

Machine learning problems can be broadly classified into *supervised learning*, *unsupervised learning* and *reinforcement learning*. In supervised learning, we have set of *feature vectors* and their corresponding *target values*. The aim of supervised learning is to learn a model to accurately predict targets given unseen feature vectors. In other words, the computer must learn a mapping from feature vectors to target values. The feature vectors might be called independent variable and the target values might be called dependent variable. Learning is done using an objective function which directly depends on target values. For example, classification of traffic signs is a supervised learning problem.

In unsupervised setting, we only have a set of feature vectors without any target value. The main goal of unsupervised learning is to learn structure of data. Here, because target values do not exist, there is not a specific way to evaluate learnt models. For instance, assume we have a dataset with 10,000 records in which each data is a vector consists of [driver's age, driver's gender, driver's education level, driving experience, type of car, model of car, car manufacturer, GPS point of accident, temperature, humidity, weather condition, daylight, time, day of week, type of road]. The goal might be to divide this dataset into 20 categories. Then, we can analyze categories to see how many records fall into each category and what is common among these records. Using this information, we might be able to say in which conditions car accidents happen more frequently. As we can see in this example, there is not a clear way to tell how well the records are categorized.

Reinforcement learning usually happens in dynamic environments where series of actions lead the system into a point of getting a reward or punishment. For example, consider a system that is learning to drive a car. The system starts to driver and several seconds later it hits an obstacle. Series of actions has caused the system to hit the obstacle. Notwithstanding, there is no information to tell us how good was the action which the systems performed at a specific time. Instead, the system is

punished because it hit the obstacle. Now, the system must figure out which actions were not correct and act accordingly.

2.1 Formulation

Supervised learning mainly breaks down into *classification* and *regression*. The main difference between them is the type of target values. While target values of a regression problem are real/discrete numbers, target values of a classification problem are categorical numbers which are called *labels*. To be more specific, assume $\mathcal{F}_r : \mathbb{R}^d \rightarrow \mathbb{R}$ is a regression model which returns a real number. Moreover, assume we have the pair (\mathbf{x}_r, y_r) including a d -dimensional input vector \mathbf{x}_r and real number y_r . Ideally, $\mathcal{F}_r(\mathbf{x}_r)$ must be equal to y_r . In other words, we can evaluate the accuracy of the prediction by simply computing $|\mathcal{F}_r(\mathbf{x}_r) - y_r|$.

In contrast, assume the classification model

$$\mathcal{F}_c : \mathbb{R}^d \rightarrow \{\text{speedlimit}, \text{danger}, \text{prohibitive}, \text{mandatory}\} \quad (2.1)$$

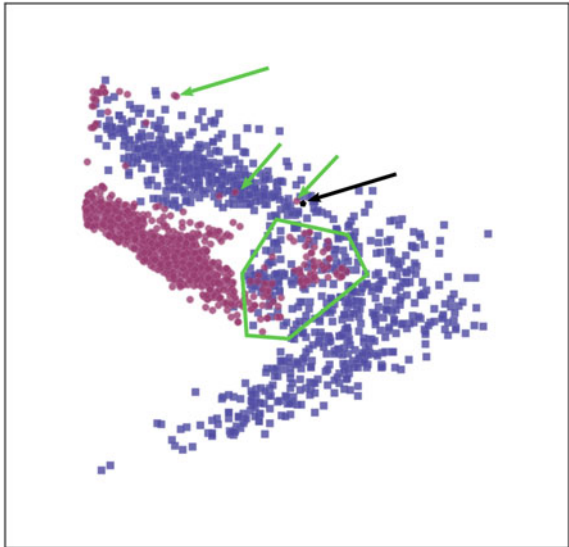
which returns a categorical number/label. Given the pair $(\mathbf{x}_c, \text{danger})$, $\mathcal{F}_c(\mathbf{x}_c)$ must be ideally equal to *danger*. However, it might return *mandatory* wrongly. It is not possible to simply subtract the output of \mathcal{F}_c with the actual label to ascertain how much the model has deviated from actual output. The reason is that there is not a specific definition of *distance* between labels. For example, we cannot tell what is the distance between “danger” and “prohibitive” or “danger” and “mandatory”. In other words, the label space is not an ordered set. Both traffic sign detection and recognition problems are formulated using a classification model. In the rest of this section, we will explain the fundamental concepts using simple examples.

Assume a set of pairs $\mathcal{X} = \{(\mathbf{x}_0, y_0), \dots, (\mathbf{x}_n, y_n)\}$ where $\mathbf{x}_i \in \mathbb{R}^2$ is a two-dimensional input vector and $y_i \in \{0, 1\}$ is its label. Despite the fact that 0 and 1 are numbers, we treat them as categorical labels. Therefore, it is not possible to compute their distance. The target value y_i in this example can only take one of the two values. These kind of classification problems in which the target value can only take two values are called *binary classification* problems. In addition, because the input vectors are two-dimensional we can easily plot them. Figure 2.1 illustrates the scatter plot of a sample \mathcal{X} .

The blue squares show the points belonging to one class and the pink circles depicts the points belonging to the other class. We observe that the two classes overlap inside the green polygon. In addition, the vectors shown by the green arrows are likely to be noisy data. More importantly, these two classes are not *linearly separable*. In other words, it is not possible to perfectly separate these two classes from each other by drawing a line on the plane.

Assume we are given a $x_q \in \mathbb{R}^2$ and we are asked to tell which class x_q belongs to. This point is shown using a black arrow on the figure. Note that we do not know the target value of x_q . To answer this question, we first need to learn a model from

Fig. 2.1 A dataset of two-dimensional vectors representing two classes of objects



\mathcal{X} which is able to discriminate the two classes. There are many ways to achieve this goal in literature. However, we are only interested in a particular technique called *linear models*. Before explaining this technique, we mention a method called *k-nearest neighbor*.

2.1.1 K-Nearest Neighbor

From one perspective, machine learning models can be categorized into *parametric* and *nonparametric* models. Roughly speaking, parametric models have some parameters which are directly learnt from data. In contrast, nonparametric models do not have any parameters to be learnt from data. K-nearest neighbor (KNN) is a nonparametric method which can be used in regression and classification problem.

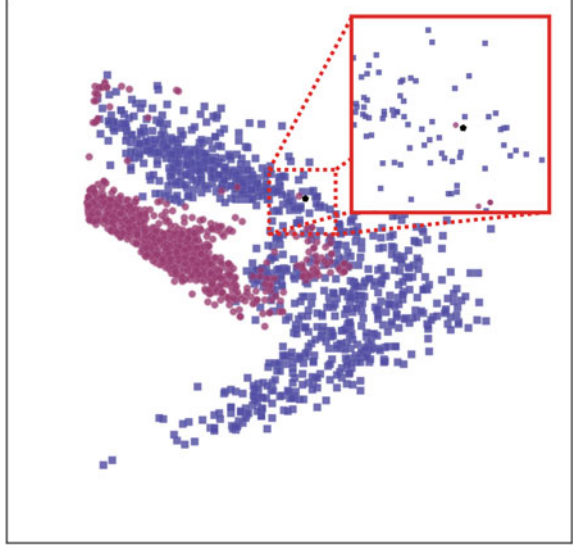
Given the training set \mathcal{X} , KNN stores all these samples in memory. Then, given the query vector \mathbf{x}_q , it finds K closest samples from \mathcal{X} to \mathbf{x}_q .¹ Denoting the K closest neighbors of \mathbf{x}_q with $N_K(\mathbf{x}_q; \mathcal{X})$,² the class of \mathbf{x}_q is determined by:

$$F(\mathbf{x}_q) = \arg \max_{v \in \{0,1\}} \sum_{p \in N_K(\mathbf{x}_q)} \delta(v, f(p)) \quad (2.2)$$

¹Implementations of the methods in this chapter are available at github.com/pcnn/.

²You can read this formula as “ N_K of \mathbf{x}_q given the dataset \mathcal{X} ”.

Fig. 2.2 K-nearest neighbor looks for the K closest points in the training set to the query point



$$\delta(a, b) = \begin{cases} 1 & a = b \\ 0 & a \neq b \end{cases} \quad (2.3)$$

where $f(p)$ returns the label of training sample $p \in \mathcal{X}$. Each of K closest neighbors vote for \mathbf{x}_q according to their label. Then, the above equation counts the votes and returns the majority of votes as the class of \mathbf{x}_q . We explain the meaning of this equation on Fig. 2.2. Assuming $K = 1$, KNN looks for the closest point to \mathbf{x}_q in the training set (shown by black polygon on the figure). According to the figure, the red circle is the closest point. Because $K = 1$, there is no further point to vote. Consequently, the algorithm classifies \mathbf{x}_q as red.

By setting $K = 2$ the algorithm searches the two closest points which in this case are one red circle and one blue square. Then, the algorithm counts the votes for each label. The votes are equal in this example. Hence, the method is not confident with its decision. For this reason, in practice, we set K to an odd number so one of the labels always has the majority of votes. If we set $K = 3$, there will be two votes for the blue class and one vote for the red class. As the result, \mathbf{x}_q will be classified as blue.

We classified every point on the plane using different values of K and \mathcal{X} . Figure 2.3 illustrates the result. The black solid line on the plots shows the border between two regions with different class labels. This border is called *decision boundary*. When $K = 1$ there is always a region around the noisy points, where they are classified as the red class. However, by setting $K = 3$ those noisy regions disappear and they become part of the correct class. As the value of K increases, the decision boundary becomes more smooth and small regions disappear.

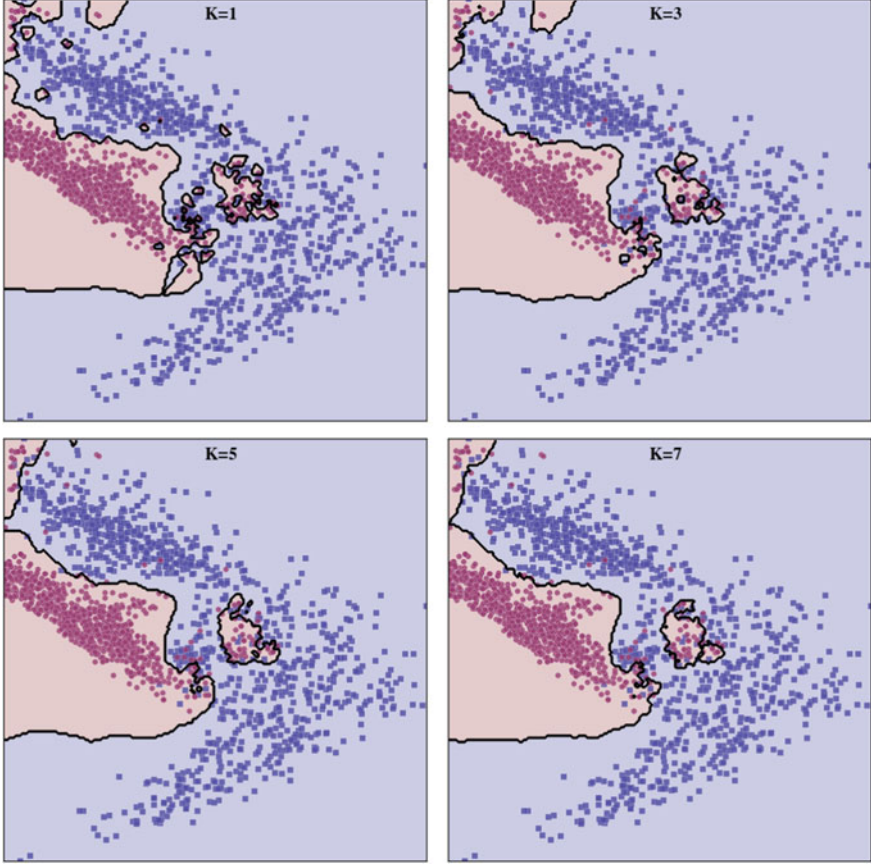


Fig. 2.3 K-nearest neighbor applied on every point on the plane for different values of K

The original KNN does not take into account the distance of its neighbor when it counts the votes. In some cases, we may want to weight the votes based on the distance from neighbors. This can be done by adding a weight term to (2.2):

$$F(\mathbf{x}_q) = \arg \max_{v \in \{0,1\}} \sum_{p \in N_K(\mathbf{x}_q)} w_i \delta(v, f(p)) \quad (2.4)$$

$$w_i = \frac{1}{d(\mathbf{x}_q, p)}. \quad (2.5)$$

In the above equation, $d(\cdot)$ returns the distance between two vectors. According to this formulation, the weight of each neighbor is equal to the inverse of its distance from \mathbf{x}_q . Therefore, closer neighbors have higher weights. KNN can be easily extended to datasets with more than two labels without any modifications. However, there

are two important issues with this method. First, finding the class of a query vector requires to separately compute the distance from all of the samples in training set. Unless we devise a solution such as partitioning the input space, this can be time and memory consuming when we are dealing with large datasets. Second, it suffers from a phenomena called *curse of dimensionality*. To put it simply, Euclidean distance becomes very similar in high-dimensional spaces. As the result, if the input of KNN is a high-dimensional vector then the difference between the closest and farthest vectors might be very similar. For this reason, it might classify the query vectors incorrectly.

To alleviate these problems, we try to find a *discriminant function* in order to directly model the decision boundary. In other words, a discriminant function models the decision boundary using training samples in \mathcal{X} . A discriminant function could be a nonlinear function. However, one of the easy ways to model decision boundaries is linear classifiers.

2.2 Linear Classifier

Assume a binary classification problem in which labels of the d -dimensional input vector $\mathbf{x} \in \mathbb{R}^d$ can be only 1 or -1 . For example, detecting traffic signs in an image can be formulated as a binary classification problem. To be more specific, given an image patch, the aim detection is to decide if the image represents a traffic sign or a non-traffic sign. In this case, images of traffic signs and non-traffic signs might be indicated using labels 1 and -1 , respectively. Denoting the i^{th} element of \mathbf{x} with x_i , it can be classified by computing the following linear relation:

$$f(\mathbf{x}) = w_1x_1 + \cdots + w_ix_i + \cdots + w_dx_d + b \quad (2.6)$$

where w_i is a trainable parameter associated with x_i and b is another trainable parameter which is called *intercept* or *bias*. The above equation represents a *hyperplane* in a d -dimensional Euclidean space. The set of weights $\{\forall_{i=1\dots d} w_i\}$ determines the orientation of the hyperplane and b indicates the distance of the hyperplane from origin. We can also write the above equation in terms of matrix multiplications:

$$f(\mathbf{x}) = \mathbf{w}\mathbf{x}^T + b \quad (2.7)$$

where $\mathbf{w} = [w_1, \dots, w_d]$. Likewise, it is possible to augment \mathbf{w} with b and show all parameters of the above equation in a single vector $\mathbf{w}_{w|b} = [b, w_1, \dots, w_d]$. With this formulation, we can also augment \mathbf{x} with 1 to obtain $\mathbf{x}_{x|1} = [1, x_1, \dots, x_d]$ and write the above equation using the following matrix multiplication:

$$f(\mathbf{x}) = \mathbf{w}_{w|b}\mathbf{x}_{x|1}^T. \quad (2.8)$$

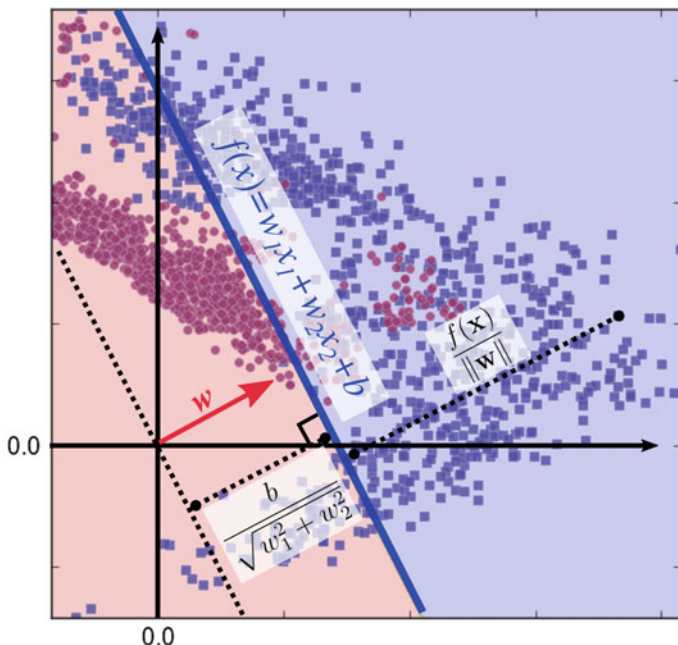


Fig. 2.4 Geometry of linear models

From now on in this chapter, when we write \mathbf{w} , \mathbf{x} we are referring to $\mathbf{w}_{w|b}$ and $\mathbf{x}_{x|1}$, respectively. Finally, \mathbf{x} is classified by applying the sign function on $f(\mathbf{x})$ as follows:

$$F(\mathbf{x}) = \begin{cases} 1 & f(\mathbf{x}) > 0 \\ NA & f(\mathbf{x}) = 0 \\ -1 & f(\mathbf{x}) < 0 \end{cases} \quad (2.9)$$

In other words, \mathbf{x} is classified as 1 if $f(\mathbf{x})$ is positive and it is classified as -1 when $f(\mathbf{x})$ is negative. The special case happens when $f(\mathbf{x}) = 0$ in which \mathbf{x} does not belong to any of these two classes. Although it may never happen in practice to have a \mathbf{x} such that $f(\mathbf{x})$ is exactly zero, it explains an important theoretical concept which is called *decision boundary*. We shall mention this topic shortly. Before, we further analyze \mathbf{w} with respect to \mathbf{x} . Clearly, $f(\mathbf{x})$ is zero when \mathbf{x} is exactly on the hyperplane. Considering the fact that \mathbf{w} and \mathbf{x} are both $d + 1$ dimensional vectors, (2.8) denotes the *dot product* of the two vectors. Moreover, we know from linear algebra that the dot product of two orthogonal vectors is 0. Consequently, the vector \mathbf{w} is orthogonal to every point on the hyperplane.

This can be studied from another perspective. This is illustrated using a two-dimensional example on Fig. 2.4. If we rewrite (2.6) in slope-intercept form, we will obtain:

$$x_2 = -\frac{w_1}{w_2}x_1 - \frac{b}{w_2}. \quad (2.10)$$

where the slope of the line is equal to $m = -\frac{w_1}{w_2}$. In addition, a line is perpendicular to the above line if its slope is equal to $m' = \frac{-1}{m} = \frac{w_2}{w_1}$. As the result, the weight vector $\mathbf{w} = [w_1, w_2]$ is perpendicular to the every point on the above line since its slope is equal to $\frac{w_2}{w_1}$. Let us have a closer look to the geometry of the linear model. The distance of point $\mathbf{x}' = [x'_1, x'_2]$ from the linear model can be found by projecting $\mathbf{x} - \mathbf{x}'$ onto \mathbf{w} which is given by:

$$r = \frac{|f(\mathbf{x})|}{\|\mathbf{w}\|} \quad (2.11)$$

Here, \mathbf{w} refers to the weight vector before augmenting with b . Also, the signed distance can be obtained by removing the abs (absolute value) operator from the numerator:

$$r_{signed} = \frac{f(\mathbf{x})}{\|\mathbf{w}\|}. \quad (2.12)$$

When \mathbf{x} is on the line (i.e., a hyperplane in N-dimensional space) then $f(\mathbf{x}) = 0$. Hence, the distance from the decision boundary will be zero. Set of all points $\{\mathbf{x} \mid \mathbf{x} \in \mathbb{R}^d \wedge f(\mathbf{x}) = 0\}$ represents the boundary between the regions with labels -1 and 1 . This boundary is called decision boundary. However, if \mathbf{x} is not on the decision boundary its distance will be a nonzero value. Also, the sign of the distance depends on the region that the point falls into. Intuitively, the model is more confident about its classification when a point is far from decision boundary. In contrary, as it gets closer to the decision boundary the confidence of the model decreases. This is the reason that we sometimes call $f(\mathbf{x})$ the *classification score* of \mathbf{x} .

2.2.1 Training a Linear Classifier

According to (2.9), output of a linear classifier could be 1 or -1 . This means that labels of the training data must be also member of set $\{-1, 1\}$. Assume we are given the training set $\mathcal{X} = \{(\mathbf{x}_0, y_0), \dots, (\mathbf{x}_n, y_n)\}$ where $\mathbf{x}_i \in \mathbb{R}^d$ is a d-dimensional vector and $y_i \in \{-1, 1\}$ showing label of the sample. In order to train a linear classifier, we need to define an *objective* function. For any \mathbf{w}_t , the objective function uses \mathcal{X} to tell how accurate is the $f(\mathbf{x}) = \mathbf{w}_t \mathbf{x}^T$ at classification of samples in \mathcal{X} . The objective function may be also called *error* function or *loss* function. Without the loss function, it is not trivial to assess the goodness of a model.

Our main goal in training a classification model is to minimize the number of samples which are classified incorrectly. We can formulate this objective using the following equation:

$$\mathcal{L}_{0/1}(\mathbf{w}) = \sum_{i=1}^n H_{0/1}(\mathbf{w} \mathbf{x}_i^T, y_i) \quad (2.13)$$

$$H_{0/1}(\mathbf{w} \mathbf{x}_i^T, y_i) = \begin{cases} 1 & \mathbf{w} \mathbf{x}_i^T \times y_i < 0 \\ 0 & \text{otherwise} \end{cases} \quad (2.14)$$

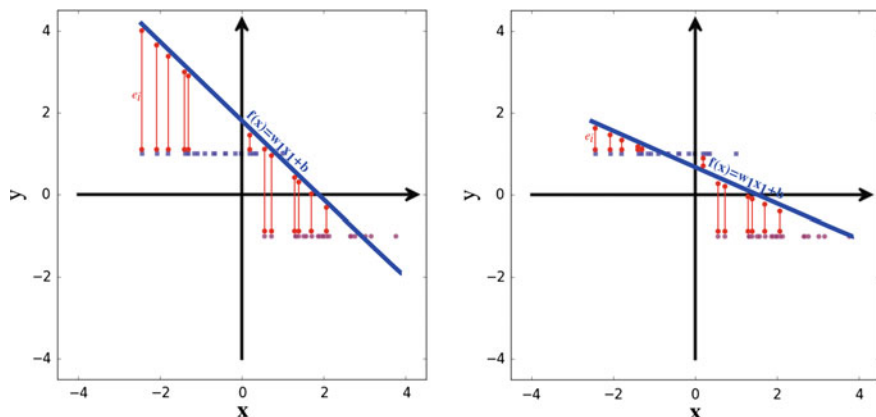


Fig.2.5 The intuition behind squared loss function is to minimize the squared difference between the actual response and predicted value. *Left* and *right* plots show two lines with different w_1 and b . The line in the *right* plot is *fitted* better than the line in the *left* plot since its prediction error is lower in total

The above loss function is called *0/1 loss* function. A sample is classified correctly when the sign of $\mathbf{w}\mathbf{x}^T$ and y_i are identical. If \mathbf{x} is not correctly classified by the model, the signs of these two terms will not be identical. This means that one of these two terms will be negative and the other one will be positive. Therefore, their multiplication will be negative. We see that $H_{0/1}(\cdot)$ returns 1 when the sample is classified incorrectly. Based on this explanation, the above loss function counts the number of misclassified samples. If all samples in \mathcal{X} is classified correctly, the above loss function will be zero. Otherwise, it will be greater than zero. There are two problems with the above loss function which makes it impractical. First, the 0/1 loss function is nonconvex. Second, it is hard to optimize this function using gradient-based optimization methods since the function is not continuous at 0 and its gradient is zero elsewhere.

Instead of counting the number of misclassified samples, we can formulate the classification problem as a regression problem and use the *squared* loss function. This can be better described using a one-dimensional input vector $\mathbf{x} \in \mathbb{R}$ in Fig. 2.5:

In this figure, circles and squares illustrate the samples with labels -1 and 1 , respectively. Since, \mathbf{x} is one-dimensional (scalar), the linear model will be $f(\mathbf{x}) = w_1x_1 + b$ with only two trainable parameters. This model can be plotted using a line in a two-dimensional space. Assume the line shown in this figure. Given any \mathbf{x} the output of the function is a real number. In the case of circles, the model should ideally return -1 . Similarly, it should return 1 for all squares in this figure. Notwithstanding, because $f(\mathbf{x})$ is a linear model $f(\mathbf{x}_1) \neq f(\mathbf{x}_2)$ if $\mathbf{x}_1 \neq \mathbf{x}_2$. This means, it is impossible that our model returns 1 for every square in this figure. In contrast, it will return a unique value for each point in this figure.

For this reason, there is an *error* between the actual output of a point (circle or square) and the predicted value by the model. These errors are illustrated using

red solid lines in this figure. The estimation error for \mathbf{x}_i can be formulated as $e_i = (f(\mathbf{x}_i) - y_i)$ where $y_i \in \{-1, 1\}$ is the actual output of \mathbf{x}_i as we defined previously in this section. Using this formulation, we can define the *squared* loss function as follows:

$$\mathcal{L}_{sq}(\mathbf{w}) = \sum_{i=1}^n \sqrt{(e_i)^2} = \sum_{i=1}^n \sqrt{(\mathbf{w}\mathbf{x}_i^T - y_i)^2}. \quad (2.15)$$

In this equation, $\mathbf{x} \in \mathbb{R}^d$ is a d -dimensional vector and $y_i \in \{-1, 1\}$ is its actual label. This loss function treat the labels as real number rather than categorical values. This makes it possible to estimate the prediction error by subtracting predicted values from actual values. Note from Fig. 2.5 that e_i can be a negative or a positive value. In order to compute the magnitude of e_i , we first compute the square of e_i and apply square root in order to compute the absolute value of e_i . It should be noted that we could define the loss function as $\sum_{i=1}^n |\mathbf{w}\mathbf{x}_i^T - y_i|$ instead of $\sum_{i=1}^n \sqrt{(\mathbf{w}\mathbf{x}_i^T - y_i)^2}$. However, as we will see shortly, the second formulation has a desirable property when we utilize a gradient-based optimization to minimize the above loss function.

We can further simplify (2.15). If we unroll the sum operator in (2.15), it will look like:

$$\mathcal{L}_{sq}(\mathbf{w}) = \sqrt{(\mathbf{w}\mathbf{x}_1^T - y_1)^2} + \dots + \sqrt{(\mathbf{w}\mathbf{x}_n^T - y_n)^2}. \quad (2.16)$$

Taking into account the fact that square root is a monotonically increasing function and it is applied on each term individually, eliminating this operator from the above equation does not change the minimum of $\mathcal{L}(\mathbf{w})$. By applying this on the above equation, we will obtain:

$$\mathcal{L}_{sq}(\mathbf{w}) = \sum_{i=1}^n (\mathbf{w}\mathbf{x}_i^T - y_i)^2. \quad (2.17)$$

Our objective is to minimize the prediction error. In other words:

$$\mathbf{w} = \min_{\mathbf{w}' \in \mathbb{R}^{d+1}} \mathcal{L}(\mathbf{w}') \quad (2.18)$$

This is achievable by minimizing \mathcal{L}_{sq} with respect to $\mathbf{w} \in \mathbb{R}^{d+1}$. In order to minimize the above loss function, we can use an iterative gradient-based optimization method such as *gradient descend* (Appendix A). Starting with an the initial vector $\mathbf{w}_{sol} \in \mathbb{R}^{d+1}$, this method iteratively changes \mathbf{w}_{sol} proportional to the gradient vector $\nabla \mathcal{L} = [\frac{\delta \mathcal{L}}{\delta w_0}, \frac{\delta \mathcal{L}}{\delta w_1}, \dots, \frac{\delta \mathcal{L}}{\delta w_d}]$. Here, we have shown the intercept using w_0 instead of b . Consequently, we need to calculate the partial derivative of the loss function with respect to each of parameters in \mathbf{w} as follows:

$$\begin{aligned} \frac{\delta \mathcal{L}}{\delta w_i} &= 2 \sum_{i=1}^n x_i (\mathbf{w}\mathbf{x}_i^T - y_i) \quad \forall i = 1 \dots d \\ \frac{\delta \mathcal{L}}{\delta w_0} &= 2 \sum_{i=1}^n (\mathbf{w}\mathbf{x}_i^T - y_i) \end{aligned} \quad (2.19)$$

One problem with the above equation is that \mathcal{L}_{sq} might be a large value if there are many training samples in \mathcal{X} . For this reason, we might need to use very small learning rate in the gradient descend method. To alleviate this problem, we can compute the *mean square error* by dividing \mathcal{L}_{sq} with the total number of training samples. In addition, we can eliminate 2 in the partial derivative by multiplying \mathcal{L}_{sq} by 1/2. The final squared loss function can be defined as follows:

$$\mathcal{L}_{sq}(\mathbf{w}) = \frac{1}{2n} \sum_{i=1}^n (\mathbf{w}\mathbf{x}_i^T - y_i)^2 \quad (2.20)$$

with its partial derivatives equal to:

$$\begin{aligned} \frac{\delta \mathcal{L}}{\delta w_i} &= \frac{1}{n} \sum_{i=1}^n x_i (\mathbf{w}\mathbf{x}^T - y_i) \quad \forall i = 1 \dots d \\ \frac{\delta \mathcal{L}}{\delta w_0} &= \frac{1}{n} \sum_{i=1}^n (\mathbf{w}\mathbf{x}^T - y_i) \end{aligned} \quad (2.21)$$

Note that the location of minimum of the (2.17) is identical to (2.20). The latter function is just multiplied by a constant value. However, adjusting the learning rate is easier when we use (2.20) to find optimal \mathbf{w} . One important property of the squared loss function with linear models is that it is a convex function. This means, the gradient descend method will always converge at the global minimum regardless of the initial point. It is worth mentioning this property does not hold if the classification model is nonlinear function of its parameters. We minimized the square loss function on the dataset shown in Fig. 2.1. Figure 2.6 shows the status of the gradient descend in four different iterations.

The background of the plots shows the label of each region according to sign of classification score computed for each point on the plane. The initial model is very inaccurate since most of the vectors are classified as red. However, it becomes more accurate after 400 iterations. Finally, it converges at Iteration 2000. As you can see, the amount of change in the first iterations is higher than the last iterations. By looking at the partial derivatives, we realize that the change of a parameter is directly related to the prediction error. Because the prediction error is high in the first iterations, parameters of the model changes considerably. As the error reduces, parameters also change slightly. The intuition behind the least square loss function can be studied from another perspective.

Assume the two hypothetical lines parallel to the linear model shown in Fig. 2.7. The actual distance of these lines from the linear model is equal to 1. In the case of negative region, the signed distance of the hypothetical line is -1 . On the other hand, we know from our previous discussion that the normalized distance of samples \mathbf{x} from the decision boundary is equal to $\frac{f(\mathbf{x})}{\|\mathbf{w}\|}$ where, here, \mathbf{w} refers to the parameter vector before augmenting. If consider the projection of \mathbf{x} on \mathbf{w} and utilize the fact that $\mathbf{w}\mathbf{x} = \|\mathbf{w}\| \|\mathbf{x}\| \cos(\theta)$, we will see that the unnormalized distance of sample \mathbf{x} from

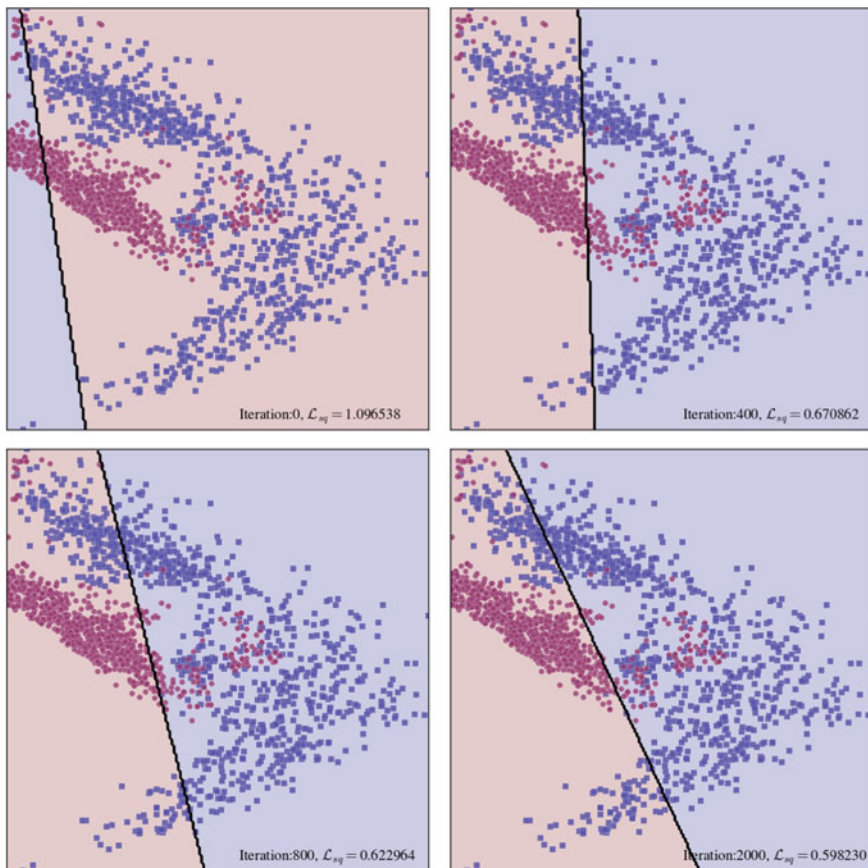


Fig. 2.6 Status of the gradient descent in four different iterations. The parameter vector \mathbf{w} changes greatly in the first iterations. However, as it gets closer to the minimum of the squared loss function, it changes slightly

the linear model is equal to $f(\mathbf{x})$. Based on that, least square loss tries to minimize the sum of unnormalized distance of samples from their actual hypothetical line.

One problem with least square loss function is that it is sensitive to outliers. This is illustrated using an example on Fig. 2.8. In general, noisy samples do not come from the same distribution as clean samples. This means that they might not be close to clean samples in the d -dimensional space. On the one hand, square loss function tries to minimize the prediction error between the samples. On the other hand, because the noisy samples are located far from the clean samples, they have a large prediction error. For this reason, some of the clean samples might be sacrificed in order to reduce the error with the noisy sample. We can see in this figure that because of noisy sample, the model is not able to fit on the data accurately.

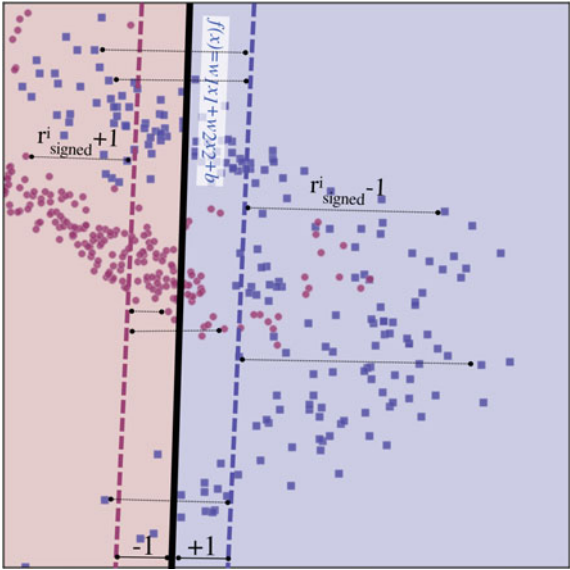


Fig. 2.7 Geometrical intuition behind least square loss function is to minimize the sum of unnormalized distances between the training samples x_i and their corresponding hypothetical line

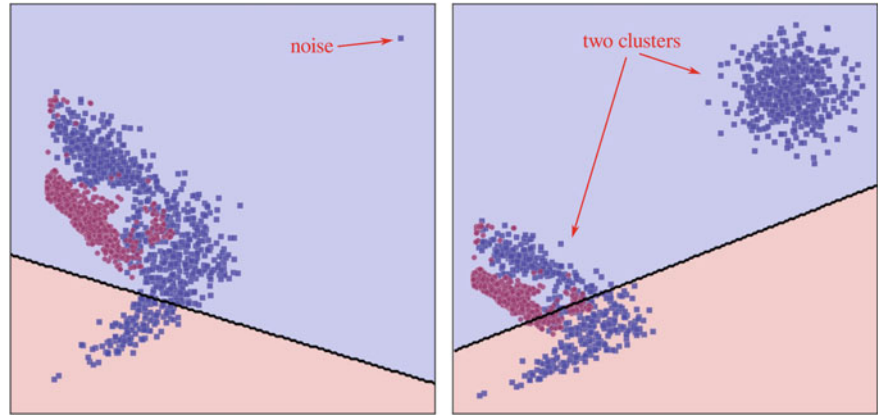


Fig. 2.8 Square loss function may fit inaccurately on training data if there are noisy samples in the dataset

It is also likely in practice that clean samples form two or more separate clusters in the d -dimensional space. Similar to the scenario of noisy samples, squared loss tries to minimize the prediction error of the samples in the far cluster as well. As we can see on the figure, the linear model might not be accurately fitted on the data if clean samples form two or more separate clusters.

This problem is due to the fact that the squared loss does not take into account the label of the prediction. Instead, it considers the classification score and computes the prediction error. For example, assume the training pairs:

$$\{(\mathbf{x}_a, 1), (\mathbf{x}_b, 1), (\mathbf{x}_c, -1), (\mathbf{x}_d, -1)\} \quad (2.22)$$

Also, suppose two different configurations \mathbf{w}_1 and \mathbf{w}_2 for the parameters of the linear model with the following responses on the training set:

$$\begin{array}{l|l} f_{\mathbf{w}_1}(\mathbf{x}_a) = 10 & f_{\mathbf{w}_2}(\mathbf{x}_a) = 5 \\ f_{\mathbf{w}_1}(\mathbf{x}_b) = 1 & f_{\mathbf{w}_2}(\mathbf{x}_b) = 2 \\ f_{\mathbf{w}_1}(\mathbf{x}_c) = -0.5 & f_{\mathbf{w}_2}(\mathbf{x}_c) = 0.2 \\ f_{\mathbf{w}_1}(\mathbf{x}_d) = -1.1 & f_{\mathbf{w}_2}(\mathbf{x}_d) = -0.5 \\ \hline \mathcal{L}_{sq}(\mathbf{w}_1) = 10.15 & \mathcal{L}_{sq}(\mathbf{w}_2) = 2.33 \end{array} \quad (2.23)$$

In terms of squared loss, \mathbf{w}_2 is better than \mathbf{w}_1 . But, if we count the number of misclassified samples we see that \mathbf{w}_1 is the better configuration. In classification problems, we are mainly interested in reducing the number of incorrectly classified samples. As the result, \mathbf{w}_1 is favorable to \mathbf{w}_2 in this setting. In order to alleviate this problem of squared loss function we can define the following loss function to estimate 0/1 loss:

$$\mathcal{L}_{sg}(\mathbf{w}) = \sum_{i=1}^n 1 - \text{sign}(f(\mathbf{x}_i))y_i. \quad (2.24)$$

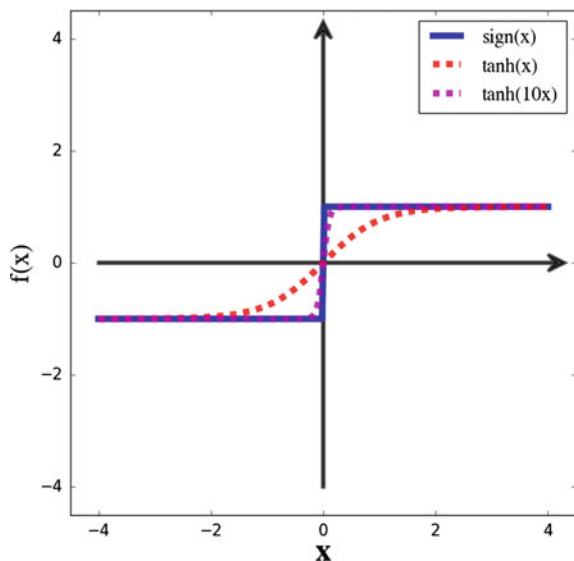
If $f(\mathbf{x})$ predicts correctly, its sign will be identical to the sign of y_i in which their multiplication will be equal to +1. Thus, the outcome of $1 - \text{sign}(f(\mathbf{x}_i))y_i$ will be zero. In contrary, if $f(\mathbf{x})$ predicts incorrectly, its sign will be different from y_i . So, their multiplication will be equal to -1. That being the case, the result of $1 - \text{sign}(f(\mathbf{x}_i))y_i$ will be equal to 2. For this reason, \mathbf{w}_{sg} returns the twice of number of misclassified samples.

The above loss function look intuitive and it is not sensitive to far samples. However, finding the minimum of this loss function using gradient-based optimization methods is hard. The reason is because of *sign* function. One solution to solve this problem is to approximate the sign function using a differentiable function. Fortunately, *tanh* (Hyperbolic tangent) is able to accurately approximate the sign function. More specifically, $\tanh(kx) \approx \text{sign}(x)$ when $k \gg 1$. This is illustrated in Fig. 2.9. As k increases, the *tanh* function will be able to approximate the sign function more accurately.

By replacing the sign function with *tanh* in (2.24), we will obtain:

$$\mathcal{L}_{sg}(\mathbf{w}) = \sum_{i=1}^n 1 - \tanh(kf(\mathbf{x}_i))y_i. \quad (2.25)$$

Fig. 2.9 The sign function can be accurately approximated using $\tanh(kx)$ when $k \gg 1$



Similar to the squared loss function, the sign loss function can be minimized using the gradient descend method. To this end, we need to compute the partial derivatives of the sign loss function with respect to its parameters:

$$\begin{aligned} \frac{\partial \mathcal{L}_{sg}(\mathbf{w})}{\partial w_i} &= -kx_i y (1 - \tanh^2(kf(\mathbf{x}))) \\ \frac{\partial \mathcal{L}_{sg}(\mathbf{w})}{\partial w_0} &= -ky (1 - \tanh^2(kf(\mathbf{x}))) \end{aligned} \quad (2.26)$$

If we train a linear model using the sign loss function and the gradient descend method on the datasets shown in Figs. 2.1 and 2.8, we will obtain the results illustrated in Fig. 2.10. According to the results, the sign loss function is able to deal with separated clusters of samples and outliers as opposed to the squared loss function.

Even though the sign loss using the tanh approximation does a fairly good job on our sample dataset, it has one issue which makes the optimization slow. In order to explain this issue, we should study the derivative of tanh function. We know from calculus that $\frac{\partial \tanh(x)}{\partial x} = 1 - \tanh^2(x)$. Figure 2.11 shows its plot. We can see that the derivative of tanh saturates as $|x|$ increases. Also, it saturates more rapidly if we set k to a positive number greater than 1. On the other hand, we know from (2.26) that the gradient of the sign loss function directly depends on the derivative of tanh function. That means if the derivative of a sample falls into the saturated region, its magnitude is close to zero. As a consequence, parameters change very slightly. This phenomena which is called the *saturated gradients* problem slows down the convergence speed of the gradient descend method. As we shall see in the next chapters, in complex models such as neural networks with millions of parameters, the model may not

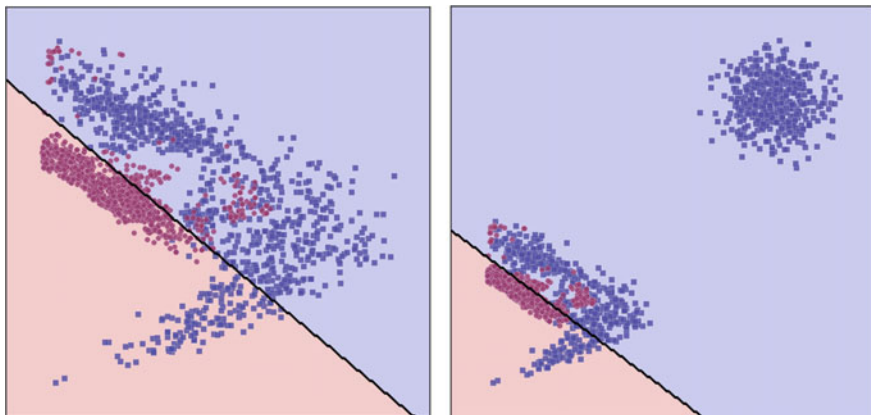
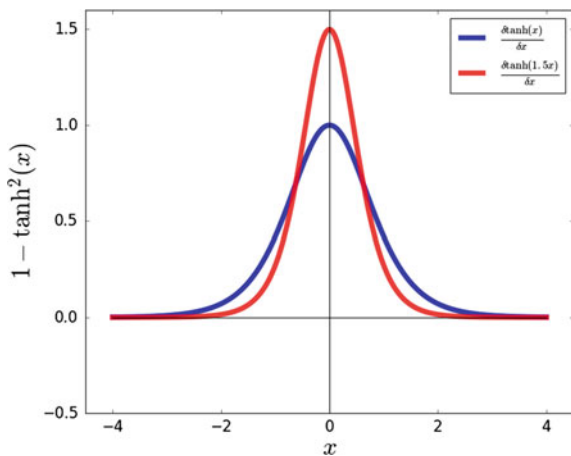


Fig. 2.10 The sign loss function is able to deal with noisy datasets and separated clusters problem mentioned previously

Fig. 2.11 Derivative of $\tanh(kx)$ function saturates as $|x|$ increases. Also, the ratio of saturation growth rapidly when $k > 1$



be able to adjust the parameters of initial layers since the saturated gradients are propagated from last layers back to the first layers.

2.2.2 Hinge Loss

Earlier in this chapter, we explained that the normalized distance of sample \mathbf{x} from the decision boundary is equal to $\frac{|f(\mathbf{x})|}{\|\mathbf{w}\|}$. Likewise, *margin* of \mathbf{x} is obtained by computing $(\mathbf{w}\mathbf{x}^T)y$ where y is the corresponding label of \mathbf{x} . The margin tell us how correct is the classification of the sample. Assume that the label of \mathbf{x}_a is -1 . If $\mathbf{w}\mathbf{x}_a^T$ is negative, its multiplication with $y = -1$ will be positive showing that the sample is classified correctly with a confidence analogous to $|\mathbf{w}\mathbf{x}_a^T|$. Likewise, if $\mathbf{w}\mathbf{x}_a^T$ is positive, its

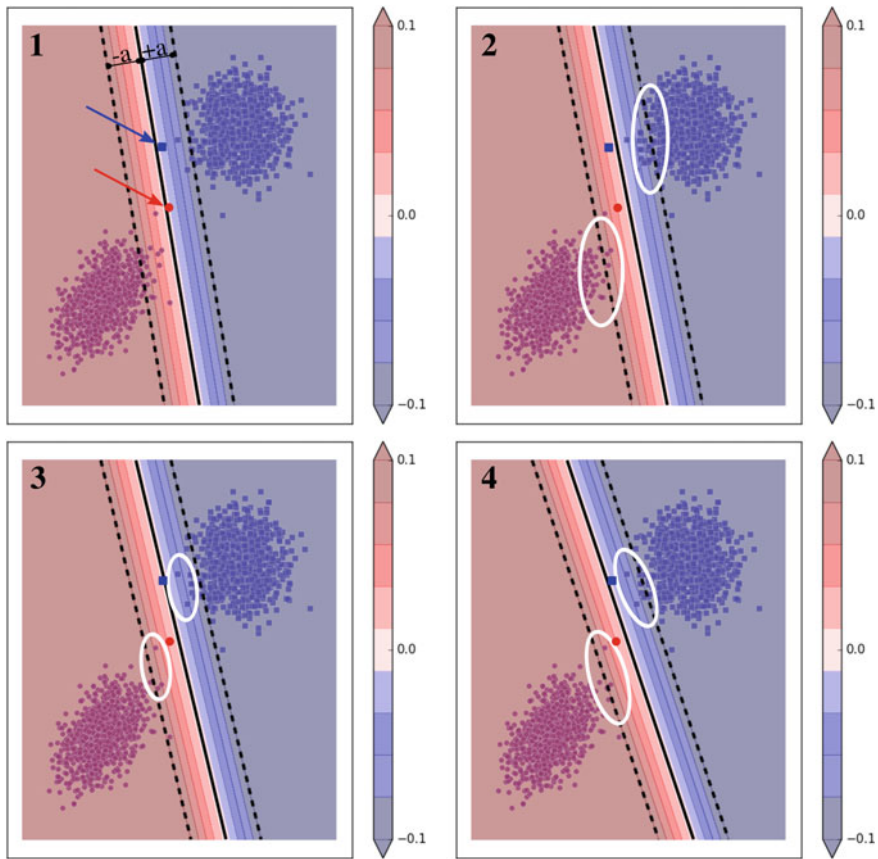


Fig.2.12 Hinge loss increases the margin of samples while it is trying to reduce the classification error. Refer to text for more details

multiplication with $y = -1$ will be negative showing that the sample is classified incorrectly with a magnitude equal to $|\mathbf{w}\mathbf{x}^T|$.

The basic idea behind *hinge* loss is not only to train a classifier but also to increase margin of samples. This is an important property which may increase tolerance of the classifier against noisy samples. This is illustrated in Fig.2.12 on a synthetic dataset which are perfectly separable using a line. The solid line shows the decision boundary and the dashed lines illustrate the borders of the critical region centered at the decision boundary of this model. It means that the margin of samples in this region is less than $|a|$. In contrast, margin of samples outside this region is high which implies that the model is more confident in classification of samples outside this region. Also, the colorbar next to each plots depicts the margin corresponding to each color on the plots.

In the first plot, two *test* samples are indicated which are not used during the training phase. One of them belongs to circles and the another one belongs to squares. Although the line adjusted on the training samples is able to perfectly discriminate the training samples, it will incorrectly classify the test red sample. Comparing the model in the second plot with the first plot, we observe that fewer circles are inside the critical region but the number of squares increase inside this region. In the third plot, the overall margin of samples are better if we compare the samples marked with white ellipses on these plots. Finally, the best overall margin is found in the fourth plot where the test samples are also correctly classified.

Maximizing the margin is important since it may increase the tolerance of model against noise. The test samples in Fig. 2.12 might be noisy samples. However, if the margin of the model is large, it is likely that these samples are classified correctly. Nonetheless, it is still possible that we design a test scenario where the first plot could be more accurate than the fourth plot. But, as the number of training samples increases a classifier with maximum margin is likely to be more stable. Now, the question is how we can force the model by a loss function to increase its accuracy and margin simultaneously? The hinge loss function achieves these goals using the following relation:

$$\mathcal{L}_{hinge}(\mathbf{w}) = \frac{1}{n} \sum_{i=1}^n \max(0, a - \mathbf{w}\mathbf{x}_i^T y_i) \quad (2.27)$$

where $y_i \in \{-1, 1\}$ is the label of the training sample \mathbf{x}_i . If signs of $\mathbf{w}\mathbf{x}_i$ and y_i are equal, the term inside the sum operator will return 0 since the value of the second parameter in the max function will be negative. In contrast, if their sign are different, this term will be equal to $a - \mathbf{w}\mathbf{x}_i^T y_i$ increasing the value of loss. Moreover, if $\mathbf{w}\mathbf{x}_i^T y_i < a$ this implies that \mathbf{x} is within the critical region of the model and it increases the value of loss. By minimizing the above loss function we will obtain a model with maximum margin and high accuracy at the same time. The term inside the sum operator can be written as:

$$\max(0, a - \mathbf{w}\mathbf{x}_i^T y_i) = \begin{cases} a - \mathbf{w}\mathbf{x}_i^T y_i & \mathbf{w}\mathbf{x}_i^T y_i < a \\ 0 & \mathbf{w}\mathbf{x}_i^T y_i \geq a \end{cases} \quad (2.28)$$

Using this formulation and denoting $\max(0, a - \mathbf{w}\mathbf{x}_i^T y_i)$ with H , we can compute the partial derivatives of $\mathcal{L}_{hinge}(\mathbf{w})$ with respect to \mathbf{w} :

$$\begin{aligned} \frac{\delta H}{\delta w_i} &= \begin{cases} -x_i y_i & \mathbf{w}\mathbf{x}_i^T y_i < a \\ 0 & \mathbf{w}\mathbf{x}_i^T y_i \geq a \end{cases} \\ \frac{\delta H}{\delta w_0} &= \begin{cases} -y_i & \mathbf{w}\mathbf{x}_i^T y_i < a \\ 0 & \mathbf{w}\mathbf{x}_i^T y_i \geq a \end{cases} \end{aligned} \quad (2.29)$$

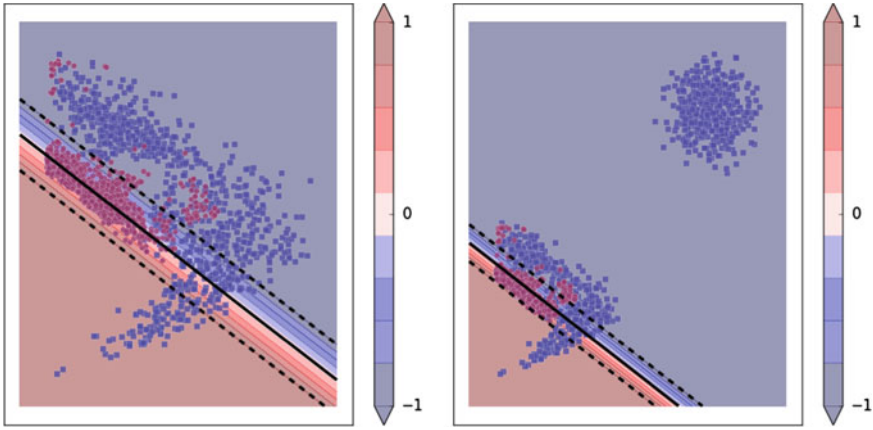


Fig. 2.13 Training a linear classifier using the hinge loss function on two different datasets

$$\begin{aligned}\frac{\delta \mathcal{L}_{hinge}(\mathbf{w})}{\delta w_i} &= \frac{1}{n} \sum_{i=1}^n \frac{\delta H}{w_i} \\ \frac{\delta \mathcal{L}_{hinge}(\mathbf{w})}{\delta w_0} &= \frac{1}{n} \sum_{i=1}^n \frac{\delta H}{w_0}\end{aligned}\tag{2.30}$$

It should be noted that, $\mathcal{L}_{hinge}(\mathbf{w})$ is not continuous at $\mathbf{w}\mathbf{x}_i^T y_i = a$ and, consequently, it is not differentiable at $\mathbf{w}\mathbf{x}_i^T y_i = a$. For this reason, the better choice for optimizing the above function might be a *subgradient*-based method. However, it might never happen in a training set to have a sample in which $\mathbf{w}\mathbf{x}_i^T y_i$ is exactly equal to a . For this reason, we can still use the gradient descend method for optimizing this function.

Furthermore, the loss function does not depend on the value of a . It only affects the magnitude of \mathbf{w} . In other words, \mathbf{w} is always adjusted such that as few training samples as possible fall into the critical region. For this reason, we always set $a = 1$ in practice. We minimized the hinge loss on the dataset shown in Figs. 2.1 and 2.8. Figure 2.13 illustrates the result. As before, the region between the two dashed lines indicates the critical region.

Based on the results, the model learned by the hinge loss function is able to deal with separated clusters problem. Also, it is able to learn an accurate model for the nonlinearly separable dataset. A variant of hinge loss called *squared hinge* loss has been also proposed which is defined as follows:

$$\mathcal{L}_{hinge}(\mathbf{w}) = \frac{1}{n} \sum_{i=1}^n \max(0, 1 - \mathbf{w}\mathbf{x}_i^T y_i)^2\tag{2.31}$$

The main difference between the hinge loss and the squared hinge loss is that the latter one is smoother and it may make the optimization easier. Another variant of the hinge loss function is called *modified Huber* and it is defined as follows:

$$\mathcal{L}_{huber}(\mathbf{w}) = \begin{cases} \max(0, 1 - y\mathbf{w}\mathbf{x}^T)^2 & y\mathbf{w}\mathbf{x}^T \geq -1 \\ -4y\mathbf{w}\mathbf{x}^T & \text{otherwise} \end{cases} \quad (2.32)$$

The modified Huber loss is very close to the squared hinge and they may only differ in the convergence speed. In order to use any of these variants to train a model, we need to compute the partial derivative of the loss functions with respect to their parameters.

2.2.3 Logistic Regression

None of the previously mentioned linear models are able to compute the probability of samples \mathbf{x} belonging to class $y = 1$. Formally, given a binary classification problem, we might be interested in computing $p(y = 1|\mathbf{x})$. This implies that $p(y = -1|\mathbf{x}) = 1 - p(y = 1|\mathbf{x})$. Consequently, the sample \mathbf{x} belongs to class 1 if $p(y = 1|\mathbf{x}) > 0.5$. Otherwise, it belongs to class -1. In the case that $p(y = 1|\mathbf{x}) = 0.5$, the sample is exactly on the decision boundary and it does not belong to any of these two classes. The basic idea behind *logistic regression* is to learn $p(y = 1|\mathbf{x})$ using a linear model. To this end, logistic regression transforms the score of a sample into probability by passing the score through a *sigmoid* function. Formally, logistic regression computes the *posterior* probability as follows:

$$p(y = 1|\mathbf{x}; \mathbf{w}) = \sigma(\mathbf{w}\mathbf{x}^T) = \frac{1}{1 + e^{-\mathbf{w}\mathbf{x}^T}}. \quad (2.33)$$

In this equation, $\sigma : \mathbb{R} \rightarrow [0, 1]$ is the *logistic sigmoid* function. As it is shown in Fig. 2.14, the function has a S shape and it saturates as $|x|$ increases. In other words, derivative of function approaches to zero as $|x|$ increases.

Since range of the sigmoid function is $[0, 1]$ it satisfies requirements of a probability measure function. Note that (2.33) directly models the *posterior* probability which means by using appropriate techniques that we shall explain later, it is able to model likelihood and a priori of classes. Taking into account the fact that (2.33) returns the probability of a sample, the loss function must be also build based on probability of the whole training set given a specific \mathbf{w} . Formally, given a dataset of n training samples, our goal is to maximize their joint probability which is defined as:

$$\mathcal{L}_{logistic}(\mathbf{w}) = p(\mathbf{x}_1 \cap \mathbf{x}_2 \cap \dots \cap \mathbf{x}_n) = p\left(\bigcap_{i=1}^n \mathbf{x}_i\right). \quad (2.34)$$

Modeling the above joint probability is not trivial. However, it is possible to decompose this probability into smaller components. To be more specific, the probability

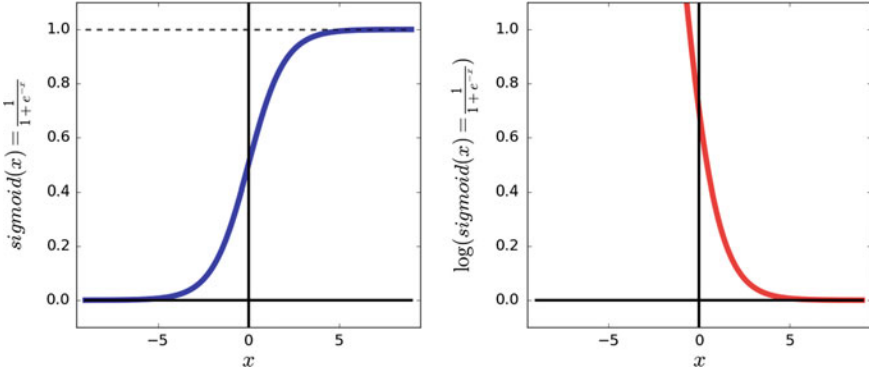


Fig. 2.14 Plot of the sigmoid function (*left*) and logarithm of the sigmoid function (*right*). The domain of the sigmoid function is real numbers and its range is $[0, 1]$

of \mathbf{x}_i does not depend on the probability of \mathbf{x}_j . For this reason and taking into account the fact that $p(A, B) = p(A)p(B)$ if A and B are independent events, we can decompose the above joint probability into product of probabilities:

$$\mathcal{L}_{\text{logistic}}(\mathbf{w}) = \prod_{i=1}^n p(y_i | \mathbf{x}_i) \quad (2.35)$$

where $p(\mathbf{x}_i)$ is computed using:

$$p(y_i | \mathbf{x}_i) = \begin{cases} p(y = 1 | \mathbf{x}; \mathbf{w}) & y_i == 1 \\ 1 - p(y = 1 | \mathbf{x}; \mathbf{w}) & y_i == -1 \end{cases} \quad (2.36)$$

Representing the negative class with 0 rather than -1 , the above equation can be written as:

$$p(\mathbf{x}_i) = p(y = 1 | \mathbf{x}; \mathbf{w})^{y_i} (1 - p(y = 1 | \mathbf{x}; \mathbf{w}))^{1-y_i}. \quad (2.37)$$

This equation which is called *Bernoulli* distribution is used to model random variables with two outcomes. Plugging (2.33) into the above equation we will obtain:

$$\mathcal{L}_{\text{logistic}}(\mathbf{w}) = \prod_{i=1}^n \left(\sigma(\mathbf{w}\mathbf{x}^T)^{y_i} (1 - \sigma(\mathbf{w}\mathbf{x}^T))^{1-y_i} \right). \quad (2.38)$$

Optimizing the above function is hard. The reason is because of \prod operator which makes the derivative of the loss function intractable. However, we can apply logarithm trick to change the multiplication into summation. In other words, we can compute $\log(\mathcal{L}_{\text{logistic}}(\mathbf{w}))$:

$$\log(\mathcal{L}_{\text{logistic}}(\mathbf{w})) = \log \left(\prod_{i=1}^n \left(\sigma(\mathbf{w}\mathbf{x}^T)^{y_i} (1 - \sigma(\mathbf{w}\mathbf{x}^T))^{1-y_i} \right) \right). \quad (2.39)$$

We know from properties of logarithm that $\log(A \times B) = \log(A) + \log(B)$. As the result, the above equation can be written as:

$$\log(\mathcal{L}_{logistic}(\mathbf{w})) = \sum_{i=1}^n y_i \log \sigma(\mathbf{w}\mathbf{x}_i^T) + (1 - y_i) \log(1 - \sigma(\mathbf{w}\mathbf{x}_i^T)). \quad (2.40)$$

If each sample in the training set is classified correctly, $p(\mathbf{x}_i)$ will be close to 1 and if it is classified incorrectly, it will be close to zero. Therefore, the best classification will be obtained if we find the maximum of the above function. Although this can be done using gradient *ascend* methods, it is preferable to use gradient descend methods. Because gradient descend can be only applied on minimization problems, we can multiply both sides of the equation with -1 in order to change the maximum of the loss into minimum:

$$E = -\log(\mathcal{L}_{logistic}(\mathbf{w})) = -\sum_{i=1}^n y_i \log \sigma(\mathbf{w}\mathbf{x}_i^T) + (1 - y_i) \log(1 - \sigma(\mathbf{w}\mathbf{x}_i^T)). \quad (2.41)$$

Now, we can use gradient descend to find the minimum of the above loss function. This function is called *cross-entropy* loss. In general, these kind of loss functions are called *negative log-likelihood* functions. As before, we must compute the partial derivatives of the loss function with respect to its parameters in order to apply the gradient descend method. To this end, we need to compute the derivative of $\sigma(a)$ with respect to its parameter which is equal to:

$$\frac{\delta \sigma(a)}{a} = \sigma(a)(1 - \sigma(a)). \quad (2.42)$$

Then, we can utilize the chain rule to compute the partial derivative of the above loss function. Doing so, we will obtain:

$$\begin{aligned} \frac{\delta E}{w_i} &= (\sigma(\mathbf{w}\mathbf{x}_i^T) - y_i)x_i \\ \frac{\delta E}{w_0} &= \sigma(\mathbf{w}\mathbf{x}_i^T) - y_i \end{aligned} \quad (2.43)$$

Note that in contrast to the previous loss functions, here, $y_i \in \{0, 1\}$. In other words, the negative class is represented using 0 instead of -1 . Figure 2.15 shows the result of training linear models on the two previously mentioned datasets. We see that logistic regression is find an accurate model even when the training samples are scattered in more than two clusters. Also, in contrast to the squared function, it is less sensitive to outliers.

It is possible to formulate the logistic loss with $y_i \in \{-1, 1\}$. In other words, we can represent the negative class using -1 and reformulate the logistic loss function.

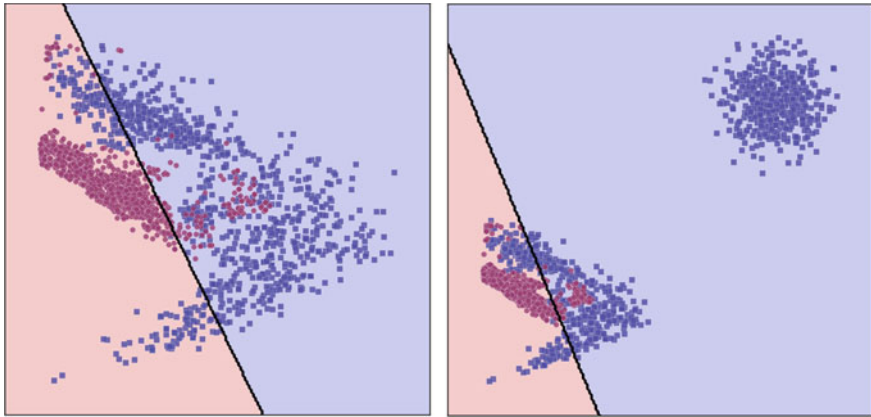


Fig. 2.15 Logistic regression is able to deal with separated clusters

More specifically, we can rewrite the logistic equations as follows:

$$\begin{aligned} p(y = 1|\mathbf{x}) &= \frac{1}{1 + e^{-\mathbf{w}\mathbf{x}^T}} \\ p(y = -1|\mathbf{x}) &= 1 - p(y = 1|\mathbf{x}) = \frac{1}{1 + e^{+\mathbf{w}\mathbf{x}^T}} \end{aligned} \quad (2.44)$$

This implies that:

$$p(y_i|\mathbf{x}_i) = \frac{1}{1 + e^{-y_i\mathbf{w}\mathbf{x}^T}} \quad (2.45)$$

Plugging this in (2.35) and taking the negative logarithm, we will obtain:

$$\mathcal{L}_{logistic}(\mathbf{w}) = \sum_{i=1}^n \log(1 + e^{-y_i\mathbf{w}\mathbf{x}^T}) \quad (2.46)$$

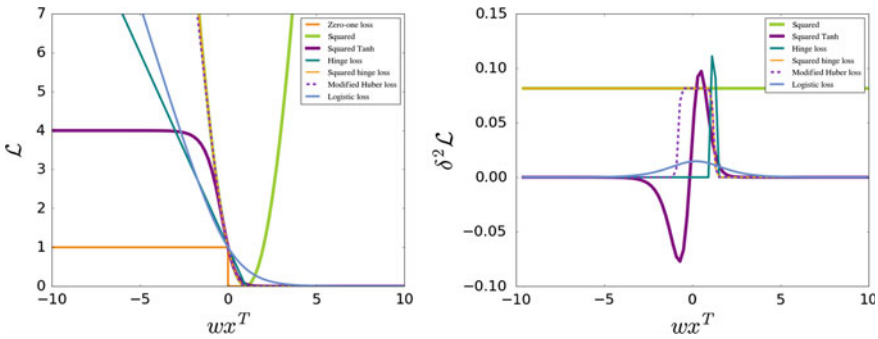
It should be noted that (2.41) and (2.46) are identical and they can lead to the same solution. Consequently, we can use any of them to fit a linear model. As before, we only need to compute partial derivatives of the loss function and use them in the gradient descend method to minimize the loss function.

2.2.4 Comparing Loss Function

We explained 7 different loss functions for training a linear model. We also discussed some of their properties in presence of outliers and separated clusters. In this section, we compare these loss functions from different perspectives. Table 2.1 compares different loss functions. Besides, Fig. 2.16 illustrates the plot of the loss functions along with their second derivative.

Table 2.1 Comparing different loss functions

Loss function	Equation	Convex
Zero-one loss	$\mathcal{L}_{0/1}(\mathbf{w}) = \sum_{i=1}^n H_{0/1}(\mathbf{w}\mathbf{x}_i^T, y_i)$	No
Squared loss	$\mathcal{L}_{sq}(\mathbf{w}) = \sum_{i=1}^n (\mathbf{w}\mathbf{x}_i^T - y_i)^2$	Yes
Tanh Squared loss	$\mathbf{w}_{sg} = \sum_{i=1}^n 1 - \tanh(kf(\mathbf{x}_i))y_i$	No
Hinge loss	$\mathcal{L}_{hinge}(\mathbf{w}) = \frac{1}{n} \sum_{i=1}^n \max(0, 1 - \mathbf{w}\mathbf{x}_i^T y_i)$	Yes
Squared hinge loss	$\mathcal{L}_{hinge}(\mathbf{w}) = \frac{1}{n} \sum_{i=1}^n \max(0, 1 - \mathbf{w}\mathbf{x}_i^T y_i)^2$	Yes
Modified Huber	$\mathcal{L}_{huber}(\mathbf{w}) = \begin{cases} \max(0, 1 - y\mathbf{w}\mathbf{x}^T)^2 & y\mathbf{w}\mathbf{x}^T \geq -1 \\ -4y\mathbf{w}\mathbf{x}^T & otherwise \end{cases}$	Yes
Logistic loss	$-\log(\mathcal{L}_{logistic}(\mathbf{w})) = -\sum_{i=1}^n y_i \log \sigma(\mathbf{w}\mathbf{x}^T) + (1 - y_i) \log(1 - \sigma(\mathbf{w}\mathbf{x}^T))$	Yes

**Fig. 2.16** Tanh squared loss and zero-one loss functions are not convex. In contrast, the squared loss, the hinge loss, and its variant and the logistic loss functions are convex

Informally, a one variable function is convex if for every pair of points x and y , the function falls below their connecting line. Formally, if the second derivative of a function is positive, the function is convex. Looking at the plots of each loss function and their derivatives, we realize that the Tanh squared loss and the zero-one loss functions are not convex. In contrast, hinge loss and its variants as well as the logistic loss are all convex functions. Convexity is an important property since it guarantees that the gradient descend method will find the global minimum of the function provided that the classification model is linear.

Let us have a closer look at the logistic loss function on the dataset which is linearly separable. Assume the parameter vector $\hat{\mathbf{w}}$ such that two classes are separated perfectly. This is shown by the top-left plot in Fig. 2.17. However, because the magnitude of $\hat{\mathbf{w}}$ is low $\sigma(\mathbf{w}\mathbf{x}^T)$ is smaller than 1 for the points close to the decision boundary. In order to increase the value of $\sigma(\mathbf{w}\mathbf{x}^T)$ without affecting the classification accuracy, the optimization method may increase the magnitude of $\hat{\mathbf{w}}$. As we can see in the other plots, as the magnitude increases, the logistic loss reduces. Magnitude of $\hat{\mathbf{w}}$ can increase infinitely resulting the logistic to approach zero.

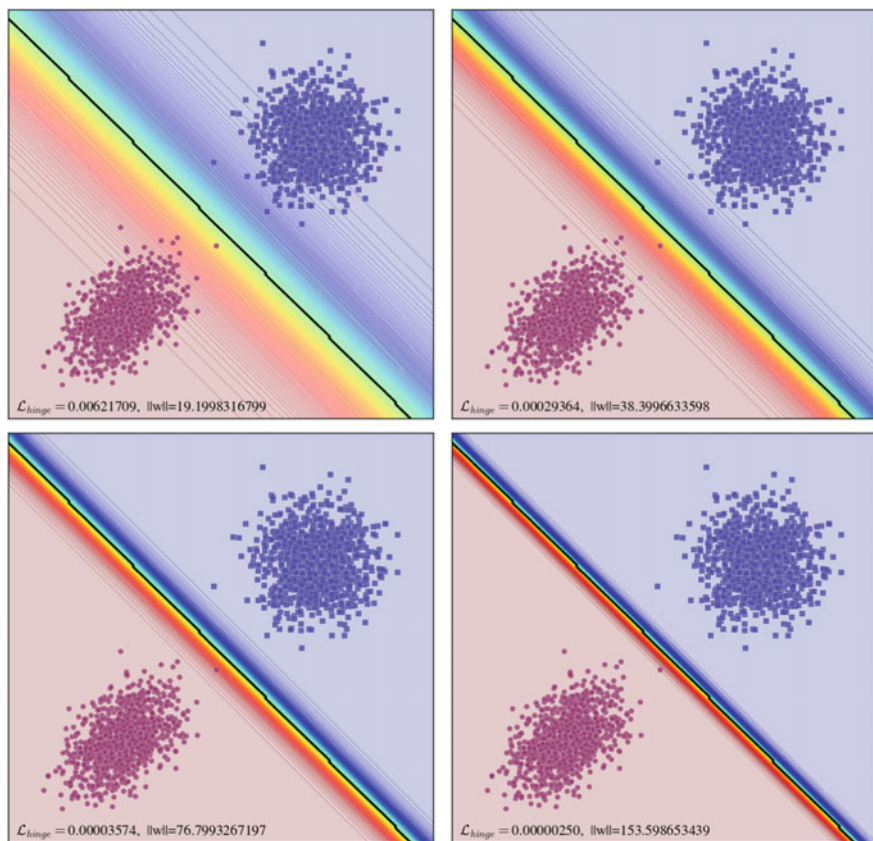


Fig. 2.17 Logistic regression tries to reduce the logistic loss even after finding a hyperplane which discriminates the classes perfectly

However, as we will explain in the next chapter, parameter vectors with high magnitude may suffer from a problem called *overfitting*. For this reason, we are usually interested in finding parameter vectors with low magnitudes. Looking at the plot of the logistic function in Fig. 2.16, we see that the function approaches to zero at infinity. This is the reason that the magnitude of model increases.

We can analyze the hinge loss function from the same perspective. Looking at the plot of the hinge loss function, we see that it becomes zero as soon as it finds a hyperplane in which all the samples are classified correctly and they are outside the critical region. We fitted a linear model using the hinge loss function on the same dataset as the previous paragraph. Figure 2.18 shows that after finding a hyperplane that classifies the samples perfectly, the magnitude of \mathbf{w} increases until all the samples are outside the critical region. At this point, the error becomes zero and \mathbf{w} does not change anymore. In other words, $\|\mathbf{w}\|$ has an upper bound when we find it using the hinge loss function.

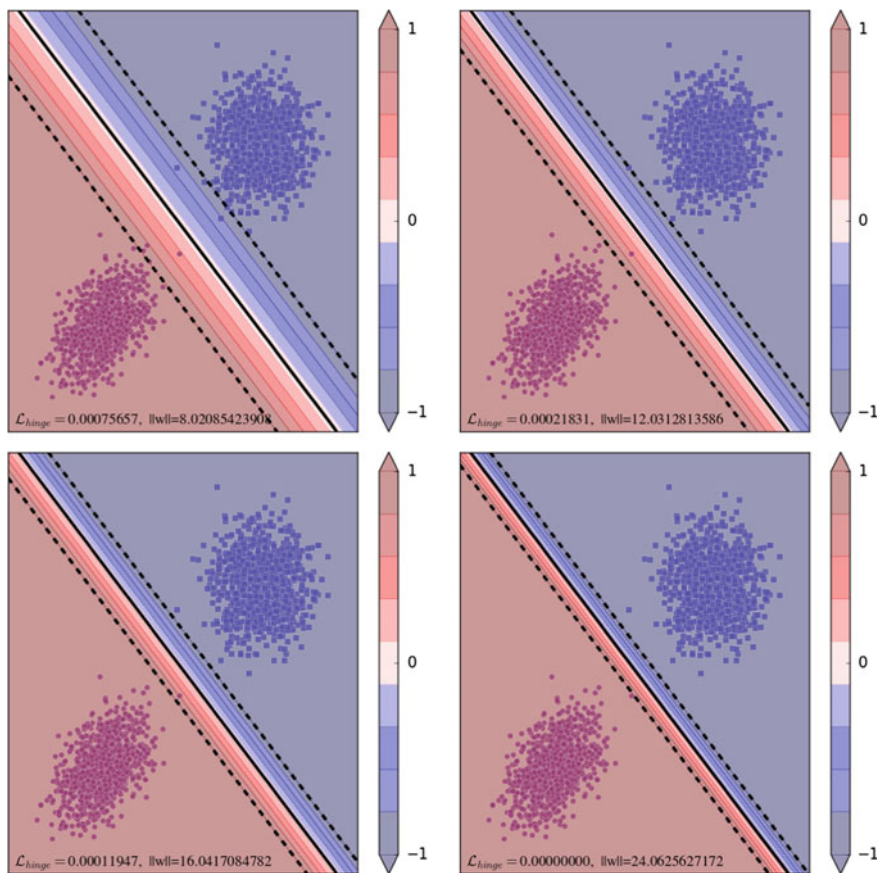


Fig. 2.18 Using the hinge loss function, the magnitude of \mathbf{w} changes until all the samples are classified correctly and they do not fall into the critical region

The above argument about the logistic regression does not hold when the classes are not linearly separable. In other words, in the case that classes are nonlinearly separable, it is not possible to perfectly classify all the training samples. Consequently, some of the training samples are always classified incorrectly. In this case, as it is shown in Fig. 2.19, if $\|\mathbf{w}\|$ increases, the error of the misclassified samples also increases resulting in a higher loss. For this reason, the optimization algorithm change the value of \mathbf{w} for a limited time. In other words, there could be an upper bound for $\|\mathbf{w}\|$ when the classes are not linearly separable.

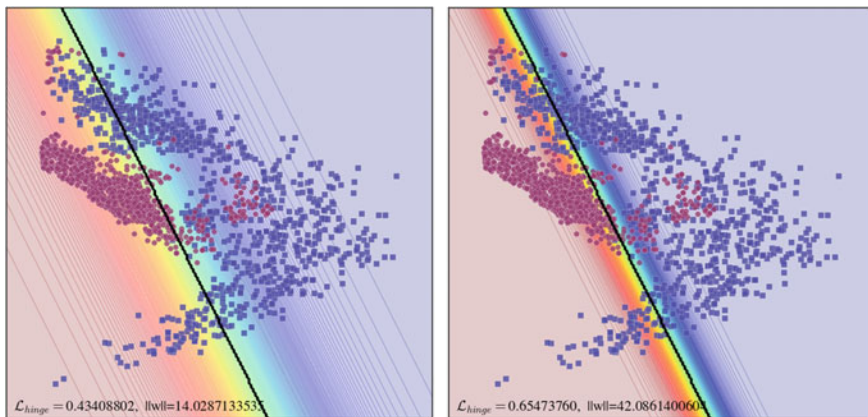


Fig. 2.19 When classes are not linearly separable, $\|\mathbf{w}\|$ may have an upper bound in logistic loss function

2.3 Multiclass Classification

In the previous section, we mentioned a few techniques for training a linear classifier on binary classification problems. Recall from the previous section that in a binary classification problem our goal is to classify the input $\mathbf{x} \in \mathbb{R}^d$ into one of two classes. A multiclass classification problem is a more generalized concept in which \mathbf{x} is classified into more than two classes. For example, suppose we want to classify 10 different speed limit signs starting from 30 to 120 km/h. In this case, \mathbf{x} represents the image of a speed limit sign. Then, our goal is to find the model $f: \mathbb{R}^d \rightarrow \mathcal{Y}$ where $\mathcal{Y} = \{0, 1, \dots, 9\}$. The model $f(\mathbf{x})$ accepts a d -dimensional real vector and returns a categorical integer between 0 and 9. It is worth mentioning that \mathcal{Y} is not an ordered set. It can be any set with 10 different *symbols*. However, for the sake of simplicity, we usually use integer numbers to show classes.

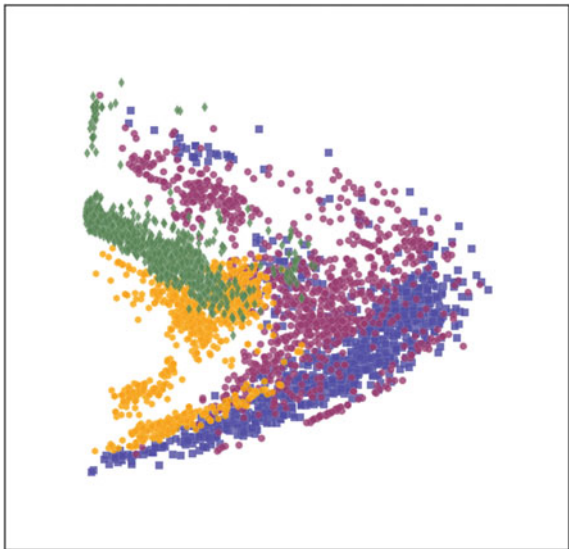
2.3.1 One Versus One

A multiclass classifier can be build using a group of binary classifiers. For instance, assume the 4-class classification problem illustrated in Fig. 2.20 where $\mathcal{Y} = \{0, 1, 2, 3\}$. One technique for building a multiclass classifier using a group of binary classifier is called *one-versus-one* (OVO).

Given the dataset $\mathcal{X} = \{(\mathbf{x}_0, y_0), \dots, (\mathbf{x}_n, y_n)\}$ where $\mathbf{x}_i \in \mathbb{R}^d$ and $y_i \in \{0, 1, 2, 3\}$, we first pick the samples from \mathcal{X} with label 0 or 1. Formally, we create the following dataset:

$$\mathcal{X}_{0|1} = \{\mathbf{x}_i \mid \mathbf{x}_i \in \mathcal{X} \wedge y_i \in \{0, 1\}\} \quad (2.47)$$

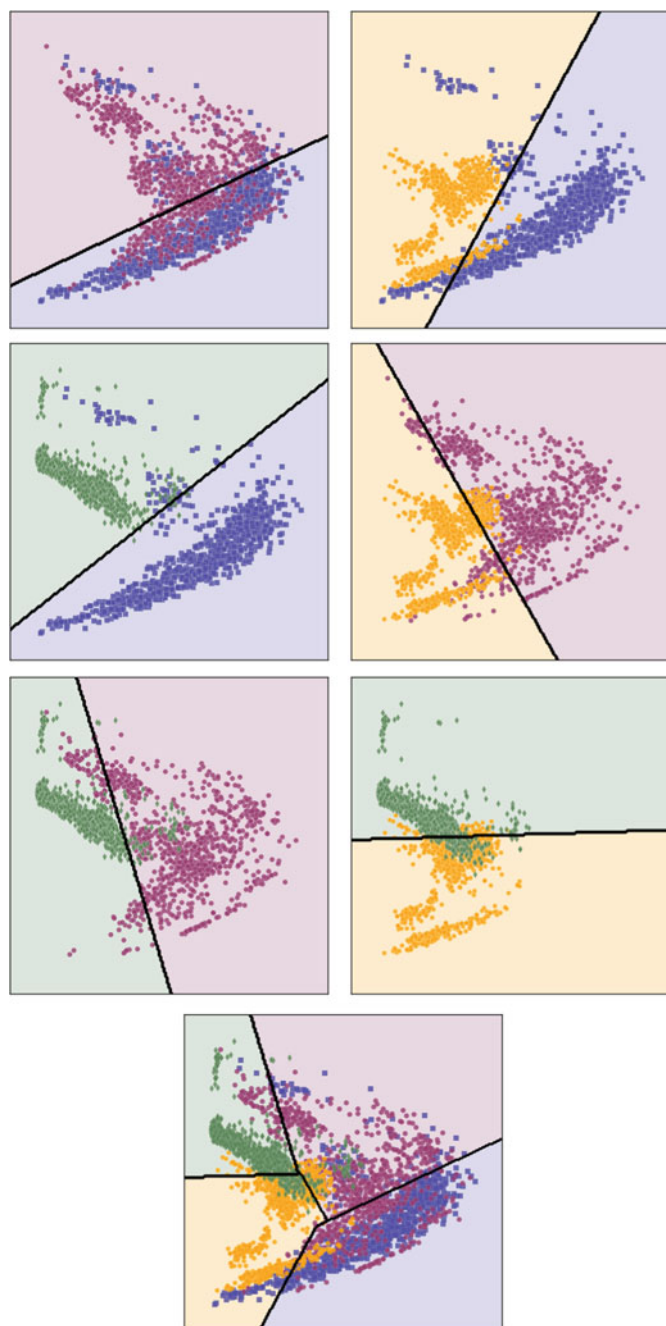
Fig. 2.20 A samples dataset including four different classes. Each class is shown using a unique color and shape



and a binary classifier is fitted on $\mathcal{X}_{0|1}$. Similarly, $\mathcal{X}_{0|2}$, $\mathcal{X}_{0|3}$, $\mathcal{X}_{1|2}$, $\mathcal{X}_{1|3}$ and $\mathcal{X}_{2|3}$ are created a separate binary classifiers are fitted on each of them. By this way, there will be six binary classifiers. In order to classify the new input \mathbf{x}_q into one of four classes, it is first classified using each of these 6 classifiers. We know that each classifier will yield an integer number between 0 and 3. Since there are six classifiers, one of the integer numbers will be repeated more than others. The class of \mathbf{x}_q is equal to the number with highest occurrence. From another perspective, we can think of the output of each binary classifier as a vote. Then, the winner class is the one with majority of votes. This method of classification is called *majority voting*. Figure 2.21 shows six binary classifiers trained on six pairs of classes mentioned above. Besides, it illustrates how points on the plane are classified into one of four classes using this technique.

This example can be easily extended to a multiclass classification problem with N classes. More specifically, all pairs of classes $\mathcal{X}_{a|b}$ are generated for all $a = 1 \dots N - 1$ and $b = a + 1 \dots N$. Then, a binary model $f_{a|b}$ is fitted on the corresponding dataset. By this way, $\frac{N(N-1)}{2}$ binary classifiers will be trained. Finally, an unseen sample \mathbf{x}_q is classified by computing the majority of votes produces by all the binary classifiers.

One obvious problem of one versus one technique is that the number of binary classifiers quadratically increases with the number of classes in a dataset. This means that using this technique we need to train 31125 binary classifiers for a 250-class classification problem such as traffic sign classification. This makes the one versus one approach impractical for large values of N . In addition, sometimes ambiguous results might be generated by one versus one technique. This may happen when there are two or more classes with majority of votes. For example,



◀ **Fig.2.21** Training six classifiers on the four class classification problem. One versus one technique considers all unordered pairs of classes in the dataset and fits a separate binary classifier on each pair. A input \mathbf{x} is classified by computing the majority of votes produced by each of binary classifiers. The *bottom plot* shows the class of every point on the plane into one of four classes

assume that the votes of 6 classifiers in the above example for an unseen sample are 1, 1, 2, and 2 for classes 0, 1, 2, and 3, respectively. In this case, the Class 2 and Class 3 have equally the majority votes. Consequently, the unseen sample cannot be classified. This problem might be addressed by taking into account the classification score (i.e., $\mathbf{w}\mathbf{x}^T$) produced by the binary classifiers. However, the fact remains that one versus one approach is not practical in applications with many classes.

2.3.2 One Versus Rest

Another popular approach for building a multiclass classifier using a group of binary classifiers is called *one versus rest* (OVR). It may also be called *one versus all* or *one against all* approach. As opposed to one versus one approach where $\frac{N(N-1)}{2}$ binary classifiers are created for a N-class classification problem, one versus rest approach trains only N binary classifiers to make predictions. The main difference between these two approaches are the way that they create the binary datasets.

In one versus rest technique, a binary dataset for class a is created as follows:

$$\mathcal{X}_{a|rest} = \{(\mathbf{x}_i, 1) | \mathbf{x}_i \in \mathcal{X} \wedge y_i = a\} \cup \{(\mathbf{x}_i, -1) | \mathbf{x}_i \in \mathcal{X} \wedge y_i \neq a\}. \quad (2.48)$$

Literally, $\mathcal{X}_{a|rest}$ is composed of all the samples in \mathcal{X} . The only difference is the label of samples. For creating $\mathcal{X}_{a|rest}$, we pick all the samples in \mathcal{X} with label a and add them to $\mathcal{X}_{a|rest}$ after changing their label to 1. Then, the label of all the remaining samples in \mathcal{X} is changed to -1 and they are added to $\mathcal{X}_{a|rest}$. For a N-class classification problem, $\mathcal{X}_{a|rest}$ is generated for all $a = 1 \dots N$. Finally, a binary classifier $f_{a|rest}(\mathbf{x})$ is trained on each $\mathcal{X}_{a|rest}$ using the method we previously mentioned in this chapter. An unseen sample \mathbf{x}_q is classified by computing:

$$\hat{y}_q = \arg \max_{a=1 \dots N} f_{a|rest}(\mathbf{x}_q). \quad (2.49)$$

In other words, the score of all the classifiers are computed. The classifier with the maximum score shows the class of the sample \mathbf{x}_q . We applied this technique on the dataset shown in Fig. 2.20. Figure 2.22 illustrates how the binary datasets are generated. It also shows how every point on the plane are classified using this technique.

Comparing the results from one versus one and one versus all, we observe that they are not identical. One advantage of one versus rest over one versus one approach is that the number of binary classifiers increases linearly with the number of classes.

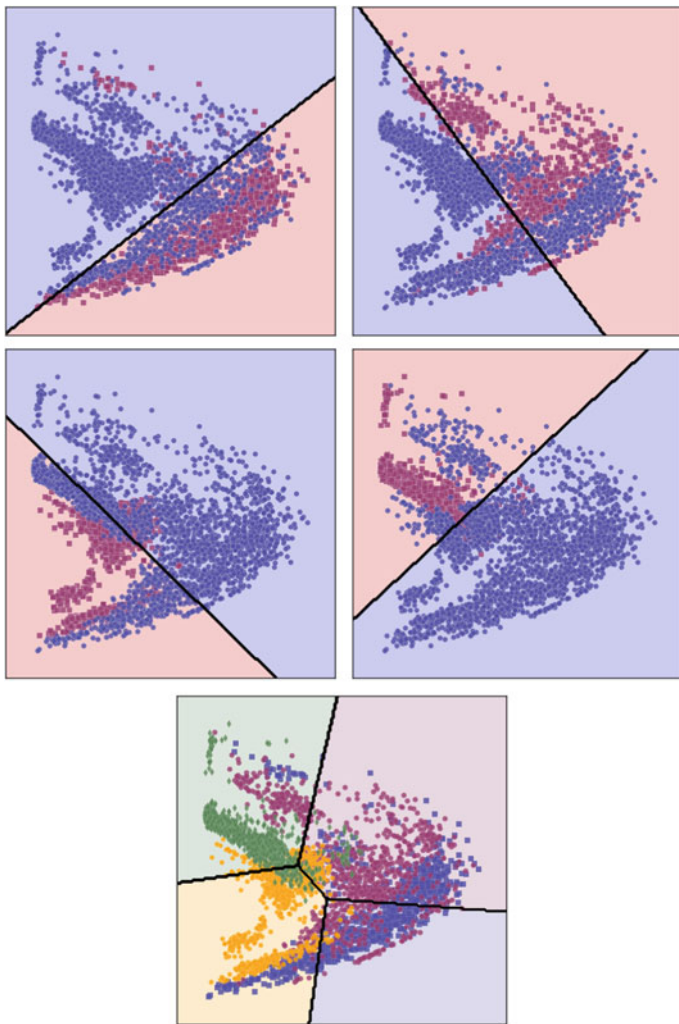


Fig. 2.22 One versus rest approach creates a binary dataset by changing the label of the class-of-interest to 1 and the label of the other classes to -1 . Creating binary datasets is repeated for all classes. Then, a binary classifier is trained on each of these datasets. An unseen sample is classified based on the classification score of the binary classifiers

For this reason, one versus rest approach is practical even when the number of classes is high. However, it poses another issue which is called *imbalanced* dataset.

We will talk thoroughly about imbalanced datasets later in this book. But, to give an insight about this problem, consider a 250-class classification problem where each class contains 1000 training samples. This means that the training dataset contains 250,000 samples. Consequently, $\mathcal{X}_{a|rest}$ will contain 1000 samples with label 1

(positive samples) and 249,000 samples with label -1 (negative samples). We know from previous section that a binary classifier is trained by minimizing a loss function. However, because the number of negative samples is 249 times more than the samples with label 1, the optimization algorithm will in fact try to minimize the loss occurred by the negative samples. As the result, the binary model might be highly biased toward negative samples and it might classify most of unseen positive samples as negative samples. For this reason, one versus rest approach usually requires a solution to tackle with highly imbalanced dataset $\mathcal{X}_{a|rest}$.

2.3.3 Multiclass Hinge Loss

An alternative solution to one versus one and one versus all techniques is to partition the d -dimensional space into N distinct regions using N linear models such that:

$$\begin{aligned}\mathcal{L}_{0/1}(\mathbf{W}) &= \sum_{i=1}^n H(\mathbf{x}, y_i) \\ H(\mathbf{x}, y_i) &= \begin{cases} 0 & y_i = \arg \max_{j=1 \dots N} f_j(\mathbf{x}_i) \\ 1 & \text{otherwise} \end{cases}\end{aligned}\quad (2.50)$$

is minimum for all the samples in the training dataset. In this equation, $\mathbf{W} \in \mathbb{R}^{N \times d+1}$ is a weight matrix indicating the weights (d weights for each linear model) and biases (1 bias for each linear model) of N linear models. Also, $\mathbf{x}_i \in \mathbb{R}^d$ is defined as before and $y_i \in \{1, \dots, N\}$ can take any of the categorical integer values between 1 and N and it indicates the class of \mathbf{x}_i . This loss function is in fact the generalization of the 0/1 loss function into N classes. Here also the objective of the above loss function is to minimize the number of incorrectly classified samples. After finding the optimal weight matrix \mathbf{W}^* , an unseen sample \mathbf{x}_q is classified using:

$$\hat{y}_q = \arg \max_{i=1 \dots N} f_i(\mathbf{x}_q; \mathbf{W}_i^*) \quad (2.51)$$

where \mathbf{W}_i^* depicts the i^{th} row of the weight matrix. The weight matrix \mathbf{W}^* might be found by minimizing the above loss function. However, optimizing this function using iterative gradient methods is a hard task. Based on the above equation, the sample \mathbf{x}^c belonging to class c is classified correctly if:

$$\forall j=1 \dots N \wedge j \neq i \quad \mathbf{W}_c \mathbf{x}_i > \mathbf{W}_j \mathbf{x}_i. \quad (2.52)$$

In other words, the score of the c^{th} model must be greater than all other models so \mathbf{x}^c is classified correctly. By rearranging the above equation, we will obtain:

$$\forall j=1 \dots N \wedge j \neq i \quad \mathbf{W}_j \mathbf{x}_i - \mathbf{W}_c \mathbf{x}_i \leq 0. \quad (2.53)$$

Assume that $\mathbf{W}_j \mathbf{x}_i$ is fixed. As $\mathbf{W}_c \mathbf{x}_i$ increases, their difference becomes more negative. In contrast, if the sample is classified incorrectly, their difference will be greater than zero. Consequently, if:

$$\max_{j=1 \dots N \wedge j \neq i} \mathbf{W}_j \mathbf{x}_i - \mathbf{W}_c \mathbf{x}_i \quad (2.54)$$

is negative, the sample is classified correctly. In contrary, if it is positive the sample is misclassified. In order to increase the stability of the models we can define the margin $\varepsilon \in \mathbb{R}^+$ and rewrite the above equation as follows:

$$H(\mathbf{x}_i) = \varepsilon + \max_{j=1 \dots N \wedge j \neq i} \mathbf{W}_j \mathbf{x}_i - \mathbf{W}_c \mathbf{x}_i. \quad (2.55)$$

The sample is classified correctly if $H(\mathbf{x}_i)$ is negative. The margin variable ε eliminates the samples which are very close to the model. Based on this equation, we can define the following loss function:

$$\mathcal{L}(\mathbf{W}) = \sum_{i=1}^n \max(0, \varepsilon + \max_{j \neq i} \mathbf{W}_j \mathbf{x}_i - \mathbf{W}_c \mathbf{x}_i). \quad (2.56)$$

This loss function is called *multiclass hinge* loss. If the sample is classified correctly and it is outside the critical region, $\varepsilon + \max_{j=1 \dots N \wedge j \neq i} \mathbf{W}_j \mathbf{x}_i - \mathbf{W}_c \mathbf{x}_i$ will be negative. Hence, output of $\max(0, -)$ will be zero indicating that we have not made a loss on \mathbf{x}_i using the current value for \mathbf{W} . Nonetheless, if the sample is classified in correctly or it is within the critical region $\varepsilon + \max_{j=1 \dots N \wedge j \neq i} \mathbf{W}_j \mathbf{x}_i - \mathbf{W}_c \mathbf{x}_i$ will be a positive number. As the result, $\max(0, +)$ will be positive indicating that we have made a loss on \mathbf{x}_i . By minimizing the above loss function, we will find \mathbf{W} such that the number misclassified samples is minimum.

The multiclass hinge loss function is a differentiable function. For this reason, gradient-based optimization methods such as gradient descend can be used to find the minimum of this function. To achieve this goal, we have to find the partial derivatives of the loss function with respect to each of the parameters in \mathbf{W} . Given a sample \mathbf{x}_i and its corresponding label y_i , partial derivatives of (2.56) with respect to $\mathbf{W}_{m,n}$ is calculated as follows:

$$\frac{\delta \mathcal{L}(\mathbf{W}; (\mathbf{x}_i, y_i))}{\delta \mathbf{W}_{m,n}} = \begin{cases} x_n & \varepsilon + \mathbf{W}_m \mathbf{x}_i - \mathbf{W}_{y_i} \mathbf{x}_i > 0 \text{ and } m = \arg \max_{p \neq y_i} \mathbf{W}_p \mathbf{x}_i - \mathbf{W}_{y_i} \mathbf{x}_i \\ -x_n & \varepsilon + \max_{p \neq m} \mathbf{W}_p \mathbf{x}_i - \mathbf{W}_m \mathbf{x}_i > 0 \text{ and } m = y_i \\ 0 & \text{otherwise} \end{cases} \quad (2.57)$$

$$\frac{\delta \mathcal{L}(\mathbf{W})}{\delta \mathbf{W}_{m,n}} = \sum_{i=1}^n \frac{\delta \mathcal{L}(\mathbf{W}; (\mathbf{x}_i, y_i))}{\delta \mathbf{W}_{m,n}} \quad (2.58)$$

In these equations, $\mathbf{W}_{m,n}$ depicts the n^{th} parameter of the m^{th} model. Similar to the binary hinge loss, ε can be set to 1. In this case, the magnitude of the models will be adjusted such that the loss function is minimum. If we plug the above partial

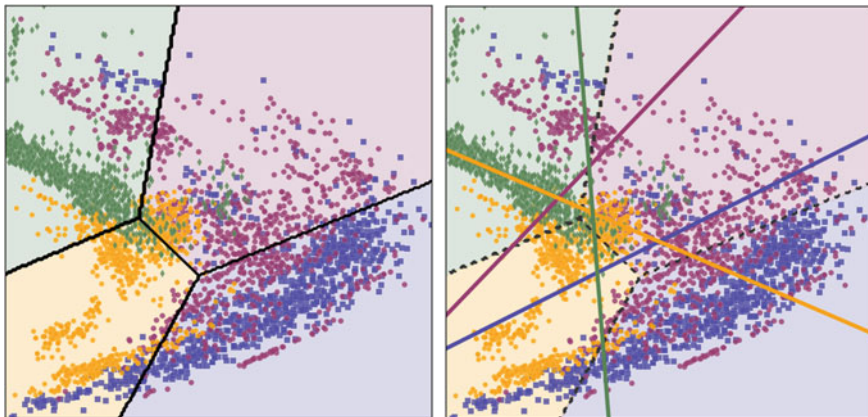


Fig. 2.23 A two-dimensional space divided into four regions using four linear models fitted using the multiclass hinge loss function. The plot on the *right* shows the linear models (lines in two-dimensional case) in the space

derivatives into the gradient descend method and apply it on the dataset illustrated in Fig. 2.20, we will obtain the result shown in Fig. 2.23.

The left plot in this figure shows how the two-dimensional space is divided into four distinct regions using the four linear models. The plot on the right also illustrates the four lines in this space. It should be noted that it is the maximum score of a sample from all the models that determined the class of the sample.

2.3.4 Multinomial Logistic Function

In the case of binary classification problems, we are able to model the probability of \mathbf{x} using the logistic function in (2.33). Then, a linear model can be found by maximizing the joint probability of training samples. Alternatively, we showed in (2.46) that we can minimize the negative of logarithm of probabilities to find a linear model for a binary classification problem.

It is possible to extend the logistic function into a multiclass classification problem. We saw before that N classes can be discriminated using N different lines. In addition, we showed how to model the posterior probability of input \mathbf{x} using logistic regression in (2.33). Instead of modeling $p(y = 1|\mathbf{x}; \mathbf{w})$, we can alternatively model $\ln p(y = 1|\mathbf{x}; \mathbf{w})$ given by:

$$\ln p(y = 1|\mathbf{x}; \mathbf{w}) = \mathbf{w}\mathbf{x}^T - \ln Z \quad (2.59)$$

where $\ln Z$ is a normalization factor. This model is called *log-linear* model. Using this formulation, we can model the posterior probability of N classes using N log-linear

models:

$$\begin{aligned}
 \ln p(y = 1|\mathbf{x}; \mathbf{w}_1) &= \mathbf{w}_1 \mathbf{x}^T - \ln Z \\
 \ln p(y = 2|\mathbf{x}; \mathbf{w}_2) &= \mathbf{w}_2 \mathbf{x}^T - \ln Z \\
 &\dots \\
 \ln p(y = N|\mathbf{x}; \mathbf{w}_N) &= \mathbf{w}_N \mathbf{x}^T - \ln Z
 \end{aligned} \tag{2.60}$$

If we compute the exponential of the above equations we will obtain:

$$\begin{aligned}
 p(y = 1|\mathbf{x}; \mathbf{w}_1) &= \frac{e^{\mathbf{w}_1 \mathbf{x}^T}}{Z} \\
 p(y = 2|\mathbf{x}; \mathbf{w}_2) &= \frac{e^{\mathbf{w}_2 \mathbf{x}^T}}{Z} \\
 &\dots \\
 p(y = N|\mathbf{x}; \mathbf{w}_N) &= \frac{e^{\mathbf{w}_N \mathbf{x}^T}}{Z}
 \end{aligned} \tag{2.61}$$

We know from probability theory that:

$$\sum_{c=1}^N p(y = c|\mathbf{x}; \mathbf{w}_1) = 1 \tag{2.62}$$

Using this property, we can find the normalization factor Z that satisfies the above condition. If we set:

$$\frac{e^{\mathbf{w}_1 \mathbf{x}^T}}{Z} + \frac{e^{\mathbf{w}_2 \mathbf{x}^T}}{Z} + \dots + \frac{e^{\mathbf{w}_N \mathbf{x}^T}}{Z} = 1 \tag{2.63}$$

as solve the above equation for Z , we will obtain:

$$Z = \sum_{i=1}^N e^{\mathbf{w}_i \mathbf{x}^T} \tag{2.64}$$

Using the above normalization factor and given the sample \mathbf{x}_i and its true class c , the posterior probability $p(y = c|\mathbf{x}_i)$ is computed by:

$$p(y = c|\mathbf{x}_i) = \frac{e^{\mathbf{w}_c \mathbf{x}_i^T}}{\sum_{j=1}^N e^{\mathbf{w}_j \mathbf{x}_i^T}} \tag{2.65}$$

where N is the number of classes. The denominator in the above equation is a normalization factor so $\sum_{c=1}^N p(y = c|\mathbf{x}_i) = 1$ holds true and, consequently, $p(y = c|\mathbf{x}_i)$ is a valid probability function. The above function which is called *softmax* function is commonly used to train convolutional neural networks. Given, a dataset

of d -dimensional samples \mathbf{x}_i with their corresponding labels $y_i \in \{1, \dots, N\}$ and assuming the independence relation between the samples (see Sect. 2.2.3), likelihood of all samples for a fixed \mathbf{W} can be written as follows:

$$p(\mathcal{X}) = \prod_{i=1}^n p(y = y_i | \mathbf{x}_i). \quad (2.66)$$

As before, instead of maximizing the likelihood, we can minimize the negative of log-likelihood that is defined as follows:

$$-\log(p(\mathcal{X})) = -\sum_{i=1}^n \log(p(y = y_i | \mathbf{x}_i)). \quad (2.67)$$

Note that the product operator has changed to the summation operator taking into account the fact that $\log(ab) = \log(a) + \log(b)$. Now, for any \mathbf{W} we can compute the following loss:

$$\mathcal{L}_{softmax}(\mathbf{W}) = -\sum_{i=1}^n \log(y_c) \quad (2.68)$$

where $\mathbf{W} \in \mathbb{R}^{N \times d+1}$ represents the parameters for N linear models and $y_c = p(y = y_i | \mathbf{x}_i)$. Before computing the partial derivatives of the above loss function, we explain how to show the above loss function using a *computational* graph. Assume computing $\log(y_c)$ for a sample. This can be represented using the graph in Fig. 2.24.

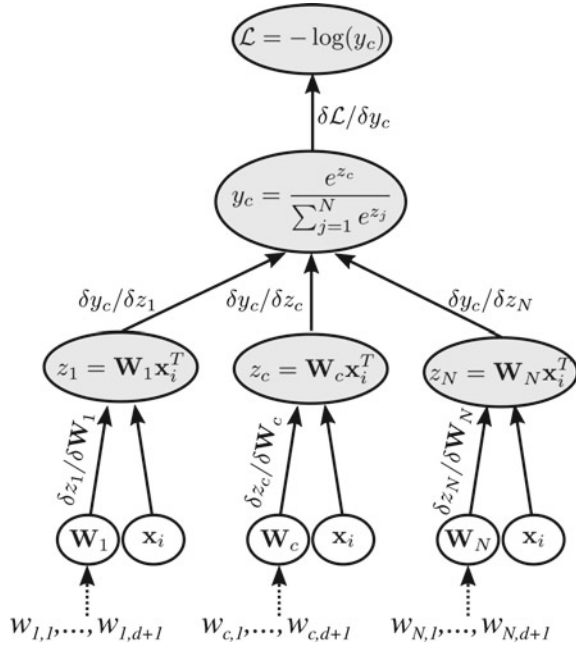
Computational graph is a directed acyclic graph where each non-leaf node in this graph shows a computational unit which accepts one or more inputs. Leaves also show the input of the graph. The computation starts from the leaves and follows the direction of the edges until it reaches to the final node. We can compute the gradient of each computational node with respect to its inputs. The labels next to each edge shows the gradient of its child node (top) with respect to its parent node (bottom). Assume, we want to compute $\delta \mathcal{L} / \delta \mathbf{W}_1$. To this end, we have to sum all the paths from \mathcal{L} to \mathbf{W}_1 and multiply the gradients represented by edges along each path. This result will be equivalent to *multivariate chain rule*. According to this, $\delta \mathcal{L} / \delta \mathbf{W}_1$ will be equal to:

$$\frac{\delta \mathcal{L}}{\delta \mathbf{W}_1} = \frac{\delta \mathcal{L}}{\delta y_c} \frac{\delta y_c}{\delta z_1} \frac{\delta z_1}{\delta \mathbf{W}_1}. \quad (2.69)$$

Using this concept, we can easily compute $\delta \mathcal{L} / \delta \mathbf{W}_{i,j}$ where $\mathbf{W}_{i,j}$ refers to the j^{th} parameter of u^{th} model. For this purpose, we need to compute $\frac{\delta y_c}{\delta z_i}$ which is done as follows:

$$\frac{\delta y_c}{\delta z_i} = \frac{\delta \frac{e^{z_c}}{\sum_{m=1}^N e^{z_m}}}{\delta z_i} = \begin{cases} \frac{e^{z_c} \sum_m e^{z_m} - e^{z_c} e^{z_c}}{(\sum_m e^{z_m})^2} = y_c(1 - y_c) & i = c \\ \frac{-e^{z_i} e^{z_c}}{(\sum_m e^{z_m})^2} = -y_i y_c & i \neq c \end{cases} \quad (2.70)$$

Fig. 2.24 Computational graph of the softmax loss on one sample



Now, we can compute $\delta \mathcal{L} / \delta \mathbf{W}_{i,j}$ by plugging the above derivative into the chain rule obtained by the computational graph for sample \mathbf{x} with label y_c .

$$\frac{\delta \mathcal{L}}{\delta \mathbf{W}_{i,j}} = \begin{cases} -(1 - y_c)x_j & i = c \\ y_i x_j & i \neq c \end{cases} \quad (2.71)$$

With this formulation, the gradient of all the samples will be equal to sum of the gradient of each sample. Now, it is possible to minimize the softmax loss function using the gradient descend method. Figure 2.25 shows how the two-dimensional space in our example is divided into four regions using the models trained by the softmax loss function. Comparing the results from one versus one, one versus all, the multiclass hinge loss and the softmax loss, we realize that their results are not identical. However, the two former techniques is not usually used for multiclass classification problems because of the reasons we mentioned earlier. Also, there is not a practical rule of thumb to tell if the multiclass hinge loss better or worse than the softmax loss function.

2.4 Feature Extraction

In practice, it is very likely that samples in the training set $\mathcal{X} = \{(\mathbf{x}_1, y_1), \dots, (\mathbf{x}_n, y_n)\}$ are not linearly separable. The multiclass dataset in the previous section is

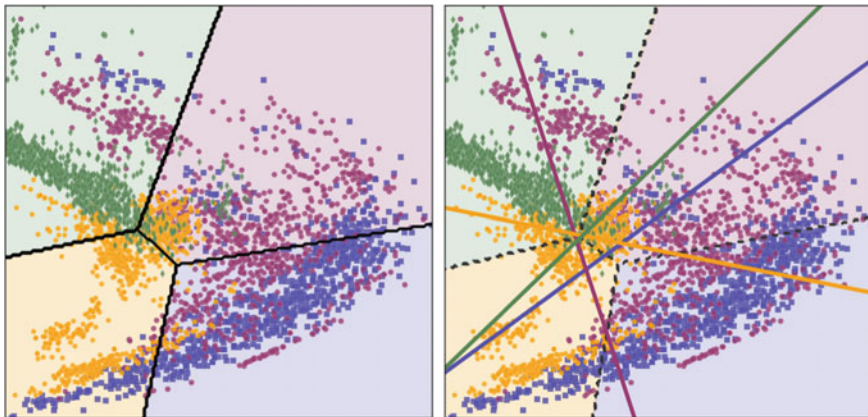


Fig.2.25 The two-dimensional space divided into four regions using four linear models fitted using the softmax loss function. The plot on the *right* shows the linear models (lines in two-dimensional case) in the space

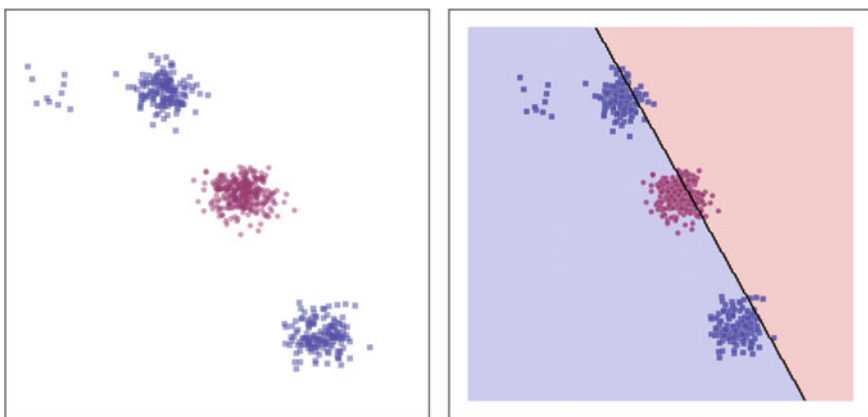


Fig.2.26 A linear classifier is not able to accurately discriminate the samples in a nonlinear dataset

an example of such a dataset. Figure 2.26 shows a nonlinear dataset and the linear classifier fitted using logistic regression. Samples of each class are illustrated using a different marker and different color.

Clearly, it is impossible to perfectly discriminate these two classes using a line. There are mainly two solutions for solving this problem. The first solution is to train a nonlinear classifier such as random forest on the training dataset. This method is not within the scope of this book. The second method is to project the original data into another space using the transformation function $\Phi : \mathbb{R}^d \rightarrow \mathbb{R}^{\hat{d}}$ where classes are linearly separable in the transformed space. Here, \hat{d} can be any arbitrary integer number. Formally, given the sample $\mathbf{x} \in \mathbb{R}^d$, it is transformed into a \hat{d} -dimensional

space using:

$$\Phi(\mathbf{x}) = \hat{\mathbf{x}} = \begin{bmatrix} \phi_1(\mathbf{x}) \\ \phi_2(\mathbf{x}) \\ \vdots \\ \phi_d(\mathbf{x}) \end{bmatrix} \quad (2.72)$$

where $\phi_i : \mathbb{R}^d \rightarrow \mathbb{R}$ is a scalar function which accepts a d-dimensional input and return a scalar. Also, ϕ_i can be any function. Sometimes, an expert can design these functions based on the requirements of the problem. To transform the above nonlinear dataset, we define $\Phi(\mathbf{x})$ as follows:

$$\Phi(\mathbf{x}) = \hat{\mathbf{x}} = \begin{bmatrix} \phi_1(\mathbf{x}) = e^{-10\|\mathbf{x}-c_1\|^2} \\ \phi_2(\mathbf{x}) = e^{-20\|\mathbf{x}-c_2\|^2} \end{bmatrix} \quad (2.73)$$

where $c_1 = (0.56, 0.67)$ and $c_2 = (0.19, 0.11)$. By applying this function on each sample, we will obtain a new two-dimensional space where the samples are non-linearly transformed. Figure 2.27 shows how samples are projected into the new two-dimensional space. It is clear that the samples in the new space become linearly separable. In other words, the dataset $\hat{\mathcal{X}} = \{(\Phi(\mathbf{x}_1), y_1), \dots, (\Phi(\mathbf{x}_n), y_n)\}$ is linearly separable. Consequently, the samples in $\hat{\mathcal{X}}$ can be classified using a linear classifier in the previous section. Figure 2.28 shows a linear classifier fitted on the data in the new space.

The decision boundary of a linear classifier is a hyperplane (a line in this example). However, because $\Phi(\mathbf{x})$ is a nonlinear transformation, if we apply the inverse transform from the new space to the original space, the decision boundary will not be a hyperplane anymore. Instead, it will be a nonlinear decision boundary. This is illustrated in the right plot of Fig. 2.28.

Choice of $\Phi(\mathbf{x})$ is the most important step in transforming samples into a new space where they are linearly separable. In the case of high-dimensional vectors such as images, finding an appropriate $\Phi(\mathbf{x})$ becomes even harder. In some case, $\Phi(\mathbf{x})$ might be composition of multiple functions. For example, one can define $\Phi(\mathbf{x}) = \Psi(\Omega(\Gamma(\mathbf{x})))$ where $\Phi : \mathbb{R}^d \rightarrow \mathbb{R}^{\hat{d}}$, $\Psi : \mathbb{R}^{d_2} \rightarrow \mathbb{R}^{\hat{d}}$, $\Omega : \mathbb{R}^{d_1} \rightarrow \mathbb{R}^{d_2}$ and, $\Gamma : \mathbb{R}^d \rightarrow \mathbb{R}^{d_1}$. In practice, there might be infinite number of functions to make samples linearly separable.

Let us apply our discussions so far on a real world problem. Suppose the 43 classes of traffic signs shown in Fig. 2.29 that are obtained from the German traffic sign recognition benchmark (GTSRB) dataset. For the purpose of this example, we randomly picked 1500 images for each class. Assume a 50×50 RGB image. Taking into account the fact that each pixel in this image is represented by a three-dimensional vector, the flattened image will be a $50 \times 50 \times 3 = 7500$ dimensional vector. Therefore, the *training dataset* \mathcal{X} is composed of 1500 training sample pair (\mathbf{x}_i, y_i) where $\mathbf{x} \in \mathbb{R}^{7500}$ and $y_i \in \{0, \dots, 42\}$.

Beside the training dataset, we also randomly pick 6400 *test samples* $(\hat{\mathbf{x}}, \hat{y}_i)$ from the dataset that are not included in \mathcal{X} . Formally, we have another dataset $\hat{\mathcal{X}}$ of

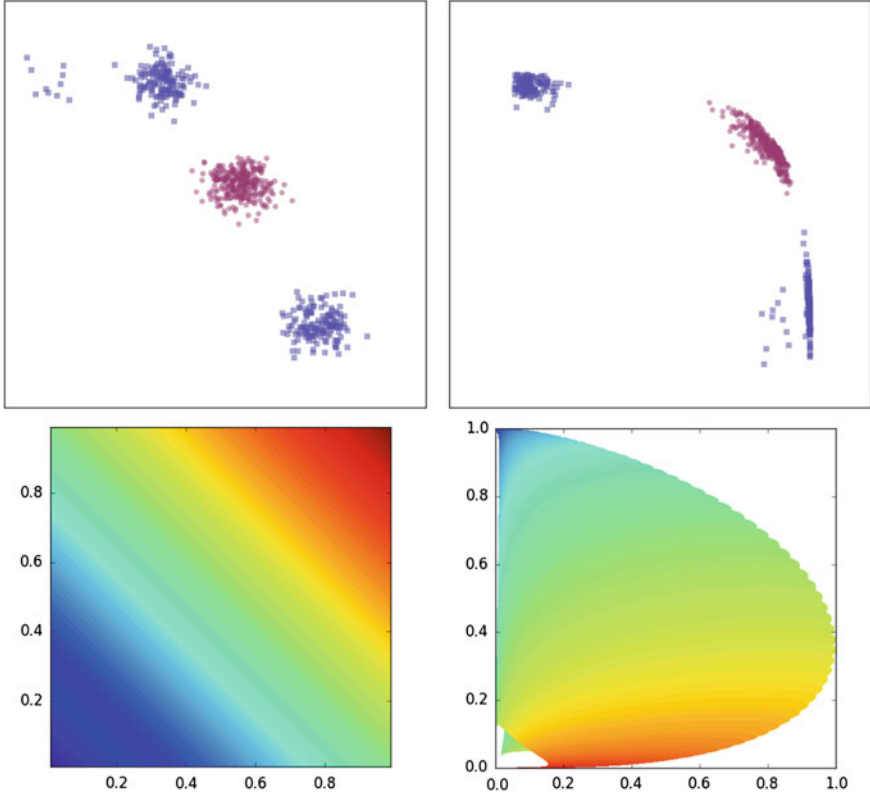


Fig.2.27 Transforming samples from the original space (*left*) into another space (*right*) by applying $\Phi(\mathbf{x})$ on each sample. The *bottom colormaps* show how the original space is transformed using this function

traffic signs where $\hat{\mathbf{x}} \in \mathbb{R}^{7500}$ and $\hat{\mathbf{x}} \notin \mathcal{X}$ and $\hat{y}_i \in \{0, \dots, 42\}$. It is very important in testing a model to use unseen samples. We will explain this topic thoroughly in the next chapters. Finally, we can train a linear classifier $F(\mathbf{x})$ using \mathcal{X} to discriminate the 43 classes of traffic signs. Then, $F(\mathbf{x})$ can be tested using \mathcal{X} and computing *classification accuracy*.

To be more specific, we pick every sample $\hat{\mathbf{x}}_i$ and predict its class label using $F(\hat{\mathbf{x}}_i)$. Recall from previous sections that for a softmax model with 43 liner models, the class of sample $\hat{\mathbf{x}}_i$ is computed using $F(\hat{\mathbf{x}}_i) = \arg \max_{i=1 \dots 43} f_i(\hat{\mathbf{x}}_i)$ where $f_i(\hat{\mathbf{x}}_i) = \mathbf{w}\hat{\mathbf{x}}_i$ is the score computed by the i^{th} model. With this formulation, the classification accuracy of the test samples is obtained by computing:

$$acc = \frac{1}{6400} \sum_{i=1}^{6400} \mathbf{1}[F(\hat{\mathbf{x}}_i) == \hat{y}_i] \quad (2.74)$$

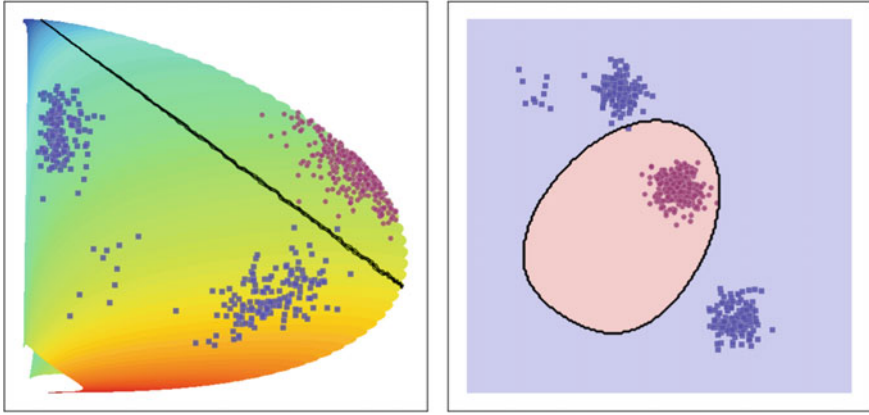


Fig. 2.28 Samples become linearly separable in the new space. As the result, a linear classifier is able to accurately discriminate these samples. If we transform the linear model from the new space into the original space, the linear decision boundary become a nonlinear boundary



Fig. 2.29 43 classes of traffic in obtained from the GTSRB dataset (Stallkamp et al. 2012)

where $\mathbf{1}[\cdot]$ is the indicator function and it returns 1 when the input is true. The quantity acc is equal to 1 when all the samples are classified correctly and it is equal to 0 when all of them are misclassified. We trained a linear model on this dataset using the raw pixel values. The accuracy on the test set is equal to 73.17%. If we ignore the intercept, the parameters vector $\mathbf{w} \in \mathbb{R}^{7500}$ of the linear model $f(x) = \mathbf{w}\mathbf{x}^T$ has the same dimension as the input image. One way to visualize and study the parameter vector is to reshape \mathbf{w} into a $50 \times 50 \times 3$ image. Then, we can plot each channel in this three-dimensional array using a colormap plot. Figure 2.30 shows weights of the model related to Class 1 after reshaping.

We can analyze this figure to see what a linear model trained on raw pixel intensities exactly learns. Consider the linear model $f(\mathbf{x}) = w_1x_1 + \dots + w_nx_n$ without the intercept term. Taking into account the fact that pixel intensities in a regular RGB image are positive values, x_i in this equation is always a positive value. Therefore, $f(\mathbf{x})$ will return a higher value if w_i is a high positive number. In contrary, $f(\mathbf{x})$ will return a smaller value if w_i is a very small negative number. From another perspective, we can interpret positive weights as “likes” and negative weights as “dislikes” of the linear model.

That being said if w_i is negative, the model does not like high values of x_i . Hence, if the intensity of pixel at x_i is higher than zero it will reduce the classification score.

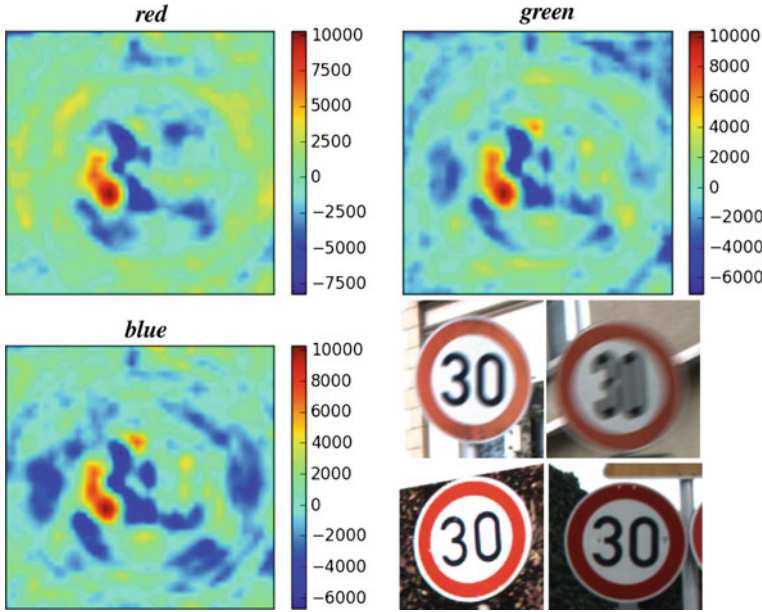


Fig. 2.30 Weights of a linear model trained directly on raw pixel intensities can be visualized by reshaping the vectors so they have the same shape as the input image. Then, each channel of the reshaped matrix can be shown using a colormap

In contrast, if w_i is positive, the model likes high values of x_i . In other words, as the intensity of x_i increases, the model becomes more confident about the classification since it increases the classification score.

Looking at this figure, we see a red region in the middle of red, green and blue channels. According to the color map next to each plot, red regions correspond to weights with high positive values. Since, the same region is red in all three channels, we can imply that the model likes to see the white color in that specific region. Then, we observe that the region analogous to the rim of the sign has high positive weight in the red channel and small negative weights in the blue channel. Also, the weights of the green channel for that region is close to zero. This means that the model likes to see high red values in that region and it dislikes blue values in that region. This choice made by the model also seems rational for a human expert. This argument can be applied on the other classes of traffic signs, as well.

Remember that the accuracy of the model trained on raw pixel intensities was equal to 73.17%. Now, the question is why the accuracy of the model is very low? To answer this question, we start with a basic concept. A two-dimensional vector (x_1, x_2) can be illustrated using a *point* in a two-dimensional space. Moreover, a three-dimensional vector (x_1, x_2, x_3) can be shown using a point in a three-dimensional space. Similarly, a d-dimensional vector (x_1, \dots, x_d) is a point in a d-dimensional space. It is trivial for a human to imagine the points in two-dimensional and three-dimensional spaces.

But, it might be difficult at first to imagine higher dimensions. For starting, it suffice to know that a d -dimensional vector is a point in a d -dimensional space.

Each RGB image in the above example will be a point in a 7500-dimensional space. We can study the above question in this space. There are mainly two possibilities that reduces the accuracy of a linear model in this space defined by raw images. First, like the dataset in Fig. 2.26 the classes of traffic signs might be completely disjoint but they might not be linearly separable. Second, similar to the dataset in Fig. 2.20, the classes might have overlap with each other. The latter problem is commonly known as *interclass similarity* meaning that samples of two or more classes are similar. In both cases, a linear model is not able to accurately discriminate the classes.

Although there might not be a quick remedy to the second problem, the first problem might be addressed by transforming the raw vectors into another space using the feature transformation function $\Phi(\mathbf{x})$. Knowing the fact that output of $\Phi(\mathbf{x})$ is a \hat{d} -dimensional vector, the question in designing $\Phi(\mathbf{x})$ is what should be the value of \hat{d} ? Even if we found a way to determine the value of \hat{d} , the next question is what should be the transformation function $\phi_i(\mathbf{x})$, $i = 1, \dots, \hat{d}$? There are infinite ways to define this function. For this reason, it is not trivial in practice to define $\Phi(\mathbf{x})$ for an image (it might not be a tedious task for other modalities with low dimensions).

To alleviate this problem, researchers came up with the idea of *feature extraction* algorithms. In general, a feature extraction algorithm processes an image and generates a more informative vector which better separates classes. Notwithstanding, a feature extraction algorithm does not guarantee that the classes will be linearly separable. Despite this, in most cases, a feature extraction is applied on an image before feeding it to a classifier. In other words, we do not classify images using raw pixel values. Instead, we always extract their feature and train a classifier on top of the feature vectors.

One of the widely used feature extraction algorithms is called *histogram of oriented gradients* (HOG). It starts by applying the gamma correction transformation on the image and computing its first derivatives. Then, the image is divided into small patches called cells. Within each cell, a histogram is computed based on the orientation of the gradient vector and its magnitude using the pixels inside that cell. Then, blocks are formed by considering neighbor cells and the histogram of the cells within that block are concatenated. Finally, the feature vector is obtained by concatenating the vectors of all blocks. The whole process of this algorithm can be easily represented in terms of mathematical equations.

Assume that $\Phi_{hog}(\mathbf{x}) : \mathbb{R}^d \rightarrow \mathbb{R}^{d_{hog}}$ denotes the HOG features. We can now apply $\Phi_{hog}(\mathbf{x})$ on each sample of the training set \mathcal{X} in order to obtain $\hat{\mathcal{X}} = \{(\Phi_{hog}(\mathbf{x}_1), y_1), \dots, (\Phi_{hog}(\mathbf{x}_n), y_n)\}$. Then, a linear classifier is trained using $\hat{\mathcal{X}}$. By doing this, the accuracy of the classification increases to 88.90%. Comparing with the accuracy of the classifier trained on raw pixel intensities (i.e., 73.17%), the accuracy increases 15.73%.

There might different reasons that the accuracy is not still very high. First, the feature extraction function $\Phi_{hog}(\mathbf{x})$ might not be able to perfectly make the classes linearly separable. This could be due to the fact that there are traffic signs such as

“left bend ahead” and “right bend ahead” with slight differences. The utilized feature extraction function might not be able to effectively model these differences such that these classes become linearly separable. Second, the function $\Phi_{hog}(\mathbf{x})$ may cause some of the classes to have overlap with other classes. Both or one of these reasons can be responsible for having a low accuracy.

Like before, it is possible to create another function whose input is $\Phi_{hog}(\mathbf{x})$ and its output is a \hat{d} dimensional vector. For example, we can define the following function:

$$\Phi(\Phi_{hog}(\mathbf{x})) = \begin{bmatrix} \phi_1(\Phi_{hog}(\mathbf{x})) \\ \phi_2(\Phi_{hog}(\mathbf{x})) \\ \vdots \\ \phi_{\hat{d}}(\Phi_{hog}(\mathbf{x})) \end{bmatrix} = \begin{bmatrix} e^{-\gamma \|\Phi_{hog}(\mathbf{x}) - \mathbf{c}_1\|^2} \\ e^{-\gamma \|\Phi_{hog}(\mathbf{x}) - \mathbf{c}_2\|^2} \\ \vdots \\ e^{-\gamma \|\Phi_{hog}(\mathbf{x}) - \mathbf{c}_{\hat{d}}\|^2} \end{bmatrix} \quad (2.75)$$

where $\gamma \in \mathbb{R}$ is a scaling constant and $\mathbf{c}_i \in \mathbb{R}^{d_{hog}}$ is parameters which can be defined manually or automatically. Doing so, we can generate a new dataset $\hat{\mathcal{X}} = \{\Phi(\Phi_{hog}(\mathbf{x}_1)), y_1), \dots, (\Phi(\Phi_{hog}(\mathbf{x}_n)), y_n)\}$ and train a linear classifier on top of this dataset. This increases the accuracy from 88.90 to 92.34%. Although the accuracy is higher it is not still high enough to be used in practical applications. One may add another feature transformation whose input is $\Phi(\Phi_{hog}(\mathbf{x}))$. In fact, compositing the transformation function can be done several times. But, this does not guarantee that the classes are going to be linearly separable. Some of the transformation function may increase the interclass overlap causing a drop in accuracy.

As it turns out, the key to accurate classification is to have a feature transformation function $\Phi(\mathbf{x})$ which is able to make the classes linearly separable without causing interclass overlap. But, how can we find $\Phi(\mathbf{x})$ which satisfies both these conditions? We saw in this chapter that a classifier can be directly trained on the training dataset. It might be also possible to learn $\Phi(\mathbf{x})$ using the same training dataset. If $\Phi(\mathbf{x})$ is designed by a human expert (such as the HOG features), it is called a *hand-crafted* or *hand-engineered* feature function.

2.5 Learning $\Phi(\mathbf{x})$

Despite the fairly accurate results obtained by hand-crafted features on some datasets, as we will show in the next chapters, the best results have been achieved by learning $\Phi(\mathbf{x})$ from a training set. In the previous section, we designed a feature function to make the classes in Fig. 2.26 linearly separable. However, designing that feature function by hand was a tedious task and needed many trials. Note that, the dataset shown in that figure was composed of two-dimensional vectors. Considering the fact that a dataset may contain high-dimensional vectors in real-world applications, designing an accurate feature transformation function $\Phi(\mathbf{x})$ becomes even harder.

For this reason, in many cases the better approach is to learn $\Phi(\mathbf{x})$ from data. More specifically, $\Phi(\mathbf{x}; \mathbf{w}_\phi)$ is formulated using the parameter vector \mathbf{w}_ϕ . Then, the

linear classifier for i^{th} class is defined as:

$$f_i(\mathbf{x}) = \mathbf{w}\Phi(\mathbf{x}; \mathbf{w}_\phi)^T \quad (2.76)$$

where $\mathbf{w} \in \mathbb{R}^{\hat{d}}$ and \mathbf{w}_ϕ are parameter vectors that are found using training data. Depending on the formulation of $\Phi(\mathbf{x})$, \mathbf{w}_ϕ can be any vector with arbitrary size. The parameter vector \mathbf{w} and \mathbf{w}_ϕ determine the weights for the linear classifier and the transformation function, respectively. The ultimate goal in a classification problem is to *jointly* learn this parameter vectors such that the classification accuracy is high.

This goal is exactly the same as learning \mathbf{w} such that $\mathbf{w}\mathbf{x}^T$ accurately classifies the samples. Therefore, we can use the same loss functions in order to train both parameter vectors in (2.76). Assume that $\Phi(\mathbf{x}; \mathbf{w}_\phi)$ is defined as follows:

$$\Phi(\mathbf{x}; \mathbf{w}_\phi) = \begin{bmatrix} \ln(1 + e^{(w_{11}x_1 + w_{21}x_2 + w_{01})}) \\ \ln(1 + e^{(w_{12}x_1 + w_{22}x_2 + w_{02})}) \end{bmatrix} \quad (2.77)$$

In the above equation $\mathbf{w}_\phi = \{w_{11}, w_{21}, w_{01}, w_{12}, w_{22}, w_{02}\}$ is the parameter vector for the feature transformation function. Knowing the fact that the dataset in Fig. 2.26 is composed of two classes, we can minimize the binary logistic loss function for jointly finding \mathbf{w} and \mathbf{w}_ϕ . Formally, the loss function is defined as follows:

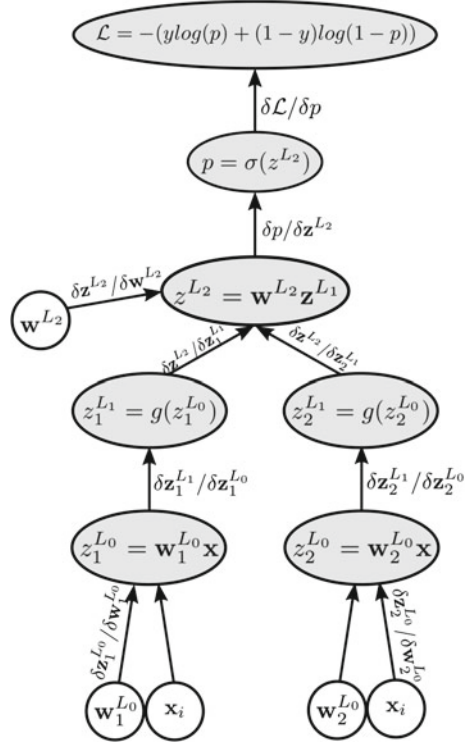
$$\mathcal{L}(\mathbf{w}, \mathbf{w}_\phi) = - \sum_{i=1}^n y_i \log(\sigma(\mathbf{w}\Phi(\mathbf{x})^T)) + (1 - y_i)(1 - \log(\sigma(\mathbf{w}\Phi(\mathbf{x})^T))) \quad (2.78)$$

The intuitive way to understanding the above loss function and computing its gradient is to build its computational graph. This is illustrated in Fig. 2.31. In the graph, $g(z) = \ln(1 + e^z)$ is a nonlinear function which is responsible for nonlinearly transforming the space. First, the dot product of the input vector \mathbf{x} is computed with two weight vectors $\mathbf{w}_1^{L_0}$ and $\mathbf{w}_2^{L_0}$ in order to obtain $z_1^{L_0}$ and $z_2^{L_0}$, respectively. Then, each of these values is passed through a nonlinear function and their dot product with \mathbf{w}^{L_2} is calculated. Finally, this score is passed through a sigmoid function and the loss is computed in the final node. In order to minimize the loss function (i.e., the top node in the graph), the gradient of the loss function has to be computed with respect to the nodes indicated by \mathbf{w} in the figure. This can be done using the chain rule or derivatives. To this end, gradient of each node with respect to its parent must be computed. Then, for example, to compute $\delta\mathcal{L}/\delta\mathbf{w}_1^{L_0}$, we have to sum all the paths from $\mathbf{w}_1^{L_0}$ to \mathcal{L} and multiply the term along each path. Since there is only one path from $\mathbf{w}_1^{L_0}$ in this graph, the gradient will be equal to:

$$\frac{\delta\mathcal{L}}{\delta\mathbf{w}_1^{L_0}} = \frac{\delta\mathbf{z}_1^{L_0}}{\delta\mathbf{w}_1^{L_0}} \frac{\delta\mathbf{z}_1^{L_1}}{\delta\mathbf{z}_1^{L_0}} \frac{\delta\mathbf{z}^{L_2}}{\delta\mathbf{z}_1^{L_1}} \frac{\delta p}{\delta\mathbf{z}^{L_2}} \frac{\delta\mathcal{L}}{\delta p} \quad (2.79)$$

The gradient of the loss with respect to the other parameters can be obtained in a similar way. After that, we should only plug the gradient vector in the gradient descend

Fig. 2.31 Computational graph for (2.78). Gradient of each node with respect to its parent is shown on the edges



method and minimize the loss function. Figure 2.32 illustrates how the system eventually learns to transform and classify the samples. According to the plots in the second and third rows, the model is able to find a transformation where the classes become linearly separable. Then, classification of the samples is done in this space. This means that the decision boundary in the transformed space is a hyperplane. If we apply the inverse transform from the feature space to the original space, the hyperplane is not longer a line. Instead, it is a nonlinear boundary which accurately discriminates the classes.

In this example, the nonlinear transformation function that we used in (2.77) is called the *softplus* function and it is defined as $g(x) = \ln(1 + e^x)$. The derivative of this function is also equal to $g'(x) = \frac{1}{1+e^{-x}}$. The softplus function can be replaced with another function whose input is a scalar and its output is a real number. Also, we there are many other ways to define a transformation function and find its parameters by minimizing the loss function.

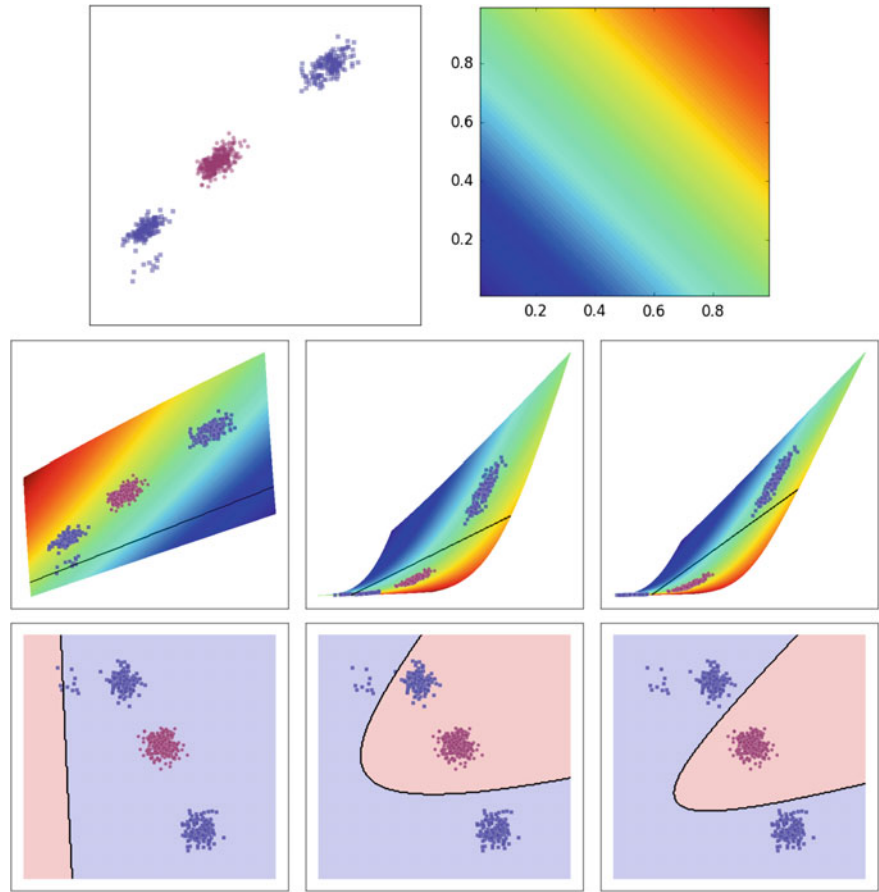


Fig. 2.32 By minimizing (2.78) the model learns to jointly transform and classify the vectors. The *first row* shows the distribution of the training samples in the two-dimensional space. The *second* and *third rows* show the status of the model in three different iterations starting from the left plots

2.6 Artificial Neural Networks

The idea of learning a feature transformation function instead of designing it by hand is very useful and it produces very accurate results in practice. However, as we pointed out above, there are infinite ways to design a trainable feature transformation function. But, not all of them might be able to make the classes linearly separable in the feature space. As the result, there might be a more general way to design a trainable feature transformation functions.

An artificial neural network (ANN) is an interconnected group of smaller computational units called neurons and it tries to mimic biological neural networks. Detailed discussion about biological neurons is not within the scope of this book. But, in order

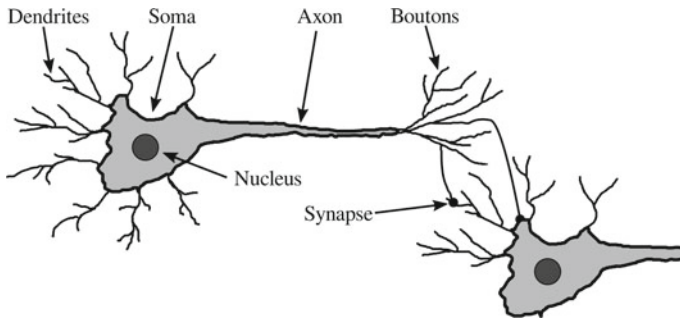


Fig. 2.33 Simplified diagram of a biological neuron

to better understand an artificial neuron we explain how a biological neuron works in general. Figure 2.33 illustrates a simplified diagram of a biological neuron.

A neuron is mainly composed of four parts including dendrites, soma, axon, nucleus and boutons. Boutons is also called axon terminals. Dendrites act as the input of the neuron. They are connected either to a sensory input (such as eye) or other neurons through synapses. Soma collects the inputs from dendrites. When the inputs pass a certain threshold it fires series of spikes across the axon. As the signal is fired, the nucleus returns to its stationary state. When it reaches to this state, the firing stops. The fired signals are transmitted to other neuron through boutons. Finally, synaptic connections transmits the signals from one neuron to another.

Depending on the synaptic strengths and the signal at one axon terminal, each dendron (i.e., one branch of dendrites) increases or decreases the potential of nucleus. Also, the direction of the signal is always from axon terminals to dendrites. That means, it is impossible to pass a signal from dendrites to axon terminals. In other words, the path from one neuron to another is always a one-way path. It is worth mentioning that each neuron might be connected to thousands of other neurons. Mathematically, a biological neuron can be formulated as follows:

$$f(\mathbf{x}) = \mathcal{G}(\mathbf{w}\mathbf{x}^T + b). \quad (2.80)$$

In this equation, $\mathbf{w} \in \mathbb{R}^d$ is the weight vector, $\mathbf{x} \in \mathbb{R}^d$ is the input and $b \in \mathbb{R}$ is the intercept term which is also called *bias*. Basically, an artificial neuron computes the weighted sum of inputs. This mimics the soma in biological neuron. The synaptic strength is modeled using \mathbf{w} and inputs from other neurons or sensors are modeled using \mathbf{x} . In addition $\mathcal{G}(x) : \mathbb{R} \rightarrow \mathbb{R}$ is a *nonlinear* function which is called *activation function*. It accepts a real number and returns another real number after applying a nonlinear transformation on it. The activation function act as the threshold function in biological neuron. Depending on the potential of nucleus (i.e., $\mathbf{w}\mathbf{x}^T + b$), the activation function returns a real number. From computational graph perspective, a neuron is a node in the graph with the diagram illustrated in Fig. 2.34.

An artificial neural network is created by connecting one or more neurons to the input. Each pair of neurons may or may not have a connection between them. With

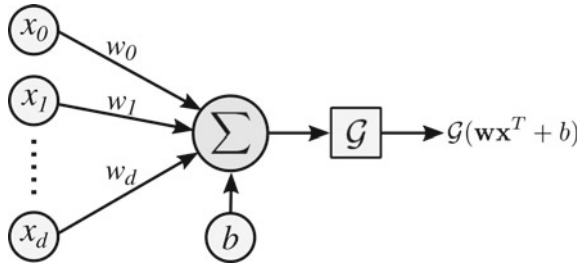


Fig. 2.34 Diagram of an artificial neuron

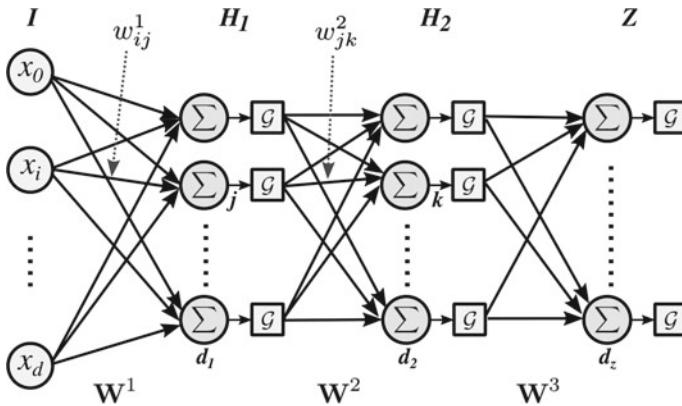


Fig. 2.35 A feedforward neural network can be seen as a directed acyclic graph where the inputs are passed through different layer until it reaches to the end

this formulation, the logistic regression model can be formulated using only one neuron where $\mathcal{G}(x)$ is the sigmoid function in (2.33). Depending on how neurons are connected, a network act differently. Among various kinds of artificial neural networks *feedforward neural network* (FNN) and *recurrent neural network* (RNN) are commonly used in computer vision community.

The main difference between these two kinds of neural networks lies in the connection between their neurons. More specifically, in a feedforward neural network the connections between neurons do not form a cycle. In contrast, in recurrent neural networks connection between neurons form a directed cycle. Convolutional neural networks are a specific type of feedforward networks. For this reason, in the remaining of this section we will only focus on feedforward networks. Figure 2.35 shows general architecture of feedforward neural networks.

A feedforward neural network includes one or more layers in which each layer contains one or more neurons. Also, number of neurons in one layer can be different from another layer. The network in the figure has one input layer and three layers with computational neurons. Any layer between the input layer and the last layer is called a *hidden layer*. The last layer is also called the *output layer*. In this chapter,

the input layer is denoted by I and hidden layers are denoted by H_i where i starts from 1. Moreover, the output layer is denoted by Z . In this figure, the first hidden layer has d_1 neurons and the second hidden layer has d_2 neurons. Also, the output layer has d_z neurons.

It should be noted that every neuron in a hidden layer or the output layer is connected to all the neurons in the previous layer. That said, there is $d_1 \times d_2$ connections between H_1 and H_2 in this figure. The connection from the i^{th} input in the input layer to the j^{th} neuron in H_1 is denoted by w_{ij}^1 . Likewise, the connection from the j^{th} neuron in H_1 to the k^{th} neuron in H_2 is denoted by w_{jk}^2 . With this formulation, the weights connecting the input layer to H_1 can be represented using $\mathbf{W}_1 \in \mathbb{R}^{d \times d_1}$ where $W(i, j)$ shows the connection from the i^{th} input to the j^{th} neuron.

Finally, the activation function \mathcal{G} of each neuron can be different from all other neurons. However, all the neuron in the same layer usually have the same activation function. Note that we have removed the bias connection in this figure to cut the clutter. However, each neuron in all the layers is also have a bias term beside its weights. The bias term in H_1 is represented by $\mathbf{b}^1 \in \mathbb{R}^{d_1}$. Similarly, the bias of h^{th} layer is represented by \mathbf{b}^h . Using this notations, the network illustrated in this figure can be formulated as:

$$f(\mathbf{x}) = \mathcal{G} \left(\mathcal{G} \left(\mathcal{G}(\mathbf{x}\mathbf{W}^1 + \mathbf{b}^1)\mathbf{W}^2 + \mathbf{b}^2 \right) \mathbf{W}^3 + \mathbf{b}^3 \right). \quad (2.81)$$

In terms of feature transformation, the hidden layers act as a feature transformation function which is a composite function. Then, the output layer act as the linear classifier. In other words, the input vector \mathbf{x} is transformed into a d_1 -dimensional space using the first hidden layer. Then, the transformed vectors are transformed into a d_2 -dimensional space using the second hidden layer. Finally, the output layer classifies the transformed d_2 -dimensional vectors.

What makes a feedforward neural network very special is the fact the a feedforward network with one layer and finite number of neurons is a universal approximator. In other words, a feedforward network with one hidden layer can approximate any continuous function. This is an important property in classification problems.

Assume a multiclass classification problem where the classes are not linearly separable. Hence, we must find a transformation function which makes the classes linearly separable in the feature space. Suppose that $\Phi_{ideal}(\mathbf{x})$ is a transformation function which is able to perfectly do this job. From function perspective, $\Phi_{ideal}(\mathbf{x})$ is a vector-valued continuous function. Since a feedforward neural network is a universal approximator, it is possible to design a feedforward neural network which is able to accurately approximate $\Phi_{ideal}(\mathbf{x})$. However, the beauty of feedforward networks is that we do not need to design a function. We only need to determine the number of hidden layers, number of neurons in each layer, and the type of activation functions. These are called *hyperparameters*. Among them, the first two hyperparameters is much more important than the third hyperparameter.

This implies that we do not need to design the equation of the feature transformation function by hand. Instead, we can just train a multilayer feedforward network to do both feature transformation and classification. Nonetheless, as we will see

shortly, computing the gradient of loss function on a feedforward neural network using multivariate chain rule is not tractable. Fortunately, gradient of loss function can be computed using a method called *backpropagation*.

2.6.1 Backpropagation

Assume a feedforward network with a two-dimensional input layer and two hidden layers. The first hidden layer consists of four neurons and the second hidden layer consists of three neurons. Also, the output layer has three neurons. According to number of neurons in the output layer, the network is a 3-class classifier. Like multiclass logistic regression, the loss of the network is computed using a softmax function.

Also, the activation functions of the hidden layers could be any nonlinear function. But, the activation function of the output layer is the identity function $\mathcal{G}_i^3(x) = x$. The reason is that the output layer calculates the classification scores which is obtained by only computing $\mathbf{w}\mathcal{G}^2$. The classification score must be passed to the softmax function without any modifications in order to compute the multiclass logistic loss. For this reason, in practice, the activation function of the output layer is the identity function. This means that, we can ignore the activation function in the output layer. Similar to any compositional computation, a feedforward network can be illustrated using a computational graph. The computational graph analogous to this network is illustrated in Fig. 2.36.

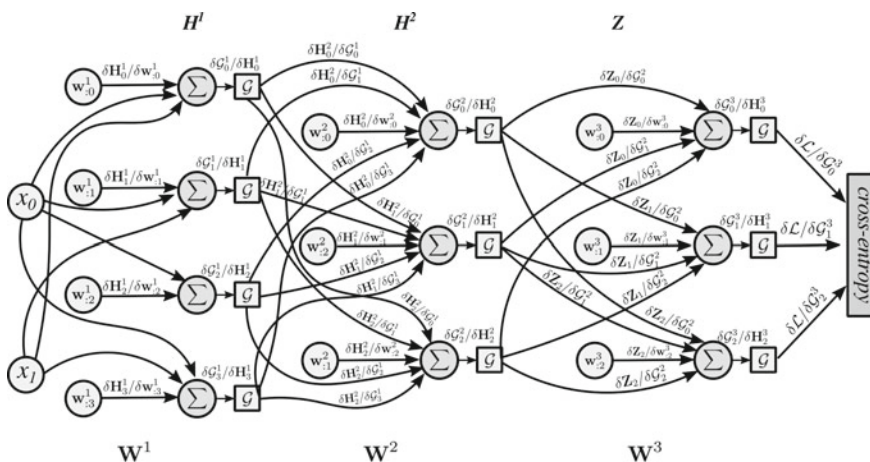


Fig. 2.36 Computational graph corresponding to a feedforward network for classification of three classes. The network accepts two-dimensional inputs and it has two hidden layers. The hidden layers consist of four and three neurons, respectively. Each neuron has two inputs including the weights and inputs from previous layer. The derivative of each node with respect to each input is shown on the edges

Each computational node related to function of soma (the computation before applying the activation function) accepts two inputs including weights and output of the previous layer. Gradient of each node with respect to its inputs is indicated on the edges. Also note that \mathbf{w}_b^a is a vector whose length is equal to the number of outputs from layer $a - 1$. Computing $\frac{\delta \mathcal{L}}{\delta \mathbf{w}_b^a}$ is straightforward and it is explained on Fig. 2.24.

Assume, we want to compute $\frac{\delta \mathcal{L}}{\delta \mathbf{w}_0^1}$.

According to the multivariate chain rule, this is equal to adding all paths starting from \mathbf{w}_0^1 and ending at \mathcal{L} in which the gradients along each path is multiplied. Based on this definition, $\frac{\delta \mathcal{L}}{\delta \mathbf{w}_0^1}$ will be equal to:

$$\begin{aligned}
 \frac{\delta \mathcal{L}}{\delta \mathbf{w}_0^1} = & \frac{\delta \mathbf{H}_0^1}{\delta \mathbf{w}_0^1} \frac{\mathcal{G}_0^1}{\delta \mathbf{H}_0^1} \frac{\delta \mathbf{H}_0^2}{\delta \mathcal{G}_0^1} \frac{\mathcal{G}_0^2}{\delta \mathbf{H}_0^2} \frac{\delta \mathbf{Z}_0}{\delta \mathcal{G}_0^2} \frac{\mathcal{G}_0^3}{\delta \mathbf{H}_0^3} \frac{\mathcal{L}}{\delta \mathcal{G}_0^3} + \\
 & \frac{\delta \mathbf{H}_0^1}{\delta \mathbf{w}_0^1} \frac{\mathcal{G}_0^1}{\delta \mathbf{H}_0^1} \frac{\delta \mathbf{H}_0^2}{\delta \mathcal{G}_0^1} \frac{\mathcal{G}_0^2}{\delta \mathbf{H}_0^2} \frac{\delta \mathbf{Z}_1}{\delta \mathcal{G}_0^2} \frac{\mathcal{G}_1^3}{\delta \mathbf{H}_1^3} \frac{\mathcal{L}}{\delta \mathcal{G}_1^3} + \\
 & \frac{\delta \mathbf{H}_0^1}{\delta \mathbf{w}_0^1} \frac{\mathcal{G}_0^1}{\delta \mathbf{H}_0^1} \frac{\delta \mathbf{H}_0^2}{\delta \mathcal{G}_0^1} \frac{\mathcal{G}_0^2}{\delta \mathbf{H}_0^2} \frac{\delta \mathbf{Z}_2}{\delta \mathcal{G}_0^2} \frac{\mathcal{G}_2^3}{\delta \mathbf{H}_2^3} \frac{\mathcal{L}}{\delta \mathcal{G}_2^3} + \\
 & \frac{\delta \mathbf{H}_0^1}{\delta \mathbf{w}_0^1} \frac{\mathcal{G}_0^1}{\delta \mathbf{H}_0^1} \frac{\delta \mathbf{H}_1^2}{\delta \mathcal{G}_0^1} \frac{\mathcal{G}_1^2}{\delta \mathbf{H}_1^2} \frac{\delta \mathbf{Z}_0}{\delta \mathcal{G}_1^2} \frac{\mathcal{G}_0^3}{\delta \mathbf{H}_0^3} \frac{\mathcal{L}}{\delta \mathcal{G}_0^3} + \\
 & \frac{\delta \mathbf{H}_0^1}{\delta \mathbf{w}_0^1} \frac{\mathcal{G}_0^1}{\delta \mathbf{H}_0^1} \frac{\delta \mathbf{H}_1^2}{\delta \mathcal{G}_0^1} \frac{\mathcal{G}_1^2}{\delta \mathbf{H}_1^2} \frac{\delta \mathbf{Z}_1}{\delta \mathcal{G}_1^2} \frac{\mathcal{G}_1^3}{\delta \mathbf{H}_1^3} \frac{\mathcal{L}}{\delta \mathcal{G}_1^3} + \\
 & \frac{\delta \mathbf{H}_0^1}{\delta \mathbf{w}_0^1} \frac{\mathcal{G}_0^1}{\delta \mathbf{H}_0^1} \frac{\delta \mathbf{H}_1^2}{\delta \mathcal{G}_0^1} \frac{\mathcal{G}_1^2}{\delta \mathbf{H}_1^2} \frac{\delta \mathbf{Z}_2}{\delta \mathcal{G}_1^2} \frac{\mathcal{G}_2^3}{\delta \mathbf{H}_2^3} \frac{\mathcal{L}}{\delta \mathcal{G}_2^3} + \\
 & \frac{\delta \mathbf{H}_0^1}{\delta \mathbf{w}_0^1} \frac{\mathcal{G}_0^1}{\delta \mathbf{H}_0^1} \frac{\delta \mathbf{H}_2^2}{\delta \mathcal{G}_0^1} \frac{\mathcal{G}_2^2}{\delta \mathbf{H}_2^2} \frac{\delta \mathbf{Z}_0}{\delta \mathcal{G}_2^2} \frac{\mathcal{G}_0^3}{\delta \mathbf{H}_0^3} \frac{\mathcal{L}}{\delta \mathcal{G}_0^3} + \\
 & \frac{\delta \mathbf{H}_0^1}{\delta \mathbf{w}_0^1} \frac{\mathcal{G}_0^1}{\delta \mathbf{H}_0^1} \frac{\delta \mathbf{H}_2^2}{\delta \mathcal{G}_0^1} \frac{\mathcal{G}_2^2}{\delta \mathbf{H}_2^2} \frac{\delta \mathbf{Z}_1}{\delta \mathcal{G}_2^2} \frac{\mathcal{G}_1^3}{\delta \mathbf{H}_1^3} \frac{\mathcal{L}}{\delta \mathcal{G}_1^3} + \\
 & \frac{\delta \mathbf{H}_0^1}{\delta \mathbf{w}_0^1} \frac{\mathcal{G}_0^1}{\delta \mathbf{H}_0^1} \frac{\delta \mathbf{H}_2^2}{\delta \mathcal{G}_0^1} \frac{\mathcal{G}_2^2}{\delta \mathbf{H}_2^2} \frac{\delta \mathbf{Z}_2}{\delta \mathcal{G}_2^2} \frac{\mathcal{G}_2^3}{\delta \mathbf{H}_2^3} \frac{\mathcal{L}}{\delta \mathcal{G}_2^3}
 \end{aligned} \tag{2.82}$$

Note that this is only for computing the gradient of the loss function with respect to the weights of one neuron in the first hidden layer. We need to repeat a similar procedure for computing the gradient of loss with respect to every node in this graph. However, although this computation is feasible for small feedforward networks, we usually need feedforward network with more layers and with thousands of neurons in each layer to classify objects in images. In this case, the simple multivariate chain rule will not be feasible to use since a single update of parameters will take a long time due to excessive number of multiplications.

It is possible to make the computation of gradients more efficient. To this end, we can factorize the above equation as follows:

$$\begin{aligned} \frac{\delta \mathcal{L}}{\delta \mathbf{w}_0^1} = \frac{\delta \mathbf{H}_0^1}{\delta \mathbf{w}_0^1} \frac{\mathcal{G}_0^1}{\delta \mathbf{H}_0^1} & \left[\left(\frac{\delta \mathbf{H}_0^2}{\mathcal{G}_0^1} \frac{\mathcal{G}_0^2}{\delta \mathbf{H}_0^2} \left(\left(\frac{\delta \mathbf{Z}_0}{\mathcal{G}_0^2} \left(\frac{\mathcal{G}_0^3}{\delta \mathbf{H}_0^3} \frac{\mathcal{L}}{\mathcal{G}_0^3} \right) \right) + \left(\frac{\delta \mathbf{Z}_1}{\mathcal{G}_0^2} \left(\frac{\mathcal{G}_1^3}{\delta \mathbf{H}_1^3} \frac{\mathcal{L}}{\mathcal{G}_1^3} \right) \right) + \left(\frac{\delta \mathbf{Z}_2}{\mathcal{G}_0^2} \left(\frac{\mathcal{G}_2^3}{\delta \mathbf{H}_2^3} \frac{\mathcal{L}}{\mathcal{G}_2^3} \right) \right) \right) \right) + \right. \\ & \left(\frac{\delta \mathbf{H}_1^2}{\mathcal{G}_0^1} \frac{\mathcal{G}_1^2}{\delta \mathbf{H}_1^2} \left(\left(\frac{\delta \mathbf{Z}_0}{\mathcal{G}_1^2} \left(\frac{\mathcal{G}_0^3}{\delta \mathbf{H}_0^3} \frac{\mathcal{L}}{\mathcal{G}_0^3} \right) \right) + \left(\frac{\delta \mathbf{Z}_1}{\mathcal{G}_1^2} \left(\frac{\mathcal{G}_1^3}{\delta \mathbf{H}_1^3} \frac{\mathcal{L}}{\mathcal{G}_1^3} \right) \right) + \left(\frac{\delta \mathbf{Z}_2}{\mathcal{G}_1^2} \left(\frac{\mathcal{G}_2^3}{\delta \mathbf{H}_2^3} \frac{\mathcal{L}}{\mathcal{G}_2^3} \right) \right) \right) \right) + \\ & \left. \left(\frac{\delta \mathbf{H}_2^2}{\mathcal{G}_0^1} \frac{\mathcal{G}_2^2}{\delta \mathbf{H}_2^2} \left(\left(\frac{\delta \mathbf{Z}_0}{\mathcal{G}_2^2} \left(\frac{\mathcal{G}_0^3}{\delta \mathbf{H}_0^3} \frac{\mathcal{L}}{\mathcal{G}_0^3} \right) \right) + \left(\frac{\delta \mathbf{Z}_1}{\mathcal{G}_2^2} \left(\frac{\mathcal{G}_1^3}{\delta \mathbf{H}_1^3} \frac{\mathcal{L}}{\mathcal{G}_1^3} \right) \right) + \left(\frac{\delta \mathbf{Z}_2}{\mathcal{G}_2^2} \left(\frac{\mathcal{G}_2^3}{\delta \mathbf{H}_2^3} \frac{\mathcal{L}}{\mathcal{G}_2^3} \right) \right) \right) \right) \right] \end{aligned} \quad (2.83)$$

Compared with (2.82), the above equation requires much less multiplications which makes it more efficient in practice. The computations starts with the most inner parentheses and moves to the most outer terms. The above factorization has a very nice property. If we carefully study the above factorization it looks like that the direction of the edges are hypothetically reversed and instead of moving from \mathbf{w}_0^1 to \mathcal{L} the gradient computations moves in the reverse direction. Figure 2.37 shows the nodes analogous to each inner computation in the above equation.

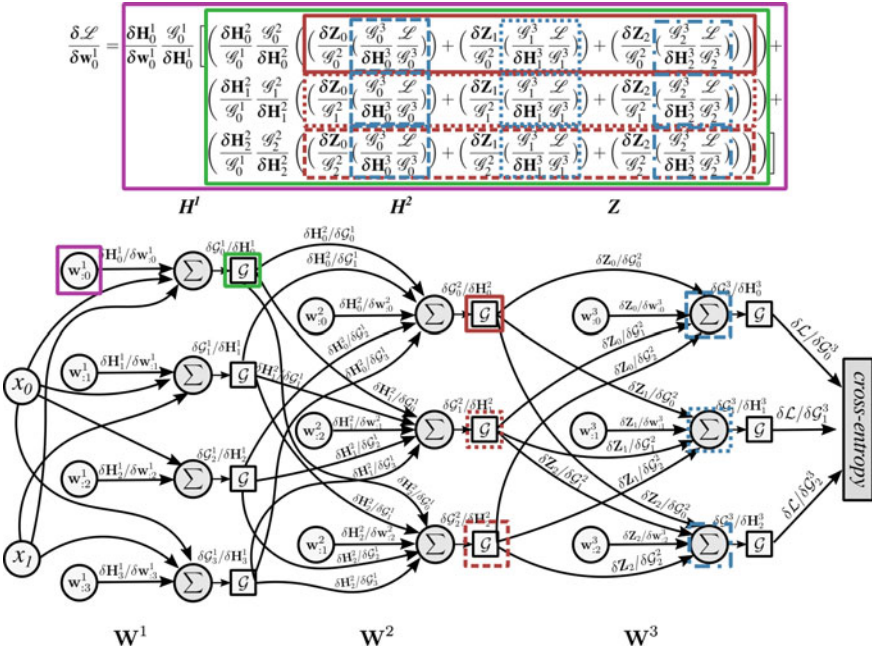


Fig. 2.37 Forward mode differentiation starts from the end node to the starting node. At each node, it sums the output edges of the node where the value of each edge is computed by multiplying the edge with the derivative of the child node. Each rectangle with different color and line style shows which part of the partial derivative is computed until that point

More precisely, assume the blue rectangles with dashed lines. These rectangles denote $\frac{\mathcal{G}_0^3}{\delta \mathbf{H}_0^3} \frac{\mathcal{L}}{\mathcal{G}_0^3}$ which corresponds to the node $\delta \mathbf{Z}_0$ on the graph. Furthermore, these rectangles in fact are equal to $\frac{\mathcal{L}}{\mathcal{Z}_0}$. Likewise, the blue rectangles with dotted lines and dashed-dotted lines denote $\frac{\mathcal{L}}{\mathcal{F}_1} = \frac{\mathcal{G}_1^3}{\delta \mathbf{H}_1^3} \frac{\mathcal{L}}{\mathcal{G}_1^3}$ and $\frac{\mathcal{L}}{\mathcal{F}_2} = \frac{\mathcal{G}_2^3}{\delta \mathbf{H}_2^3} \frac{\mathcal{L}}{\mathcal{G}_2^3}$ respectively.

The rectangles with solid red lines denote $\left(\frac{\delta \mathbf{Z}_0}{\mathcal{G}_0^2} \left(\frac{\mathcal{G}_0^3}{\delta \mathbf{H}_0^3} \frac{\mathcal{L}}{\mathcal{G}_0^3} \right) \right) + \left(\frac{\delta \mathbf{Z}_1}{\mathcal{G}_0^2} \left(\frac{\mathcal{G}_1^3}{\delta \mathbf{H}_1^3} \frac{\mathcal{L}}{\mathcal{G}_1^3} \right) \right) + \left(\frac{\delta \mathbf{Z}_2}{\mathcal{G}_0^2} \left(\frac{\mathcal{G}_2^3}{\delta \mathbf{H}_2^3} \frac{\mathcal{L}}{\mathcal{G}_2^3} \right) \right)$ which is analogous the derivative of the loss function with respect to $\delta \mathbf{H}_0^2$. In other words, before computing this rectangle, we have in fact computed $\frac{\mathcal{L}}{\mathbf{H}_0^2}$. Similarly, the dotted and dashed red rectangles illustrate $\frac{\mathcal{L}}{\mathbf{H}_1^2}$ and $\frac{\mathcal{L}}{\mathbf{H}_2^2}$ respectively. The same argument holds true with the green and purple rectangles.

Assume we want to compute $\frac{\delta \mathcal{L}}{\delta \mathbf{w}_1}$ afterwards. In that case, we do not need to compute none of the terms inside the red and blue rectangles since they have been computed once for $\frac{\delta \mathcal{L}}{\delta \mathbf{w}_0}$. This saves a great amount of computations especially when the network has many layers and neurons.

The *backpropagation* algorithm has been developed based on this factorization. It is a method for efficiently computing the gradient of leaf nodes with respect to each node on the graph using only one backward pass from the leaf nodes to input nodes. This algorithm can be applied on any computational graph. Formally, let $G = \langle \mathbf{V}, \mathbf{E} \rangle$ denotes a *directed acyclic graph* where $\mathbf{V} = \{v_1, \dots, v_K\}$ is set of nodes in the computational graph and $\mathbf{E} = (v_i, v_j) | v_i, v_j \in V$ is the set of ordered pairs (v_i, v_j) showing a directed edge from v_i to v_j . Number of edges going into a node is called *indegree* and the number of edges coming out of a node is called *outdegree*.

Formally, if $in(v_a) = \{(v_i, v_j) | (v_i, v_j) \in E \wedge v_j = v_a\}$ returns set of input edges to v_a , indegree of v_a will be equal to $|in(v_a)|$ where $|\cdot|$ returns the cardinality of a set. Likewise, $out(v_a) = \{(v_i, v_j) | (v_i, v_j) \in E \wedge v_i = v_a\}$ shows the set of output edges from v_a and $|out(v_a)|$ is equal to the outdegree of v_a . The computational node v_a is called an input if $in(v_a) = 0$ and $out(v_a) > 0$. Also, the computational node v_a is called a leaf if $out(v_a) = 0$ and $in(v_a) > 0$. Note that there must be only one leaf node in a computational graph which is typically the loss. This is due to the fact the we are always interested in computing the derivative of one node with respect to all other nodes in the graph. If there are more than one leaf node in the graph, the gradient of the leaf node of interest with respect to all other leaf nodes will be equal to zero.

Suppose that the leaf node of the graph is denoted by v_{leaf} . In addition, let $child(v_a) = \{v_j | (v_i, v_j) \in E \wedge v_i = v_a\}$ and $parent(v_a) = \{v_i | (v_i, v_j) \in E \wedge v_j = v_a\}$ returns the child nodes and parent nodes of v_a . Finally, depth of v_a is equal to number of edges on the *longest* path from input nodes to v_a . We denote the depth of v_a by $dep(v_a)$. It is noteworthy that for any node v_i in the graph that $dep(v_i) \geq dep(v_{leaf})$ the gradient of v_{leaf} with respect to v_i will be equal to zero. Based on the above discussion, the backpropagation algorithm is defined as follows:

Algorithm 1 The backpropagation algorithm

```

 $G : \langle V, E \rangle$  is a directed graph.
 $V$  is set of vertices
 $E$  is set of edges
 $v_{leaf}$  is the leaf node in  $V$ 
 $d_{leaf} \leftarrow dep(v_{leaf})$ 
 $v_{leaf}.d = 1$ 
for  $d = d_{leaf} - 1$  to 0 do
  for  $v_a \in \{v_i | v_i \in V \wedge dep(v_i) == d\}$  do
     $v_a.d \leftarrow 0$ 
    for  $v_c \in child(v_a)$  do
       $v_a.d \leftarrow v_a.d + \frac{\delta v_c}{\delta v_a} \times v_c.d$ 

```

The above algorithm can be applied on any computational graph. Generally, it computes gradient of a loss function (leaf node) with respect to all other nodes in the graph using only one backward pass from the loss node to the input node. In the above algorithm, each node is a data structure which stores information related to the computational unit including their derivative. Specifically, the derivative of v_a is stored in $v_a.d$. We execute the above algorithm on the computational graph shown in Fig. 2.38.

Based on the above discussion, $loss$ is the leaf node. Also, the longest path from input nodes to the leaf node is equal to $d_{leaf} = dep(loss) = 4$. According to the algorithm, $v_{leaf}.d$ must be set to 1 before executing the loop. In the figure, $v_{leaf}.d$ is illustrated using d_8 . Then, the loop start with $d = d_{leaf} - 1 = 3$. The first inner loop, iterates over all nodes in which their depth is equal to 3. This is equivalent to Z_0 and Z_1 on this graph. Therefore, v_a is set to Z_0 in the first iteration. The most inner loop also iterates over children of v_a . This is analogous to $child(Z_0) = \{loss\}$ which only has one child. Then, the derivative of v_a (Z_0) is set to $d_6 = v_a.d = 0 + r \times 1$.

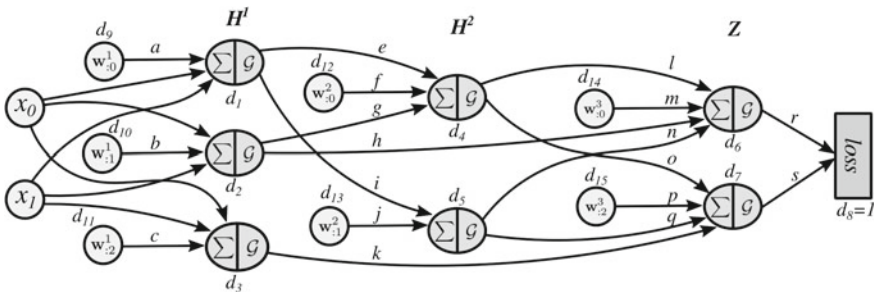


Fig. 2.38 A sample computational graph with a loss function. To cut the clutter, activations functions have been fused with the soma function of the neuron. Also, the derivatives on edges are illustrated using small letters. For example, g denotes $\frac{\delta H_0^2}{\delta H_1^1}$

Table 2.2 Trace of the backpropagation algorithm applied on Fig. 2.38

Depth	Node	Derivative
3	\mathbf{Z}_0	$d_6 = r \times 1$
3	\mathbf{Z}_1	$d_7 = s \times 1$
2	\mathbf{H}_0^2	$d_4 = l \times d_6 + o \times d_7$
2	\mathbf{H}_1^2	$d_5 = n \times d_6 + q \times d_7$
1	\mathbf{H}_0^1	$d_1 = e \times d_4 + i \times d_5$
1	\mathbf{H}_1^1	$d_2 = g \times d_4 + h \times d_5$
1	\mathbf{H}_1^1	$d_3 = k \times d_7$
0	$\mathbf{w}_{:0}^3$	$d_{14} = m \times d_6$
0	$\mathbf{w}_{:1}^3$	$d_{15} = p \times d_7$
0	$\mathbf{w}_{:0}^2$	$d_{12} = f \times d_4$
0	$\mathbf{w}_{:1}^2$	$d_{13} = j \times d_5$
0	$\mathbf{w}_{:0}^1$	$d_9 = a \times d_1$
0	$\mathbf{w}_{:1}^1$	$d_{10} = b \times d_2$
0	$\mathbf{w}_{:2}^1$	$d_{11} = c \times d_3$
0	x_0	$d_{16} = t \times d_1 + w \times d_2 + x \times d_3$
0	x_1	$d_{17} = y \times d_1 + z \times d_2 + zz \times d_3$

After that, the inner loop goes to \mathbf{Z}_1 and the most inner loop sets derivative of \mathbf{Z}_1 to $d_7 = v_a.d = 0 + s \times 1$.

At this point the inner loop finishes and the next iteration of the main loop start by setting d to 2. Then, the inner loop iterates over \mathbf{H}_0^2 and \mathbf{H}_1^2 . In the first iteration of the inner loop, \mathbf{H}_0^2 is selected and its derivative d_4 is set to 0. Next, the most inner loop iterates over children of \mathbf{H}_0^2 which are \mathbf{Z}_0 and \mathbf{Z}_1 . In the first iteration of the most inner loop d_4 is set to $d_4 = 0 + l \times d_6$ and in the second iteration it is set to $d_4 = l \times d_6 + o \times d_7$. At this point, the most inner loop is terminated and the algorithm proceeds with \mathbf{H}_1^2 . After finishing the most inner loop, the d_5 will be equal to $d_5 = n \times d_6 + q \times d_7$. Likewise, derivative of other nodes are updated. Table 2.2 shows how derivative of nodes in different depths are calculated using the backpropagation algorithm.

We encourage the reader to carefully study the backpropagation algorithm since it is a very efficient way for computing gradients in complex computational graphs. Since we are able to compute the gradient of loss function with respect to every parameter in a feedforward neural network, we can train a feedforward network using the gradient descend method (Appendix A).

Given an input \mathbf{x} , the data is forwarded throughout the network until it reaches to the leaf node. Then, the backpropagation algorithm is executed and the gradient of loss with respect to every node given the input \mathbf{x} is computed. Using this gradient, the parameters vectors are updated.

2.6.2 Activation Functions

There are different kinds of activation functions that can be used in neural networks. However, we are mainly interested in activation functions that are nonlinear and continuously differentiable. A nonlinear activation function makes it possible that a neural network learns any nonlinear functions provided that the network has enough neurons and layers. In fact, a feedforward network with linear activations in all neurons is just a linear function. Consequently, it is important that to have at least one neuron with a nonlinear activation function to make a neural network nonlinear.

Differentiability property is also important since we mainly train a neural network using gradient descend method. Although non-gradient-based optimization methods such as *genetic algorithms* and *particle swarm optimization* are used for optimizing simple functions, but gradient-based methods are the most commonly used methods for training neural networks. However, using non-gradient-based methods for training a neural network is an active research area.

Beside the above factors, it is also desirable that the activation function approximates the identity mapping near origin. To explain this, we should consider the activation of a neuron. Formally, the activation of a neuron is given by $\mathcal{G}(\mathbf{w}\mathbf{x}^T + b)$ where \mathcal{G} is the activation function. Usually, the weight vector \mathbf{w} and bias b are initialized with values close to zero by the gradient descend method. Consequently, $\mathbf{w}\mathbf{x}^T + b$ will be close to zero. If \mathcal{G} approximates the identity function near zero, its gradient will be approximately equal to its input. In other words, $\delta\mathcal{G} \approx \mathbf{w}\mathbf{x}^T + b \iff \mathbf{w}\mathbf{x}^T + b \approx 0$. In terms of the gradient descend, it is a strong gradient which helps the training algorithm to converge faster.

2.6.2.1 Sigmoid

The sigmoid activation function and its derivative are given by the following equations. Figure 2.39 shows their plots.

$$\mathcal{G}_{\text{sigmoid}}(x) = \frac{1}{1 + e^{-x}} \quad (2.84)$$

and

$$\mathcal{G}'_{\text{sigmoid}}(x) = \mathcal{G}(x)(1 - \mathcal{G}(x)). \quad (2.85)$$

The sigmoid activation $\mathcal{G}_{\text{sigmoid}}(x) : \mathbb{R} \rightarrow [0, 1]$ is smooth and it is differentiable everywhere. In addition, it is a biologically inspired activation function. In the past, sigmoid was very popular activation function in feedforward neural networks. However, it has two problems. First, it does not approximate the identity function near zero. This is due to the fact that $\mathcal{G}_{\text{sigmoid}}(0)$ is not close to zero and $\mathcal{G}'_{\text{sigmoid}}(x)$ is not close to 1. More importantly, sigmoid is a *squashing* function meaning that it saturates as $|x|$ increases. In other words, its gradient becomes very small if x is not close to origin.

This causes a serious problem in backpropagation which is known as *vanishing gradients* problem. The backpropagation algorithm multiplies the gradient of the

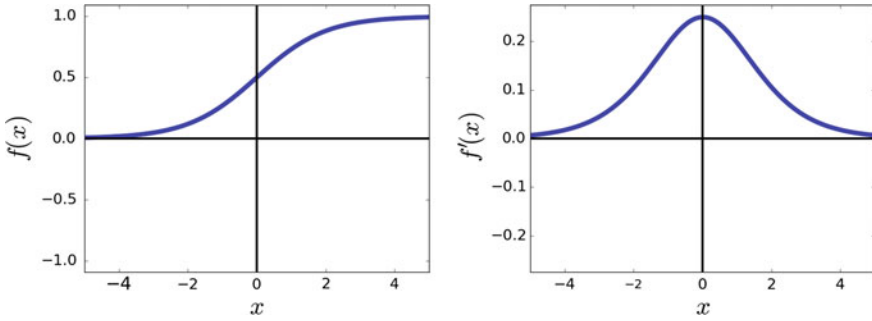


Fig. 2.39 Sigmoid activation function and its derivative

activation function with its children in order to compute the gradient of the loss function with respect to the current node. If x is far from origin, $\mathcal{G}_{sigmoid}$ will be very small. When it is multiplied by its children, the gradient of the loss with respect to that node will become smaller. If there are many layers with sigmoid activation, the gradient starts to become approximately zero (i.e., gradient vanishes) in the first layers. For this reason, the weight changes will be very small or even negligible. This cause the network to stuck in the current configuration of parameters and do not learn anymore. For these reasons, sigmoid activation function is not used in deep architectures since training the network become nearly impossible.

2.6.2.2 Hyperbolic Tangent

The *hyperbolic tangent* activation function is in fact a rescaled version of the sigmoid function. Its defined by the following equations. Figure 2.40 illustrates the plot of the function and its derivative.

$$\mathcal{G}_{tanh}(x) = \frac{e^x + e^{-x}}{e^x - e^{-x}} = \frac{2}{1 + e^{-2x}} - 1 \quad (2.86)$$

$$\mathcal{G}'_{tanh}(x) = 1 - \mathcal{G}_{tanh}(x)^2 \quad (2.87)$$

The hyperbolic tangent function $\mathcal{G}_{tanh}(x) : \mathbb{R} \rightarrow [-1, 1]$ is a smooth function which is differentiable everywhere. Its range is $[-1, 1]$ as opposed to range of the sigmoid function which is $[0, 1]$. More importantly, the hyperbolic tangent function approximates the identity function close to origin. This is easily observable from the plots where $\mathcal{G}_{tanh}(0) \approx 0$ and $\mathcal{G}'_{tanh}(0) \approx 1$. This is a desirable property which increases the convergence speed of the gradient descend algorithm. However, similar to the sigmoid activation function, it saturates as $|x|$ increases. Therefore, it may suffer from vanishing gradient problems in feedforward neural networks with many layers. Nonetheless, the hyperbolic activation function is preferred over the sigmoid function because it approximates the identity function near origin.

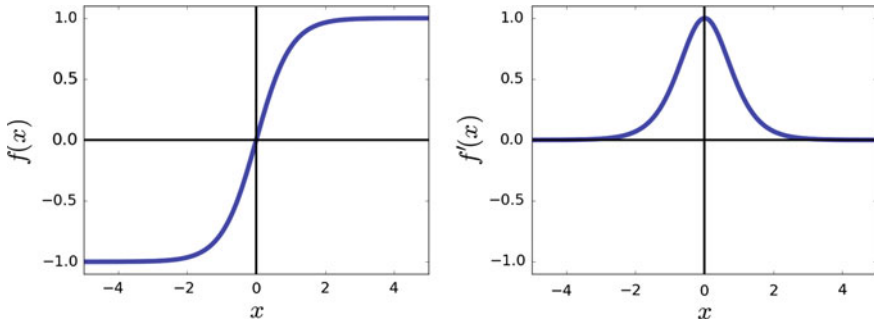


Fig. 2.40 Tangent hyperbolic activation function and its derivative

2.6.2.3 Softsign

The softsign activation function is closely related to the hyperbolic tangent function. However, it has more desirable properties. Formally, the softsign activation function and its derivative are defined as follows:

$$\mathcal{G}_{softsign}(x) = \frac{x}{1 + |x|} \quad (2.88)$$

$$\mathcal{G}'_{softsign}(x) = \frac{1}{(1 + |x|)^2} \quad (2.89)$$

Similar to the hyperbolic tangent function, the range of the softsign function is $[-1, 1]$. Also, the function is equal to zero at origin and its derivative at origin is equal to 1. Therefore, it approximates the identity function at origin. Comparing the function and its derivative with hyperbolic tangent, we observe that it also saturates as $|x|$ increases. However, the saturation ratio of the softsign function is less than the hyperbolic tangent function which is a desirable property. In addition, gradient of the softsign function near origin drops with a greater ratio compared with the hyperbolic tangent. In terms of computational complexity, softsign requires less computation than the hyperbolic tangent function. The softsign activation function can be used as an alternative to the hyperbolic tangent activation function (Fig. 2.41).

2.6.2.4 Rectified Linear Unit

Using the sigmoid, hyperbolic tangent and softsign activation functions is mainly limited to neural networks with a few layers. When a feedforward network has few hidden layers it is called a *shallow* neural network. In contrast, a network with many hidden layers is called a *deep* neural network. The main reason is that in deep neural networks, gradient of these three activation functions vanishes during the backpropagation which causes the network to stop learning in deep networks.

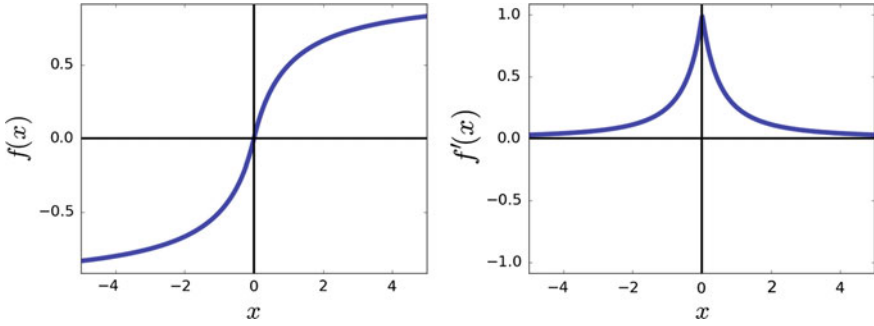


Fig. 2.41 The softsign activation function and its derivative

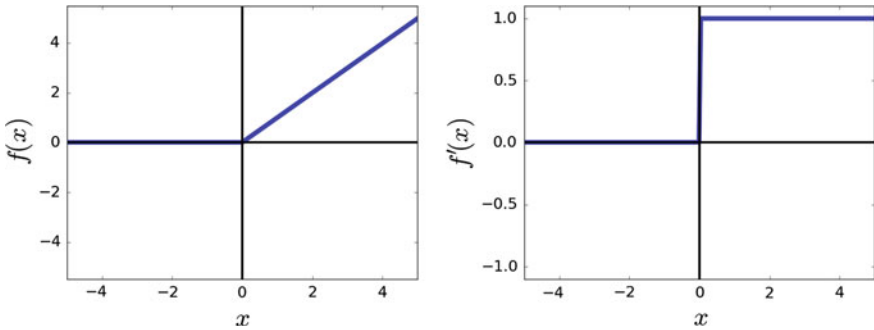


Fig. 2.42 The rectified linear unit activation function and its derivative

A *rectified linear unit* (ReLU) is an activation function which is computationally very efficient and it is defined as follows:

$$\mathcal{G}_{relu}(x) = \max(0, x) \quad (2.90)$$

$$\mathcal{G}'_{relu}(x) = \begin{cases} 0 & x < 0 \\ 1 & x \geq 0 \end{cases} \quad (2.91)$$

ReLU is a very simple nonlinear activation function which actually works very well in practice. Its derivative in \mathbb{R}^+ is always 1 and it does not saturate in \mathbb{R}^+ . In other words, the range of this function is $[0, \infty)$. However, this function does not approximate the identity function near origin. But because it does not saturate in \mathbb{R}^+ it always produce a strong gradient in this region. Consequently, it does not suffer from the vanishing gradient problem. For this reason, it is a good choice for deep neural networks (Fig. 2.42).

One property of the ReLU activation is that it may produce dead neurons during the training. A dead neuron always return 0 for every sample in the dataset. This may happen because the weight of a dead neuron have been adjusted such that $\mathbf{w}\mathbf{x}$ for the

neuron is always negative. As the result, when it is passed to the ReLU activation function, it always return zero. The advantage of this property is that, the output of a layer may have entries which are always zero. This outputs can be removed from the network to make it computationally more efficient. The negative side of this property is that dead neuron may affect the overall accuracy of the network. So, it is always a good practice to check the network during training for dead neurons.

2.6.2.5 Leaky Rectified Linear Unit

The basic idea behind *Leaky ReLU* (Maas et al. 2013) is to solve the problem of dead neuron which is inherent in ReLU function. The leaky ReLU is defined as follows:

$$\mathcal{G}_{rrelu}(x) = \begin{cases} \alpha x & x < 0 \\ x & x \geq 0 \end{cases} \quad (2.92)$$

$$\mathcal{G}'_{rrelu}(x) = \begin{cases} \alpha & x < 0 \\ 1 & x \geq 0 \end{cases} \quad (2.93)$$

One interesting property of leaky ReLU is that its gradient does not vanish in negative region as opposed to ReLU function. Rather, it returns the constant value α . The hyperparameter α usually takes a value between $[0, 1]$. Common value is to set α to 0.01. But, on some datasets it works better with higher values as it is proposed in Xu et al. (2015). In practice, leaky ReLU and ReLU may produce similar results. This might be due to the fact that the positive region of these function is identical (Fig. 2.43).

2.6.2.6 Parameterized Rectified Linear Unit

Parameterized rectified linear unit is in fact (PReLU) the leaky ReLU (He et al. 2015). The difference is that α is treated as a parameter of the neural network so it

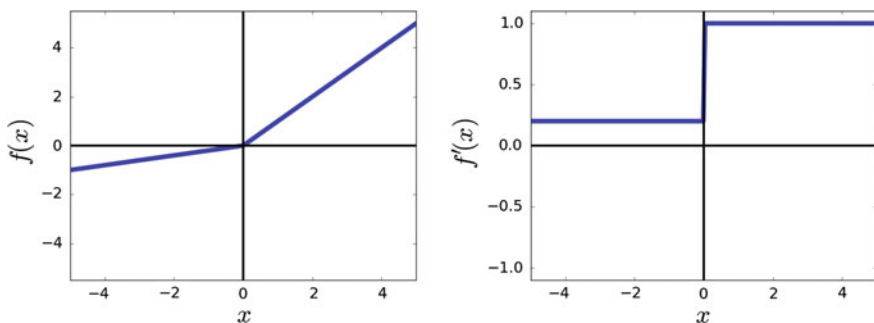


Fig. 2.43 The leaky rectified linear unit activation function and its derivative

can be learned from data. The only thing that needs to be done is to compute the gradient of the leaky ReLU function with respect to α which is given by:

$$\frac{\delta \mathcal{G}_{prelu}(x)}{\delta \alpha} = \begin{cases} x & x < 0 \\ \alpha & x \geq 0 \end{cases} \quad (2.94)$$

Then, the gradient of the loss function with respect to α is obtained using the back-propagation algorithm and it is updated similar to other parameters of the neural network.

2.6.2.7 Randomized Leaky Rectified Linear Unit

The main idea behind *randomized rectified linear unit* (RReLU) is to add randomness to the activations during training of a neural network. To achieve this goal, the RReLU activation draws the value of α from the uniform distribution $\mathcal{U}(a, b)$ where $a, b \in [0, 1)$ during training of the network. Drawing the value of α can be done once for all the network or it can be done for each layer separately. To increase the randomness, one may draw different α from the uniform distribution for each neuron in the network. Figure 2.44 illustrates how the function and its derivative vary using this method.

In the test time, the parameter α is set to the constant value $\bar{\alpha}$. This value is obtained by computing the mean value of α for each neuron that is assigned during training. Since the value of alpha is drawn from $\mathcal{U}(a, b)$, then value of $\bar{\alpha}$ can be easily obtained by computing the expected value of $\mathcal{U}(a, b)$ which is equal to $\bar{\alpha} = \frac{a+b}{2}$.

2.6.2.8 Exponential Linear Unit

Exponential linear units (ELU) (Clevert et al. 2015) can be seen as a smoothed version of the shifted ReLU activation function. By shifted ReLU we mean to change the original ReLU from $\max(0, x)$ to $\max(-1, x)$. Using this shift, the activation passes a negative number near origin. The exponential linear unit approximates the shifted

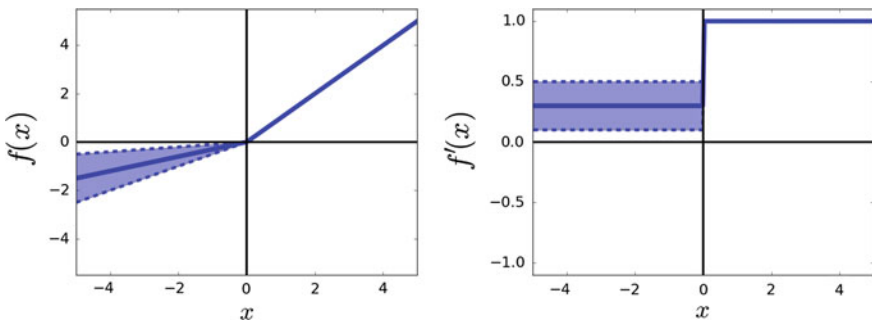


Fig. 2.44 The softplus activation function and its derivative

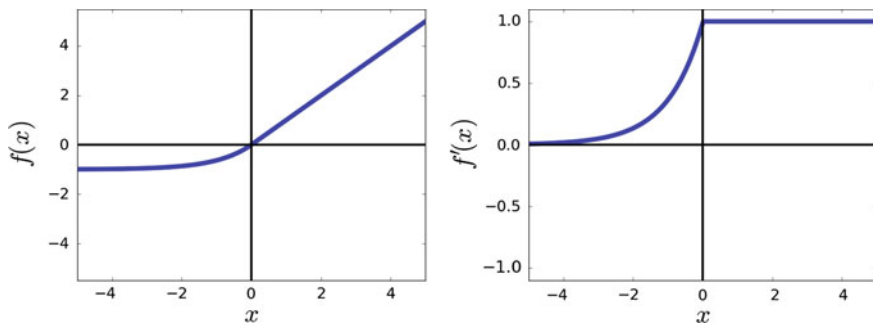


Fig. 2.45 The exponential linear unit activation function and its derivative

ReLU using a smooth function which is given by:

$$\mathcal{G}_{elu}(x) = \begin{cases} \alpha(e^x - 1) & x < 0 \\ x & x \geq 0 \end{cases} \quad (2.95)$$

$$\mathcal{G}'_{elu}(x) = \begin{cases} \mathcal{G}(x) + \alpha & x < 0 \\ 1 & x \geq 0 \end{cases} \quad (2.96)$$

The ELU activation usually speeds up the learning. Also, as it is illustrated in the plot, its derivative does not drop immediately in the negative region. Instead, the gradient of the negative region saturates nonlinearly (Fig. 2.45).

2.6.2.9 Softplus

The last activation function that we explain in this book is called *Softplus*. Broadly speaking, we can think of the softplus activation function as a smooth version of the ReLU function. In contrast to the ReLU which is not differentiable at origin, the softplus function is differentiable everywhere. In addition, similar to the ReLU activation, its range is $[0, \infty)$. The function and its derivative are defined as follows:

$$\mathcal{G}_{softplus} = \ln(1 + e^x) \quad (2.97)$$

$$\mathcal{G}'_{softplus} = \frac{1}{1 + e^{-x}} \quad (2.98)$$

The derivative of the softplus function is the sigmoid function which means the range of derivative is $[0, 1]$. The difference with ReLU is the fact that the derivative of softplus is also a smooth function which saturates as $|x|$ increases (Fig. 2.46).

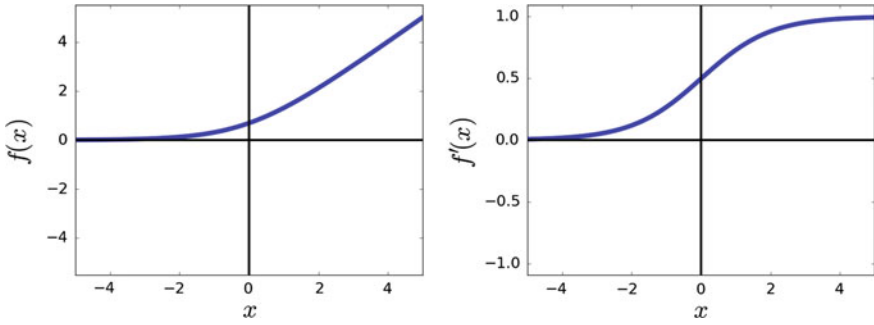


Fig. 2.46 The softplus activation function and its derivative

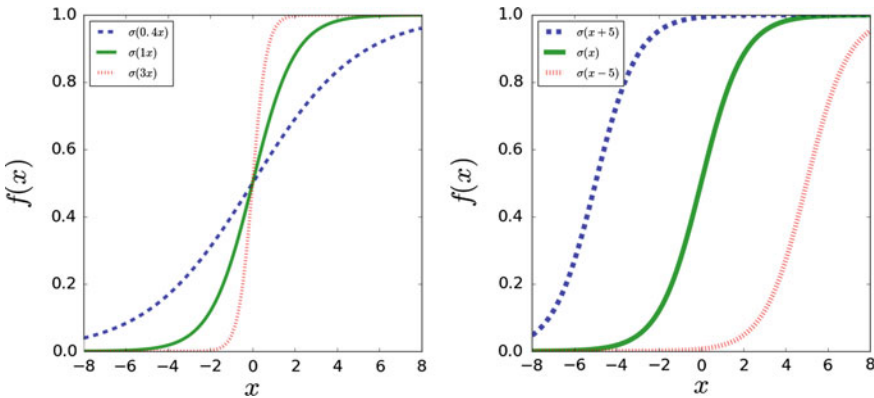


Fig. 2.47 The weights affect the magnitude of the function for a fixed value of bias and \mathbf{x} (left). The bias term shifts the function to left or right for a fixed value of \mathbf{w} and \mathbf{x} (right)

2.6.3 Role of Bias

Basically, the input to an activation function is $\mathbf{w}\mathbf{x}^T + b$. The first term in this equation, computes the dot product between \mathbf{w} and \mathbf{x} . Assume that x is a one-dimensional vector (scaler). To see the effect of w , we can set $b = 0$ and keep the value of x fixed. Then, the effect of \mathbf{w} can be illustrated by plotting the activation function for different values of \mathbf{w} . This is shown in left plot in Fig. 2.47.

We observe that changing the weights affects the magnitude of activation function. For example, assume a neural network without a hidden layer where the output layer has only one neuron with sigmoid activation function. The output of the neural network for inputs $\mathbf{x}_1 = 6$ and $\mathbf{x}_2 = -6$ are equal to $\sigma(6w + b) = 0.997$ and $\sigma(-6w + b) = 0.002$ when $w = 1$ and $b = 0$. Suppose we want to find \mathbf{w} and keep $b = 0$ such that $\sigma(6w + b) = 0.999$ and $\sigma(-6w + b) = 0.269$. There is no \mathbf{w} which perfectly satisfies these two conditions. But, it is possible to find \mathbf{w} that approximates the above values as accurate as possible. To this end, we only need to minimize the

squared error loss of the neuron. If we do this, the approximation error will high indicating that it is not possible to approximate these values accurately.

However, it is possible to find b where $\sigma(6w + b) = 0.999$ and $\sigma(-6w + b) = 0.269$ when $\mathbf{w} = 1$. To see the effect of b , we can keep \mathbf{w} and \mathbf{x} fixed and change the value of b . The right plot in Fig. 2.47 shows the result. It is clear that the bias term shifts the activation function to left or right. It gives a neuron more freedom to be fitted on data.

According to the above discussion, using bias term in a neuron seems necessary. However, bias term might be omitted in very deep neural networks. Assume the final goal of a neural network is to estimated ($x = 6, f(x) = 0.999$) and ($x = -6, f(x) = 0.269$). If we are forced to only use a single layer neural network with only one neuron in the layer, the estimation error will be high without a bias term. But, if we are allowed to use more layers and neurons, then it is possible to design a neural network that accurately approximates these pairs of data.

In deep neural networks, even if the bias term is omitted, the network might be able to shift the input across different layers if it reduces the loss. Though, it is a common practice to keep the bias term and train it using data. Omitting the bias term may only increase the computational efficiency of a neural network. If the computational resources are not limited, it is not necessary to remove this term from neurons.

2.6.4 Initialization

The gradient descend algorithm starts by setting an initial value for parameters. A feedforward neural network has mainly two kind of parameters including weights and biases. All biases are usually initialized to zero. There are different algorithms for initializing the weights. To common approach is to initialize them using a uniform or a normal distribution. We will explain initialization methods in the next chapter.

The most important thing to keep in mind is that, weights of the neurons must be different. If they all have the same value. Neurons in the same layer will have identical gradients leading to the same update rule. For this reason, weights must be initialized with different values. Also, they are commonly initialized very close to zero.

2.6.5 How to Apply on Images

Assume the dataset $\mathcal{X} = \{(\mathbf{x}_1, y_1), \dots, (\mathbf{x}_n, y_n)\}$ where the input vector $\mathbf{x}_i \in \mathbb{R}^{1000}$ is a 1000-dimensional vector and $y_i = [0, \dots, c]$ is an integer number indicating the class of the vector. A rule of thumb in designing a neural network for classification of these vectors is to have more neurons in the first hidden layer and start to decrease the number of neurons in the subsequent layers. For instance, we can design a neural network with three hidden layers where the first hidden layer has 5000 neurons, the second hidden layer has 2000 neurons and the third hidden layer has 500 neurons. Finally, the output layer also will contain c neurons.

One important step in designing a neural network is to count the total number of parameters in the network. For example, there are $5000 \times 1000 = 5,000,000$ weights between the input layer and the first hidden layer. Also, the first hidden layer has 5000 biases. Similarly, the number of parameters between the first hidden layer and second hidden layer is equal to $5000 \times 2000 = 10,000,000$ plus 2000 biases. The number of parameters between the second hidden layer and the third hidden layer is also equal to $2000 \times 500 = 1,000,000$ plus 500 biases. Finally, the number of weights and biases between the third hidden layer and the output layer is equal to $500 \times c + c$. Overall, this neural network is formulated using $16,007,200 + 500c + c$ parameters. Even for this shallow neural network, the number of parameters is very high. Training this neural network requires a dataset with many training samples. Collecting this dataset might not be practical.

Now, suppose our aim is to classify traffic signs. The input of the classifier might be $50 \times 50 \times 3$ images. Our aim is to classify 100 classes of traffic signs. We mentioned before that training a classifier directly on pixel intensities does not produce accurate results. Better results were obtained by extracting features using the histogram of oriented gradients. We also mentioned that neural networks learn the feature transformation function automatically from data.

Consequently, we can design a neural network where the input of the network is raw images and its output is the classification scores of the image per each class of traffic sign. The neural network learns to extract features from the image so that they become linearly separable in the last hidden layer. A $50 \times 50 \times 3$ image can be stored in a three-dimensional matrix. If we flatten this matrix, the results will be a 7500-dimensional vector.

Suppose a neural network containing three hidden layers with 10000-8000-3000 neurons in these layers. This network is parameterized using 179,312,100 parameters. A dramatically smaller neural network with three hidden layers such as 500-300-250 will also have 4,001,150 parameters. Although the number of parameters in the latter neural network is still high, it may not produce accurate results. In addition, the number of parameters in the former network is very high which makes it impossible to train this network with the current algorithms, hardware and datasets.

Besides, classification of objects is a complex problem. The reason is that some of traffic signs differ only slightly. Also, their illumination changes during day. There are also other factors that we will discuss in the later chapters. For these reasons, accurately learning a feature transformation function that traffic signs linearly separable in the feature space requires a deeper architecture. As the depth of neural network increases, the number of parameters may also increase. The reason that a deeper model is preferable over a shallower model is described on Fig. 2.48.

The wide black line on this figure shows the function that must be approximated using a neural network. The red line illustrates the output of a neural network including four hidden layers with 10-10-9-6 architecture using the hyperbolic tangent activation functions. In addition, the white line shows the output of a neural network consisting of five layers with 8-6-4-3-2 architecture using the hyperbolic tangent activation function. Comparing the number of parameters in these two networks, the shallower network has 296 parameters and the deeper network has 124 parameters. In

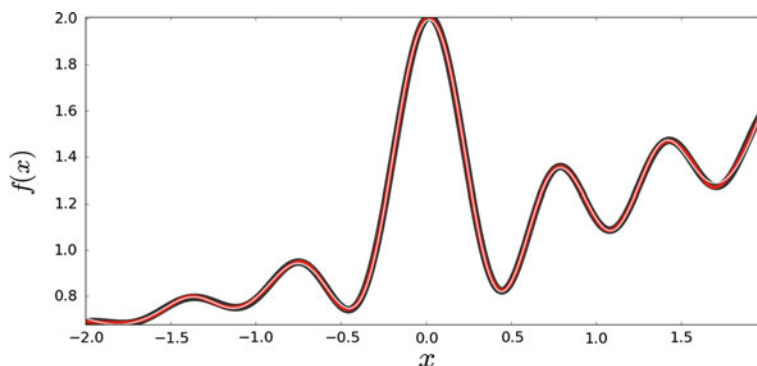


Fig. 2.48 A deeper network requires less neurons to approximate a function

general, deeper models require less parameters for modeling a complex function. It is obvious from figure that the deeper model is approximated the function accurately despite the fact that it has much less parameters.

Feedforward neural networks that we have explained in this section are called *fully connected* feedforward neural networks. The reason is that every neuron in one layer is connected to *all* neurons in the previous layer. As we explained above, modeling complex functions such as extracting features from an image may require deep neural networks. Training deep fully connected networks on dataset of images is not tractable due to very high number of parameters. In the next chapter, we will explain a way to dramatically reduce the number of parameters in a neural network and train them on images.

2.7 Summary

In this chapter, we first explained what are classification problems and what is a decision boundary. Then, we showed how to model a decision boundary using linear models. In order to better understand the intuition behind a linear model, they were also studied from geometrical perspective. A linear model needs to be trained on a training dataset. To this end, there must be a way to assess how good is a linear model in classification of training samples. For this purpose, we thoroughly explained different loss functions including 0/1 loss, squared loss, hinge loss, and logistic loss. Then, methods for extending binary models to multiclass models including one-versus-one and one-versus-rest were reviewed. It is possible to generalize a binary linear model directly into a multiclass model. This requires loss functions that can be applied on multiclass dataset. We showed how to extend hinge loss and logistic loss into multiclass datasets.

The big issue with linear models is that they perform poorly on datasets in which classes are not linearly separable. To overcome this problem, we introduced the

idea of feature transformation function and applied it on a toy example. Designing a feature transformation function by hand could be a tedious task especially when they have to be applied on high-dimensional datasets. A better solution is to learn a feature transformation function directly from training data and training a linear classifier on top of it.

We developed the idea of feature transformation from simple functions to compositional functions and explained how neural networks can be used for simultaneously learning a feature transformation function together with a linear classifier. Training a complex model such as neural network requires computing gradient of loss function with respect to every parameter in the model. Computing gradients using conventional chain rule might not be tractable. We explained how to factorize a multivariate chain rule and reduce the number of arithmetic operations. Using this formulation, we explained the backpropagation algorithm for computing gradients on any computational graph.

Next, we explained different activation functions that can be used in designing neural networks. We mentioned why ReLU activations are preferable over traditional activations such as hyperbolic tangent. Role of bias in neural networks is also discussed in detail. Finally, we finished the chapter by mentioning how an image can be used as the input of a neural network.

2.8 Exercises

2.1 Find an equation to compute the distance of point \mathbf{p} from a line.

2.2 Given the convex set $X \subset \mathbb{R}^d$, we know that function $f(x) : X \rightarrow \mathbb{R}$ is convex if:

$$\forall_{\mathbf{x}_1, \mathbf{x}_2 \in X, \alpha \in [0, 1]} f(\alpha \mathbf{x}_1 + (1 - \alpha) \mathbf{x}_2) \leq \alpha f(\mathbf{x}_1) + (1 - \alpha) f(\mathbf{x}_2). \quad (2.99)$$

Using the above definition, show why 0/1 loss function is nonconvex?

2.3 Prove that square loss is a convex function.

2.4 Why setting a in the hinge loss to different values does not affect the classification accuracy of the learn model?

2.5 Compute the partial derivative of the squared hinge loss and modified Huber loss functions.

2.6 Apply $\log(A \times B) = \log(A) \log(B)$ on (2.39) to obtain (2.39).

2.7 Show that:

$$\frac{\delta \sigma(a)}{a} = \sigma(a)(1 - \sigma(a)). \quad (2.100)$$

2.8 Find the partial derivative of (2.41) with respect to w_i using the chain rule of derivatives.

2.9 Show how we obtained (2.46).

2.10 Compute the partial derivatives of (2.46) and use them in the gradient descend method for minimizing the loss represented by this equation.

2.11 Compute the partial derivatives of (2.56) and obtain (2.57).

2.12 Draw an arbitrary computation graph with three leaf nodes and call them A , B and C . Show that $\delta C / \delta A = 0$ and $\delta C / \delta B = 0$

2.13 Show that a feedforward neural network with linear activation functions in all layers is in fact just a linear function.

2.14 Show that it is impossible to find a \mathbf{w} such that:

$$\begin{aligned} \sigma(6w) &= \frac{1}{1 + e^{-6w}} = 0.999 \\ \sigma(-6w) &= \frac{1}{1 + e^{6w}} = 0.269 \end{aligned} \quad (2.101)$$

References

- Clevert DA, Unterthiner T, Hochreiter S (2015) Fast and accurate deep network learning by exponential linear units (ELUs). 1997, pp 1–13. [arXiv:1511.07289](#)
- He K, Zhang X, Ren S, Sun J (2015) Delving deep into rectifiers: surpassing human-level performance on ImageNet classification. [arXiv:1502.01852](#)
- Maas AL, Hannun AY, Ng AY (2013) Rectifier nonlinearities improve neural network acoustic models. In: ICML workshop on deep learning for audio, speech and language processing, vol 28. http://www.stanford.edu/~awni/papers/relu_hybrid_icml2013_final.pdf
- Stallkamp J, Schlipsing M, Salmen J, Igel C (2012) Man vs. computer: benchmarking machine learning algorithms for traffic sign recognition. *Neural Netw* 32:323–332. doi:[10.1016/j.neunet.2012.02.016](#)
- Xu B, Wang N, Chen T (2015) Empirical evaluation of rectified activations in convolutional network. [arXiv:1505.00853v2](#)

<http://www.springer.com/978-3-319-57549-0>

Guide to Convolutional Neural Networks

A Practical Application to Traffic-Sign Detection and
Classification

Habibi Aghdam, H.; Jahani Heravi, E.

2017, XXIII, 282 p. 150 illus., 111 illus. in color.,

Hardcover

ISBN: 978-3-319-57549-0