Machine Learning
A Quantitative Approach

Henry H. Liu

PerfMath

ISBN-13: 978-1986487528
ISBN-10: 1986487520

10 9 8 7 6 5 4 3 2 1
03232019

1 Introduction to Machine Learning

None

2 Machine Learning Fundamentals Illustrated with Regression

2.5 EVALUATING A MACHINE LEARNING MODEL

In this section, we discuss how a machine learning model like the linear regression model we have just used can be evaluated. We start with the mathematical representation of the linear model, and then we discuss what metrics should be used to evaluate the model. Let's begin with the mathematical representation of the linear regression model first next.

2.5.1 Mathematical representation of the linear regression model

The simplest linear regression model can be described mathematically as follows:

$$y(\boldsymbol{w}, \boldsymbol{x}) = w_0 + \sum_{m=1}^{M} w_m x^{(m)} = \sum_{m=0}^{M} w_m x^{(m)} \qquad (2.1)$$

where $\boldsymbol{w} = (w_0, w_1, \ldots, w_M)$ is called a weight parameter vector, while $\boldsymbol{x} = (x^{(1)}, \ldots, x^{(M)})$ is called an *input variable vector* or *feature vector* of dimension M, such as the features listed in Table 2.1 for our fuel economy use case. Note that we added a constant value of 1 to the $\boldsymbol{x}$ vector so that we can include w_0 into the right most sum of Eq. (2.1).

In fact, when we say "*linear*," we mean the function $y(\boldsymbol{w},\boldsymbol{x})$ is a linear combination of the weight parameters, not necessarily the feature variable $\boldsymbol{x}$. We can actually rewrite Eq. (2.1) into the following form using generic basis functions $\phi_m(x)$ of any form

$$y(\boldsymbol{w},\boldsymbol{x}) = w_0 + \sum_{m=1}^{M} w_m \phi_m(x) \qquad (2.2)$$

Note that in Eqs. (2.1) and (2.2), w_0 is called a *bias* or *intercept*, which is the value of $y(\boldsymbol{w},\boldsymbol{x})$ when $\boldsymbol{x}$ or basis functions are all zeroes.

The linear regression model optimizes the parameter vector by minimizing the *residual sum of squares* (RSS) as defined below:

$$RSS = \sum_{n=1}^{N} (y(x_n) - y_n)^2 \qquad (2.3)$$

where y_n is the n^{th} observed value of the variable y to be predicted, $x_n = (x^{(1)}, x^{(2)}, \ldots, x^{(M)})_n$ the n^{th} data point or row value of the input variable x, and $y(x_n)$ the value of the variable y predicted on x_n.

Next, we discuss the metrics used for evaluating the performance of the linear models we described above in Eq. (2.2).

2.5.2 MSE, RMSE and R^2 (coefficient of determination) as performance metrics

The mean squared error (MSE) is one of the most common metrics for evaluating the performance of a machine learning model, including the linear regression model we discussed previously. It is defined as follows:

$$MSE = \frac{1}{N} RSS = \frac{1}{N} \sum_{n=1}^{N} (y(x_n) - y_n)^2 \qquad (2.4)$$

i.e., the MSE is the mean of the residual sum of squares, also known as the square of the errors with the error defined as follows:

$$\epsilon_n = y(x_n) - y_n \qquad (2.5)$$

Note that by taking the square of the error instead of just the error, the sign of the error is eliminated, which does not matter in terms of minimizing the errors.

However, the MSE does not have the same dimension as the predictor y or the error, so most often the *root mean squared error* (RMSE) is used in place of the MSE for evaluating the performance or accuracy of a machine learning model:

$$RMSE = \sqrt{\frac{1}{N} \sum_{n=1}^{N} (y(x_n) - y_n)^2} \qquad (2.6)$$

Now, the RMSE has the same unit as the predictor, but still there is an issue here, that is, it is absolute rather than relative. Ideally, we would like to use a metric that is normalized to the range of [0, 1] so that we would know how far we were from hitting a perfect target. That is the metric of R^2 (pronounced "R squared") as discussed next.

R^2 is formally termed the coefficient of determination. Its definition starts with the mean of the observed data as shown below:

$$y_{mean} = \frac{1}{N}\sum_{n=1}^{N} y_n \tag{2.7}$$

Then, the total sum of squares is defined as the sum of squares of observed y_n relative to y_{mean} as follows:

$$SS_{total} = \sum_{n=1}^{N} (y_n - y_{mean})^2 \tag{2.8}$$

Given the definition of the RSS in Eq. (2.3), the R^2 (the coefficient of determination) is now:

$$R^2 = 1 - \frac{SS_{rss}}{SS_{total}} \tag{2.9}$$

So if we can make a perfect fit so that the residual sum of squares reaches zero, then we will have $R^2 = 1$. Instead, if we came out with $R^2 = 0.15$, for example, we would know that we were 85% away from a perfect fit. Or in a more formal term, R^2 is related to something called the *fraction of variance unexplained* (FVU) as follows:

$$R^2 = 1 - \text{FVU} \tag{2.10}$$

as the numerator of SS_{rss} in the second term of Eq. (2.9) is considered unexplained variance as the result of model fitting errors. For this reason, R^2 is a better measure for the accuracy or performance of a model than MSE.

On the other hand, we can define the regression sum of squares, also called the *explained sum of squares* as follows:

$$SS_{regression} = \sum_{n=1}^{N} (y(x_n) - y_{mean})^2 \tag{2.11}$$

Note that both the total sum of squares SS_{total} and regression sum of squares $SS_{regression}$ are defined relative to the mean of the sampled or observed predictor values. For simple linear regression models as defined in Eq. (2.1), the following relationships hold:

$$RSS + SS_{regression} = SS_{total} \tag{2.12}$$

$$R^2 = \frac{SS_{regression}}{SS_{total}} = \frac{SS_{regression}/N}{SS_{total}/N} \tag{2.13}$$

2.6.3 L_1 and L_2 Norms

Given a vector $\boldsymbol{x} = (x_1, \ldots, x_n)$ in an n-dimensional space, the general definition of a p-norm is given by

$$||\boldsymbol{x}||_p = \left(\sum_{i=1}^{n} |x_i|^p\right)^{1/p} \qquad (2.15)$$

Now for $p = 1$, we would have

$$||\boldsymbol{x}||_1 = \sum_{i=1}^{n} |x_i| \qquad (2.16)$$

And for $p = 2$, we would have

$$||\boldsymbol{x}||_2 = \left(\sum_{i=1}^{n} |x_i|^2\right)^{1/2} \qquad (2.17)$$

Eq. (2.16) is called 1-norm and Eq. (2.17) is called 2-norm. To explain what they mean, let's use Figure 2.13, which shows two vectors (p, q) in an n-dimensional space. The 1-norm distance d_1 measures the sum of the absolute distance of the two vectors in each dimension, namely, the two blue line segments, as follows:

$$d_1(\boldsymbol{p}, \boldsymbol{q}) = ||\boldsymbol{p} - \boldsymbol{q}||_1 = \sum_{i=1}^{n} |p_i - q_i| \qquad (2.18)$$

The d_1 distance is also called taxicab distance, similar to how a taxicab drives through blocks in a city. It is also known as rectilinear distance, L_1 distance or l_1 norm, city block distance, Manhattan distance.

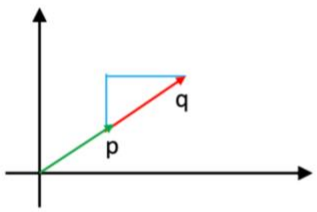

Figure 2.13 Difference between L_1 distance (l_1 norm) and L_2 distance (l_2 norm).

The 2-norm distance d_2 measures the squared root of the sum of the squared difference of the two vectors in each dimension, namely, the distance of the red line segment, as follows:

$$d_2(\boldsymbol{p}, \boldsymbol{q}) = ||\boldsymbol{p} - \boldsymbol{q}||_2 = \left(\sum_{i=1}^{n} (p_i - q_i)^2\right)^{1/2} \qquad (2.19)$$

The d_2 distance is also called Euclidean distance, which is the normal sense of the distance between two points. Now if you compare Eq. (2.19) with Eq. (2.3), the squared d_2 distance is just how RSS (residual sum of squares) is defined.

It is important to understand that the higher the norm index, the larger the effects of large values. Therefore, 2-norm distances are more sensitive to large-valued outliers than 1-norm distances. For this reason, outlier detection machine learning algorithms may use higher-indexed norms. On the other hand, if you want to minimize the effects of few outliers, instead of using the RMSE metric, you may consider using the Mean Absolute Error (MAE) metric as defined below:

$$MAE\ (X) = \frac{1}{N}\sum_{n=1}^{N} |y(x_n) - y_n| \qquad (2.20)$$

where N is the size of the sample data, $y(x_n)$ is the calculated value, and y_n the observed value.

Almost all the machine learning algorithms strive to minimize the errors among observed and calculated or predicted values, using one or another form of norm, as will be discussed with Ridge, LASSO and Elastic Net regularization techniques.

As is shown in Figure 2.10, the fuel economy prediction curve becomes very irregular with the polynomial degree of 8. This kind of phenomenon is known as overfitting as we mentioned earlier. With every overfitting scenario, the underlying fact is that the model is trying to fit the noise of the data or chase the pattern of the noise in the data, which apparently is not what the model is supposed to do and may result in poor generalization of the model to future unseen data. *Regularization* has turned out to be one of the effective measures for regulating overfitting. Next, we discuss three commonly used techniques for combating overfitting: *Ridge*, *LASSO* (*Least Absolute Shrinkage Selection Operator*) and *Elastic Net* regularizations.

Let's begin with the Ridge regularization first next.

2.6.4 Ridge regularization

The Ridge regularization, also known as Tikhonov regularization, attempts to minimize the following cost function:

$$J(\boldsymbol{w}) = \min_{w} ||\boldsymbol{wx} - \boldsymbol{y}||_2^2 + \alpha||\boldsymbol{w}||_2^2 \qquad (2.21)$$

where α is called a hyper-parameter, meaning that it's not a model parameter such as $\boldsymbol{w}$, and $\| \boldsymbol{w} \|_2$ is called the l_2 norm of the weight vector $\boldsymbol{w}$, as discussed in the preceding section. As is seen, the first term is just the RSS (residual sum of squares) as described in Eq. (2.3), while the second term, an l_2 term, is what the Ridge regularization is about: It helps control the effect of the weight vector $\boldsymbol{w}$. If $\alpha = 0$, then the effects of the Ridge regularization is zero; the larger the value of the α hyper-parameter, the larger the effects of the Ridge regularization on overfitting. We demonstrate this with our fuel economy machine learning project next.

2.6.5 LASSO regularization

Now, let's discuss the LASSO (*Least Absolute Shrinkage Selection Operator*) regularization. Instead of adding an l_2 norm term, the LASSO regularization adds an l_1 norm term to the cost function as shown below:

$$J(\boldsymbol{w}) = \min_{w} \frac{1}{2N} ||\boldsymbol{wx} - \boldsymbol{y}||_2^2 + \alpha||\boldsymbol{w}||_1 \qquad (2.23)$$

The general notion is that LASSO regularization forces certain coefficients to be set to zero by adding an l_1 norm term to the cost function, which effectively forces choosing a simpler model that does not include those coefficients, thus effectively suppresses overfitting. This is different from Ridge regularization, which shrinks but does not set those coefficients to zero. Perhaps this is because l_1 norm

2.6.6 Elastic Net regularization

Now, let's discuss the Elastic Net regularization. Instead of adding only an l_2 norm term like Ridge or only an l_1 norm term like LASSO to the cost function, Elastic Net regularization adds both an l_2 norm term and an l_1 norm term as shown below:

$$J(\boldsymbol{w}) = \min_{w} \frac{1}{2N} ||\boldsymbol{wx} - \boldsymbol{y}||_2^2 + \frac{\alpha(1-\rho)}{2}||w||_2^2 + \alpha\rho||\boldsymbol{w}||_1 \qquad (2.24)$$

As is seen, for $\rho = 0$, Elastic Net falls back to Ridge, while for $\rho = 1$, Elastic Net falls back to LASSO. As we discussed in the previous section about LASSO, when there are multiple features correlated with one another, LASSO tends to pick one of them. With Elastic Net, however, all correlated features will be picked up. Therefore, in general, one should favor Elastic Net over LASSO.

Now we are ready to formulate the bias-variance trade-off using the above three parameters by following a 3-step procedure as described in Bishop, 2006, *Pattern Recognition and Machine Learning*, p. 151:

- Compute the *average prediction* as follows:

$$\bar{y}(x_n) = \frac{1}{L}\sum_{l=1}^{L} y^{(l)}(x_n) \qquad (2.26)$$

- Compute the $(bias)^2$ as follows:

$$(bias)^2 = \frac{1}{N}\sum_{n=1}^{N}\{\bar{y}(x_n) - h(x_n)\}^2 \qquad (2.27)$$

- Compute the *variance* as follows:

$$variance = \frac{1}{N}\sum_{n=1}^{N}\frac{1}{L}\sum_{l=1}^{L}\{y^{(l)}(x_n) - \bar{y}(x_n)\}^2 \qquad (2.28)$$

3 Pattern Recognition with Classification

None.

4 Optimization and Search Illustrated with Logistic Regression

Apparently, optimization is very important for machine learning, as the primary goal of a machine learning task is to minimize the errors between target and predicted values or maximizing the likelihood of a certain measure while getting it done as quickly as possible. However, let's start with an abstract, simple function as shown below, and see how we can find its minima as quickly as possible:

$$f(\boldsymbol{\theta}) = 0.5\left(\theta_1^{\,2} - \theta_2\right)^2 + 0.5(\theta_1 - 1)^2 \qquad (4.1)$$

Here, we can take $\boldsymbol{\theta} = (\theta_1, \theta_2)$ as a vector with two components of θ_1, θ_2. This function, from (Aoki 1971, 106), has a global minimum of $f(\boldsymbol{\theta}) = 0$ at $\theta_1 = \theta_2 = 1$, as can be verified easily. However, it's not a convex function, as at the plane of $\theta_1 = 0$, $f(\theta_2) = 0.5\theta_2^{\,2} + 0.5$ has a local minimum of 0.5 at $\theta_2 = 0$.

Next, let's see how we can find its global minimum of $f(\boldsymbol{\theta}) = 0$ at $\theta_1 = \theta_2 = 1$.

4.1.1 Optimization technique I: Batch gradient descent

By examining Figure 4.1, we notice that we would reach a minimum if we keep pushing along the steepest slope, commonly known as *gradient search* or *gradient descent*. Since this function represents a known distribution of f over a parameter space of (θ_1, θ_2), we can easily calculate its gradient vector of $\boldsymbol{g} = (g_1, g_2)$ as follows:

$$g_1 = \frac{\partial f}{\partial \theta_1} = 2\left(\theta_1^2 - \theta_2\right)\theta_1 + (\theta_1 - 1) \qquad (4.2a)$$

$$g_2 = \frac{\partial f}{\partial \theta_2} = \theta_2 - \theta_1^2 \qquad (4.2b)$$

The step size for updating the parameters to get closer and closer to the minimum point step-by-step is called *learning rate*, typically denoted with the Greek letter η, which is applied as follows from step k to step $k + 1$:

$$\theta_i^{(k+1)} = \theta_i^{(k)} - \eta g_i \qquad (i = 1, 2) \qquad (4.3)$$

This is called *line minimization* or *line search* as the learning rate appears in a linear fashion in Eq. (4.3)

Next, let's demonstrate how this gradient decent optimization technique works with this specific example. Before we start, we need to consider a few things as listed below:

- **Start point**: We arbitrarily choose to start from $\theta_1 = 0, \theta_2 = 2.5$.
- **Learning rate**: Let's choose a fixed, constant learning rate of $\eta = 0.2$ to begin with.
- **Stop criterion**: Let's choose $\varepsilon = 0.001$ as the destination near the true convex, which is absolute 0. We compare the absolute value of $\eta|g_1 + g_2|$ with ε and stop if the former becomes smaller than the latter, which means that the function has reached the bottom where gradients have become approximately *flat*.
- **Visualization**: We want to draw a 2D contour plot first to show all concentric constant-value contours, and then add the trace of how the global minimum is reached step-by-step.

4.1.3 Learning scheduler

As is seen, learning rate plays a very important role in gradient descent optimization. Too small learning rates may cause a delayed convergence to an optimum (minimum or maximum) while too large learning rates may cause instabilities with reaching the final optimum. Therefore, very often a dynamic learning schedule is desirable for a machine learning model training process. Learning scheduling is also referred to as simulated annealing, similar to cooling down a molten metal in the process of annealing in metallurgy.

In this section, we experiment with a simple learning scheduler composed as follows:

$$\eta = \frac{\eta_0}{1 + \alpha k} \quad (k = 0, 1, \ldots) \qquad (4.4)$$

where η_0 is the initial, constant learning rate, k the step counter starting from 0, and α a strength parameter that controls how much impact the scheduler would apply. The above scheduler is implemented in the script named `gradient_descent_learning_schedule.py`, adapted from the previous script of `gradient_descent_stochastic.py`.

4.1.4 Heavy ball method

The heavy ball method for mitigating the zig-zag behavior of gradient descent, described in (Bertsekas, 1999), works by adding a momentum term to the parameter search iteration as follows:

$$\boldsymbol{\theta}_{k+1} = \boldsymbol{\theta}_k - \eta_k \boldsymbol{g}_k + \mu_k(\boldsymbol{\theta}_k - \boldsymbol{\theta}_{k-1}) \qquad (0 < \mu_k < 1) \quad (4.5)$$

That is, it adds the difference for the preceding step, with another parameter introduced to control the impact of the momentum term. To see how effective that momentum term is, I modified the previous script `gradient_descent_learning_schedule.py` and renamed it `gradient_descent_momentum.py`, with that momentum added while having SGD removed. Figure 4.5 shows the results from two runs, one with $\mu = 0.5$ (left) and the other with $\mu = 0.8$ (right), respectively. As is seen, the efficacy with the heavy ball method is huge with both $\mu = 0.5$, and $\mu = 0.8$, especially in the latter case that the zig-zag behavior has been completed eliminated within the inner most contour.

Let's understand how the conjugate gradient search works. We start by assuming that we want to use a line search method to arrive at the optimal parameters set for a given optimization function $f(\theta)$ as follows:

$$f(\boldsymbol{\theta}) = f(\boldsymbol{\theta} + \eta \boldsymbol{g}) \qquad (4.6)$$

Here, we consider all 1D arrays as row vectors. Now let's expand the above function using the Taylor series expansion as follows:

$$f(\boldsymbol{\theta} + \eta \boldsymbol{g}) \approx f(\boldsymbol{\theta}) + \eta \boldsymbol{g} \left(\frac{d}{d\eta} f(\boldsymbol{\theta} + \eta \boldsymbol{g}) \right) \Big|_{\eta=0} + \frac{\eta^2}{2} \boldsymbol{g}.\boldsymbol{g}^T \left(\frac{d^2}{d\eta^2} f(\boldsymbol{\theta} + \eta \boldsymbol{g}) \right) \Big|_{\eta=0} + \cdots \quad (4.7)$$

Now, differentiating the above equation with respect to η would give us:

$$\left(\frac{d}{d\eta} f(\boldsymbol{\theta} + \eta \boldsymbol{g}) \right) \approx \boldsymbol{J}(\boldsymbol{\theta}) \boldsymbol{g}^T + \eta \boldsymbol{g}^T \boldsymbol{H}(\boldsymbol{\theta}) \boldsymbol{g} \quad (4.8)$$

where $\boldsymbol{J}(\boldsymbol{\theta})$ and $\boldsymbol{H}(\boldsymbol{\theta})$ are the *Jacobian* vector (all first order partial derivatives) and *Hessian* matrix (all second order partial derivatives) of the parameter vector $\boldsymbol{\theta}$, respectively.

Setting the right hand side of the above equation to zero would give us:

$$\eta = \frac{\boldsymbol{J}(\boldsymbol{\theta}) \boldsymbol{g}^T}{\boldsymbol{g}^T \boldsymbol{H}(\boldsymbol{\theta}) \boldsymbol{g}} \quad (4.9)$$

Now for the space $(\eta, \boldsymbol{g})$, we update η first and then the parameter vector as follows:

$$\theta_{k+1} = \theta_k + \eta_k g_k \quad (4.10)$$

The gradient vector $\boldsymbol{g}$ can be updated by calculating a quantity expressed as β_{k+1} first using the following formula known as the *Fletcher-Reeves* formula:

$$\beta_{k+1} = \frac{\boldsymbol{J}(\boldsymbol{\theta_{k+1}}) \boldsymbol{J}(\boldsymbol{\theta_{k+1}})^T}{\boldsymbol{J}(\boldsymbol{\theta_k}) \boldsymbol{J}(\boldsymbol{\theta_k})^T} \quad (4.11)$$

Then we can calculate the new gradient vector as follows:

$$\boldsymbol{g}_{k+1} = -\boldsymbol{J}(\boldsymbol{\theta_{k+1}}) - \beta_{k+1} \boldsymbol{J}(\boldsymbol{\theta_k}) \quad (4.12)$$

The iteration stops when the following criterion is met:

$$\boldsymbol{g}_k \boldsymbol{g}_k^T < \varepsilon^2 \quad (4.13)$$

4.3.1 Logistic function

The logistic function, or logit function, is an S-shaped sigmoid function, often denoted as σ (.), and is expressed as follows:

$$\sigma(t) = \frac{1}{1 + e^{-t}} \qquad (4.14)$$

Figure 4.10 shows the above logit function, developed by statistician David Cox in 1958. As is seen, it has some very interesting properties. For $t = 0$, its value is 0.5. For $t < 0$, its values are below 0.5, whereas for $t > 0$, its values are above 0.5. This makes it suitable as a binary classifier, as we can just take $t = h_w(x) = \mathbf{w}^T\mathbf{x}$, and if $\sigma(h_w(\mathbf{x})) < 0.5$, we take it as $y = 0$ or `False`, and if $\sigma(h_w(\mathbf{x})) \geq 0.5$, we take it as $y = 1$ or `True`. Here, **w** and **x** are parameter and input vectors, respectively. This kind of delicacy can

be further expressed mathematically as follows, using the conventional hat-notation, which means *estimated*:

$$\hat{p} = h_w(\boldsymbol{x}) = \sigma(\mathbf{w}^T\boldsymbol{x}) \qquad (4.15a)$$

$$\hat{y} = \begin{cases} 0, & if\ \hat{p} < 0.5 \\ 1, & if\ \hat{p} \geq 0.5 \end{cases} \qquad (4.15b)$$

4.3.2 Cost function for logistic function

In order to optimize the logistic function based binary classifier, we need a cost function or loss function for it. We might get some kind of heuristics by taking the inverse of the logistic function shown in Eq. (4.15a) as follows:

$$\boldsymbol{w}^T\boldsymbol{x} = \log(p) - \log(1 - p) \qquad (4.16)$$

Based on the behavior of the –log(p) function and –log(1–p) function as shown in Figure 4.11, the logistic regression cost function is known as *log loss* and is defined as follows:

$$J(w) = -\frac{1}{N}\sum_{i=1}^{N}[y_i \log(\hat{p}_i) + (1 - y_i)\log(1 - \hat{p}_i)] \qquad (4.17)$$

Note that we have to put y_i and $(1–y_i)$ for those two log terms in Eq. (4.17), since y_i is a binary variable and takes either the value of 0 or 1. That way, those two terms in Eq. (4.17) would be isolated from each other.

4.3.5 Softmax regression

Softmax regression is just a generalization of the binary logistic regression to the multiclass or multinomial logistic regression. The generalization is done by generalizing the fitting of $\mathbf{w}^T\mathbf{x}$ to

$$\mathbf{s}_k(\mathbf{x}) = \mathbf{w}_k{}^T\mathbf{x} \tag{4.18}$$

where k represents the k^{th} class to be classified, and all parameter vectors ($\mathbf{w}_k$) would constitute a parameter matrix $\boldsymbol{W}$.

Now, the logistic probability for each class is normalized against all classes and takes the form of

$$\hat{p}_k = \sigma(s(\boldsymbol{x}))_k = \frac{\exp(\boldsymbol{s}_k(\mathbf{x}))}{\sum_{i=1}^{K}\exp(\boldsymbol{s}_i(\mathbf{x}))} \tag{4.19}$$

Since softmax is a multiclass classifier, its cost function takes the form of

$$J(\boldsymbol{W}) = -\frac{1}{N}\sum_{i=1}^{N}\sum_{k=1}^{K} y_{k,i}\log(\hat{p}_{k,i}) \tag{4.20}$$

Now, if we set $K = 2$ for two classes in the above equation, we would get exactly the same equation as shown in Eq. (4.17) for the binary logistic regression model.

Since the *Iris* dataset has three species of flowers, we can try the *softmax* multiclass classification with it and see how it would sort out those three different types of flowers. Since we want to see the decision boundaries, we can choose two features a time for all three classes. Listing 4.4 shows the script named `logistic_regression_model_softmax.py` that achieves the above purpose. It works as follows:

- Since the fifth column of the *Iris* dataset is labeled literally, we need to convert all values for that column into numerical numbers such as [0, 1, 2] corresponding to the three different specifies. This is accomplished at line 4 using *pandas*' `DataFrame` function of `replace`.
- Since we want to do multinomial classification, we have to set the `multi_class` attribute to "`multinomial`" specifically as shown at line 8. In addition, not all solvers are supported for multinomial classification, so we have to set a solver manually, such as "`lbfgs`" as shown at line 8.
- Finally, note that the `predict_proba` function at line 12 outputs probabilities corresponding to each of all three classes, while the `predict` function at line 13 outputs the values for each instance, based on the probabilities for each class. Listing 4.5 shows a partial output of these two functions to help clarify further.

4.4 KULLBACK-LEIBLER DIVERGENCE (CROSS ENTROPY)

When we talk about the logit or Softmax regression, we are actually comparing the probabilistic distributions of different classes, where the largest probability is assigned to the class of the sample it belongs to. In fact, there is a formal theoretical framework that defines the dissimilarity of the two probability distributions, p and q, which is known as the *Kullback Leibler divergence* (KL divergence) or *relative entropy*. The KL divergence is defined as follows:

$$KL(p\|q) \triangleq \sum_{k=1}^{K} p_k \log \frac{p_k}{q_k} \tag{4.21}$$

Here the two vertical bars mean *w.r.t.* (with respect to), and the symbol $\triangleq$ means "*defined as*." Obviously, if the two distributions p and q are exactly the same at each data point, then the associated KL divergence is zero.

What is interesting is that we can rewrite the above equation into the following form:

$$KL(p\|q) = \sum_{k} p_k log\, p_k - \sum_{k} p_k log\, q_k = -H(p) + H\,(p,q) \tag{4.22}$$

where H(p, q) is called the cross entropy, defined as:

$$H(p,q) \triangleq -\sum_{k} p_k log\, q_k \tag{4.23}$$

Now compare Eq. (4.23) with Eq. (4.17) in the case of binary classification. Since $p = 1 - q$, we see that minimizing the loss for the logistic regression is equivalent to maximizing the cross entropy between the two classes. This is why the logit and softmax regression models are also known as *MaxEntropy* models.

5 Rule-Based Learning: Decision Trees

He assumed that a collection C (or in our modern term, a dataset) contains p objects of class P and n objects of class N, then:

1. *Any correct decision tree for C will classify objects in the same proportion as their representation in C. An arbitrary object will be determined to belong to class P with probability p/(p + n) and to class N with probability n/(p + n).* This is equal to saying that it's probability based, similar to the logic regression we discussed in the previous chapter, but he did not use logistic function at all; otherwise, his systems would not be referred to as "*decision trees*."
2. *When a decision tree is used to classify an object, it returns a class. A decision tree can thus be regarded as a source of a message 'P' or 'N', with the expected information needed to generate this message given by*

$$I(p,n) = -\frac{p}{p+n}log_2\frac{p}{p+n} - \frac{n}{p+n}log_2\frac{n}{p+n} \tag{5.1}$$

Next, he detailed his algorithm for building a decision tree that begins with an attribute, say A, as the root of a decision tree, which has a set of values $\{A_1, A_2, \ldots, A_i, \ldots, A_v\}$. Then, C will be partitioned into $\{C_1, C_2, \ldots, C_i, \ldots, C_v\}$ where C_i contains those objects in C that have value A_i of A. Once again, assuming C_i contains p_i objects of class P and n_i of class N, the expected information required for the sub-tree for C_i is $I(p_i, n_i)$. The expected information required for the tree with A as root is then obtained as the weighted average

$$E(A) = \sum_{i=1}^{V}\frac{p_i + n_i}{p+n}I(p_i, n_i) \tag{5.2}$$

where the weight for the i^{th} branch is the proportion of the objects in C that belong to C_i. The information gained by branching on A is therefore

$$gain(A) = I(p,n) - E(A) \tag{5.3}$$

5.2.1 Gini impurity

The Gini impurity is a measure of the impurity of a binary tree node in terms of the proportions of the other classes against the total number of samples represented by that node. It is defined as follows:

$$G_i = 1 - \sum_{k=1}^{K} p_{i,k}^2 \tag{5.4}$$

where $p_{i,k}$ is the portion of the other classes relative to the class that node i represents. Let's use an example to help explain. Let's say we are dealing with a classification problem that has three classes of $[C_1, C_2, C_3]$. Let node i represent class C_2. The samples assigned to this node are [0, 90, 10]. Then the Gini impurity for this node is (0 + 10)/(0 + 90 + 10) = 0.1, e.g., only 10% of samples in this node are non-C_2 classes.

However, we want to use a program to help us divide the input space. We give the program both the data and the Gini impurity measure as the guide line that we would want to split the nodes by minimizing the Gini impurity for a node until it had bottomed to a leaf node. Specifically, with the Gini impurity defined quantitatively in Eq. (5.4), we can now define the loss or cost function for a node. It is defined as follows:

$$J(x_i, r(x_i)) = p_{left} G_{left} + p_{right} G_{right} \qquad (5.5)$$

where x_i is an attribute, $r(x_i)$ is a rule defined with a condition like $x_i < const$, p_{left} and p_{right} are the sample ratios for left node and right node, respectively, with $p_{left} + p_{right} = 1$, and G_{left} and G_{right} are the Gini impurities for the left node and right node, respectively. The pair $(x_i, r(x_i))$ seems to be the *key* to *inducing rule by data*, while the right-hand side of Eq. (5.5) just works for the left hand. This now becomes a standard optimization and search problem, which may vary from implementation to implementation.

Next, let's check the entropy and Gini impurity numbers shown in Figure 5.2. Let's look at the left chart for the entropy number of 1.0 in the middle white box first. According to Eq. (5.1), we have the entropy at node i defined as

$$H_i = -\sum_{k=1}^{K} p_{i,k} log_2 p_{i,k} = -\frac{0}{100} log_2 \frac{0}{100} - \frac{50}{100} log_2 \frac{50}{100} - \frac{50}{100} log_2 \frac{50}{100} = 1.0$$

Then, according to Eq. (5.4), the Gini impurity for the same middle white box node in Figure 5.2 is

$$G_i = 1 - \sum_{k=1}^{K} p_{i,k}^2 = 1 - \left(\frac{0}{100}\right)^2 - \left(\frac{50}{100}\right)^2 - \left(\frac{50}{100}\right)^2 = 0.5$$

Both the entropy number and the Gini impurity number we arrived at match the corresponding numbers shown in the middle white boxes of Figure 5.2.

6 Instance-Based Learning: Support Vector Machines

$$y = \boldsymbol{w}^T\boldsymbol{x} + b \tag{6.1}$$

where $\boldsymbol{w}$ may represent a rotation while $\boldsymbol{b}$ may represent a translation.

Now for a binary classifier, we can define two lines, one by $\boldsymbol{w}^T\boldsymbol{x} + b = 1$, and the other by $\boldsymbol{w}^T\boldsymbol{x} + b = -1$. We could have used $\boldsymbol{w}^T\boldsymbol{x} + b = \pm c$ instead of $\boldsymbol{w}^T\boldsymbol{x} + b = \pm 1$, but it doesn't matter, because we can always scale input to make c = 1. We now have the conditions for the support vectors as follows:

$$support\ vectors: \begin{cases} \boldsymbol{w}^T\boldsymbol{x}_{s.v.} + b = 1, & one\ class \\ \boldsymbol{w}^T\boldsymbol{x}_{s.v.} + b = -1, & the\ other\ class \end{cases} \tag{6.2}$$

We can take $y = 0$ for one class and $y = 1$ for the other class, accordingly. Note that we have made an assumption that we are dealing with a 2D input space. For a more generic case, such as in an n-dimensional input space, the support vectors will lie on a $(n-1)$-dimensional hyper-plane or hyper-surface.

Now let's look at the implication of the following condition:

$$\boldsymbol{w}^T\boldsymbol{x} + b = 0 \tag{6.3}$$

The above condition defines the center line that satisfies the $\mathbf{w}^T\mathbf{x}_{s,v} + b = \pm 1$ conditions. Therefore, $\mathbf{w}$ is a normal vector to the hyper-plane. Since in geometry, a normal is an object such as a line or vector that is perpendicular to a given object, the parameter $b/\|\mathbf{w}\|$ determines the offset of the support vector hyper-planes, and it is in fact half of the distance between the two supporting hyper-planes. Now from the cost or loss perspective, we would always like to maximize that distance, or minimize $\|\mathbf{w}\|$, since that distance, which is also called *margin*, is inversely proportional to $\|\mathbf{w}\|$. This tells us how we can abstract linear SVM classification as an optimization problem, namely,

$$\underset{w,b}{\text{minimize}} \frac{1}{2}\boldsymbol{w}^T\boldsymbol{w} \tag{6.4}$$

subject to

$$y_i(\boldsymbol{w}^T \cdot \boldsymbol{x} + b) \geq 1, \quad (i = 1, 2\ for\ a\ binary\ classifier) \tag{6.5}$$

Eqs. (6.4) and (6.5) define a quadratic programming problem, which is elaborated more next.

6.1.4 Hard margins versus software margins: The primal problem

Eqs. (6.4) and (6.5) define hard margins for a linearly separable dataset, where it is possible to find margin support lines, within which there would be no instances. Such margins are called *hard* margins. When a dataset is not linearly separable, we have to allow some instances to fall within the two margin lines. In such cases, margins become *soft* margins in the sense that they only approximate the boundaries for the classified classes. A soft margin problem is said to be a *primal problem*, defined by the following optimization problem (http://scikit-learn.org/stable/modules/svm.html#svm-classification) :

$$\min_{w,b,\zeta} \frac{1}{2} w^T w + C \sum_{i=1}^{N} \zeta_i \tag{6.6}$$

$$subject\ to\ y_i(\boldsymbol{w}^T \boldsymbol{x_i} + b) \geq 1 - \zeta_i, \quad \zeta_i \geq 0, \qquad i = 1, \ldots, N \tag{6.7}$$

Here, each symbol is defined as follows:

- $\boldsymbol{x}_i$ is a binary training vector in two classes in a dataset of size N;
- y_i is a binary target variable;
- ζ_i (pronounced *zeta-i*) is a so-called *slack* variable for each instance that measures how much the instance is allowed to penetrate across the margin line toward the other side;
- C is a hyper-parameter that further adjusts the effect of the slack variable ζ_i.

In fact, $\boldsymbol{x}_i$ in Eq. (6.7) can be replaced with a more generic function $\phi(\boldsymbol{x}_i)$, which would result in a whole chain of kernel functions that can be used to solve non-linear classification problems with SVM. This will be further explained when we discuss the derived *dual* problems in the section of nonlinear SVMs.

Another term known as the hinge loss function is introduced to cope with non-linearly separable datasets, as discussed next.

6.1.5 Hinge loss functions

To classify non-linearly separable datasets with the linear SVM models, the following hinge loss function is introduced:

$$f_{hinge} = \max(0, 1 - y_i(\boldsymbol{w^T} \cdot \boldsymbol{x} + b)) \tag{6.8}$$

The above hinge function implies that if the constraint expressed in Eq. (6.5) is satisfied, that is, all instances for that class lie on the correct side of the margin, then the hinge function will be zero. For those instances on the wrong side of the margin, the hinge function's value is proportional to the distance from the margin.

Taken the hinge loss function into account, the cost function now takes the form of

$$J(w) = \lambda \boldsymbol{w}^T \boldsymbol{w} + \left[\frac{1}{N} \sum_{i=1}^{N} max(0, 1 - y_i(\boldsymbol{w}^T \cdot \boldsymbol{x} + b)) \right], \quad (6.9)$$

where the parameter λ determines the degree of the soft margin that may contain instances of non-linearly separable classes. On the other hand, the parameter λ acts like a regularization parameter that can be used to control the margin width proportional to $\mathbf{w}^T\mathbf{w}$, as we will see next with an example.

6.3 NONLINEAR SVMS

From the previous examples, we see that margin lines are all straight lines. When the boundaries that separate two different classes are curved or nonlinear, the linear SVC model we discussed would no longer work. We have to seek new approaches to solving non-linear boundary problems. And the keys are with the definition of the dual problem and the concept of kernels, as discussed next.

6.3.1 Lagrangian dual problem

Now let's rewrite the hard margin optimization problem expressed in Eqs. (6.4) and (6.5) as follows:

$$\underset{w,b}{\text{minimize}} \frac{1}{2} \boldsymbol{w}^T \boldsymbol{w},$$

subject to

$$y_i(\boldsymbol{w}^T \cdot \boldsymbol{x} + b) - 1 \geq 0, \quad (i = 1, 2, \ldots, N)$$

As is seen, we just re-arranged the second equation by moving "1" over to the left of the ≥ sign. Now, this can easily be arranged into an optimization problem, using the Karush-Kuhn-Tucker (KKT) multipliers, as follows:

$$L(w, b, \mu) = \frac{1}{2}\boldsymbol{w}^T\boldsymbol{w} - \sum_{i=1}^{N} \mu_i[y_i(\boldsymbol{w}^T \cdot \boldsymbol{x} + b) - 1], \quad (6.10)$$

$$subject\ to\ \mu_i \geq 0\ for\ i = 1, 2, \ldots, N \quad (6.11)$$

Now the original condition has become part of the function (L) to be optimized with the constraints that all its coefficients (μ_i) must be ≥ 0, where the subscript i denotes the i^{th} instance of the data.

Next, to minimize L in Eq. (6.10), we can take the partial derivatives of L against $\boldsymbol{w}$ and b and set them to zero, which would give us:

$$\hat{\boldsymbol{w}} = \sum_{i=1}^{N} \mu_i y_i \boldsymbol{x}_i \quad (6.12)$$

$$with \quad \sum_{i=1}^{N} \mu_i y_i = 0 \quad (6.13)$$

For support vectors, the condition $y_i(\boldsymbol{w}^T \cdot \boldsymbol{x} + b) - 1 = 0$ holds true, so b can be estimated by averaging over all support vectors as:

$$\hat{b} = \frac{1}{n_{s,v}} \sum_{\substack{i=1 \\ \mu_i > 0}}^{N} [\mathbf{1} - y_i(\hat{\boldsymbol{w}}^T \cdot \boldsymbol{x})] \quad (6.14)$$

Now by substituting Eq. (6.12) for $\hat{\boldsymbol{w}}$ and (6.14) for $\hat{b}$ back into Eq. (6.10), we have:

$$L(\hat{\boldsymbol{w}}, \hat{b}, \mu) = \frac{1}{2}\sum_{i=1}^{m}\sum_{j=1}^{m} \mu_i\mu_j y_i y_j (\boldsymbol{x}_i^T \cdot \boldsymbol{x}_j) - \sum_{i=1}^{m} \mu_i \quad (6.15)$$

$$with\ \mu_i \geq 0 \qquad (i = 1, 2, \ldots, m)$$

Eq. (6.15) is the *dual* problem of the original *primal* problem expressed in Eq. (6.4).

By inspecting Eq. (6.15), we see the term $(\boldsymbol{x}_i^T \cdot \boldsymbol{x}_j)$ at its core, double multiplied by the target values $(y_i y_j)$ and KKT multipliers $(\mu_i \mu_j)$, and summed twice. That term is called a *kernel*, which is an *inner product* of two vectors. Now, suppose we want to fit a boundary with a second degree polynomial, then, we can just use the second degree polynomial kernel of $(\boldsymbol{x}_i^T \cdot \boldsymbol{x}_j)^2$ in Eq. (6.15). This is the so called *kernel trick*, as we do not have to transform the input variables first and then combine their polynomial terms – everything we need is already given to us by that inner product of the two vectors. Furthermore, we can generalize the above kernel into a more generic form of:

$$k(\boldsymbol{x}, \boldsymbol{x}') = \phi^T(\boldsymbol{x}) \cdot \phi(\boldsymbol{x}') \qquad (6.16)$$

Here, we care more about the inner product of the function ϕ with its transpose more than the actual form of the function ϕ itself. A few more commonly used kernel functions are shown in Table 6.1, such as *linear*, *polynomial*, *Gaussian radial base function* and *sigmoid*. Each of these functions maps a problem from its original input space to a new space, where non-separable data in the original space may become separable in the new space.

Table 6.1 Commonly used kernel functions

Kernel	Function $k(\boldsymbol{x}, \boldsymbol{x}')$
Linear	$k(\boldsymbol{x}, \boldsymbol{x}') = \boldsymbol{x}^T \cdot \boldsymbol{x}'$
Polynomial	$k(\boldsymbol{x}, \boldsymbol{x}') = (\gamma \boldsymbol{x}^T \cdot \boldsymbol{x}' + c)^n$
Gaussian radial base function (RBF)	$k(\boldsymbol{x}, \boldsymbol{x}') = exp(-\gamma \|\|\boldsymbol{x} - \boldsymbol{x}'\|\|)^2$
Sigmoid	$k(\boldsymbol{x}, \boldsymbol{x}') = tanh(\gamma \boldsymbol{x}^T \cdot \boldsymbol{x}' + c)$

In addition, if we take $\boldsymbol{x}$ as the training instances and $\boldsymbol{x}'$ as unseen new instances, then a kernel function essentially measures the proximity of a new instance to the training instances, which allows us to make predictions on the unseen new instances. This is different from the parametric paradigm of machine learning that once a model is trained, the training data used to train the model are discarded. Because of this, SVMs belong to the *instance-based machine-learning* paradigm.

So what is a positive definite kernel? Its generic definition is as follows:

Let $X = \{x_1, x_2, \ldots, x_n\}$ be a non-empty sample set. A symmetric function $k(x_i, x_j)$ is called a *positive definite* (p.d.) kernel on X if the following relation holds true for any $\{x_1, x_2, \ldots, x_n\}$ and $\{\mu_1, \mu_2, \ldots, \mu_n\}$:

$$\sum_{i=1}^{n} \sum_{j=1}^{n} \mu_i \mu_j k(\boldsymbol{x}_i, \boldsymbol{x}_j) \geq 0 \qquad (6.17)$$

This theorem explains why linear SVMs can solve nonlinear problems with the help of kernels.

Next, let's use an example to help illustrate how SVM nonlinear models work.

7 Random Forests and Ensemble Learning

7.4.1 Hard voting versus soft voting

Hard voting is essentially a majority voting mechanism. Let's use the MNIST dataset as an example. If we use n individual classifiers and more than half of them predict a hand-written digit as, say, "9", then that digit is taken as "9". In case it is a tie, then it's broken more or less arbitrarily, such as by making a decision based on the ascending sort order. For example, if half of the classifiers say a hand-written digit is a "4" and the other half say it's a "9", then it is assigned as a "4", as "4" is ahead of "9" in the ascending sort order.

On the other hand, soft voting uses the weighted average probability to decide the class of an instance, or mathematically, it is decided as follows:

$$c_k = \underset{k}{\operatorname{argmax}} \sum_{i=1}^{n} w_i p_i \qquad (7.1)$$

Here, w_i and p_i are the weight and probability of an instance associated with each classifier, respectively, and c_k is the class that has the largest value of the weighted average from all classifiers.

8 Dimensionality Reduction

8.1.1 Eigenvalues and eigenvectors

Eigenvalues and eigenvectors are foundational mathematical concepts for understanding principal component analysis (PCA) and thus dimensionality reduction. Therefore, this section briefly covers these two concepts instead of jumping to PCA immediately. However, if you find that the coverage here is not sufficient enough, please feel free to augment it by looking up more formal, detailed texts.

First, we start with the concept of transformation, which means applying some kind of operations so that the original entity is transformed into another form that is different from the original. Let's say we have a vector $\boldsymbol{v}$, which is transformed by an abstract function T in the following form:

$$T(\boldsymbol{v}) = \lambda \boldsymbol{v}, \qquad (8.1)$$

where $\boldsymbol{v}$ is known as a vector and λ is a scalar, which has a few different names such as the *eigenvalue*, *characteristic value*, or *characteristic root* associated with the eigenvector $\boldsymbol{v}$. Apparently, this is a *linear* transformation since the original vector $\boldsymbol{v}$ is transformed into itself with a constant λ.

Now we want to get more specific about the transformation T. Since we want to deal with high-dimensional data, let's assume that T represents a matrix X so that

$$\boldsymbol{X}\boldsymbol{v} = \lambda\boldsymbol{v} \tag{8.2}$$

Of course, in this case, the dimensions of $\boldsymbol{X}$ and $\boldsymbol{v}$ must match with each other, namely, the second dimension of $\boldsymbol{X}$ must be the same as the dimension of $\boldsymbol{v}$, e.g., $(m\times n)$ and (n) for $\boldsymbol{X}$ and $\boldsymbol{v}$, respectively. Since $\boldsymbol{v}$ is an eigenvector as we explained with Eq. (8.1), the above Eq. (8.2) is known as the *eigenvalue equation* for the matrix $\boldsymbol{X}$.

Now, we can rewrite Eq. (8.2) as

$$(\boldsymbol{X} - \lambda\boldsymbol{I})\boldsymbol{v} = 0 \tag{8.3}$$

where I is an *identity matrix* that has 1's for all its diagonal elements and 0's for its off-diagonal elements.

Next, what are the solutions to Eq. (8.3)? The mathematician Gottfried Wilhelm Leibniz had given us the answer more than 350 years ago, that is, Eq. (8.3) has a non-zero solution $\boldsymbol{v}$ if and only if the determinant of the matrix $(\boldsymbol{X} - \lambda\boldsymbol{I})$ is zero, or

$$|\boldsymbol{X} - \lambda\boldsymbol{I}| = (\lambda_1 - \lambda)(\lambda_2 - \lambda)\ldots(\lambda_n - \lambda) = 0 \tag{8.4}$$

where those *linear* factors in the middle are called the *characteristic polynomi*al of $\boldsymbol{X}$, with the numbers $\lambda_1, \lambda_2, \ldots, \lambda_n$ called the roots of the polynomial and the eigenvalues of $\boldsymbol{X}$, which may be real but in general complex numbers.

So why are we so interested in those eigenvalues and eigenvectors? That's because we want to simplify our matrices into something that we can understand more easily. Then, what is the simplest form of a matrix? The answer is that diagonalized matrices are the simplest form we can have, as such matrices have all zeros except the diagonal elements, as discussed next.

8.1.2 Matrix diagonalization

Equipped with the knowledge of eigenvalues and eigenvectors, now we can easily understand how to diagonalize a matrix. Suppose a matrix $\boldsymbol{X}$ has n linearly independent eigenvectors $v_1, v_2, \ldots, v_n$ with associated eigenvalues $\lambda_1, \lambda_2, \ldots, \lambda_n$, where those eigenvalues do not need to be distinct, we can define a square matrix with those eigenvectors as follows:

$$V = [v_1 \quad v_2 \quad \ldots \quad v_n] \tag{8.5}$$

Since the eigenvectors as the column vectors of V are linearly independent, V is invertible, which leads to:

$$V^{-1}XV = \Lambda \tag{8.6}$$

where Λ is a diagonal matrix. This diagonalization is also called *eigen-decomposition* and a similarity transformation as it is a linear transformation in nature. In a more formal term, $\boldsymbol{X}$ and Λ represent the same linear transform expressed in two different bases. On the other hand, not all matrices are diagonalizable. If a matrix is not diagonalizable, it is said to be *defective*.

Next, we discuss matrix diagonalization applied to the singular value decomposition, which is one of the main methods for extracting principal components for PCA.

8.1.3 Singular value decomposition (SVD) theorem

For any finite-dimensional $m \times n$ real or complex matrix X, the singular value decomposition (SVD) theorem states that we can factorize the matrix X into the following form:

$$X_{m\times n} = U_{m\times m}\Sigma_{m\times n}V^T_{n\times n} \tag{8.7}$$

where U and V are unitary orthogonal matrices and satisfy the following conditions:

$$U^TU = I_{m\times m} \quad and \quad V^TV = I_{n\times n} \tag{8.8}$$

The columns of U are the left singular vectors, Σ has the same dimension as X and has singular values lying on the diagonal in descending order, and V^T has rows that are the right singular vectors. You might wonder what *singular values* are. This question can be answered by calculating X^TX as follows:

$$X^TX = (V\Sigma U^T)(U\Sigma V^T) = V\Sigma(U^TU)\Sigma V^T = V\Sigma^2V^T \tag{8.9}$$

Therefore, we have

$$\sqrt{X^TX} = |\Sigma| \tag{8.10}$$

which means that the singular values are simply the absolute values of the eigenvalues. Now the concepts of *eigenvalues* and *diagonalization* are unified in the single framework of the singular value decomposition as shown above.

The above equations also show how to diagonalize a matrix X. One can follow the below procedure:

- The left singular vectors as the column vectors of U are made up by the eigenvectors of XX^T.
- The right singular vectors as the column vectors of V are made up by the eigenvectors of X^TX.
- The singular values in the diagonal matrix Σ are square roots of eigenvalues of XX^T or X^TX.

From a geometric point of view, in fact, U and V represent rotations while Σ represents a scaling so that Σ is also known as a *scaling matrix*, where scaling is applied along each axis by the amount determined by each of the singular values lying on the diagonal. This is why the SVD is so helpful for PCA, as discussed next.

Given Eq. (8.7), we can calculate a metric known as the score matrix S as follows:

$$S = XV = U\Sigma V^T V = U\Sigma \tag{8.11}$$

where the score matrix S has the same dimension of $m \times n$ as the data matrix X. However, what's different is that each column of S is given by a corresponding left singular vector of X multiplied by the corresponding singular value of the diagonal matrix Σ, both of which are known by the SVD procedure we detailed above. Since the singular values of Σ lie on the diagonal in descending order and the eigenvectors of U are orthonormal to each other, we can truncate the score matrix S by keeping only the first l largest singular values and corresponding singular vectors to have a truncated score matrix S_l as follows:

$$S_l = U_l \sum_l = XV_l \tag{8.12}$$

The components of S_l correspond to the first l principal components of X in descending order, which are uncorrelated with each other as explained next.

8.1.5 Principal components analysis (PCA) versus covariance matrix

The covariance matrix C of the data matrix X is proportional to the components of $S^T S$ of its score matrix S, and the covariance between the two principal components, PC_j and PC_k, are defined as follows:

$$C(PC_j, PC_k) \propto S^T S = (Xv_j)^T (Xv_k) = v_j{}^T X^T X v_k = v_j{}^T \lambda_k v_k = \lambda_k v_j{}^T v_k \tag{8.13}$$

Since the eigenvectors of V are orthogonal with each other for $j \neq k$, the covariance between two different principal components shown in Eq. (8.13) is equal to zero; thus, there is no covariance between different principal components in the transformed space for the dataset, or in other words, the correlation or collinearity among different features have been removed after the transformation. This important property allows us to examine the variance projected onto each principal axis, which is another method to PCA, as discussed next.

8.1.6 Principal component analysis (PCA) by maximizing variance

The more formal, generic description for PCA is that *PCA maps data linearly from higher dimension to lower dimension by maximizing the variance of the data along axes in lower dimensional subspace*. In addition to the SVD procedure outlined in the previous section, it can also be achieved by constructing the covariance matrix of the data and computing the eigen vectors on the covariance matrix as mentioned above. The eigen vectors having the largest eigen values preserve the largest fraction of the variance of the original data, which makes them more effective and more efficient than the original entire dataset.

Now, let's see how we can put the above notions into a formal, quantitative framework mathematically. Let's say we want to project a data matrix X onto an orthogonal set of unitary vector basis of (v_1, v_2, …,

v_n), and we call the projected components as principal components, with principal component scores defined as the dot product of $\boldsymbol{x}_i$ and $\boldsymbol{v}_k$, similar to Eq. (8.11), as follows:

$$s_{k,i} = \boldsymbol{x}_i \cdot \boldsymbol{v}_k \tag{8.14}$$

where k represents the k^{th} principal component and i the i^{th} instance of the dataset. Here the principal component score $s_{k,i}$ apparently represents the variance of data out of a single instance projected on to a particular principal axis. Note that here we do not assume that we already know those principal axes. Instead, we want to find out the first and the largest principal component aggregated from the entire dataset, i.e.,

$$v_1 = \arg\max_{\|v\|=1}\left\{\sum_i s_{1,i}^2\right\} = \arg\max_{\|v\|=1}\left\{\sum_i (\boldsymbol{x}_i \cdot \boldsymbol{v})^2\right\} = \arg\max_{\|v\|=1}\{v^T X^T X v\} \tag{8.15}$$

Since this is the first principal component, it represents the largest principal component or the largest variance obtained with the given *argmax* condition.

For the subsequent principal components, first, we need to subtract the $k-1$ principal components from the entire matrix X as follows:

$$\hat{X}_k = X - \sum_{j=1}^{k-1} X v_j v_j^T \tag{8.16}$$

Then, we can find the k^{th} component, which extracts the maximum variance from the above new data matrix as follows

$$v_k = \arg\max_{\|v\|=1}\left\{v^T \hat{X}_k{}^T \hat{X}_k v\right\} \tag{8.17}$$

Now, we have obtained all components v_k and the full principal component matrix can be obtained with Eq. (8.14), which is the same as Eq. (8.11) obtained with the SVD. We can do the same dimensionality reduction as shown in Eq. (8.12).

This completes our theoretical coverage of PCA. I hope it is clear enough for you to follow. This coverage is provided based on my belief that it's always clearer and less ambiguous if we use math to understand the subjects we are interested in. If it turns out that it is not so easy, you can just remember that PCA decomposes a data matrix into a series of uncorrelated principal components lying on the diagonal in descending order. We don't need to do any matrix decomposition ourselves anyway, as such tasks are already standardized with various libraries such as the *sklearn*, as discussed next.

As the authors pointed out, the LLE differs from some of the previous methods such as the multidimensional scaling (MDS) and Isomap, which all compute embeddings that attempt to preserve pairwise distances: LLE recovers global nonlinear structure from locally linear fits. The key notion here is the locally linear patch of the manifold. The authors gave a method for how to reconstruct each data point on such patches by minimizing the reconstruction errors as a squared residual from the data point to its neighbors, which adds up the squared distances between all data points and their reconstructions as follows:

$$\varepsilon(W) = \sum_i \left| \vec{X}_i - \sum\nolimits_j W_{ij} \vec{X}_j \right|^2 \qquad (8.18)$$

9 Introduction to Artificial Neural Networks

Let c_i be any neuron with a threshold $ß_i > 0$, and let $c_{i1}, c_{i2}, \ldots, c_{ip}$ have respectively $n_{i1}, n_{i2}, \ldots, n_{ip}$ *excitatory synapses* upon it. Let $c_{j1}, c_{j2}, \ldots, c_{jq}$ have *inhibitory synapses* upon it. Let k_i be the set of the subclasses of $\{n_{i1}, n_{i2}, \ldots, n_{ip}\}$ such that the sum of their members exceeds θ_i. We shall then be able to write the action N_i as follows, in accordance with the assumptions mentioned above:

$$N_i(z_1) .\equiv. S\left\{\prod_{m=1}^{q} \sim N_{jm}(z_1) \cdot \sum_{\alpha \in k_i} \prod_{S \in \alpha} N_{iS}(z_1)\right\}, \qquad (9.1)$$

where the "Σ" and "∏"symbols are syntactical symbols for disjunctions (equivalent to logical *OR*) and conjunctions (equivalent to logical *AND*), respectively, which are finite in each case.

Now given that N in Eq. (9.1) represents *action*, we read from right to left as follows:

- First we do a logical *AND* (the right-most ∏) for all actions of each subclass member;
- Then we do a logical *OR* (Σ) for all outputs from the above step;
- Then we negate each inhibitory synapse's action and do a logical *AND* (the left-most ∏) with all of them;
- Finally, we get the action for the neuron by applying functor S.

McCulloch and Pitts came up with 12 configurations (let's call them *basic units*) for implementing their nets. Figure 9.1 shows the first four of them, which correspond to: (a) a pass-through (or an identity operation), (b) a logical *OR* that if at least one is *on* then the output is *on*, (c) a logical *AND* that the output is *on* only if both are *on*, and (d) a logical *AND* of the first neuron and the negation of the second neuron that is equivalent to the logical expression of $(S_1 \wedge \sim S_2)$, which means that the output is *on* only if one input is *on* and the other input is *off*.

However, I am most impressed by his quantitative formulation that the weight between two neurons increases if both activate simultaneously, and reduces if they activate separately. Furthermore, nodes that tend to be either both positive or both negative at the same time have strong positive weights, while those that tend to be opposite have strong negative weights. This seems to be already explained in the propositional logic that McCulloch and Pitts developed in 1943 as excitatory and inhibitory synapses, as shown in Figure 9.1. However, what was missing in 1943, as I mentioned at the end of the preceding section, is the following:

$$w_{ij} = x_i x_j \qquad (9.2)$$

So, given what you have learnt from part I of the text, you should have guessed what Eq. (9.2) is about. This equation describes the weight w_{ij} from neuron j to neuron i, which have their inputs of x_i and x_j, respectively. This is considered *pattern learning*, since the weights are updated after every experience or training example. There are two special cases here: (1) $w_{ij} = 0$ if $i = j$, which is called no reflexive connection condition, and (2) $w_{ij} = 1$ if the connected neurons have the same activation for a pattern with binary neurons that have activations of 0 or 1 as is the case with the logical calculus developed by McCulloch and Pitts in 1943.

$$w_{ij} = \frac{1}{n}\sum_{k=1}^{n} x_{ik}\, x_{jk} \qquad (9.3)$$

where w_{ij} is the weight averaged over all learning experiences or training samples.

In literature, the Hebb learning is also generalized to so-called linear neurons as:

$$y = \sum_{j} w_j\, x_i \qquad (9.4)$$

where y is the postsynaptic response. Or, in another form that formulates the change in the i^{th} synaptic weight w_i:

$$\Delta w_i = \eta x_i y \qquad (9.5)$$

where η is the learning rate.

First, an *XOR* operator gives a *true* output if one, and only one, of the inputs is *true*. This is demonstrated with Table 9.1 below. The other way to state an *XOR* operator is that the output is *true* if and only if the sum of the inputs is odd; or if the sum of the inputs is even, then the output is *false*. Thus, for A = B = 0 or 0 + 0 = 0 and A = B = 1 or 1 + 1 = 2, the output would be *false*, which explains why it is called exclusive *OR*.

On the other hand, an *XOR* operator can be defined as:

$$XOR = A\cdot\bar{B} + B\cdot\bar{A} = (A+B)(\bar{A}+\bar{B}) = (A+B)\cdot\overline{(A+B)} \qquad (9.6)$$

9.2.1 single layer perceptron network model

By examining Figure 9.3, Rosenblatt's perceptron configuration is a single layer perceptron in the sequence of the source *(S)*, *A*-units and *R*-units, where the source represents stimuli or inputs (x_i), the *A*-units serve as the source sets for the *R*-units by passing inputs they receive from the source to the *R*-units, and the *R*-units generate outputs by filtering *activations*. This whole process can be formulated by the following two equations:

$$a_j = \sum_{i=1}^{m} w_{ji} x_i + w_{j0} \tag{9.7}$$

and

$$r_j = heaviside(a_j) = heaviside(\sum_{i=1}^{m} w_{ji} x_i + w_{j0}) \tag{9.8}$$

In Eq. (9.7), x_i represents the i^{th} input variable from the i^{th} *A*-unit, w_{ji} is the connection *weight* parameter on the j^{th} *R*-unit between the j^{th} *R*-unit and the i^{th} *A*-unit, w_{j0} is called the *bias*, and a_j is the activation arrived at the j^{th} *R*-unit.

Eq. (9.8) is a Heaviside function, representing the response or output at the j^{th} *R*-unit, in the form:

$$heaviside(z) = \begin{cases} 0, & if\ z < 0 \\ 1, & if\ z \geq 0 \end{cases} \tag{9.9}$$

Now learning lies with how to update the weight parameters w_{ji} to correct wrong responses. Let's say the correct response for an input value of x_i is t_j, and the machine's response is r_j. Then, the error can be denoted as

$$\Delta r_j = r_j - t_j \tag{9.10}$$

Since this is supervised learning, the error would be zero if the response were the same as the known target value for the input in question. If the response were wrong, we would want to correct the connection weight parameter by reinforcing it with a change proportional to the error by the following amount

$$\Delta w_{ji} = -\eta \Delta r_j = -\eta(r_j - t_j) \tag{9.11}$$

where η is the learning rate as we already learnt from part I of this text. Let's say the correct response t_j = 1, and the machine's response r_j is indeed 1, then there is no need to correct or reinforce the connection weight parameter w_{ji} and Eq. (9.11) indeed gives 0. Now if the machine's response r_j is 0, instead, which is smaller than the correct response t_j = 1, then Eq. (9.11) would give a positive amount equal to η, which would be added to the connection weight parameter w_{ji} for the next training. This is how the connection weight parameter is reinforced. Similarly, if the correct response t_j = 0, and the machine's response r_j is 1, then Eq. (9.11) would give a negative amount equal to -η, which would be subtracted from the connection weight parameter w_{ji} for the next training. As you see, the key to understanding machine learning is to understand how connection weight parameters are forced to move toward the direction of smaller and smaller errors against the target values. Therefore, the number of training instances, the number of training iterations, and the learning rate all matter.

Now, compare this with Rosenblatt's perceptron as shown in Figure 9.3, we notice one obvious difference, which is the hidden layer. This is a very important differentiation, as we can even have more than one hidden layer. Besides, we can express it differently by generalizing and combining Eq. (9.7) and (9.8) into:

$$y_k(\boldsymbol{x}, \boldsymbol{w}) = f_2\left\{\sum_{j=1}^{n} w_{kj}^{(2)} f_1\left[\sum_{i=1}^{m} w_{ji}^{(1)} x_i + w_{j0}^{(1)}\right] + w_{k0}^{(2)}\right\} \tag{9.12}$$

We need to read the above equation inside out, which corresponds to the sequence of from left to right in Figure 9.8. We introduced two generic activation functions, f_1 and f_2, which used to be the *Heaviside* function in Rosenblatt's perceptron model. The forms of these two functions are not important here: all we need to know is that they are non-linear, differentiable functions, which is equal to saying that artificial neural networks are nonlinear, or linear artificial neural networks are rarely useful.

On the other hand, we can absorb the bias terms into the two sums, and re-write Eq. (9.12) into:

$$y_k(\boldsymbol{x}, \boldsymbol{w}) = f_2\left\{\sum_{j=0}^{n} w_{kj}^{(2)} f_1\left[\sum_{i=0}^{m} w_{ji}^{(1)} x_i\right]\right\}, \tag{9.13}$$

9.2.3 Error backpropagation algorithm

If there is one machine learning algorithm that you really need to understand well, what would it be? I would recommend the error backpropagation algorithm. This algorithm is considered one of the turning points for machine learning. Therefore, in this section, I'd like to help you understand it with an end-to-end mathematical derivation, which is really not that difficult.

We start by stating that machine learning is to some extent a trial-by-error business. It doesn't matter that initially the outcomes are far off. The importance is that a machine can learn from its mistakes by correcting the errors it has made and reduce the errors iteratively.

Now let's say we are dealing with a parametric nonlinear feed-forward network as discussed previously. Let's further state that we want to examine the errors that a machine makes at a particular step and assume that the total error $E(\boldsymbol{w}, \boldsymbol{x})$ from all data points or examples or instances or features is defined by:

$$E(\boldsymbol{w}, \boldsymbol{x}) = \sum_{n=1}^{N} E_n(\boldsymbol{w}, x_n), \tag{9.14}$$

where $E_n(\boldsymbol{w}, x_n)$ is the error from an arbitrary data point x_n. Let's look at the error at the last step of the output since that's when we know the error when the machine gives a prediction, right or wrong. Assume that the target value at the k^{th} output unit is t_{nk}, and the predicted value is y_{nk} corresponding to a prior hidden unit z_{ni}, then we can write the error function as

$$E_n = \frac{1}{2}(y_{nk} - t_{nk})^2, \tag{9.15}$$

where the predicted value $y_{nk} = y_k(x_n, w)$. The gradient of the error function expressed by Eq. (9.15) with respect to an arbitrary connection weight coefficient (hidden unit or output unit) would be:

$$\frac{\partial E_n}{\partial w_{ki}} = (y_{nk} - t_{nk})\frac{\partial y_{nk}}{\partial w_{ki}} = (y_{nk} - t_{nk})x_{ni}, \tag{9.16}$$

where we have assumed a linear dependence between y_{nk} and w_{ki} as follows:

$$y_{nk} = w_{jk}x_{nk}, \tag{9.17}$$

Since each unit computes an activation, similar to Eq. (9.7), we can define the activation at the k^{th} unit as

$$a_k = \sum_i w_{ki} z_i, \tag{9.18}$$

where we consider z_i the input from a hidden unit to the k^{th} unit in question to be generic. We also learnt from the previous section that an activation such as the one shown in Eq. (9.18) is transformed by a nonlinear function, say, $f(.)$, so that we have:

$$z_k = f(a_k). \tag{9.19}$$

This part is called *forward propagation* in the sense that the i-unit is prior to the k-unit and w_{ki} is the connection weight from the i-unit to the k-unit, and that we compute the activation a_k and apply the transformation at the k-unit according to Eqs. (9.18) and (9.19), respectively.

Now we can examine how to calculate the error gradient with respect to the activation a_k as expressed in Eq. (9.18). We start with Eq. (9.16) again, and we would have:

$$\frac{\partial E_n}{\partial w_{ki}} = \frac{\partial E_n}{\partial a_k} \frac{\partial a_k}{\partial w_{ki}}. \tag{9.20}$$

For convenience, we denote the first term on the right-hand-side of Eq. (9.20) with a single symbol as

$$\delta_k \equiv \frac{\partial E_n}{\partial a_k}. \tag{9.21}$$

The second term on the right-hand-side of Eq. (9.20) turns to be just as simple as, according to Eq. (9.18):

$$\frac{\partial a_k}{\partial w_{ki}} = z_i. \tag{9.20}$$

Now Eq. (9.20) can be re-written as:

$$\frac{\partial E_n}{\partial w_{ki}} = \delta_k z_i. \tag{9.22}$$

Now the error gradient $\frac{\partial E_n}{\partial w_{ki}}$ depends on how to compute δ_k. If the output $y_k = a_k$, i.e., no nonlinear transformation is applied to the activation or it is simply an identity function of $y_k = a_k$, then, δ_k is just the error:

$$\delta_k = y_k - t_k, \tag{9.23}$$

which is why δ's are referred to as *errors*. However, if the output activation function is nonlinear, such as:

$$y_k = \sigma(a_k), \tag{9.24}$$

Then, Eq. (9.23) would be:

$$\delta_k = (y_k - t_k)\sigma'(a_k), \tag{9.25}$$

which is equal to the *error* multiplied by the derivative of the activation function, and δ's are still related to errors.

Finally, we get to the part of the error back-propagation: how do we compute δ_i defined similar to Eq. (9.21) when i refers to a hidden unit prior to the output layer? To compute δ_i for a hidden unit labeled i, we apply the *chain rule* for partial derivatives as follows:

$$\delta_i \equiv \frac{\partial E_n}{\partial a_i} = \sum_k \frac{\partial E_n}{\partial a_k}\frac{\partial a_k}{\partial a_i} = \sum_k \delta_k \frac{\partial a_k}{\partial a_i} = f'(a_i)\sum_k w_{ki}\delta_k, \tag{9.26}$$

for which we have used

$$a_k = \sum_i w_{ki} z_i = \sum_i w_{ki} f(a_i), \tag{9.27}$$

and

$$\frac{\partial a_k}{\partial a_i} = f'(a_i) w_{ki}, \tag{9.28}$$

where other connection weights except w_{ki} do not contribute to the partial derivative.

Eq. (9.26) is called the *error back-propagation* as we back-propagated the errors from the output later to the prior hidden layer. If we have multiple hidden layers, Eq. (9.26) can be applied recursively.

This error back-propagation algorithm can be summed up as follows:

1. For a given input vector $\boldsymbol{x}_n$, compute the activation and activation function of a layer of hidden units, $(a_i, f(a_i))$, according to (9.18) and (9.19), with proper subscripts, in the forward direction until done at the output layer.
2. Compute the errors of δ_k for all output units using Eq. (9.23) or Eq. (9.25), depending on whether the output layer activation function is linear or not.
3. Then, back-propagate the errors from the output layer to the hidden layer recursively using Eq. (9.26).
4. Use an equation similar to Eq. (9.22) with proper indices to compute the error gradients.
5. For batch methods, aggregate the gradients for all data points using the following equation:

$$\frac{\partial E}{\partial w_{ji}} = \sum_n \frac{\partial E_n}{\partial w_{ji}} \tag{9.29}$$

Finally, the complexity of the error back-propagation is just $O(W)$ (or the number of the connection weight matrix) as the computations are dominated by those connection weight matrix multiplications at each layer.

As is seen, from a topology point of view, a Hopfield net is a complete undirected graph $G = <V, f>$, where V are vertices and f is a function that maps each pair of units (i, j) to a connectivity weight w_{ij} whose value is determined by the following rules:

$$w_{ij} = \begin{cases} 0, & if\ i = j\ (no\ self\ connect) \\ w_{ij}, & if\ i \neq j \end{cases} \tag{9.30}$$

In fact, the Hopfield network model originated from a mathematical model of ferromagnetism in statistical mechanics, which is called the *Ising spin model*, named after the physicist Ernst Ising. The ferromagnetism theory explains how materials become magnetized.

The Ising spin model describes how spins can help form ferrormagnetism. It assumes that the spins, noted as s_i, can take a positive value of +1 or a negative value of -1, and interact in pairs, with the total energy of the system (E) defined as follows:

$$E = -\sum_{ij} J_{ij} s_i s_j - \mu H \sum_{i=1}^{N} s_i \tag{9.31}$$

where the sum counts each pair of spins only once, μ is the magnetic moment, H is the magnetic field strength, and J_{ij} is known as exchange energy. Here, there are three scenarios with regard to the term J_{ij}:

Now let's come back to the Hopfield network model. The Hopfield network model, as shown in Figure 9.9, assumes that each unit has a state s_i that is updated according to the following formulation:

$$s_i = \begin{cases} +1, & if \sum_j w_{ij} s_j \geq \theta_i \\ -1, & otherwise \end{cases} \tag{9.32}$$

Here we see that it's the same symbol (s_i) that can take the same set of values of either +1 or −1 with the only difference that the state of the unit i has a threshold value of θ_i and the state transition is determined by whether the sum of the weighted state from other units exceeds that threshold.

Furthermore, similar to an Ising system as described by Eq. (9.31), the energy of a Hopfield network is defined, by:

$$E = -\frac{1}{2}\sum_{ij} w_{ij} s_i s_j + \sum_{i=1}^{N} \theta_i s_i \tag{9.33}$$

Comparing Eqs. (9.31) with (9.33) shows that the two equations are similar. With the Hopfield network model, the second term is determined by the sum of the state of each unit and its threshold, while the first term represents the interactions among all units. Similar to the Ising spin model, when two units have the same state, the product of $s_i s_j$ would be positive, which will lower the energy of the system. However, when two units have different states, the product of $s_i s_j$ would be negative, which will elevate the energy of the system. In fact, the Hopfield networks use the Hebbian learning rule to update the connection weight w_{ij}, in the way that simultaneous activation of units leads to increased connection weight, which helps lower the energy of the system as well. The network is considered properly trained when the energy of the system is driven to a local minimum, which the network should remember.

Boltzmann machines are named after the Boltzmann distribution in statistical mechanics. First, let's clarify what a Boltzmann distribution is.

Suppose we have a system that is defined by its N energy states ε_i (i = 1, .., N). Then, under the thermal equilibrium condition, the probability for the system to be at the energy state ε_i obeys the following distribution:

$$p_i = \frac{e^{-\varepsilon_i/kT}}{\sum_{j=1}^{N} e^{-\varepsilon_j/kT}}, \qquad (9.34)$$

where k is the Boltzmann constant, T the temperature of the system and N the number of all states *accessible* to the system. The denominator in Eq. (9.34) is also known as the *canonical partition function* or the *state sum*, and often denoted as Z:

$$Z = \sum_{j=1}^{N} e^{-\varepsilon_j/kT}, \qquad (9.35)$$

In our machine learning context, we can replace $1/kT$ with γ so that Eq. (9.34) can be rewritten as:

$$p_i = \frac{e^{-\gamma\varepsilon_i}}{Z}, \qquad (9.36)$$

Eq. (9.36) shows that lower energy states have higher probabilities. Fluctuations may occur, but the probabilities of transitioning to higher energy levels decrease exponentially. When the system temperature is raised, the probabilities of transitioning to higher energy levels increase accordingly. With real metal materials, there is a metallurgical process known as *annealing*, which has a material heated up slowly and then cooled down slowly. During this process, the crystalline structure of the material can reach a global energy minimum. The high temperature excites all particles such as atoms and molecules, but during the cooling phase, particles assume their optimal position in the crystalline lattice by losing energy, leading to fewer fractures and irregularities in the crystal. The similar process in machine learning is called *simulated annealing*, though.

In addition, the ratio of probabilities between states i and j is given by:

$$\frac{p_i}{p_j} = e^{-\gamma(\varepsilon_i - \varepsilon_j)} \tag{9.37}$$

Another important property with the Boltzmann distribution is that it maximizes the following entropy, which is one of the key concepts in machine learning:

$$H(p_1, p_2, \ldots, p_N) = -\sum_{i=1}^{N} p_i log_2 p_i, \tag{9.38}$$

Therefore, once again, the Boltzmann distribution describes the states of a system under the thermal equilibrium condition.

Now let's get back to Boltzmann machines. We follow (Rojas, 1996) to derive the Boltzmann learning algorithm, which would be a good learning exercise for gaining a deeper understanding of how a neural network model is formulated precisely.

The global energy E of a Boltzmann machine is defined as, in general:

$$E = -\frac{1}{2}\sum_{i=1}^{n}\sum_{j=1}^{n} w_{ij}\, x_i\, x_j + \sum_{i=1}^{n} \theta_i x_i, \tag{9.39}$$

where:

- The constant n is the total number of units, both visible and hidden (which receive no external input).
- The state variable x_i takes a binary value of 0 or 1.
- The weight parameter w_{ij} represents the connection strength between units i and j.
- The variable θ_i is the bias of unit i.

Since a Boltzmann machine consists of a visible layer and a hidden layer, let's index these two layers with v and h for convenience, respectively. Furthermore, without losing generality, let's assume $\theta_i = 0$, i.e., the bias and thus the threshold is zero. From the probabilistic perspective, the following relationship holds:

$$P_v = \sum_h P_{vh}, \tag{9.40}$$

where P_v is the probability that the input units assume state v, P_{vh} is the joint probability of the combined network state vh, and the sum represents all such states, implying that all network states will be visited at the beginning. According to Eq. (9.36), Eq. (9.40) can be further expressed as:

$$P_v = \frac{1}{Z}\sum_h \exp(-\gamma E_{vh}), \tag{9.41}$$

where Z is now:

$$Z = \sum_{vh} \exp(-\gamma E_{vh}), \tag{9.42}$$

The energy of the network in state vh is:

$$E_{vh} = -\frac{1}{2}\sum_{i=1}^{n}\sum_{j=1}^{n} w_{ij}\, x_i^{vh} x_j^{vh}, \tag{9.43}$$

Now we assume that the input data follows a probability distribution of F_v, and we want the network to learn this distribution. Then, the dis-similarity or distance between the assumed and the actual behavior of the network can be measured by the KL divergence, which was introduced in Chapter 4, as follows:

$$D = \sum_v F_v log \frac{F_v}{P_v}. \tag{9.44}$$

We know that the dis-similarity D would be zero if the two distributions were equal, which means that the learning algorithm should attempt to minimize D. We also know that this can be achieved by keeping adjusting the connection strength parameter with gradient descent:

$$\Delta w_{ij} = -\eta \frac{\partial D}{\partial w_{ij}} = \eta \sum_v \frac{F_v}{P_v}\frac{\partial P_v}{\partial w_{ij}}, \tag{9.45}$$

The partial derivative in the above equation can be obtained by combining Eqs. (9.41), (9.42) and (9.43):

$$\frac{\partial P_v}{\partial w_{ij}} = \gamma\left(\sum_h x_i^{vh} x_j^{vh} P_{vh} - P_v < x_i x_j >_{free}\right), \tag{9.46}$$

where the term $< x_i x_j >_{free}$ is:

$$< x_i x_j >_{free} = \sum_{v,h} x_i^{vh} x_j^{vh} P_{vh}, \tag{9.47}$$

which is the expected value of the product of the unit states x_i and x_j in a free running network. Substituting the partial derivative in Eq. (9.45) with (9.46) gives us:

$$\Delta w_{ij} = \eta\gamma\left(\sum_v \frac{F_v}{P_v} \sum_h x_i^{vh} x_j^{vh} P_{vh} - \sum_v F_v < x_i x_j >_{free}\right). \tag{9.48}$$

Next, we introduce the conditional probability $P_{h|v}$, which is the probability that the hidden units assume state h when the visible units are in state v, that is:

$$P_{vh} = P_{h|v} P_v. \tag{9.49}$$

Substituting (9.49) into (9.48) gives us:

$$\Delta w_{ij} = \eta\gamma\left(\sum_{v,h} F_v P_{h|v} x_i^{vh} x_j^{vh} - \sum_v F_v < x_i x_j >_{free}\right) = \eta\gamma(< x_i x_j >_{fixed} - < x_i x_j >_{free}). \tag{9.50}$$

where the term $<x_i x_j>_{fixed}$ is equal to:

$$< x_i x_j >_{fixed} = \sum_v F_v P_{h|v} x_i^{vh} x_j^{vh}. \tag{9.51}$$

We used the fact that $\sum_v F_v = 1$ when deriving Eq. (9.50).

Eq. (9.50) is quite amazing: it clearly shows what Boltzmann learning is about. The terms of $<x_i x_j>_{fixed}$ and $< x_i x_j >_{free}$ are the stochastic correlation matrices of the network states at the visible layer and system level, respectively. Without the hidden layer, the difference would be just zero. When the hidden layer is added, it allows the units at the visible layer to lose energy gradually to the hidden layer. Then, when a balance at the system level is reached, the KL divergence approaches zero, and the data distribution at the input layer is learnt to be similar to the Boltzmann distribution, which can be used to generate new data to augment existing data. In this sense, a Boltzmann machine is a *generative* neural network model. It is also *asynchronous*, since only one unit is allowed to update its state a time.

The paper by Carreira-Perpiñán and Hinton started with a succinct probability distribution $p(\boldsymbol{x}; \boldsymbol{W})$ over an input data vector $\boldsymbol{x}$ with a connection weight parameter matrix $\boldsymbol{W}$ as follows:

$$p(\boldsymbol{x}; \boldsymbol{W}) = \frac{1}{Z(\boldsymbol{W})} e^{-E(\boldsymbol{x};\boldsymbol{W})}, \tag{9.52}$$

where Z is the same partition function as (9.42) and E is the same energy function as (9.43) for a Boltzmann distribution. As described in the last section, the KL divergence is defined similarly as

$$KL(p_0||p_\infty) = \sum_{\boldsymbol{x}} p_0(\boldsymbol{x}) log \frac{p_0(\boldsymbol{x})}{p(\boldsymbol{x};\boldsymbol{W})}, \tag{9.53}$$

where p_0 is the data distribution and p_∞ is the *eventual* distribution at the end of the simulated annealing or when the equilibrium condition is reached, which may take $n \to \infty$ steps of iterations. The authors introduced the concept of *contrastive divergence* for this particular subject of RBM, which is the difference between two divergences as shown below:

$$CD_n = KL(p_0||p_\infty) - KL(p_n||p_\infty), \tag{9.54}$$

Now assume that there are v visible units $x = (x_1, \ldots, x_v)^T$ that encode a data vector, and h hidden units $y = (y_1, \ldots, y_v)^T$; all units are binary variables taking values in $\{0, 1\}$. The energy is then

$$E(\boldsymbol{x}, \boldsymbol{y}; \boldsymbol{W}) = -\boldsymbol{y}^T \boldsymbol{W} \boldsymbol{x}, \tag{9.55}$$

where $\boldsymbol{x}$, $\boldsymbol{y}$ and $\boldsymbol{W}$ are v-dimensional, h-dimensional, and $h \times v$ dimensional, respectively. Since there is no inter-connection among each pair of the units of a layer (visible or hidden), the visible units are conditionally independent given the hidden units and the hidden units are conditionally independent given the visible units, simply because the units at each layer are not connected to each other or they are not aware of each other. This makes the learning simpler than in general Boltzmann machines as we introduced in the last section, since one step of Gibbs sampling can be carried out in two half-steps: the first updates all the hidden units and the second updates all the visible units, or each layer can be handled independently since its units are conditionally independent.

Now, let's rewrite Eq. (9.50), the weight update for a generic Boltzmann machine, here:

$$\Delta w_{ij} = \eta\gamma\left(< x_i x_j >_{fixed} - < x_i x_j >_{free}\right). \tag{9.56}$$

And compare (9.56) with the weight updates for the RBM shown below in two different cases for *ML* learning and CD_n learning, respectively:

$$w_{ij}^{(\tau+1)} = w_{ij}^{(\tau+1)} + \eta\left(\ll y_i x_j >_{p(\boldsymbol{y}|\boldsymbol{x};\boldsymbol{W})}>_0 - < y_i x_j >_\infty\right) \qquad ML\ learning, \tag{9.57}$$

$$w_{ij}^{(\tau+1)} = w_{ij}^{(\tau+1)} + \eta\left(\ll y_i x_j >_{p(\boldsymbol{y}|\boldsymbol{x};\boldsymbol{W})}>_0 - < y_i x_j >_n\right) \qquad CD_n\ learning, \tag{9.58}$$

Gibbs sampling: We mentioned that one of the key techniques that the RBM uses to speed up training is the Gibbs sampling. So, what is Gibbs sampling exactly? There are numerous terms in machine learning, so you might want to expand your knowledge in this area. Here, we give a brief explanation about what Gibbs sampling is.

First, let's review the rules of probability. The two simple rules of sum rule and product rule form the basis of all probability machinery:

$$Sum\ rule: \quad p(x) = \sum_{y} p(x,y)$$

$$Product\ rule: \quad p(x,y) = p(y|x)p(x)$$

In the above equations, $p(x, y)$ is a *joint* probability, $p(y|x)$ is a *conditional* probability of y given x, and p(x) is a *marginal* (partial) probability.

Now let's see how Gibbs sampling is carried out. Gibbs sampling is used to sample from a conditional distribution when the joint distribution, e.g., $p(x, y, z)$, is unknown or hard to sample from. Assume that at step τ, we have a sample of $(x^{(\tau)}, y^{(\tau)}, z^{(\tau)})$. At step τ+1:

- We first get a new sample of $x^{(\tau+1)}$ from sampling $p(x| y^{(\tau)}, z^{(\tau)})$;
- Then we get a new sample of $y^{(\tau+1)}$ from sampling $p(y| x^{(\tau+1)}, z^{(\tau)})$ with the new sample of $x^{(\tau+1)}$ in the conditional distribution;
- Finally, we get a new sample of $z^{(\tau+1)}$ from sampling $p(z| x^{(\tau+1)}, y^{(\tau+1)})$ with the new samples of $x^{(\tau+1)}$ and $y^{(\tau+1)}$ in the conditional distribution.

This process can be repeated for up to as many times as required until all samples are obtained, which would obey a joint distribution of $p(x, y, z)$ approximately. This is how Gibbs sampling works conceptually.

10 Convolutional Neural Networks

10.2.1 Convolution versus cross-correlation

In fact, there are two similar mathematical terms: *convolution* and *cross-correlation*. These two terms are similar but not exactly the same. The prefix *convolutional* in the term CNN actually means *cross-correlational*, but people have gotten used to call it *convolutional*. The difference between the two will become clear, after we examine the definition of each next.

The convolution between two continuous functions f and g is defined using a star symbol between the two as:

$$(f * g)(t) \triangleq \int_{-\infty}^{\infty} f(\tau)g(t-\tau)d\tau, \qquad (10.1)$$

A few notes about the above definition:

- While the symbol t is used, it does not need to represent the time domain, e.g., it could be another symbol like x that represents the space domain. Regardless of what symbol is used, it just means *shifting*. If the problem is in the time domain, the above definition means a weighted average of the function $f(\tau)$ at time t where the weighting is given by $g(-\tau)$ shifted by amount t. As t changes, we can think that the input function $f(\tau)$ is weighed and scanned by the weighting function.
- It doesn't have to be one dimensional. For example, it could be two dimensional with two variables for each function.
- It doesn't have to be continuous. For example, the discrete convolution of f and g is given by:

$$(f * g)[n] = \sum_{m=-\infty}^{\infty} f[m]g[n-m]. \qquad (10.2)$$

Now let's switch to the definition of cross-correlation. Cross-correlation between two continuous functions f and g is defined using a star symbol as well between the two functions as:

$$(f * g)(\tau) \triangleq \int_{-\infty}^{\infty} f^*(\tau)g(t+\tau)d\tau, \qquad (10.3)$$

where f^* represents the complex conjugate of f, i.e., if f is a complex function that $f = u + iv$ then its complex conjugate would be $f = u - iv$.

As is seen, now the minus sign for τ with the definition of convolution is flipped to the plus sign for the definition of cross-correlation. Cross-correlation can be used with discrete functions as well, such as:

$$(f * g)[n] = \sum_{m=-\infty}^{\infty} f[m]g[m+n], \qquad (10.4)$$

namely, the minus sign is flipped to the plus sign for the weighting function.

10.2.3 Zero-Padding

As we have just learnt, the convolution part of a CNN works by shifting a receptive field along a dimension of the input layer with a stride until reaching the end of that dimension. The question is what if there are not enough units left at the end to make up a complete receptive field. The solution is to add zero-valued units, which is called *zero-padding*.

In fact, there is a formula to compute how many zero-valued units to pad on the input layer along each dimension, with given output size, stride, input size, and receptive field size along that dimension. You might find various formulas or directions from other texts, but I summarized this formula, which is easy to remember and convenient:

$$n_{zero\_padding} = \left[\left(n_{output} \times s - n_{input}\right) + (f - s)\right] mod\ f, \qquad (10.5)$$

where (along the dimension we are concerned with):

- n_{output} is the dimension of the output layer, for example, a hidden layer;
- s is stride, i.e., how many units to jump for making the next receptive field for the next unit on the output layer;
- n_{input} is the dimension of the input layer;
- and f is the dimension of the receptive field. Now let's explain (10.5) as follows:
- First, assuming that the receptive field is the same as the stride, i.e., no overlapping on the input layer for any two adjacent units on the output layer, then, the second term in (10.5) is zero, and we are concerned with the first term only. In this case, it's really about whether the input dimension is divisible by the output dimension, which should be equal to the receptive field or stride. If not divisible, the number of the zero-padding units would be equal to $n_{output} \times s - n_{input}$, or the difference between $n_{output} \times s$ and n_{input}. We need to take the modulo against the receptive field to minimize the number of zero-padding to an absolutely minimally necessary number.
- Next, if the receptive field and the stride are not the same, then the difference contributes to the number of zero-padding units needed. This is understandable as unequal receptive field and stride may cause some imbalance given many possible situations arising from the first term.

16 and the output layer dimension $n_{output} = 8$, as well as the stride $s = 2$ and receptive $f = 3$, then, according to Eq. (10.5), we have:

$$n_{zero\_padding} = [(8 \times 2 - 16) + (3 - 2)]\ mod\ 3 = [0 + 1]\ mod\ 3 = 1,$$

which means that we only need to pad the input layer with one zero-valued unit. We see that the first term is zero, but the second term is not, which results in 1. Figure 10.17 confirms this result.

Now in Figure 10.19, you may notice that we added a term of "ReLU." This term refers to the Rectified Linear Unit, which is defined as simple as:

$$f(x) = \max(0, x), \qquad (10.6)$$

i.e., for $x < 0$, $x = 0$, and for $x \geq 0$, $f(x) = x$ or $f(x)$ is a linear function of x. Remember that each unit requires an activation function to perform the transformation of the weighted sum of the inputs from connected units in the prior layer. Le Cun used a hyperbolic tangent function in his 1989 paper, which is defined as follows:

$$tanh(x) = \frac{\sinh(x)}{\cosh(x)} = \frac{e^x - e^{-x}}{e^x + e^{-x}}, \qquad (10.7)$$

- **Output layer**: Finally, the output layer has 10 units for classifying the full ASCII set as shown in Figure 10.22. This layer is composed of Euclidean Radial Basis Function units (RBF), one for each class, with 84 inputs each from the F6 layer. The output of each RBF unit, y_i, is computed as follows:

$$y_i = \sum_j (x_j - w_{ij})^2, \qquad (10.8)$$

$$mini\_mean: \mu_B \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \qquad (10.9)$$

$$mini\_mean\ variance: \sigma_B^2 \leftarrow \frac{1}{m}(x_i - \mu_B)^2 \qquad (10.10)$$

$$normalize: \hat{x}_i \leftarrow \frac{x_i - \mu_B}{\sqrt{\sigma_B^2 + \varepsilon}} \qquad (10.11)$$

$$scale\ and\ shift: y_i = \gamma \hat{x}_i + \beta \qquad (10.12)$$

where (10.9) and (10.10) are mean and mean variance, respectively, γ and β are learnt parameters applied to the zero-centered and normalized activation $\hat{x}_i$, and ε is a smoothing parameter to avoid dividing by zero, typically 0.001.The formulas for normalizing the errors were given in the original paper as well.

10.6.3 He Initialization

As we know, weight normalization is a necessary task for every machine learning project. The method here, known as *He initialization*, is widely used for this task. This technique was developed by the same authors of the ResNet, in a paper titled "*Delving deep into rectifiers: Surpassing human-level performance on ImageNet classification*," available online at https://arxiv.org/pdf/1502.01852.pdf. This technique helped achieve the first, surpassing human-level performance (5.1%) with the ImageNet classification task. In this section, we give a brief introduction to this initialization technique, as if you work on any deep learning project, most likely you will use this technique to initialize the weight parameters of your model.

The He initialization is defined by the following activation function:

$$f(y_i) = \begin{cases} y_i, & if\ y_i > 0 \\ a_i y_i, & if\ y_i \le 0 \end{cases}. \qquad (10.13)$$

Here y_i is the input of the nonlinear activation f on the ith channel, and a_i is a coefficient controlling the slope of the negative part. The subscript i in a_i indicates that we allow the nonlinear activation to vary on different channels. When $a_i = 0$, it becomes ReLU; when a_i is a learnable parameter, Eq. (10.13) is known as *Parametric ReLU* (PReLU), which is equivalent to:

$$f(y_i) = \max(0, y_i) + \min(0, y_i), \qquad (10.14)$$

Figure 10.33 shows the functions of ReLU and PReLU. If a_i is a small and fixed value, e.g., $a_i = 0.01$, PReLU becomes the *Leaky ReLU* (LReLU). The motivation of LReLU is to avoid zero gradients.

11 Recurrent Neural Networks

Due to its memory capability, the input, output and hidden state of a simple cell are expressed as functions of time, such as $x_{(t)}$, $y_{(t)}$ and $h_{(t)}$, as shown in Figure 11.1 on the left. For simple cells, the output and the hidden state are the same, i.e., $y_{(t)} = h_{(t)}$. However, the input is a sum of the current input and the previous hidden state, i.e., $x_{(t)} + h_{(t-1)}$, and the output or hidden state is a function of both the current input and the previous state:

$$y_t = h_t = f(x_t, h_{t-1}), \qquad (11.1)$$

11.2.1 Back-propagation through time (BPTT)

In order to explain how the technique of back-propagation through time (BPTT) works, we added weights to Figure 11.1 and re-presented it here as shown in Figure 11.4. Note from left to right that after unrolling in time, we kept all parameters the same for all unrolled units in order to: (1) save training time, and (2) reduce the number of parameters to fit to minimize the potential of overfitting. Weights are slowing changing long term memories, and therefore, doing so would not affect performance.

Here is a summary of what those items in Figure 11.4 mean exactly:

- Inputs $x_{(t-1)}$, $x_{(t)}$, and $x_{(t+1)}$ represent a sequence, which could be a sequence of words, with each of its members being a one-hot vector representing a word of the sentence.
- Outputs $y_{(t-1)}$, $y_{(t)}$, and $y_{(t+1)}$ represent a sequence as well, which could be a sequence of words if the input were a sequence of words, with each of its members being a one-hot vector representing a word of the newly translated or predicted sentence.
- Hidden states $h_{(t-1)}$, $h_{(t)}$, and $h_{(t+1)}$ represent the "memory" of the network. They are computed based on the current input and the preceding state, such as shown below:

$$h_{(t)} = f(W_x x_t, W_h h_{t-1}), \qquad (11.2)$$

Let $d_k(t)$ represent the output unit k's target at time t. Using mean squared error, unit k's error signal is

$$\theta_k(t) = f_k'\big(net_k(t)\big)\big(d_k(t) - y_k(t)\big), \qquad (11.3)$$

where

$$y_i(t) = f_i\big(net_i(t)\big) \qquad (11.4)$$

is the activation of a non-input unit i with differentiable activation function f_i, and

$$net_i(t) = \sum_j w_{ij} y_j(t-1) \qquad (11.5)$$

is unit i's current net input, and w_{ij} is the weight for the connection from unit j to unit i. The non-output unit j's back-propagated error signal is

$$\theta_j(t) = f_j'\left(net_j(t)\right) \sum_i w_{ij}\theta_i(t+1). \qquad (11.6)$$

The corresponding contribution to w_{jl}'s total update is $\alpha\theta_j(t)y_l(t-1)$, where α is the learning rate, and l stands for an arbitrary unit connected to unit j.

Now, to help you understand Eqs. (11.3) – (11.6), let me explain a few things here:

- Eq. (11.3) is essentially our Eq. (9.25) except that the authors used d instead of t for the target labeled value, net_k for activation a_k, f for sigmoid function σ, and finally θ_k instead of δ_k for the error signal. Besides, all quantities are time-varying. The authors' notations were conventions at that time, and our notations are conventions at present.
- Eq. (11.5) is equivalent to our Eq. (9.18), which is forward-looking, i.e., not backward-looking.
- Eq. (11.6) is equivalent to our Eq. (9.26) except differences in notations.

Next, the authors analyzed the above conventional BPTT with an arbitrary propagation "back into time" for q time steps, i.e., with units $v \leftarrow q \leftarrow u$, and arrived at the following interesting conclusions:

- If $|f_{l_m}'(net_{l_m}(t-m)w_{l_m}w_{l_{m-1}}| > 1.0$, the error blows up, and conflicting error signals arriving at unit v can lead to oscillating weights and unstable learning.
- If $|f_{l_m}'(net_{l_m}(t-m)w_{l_m}w_{l_{m-1}}| < 1.0$, the error vanishes, and nothing can be learnt in acceptable time.
- With conventional logistic sigmoid activation functions, the error flow tends to vanish as long as the weights have absolute values below 4.0, especially at the beginning of the training phase.

Constant Error Flow: Naïve Approach

A single unit. Based on the above analysis, how can we achieve constant error flow through a single unit j with a single connection to itself so that error gradients will neither vanish nor explode? According to Eq. (11.6), at time t, j's local error back flow is:

$$\theta_j(t) = f_j'\left(net_j(t)\right) w_{jj}\theta_j(t+1). \qquad (11.7)$$

This means that to enforce constant error flow through j, we require $\theta_j(t) = \theta_j(t+1)$, or

$$f_j'\left(net_j(t)\right)w_{jj} = 1.0. \tag{11.8}$$

The constant error carrousel. Integrating (11.8) gives us

$$f_j\left(net_j(t)\right) = \frac{net_j(t)}{w_{jj}}. \tag{11.9}$$

This means that f_j has to be linear, and unit j's activation has to remain constant:

$$y_j(t+1) = f_j\left(net_j(t+1)\right) = f_j\left(w_{jj}y_j(t)\right) = y_j(t). \tag{11.10}$$

The above condition can be satisfied by choosing an identity function $f_j : f_j(x) = x, \forall x$, and by setting $w_{jj} = 1.0$. This was referred to as the constant error carrousel (CEC), which was the foundation for LSTM. These interesting assumptions led to adding a forget-gate later, as will be discussed in the next section.

Now with in_j's activation at time t denoted by $y^{in_j}(t)$ and out_j's by $y^{out_j}(t)$, we have

$$y^{in_j}(t) = f_{in_j}\left(net_{in_j}(t)\right), \tag{11.11}$$

$$y^{out_j}(t) = f_{out_j}\left(net_{out_j}(t)\right), \tag{11.12}$$

where

$$net_{<in,out,c>_j}(t) = \sum_u w_{<in,out,c>_j u}y^u(t-1). \tag{11.13}$$

$$y^{c_j}(t) = y^{out_j}(t)h\left(s_{c_j}(t)\right), \tag{11.14}$$

where the "internal state" $s_{c_j}(t)$ is

$$s_{c_j}(0) = 0, \quad for\ t = 0, \tag{11.15a}$$

$$s_{c_j}(t) = s_{c_j}(t-1) + y^{in_j}(t)g\left(net_{c_j}(t)\right) \quad for\ t > 0. \tag{11.15b}$$

Eqs. (11.14) and (11.15b) have different implications: (11.14) implies that the function h scales memory cell outputs computed from the internal state s_{c_j}, while (11.15b) implies that the function g squashes net_{c_j}.

After adding the forget-gate for erasing memory when needed, the set of equations we need to add are:

$$net_{\varphi_j}(t) = \sum_u w_{\varphi_j u} y^u(t-1). \qquad (11.16)$$

$$y^{\varphi_j}(t) = f_{\varphi_j}\left(net_{\varphi_j}(t)\right), \qquad (11.17)$$

The revised update equation for s_c in the extended LSTM algorithms is

$$s_{c_j}(t) = y^{\varphi_j}(t) s_{c_j}(t-1) + y^{in_j}(t) g\left(net_{c_j}(t)\right) \; for \; t > 0. \qquad (11.18)$$

where we have marked the forget-gate's output in red.

Equations

Assuming that the input sequence is $\mathbf{x} = (x_1, \ldots, x_T)$, then, a standard RNN's hidden vector sequence $\mathbf{h} = (h_1, \ldots, h_T)$ and output vector sequence $\mathbf{y} = (y_1, \ldots, y_T)$ are computed by iterating the following equations from $t = 1$ to T:

$$h_t = H(W_{xh}x_t + W_{hh}h_{t-1} + b_h), \quad (11.19)$$

$$y_t = W_{hy}h_t + b_y, \quad (11.20)$$

To help understand the above two equations, a regular hidden unit is redrawn, as shown in Figure 11.12. To calculate the hidden state, we sum up all inputs, add the bias, and apply the hidden activation function H, which gives the hidden state of (11.19) at time t. To calculate the output, just multiply the state with the connection weight and add the bias, according to (11.20).

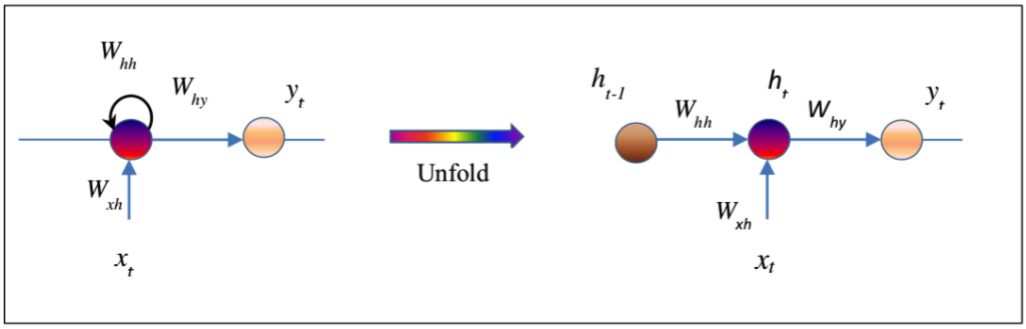

Figure 11.12 A regular RNN hidden unit.

As we stated earlier, generic RNNs do not work well due to gradient vanishing and exploding issues, which led to the invention of the LSTM that uses gates to control memory read and write operations for long range context. For the version of the LSTM used for this example as shown on the left of Figure 11.11, the sigmoid function σ is used for the hidden function h. Now, the state vectors for the *input gate* (i), *forget gate* (f), *cell activation* (c), *output gate* (o), and *hidden unit* (h), can be expressed as:

$$i_t = \sigma(W_{xi}x_t + W_{hi}h_{t-1} + W_{ci}c_{t-1} + b_i), \quad (11.21)$$

$$f_t = \sigma(W_{xf}x_t + W_{hf}h_{t-1} + W_{cf}c_{t-1} + b_f), \quad (11.22)$$

$$c_t = f_t c_{t-1} + i_t tanh(W_{xc}x_t + W_{hc}h_{t-1} + b_c), \quad (11.23)$$

$$o_t = \sigma(W_{xo}x_t + W_{ho}h_{t-1} + W_{co}c_t + b_o), \quad (11.24)$$

$$h_t = o_t tanh(c_t), \quad (11.25)$$

Here, all vectors have the same size. For the connections added from the cell to each gate, their weight matrices are diagonal, so element m in each gate vector only receives input from element m of the cell vector.

For the BRNN, the forward hidden sequence, the backward hidden sequence, and the output sequence can be computed by iterating the backward layer from $t = T$ to 1, the forward layer from $t = 1$ to T, and then updating the output layer as follows:

$$\vec{h}_t = H\left(W_{x\vec{h}}x_t + W_{\vec{h}\vec{h}}\vec{h}_{t-1} + b_{\vec{h}}\right), \tag{11.26}$$

$$\overleftarrow{h}_t = H\left(W_{x\overleftarrow{h}}x_t + W_{\overleftarrow{h}\overleftarrow{h}}\overleftarrow{h}_{t+1} + b_{\overleftarrow{h}}\right), \tag{11.27}$$

$$y_t = W_{\vec{h}y}\vec{h}_t + W_{\overleftarrow{h}y}\overleftarrow{h}_t + b_y, \tag{11.28}$$

Now, we stack the same hidden layer as described above to form a deep, N-layer BRNN, with the output from one layer to be the input of the next layer. The hidden vector sequence $\boldsymbol{h}^n$ is iteratively computed from $n = 1$ to N and $t = 1$ to T, according to:

$$h_t^n = H\left(W_{h^{n-1}h^n}h_t^{n-1} + W_{h^nh^n}h_{t-1}^n + b_h^n\right), \tag{11.29}$$

where $\boldsymbol{h}^0 = \boldsymbol{x}$. The network outputs y_t is:

$$y_t = W_{h^Ny}h_t^N + b_y \tag{11.30}$$

Replacing each hidden sequence $\boldsymbol{h}^n$ with the forward and backward sequences and ensuring output from one layer used as the input for the next layer forms a deep bidirectional RNN or DBRNN. This example used LSTM as the hidden layer and it was the first deep, bidirectional LSTM or DBLSTM. It turned out that it yielded a dramatic improvement over single-layer LSTM.

12 Autoencoders

Now from a mathematical point of view, we can consider that the inputs are first transformed from x to z as follows:

$$\boldsymbol{z} = f(\boldsymbol{W}\boldsymbol{x} + \boldsymbol{b}), \qquad (12.1)$$

where f is a nonlinear activation function, such as a sigmoid function or Rectified Linear Unit (ReLU), $\boldsymbol{W}$ and $\boldsymbol{b}$ are the weight matrix and bias, respectively. This part is the same as we explained before.

Next, we can consider that code $\boldsymbol{z}$ is transformed to the output $\boldsymbol{x}'$ similarly

$$\boldsymbol{x}' = g(\boldsymbol{W}'\boldsymbol{z} + \boldsymbol{b}'), \qquad (12.2)$$

where g is another nonlinear activation function, such as a sigmoid function or ReLU, and $\boldsymbol{W}'$ and $\boldsymbol{b}'$ are the weight matrix and bias as well, respectively. This part is still the same as we explained before. Because of the symmetry, the first part is called the *encoder* and the second part the *decoder*.

Now, where is the learning part? The learning part is that the autoencoder is trained to minimize the difference between $\boldsymbol{x}$ and $\boldsymbol{x}'$, i.e., minimizing the distance of:

$$l_2(\boldsymbol{x}, \boldsymbol{x}') = \|\boldsymbol{x} - \boldsymbol{x}'\|^2 \qquad (12.3)$$

12.3.1 A review of the Bayes' theorem

Assume that we have two random variables x and z, and we use letter p to represent probability. Then, the Bayes' theorem can be expressed as:

$$p(z|x)(posterior) = p(z)(prior) \times \left(\frac{p(x|z)(marginal\ likelihood)}{p(x)(marginal\ probability)} \right), \qquad (12.4)$$

where all terms have been annotated with what they are. For example, $p(z|x)$ is the *posterior* probability, $p(z)$ is the *prior* probability, $p(x|z)$ is the *marginal likelihood*, and $p(x)$ is the *marginal* probability. You can think of x as the observed data, and z a parameter or condition, like the *code* with our autoencoders.

A particularly good example from https://en.wikipedia.org/wiki/Posterior_probability explains the Bayes' theorem really well. So let's use it to explain all above terms in (12.4) next.

12.3.2 Variational inference

First, let's start with a function that has a generic form of $y = f(x)$ that we are familiar with. We can take this as a mapping from x→y by a function *f*. We are often concerned with the amount of change in *y* if there is a change in *x*, which is governed by the first order derivative of *f*, like $dy = (df/dx)dx$.

Similarly, we can define a functional as a mapping that takes a function as input and returns the value of the functional as the output. A good example is the *entropy* we are familiar with

$$H[p] = -\int p(x) ln[p(x)]dx, \qquad (12.5)$$

where *p* is the probability of the variable *x* or the functional we have referred to as above.

Now, to put it into context, we are concerned with the amount of change in entropy corresponding to an infinitesimal change in the functional, which comes from the rules of calculus of variations, similar to the standard calculus. This is interesting to us, as you already know that many machine learning problems can be abstracted into optimization problems in which the quantity being optimized is a *functional*. The solutions to such optimization problems often lie with exploring all possible input methods and finding one that maximizes or minimizes the functional.

Now let's consider how the concept of variational optimization can be applied to the machine learning inference problem. Suppose we have a model with all observed inputs expressed by $\boldsymbol{X}$ and all latent variables or parameters by $\boldsymbol{Z}$. Our model is now *probabilistic* and can be described by a joint probability function $p(\boldsymbol{X}, \boldsymbol{Z})$. Our goal is to find an approximation for the posterior distribution $p(\boldsymbol{Z}|\boldsymbol{X})$ as well as the marginal probability distribution or model evidence $p(\mathbf{X})$, similar to the girls-boys example given in the previous section, but we are concerned with a distribution here, not a particular prediction. The log marginal probability can be decomposed into (Bishop, 2006):

$$\ln[p(X)] = L(q) + KL(q\|p), \qquad (12.6)$$

where the above two terms are defined by

$$L(q) = \int q(\boldsymbol{Z}) \ln\left[\frac{p(\boldsymbol{X}, \boldsymbol{Z})}{q(\boldsymbol{Z})}\right] d\boldsymbol{Z}, \qquad (12.7)$$

and

$$KL(q||p) = -\int q(\boldsymbol{Z}) \ln\left[\frac{p(\boldsymbol{Z}|\boldsymbol{X})}{q(\boldsymbol{Z})}\right] d\boldsymbol{Z}, \tag{12.8}$$

In order for Eq. (12.6] to hold, we seem to require that

$$p(\boldsymbol{X}) = \int \left[\frac{p(\boldsymbol{X}, \boldsymbol{Z})}{p(\boldsymbol{Z}|\boldsymbol{X})}\right]^{q(\boldsymbol{Z})} d\boldsymbol{Z}, \tag{12.9}$$

You might have noticed that we introduced a new function $q(\mathbf{Z})$. This is perhaps the most interesting part. The rationale is that in general, the direct optimization of $p(\mathbf{X})$ is difficult or intractable, but optimization of the likelihood function or the joint probability $p(\boldsymbol{X}, \boldsymbol{Z})$ might be significantly easier, which is why we see the joint probability $p(\boldsymbol{X}, \boldsymbol{Z})$ in Eq. (12.7). However, we do not just stop here. The tricky part is that we introduced the function $q(\mathbf{Z})$, which is an extra dimension. The first term in (12.6) or (12.7) is called the lower bound because the KL divergence is always positive. Now we can maximize the lower bound $L(q)$ by optimization w.r.t. the distribution $q(\mathbf{Z})$, which is equivalent to minimizing the KL divergence in Eq. (12.8). If we give enough freedom for any possible choice for $q(\mathbf{Z})$, then when the lower bound $L(q)$ is maximized, the KL divergence approaches zero, which implies that $q(\mathbf{Z})$ equals the posterior distribution $p(\boldsymbol{Z}|\boldsymbol{X})$.

Therefore, it seems that the next obstacle is how we find $q(Z)$ that minimizes the KL divergence and maximizes the lower bound $L(q)$, which is a functional with respect to $q(\mathbf{Z})$. One way to overcome that obstacle is to use a parametric distribution $q(\boldsymbol{Z}|\boldsymbol{\theta})$ governed by a set of *learnable* parameters $\boldsymbol{\theta}$. This is what the variational autoencoders can do, as explained next.

12.3.3 Auto-encoding variational Bayes (AEVB)

Now it's much easier for us to understand the auto-encoding variational Bayes (AEVB), proposed by Kingma and Welling in 2014, in a paper titled *Auto-Encoding Variational Bayes* (*AEVB*). They came up with an algorithm that works efficiently in the case that the marginal likelihood $p_\theta(x) = \int p_\theta(\boldsymbol{z}) p_\theta(\boldsymbol{x}|\boldsymbol{z}) d\boldsymbol{z}$ and the posterior density $p_\theta(\boldsymbol{z}|\boldsymbol{x}) = p_\theta(\boldsymbol{x}|\boldsymbol{z}) p_\theta(\boldsymbol{z}) / p_\theta(\boldsymbol{x})$ are intractable. These are exactly the issues we discussed in the preceding section. The second problem they have addressed is that with a large dataset, batch optimization is too costly and sampling-based solutions such as Monte Carlo EM would be too slow as well, since it involves a typically expensive sampling loop per data point. Here, EM is a two-step (E-step and M-step), iterative method to find the maximum likelihood estimate (MLE) or maximum a posterior (MAP) estimate of parameters in statistical models with latent variables. Kingma and Welling came up with a solution to those problems by efficiently approximating:

1. The MLE or MAP estimate for the parameters $\boldsymbol{\theta}$, which allows one to mimic the hidden random process and generate artificial data that resembles the real data. Here, MLE and MAP are equivalent since $p_\theta(\boldsymbol{z}|\boldsymbol{x}) \propto p_\theta(\boldsymbol{x}|\boldsymbol{z}) p_\theta(\boldsymbol{z})$.
2. The posterior inference of the latent variable $\boldsymbol{z}$ in the form of $p_\theta(\boldsymbol{z}|\boldsymbol{x})$ with a given observed value $\boldsymbol{x}$ for a choice of parameters $\boldsymbol{\theta}$. This is useful for coding or data representation tasks, which belongs to the objective of the encoder of an autoencoder.

3. The marginal inference of the variable $\boldsymbol{x}$ in the form of $p_\theta(\boldsymbol{x})$. This allows us to perform all kinds of inference tasks where a prior over $\boldsymbol{x}$ is required, which belongs to the objective of the decoder of an autoencoder.

To realize the above solution, a recognition model $q_\phi(\boldsymbol{z}|\boldsymbol{x})$ was introduced to approximate the intractable true posterior $p_\theta(\boldsymbol{z}|\boldsymbol{x})$. This recognition model can be taken as a probabilistic *encoder*, since given a data point $\boldsymbol{x}$ it produces a distribution over the possible values of the code $\boldsymbol{z}$ from which the data point $\boldsymbol{x}$ could have been generated. Similarly, $p_\theta(\boldsymbol{x}|\boldsymbol{z})$ can be taken as a probabilistic *decoder*, since given a code z it produces a distribution over the possible corresponding values of $\boldsymbol{x}$.

The algorithm boils down to obtain a differentiable estimator of the variational lower bound, defined as:

$$\mathrm{L}(\boldsymbol{\theta}, \boldsymbol{\phi}; \boldsymbol{x}) = -D_{KL}[q_\phi(\boldsymbol{z}|\boldsymbol{x})||p_\theta(\boldsymbol{z})] + E_{q_\phi(\boldsymbol{z}|\boldsymbol{x})}[log\ p_\theta(\boldsymbol{x}|\boldsymbol{z})], \qquad (12.10)$$

where the symbol $E[f] = \int p(x)f(x)dx$ represents the expectation or average value of a function f.

In order to optimize the lower bound with gradient ascent, it turns out that a "re-parameterization trick" is required, i.e., re-parameterizing the random stochastic variable $\tilde{\boldsymbol{z}} \sim q_\phi(\boldsymbol{z}|\boldsymbol{x})$ into a deterministic variable

$$\tilde{\boldsymbol{z}} = g_\phi(\varepsilon, x) \quad \text{with} \quad \varepsilon \sim p(\varepsilon)). \qquad (12.11)$$

Here, the symbol '~' in $\tilde{\boldsymbol{z}} \sim q_\phi(\boldsymbol{z}|\boldsymbol{x})$ means that $\tilde{\boldsymbol{z}}$ is distributed according to distribution $q_\phi(\boldsymbol{z}|\boldsymbol{x})$, and so forth. Given (12.11) for z as a function, the second term of (12.10) can be obtained with Monte Carlo estimates of expectations. On the other hand, the KL-divergence term can be integrated analytically, such that only the expected reconstruction error [the second term of (12.10)] requires estimation by sampling. This sampling workaround is known as the *re-parameterization trick*.

Now, let's take the latent variable z in the univariate Gaussian form such that $\boldsymbol{z} \sim p(\boldsymbol{z}|\boldsymbol{x}) = N(\mu, \sigma^2)$. Then a valid re-parameterization would be $\boldsymbol{z} = \boldsymbol{\mu} + \boldsymbol{\sigma}\varepsilon$ with ε being an auxiliary noise variable $\varepsilon \sim N(0, 1)$, a normal distribution of zero mean and standard deviation of 1. Modeling the latent variable z results in the following estimator with data point $x^{(i)}$:

$$\mathrm{L}(\theta; \mathrm{x}^{(\mathrm{i})}) \approx \frac{1}{2}\sum_{j=1}^{J}\left(1 + \log\left((\sigma_j^{(i)})^2\right) - (\mu_j^{(i)})^2 - (\sigma_j^{(i)})^2\right) + \frac{1}{L}\sum_{l=1}^{L} log p_\theta\left(\mathrm{x}^{(\mathrm{i})}|\boldsymbol{z}^{(i,l)}\right), \qquad (12.12)$$

where J represents the mini-batch size, L the number of samples per data point, $\boldsymbol{z}^{(i,l)}$ the i^{th} component of the vector z represented by:

$$\boldsymbol{z}^{(i,l)} = \boldsymbol{\mu}^{(i)} + \boldsymbol{\sigma}^{(i)} \cdot \boldsymbol{\varepsilon}^{(l)} \ \ and\ \boldsymbol{\varepsilon}^{(l)} \sim N(0, I), \qquad (12.13)$$

where '·' represents the element-wise product.

Trainings for both generator and discriminator are governed by their respective cost functions. To learn the generator's distribution p_g over data $\boldsymbol{x}$, a prior $p_z(z)$ on input noise variable z is introduced, which defines a mapping function $G(z;\ \theta_g)$ that maps samples z from the prior $p_z(z)$ to data space, with G a differentiable function represented by an MLP with parameters θ_g. For the discriminator D, the goal is to identify whether the input is from the real data distribution p_{data} or the generator's distribution p_g, which can be estimated with a scalar output that may represent a quantity like the confidence score. Thus, the cost function for the discriminator MLP D can be defined as:

$$J_D(\theta_d, \theta_g) = E_{x\sim p_{data}(x)} logD(x) + E_{z\sim p_z(z)} \log(1 - D(G(z))), \tag{12.14}$$

which is equivalent to maximizing the probability of the sample coming from the training set, not from the generator.

The cost function for the generator MLP G can be defined as:

$$J_G(\theta_d, \theta_g) = E_{z\sim p_z(z)} \log(1 - D(G(z))), \tag{12.15}$$

which is equivalent to minimizing the probability of the sample not from the generator. However, during early training, $\log(1 - D(G(z)))$ saturates; therefore, the generator is trained to maximize $\log(D(G(z)))$ instead.

Since the two models compete against each other, the gain for one party is a loss for the other. Therefore, Eqs. (12.14) and (12.15) can be combined into one with the familiar two-player *min-max* value function $V(G, D)$ from game theory as follows:

$$\min_G \max_D V(D, G) = E_{x\sim p_{data}(x)} logD(x) + E_{z\sim p_z(z)} \log(1 - D(G(z))), \tag{12.16}$$

The adversarial autoencoders begin with an aggregated posterior distribution of $q(z)$ on the hidden code vector of the autoencoder, as shown below:

$$q(z) = \int q(z|x)p_d(x)dx, \tag{12.17}$$

where $q(z|\boldsymbol{x})$ is an encoding distribution, and $p_d(\boldsymbol{x})$ is the data distribution. So far, everything is with a regular autoencoder itself. Now suppose that we make an arbitrary prior $p(z)$ available to the autoencoder by connecting a generative adversarial network (GAN) with the autoencoder that Eq. (12.17) refers to, as shown in Figure 12.8. We then train the GAN to match $q(z)$ with $p(z)$, namely, to have an aggregated posterior to match a prior. The GAN's generator now becomes another encoder to the autoencoder $q(z|\boldsymbol{x})$ above, and the autoencoder attempts to minimize the reconstruction error in the meantime. In the end, the autoencoder learns a deep generative model that maps the imposed prior $p(z)$ to the data distribution.