow f(\mathbf{b}_o + \mathbf{W}_o^T \mathbf{z}_i)$ 
        // Backpropagation phase: net gradients
10     $\delta_o \leftarrow \mathbf{o}_i \odot (\mathbf{1} - \mathbf{o}_i) \odot (\mathbf{o}_i - \mathbf{y}_i)$ 
11     $\delta_h \leftarrow \mathbf{z}_i \odot (\mathbf{1} - \mathbf{z}_i) \odot (\mathbf{W}_o \cdot \delta_o)$ 
        // Gradient descent for bias vectors
12     $\nabla_{\mathbf{b}_o} \leftarrow \delta_o; \quad \mathbf{b}_o \leftarrow \mathbf{b}_o - \eta \cdot \nabla_{\mathbf{b}_o}$ 
13     $\nabla_{\mathbf{b}_h} \leftarrow \delta_h; \quad \mathbf{b}_h \leftarrow \mathbf{b}_h - \eta \cdot \nabla_{\mathbf{b}_h}$ 
        // Gradient descent for weight matrices
14     $\nabla_{\mathbf{W}_o} \leftarrow \mathbf{z}_i \cdot \delta_o^T; \quad \mathbf{W}_o \leftarrow \mathbf{W}_o - \eta \cdot \nabla_{\mathbf{W}_o}$ 
15     $\nabla_{\mathbf{W}_h} \leftarrow \mathbf{x}_i \cdot \delta_h^T; \quad \mathbf{W}_h \leftarrow \mathbf{W}_h - \eta \cdot \nabla_{\mathbf{W}_h}$ 
16     $t \leftarrow t + 1$ 
17 until  $t \geq \text{maxiter}$ 
```

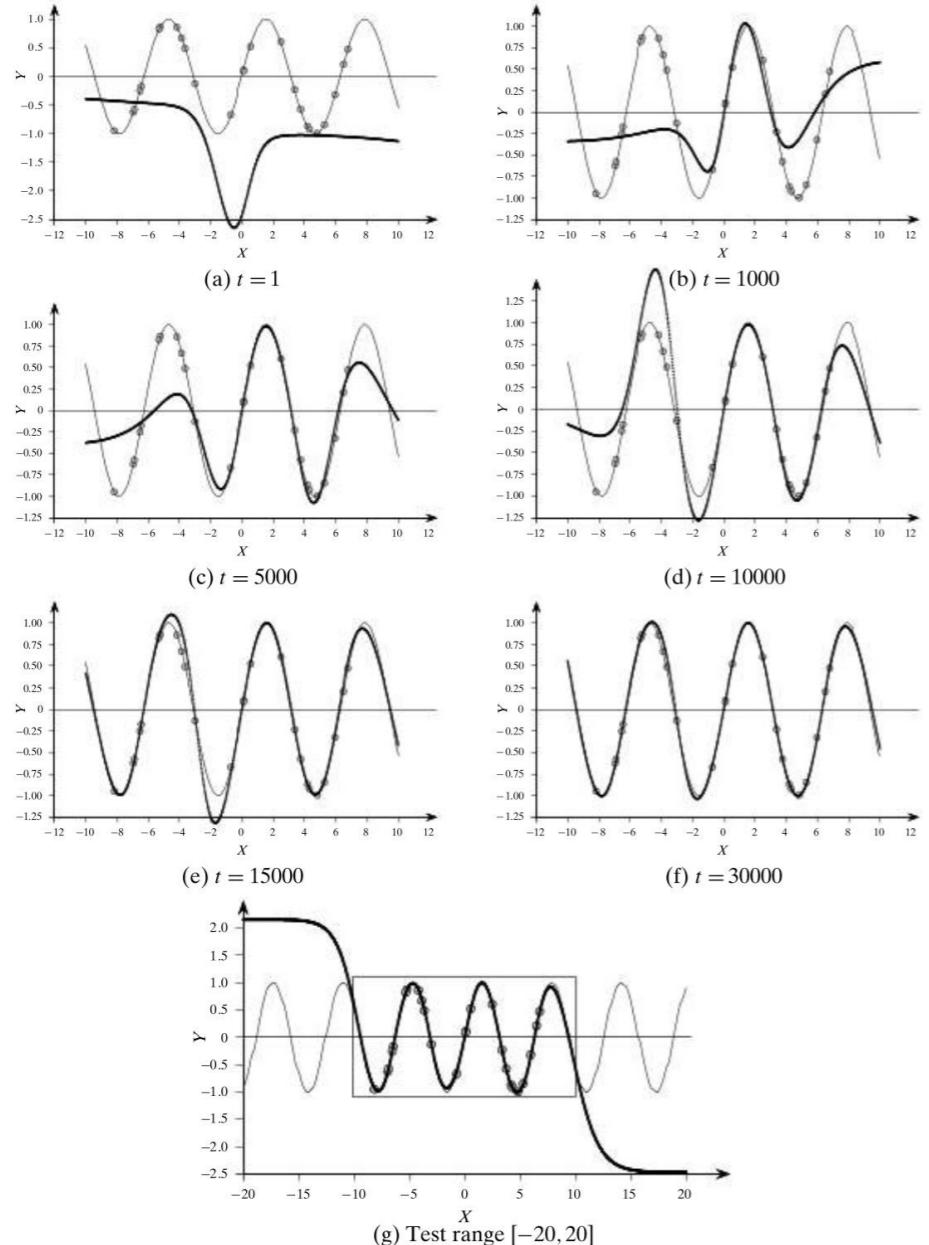


Figure 25.8. MLP for sine curve: 10 hidden neurons with hyperbolic tangent activation functions. The gray dots represent the training data. The bold line is the predicted response, whereas the thin line is the true response. (a)–(f): Predictions after different number of iterations. (g): Testing outside the training range. Good fit within the training range $[-10, 10]$ shown in the box.

Example 25.3 (MLP with one hidden layer). We now illustrate an MLP with a hidden layer using a non-linear activation function to learn the sine curve. Figure 25.8(a) shows the training data (the gray points on the curve), which comprises $n = 25$ points x_i sampled randomly in the range $[-10, 10]$, with $y_i = \sin(x_i)$. The testing data comprises 1000 points sampled uniformly from the same range. The figure also shows the desired output curve (thin line). We used an MLP with one input neuron ($d = 1$), ten hidden neurons ($m = 10$) and one output neuron ($p = 1$). The hidden neurons use tanh activations, whereas the output unit uses an identity activation. The step size is $\eta = 0.005$.

The input to hidden weight matrix $\mathbf{W}_h \in \mathbb{R}^{1 \times 10}$ and the corresponding bias vector $\mathbf{b}_h \in \mathbb{R}^{10 \times 1}$ are given as:

$$\mathbf{W}_h = (-0.68, 0.77, -0.42, -0.72, -0.93, -0.42, -0.66, -0.70, -0.62, -0.50)$$

$$\mathbf{b}_h = (-4.36, 2.43, -0.52, 2.35, -1.64, 3.98, 0.31, 4.45, 1.03, -4.77)^T$$

The hidden to output weight matrix $\mathbf{W}_o \in \mathbb{R}^{10 \times 1}$ and the bias term $\mathbf{b}_o \in \mathbb{R}$ are given as:

$$\mathbf{W}_o = (-1.82, -1.69, -0.82, 1.37, 0.14, 2.37, -1.64, -1.92, 0.78, 2.17)^T$$

$$\mathbf{b}_o = -0.16$$

Figure 25.8(a) shows the output of the MLP on the test set, after the first iteration of training ($t = 1$). We can see that initially the predicted response deviates significantly from the true sine response. Figure 25.8(a)–(f) show the output from the MLP after different number of training iterations. By $t = 15000$ iterations the output on the test set comes close to the sine curve, but it takes another 15000 iterations to get a closer fit. The final SSE is 1.45 over the 1000 test points.

We can observe that, even with a very small training data of 25 points sampled randomly from the sine curve, the MLP is able to learn the desired function. However, it is also important to recognize that the MLP model has not really learned the sine function; rather, it has learned to approximate it only in the specified range $[-10, 10]$. We can see in Figure 25.8(g) that when we try to predict values outside this range, the MLP does not yield a good fit.

پرسپترون چندلایه: یک لایه‌ی پنهان (Multilayer Perceptron: One Hidden Layer)

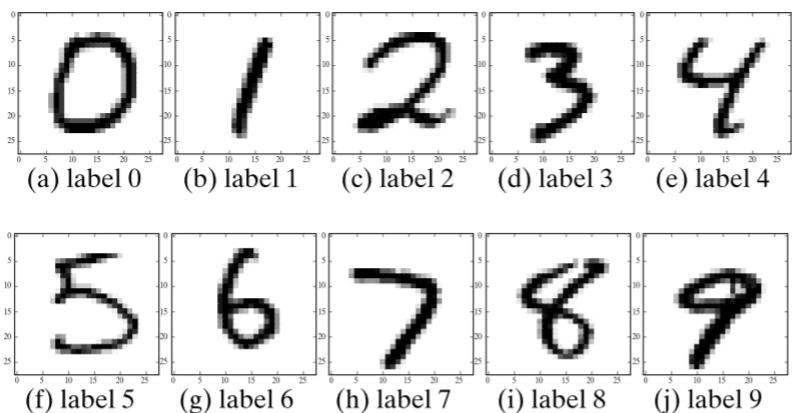


Figure 25.9. MNIST dataset: Sample handwritten digits.

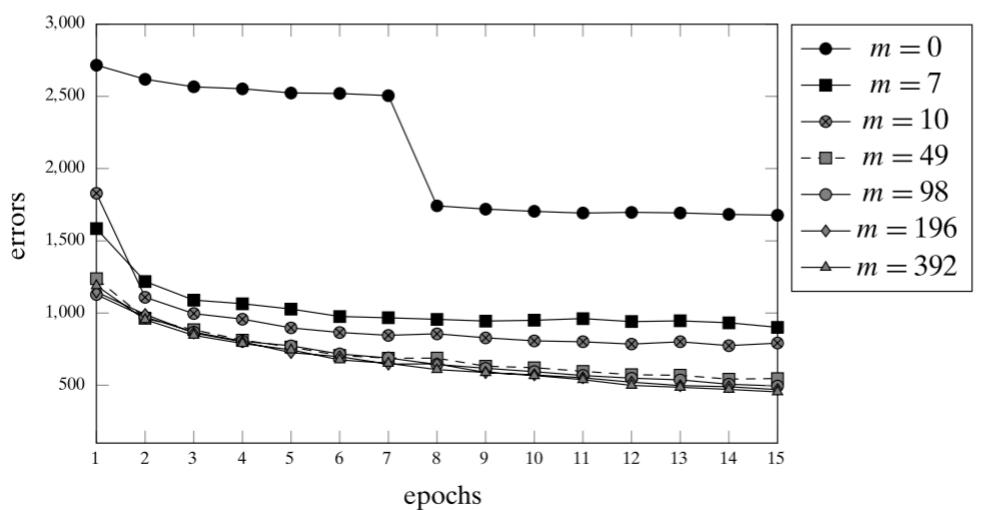


Figure 25.10. MNIST: Prediction error as a function of epochs.

Example 25.4 (MLP for handwritten digit classification). In this example, we apply an MLP with one hidden layer for the task of predicting the correct label for a hand-written digit from the MNIST database, which contains 60,000 training images that span the 10 digit labels, from 0 to 9. Each (grayscale) image is a 28×28 matrix of pixels, with values between 0 and 255. Each pixel is converted to a value in the interval $[0, 1]$ by dividing by 255. Figure 25.9 shows an example of each digit from the MNIST dataset.

Since images are 2-dimensional matrices, we first *flatten* them into a vector $\mathbf{x} \in \mathbb{R}^{784}$ with dimensionality $d = 28 \times 28 = 784$. This is done by simply concatenating all of the rows of the images to obtain one long vector. Next, since the output labels are categorical values that denote the digits from 0 to 9, we need to convert them into binary (numerical) vectors, using *one-hot* encoding. Thus, the label 0 is encoded as $\mathbf{e}_1 = (1, 0, 0, 0, 0, 0, 0, 0)^T \in \mathbb{R}^{10}$, the label 1 as $\mathbf{e}_2 = (0, 1, 0, 0, 0, 0, 0, 0)^T \in \mathbb{R}^{10}$, and so on, and finally the label 9 is encoded as $\mathbf{e}_{10} = (0, 0, 0, 0, 0, 0, 0, 0, 0, 1)^T \in \mathbb{R}^{10}$. That is, each input image vector \mathbf{x} has a corresponding target response vector $\mathbf{y} \in \{\mathbf{e}_1, \mathbf{e}_2, \dots, \mathbf{e}_{10}\}$. Thus, the input layer for the MLP has $d = 784$ neurons, and the output layer has $p = 10$ neurons.

For the hidden layer, we consider several MLP models, each with a different number of hidden neurons m . We try $m = 0, 7, 49, 98, 196, 392$, to study the effect of increasing the number of hidden neurons, from small to large. For the hidden layer, we use ReLU activation function, and for the output layer, we use softmax activation, since the target response vector has only one neuron with value 1, with the rest being 0. Note that $m = 0$ means that there is no hidden layer – the input layer is directly connected to the output layer, which is equivalent to a multiclass logistic regression model. We train each MLP for $t = 15$ epochs, using step size $\eta = 0.25$.

During training, we plot the number of misclassified images after each epoch, on the separate MNIST test set comprising 10,000 images. Figure 25.10 shows the number of errors from each of the models (with a different number of hidden neurons m), after each epoch. The final test error at the end of training is given as

m	0	7	10	49	98	196	392
errors	1677	901	792	546	495	470	454

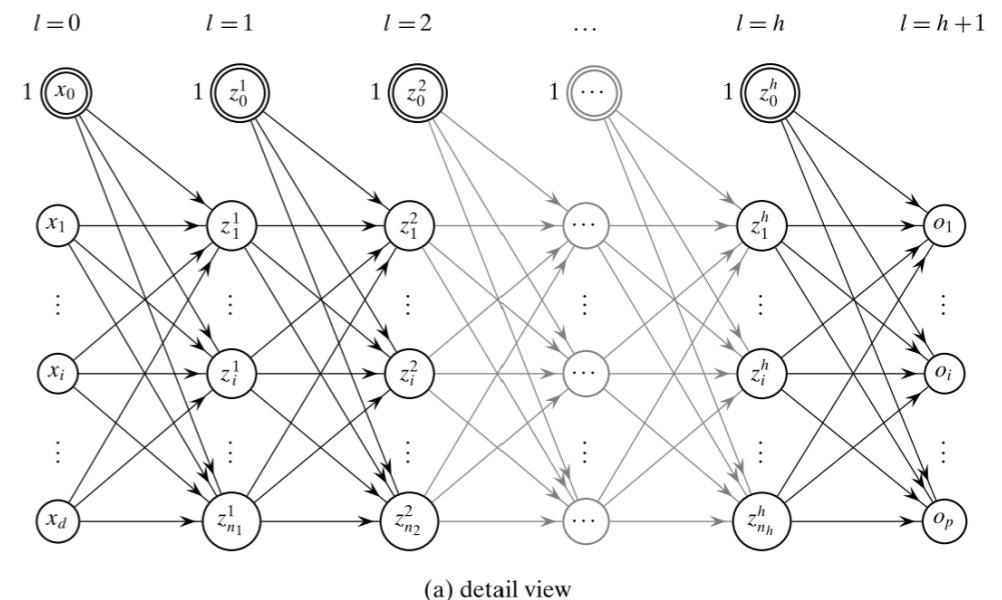
We can observe that adding a hidden layer significantly improves the prediction accuracy. Using even a small number of hidden neurons helps, compared to the logistic regression model ($m = 0$). For example, using $m = 7$ results in 901 errors (or error rate 9.01%) compared to using $m = 0$, which results in 1677 errors (or error rate 16.77%). On the other hand, as we increase the number of hidden neurons, the error rate decreases, though with diminishing returns. Using $m = 196$, the error rate is 4.70%, but even after doubling the number of hidden neurons ($m = 392$), the error rate goes down to only 4.54%. Further increasing m does not reduce the error rate.

فاز تغذیه پیش رو (Feed-forward Phase)

$$\begin{aligned}\mathbf{o} &= f^{h+1}(\mathbf{b}_h + \mathbf{W}_h^T \cdot \mathbf{z}^h) \\ &= f^{h+1}(\mathbf{b}_h + \mathbf{W}_h^T \cdot f^h(\mathbf{b}_{h-1} + \mathbf{W}_{h-1}^T \cdot \mathbf{z}^{h-1})) \\ &= \vdots \\ &= f^{h+1}(\mathbf{b}_h + \mathbf{W}_h^T \cdot f^h(\mathbf{b}_{h-1} + \mathbf{W}_{h-1}^T \cdot f^{h-1}(\cdots f^2(\mathbf{b}_1 + \mathbf{W}_1^T \cdot f^1(\mathbf{b}_0 + \mathbf{W}_0^T \cdot \mathbf{x}))))))\end{aligned}$$

فاز انتشار پس رو (Backpropagation Phase)

$$\begin{aligned}w_{ij}^l &= w_{ij}^l - \eta \cdot \nabla_{w_{ij}^l} \quad b_j^l = b_j^l - \eta \cdot \nabla_{b_j^l} \\ \nabla_{w_{ij}^l} &= \frac{\partial \mathcal{E}_{\mathbf{x}}}{\partial w_{ij}^l} = \frac{\partial \mathcal{E}_{\mathbf{x}}}{\partial \text{net}_j} \cdot \frac{\partial \text{net}_j}{\partial w_{ij}^l} = \delta_j^{l+1} \cdot z_i^l = z_i^l \cdot \delta_j^{l+1} \\ \frac{\partial \text{net}_j}{\partial w_{ij}^l} &= \frac{\partial}{\partial w_{ij}^l} \left\{ b_j^l + \sum_{k=0}^{n_l} w_{kj}^l \cdot z_k^l \right\} = z_i^l \\ \nabla_{b_j^l} &= \frac{\partial \mathcal{E}_{\mathbf{x}}}{\partial b_j^l} = \frac{\partial \mathcal{E}_{\mathbf{x}}}{\partial \text{net}_j} \cdot \frac{\partial \text{net}_j}{\partial b_j^l} = \delta_j^{l+1} \\ \frac{\partial \text{net}_j}{\partial b_j^l} &= \frac{\partial}{\partial b_j^l} \left\{ b_j^l + \sum_{k=0}^{n_l} w_{kj}^l \cdot z_k^l \right\} = 1\end{aligned}$$



(a) detail view

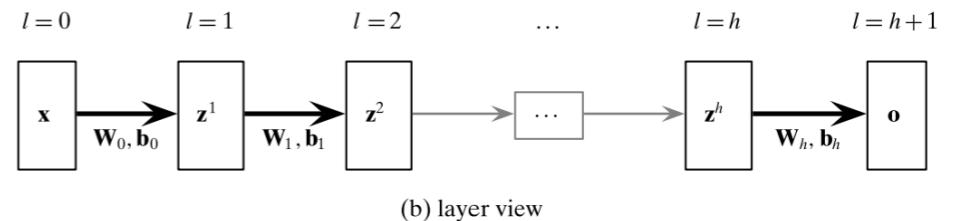


Figure 25.11. Deep multilayer perceptron, with h hidden layers.

تابع فعال ساز f_i^l برای نورون z^l در لایه l استفاده می شود.

$$\begin{aligned}\mathbf{z}^l &= f^l(\mathbf{net}_l) = f^l(\mathbf{b}_{l-1} + \mathbf{W}_{l-1}^T \cdot \mathbf{z}^{l-1}) \\ z_i^l &= f^l(\text{net}_i^l) = f^l\left(b_i^{l-1} + \sum_{j=1}^{n_{l-1}} W_{j,i}^{l-1} z_j^{l-1}\right), \quad i = 1, \dots, n_l \\ \mathbf{z}^0 &= \mathbf{x}, \quad \mathbf{z}^{h+1} = \mathbf{o} \quad \mathbf{b}_l \in \mathbb{R}^{n_l+1} \quad \mathbf{W}_l \in \mathbb{R}^{n_{l-1} \times n_l+1}\end{aligned}$$

خطای مربعی (Squared Error)

$$\partial \mathcal{E}_x = \frac{\partial \mathcal{E}_x}{\partial \mathbf{o}} = \mathbf{o} - \mathbf{y}$$

و اگر تابع فعالساز در نرون خروجی خطی باشد، جمله‌ی گرادیان آن ساده‌ی شود.

$$\partial \mathbf{f}^{h+1} = \mathbf{1}$$

خطای آنتروپی متقاطع (Cross-Entropy) با خروجی دوتایی (Binary Output)

$$\mathcal{E}_x = -(y \cdot \ln(o) + (1 - y) \cdot \ln(1 - o))$$

$$\partial \mathcal{E}_x = \frac{\partial \mathcal{E}_x}{\partial o} = \frac{o - y}{o \cdot (1 - o)}$$

و اگر تابع فعالساز سیگموید (Sigmoid) باشد.

$$\partial \mathbf{f}^{h+1} = \frac{\partial f(\text{net}_o)}{\partial \text{net}_o} = o \cdot (1 - o)$$

$$\delta^{h+1} = \partial \mathcal{E}_x \cdot \partial \mathbf{f}^{h+1} = \frac{o - y}{o \cdot (1 - o)} \cdot o \cdot (1 - o) = o - y$$

گرادیان کل در لایه‌ی خروجی (Net Gradients at Output Layer)

اگر تمام نرون‌های خروجی از هم مستقل باشند بطور مثال هنگامی که تابه فعالساز تابع خطی (Linear) و یا تابع سیگموید (Sigmoid) باشد.

$$\delta_j^{h+1} = \frac{\partial \mathcal{E}_x}{\partial \text{net}_j} = \frac{\partial \mathcal{E}_x}{\partial f^{h+1}(\text{net}_j)} \cdot \frac{\partial f^{h+1}(\text{net}_j)}{\partial \text{net}_j} = \frac{\partial \mathcal{E}_x}{\partial o_j} \cdot \frac{\partial f^{h+1}(\text{net}_j)}{\partial \text{net}_j}$$

$$\delta^{h+1} = \partial \mathbf{f}^{h+1} \odot \partial \mathcal{E}_x$$

و اما اگر نرون‌های خروجی از هم مستقل نباشند مانند وقتی که از تابع فعالساز سافتمنکس (Softmax) استفاده می‌کنیم.

$$\delta_j^{h+1} = \frac{\partial \mathcal{E}_x}{\partial \text{net}_j} = \sum_{i=1}^p \frac{\partial \mathcal{E}_x}{\partial f^{h+1}(\text{net}_i)} \cdot \frac{\partial f^{h+1}(\text{net}_i)}{\partial \text{net}_j}$$

$$\delta^{h+1} = \partial \mathbf{F}^{h+1} \cdot \partial \mathcal{E}_x$$

$$\partial \mathbf{F}^{h+1} = \begin{pmatrix} \frac{\partial o_1}{\partial \text{net}_1} & \frac{\partial o_1}{\partial \text{net}_2} & \cdots & \frac{\partial o_1}{\partial \text{net}_p} \\ \frac{\partial o_2}{\partial \text{net}_1} & \frac{\partial o_2}{\partial \text{net}_2} & \cdots & \frac{\partial o_2}{\partial \text{net}_p} \\ \vdots & \vdots & \cdots & \vdots \\ \frac{\partial o_p}{\partial \text{net}_1} & \frac{\partial o_p}{\partial \text{net}_2} & \cdots & \frac{\partial o_p}{\partial \text{net}_p} \end{pmatrix}$$

$$\delta^{h+1} = \partial \mathbf{F}^{h+1} \cdot \partial \mathcal{E}_{\mathbf{x}}$$

$$= \begin{pmatrix} o_1 \cdot (1 - o_1) & -o_1 \cdot o_2 & \cdots & -o_1 \cdot o_K \\ -o_1 \cdot o_2 & o_2 \cdot (1 - o_2) & \cdots & -o_2 \cdot o_K \\ \vdots & \vdots & \cdots & \vdots \\ -o_1 \cdot o_K & -o_2 \cdot o_K & \cdots & o_K \cdot (1 - o_K) \end{pmatrix} \cdot \begin{pmatrix} -\frac{y_1}{o_1} \\ -\frac{y_2}{o_2} \\ \vdots \\ -\frac{y_K}{o_K} \end{pmatrix}$$

$$= \begin{pmatrix} -y_1 + y_1 \cdot o_1 + \sum_{i \neq 1}^K y_i \cdot o_1 \\ -y_2 + y_2 \cdot o_2 + \sum_{i \neq 2}^K y_i \cdot o_2 \\ \vdots \\ -y_K + y_K \cdot o_K + \sum_{i \neq K}^K y_i \cdot o_K \end{pmatrix} = \begin{pmatrix} -y_1 + o_1 \cdot \sum_{i=1}^K y_i \\ -y_2 + o_2 \cdot \sum_{i=1}^K y_i \\ \vdots \\ -y_K + o_K \cdot \sum_{i=1}^K y_i \end{pmatrix}$$

$$= \begin{pmatrix} -y_1 + o_1 \\ -y_2 + o_2 \\ \vdots \\ -y_K + o_K \end{pmatrix}, \text{ since } \sum_{i=1}^K y_i = 1$$

$$= \mathbf{o} - \mathbf{y}$$

خطای آنتروپی متقاطع (K Output) با خروجی چندتایی (Cross-Entropy)

$$\mathcal{E}_{\mathbf{x}} = - \sum_{i=1}^K y_i \cdot \ln(o_i) = - \left(y_1 \cdot \ln(o_1) + \cdots + y_K \cdot \ln(o_K) \right)$$

$$\partial \mathcal{E}_{\mathbf{x}} = \left(\frac{\partial \mathcal{E}_{\mathbf{x}}}{\partial o_1}, \frac{\partial \mathcal{E}_{\mathbf{x}}}{\partial o_2}, \dots, \frac{\partial \mathcal{E}_{\mathbf{x}}}{\partial o_K} \right)^T = \left(-\frac{y_1}{o_1}, -\frac{y_2}{o_2}, \dots, -\frac{y_K}{o_K} \right)^T$$

معمولا همراه با جمله‌ی خطای آنتروپی متقاطع با خروجی چندتایی از تابع فعالساز سافتمنکس (Softmax) استفاده می‌شود.

$$o_j = \text{softmax}(net_j) = \frac{\exp\{net_j\}}{\sum_{i=1}^K \exp\{net_i\}} \quad \sum_{j=1}^K o_j = 1$$

$$\partial \mathbf{F}^{h+1} = \begin{pmatrix} \frac{\partial o_1}{\partial net_1} & \frac{\partial o_1}{\partial net_2} & \cdots & \frac{\partial o_1}{\partial net_K} \\ \frac{\partial o_2}{\partial net_1} & \frac{\partial o_2}{\partial net_2} & \cdots & \frac{\partial o_2}{\partial net_K} \\ \vdots & \vdots & \cdots & \vdots \\ \frac{\partial o_K}{\partial net_1} & \frac{\partial o_K}{\partial net_2} & \cdots & \frac{\partial o_K}{\partial net_K} \end{pmatrix}$$

$$= \begin{pmatrix} o_1 \cdot (1 - o_1) & -o_1 \cdot o_2 & \cdots & -o_1 \cdot o_K \\ -o_1 \cdot o_2 & o_2 \cdot (1 - o_2) & \cdots & -o_2 \cdot o_K \\ \vdots & \vdots & \cdots & \vdots \\ -o_1 \cdot o_K & -o_2 \cdot o_K & \cdots & o_K \cdot (1 - o_K) \end{pmatrix}$$

آموزش پرسپترون‌های چندلایه‌ی عمیق (Training Deep MLPs)

Algorithm 25.2: Deep MLP Training: Stochastic Gradient Descent

```

DEEP-MLP-TRAINING ( $\mathbf{D}, h, \eta, \text{maxiter}, n_1, n_2, \dots, n_h, f^1, f^2, \dots, f^{h+1}$ ):
```

- 1 $n_0 \leftarrow d$ // input layer size
- 2 $n_{h+1} \leftarrow p$ // output layer size
- // Initialize weight matrices and bias vectors
- 3 **for** $l = 0, 1, 2, \dots, h$ **do**
- 4 $\mathbf{b}_l \leftarrow$ random n_{l+1} vector with small values
- 5 $\mathbf{W}_l \leftarrow$ random $n_l \times n_{l+1}$ matrix with small values
- 6 $t \leftarrow 0$ // iteration counter
- 7 **repeat**
- 8 **foreach** $(\mathbf{x}_i, \mathbf{y}_i) \in \mathbf{D}$ in random order **do**
- 9 // Feed-Forward Phase
- 10 $\mathbf{z}^0 \leftarrow \mathbf{x}_i$
- 11 **for** $l = 0, 1, 2, \dots, h$ **do**
- 12 $\mathbf{z}^{l+1} \leftarrow f^{l+1}(\mathbf{b}_l + \mathbf{W}_l^T \cdot \mathbf{z}^l)$
- 13 $\mathbf{o}_i \leftarrow \mathbf{z}^{h+1}$
- 14 // Backpropagation Phase
- 15 **if** independent outputs **then**
- 16 $\delta^{h+1} \leftarrow \partial \mathbf{f}^{h+1} \odot (\mathbf{y}_i - \mathbf{o}_i)$ // net gradients at output
- 17 **else**
- 18 $\delta^{h+1} \leftarrow \partial \mathbf{F}^{h+1} \cdot \partial \mathbf{E}_{\mathbf{x}_i}$ // net gradients at output
- 19 **for** $l = h, h-1, \dots, 1$ **do**
- 20 $\delta^l \leftarrow \partial \mathbf{f}^l \odot (\mathbf{W}_l \cdot \delta^{l+1})$ // net gradients at layer l
- 21 // Gradient Descent Step
- 22 **for** $l = 0, 1, \dots, h$ **do**
- 23 $\nabla_{\mathbf{W}_l} \leftarrow \mathbf{z}^l \cdot (\delta^{l+1})^T$ // weight gradient matrix at layer l
- 24 $\nabla_{\mathbf{b}_l} \leftarrow \delta^{l+1}$ // bias gradient vector at layer l
- 25 **for** $l = 0, 1, \dots, h$ **do**
- 26 $\mathbf{W}_l \leftarrow \mathbf{W}_l - \eta \cdot \nabla_{\mathbf{W}_l}$ // update \mathbf{W}_l
- $\mathbf{b}_l \leftarrow \mathbf{b}_l - \eta \cdot \nabla_{\mathbf{b}_l}$ // update \mathbf{b}_l
- 25 $t \leftarrow t + 1$
- 26 **until** $t \geq \text{maxiter}$

گرادیان کل در لایه‌های پنهان (Net Gradients at Hidden Layers)

$$\begin{aligned}\delta_j^l &= \frac{\partial \mathcal{E}_{\mathbf{x}}}{\partial \text{net}_j} = \sum_{k=1}^{n_{l+1}} \frac{\partial \mathcal{E}_{\mathbf{x}}}{\partial \text{net}_k} \cdot \frac{\partial \text{net}_k}{\partial f^l(\text{net}_j)} \cdot \frac{\partial f^l(\text{net}_j)}{\partial \text{net}_j} \\ &= \frac{\partial f^l(\text{net}_j)}{\partial \text{net}_j} \cdot \sum_{k=1}^{n_{l+1}} \delta_k^{l+1} \cdot w_{jk}^l\end{aligned}$$

$\delta^l = \partial \mathbf{f}^l \odot (\mathbf{W}_l \cdot \delta^{l+1})$

1
for linear

z'(1-z)
for sigmoid

(1-z ⊙ z)
for tanh

$$\delta^h = \partial \mathbf{f}^h \odot (\mathbf{W}_h \cdot \delta^{h+1})$$

در حالت کل softmax در لایه‌های پنهان کاربردی ندارد.

$$\delta^{h-1} = \partial \mathbf{f}^{h-1} \odot (\mathbf{W}_{h-1} \cdot \delta^h) = \partial \mathbf{f}^{h-1} \odot \left(\mathbf{W}_{h-1} \cdot \left(\partial \mathbf{f}^h \odot (\mathbf{W}_h \cdot \delta^{h+1}) \right) \right)$$

⋮

$$\delta^1 = \partial \mathbf{f}^1 \odot \left(\mathbf{W}_1 \cdot \left(\partial \mathbf{f}^2 \odot \left(\mathbf{W}_2 \cdot \dots \cdot \left(\partial \mathbf{f}^h \odot (\mathbf{W}_h \cdot \delta^{h+1}) \right) \right) \right) \right)$$

$$\nabla_{\mathbf{W}_l} = \mathbf{z}^l \cdot (\delta^{l+1})^T$$

$$\nabla_{\mathbf{b}_l} = \delta^{l+1}$$

$$\mathbf{W}_l = \mathbf{W}_l - \eta \cdot \nabla_{\mathbf{W}_l}$$

$$\mathbf{b}_l = \mathbf{b}_l - \eta \cdot \nabla_{\mathbf{b}_l}$$

یکی از نکاتی که هنگام آموزش MLP‌های بسیار عمیق وجود دارد، مشکل ناچیز شدن (Exploding) و انفجار (Vanishing) گرادیانها است.

در مشکل ناچیز شدن گرادیانها، مقدار گرادیان کل به شکل نمایی و تصاعدی با فاصله و دور شدن از لایه‌ی خروجی کاهش می‌یابد و ناچیز می‌شود. در نتیجه در به روزرسانی‌ها، وزن‌ها و اریب‌ها تغییرات بسیار ناچیزی دارند و فرآیند یادگیری در شبکه بسیار کند می‌شود.

از طرف دیگر در مشکل انفجار گرادیان، مقدار گرادیان کل با فاصله از لایه‌ی خروجی به شکل تصاعدی افزایش می‌یابد. که منجر به مقادیر بسیار بزرگ وزن‌ها و اریب‌ها و در نهایت توقف یادگیری می‌شود. مشکل انفجار گرادیان را می‌توان با اعمال یک کران بالا برای گرادیان‌ها حل نمود و مقادیر گرادیان‌ها را با رسیدن به یک آستانه‌ی بالایی به یک مقدار کم تنظیم مجدد نمود.

اما حل مشکل ناچیز شدن گرادیان‌ها به این سادگی نیست. عموماً استفاده از تابع فعال‌ساز sigmoid منجر به ناچیز شدن گرادیان‌ها می‌شود که در این حالت بهتر است آن را با تابع فعال‌ساز ReLU جایگزین نمود.

ابتدا ماتریس‌های وزن و بردارهای اولیه بسیار کوچک و تصادفی مثلاً در بازه‌ی [0.01, 0.01] اختیار می‌کنند.

در فاز تغذیه پیش رو با استفاده از مقادیر ورودی از ماتریس داده‌ها و ضرایب اولیه مقادیر خروجی **0** محاسبه می‌شوند.

فاز انتشار خطای پس رو با محاسبه خطای بین مقادیر محاسبه شده‌ی خروجی **0** و مقادیر پاسخ واقعی \hat{y} آغاز می‌شود. سپس گرادیان کل در لایه‌ی خروجی δ^{h+1} را محاسبه و گام به گام به گام گرادیان کل در لایه‌های پنهان را از δ^h تا δ^1 بدست می‌آوریم. ازین گرادیان‌های کل برای محاسبه گرادیان ماتریس‌های وزن و بردارهای اولیه در لایه‌های مختلف استفاده می‌شود.

پس از استفاده از هر نقطه برای به روزرسانی وزن‌ها، یک تکرار یا دوره تمرین (epoch) کامل می‌شود. آموزش زمانی که تعداد epoch‌ها به بیشینه‌ی از پیش تعیین شده (maxiter) رسید متوقف می‌شود.

در یک رویکرد کاهش گرادیان تصادفی (SGD) نقاط برای آموزش بطور تصادفی انتخاب و وزن‌ها بعد از مشاهده و انتخاب هر نقطه به روزرسانی می‌شوند.

اما در عمل بجای استفاده از نقاط منفرد، به روزرسانی گرادیان‌ها در یک زیرمجموعه با تعداد ثابت از نقاط آموزشی به نام minibatch انجام می‌شود. یعنی ابتدا داده‌های آموزش به دسته‌های کوچک minibatch‌ها بطور تصادفی تقسیم می‌شوند و سپس با اطلاعات هر دسته گرادیان‌ها محاسبه و وزن‌ها به روزرسانی می‌گردند. اندازه‌ی minibatch‌ها به عنوان یک پارامتر اضافی به مساله وارد می‌شود.

رویکرد استفاده از minibatch منجر به تخمین بهتر گرادیان‌ها، هم‌گرایی سریعتر و افزایش سرعت یادگیری می‌شود. در ضمن امکان استفاده از محاسبات ماتریسی و برداری را نیز مهیا می‌کند.

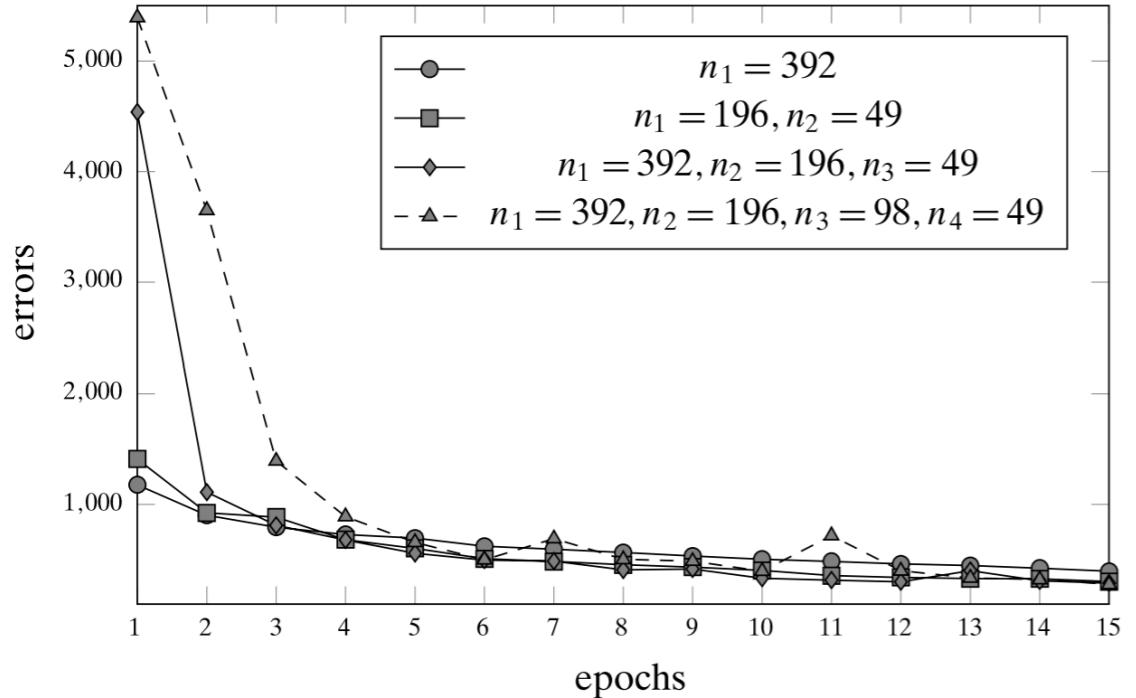


Figure 25.12. MNIST: Deep MLPs; prediction error as a function of epochs.

Example 25.5 (Deep MLP). We now examine deep MLPs for predicting the labels for the MNIST handwritten images dataset that we considered in Example 25.4. Recall that this dataset has $n = 60000$ grayscale images of size 28×28 that we treat as $d = 784$ dimensional vectors. The pixel values between 0 and 255 are converted to the range 0 and 1 by dividing each value by 255. The target response vector is a one-hot encoded vector for class labels $\{0, 1, \dots, 9\}$. Thus, the input to the MLP \mathbf{x}_i has dimensionality $d = 784$, and the output layer has dimensionality $p = 10$. We use softmax activation for the output layer. We use ReLU activation for the hidden layers, and consider several deep models with different number and sizes of the hidden layers. We use step size $\eta = 0.3$ and train for $t = 15$ epochs. Training was done using minibatches, using batch size of 1000.

During the training of each of the deep MLPs, we evaluate its performance on the separate MNIST test dataset that contains 10,000 images. Figure 25.12 plots the number of errors after each epoch for the different deep MLP models. The final test error at the end of training is given as

hidden layers	errors
$n_1 = 392$	396
$n_1 = 196, n_2 = 49$	303
$n_1 = 392, n_2 = 196, n_3 = 49$	290
$n_1 = 392, n_2 = 196, n_3 = 98, n_4 = 49$	278

We can observe that as we increase the number of layers, we do get performance improvements. The deep MLP with four hidden layers of sizes $n_1 = 392, n_2 = 196, n_3 = 98, n_4 = 49$ results in an error rate of 2.78% on the training set, whereas the MLP with a single hidden layer of size $n_1 = 392$ has an error rate of 3.96%. Thus, the deeper MLP significantly improves the prediction accuracy. However, adding more layers does not reduce the error rate, and can also lead to performance degradation.