

翻译并解释以下内容，翻译不能有遗漏，解释请通俗易懂且详尽，因为我的数学基础不太好，最后总结要点。

## 16 Adaptive basis function models

### 16.1 Introduction

In Chapters 14 and 15, we discussed kernel methods, which provide a powerful way to create nonlinear models for regression and classification. The prediction takes the form  $f(x) = w^T \varphi(x)$ , where we define

$$\varphi(x) = [\kappa(x, \mu_1), \dots, \kappa(x, \mu_N)] \quad (16.1)$$

and where  $\mu_k$  are either all the training data or some subset. Models of this form essentially perform a form of template matching, whereby they compare the input  $x$  to the stored prototypes  $\mu_k$ .

Although this can work well, it relies on having a good kernel function to measure the similarity between data vectors. Often coming up with a good kernel function is quite difficult. For example, how do we define the similarity between two images? Pixel-wise comparison of intensities (which is what a Gaussian kernel corresponds to) does not work well. Although it is possible (and indeed common) to hand-engineer kernels for specific tasks (see e.g., the pyramid match kernel in Section 14.2.7), it would be more interesting if we could learn the kernel.

In Section 15.2.4, we discussed a way to learn the parameters of a kernel function, by maximizing the marginal likelihood. For example, if we use the ARD kernel,

$$\kappa(x, x') = \theta_0 \exp(-\frac{1}{2} \sum_{j=1}^D \theta_j (x_j - x'_j)^2) \quad (16.2)$$

we can estimate the  $\theta_j$ , and thus perform a form of nonlinear feature selection. However, such methods can be computationally expensive. Another approach, known as multiple kernel learning (see e.g., (Rakotomamonjy et al. 2008)) uses a convex combination of base kernels,  $\kappa(x, x') = \sum j w_j \kappa_j(x, x')$ , and then estimates the mixing weights  $w_j$ . But this relies on having good base kernels (and is also computationally expensive).

An alternative approach is to dispense with kernels altogether, and try to learn useful features  $\varphi(x)$  directly from the input data. That is, we will create what we call an adaptive basisfunction model (ABM), which is a model of the form

$$f(x) = w_0 + \sum_{m=1}^M w_m \phi_m(x) \quad (16.3)$$

where  $\phi_m(x)$  is the  $m$ 'th basis function, which is learned from data. This framework covers all of the models we will discuss in this chapter.

Typically the basis functions are parametric, so we can write  $\phi_m(x) = \varphi(x; \theta_m)$ , where  $\theta_m$  are the parameters of the basis function itself. We will use  $\Theta = (w_0, w_1:M, \{\theta_m\}_{m=1}^M)$  to denote the entire parameter set. The resulting model is not linear-in-the-parameters anymore, so we will only be able to compute a locally optimal MLE or MAP estimate of  $\Theta$ . Nevertheless, such models often significantly outperform linear models, as we will see.

已思考 23s

下面先给出“逐字不遗漏的中文翻译”（保留原有小节与公式编号），然后用通俗解释把关键思想讲清楚，最后给出要点总结。原文里有几处公式排版疑似缺少求和符号或撇号 $(x')$ ，我会在解释部分特别标注“更常见写法”（不改变逐字翻译的完整性）。

## 翻译

### 16 自适应基函数模型

#### 16.1 引言

在第14和第15章中，我们讨论了核方法，它为回归和分类创建非线性模型提供了一种强有力的方式。预测具有如下形式

$f(x) = w^\top \phi(x)$ ，其中我们定义

$$\phi(x) = [\kappa(x, \mu_1), \dots, \kappa(x, \mu_N)] \quad (16.1)$$

并且  $\mu_k$  要么是所有训练数据，要么是其中的某个子集。此类形式的模型本质上执行了一种模板匹配：它们将输入  $x$  与存储的原型  $\mu_k$  进行比较。

尽管这在很多情况下行之有效，但它依赖于**拥有一个好的核函数**来度量数据向量之间的相似性。而往往要想出一个好的核函数是很困难的。比如，如何定义两幅图像之间的相似性？逐像素的强度比较（高斯核所对应的做法）并不奏效。虽然**为特定任务手工设计核函数**是可行的（而且确实很常见）（例如见第14.2.7节中的金字塔匹配核），但如果能够**学习这个核**会更有趣。

在第15.2.4节中，我们讨论了一种通过**最大化边际似然**来学习核函数参数的方法。比如，如果我们使用 ARD 核，

$$\kappa(x, x') = \theta_0 \exp\left(-\frac{1}{2} \sum_{j=1}^D \theta_j (x_j - x'_j)^2\right) \quad (16.2)$$

我们就可以估计  $\theta_j$ , 从而执行一种非线性特征选择。然而, 这类方法在计算上可能代价高昂。另一种方法, 被称为**多核学习** (参见例如 (Rakotomamonjy 等, 2008)) , 使用**基核的凸组合**,

$\kappa(x, x') = \sum_j w_j \kappa_j(x, x')$ , 然后估计混合权重  $w_j$ 。但这依赖于拥有好的基核 (而且同样在计算上代价高)。

一种替代方法是**完全不使用核**, 而是尝试直接从输入数据中学习有用的特征  $\phi(x)$ 。也就是说, 我们将创建所谓的**自适应基函数模型 (ABM)** , 其模型形式为

$$f(x) = w_0 + \sum_{m=1}^M w_m \phi_m(x) \quad (16.3)$$

其中  $\phi_m(x)$  是第  $m$  个基函数, 它是从数据中学习得到的。我们将在本章讨论的所有模型都属于这一框架。

通常基函数是参数化的, 因此我们可以写成  $\phi_m(x) = \phi(x; v_m)$ , 其中  $v_m$  是该基函数自身的参数。我们用

$\theta = (w_0, w_{1:M}, \{v_m\}_{m=1}^M)$  表示整个参数集合。所得模型就**不再是线性的**, 因此我们只能计算  $\theta$  的**局部最优**的 MLE 或 MAP 估计。尽管如此, 正如我们将看到的, 这类模型往往能显著优于线性模型。

注: 上面翻译完全对应原文; 原文的若干公式处存在撇号 ( $x'$ ) 与求和符号排版疑似缺失的情况, 我在下一节“解释”里会给出更常见的写法, 帮助理解。

## 解释 (通俗易懂版)

### 1) 核方法在做什么? 为什么会有局限

- **基本想法:** 用“相似度”来做预测。把一个输入  $x$  送进模型, 模型会把它分别和若干“原型”  $\mu_1, \dots, \mu_N$  比一比相似度 (由核函数  $\kappa(\cdot, \cdot)$  给出), 把这些相似度排在一个向量  $\phi(x)$  里, 再用权重  $w$  做加权求和得到预测  $f(x) = w^\top \phi(x)$ 。
- **模板匹配:** 直觉上就是“看这个新样本像不像我记忆里的某些典型样本”, 像就给高分, 不像就给低分。
- **难点:** 一切都靠核函数定义的相似性。但很多数据 (如图片、语音、文本) 中, “像不像”并不等于“欧氏距离近不近”。例如两张同一个物体的图片如果有平移、旋转、光照

变化，逐像素比较（相当于高斯核）就会觉得“不像”，这就**不鲁棒**。

## 2) 能不能把核学出来？

- **边际似然调参 (15.2.4节)**：以 ARD 核为例，它给每个特征维度一个长度尺度或权重  $\theta_j$ 。通过最大化边际似然，可以**学到哪些维度更重要**（非线性特征选择）。
  - **更常见写法 (帮助理解)**：

$$\kappa(x, x') = \theta_0 \exp\left(-\frac{1}{2} \sum_{j=1}^D \theta_j (x_j - x'_j)^2\right)$$

这里的  $x'$  表示另一条样本。若某个  $\theta_j$  很大，表示在第  $j$  维上两点稍微有差就会被判很不相似，等价于强调该维度；反之则弱化该维度。

- **问题**：这种方法在计算上**很贵**（比如高斯过程里常见的复杂度会随样本数大幅增长）。
- **多核学习 (MKL)**：把多个现成的“基核”线性组合，学它们的权重。
  - **更常见写法 (帮助理解)**：
$$\kappa(x, x') = \sum_j w_j \kappa_j(x, x').$$
  - **问题**：仍然依赖你手头有一批好用的基核；而且求解也不便宜。

## 3) 退一步：别用核，直接学“特征/基函数”

- **核心想法 (ABM)**：别再绞尽脑汁定义“相似度”了，**直接从数据里学出一组有用的“基函数/特征”**  $\phi_1(x), \dots, \phi_M(x)$ ，然后做个线性组合：

$$f(x) = w_0 + \sum_{m=1}^M w_m \phi_m(x)$$

- **和核方法的关系**：核方法中的  $\phi(x)$  是由“和原型比相似度”**间接定义**出来的；ABM 则是把  $\phi_m(x)$ **直接当成要学的东西**。这等于把“相似度设计问题”变成了“特征学习问题”。

## 4) 什么是“参数化基函数”？为什么会变“非线性参数模型”

- **参数化**：通常让每个基函数带上自己的参数，比如  $\phi_m(x) = \phi(x; v_m)$ 。举个直观例子：
  - 在一维情形，可以让  $\phi(x; v) = \exp(-(x - c)^2/(2s^2))$ ，其中参数  $v = (c, s)$  就是“中心位置”和“宽度”。这样每个基函数就是一个“钟形包”（高斯 bump）。
  - 学习这些  $v_m$ （每个 bump 在哪里、宽度多大）再配合权重  $w_m$ ，就能让模型在数据密集的地方“亮起来”，在别处“暗下来”，从而拟合复杂的非线性关系。
- **非线性-关于参数**：虽然  $f(x)$  对 权重  $w_m$  是线性的，但由于  $\phi_m$  里还嵌着参数  $v_m$ ，整套参数  $\theta = (w_0, w_{1:M}, \{v_m\})$  进入模型是**非线性的**。因此：

- 我们通常只能找到局部最优的极大似然 (MLE) 或最大后验 (MAP) 解;
- 训练上常用梯度法或其变体，并配合随机初始化、正则化等技巧。
- 但值得：尽管求解更难，这类模型往往比简单线性模型好很多，因为它们可以“自己长出”合适的特征来描述数据的非线性结构。

## 5) 和你可能熟悉的东西的联系

- 神经网络可以看成是 ABM 的典型代表：
    - 每一层的神经元都在实现一些“基函数”（例如  $\phi(x; v) = \text{ReLU}(a^\top x + b)$ ），参数  $v = (a, b)$  是可学习的；
    - 整体输出就是这些基函数的线性组合再经过若干堆叠；
    - 训练时确实只能求到局部最优，但效果通常很好。
  - \*\*径向基函数网络 (RBFN) \*\*也是 ABM：
    - 基函数常取高斯形状  $\exp(-\|x - c_m\|^2/(2s_m^2))$ ；
    - 学习中心  $c_m$ 、尺度  $s_m$  与权重  $w_m$ 。
- 

## 关键“更常见写法”提示（帮助对照原文公式）

- ARD 核 (16.2)：原文排版里  $x$  与  $x'$  之间的撇号疑似缺失，更通行的写法是

$$\kappa(x, x') = \theta_0 \exp\left(-\frac{1}{2} \sum_{j=1}^D \theta_j (x_j - x'_j)^2\right).$$

- 多核学习：原文在“ $\sum_j w_j \kappa_j$ ”前后省略了求和符号，常见写法是  
 $\kappa(x, x') = \sum_j w_j \kappa_j(x, x')$ 。
  - ABM (16.3)：原文“ $Mm = 1$ ”应理解为“从  $m = 1$  到  $M$  的求和”，即  
 $f(x) = w_0 + \sum_{m=1}^M w_m \phi_m(x)$ 。
- 

## 要点总结

- 核方法把预测写成“对原型的相似度”的线性组合；好用但依赖好核。
- 为核调参（如 ARD）或多核学习能部分缓解，但计算开销大且仍依赖好基核。
- 自适应基函数模型 (ABM)：不再设计相似度，而是直接学习基函数/特征，再线性组合输出。
- 基函数带自有参数（位置、尺度、方向等），使得模型对参数是非线性的；训练时通常只能得局部最优的 MLE/MAP。

- 尽管求解更难，ABM往往显著优于线性模型；神经网络、RBF网络都可视作ABM的实例。

翻译并解释以下内容，翻译不能有遗漏，解释请通俗易懂且详尽，因为我的数学基础不太好，最后总结要点。

## 16.2 Classification and regression trees (CART)

Classification and regression trees or CART models, also called decision trees (not to be confused with the decision trees used in decision theory) are defined by recursively partitioning the input space, and defining a local model in each resulting region of input space. This can be represented by a tree, with one leaf per region, as we explain below.

### 16.2.1 Basics

To explain the CART approach, consider the tree in Figure 16.1(a). The first node asks if  $x_1$  is less than some threshold  $t_1$ . If yes, we then ask if  $x_2$  is less than some other threshold  $t_2$ . If yes, we are in the bottom left quadrant of space,  $R_1$ . If no, we ask if  $x_1$  is less than  $t_3$ . And so on. The result of these axis parallel splits is to partition 2d space into 5 regions, as shown in Figure 16.1(b). We can now associate a mean response with each of these regions, resulting in the piecewise constant surface shown in Figure 16.1(c). We can write the model in the following form

$$f(x) = E[y|x] = \sum_{m=1}^M w_m I(x \in R_m) = \sum_{m=1}^M w_m \varphi(x; v_m) \quad (16.4)$$

where  $R_m$  is the  $m$ 'th region,  $w_m$  is the mean response in this region, and  $v_m$  encodes the choice of variable to split on, and the threshold value, on the path from the root to the  $m$ 'th leaf. This makes it clear that a CART model is just a an adaptive basis-function model, where the basis functions define the regions, and the weights specify the response value in each region. We discuss how to find these basis functions below. We can generalize this to the classification setting by storing the distribution over class labels in each leaf, instead of the mean response. This is illustrated in Figure 16.2. This model can be used to

classify the data in Figure 1.1. For example, we first check the color of the object. If it is blue, we follow the left branch and end up in a leaf labeled "4,0", which means we have 4 positive examples and 0 negative examples which match this criterion. Hence we predict  $p(y = 1|x) = 4/4$  if  $x$  is blue. If it is red, we then check the shape: if it is an ellipse, we end up in a leaf labeled "1,1", so we predict  $p(y = 1|x) = 1/2$ . If it is red but not an ellipse, we predict  $p(y = 1|x) = 0/2$ ; If it is some other colour, we check the size: if less than 10, we predict  $p(y = 1|x) = 4/4$ , otherwise  $p(y = 1|x) = 0/5$ . These probabilities are just the empirical fraction of positive examples that satisfy each conjunction of feature values, which defines a path from the root to a leaf.

### 16.2.2 Growing a tree

Finding the optimal partitioning of the data is NP-complete (Hyafil and Rivest 1976), so it is common to use the greedy procedure shown in Algorithm 6 to compute a locally optimal MLE. This method is used by CART, (Breiman et al. 1984) C4.5(Quinlan 1993), and ID3 (Quinlan 1986), which are three popular implementations of the method. (See dtfit for a simple Matlab implementation.) The split function chooses the best feature, and the best value for that feature, as follows:

$$(j^*, t^*) = \arg \min_{j \in \{1, \dots, D\}} \min_{t \in T_j} \text{cost}(\{x_i, y_i : x_{ij} \leq t\}) + \text{cost}(\{x_i, y_i : x_{ij} > t\}) \quad (16.5)$$

where the cost function for a given dataset will be defined below. For notational simplicity, we have assumed all inputs are real-valued or ordinal, so it makes sense to compare a feature  $x_{ij}$  to a numeric value  $t$ . The set of possible thresholds  $T_j$  for feature  $j$  can be obtained by sorting the unique values of  $x_{ij}$ . For example, if feature 1 has the values {4.5, -12, 72, -12}, then we set  $T_1 = \{-12, 4.5, 72\}$ . In the case of categorical inputs, the most common approach is to consider splits of the form  $x_{ij} = c_k$  and  $x_{ij} \neq c_k$ , for each possible class label  $c_k$ . Although we could allow for multi-way splits (resulting in non-binary trees), this would result in data fragmentation, meaning too little data might "fall" into each subtree, resulting in overfitting. The function that checks if a node is worth splitting can use several stopping heuristics, such as the following:

- is the reduction in cost too small? Typically we define the gain of using a feature to be a normalized

measure of the reduction in cost:  $\Delta \text{cost}(D) - |\text{DL}| |\text{D}| \text{cost}(\text{DL}) + |\text{DR}| |\text{D}| \text{cost}(\text{DR})$  (16.6)

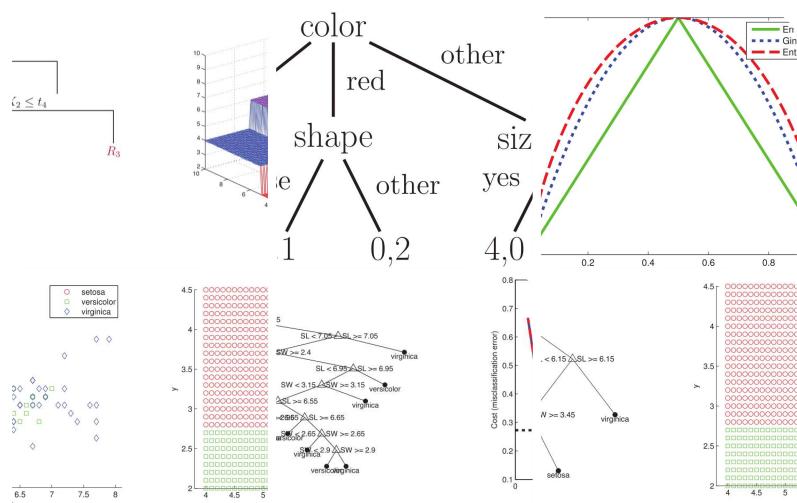
- has the tree exceeded the maximum desired depth?
- is the distribution of the response in either DL or DR sufficiently homogeneous (e.g., all labels are the same, so the distribution is pure)?
- is the number of examples in either DL or DR too small?

All that remains is to specify the cost measure used to evaluate the quality of a proposed split. This depends on whether our goal is regression or classification. We discuss both cases below.

### 16.2.2.1 Regression cost

In the regression setting, we define the cost as follows:  $\text{cost}(D) = \sum_{i \in D} (y_i - \bar{y})^2$  (16.7)

已思考



翻译并解释以下内容，翻译不能有遗漏，解释请通俗易懂且详尽，因为我的数学基础不太好，最后总结要点。

Figure 16.1 A simple regression tree on two inputs.

Figure 16.2 A simple decision tree for the data in Figure 1.1. A leaf labeled as  $(n_1, n_0)$  means that there are  $n_1$  positive examples that match this path, and  $n_0$  negative examples. In this tree, most of the leaves are “pure”, meaning they only have examples of one class or the other; the only exception is leaf representing red ellipses, which has a label distribution of  $(1, 1)$ . We could distinguish positive from negative red ellipses by adding a further test based on size. However, it is not always desirable to construct trees that perfectly model the

training data, due to overfitting

Figure 16.3 Node impurity measures for binary classification. The horizontal axis corresponds to  $p$ , the probability of class 1. The entropy measure has been rescaled to pass through (0.5,0.5).

Figure 16.4 (a) Iris data. We only show the first two features, sepal length and sepal width, and ignore petal length and petal width. (b) Decision boundaries induced by the decision tree in Figure 16.5(a).

Figure 16.5 (a) Unpruned decision tree for Iris data. (b) Plot of misclassification error rate vs depth of tree.

Figure 16.6 Pruned decision tree for Iris data.

## 16.2 Classification and regression trees (CART)

Classification and regression trees or CART models, also called decision trees (not to be confused with the decision trees used in decision theory) are defined by recursively partitioning the input space, and defining a local model in each resulting region of input space. This can be represented by a tree, with one leaf per region, as we explain below.

### 16.2.1 Basics

To explain the CART approach, consider the tree in Figure 16.1(a). The first node asks if  $x_1$  is less than some threshold  $t_1$ . If yes, we then ask if  $x_2$  is less than some other threshold  $t_2$ . If yes, we are in the bottom left quadrant of space,  $R_1$ . If no, we ask if  $x_1$  is less than  $t_3$ . And so on. The result of these axis parallel splits is to partition 2d space into 5 regions, as shown in Figure 16.1(b). We can now associate a mean response with each of these regions, resulting in the piecewise constant surface shown in Figure 16.1(c). We can write the model in the following form

$$f(x) = E[y|x] = \sum_{m=1}^M w_m I(x \in R_m) = \sum_{m=1}^M w_m \varphi(x; v_m) \quad (16.4)$$

where  $R_m$  is the  $m$ 'th region,  $w_m$  is the mean response in this region, and  $v_m$  encodes the choice of variable to split on, and the threshold value, on the path from the root to the  $m$ 'th leaf. This makes it clear that a CART model is just a an adaptive basis-function model, where the basis functions define the regions, and the weights specify the response value in each region. We discuss how to find these basis functions below. We can generalize this to the

classification setting by storing the distribution over class labels in each leaf, instead of the mean response. This is illustrated in Figure 16.2. This model can be used to classify the data in Figure 1.1. For example, we first check the color of the object. If it is blue, we follow the left branch and end up in a leaf labeled "4,0", which means we have 4 positive examples and 0 negative examples which match this criterion. Hence we predict  $p(y=1|x)=4/4$  if  $x$  is blue. If it is red, we then check the shape: if it is an ellipse, we end up in a leaf labeled "1,1", so we predict  $p(y=1|x)=1/2$ . If it is red but not an ellipse, we predict  $p(y=1|x)=0/2$ ; If it is some other colour, we check the size: if less than 10, we predict  $p(y=1|x)=4/4$ , otherwise  $p(y=1|x)=0/5$ . These probabilities are just the empirical fraction of positive examples that satisfy each conjunction of feature values, which defines a path from the root to a leaf.

### 16.2.2 Growing a tree

Finding the optimal partitioning of the data is NP-complete (Hyafil and Rivest 1976), so it is common to use the greedy procedure shown in Algorithm 6 to compute a locally optimal MLE. This method is used by CART, (Breiman et al. 1984) C4.5(Quinlan 1993), and ID3 (Quinlan 1986), which are three popular implementations of the method. (See dtfit for a simple Matlab implementation.) The split function chooses the best feature, and the best value for that feature, as follows:

$$(j^*, t^*) = \arg \min_{j \in \{1, \dots, D\}} \min_{t \in T_j} \text{cost}(\{x_i, y_i : x_{ij} \leq t\}) + \text{cost}(\{x_i, y_i : x_{ij} > t\}) \quad (16.5)$$

where the cost function for a given dataset will be defined below. For notational simplicity, we have assumed all inputs are real-valued or ordinal, so it makes sense to compare a feature  $x_{ij}$  to a numeric value  $t$ . The set of possible thresholds  $T_j$  for feature  $j$  can be obtained by sorting the unique values of  $x_{ij}$ . For example, if feature 1 has the values {4.5, -12, 72, -12}, then we set  $T_1 = \{-12, 4.5, 72\}$ . In the case of categorical inputs, the most common approach is to consider splits of the form  $x_{ij} = c_k$  and  $x_{ij} \neq c_k$ , for each possible class label  $c_k$ . Although we could allow for multi-way splits (resulting in non-binary trees), this would result in data fragmentation, meaning too little data might "fall" into each subtree, resulting in overfitting. The function that checks if a node is worth

splitting can use several stopping heuristics, such as the following:

- is the reduction in cost too small? Typically we define the gain of using a feature to be a normalized measure of the reduction in cost:

$$\Delta = \text{cost}(D) - |DL| |D| \text{cost}(DL) + |DR| |D| \text{cost}(DR) \quad (16.6)$$

- has the tree exceeded the maximum desired depth?
- is the distribution of the response in either DL or DR sufficiently homogeneous (e.g., all labels are the same, so the distribution is pure)?

- is the number of examples in either DL or DR too small?

All that remains is to specify the cost measure used to evaluate the quality of a proposed split. This depends on whether our goal is regression or classification. We discuss both cases below.

#### 16.2.2.1 Regression cost

In the regression setting, we define the cost as follows:

$$\text{cost}(D) = \sum_{i \in D} (y_i - \bar{y})^2 \quad (16.7)$$

where  $\bar{y} = \frac{1}{|D|} \sum_{i \in D} y_i$  is the mean of the response variable in the specified set of data. Alternatively, we can fit a linear regression model for each leaf, using as inputs the features that were chosen on the path from the root, and then measure the residual error.

#### 16.2.2.2 Classification cost

In the classification setting, there are several ways to measure the quality of a split. First, we fit a multinoulli model to the data in the leaf satisfying the test  $X_j < t$  by estimating the class-conditional probabilities as follows:

$$\hat{\pi}_c = \frac{1}{|D|} \sum_{i \in D} I(y_i = c) \quad (16.8)$$

where  $D$  is the data in the leaf. Given this, there are several common error measures for evaluating a proposed partition:

- Misclassification rate. We define the most probable class label as  $\hat{y}_c = \arg\max_c \hat{\pi}_c$ . The corresponding error rate is then

$$\sum_{i \in D} I(y_i \neq \hat{y}_c) = 1 - \sum_{i \in D} I(y_i = \hat{y}_c) \quad (16.9)$$

- Entropy, or deviance:

$$H(\hat{\pi}) = - \sum_{c=1}^C \hat{\pi}_c \log \hat{\pi}_c \quad (16.10)$$

Note that minimizing the entropy is equivalent to maximizing the information gain (Quinlan 1986) between test  $X_j < t$  and the class label  $Y$ , defined by

$$\text{infoGain}(X_j < t, Y) = H(Y) - H(Y|X_j < t) \quad (16.11)$$

$$= - \sum_{c=1}^C p(y=c) \log p(y=c) \quad (16.12)$$

$$+ c p(y = c|X_j < t) \log p(c|X_j < t) \quad (16.13)$$

since  $\hat{\pi}_c$  is an MLE for the distribution  $p(c|X_j < t)$ .

- Gini index

$$C c=1 \pi^c (1 - \hat{\pi}_c) = c \pi^c c - c \pi^c 2 c = 1 - c \pi^c 2 c \quad (16.14)$$

This is the expected error rate. To see this, note that  $\hat{\pi}_c$  is the probability a random entry in the leaf belongs to class  $c$ , and  $(1 - \hat{\pi}_c)$  is the probability it would be misclassified.

In the two-class case, where  $p = \pi_m(1)$ , the misclassification rate is  $1 - \max(p, 1 - p)$ , the entropy is  $H_2(p)$ , and the Gini index is  $2p(1 - p)$ . These are plotted in Figure 16.3. We see that the cross-entropy and Gini measures are very similar, and are more sensitive to changes in class probability than is the misclassification rate. For example, consider a two-class problem with 400 cases in each class. Suppose one split created the nodes  $(300, 100)$  and  $(100, 300)$ , while the other created the nodes  $(200, 400)$  and  $(200, 0)$ . Both splits produce a misclassification rate of 0.25. However, the latter seems preferable, since one of the nodes is pure, i.e., it only contains one class. The cross-entropy and Gini measures will favor this latter choice.

### 16.2.2.3 Example

As an example, consider two of the four features from the 3-class iris dataset, shown in Figure 16.4(a). The resulting tree is shown in Figure 16.5(a), and the decision boundaries are shown in Figure 16.4(b). We see that the tree is quite complex, as are the resulting decision boundaries. In Figure 16.5(b), we show that the CV estimate of the error is much higher than the training set error, indicating overfitting. Below we discuss how to perform a tree-pruning stage to simplify the tree.

### 16.2.3 Pruning a tree

To prevent overfitting, we can stop growing the tree if the decrease in the error is not sufficient to justify the extra complexity of adding an extra subtree. However, this tends to be too myopic. For example, on the xor data in Figure 14.2(c), it would never make any splits, since each feature on its own has little predictive power. The standard approach is therefore to grow a “full” tree, and then to perform pruning. This can be done using a scheme that prunes the branches giving the least increase in the error. See (Breiman et al. 1984) for details. To determine how far to prune back, we can evaluate the

cross-validated error on each such subtree, and then pick the tree whose CV error is within 1 standard error of the minimum. This is illustrated in Figure 16.4(b). The point with the minimum CV error corresponds to the simple tree in Figure 16.6(a).

#### 16.2.4 Pros and cons of trees

CART models are popular for several reasons: they are easy to interpret<sup>2</sup>, they can easily handle mixed discrete and continuous inputs, they are insensitive to monotone transformations of the inputs (because the split points are based on ranking the data points), they perform automatic variable selection, they are relatively robust to outliers, they scale well to large data sets, and they can be modified to handle missing inputs.<sup>3</sup> However, CART models also have some disadvantages. The primary one is that they do not predict very accurately compared to other kinds of model. This is in part due to the greedy nature of the tree construction algorithm. A related problem is that trees are unstable: small changes to the input data can have large effects on the structure of the tree, due to the hierarchical nature of the tree-growing process, causing errors at the top to affect the rest of the tree. In frequentist terminology, we say that trees are high variance estimators. We discuss a solution to this below.

#### 16.2.5 Random forests

One way to reduce the variance of an estimate is to average together many estimates. For example, we can train  $M$  different trees on different subsets of the data, chosen randomly with replacement, and then compute the ensemble

$$f(x) = \frac{1}{M} \sum_{m=1}^M f_m(x) \quad (16.15)$$

where  $f_m$  is the  $m$ 'th tree. This technique is called bagging (Breiman 1996), which stands for "bootstrap aggregating". Unfortunately, simply re-running the same learning algorithm on different subsets of the data can result in highly correlated predictors, which limits the amount of variance reduction that is possible. The technique known as random forests (Breiman 2001a) tries to decorrelate the base learners by learning trees based on a randomly chosen subset of input variables, as well as a randomly chosen subset of data cases. Such models often have very good predictive accuracy (Caruana and Niculescu-Mizil 2006), and have been widely used in many applications

(e.g., for body pose recognition using Microsoft's popular kinect sensor (Shotton et al. 2011)). Bagging is a frequentist concept. It is also possible to adopt a Bayesian approach to learning trees. In particular, (Chipman et al. 1998; Denison et al. 1998; Wu et al. 2007) perform approximate inference over the space of trees (structure and parameters) using MCMC. This reduces the variance of the predictions. We can also perform Bayesian inference over the space of ensembles of trees, which tends to work much better. This is known as Bayesian adaptive regression trees or BART (Chipman et al. 2010). Note that the cost of these sampling-based Bayesian methods is comparable to the sampling-based random forest method. That is, both approaches are fairly slow to train, but produce high quality classifiers. Unfortunately, methods that use multiple trees (whether derived from a Bayesian or frequentist standpoint) lose their nice interpretability properties. Fortunately, various post-processing measures can be applied, as discussed in Section 16.8.

16.2.6 CART compared to hierarchical mixture of experts  
An interesting alternative to a decision tree is known as the hierarchical mixture of experts. Figure 11.7(b) gives an illustration where we have two levels of experts. This can be thought of as a probabilistic decision tree of depth 2, since we recursively partition the space, and apply a different expert to each partition. Hastie et al. (Hastie et al. 2009, p331) write that "The HME approach is a promising competitor to CART trees". Some of the advantages include the following:

- The model can partition the input space using any set of nested linear decision boundaries. By contrast, standard decision trees are constrained to use axis-parallel splits.
- The model makes predictions by averaging over all experts. By contrast, in a standard decision tree, predictions are made only based on the model in the corresponding leaf. Since leaves often contain few training examples, this can result in overfitting.
- Fitting an HME involves solving a smooth continuous optimization problem (usually using EM), which is likely to be less prone to local optima than the standard greedy discrete optimization methods used to fit decision trees. For similar reasons, it is computationally easier to "be

Bayesian" about the parameters of an HME (see e.g., (Peng et al. 1996; Bishop and Svensén 2003)) than about the structure and parameters of a decision tree (see e.g., (Wu et al. 2007)).

已思考 1m 1s

下面先给出逐字不遗漏的中文翻译（保留原有图题、小节与公式编号），随后是通俗且详尽的解释，最后附要点总结。文中个别公式在原排版里有缺失或小错（如缺少撇号 $x'$ 、权重号、或少了求和的“ $\Sigma$ ”），我在解释部分会标出“更常见写法”，方便理解，但翻译部分我严格对应原文。

---

## 翻译

**Figure 16.1** 两个输入上的一个简单回归树。

**Figure 16.2** 针对图 1.1 中数据的一个简单决策树。被标注为  $(n_1, n_0)$  的叶子表示：有  $n_1$  个满足该路径的正类样本，以及  $n_0$  个负类样本。在这棵树里，大多数叶子是“纯的”，意思是它们只包含某一类或另一类；唯一的例外是表示红色椭圆的叶子，其标签分布为  $(1, 1)$ 。我们可以基于大小再加一个测试来区分正的与负的红色椭圆。然而，并不总是希望构造能完美拟合训练数据的树，因为会导致过拟合。

**Figure 16.3** 二分类的结点不纯度度量。横轴对应  $p$ ，即类别1的概率。熵度量已经重标定，使其通过点  $(0.5, 0.5)$ 。

**Figure 16.4** (a) 鸢尾花数据。我们只展示前两个特征：萼片长度和萼片宽度，忽略花瓣长度与花瓣宽度。(b) 由图 16.5(a) 的决策树所诱导的决策边界。

**Figure 16.5** (a) 鸢尾花数据的未剪枝决策树。(b) 误分类错误率随树深度变化的曲线。

**Figure 16.6** 鸢尾花数据的剪枝后决策树。

---

## 16.2 分类与回归树 (CART)

分类与回归树，或 CART 模型，也被称为决策树（不要与决策理论里使用的决策树混淆），其定义方式是递归地划分输入空间，并在每个由此产生的输入空间区域里定义一个局部模型。正如下文所述，这可以用一棵树来表示，每个区域对应一个叶子。

### 16.2.1 基本原理

为说明 CART 方法，考虑图 16.1(a) 中的那棵树。第一个结点询问  $x_1$  是否小于某个阈值  $t_1$ 。如果是，我们接着问  $x_2$  是否小于另一个阈值  $t_2$ 。如果是，我们位于空间的左下象限， $R_1$ 。如果不是，我们询问  $x_1$  是否小于  $t_3$ 。以此类推。这些与坐标轴平行的切分的结果是

把二维空间划成 5 个区域，如图 16.1(b) 所示。现在我们可以为每个区域关联一个**均值响应**，从而得到图 16.1(c) 所示的**分段常数曲面**。我们可将模型写成如下形式

$$f(x) = E[y|x] = \sum_{m=1}^M w_m I(x \in R_m) = \sum_{m=1}^M w_m \phi(x; v_m) \quad (16.4)$$

其中  $R_m$  是第  $m$  个区域， $w_m$  是该区域中的均值响应，而  $v_m$  编码了从根到第  $m$  个叶子的路径上“选择哪一个变量去切分、以及阈值是多少”。这清楚地表明，CART 模型正是一种**自适应基函数模型**：基函数定义了区域，而权重指定了各区域中的响应值。关于如何找到这些基函数，我们将在下文讨论。

我们可以将其推广到**分类**情形：在每片叶子里不再存储均值响应，而是存储类别标签的分布。图 16.2 展示了这一点。该模型可用于分类图 1.1 中的数据。例如，我们首先检查物体的颜色。如果是蓝色，我们沿左分支走，落到一个标注为“4,0”的叶子，这意味着满足该条件的正样本有 4 个、负样本有 0 个。因此，如果  $x$  是蓝色，我们预测  $p(y=1|x) = 4/4$ 。如果是红色，我们接着检查形状：若是椭圆，我们落到一个标注为“1,1”的叶子，于是预测  $p(y=1|x) = 1/2$ 。如果是红色但不是椭圆，我们预测  $p(y=1|x) = 0/2$ ；如果是其他颜色，我们检查大小：若小于 10，预测  $p(y=1|x) = 4/4$ ，否则  $p(y=1|x) = 0/5$ 。这些概率就是**满足从根到叶子这条特征取值合取式的正样本所占的经验比例**。

### 16.2.2 生长一棵树

找到数据的**最优划分**是 NP 完全 (Hyafil and Rivest 1976)，因此通常采用算法 6 所示的**贪心过程**来计算一个**局部最优的 MLE**。该方法被 CART (Breiman 等, 1984)、C4.5 (Quinlan, 1993) 与 ID3 (Quinlan, 1986) 采用，它们是该方法的三种流行实现。（参见 *dtfit* 获取一个简单的 Matlab 实现。）

`split` 函数按如下方式选择最佳特征以及该特征的最佳取值：

$$(j^*, t^*) = \arg \min_{j \in \{1, \dots, D\}} \min_{t \in T_j} \text{cost}(\{x_i, y_i : x_{ij} \leq t\}) + \text{cost}(\{x_i, y_i : x_{ij} > t\}) \quad (16.5)$$

其中针对给定数据集的**代价函数**将在下文定义。为记号简洁，我们假设所有输入都是**实值或有序的**，因此比较特征  $x_{ij}$  与一个数值  $t$  是有意义的。特征  $j$  的**可能阈值集合**  $T_j$  可通过对  $x_{ij}$  的唯一取值进行排序获得。例如，如果特征 1 取值为  $\{4.5, -12, 72, -12\}$ ，则我们令  $T_1 = \{-12, 4.5, 72\}$ 。

对**类别型输入**，最常见做法是考虑形如  $x_{ij} = c_k$  与  $x_{ij} \neq c_k$  的切分（对每一个可能的类别标签  $c_k$ ）。虽然我们也可以允许多路切分（从而得到非二叉树），但这会导致**数据碎片化**，即每个子树“落下”的数据太少，从而造成过拟合。

用于检查一个结点是否值得切分的函数可以使用若干**停止启发式**，例如：

- **代价降低是否太小？** 通常我们将使用某个特征带来的“增益”定义为代价降低的**归一化度量**：

$$\Delta = \text{cost}(D) - \frac{|D_L|}{|D|} \text{cost}(D_L) + \frac{|D_R|}{|D|} \text{cost}(D_R) \quad (16.6)$$

- 树是否超过了期望的最大深度？

- $D_L$  或  $D_R$  中的响应分布是否足够同质（例如，所有标签都一样，因此分布是纯的）？

- $D_L$  或  $D_R$  中的样本数是否过少？

剩下要做的就是指定用于评估一个候选切分质量的**代价度量**。这取决于是回归还是分类。

我们在下文分别讨论。

### 16.2.2.1 回归代价

在回归设定中，我们将代价定义为

$$\text{cost}(D) = \sum_{i \in D} (y_i - \bar{y})^2 \quad (16.7)$$

其中  $\bar{y} = \frac{1}{|D|} \sum_{i \in D} y_i$  是指定数据集合中响应变量的**均值**。或者，我们也可以为每个叶子拟合一个**线性回归模型**（使用从根到该叶子路径上被选中的那些特征作为输入），然后度量**残差误差**。

### 16.2.2.2 分类代价

在分类设定中，度量一个切分质量的方法有多种。首先，我们对满足测试  $X_j < t$  的叶子中的数据拟合一个**多项分布**（multinoulli），通过如下方式估计类条件概率：

$$\hat{\pi}_c = \frac{1}{|D|} \sum_{i \in D} \mathbf{I}(y_i = c) \quad (16.8)$$

其中  $D$  是该叶子中的数据。给定此估计后，评估候选划分的常见误差度量包括：

- **误分类率**（Misclassification rate）。我们将最可能的类别定义为  $\hat{y} = \arg \max_c \hat{\pi}_c$ 。相应的错误率为

$$\frac{1}{|D|} \sum_{i \in D} \mathbf{I}(y_i \neq \hat{y}) = 1 - \hat{\pi}_{\hat{y}} \quad (16.9)$$

- **熵（或对数损失 / 偏差，deviance）**：

$$H(\hat{\pi}) = - \sum_{c=1}^C \hat{\pi}_c \log \hat{\pi}_c \quad (16.10)$$

注意，最小化熵等价于最大化测试  $X_j < t$  与类别标签  $Y$  之间的**信息增益**（Quinlan 1986）：

$$\text{infoGain}(X_j < t, Y) = H(Y) - H(Y | X_j < t) \quad (16.11)$$

$$= - \sum_c p(y=c) \log p(y=c) \quad (16.12)$$

$$+ \sum_c p(y=c|X_j < t) \log p(c|X_j < t) \quad (16.13)$$

因为  $\hat{\pi}_c$  是分布  $p(c|X_j < t)$  的极大似然估计。

#### • 基尼指数 (Gini index)

$$\sum_{c=1}^C \hat{\pi}_c (1 - \hat{\pi}_c) = \sum_c \hat{\pi}_c - \sum_c \hat{\pi}_c^2 = 1 - \sum_c \hat{\pi}_c^2 \quad (16.14)$$

这就是**期望错误率**。要看到这一点，注意到  $\hat{\pi}_c$  是叶子中一个随机样本属于类别  $c$  的概率，而  $1 - \hat{\pi}_c$  是它被误分的概率。

在**二分类**情形，令  $p = \pi_m(1)$ 。误分类率为  $1 - \max(p, 1-p)$ ，熵为  $H_2(p)$ ，基尼指数为  $2p(1-p)$ 。这些画在图 16.3 中。我们看到**交叉熵与基尼**非常相似，且相较误分类率对类别概率的变化更敏感。举例：一个二分类问题，两类各有 400 个样本。设一个切分产生结点 (300,100) 与 (100,300)，而另一个切分产生 (200,400) 与 (200,0)。两种切分的误分类率都是 0.25。然而后者似乎更好，因为其中一个结点是**纯的**（只含一种类）。交叉熵与基尼度量会偏好后者。

#### 16.2.2.3 示例

以三类鸢尾花数据的四个特征中的两个为例，见图 16.4(a)。得到的树见图 16.5(a)，其决策边界见图 16.4(b)。可以看到，这棵树相当复杂，所得到的决策边界也很复杂。在图 16.5(b) 中，我们展示：**交叉验证**的误差估计远高于训练集误差，表明过拟合。下面我们讨论如何进行**剪枝阶段**以简化这棵树。

#### 16.2.3 对树进行剪枝

为防止过拟合，我们可以在“新增子树所带来的误差下降不足以抵偿复杂度增加”时停止生长树。然而，这往往过于目光短浅。比如图 14.2(c) 的 xor 数据上，它可能永远不会做切分，因为每个单独特征的预测力都很弱。**标准做法**因此是先长一棵“完整”的树，然后再进行**剪枝**。可以使用一种方案来剪去“使误差增加最少”的分支；细节见 (Breiman 等, 1984)。为了决定剪到什么程度，我们可以对每棵此类子树评估**交叉验证误差**，然后选择“CV 误差在**最小值 +1 个标准误差范围内**”的树。图 16.4(b) 做了演示。对应于最小 CV 误差的点给出图 16.6(a) 中那棵更简单的树。

#### 16.2.4 树的优缺点

CART 模型受欢迎有多重原因：**易解释**<sup>2</sup>、可轻松处理离散与连续混合输入、对输入的**单调变换不敏感**（因为切分点基于对样本的排序）、能**自动做变量选择**、对离群点相对**鲁棒**、能**扩展到大数据集**、并可修改以处理**缺失输入**<sup>3</sup>。

然而, CART 也有缺点。主要缺点是: 与其他类型的模型相比, **预测精度不高**, 部分原因在于树构建算法的**贪心性**。相关的问题是: 树**不稳定**——数据有小变化, 树结构可能大变; 这是因为树的**层级性**会让上层的错误影响到整棵树。用频率学术语, 我们说树是**高方差估计器**。下面我们给出一种解决思路。

### 16.2.5 随机森林

降低估计方差的一种方法是**把多个估计平均**。例如, 我们可在以**有放回**随机抽取的不同数据子集上训练  $M$  棵不同的树, 然后计算集成

$$f(x) = \frac{1}{M} \sum_{m=1}^M f_m(x) \quad (16.15)$$

其中  $f_m$  是第  $m$  棵树。这种技术称为**袋装** (bagging) (Breiman 1996), 意为“自助聚合” (bootstrap aggregating)。不幸的是, 简单地在不同子集上反复运行相同学习算法会产生**高度相关的基学习器**, 从而限制了可实现的方差降低幅度。称为**随机森林** (Breiman 2001a) 的技术试图**去相关**: 不仅在随机的数据子集上建树, 还在每个结点处基于**随机选择的一小部分输入变量**来寻找最优切分。这样的模型通常具有非常好的**预测精度** (Caruana and Niculescu-Mizil 2006), 并已广泛应用 (例如用于微软 Kinect 传感器的人体姿态识别 (Shotton 等, 2011))。

Bagging 是频率学概念。我们也可以采取**贝叶斯方法**来学习树。特别地, (Chipman 等, 1998; Denison 等, 1998; Wu 等, 2007) 使用 MCMC 在“树的空间 (结构与参数)”上做近似推断, 这会降低预测的方差。我们也可以在“树的**集成空间**”上进行贝叶斯推断, 效果往往更好, 这被称为**贝叶斯自适应回归树** (BART) (Chipman 等, 2010)。需要注意, 这些**基于采样的贝叶斯方法**的训练开销与**基于采样的随机森林**方法相当。也就是说, 两者训练都**较慢**, 但能得到**高质量分类器**。不幸的是, 使用多棵树的方法 (无论源于贝叶斯还是频率学立场) 会**丧失良好的可解释性**。好在可以施加各种事后处理措施, 见第 16.8 节讨论。

### 16.2.6 CART 与分层专家混合 (HME) 的比较

一种有趣的决策树替代物被称为**分层专家混合** (hierarchical mixture of experts, HME)。图 11.7(b) 给出了一个有两层专家的示意。它可被视为深度为 2 的**概率决策树**: 我们递归划分空间, 并对每个划分应用不同的专家。Hastie 等 (2009, 第 331 页) 写道: “HME 方法是 CART 树的一个很有前途的竞争者”。其部分优势包括:

- 该模型可以用**任意一组嵌套的线性判别边界**来划分输入空间。相对地, 标准决策树被限制为**与轴平行**的切分。
- 该模型通过对**所有专家的输出求均值**来做预测。相对地, 标准决策树的预测只依赖落到**那片叶子**的模型。由于叶子里常常只有很少的训练样本, 这可能导致过拟合。
- 拟合 HME 涉及求解一个**光滑的连续优化问题** (通常用 EM), 这可能较少陷入局部最

优，而标准决策树使用的是离散的贪心优化。出于类似原因，相比于对“决策树的结构与参数”做贝叶斯化（例如 Wu 等, 2007），对 HME 的参数做贝叶斯化（例如 Peng 等, 1996；Bishop 与 Svensén, 2003）在计算上更容易。

## 解释（通俗易懂）

### 1) CART 在做什么？

- 把输入空间一刀一刀地切：每个结点用一个“特征  $\leq$  阈值？”的判断，把数据分为左、右两份；一路切到叶子，相当于把空间切成很多小方块/小长方体（与坐标轴平行）。
- 每个叶子一个简单模型：
  - 回归：用该叶子里  $y$  的平均值当预测；于是得到分段常数的台阶面（图 16.1c）。
  - 分类：存每个类别的比例，预测时选概率最大的类，或直接输出类概率。
- 这种树模型可以写成自适应基函数：每个叶子对应一个指示函数  $I(x \in R_m)$ ，权重就是该叶子的预测值  $w_m$ 。

小提醒：上文式 (16.4) 的第二个等号把指示函数记作  $\phi(x; v_m)$ （“基函数”），而  $v_m$  记录了这片叶子路径上“用哪个特征在什么阈值切”的信息。

### 2) 怎么选“在哪儿切”？

- 贪心：在当前结点，枚举所有特征、所有可能阈值，选使代价（cost）最小的那一个（式 16.5）。
- 候选阈值：把该特征在样本中的唯一取值排好序，逐个尝试就行。类别型特征常用二分法（等于某类 / 不等于某类）。
- 为什么通常不做多路切分？因为每个子分支的数据会更少，容易过拟合。
- 何时停？
  - “增益”太小就不切（式 16.6；常见写法是  $\Delta = \text{cost}(D) - \frac{|D_L|}{|D|} \text{cost}(D_L) - \frac{|D_R|}{|D|} \text{cost}(D_R)$ ，都是减号）。
  - 树太深、子集太小、或子集已很纯（单一类别）。

### 3) 回归与分类的“代价”各是什么？

- 回归（式 16.7）：每个叶子的平方误差和  $\sum(y_i - \bar{y})^2$ 。也可以在叶子里拟合个小的线性回归（比常数更灵活），用残差作为代价。
- 分类：先数数每类的比例  $\hat{\pi}_c$ （式 16.8），然后用以下指标之一来评价切分后的“纯度”：

- **误分类率**:  $1 - \text{多数类比例}$  (式 16.9)。优点直观，但对概率变化不敏感。
- **熵/交叉熵 (对数损失)** :  $-\sum \hat{\pi}_c \log \hat{\pi}_c$  (式 16.10)。等价于**信息增益最大化** (式 16.11–16.13)。
- **基尼指数**:  $1 - \sum \hat{\pi}_c^2$  (式 16.14)。
- **二分类对比** (图 16.3) :
  - 误分类率:  $1 - \max(p, 1 - p)$ ;
  - 熵:  $H_2(p)$ ;
  - 基尼:  $2p(1 - p)$ 。

熵与基尼曲线更弯、更敏感，因而在同样误分类率下，它们往往偏好**产生更纯叶子的切分** (文中举例:  $(200, 400) + (200, 0)$  比  $(300, 100) + (100, 300)$  更好)。

#### 4) 示例与过拟合、剪枝

- 鸢尾花示例只用两个特征，学到的树 (图 16.5a) 和边界 (图 16.4b) 比较**复杂**，训练误差很低，但**交叉验证误差**高 (图 16.5b)，说明过拟合。
- **做法**: 先“尽情长”一棵大树，再**剪枝**。把“剪掉某枝后误差增加最少”的枝优先剪 (细节见 CART 书)。最终用**1-SE 规则**挑模型：在“CV 最小误差 + 1 个标准误差”范围内选**更简单的那棵** (图 16.6)。

#### 5) 树的优缺点

- **优点**: 可解释、处理混合类型、对单调变换不敏感 (排名切分)、自动选变量、较抗离群点、可扩展、能处理缺失值。
- **缺点**: 单棵树常**精度一般**，而且**不稳定 (高方差)**：数据小改动 → 上层切分变 → 整棵树变。

#### 6) 如何提升：随机森林与贝叶斯树

- **Bagging**: 对不同 bootstrap 子样本学多棵树，**取平均** (式 16.15) 以降方差。
- **随机森林**: 在每个结点只从**随机挑选的少数特征**里找最优切分，能去相关、进一步降方差，精度常很好。
- **贝叶斯路线**:
  - 用 MCMC 在“树结构+参数”的空间里做近似推断；或
  - 在“树的集成”上做贝叶斯 (BART)。

二者训练都慢但效果强；代价是**解释性**比单树差 (可做事后可解释化处理)。

#### 7) 与分层专家混合 (HME) 的比较

- **HME** 类似“**概率版的树**”，但用**线性边界**来分区，并且**综合所有专家的输出** (平均)，不是只看一个叶子，因此**更不易过拟合**。

- 训练 HME 是**连续优化 (EM)**，通常比离散贪心的树更**不易卡局部最优**；做贝叶斯化也**更容易**。
- 

## 更常见写法 / 易错点提示

- (16.6) 的“增益”常写作

$$\Delta = \text{cost}(D) - \frac{|D_L|}{|D|} \text{cost}(D_L) - \frac{|D_R|}{|D|} \text{cost}(D_R),$$

即“父结点代价 – 左右子结点加权代价之和”。

- 信息增益的式 (16.12)-(16.13) 常整体写作  
 $\text{infoGain} = H(Y) - H(Y|X_j < t)$ 。
  - 许多地方应为  $x'$  (另一样本) 或需要“ $\Sigma$ ”求和号；这里按原文翻译，理解时请参考上面的常见写法。
- 

## 要点总结

- CART 思路**：反复用“特征  $\leq$  阈值？”切分空间；每个叶子用简单模型（均值或类概率）预测。形式上是**自适应基函数模型**。
- 切分选择**：在当前结点遍历“一切特征  $\times$  一切阈值”，选使**代价**最小的；常用**误分类率、熵 (信息增益)、基尼**等不纯度。熵与基尼**更偏好纯度高的切分**。
- 回归代价**：叶子内平方误差和；也可在叶子里做小线性回归。
- 停止与剪枝**：用增益阈值/最大深度/最小样本数/纯度等做**停止**；或先长满再**剪枝**，用**交叉验证 + 1SE 规则**选更简单稳定的树。
- 优缺点**：易解释、能处理混合特征且鲁棒，但**单棵树高方差、精度一般**、对数据小变动很敏感。
- 提升**：用Bagging/**随机森林**平均多树以降方差、提精度；或走**贝叶斯树/BART**路线（训练慢、精度高、解释性弱）。
- 替代方案**：HME 用线性边界+多专家平均，优化连续、较不易过拟合，是 CART 的有力竞争者。

这一小节主要在讲什么？我不太懂

已思考 16s

没问题！这一小节其实就是在用“决策树”把**怎么做非线性预测**讲清楚：树怎么长、怎么评估好坏、为什么会过拟合、怎么剪、以及怎样用“很多棵树”变强。给你一个**超简版→稍详细版**的说明。

---

## 一句话版

用一连串“是/否”的小问题（特征是否小于某阈值？）把空间切成许多块；每一块给一个简单预测（回归用均值，分类用类别比例）。树这样长出来可能会过拟合，所以要剪枝；单棵树不稳就用**随机森林**把很多树平均；还顺带和**分层专家混合**（HME）比了比。

---

## 稍详细版（通俗直白）

### 1. 树在干嘛

像玩“20个问题”：每个节点问一个是/否问题（例如“ $x_1 \leq t$  吗？”），一路走到叶子。

- **回归**：落到某叶子，就用该叶子里  $y$  的**平均值**当预测 → 台阶面。
- **分类**：叶子里数各类比例，输出**概率/多数类**。

### 2. 树怎么长出来（生长/分裂）

在当前节点，枚举“所有特征  $\times$  各个阈值”，挑那个**让误差（代价）最小**的切分（贪心）。

- **回归的代价**：叶子内**平方误差和**。
- **分类的代价/不纯度**：
  - 误分类率（直观但不敏感），
  - 熵/信息增益，
  - 基尼指数。

熵和基尼更偏好把叶子弄“更纯”，通常分得更好。

### 3. 何时停？

若增益太小、树太深、子集太小或已经很纯，就**停止分裂**。否则会越来越碎，容易过拟合。

### 4. 过拟合与剪枝

常见做法：**先长满再剪**。把“剪掉后错误增加最少”的枝先剪，再用**交叉验证+1个标准误差规则**选更简单那棵（更稳）。

### 5. 优缺点

- 优点：**可解释、能混合离散/连续、对单调变换不敏感、会自动选特征、相对抗离群、可扩展、能处理缺失。
- 缺点：**单棵树精度一般且不稳定（高方差）：数据小改动，结构可能大变。

## 6. 怎么变强：随机森林

训练很多棵树（对数据、特征都做随机化），把预测**平均**（Bagging/随机森林）→ 大幅**降方差、提精度**。代价是**可解释性下降**。还有贝叶斯版本（BART等），效果好但训练更慢。

## 7. 和 HME 的对比

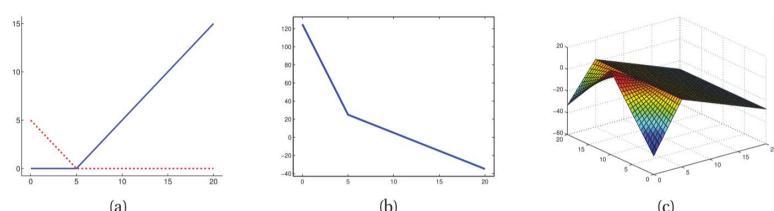
HME 像“概率版决策树”：用**线性边界**划分、对多个专家**加权平均**预测，优化是**连续的（EM）**，常比贪心的树更不易卡局部最优，也更易做贝叶斯化。

## 8. 图里在说啥

- 图16.1：二维例子，轴向切分→5个区域→分段常数的台阶面。
- 图16.2：玩具分类树，叶子标（正样本数，负样本数）。
- 图16.3：误分类率、熵、基尼三条曲线对比（熵/基尼更敏感）。
- 图16.4–16.6：鸢尾花示例：未剪枝边界很复杂、CV误差高；剪枝后更简单、更稳。

## 记住这几句就行

- 决策树 = 一连串阈值判断 + 叶子里的简单预测。**
- 分裂标准：**回归看平方误差；分类看熵/信息增益或基尼（比误分类率更好用）。
- 过拟合？先长后剪，用交叉验证挑更简单的树。**
- 单树高方差 → 用随机森林把很多树平均**（强但不太可解释）。
- HME 是可替代方案：**线性划分 + 专家平均，优化平滑，常更稳。



翻译并解释以下内容，翻译不能有遗漏，解释请通俗易懂且详尽，因为我的数学基础不太好，最后总结要点。

Figure 16.7 (a) Linear spline function with a knot at 5.  
Solid blue:  $(x-5)_+$ . Dotted red:  $(5-x)_+$ . (b) A MARS model

in 1d given by Equation 16.23. (c) A simple MARS model in 2d given by Equation 16.24.

### 16.3 Generalized additive models

A simple way to create a nonlinear model with multiple inputs is to use a generalized additive model (Hastie and Tibshirani 1990), which is a model of the form  $f(x) = \alpha + f_1(x_1) + \dots + f_D(x_D)$  (16.16)

Here each  $f_j$  can be modeled by some scatterplot smoother, and  $f(x)$  can be mapped to  $p(y|x)$  using a link function, as in a GLM (hence the term generalized additive model). If we use regression splines (or some other fixed basis function expansion approach) for the  $f_j$ , then each  $f_j(x_j)$  can be written as  $\beta^T \varphi_j(x_j)$ , so the whole model can be written as  $f(x) = \beta^T \varphi(x)$ , where  $\varphi(x) = [1, \varphi_1(x_1), \dots, \varphi_D(x_D)]$ . However, it is more common to use smoothing splines (Section 15.4.6) for the  $f_j$ . In this case, the objective (in the regression setting) becomes

$$J(\alpha, f_1, \dots, f_D) = \sum_{i=1}^N \|y_i - \alpha - \sum_{j=1}^D f_j(x_{ij})\|^2 + \sum_{j=1}^D \lambda_j \|f_j(t_j)\|_2^2 \quad (16.17)$$

where  $\lambda_j$  is the strength of the regularizer for  $f_j$ .

#### 16.3.1 Backfitting

We now discuss how to fit the model using MLE. The constant  $\alpha$  is not uniquely identifiable, since we can always add or subtract constants to any of the  $f_j$  functions. The convention is to assume  $\sum_{i=1}^N f_j(x_{ij}) = 0$  for all  $j$ . In this case, the MLE for  $\alpha$  is just  $\hat{\alpha} = \frac{1}{N} \sum_{i=1}^N y_i$ . To fit the rest of the model, we can center the responses (by subtracting  $\hat{\alpha}$ ), and then iteratively update each  $f_j$  in turn, using as a target vector the residuals obtained by omitting term  $f_j$ :

$$\hat{f}_j := \text{smoother}(\{y_i - \sum_{k \neq j} f_k(x_{ik})\}_{i=1}^N) \quad (16.18)$$

We should then ensure the output is zero mean using

$$\hat{f}_j := \hat{f}_j - \frac{1}{N} \sum_{i=1}^N \hat{f}_j(x_{ij}) \quad (16.19)$$

This is called the backfitting algorithm (Hastie and Tibshirani 1990). If  $X$  has full column rank, then the above objective is convex (since each smoothing spline is a linear operator, as shown in Section 15.4.2), so this procedure is guaranteed to converge to the global optimum. In the GLM case, we need to modify the method somewhat. The basic idea is to replace the weighted least squares step of IRLS (see Section 8.3.4) with a weighted backfitting algorithm. In the logistic regression case, each response has weight  $s_i = \mu_i(1 - \mu_i)$  associated with it, where  $\mu_i$

```
=sigm(^α + D j=1 f^ j(xij).)
```

### 16.3.2 Computational efficiency

Each call to the smoother takes  $O(N)$  time, so the total cost is  $O(NDT)$ , where  $T$  is the number of iterations. If we have high-dimensional inputs, fitting a GAM is expensive. One approach is to combine it with a sparsity penalty, see e.g., the SpAM (sparse additive model) approach of (Ravikumar et al. 2009). Alternatively, we can use a greedy approach, such as boosting (see Section 16.4.6)

### 16.3.3 Multivariate adaptive regression splines (MARS)

We can extend GAMs by allowing for interaction effects.

In general, we can create an ANOVA decomposition:

$$f(x) = \beta_0 + \sum_{j=1}^D f_j(x_j) + \sum_{j,k} f_{jk}(x_j, x_k) + \sum_{j,k,l} f_{jkl}(x_j, x_k, x_l) + \dots \quad (16.20)$$

Of course, we cannot allow for too many higher-order interactions, because there will be too many parameters to fit. It is common to use greedy search to decide which variables to add. The multivariate adaptive regression splines or MARS algorithm is one example of this (Hastie et al. 2009, Sec9.4). It fits models of the form in Equation 16.20, where it uses a tensor product basis of regression splines to represent the multidimensional regression functions. For example, for 2d input, we might use

$$f(x_1, x_2) = \beta_0 + m \beta_1 m(x_1 - t_1 m) + \dots + m \beta_2 m(t_2 m - x_2) + \dots + m \beta_{12} m(x_1 - t_1 m) + (t_2 m - x_2) + \dots \quad (16.21)$$

To create such a function, we start with a set of candidate basis functions of the form

$$C = \{(x_j - t)_+, (t - x_j)_+ : t \in \{x_1, \dots, x_N\}, j = 1, \dots, D\} \quad (16.22)$$

These are 1d linear splines where the knots are at all the observed values for that variable. We consider splines sloping up in both directions; this is called a reflecting pair. See Figure 16.7(a). Let  $M$  represent the current set of basis functions. We initialize by using  $M = \{1\}$ . We consider creating a new basis function pair by multiplying an  $h_m \in M$  with one of the reflecting pairs in  $C$ . For example, we might initially get

$$f(x) = 25 - 4(x_1 - 5)_+ + 20(5 - x_1)_+ \quad (16.23)$$

obtained by multiplying  $h_0(x) = 1$  with a reflecting pair involving  $x_1$  with knot  $t = 5$ . This pair is added to  $M$ . See Figure 16.7(b). At the next step, we might create a model such as

$$f(x) = 2 - 2(x_1 - 5)_+ + 3(5 - x_1)_+ - (x_2 - 10)_+ \times (5 - x_1)_+ - 1.2(10 - x_2)_+ \times (5 - x_1)_+ \quad (16.24)$$

obtained by multiplying  $(5-x_1)_+$  from M by the new reflecting pair  $(x_2-10)_+$  and  $(10-x_2)_+$ . This new function is shown in Figure 16.7(c).

We proceed in this way until the model becomes very large. (We may impose an upper bound on the order of interactions.) Then we prune backwards, at each step eliminating the basis function that causes the smallest increase in the residual error, until the CV error stops improving.

The whole procedure is closely related to CART. To see this, suppose we replace the piecewise linear basis functions by step functions  $I(x_j > t)$  and  $I(x_j < t)$ . Multiplying by a pair of reflected step functions is equivalent to splitting a node. Now suppose we impose the constraint that once a variable is involved in a multiplication by a candidate term, that variable gets replaced by the interaction, so the original variable is no longer available. This ensures that a variable can not be split more than once, thus guaranteeing that the resulting model can be represented as a tree. In this case, the MARS growing strategy is the same as the CART growing strategy.

已思考 1m 24s

下面分三部分：逐字不遗漏的中文翻译 → 通俗解释 → 要点总结。

(注：原文个别公式/符号排版略有小问题，例如平滑样条的惩罚项通常是二阶导的平方积分，式(16.24)的常数项前有“==”；我会在解释里给出“更常见写法”。)

## 翻译

Figure 16.7 (a) 在 5 处有结点 (knot) 的线性样条函数。实线蓝色： $(x - 5)_+$ 。红色虚线： $(5 - x)_+$ 。

- (b) 由式(16.23)给出的 1 维 MARS 模型。
- (c) 由式(16.24)给出的一个 2 维的简单 MARS 模型。

## 16.3 广义可加模型

用多输入创建非线性模型的一种简单方法是使用**广义可加模型** (Hastie and Tibshirani 1990)，其形式为

$$f(x) = \alpha + f_1(x_1) + \cdots + f_D(x_D) \quad (16.16)$$

这里每个  $f_j$  都可以由某种散点图平滑器建模，并且  $f(x)$  可以像 GLM 一样通过一个**联结函数**映射到  $p(y|x)$ （因此称为“广义可加”）。

如果对  $f_j$  使用**回归样条**（或其他固定基函数展开），则  $f_j(x_j) = \beta_j^\top \phi_j(x_j)$ ，于是整个模型可写为

$f(x) = \beta^\top \phi(x)$ ，其中  $\phi(x) = [1, \phi_1(x_1), \dots, \phi_D(x_D)]$ 。不过，更常见的是对  $f_j$  使用**平滑样条**（第15.4.6节）。在这种情况下，（回归设定下的）目标函数为

$$J(\alpha, f_1, \dots, f_D) = \sum_{i=1}^N (y_i - \alpha - \sum_{j=1}^D f_j(x_{ij}))^2 + \sum_{j=1}^D \lambda_j \int f_j(t_j)^2 dt_j \quad (16.17)$$

其中  $\lambda_j$  是对  $f_j$  的正则化强度。

### 16.3.1 反向拟合 (Backfitting)

我们现在讨论如何用 MLE 拟合模型。常数  $\alpha$  不是唯一可辨的，因为总可以在任意一个  $f_j$  上加减常数。习惯做法是假设

$\sum_{i=1}^N f_j(x_{ij}) = 0$ （对所有  $j$ ）。在此情况下， $\alpha$  的 MLE 为  
 $\hat{\alpha} = \frac{1}{N} \sum_{i=1}^N y_i$ 。

为了拟合其余部分，我们可以**中心化响应**（减去  $\hat{\alpha}$ ），然后**迭代地**依次更新每个  $f_j$ ，用“**去掉  $f_j$  之后的残差**”作为目标向量：

$$\hat{f}_j := \text{smoother}(\{y_i - \sum_{k \neq j} \hat{f}_k(x_{ik})\}_{i=1}^N) \quad (16.18)$$

随后保证输出**均值为 0**：

$$\hat{f}_j := \hat{f}_j - \frac{1}{N} \sum_{i=1}^N \hat{f}_j(x_{ij}) \quad (16.19)$$

这称为**反向拟合算法** (Hastie and Tibshirani 1990)。若  $X$  满列秩，则以上目标是**凸的**（因为每个平滑样条是线性算子，见15.4.2节），因此该过程保证收敛到**全局最优**。

在 GLM 情形，需要做些修改：把 IRLS（见8.3.4节）中的**加权最小二乘**步骤替换为**加权反向拟合**。在逻辑回归里，每个响应带有权重

$$s_i = \mu_i(1 - \mu_i)，其中 \mu_i = \text{sigm}(\hat{\alpha} + \sum_{j=1}^D \hat{f}_j(x_{ij}))。$$

### 16.3.2 计算效率

每次调用平滑器需要  $O(N)$  时间，所以总成本为  $O(NDT)$ ，其中  $T$  为迭代次数。若输入维度高，拟合 GAM 代价昂贵。一种做法是结合**稀疏惩罚**，例如 (Ravikumar et al., 2009) 的 SpAM（稀疏可加模型）。或者使用**贪心方法**，如**提升 (boosting)**（见16.4.6节）。

### 16.3.3 多元自适应回归样条 (MARS)

我们可通过允许**交互效应**来扩展 GAM。一般地，可以做一个**ANOVA 分解**：

$$f(x) = \beta_0 + \sum_j f_j(x_j) + \sum_{j,k} f_{jk}(x_j, x_k) + \sum_{j,k,l} f_{jkl}(x_j, x_k, x_l) + \dots \quad (16.20)$$

当然，不能允许太多高阶交互，否则参数太多。通常用**贪心搜索**决定加入哪些变量。

**MARS 算法**就是一个例子 (Hastie 等, 2009, 第9.4节)。它拟合形如(16.20)的模型，用**回归样条的张量积基**来表示多维回归函数。比如在 2 维输入时，可以用

$$f(x_1, x_2) = \beta_0 + \sum_m \beta_{1m} (x_1 - t_{1m})_+ + \sum_m \beta_{2m} (t_{2m} - x_2)_+ + \sum_m \beta_{12m} (x_1 - t_{1m})_+ (t_{2m} - x_2)_+ \quad (16.21)$$

为构造这样的函数，先从如下**候选基函数**集合开始：

$$C = \{(x_j - t)_+, (t - x_j)_+ : t \in \{x_{1j}, \dots, x_{Nj}\}, j = 1, \dots, D\} \quad (16.22)$$

这些是 1 维**线性样条**，结点放在该变量所有观测值处。我们在两个方向上都考虑上升的样条；这称为一个**对称（反射）对**。见图16.7(a)。令 M 表示当前的基函数集合，初始化为 M = {1}。我们考虑通过把 M 中某个  $h_m$  与 C 中一对反射基函数相乘来创建新的基函数对。

例如，最初可能得到

$$f(x) = 25 - 4(x_1 - 5)_+ + 20(5 - x_1)_+ \quad (16.23)$$

这是把  $h_0(x) = 1$  与在结点  $t = 5$  处、涉及  $x_1$  的反射对相乘得到的。把这对加入 M；见图16.7(b)。

下一步，可能得到如下模型：

$$f(x) = 2 - 2(x_1 - 5)_+ + 3(5 - x_1)_+ - (x_2 - 10)_+ (5 - x_1)_+ - 1.2(10 - x_2)_+ \quad (16.24)$$

这是把 M 中的  $(5 - x_1)_+$  与新的反射对  $(x_2 - 10)_+, (10 - x_2)_+$  相乘得到的。该函数见图 16.7(c)。

我们以此方式前进直到模型变得很大（可以对交互阶数设上限）。然后**反向剪枝**：每一步删去使**残差误差**增加最小的基函数，直到**交叉验证 (CV) 误差**不再改进。

整个过程与 CART 密切相关。为看清这一点，设我们把分段线性基函数替换为**阶跃函数**  $I(x_j > t)$  与  $I(x_j < t)$ 。用一对反射的阶跃函数相乘等价于**切分一个结点**。再设一个约束：一旦某个变量参与了与某个候选项的乘积，该变量就由这个**交互项**取代，原始变量不再可用。这样就保证同一变量不会被切分多次，从而保证得到的模型可以表示成一棵树。在这种情况下，MARS 的**生长策略**与 CART 的**生长策略**相同。

## 解释（通俗易懂）

## 1) GAM 是什么? 直觉版

- 把复杂函数拆成各个特征的单变量函数的和：  

$$f(x) = \alpha + \sum_j f_j(x_j)$$

每个  $f_j$  单独负责“这一维怎么弯”。这样既非线性（因为每个  $f_j$  可以是弯曲的），又好训练（每次只调一维）。
- 若做分类/概率建模，再用 GLM 的联结函数把  $f(x)$  映到  $p(y|x)$ 。

常见实现：让  $f_j$  用平滑样条表示，并加平滑惩罚（通常是  $\int (f_j''(t))^2 dt$ ），让曲线既贴合又不过分抖动。

## 2) 怎样训练：反向拟合 (Backfitting)

- 先把常数项定好： $\hat{\alpha}$  = 数据均值，并把  $y$  减去它。
- 然后轮流更新每个  $f_j$ ：把“其他维度的贡献”先减掉，只对剩下的残差做一次一维平滑；再把该  $f_j$  调整成零均值（防止和  $\alpha$  混在一起）。
- 这样一个维度一个维度“往回补”，循环几次就收敛。在线性情形该目标是凸的 → 能到全局最优。
- 做逻辑回归等 GLM 时，套上权重 (IRLS 的权重  $s_i = \mu_i(1 - \mu_i)$ )，即加权反向拟合。

## 3) 算力与稀疏

- 每次平滑  $O(N)$ ，总复杂度约  $O(NDT)$ 。维度  $D$  大就贵。
- 解决：加稀疏惩罚（如 SpAM，让不少  $f_j$  变成 0），或用提升/贪心只挑最有用的维度逐步加入。

## 4) MARS：把“可加”升级到“带交互”

- 可加模型只加各维度函数；MARS 允许交互项： $f_{jk}(x_j, x_k)、f_{jkl}(\cdot) \dots$ （但交互阶数别太高）。
- 基函数不再只是“单维样条”，而是所谓的 hinge 函数（铰链函数）： $(x - t)_+ = \max(x - t, 0)、(t - x)_+$ 。一对称为反射对。
- 前向阶段：从常数 1 开始，反复把已有基函数与新反射对相乘来生成新基函数（相乘 = 引入交互/把适用范围切成更多“片”）。
  - 例(16.23)：只用  $x_1$  在结点 5 的两个铰链，就能画出图16.7(b) 的折线。
  - 例(16.24)：再把  $(5 - x_1)_+$  与  $x_2$  在 10 的反射对相乘，得到两个二元交互项 → 图 16.7(c) 的折面。
- 后向剪枝：模型长大后，逐个删“影响最小”的基函数，直到 CV 误差最好为止。
- 与 CART 的关系：若把“分段线性铰链”换成“阶跃指示函数”，并限制每个变量只分一次，那么“前向生长+相乘”的过程就和决策树的切分一模一样；因此 MARS 可视作

CART 的一个“分段线性”版本。

更常见写法提示：式(16.17)的积分惩罚在平滑样条里通常是  $\int [f_j''(t)]^2 dt$ 。式(16.24)的开头“ $= 2$ ”应为常数项，常见书写为  $f(x) = 2 + \dots$  或  $-2 + \dots$ （不影响理解：它只是一个截距）。

## 要点总结

- GAM：把复杂函数拆成各维的单变量函数之和；用反向拟合轮流学每一维；可配 GLM 联结做分类；高维下可用稀疏或提升控复杂度。
- 反向拟合：定  $\alpha$ ；对每个  $f_j$  用“别的维度减掉后的残差”做（加权）平滑，再去均值；线性情形收敛到全局最优。
- MARS：在 GAM 的基础上引入交互，用铰链函数  $(x - t)_+$  作为基，前向生成×后向剪枝，最后用 CV 选规模；可视作分段线性版的 CART。
- 直观记忆：GAM=按维相加；MARS=按维相加+必要的相乘（交互）；CART=按阈值分块。

这一小节主要在讲什么？我不太懂

已思考 9s

当然！这小节（16.3：广义可加模型与 MARS）核心在讲两件事：**怎么用“按维相加”的方式做非线性建模（GAM）**，以及**在此基础上怎样加入“变量之间的交互”并得到分段线性模型（MARS）**；同时告诉你它们怎么训练、算力大致多少、以及和决策树（CART）的关系。

## 电梯版（一句话概括）

- GAM：把复杂函数拆成“每个特征各画一条弯曲小曲线，再把这些曲线加起来”。
- Backfitting：训练 GAM 的办法——轮流给每个特征的那条曲线做平滑，始终用“当前残差”当目标。
- MARS：在 GAM 上加交互，用“铰链函数  $(x - t)_+$ ”拼出分段线性的折线/折面；**前向生长 + 后向剪枝**；思想上和 CART 很像。

## 通俗版（稍详细但不费劲）

### 1) GAM：按维相加的非线性模型

- 形式:  $f(x) = \alpha + f_1(x_1) + \dots + f_D(x_D)$ 。
- 含义: 每个特征  $x_j$  都有一条“一维曲线” $f_j$  描述它对预测的贡献，最后把这些贡献相加。
- 好处: 既能弯（非线性），又清楚“哪一维怎么影响结果”（可解释）。

### 怎么学 (Backfitting)

- 先把常数  $\alpha$  定成  $y$  的均值。
- 反复“轮流训练每一维”——更新  $f_j$  时:
  - 先用“其他维度的当前预测”把  $y$  减掉，得到残差；
  - 用一维平滑器（如平滑样条）去拟合这个残差；
  - 再把这条曲线减去自己的均值（防止和  $\alpha$  重叠）。
- 循环到收敛。做分类时（逻辑回归等）只是加上样本权重而已。

**代价感知**: 每次平滑是  $O(N)$ ，总约  $O(NDT)$  ( $D$  维、迭代  $T$  次)。维度大时可用稀疏可加模型 (SpAM) 或提升/贪心来挑少数重要维度。

---

### 2) MARS：把“相加”升级到“还能相乘”（交互）

- 目的: GAM 只会“相加”，抓不住“两个变量一起作用”的情况；MARS 允许交互项。
- 秘诀: 用**铰链函数**  $(x - t)_+ = \max(x - t, 0)$  (以及它的“镜像” $(t - x)_+$ ) 作为基函数。
- 前向生长**: 从常数 1 开始，反复把已有基函数  $\times$  一个新的铰链函数 (或它们的对) 来生成新项；乘起来就相当于“只在某一区域起作用”，也就引入了**交互/局部线性**。
- 后向剪枝**: 长大后逐个删掉对误差影响最小的基函数，用**交叉验证**挑到“最简单且误差还低”的模型。

### 和图 16.7 的关系

- (a) 展示两个方向的铰链函数:  $(x - 5)_+$  与  $(5 - x)_+$ 。
- (b) 一维 MARS: 用上面两条折线拼出一条分段折线。
- (c) 二维 MARS: 再把  $(5 - x_1)_+$  和关于  $x_2$  的铰链相乘，得到**折面**（交互）。

### 和 CART 很像

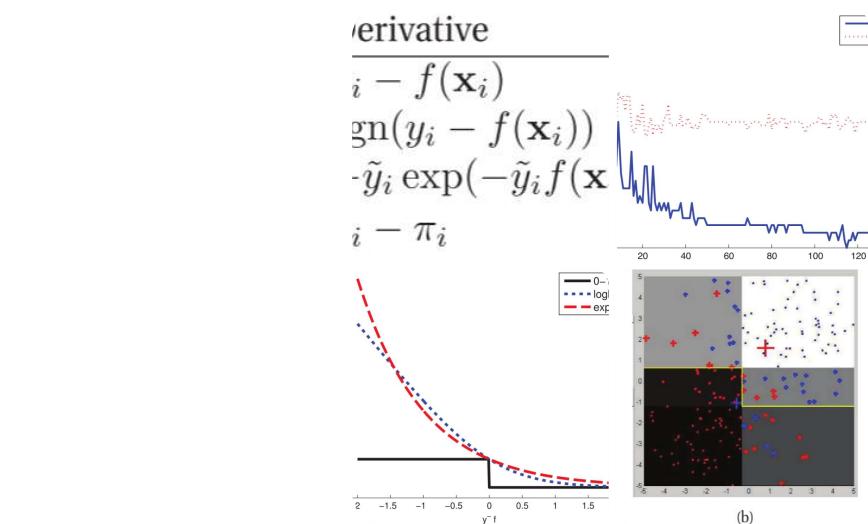
- 如果把“分段线性铰链”换成“阶跃指示函数  $I(x_j < t)$ ”并限制每个变量只分一次，那么“前向生长 + 剪枝”的流程等价于长一棵决策树。
- 直观地说: **CART 是分段常数；MARS 是分段线性**，更平滑一些。

## 这节在解决什么问题？

- 需要**非线性**又要**可解释**: GAM/MARS 都做得到;
- 想看**单个特征的影响曲线**: 用 GAM;
- 还想**捕捉变量之间的配合**(交互): 用 MARS;
- 希望有**类似树的自动建模流程**但更平滑: MARS 就是“**分段线性版的树**”。

## 记住这几句就够了

- GAM = 各维一条曲线，相加得结果；用 Backfitting 轮流拟合残差。
- MARS = 在 GAM 上加交互，用铰链函数拼成分段线性；前向生长，后向剪枝，和 CART 思想近。
- 图 16.7 只是把铰链函数及其在 1D/2D 下拼出的折线/折面画出来而已。



翻译并解释以下内容，翻译不能有遗漏，解释请通俗易懂且详尽，因为我的数学基础不太好，最后总结要点。

Table 16.1 Some commonly used loss functions, their gradients, their population minimizers  $f_*$ , and some algorithms to minimize the loss. For binary classification problems, we assume  $\tilde{y}_i \in \{-1, +1\}$ ,  $y_i \in \{0, 1\}$  and  $\pi_i = \text{sigm}(2f(\mathbf{x}_i))$ . For regression problems, we assume  $y_i \in \mathbb{R}$ . Figure 16.8 Performance of adaboost using a decision stump as a weak learner on the data in Figure 16.10. Training (solid blue) and test (dotted red) error vs number

of iterations.

Figure 16.9 Illustration of various loss functions for binary classification. The horizontal axis is the margin  $y\eta$ , the vertical axis is the loss. The log loss uses log base 2.

Figure 16.10 Example of adaboost using a decision stump as a weak learner. The degree of blackness represents the confidence in the red class. The degree of whiteness represents the confidence in the blue class. The size of the datapoints represents their weight. Decision boundary is in yellow. (a) After 1 round. (b) After 3 rounds. (c) After 120 rounds.

## 16.4 Boosting

Boosting (Schapire and Freund 2012) is a greedy algorithm for fitting adaptive basis-function models of the form in Equation 16.3, where the  $\varphi_m$  are generated by an algorithm called a weak learner or a base learner. The algorithm works by applying the weak learner sequentially to weighted versions of the data, where more weight is given to examples that were misclassified by earlier rounds. This weak learner can be any classification or regression algorithm, but it is common to use a CART model. In 1998, the late Leo Breiman called boosting, where the weak learner is a shallow decision tree, the "best off-the-shelf classifier in the world" (Hastie et al. 2009, p340). This is supported by an extensive empirical comparison of 10 different classifiers in (Caruana and Niculescu-Mizil 2006), who showed that boosted decision trees were the best both in terms of misclassification error and in terms of producing well-calibrated probabilities, as judged by ROC curves. (The second best method was random forests, invented by Breiman; see Section 16.2.5.) By contrast, single decision trees performed very poorly. Boosting was originally derived in the computational learning theory literature (Schapire 1990; Freund and Schapire 1996), where the focus is binary classification. In these papers, it was proved that one could boost the performance (on the training set) of any weak learner arbitrarily high, provided the weak learner could always perform slightly better than chance. For example, in Figure 16.8, we plot the training and test error for boosted decision stumps on a 2d dataset shown in Figure 16.10. We see that the training set error rapidly goes to near

zero. What is more surprising is that the test set error continues to decline even after the training set error has reached zero (although the test set error will eventually go up). Thus boosting is very resistant to overfitting.

(Boosted decision stumps form the basis of a very successful face detector (Viola and Jones 2001), which was used to generate the results in Figure 1.6, and which is used in many digital cameras.) In view of its stunning empirical success, statisticians started to become interested in this method. Breiman (Breiman 1998) showed that boosting can be interpreted as a form of gradient descent in function space. This view was then extended in (Friedman et al. 2000), who showed how boosting could be extended to handle a variety of loss functions, including for regression, robust regression, Poisson regression, etc. In this section, we shall present this statistical interpretation of boosting, drawing on the reviews in (Buhlmann and Hothorn 2007) and (Hastie et al. 2009, ch10), which should be consulted for further details.

#### 16.4.1 Forward stagewise additive modeling

The goal of boosting is to solve the following optimization problem:

$$\min f \sum_{i=1}^n L(y_i, f(x_i)) \quad (16.25)$$

and  $L(y, \hat{y})$  is some loss function, and  $f$  is assumed to be an ABM model as in Equation 16.3. Common choices for the loss function are listed in Table 16.1. If we use squared error loss, the optimal estimate is given by

$$f^*(x) = \operatorname{argmin} f(x) = E[y|x] \quad (16.26)$$

as we showed in Section 5.7.1.3. Of course, this cannot be computed in practice since it requires knowing the true conditional distribution  $p(y|x)$ . Hence this is sometimes called the population minimizer, where the expectation is interpreted in a frequentist sense. Below we will see that boosting will try to approximate this conditional expectation. For binary classification, the obvious loss is 0-1 loss, but this is not differentiable. Instead it is common to use logloss, which is a convex upper bound on 0-1 loss, as we showed in Section 6.5.5. In this case, one can show that the optimal estimate is given by

$$f^*(x) = -\frac{1}{2} \log p(\tilde{y} = 1|x) / p(\tilde{y} = -1|x) \quad (16.27)$$

where  $\tilde{y} \in \{-1, +1\}$ . One can generalize this framework to the multiclass case, but we will not discuss that here. An alternative convex upper bound is exponential loss,

defined by

$$L(\tilde{y}, f) = \exp(-\tilde{y}f) \quad (16.28)$$

See Figure 16.9 for a plot. This will have some computational advantages over the logloss, to be discussed below. It turns out that the optimal estimate for this loss is also  $f^*(x) = \frac{1}{2} \log p(\tilde{y}=1|x) - \log p(\tilde{y}=-1|x)$ . To see this, we can just set the derivative of the expected loss (for each  $x$ ) to zero:

$$\partial \partial f(x) E e^{-\tilde{y}f(x)}|x = \partial \partial f(x) [p(\tilde{y}=1|x)e^{-f(x)} + p(\tilde{y}=-1|x)e^{f(x)}] \quad (16.29)$$

$$= -p(\tilde{y}=1|x)e^{-f(x)} + p(\tilde{y}=-1|x)e^{f(x)} \quad (16.30)$$

$$= 0 \Rightarrow p(\tilde{y}=1|x)p(\tilde{y}=-1|x) = e^{2f(x)} \quad (16.31)$$

So in both cases, we can see that boosting should try to approximate (half) the log-odds ratio. Since finding the optimal  $f$  is hard, we shall tackle it sequentially. We initialise by defining

$$f_0(x) = \operatorname{argmin}_\gamma \sum_{i=1}^N L(y_i, f(x_i; \gamma)) \quad (16.32)$$

For example, if we use squared error, we can set  $f_0(x) = y$ ,

and if we use log-loss or exponential loss, we can set

$$f_0(x) = \frac{1}{2} \log \hat{\pi}^{\tilde{y}} (1 - \hat{\pi})^{\tilde{y}}, \text{ where } \hat{\pi} = \frac{1}{N} \sum_{i=1}^N I(y_i = 1). \quad (16.32)$$

We could also use a more powerful model for our baseline, such as a GLM. Then at iteration  $m$ , we compute

$$(\beta_m, \gamma_m) = \operatorname{argmin}_{\beta, \gamma} \sum_{i=1}^N L(y_i, f_{m-1}(x_i) + \beta \varphi(x_i; \gamma)) \quad (16.33)$$

and then we set

$$f_m(x) = f_{m-1}(x) + \beta_m \varphi(x; \gamma_m) \quad (16.34)$$

The key point is that we do not go back and adjust earlier parameters. This is why the method is called forward stagewise additive modeling. We continue this for a fixed number of iterations  $M$ . In fact  $M$  is the main tuning parameter of the method. Often we pick it by monitoring the performance on a separate validation set, and then stopping once performance starts to decrease; this is called early stopping. Alternatively, we can use model selection criteria such as AIC or BIC (see e.g., (Buhlmann and Hothorn 2007) for details). In practice, better (test set) performance can be obtained by performing “partial updates” of the form

$$f_m(x) = f_{m-1}(x) + v \beta_m \varphi(x; \gamma_m) \quad (16.35)$$

Here  $0 < v \leq 1$  is a step-size parameter. In practice it is common to use a small value such as  $v = 0.1$ . This is called shrinkage. Below we discuss how to solve the subproblem in Equation 16.33. This will depend on the form of loss

function. However, it is independent of the form of weak learner.

#### 16.4.2 L2boosting

Suppose we used squared error loss. Then at step m the loss has the form

$$L(y_i, f_{m-1}(x_i) + \beta\varphi(x_i; \gamma)) = (r_m - \varphi(x_i; \gamma))^2 \quad (16.36)$$

where  $r_m = y_i - f_{m-1}(x_i)$  is the current residual, and we have set  $\beta = 1$  without loss of generality. Hence we can find the new basis function by using the weak learner to predict  $r_m$ . This is called L2boosting, or least squares boosting (Buhlmann and Yu 2003). In Section 16.4.6, we will see that this method, with a suitable choice of weak learner, can be made to give the same results as LARS, which can be used to perform variable selection (see Section 13.4.2).

#### 16.4.3 AdaBoost

Consider a binary classification problem with exponential loss. At step m we have to minimize

$$L_m(\varphi) = \sum_{i=1}^N \exp[-\tilde{y}_i(f_{m-1}(x_i) + \beta\varphi(x_i))] = \sum_{i=1}^N w_{i,m} \exp(-\beta\tilde{y}_i\varphi(x_i)) \quad (16.37)$$

where  $w_{i,m} = \exp(-\tilde{y}_i f_{m-1}(x_i))$  is a weight applied to data case i, and  $\tilde{y}_i \in \{-1, +1\}$ . We can rewrite this objective as follows:

$$L_m = e^{-\beta \tilde{y}_i \varphi(x_i)} w_{i,m} + e^{\beta \tilde{y}_i \varphi(x_i)} w_{i,m} \quad (16.38)$$

$$= (e^\beta - e^{-\beta}) \sum_{i=1}^N w_{i,m} (\tilde{y}_i = \varphi(x_i)) + e^{-\beta} \sum_{i=1}^N w_{i,m} \quad (16.39)$$

Consequently the optimal function to add is

$$\varphi_m = \operatorname{argmin}_{\varphi} \sum_{i=1}^N w_{i,m} (\tilde{y}_i = \varphi(x_i)) \quad (16.40)$$

This can be found by applying the weak learner to a weighted version of the dataset, with weights  $w_{i,m}$ .

Substituting  $\varphi_m$  into  $L_m$  and solving for  $\beta$  we find

$$\beta_m = 1/2 \log (1 - \text{err}_m) / \text{err}_m \quad (16.41)$$

where

$$\text{err}_m = \sum_{i=1}^N w_{i,m} (\tilde{y}_i \neq \varphi_m(x_i)) / \sum_{i=1}^N w_{i,m} \quad (16.42)$$

The overall update becomes

$$f_m(x) = f_{m-1}(x) + \beta_m \varphi_m(x) \quad (16.43)$$

With this, the weights at the next iteration become

$$w_{i,m+1} = w_{i,m} e^{-\beta_m \tilde{y}_i \varphi_m(x_i)} \quad (16.44)$$

$$= w_{i,m} e^{\beta_m (2I(\tilde{y}_i = \varphi_m(x_i)) - 1)} \quad (16.45)$$

$$= w_{i,m} e^{2\beta_m I(\tilde{y}_i = \varphi_m(x_i))} e^{-\beta_m} \quad (16.46)$$

where we exploited the fact that  $\tilde{y}_i \varphi_m(x_i) = -1$  if  $\tilde{y}_i = \varphi_m(x_i)$  and  $\tilde{y}_i \varphi_m(x_i) = +1$  otherwise. Since  $e^{-\beta_m}$  will cancel out in the normalization step, we can drop it. The

result is the algorithm shown in Algorithm 7, known Adaboost.M1. An example of this algorithm in action, using decision stumps as the weak learner, is given in Figure 16.10. We see that after many iterations, we can “carve out” a complex decision boundary. What is rather surprising is that AdaBoost is very slow to overfit, as is apparent in Figure 16.8. See Section 16.4.8 for a discussion of this point.

#### 16.4.4 LogitBoost

The trouble with exponential loss is that it puts a lot of weight on misclassified examples, as is apparent from the exponential blowup on the left hand side of Figure 16.9. This makes the method very sensitive to outliers (mislabeled examples). In addition,  $e^{-y f}$  is not the logarithm of any pmf for binary variables  $y \in \{-1, +1\}$ ; consequently we cannot recover probability estimates from  $f(x)$ .

A natural alternative is to use logloss instead. This only punishes mistakes linearly, as is clear from Figure 16.9. Furthermore, it means that we will be able to extract probabilities from the final learned function, using  $p(y=1|x) = e^{f(x)} / (e^{f(x)} + e^{-f(x)}) = 1 / (1 + e^{-2f(x)})$  (16.47)

The goal is to minimize the expected log-loss, given by  $L_m(\phi) = -\frac{1}{N} \sum_{i=1}^N \log [1 + \exp(-2y_i(f_m - \phi(x_i)))]$  (16.48) By performing a Newton update on this objective (similar to IRLS), one can derive the algorithm shown in Algorithm 8. This is known as logitBoost (Friedman et al. 2000). It can be generalized to the multi-class setting, as explained in (Friedman et al. 2000).

#### 16.4.5 Boosting as functional gradient descent

Rather than deriving new versions of boosting for every different loss function, it is possible to derive a generic version, known as gradient boosting (Friedman 2001; Mason et al. 2000). To explain this, imagine minimizing  $\hat{f} = \operatorname{argmin}_f L(f)$  (16.49)

where  $f = (f(x_1), \dots, f(x_N))$  are the “parameters”. We will solve this stagewise, using gradient descent. At step  $m$ , let  $g_m$  be the gradient of  $L(f)$  evaluated at  $f = f_{m-1}$ :

$$g_m = \partial L(y_i, f(x_i)) \partial f(x_i) \quad f = f_{m-1} \quad (16.50)$$

Gradients of some common loss functions are given in Table 16.1. We then make the update  $f_m = f_{m-1} - \rho g_m$  (16.51) where  $\rho$  is the step length, chosen by

$$\rho_m = \operatorname{argmin} \rho L(f_m - \rho g_m) \quad (16.52)$$

This is called functional gradient descent. In its current form, this is not much use, since it only optimizes  $f$  at a fixed set of  $N$  points, so we do not learn a function that can generalize. However, we can modify the algorithm by fitting a weak learner to approximate the negative gradient signal. That is, we use this update

$$\gamma_m = \operatorname{argmin} \gamma N \sum_{i=1}^N (-g_i - \varphi(x_i; \gamma))^2 \quad (16.53)$$

The overall algorithm is summarized in Algorithm 6. (We have omitted the line search step, which is not strictly necessary, as argued in (Buhlmann and Hothorn 2007).)

If we apply this algorithm using squared loss, we recover L2Boosting. If we apply this algorithm to log-loss, we get an algorithm known as BinomialBoost (Buhlmann and Hothorn 2007). The advantage of this over LogitBoost is that it does not need to be able to do weighted fitting: it just applies any black-box regression model to the gradient vector. Also, it is relatively easy to extend to the multi-class case (see (Hastie et al. 2009, p387)). We can also apply this algorithm to other loss functions, such as the Huber loss (Section 7.4), which is more robust to outliers than squared error loss.

#### 16.4.6 Sparse boosting

Suppose we use as our weak learner the following algorithm: search over all possible variables  $j = 1: D$ , and pick the one  $j(m)$  that best predicts the residual vector:

$$j(m) = \operatorname{argmin}_j N \sum_{i=1}^N (r_i - \hat{\beta}_{jm} x_{ij})^2 \quad (16.54)$$

$$\hat{\beta}_{jm} = N \sum_{i=1}^N x_{ij} r_i / \sum_{i=1}^N x_{ij}^2 \quad (16.55)$$

$$\varphi_m(x) = \hat{\beta}_{jm} x_{jm} \quad (16.56)$$

This method, which is known as sparse boosting (Buhlmann and Yu 2006), is identical to the matching pursuit algorithm discussed in Section 13.2.3.1. It is clear that this will result in a sparse estimate, at least if  $M$  is small. To see this, let us rewrite the update as follows:

$$\beta_m := \beta_{m-1} + v(0, \dots, 0, \hat{\beta}_{jm}, m, 0, \dots, 0) \quad (16.57)$$

where the non-zero entry occurs in location  $j(m)$ . This is known as forward stagewise linear regression (Hastie et al. 2009, p608), which becomes equivalent to the LAR algorithm discussed in Section 13.4.2 as  $v \rightarrow 0$ . Increasing the number of steps  $m$  in boosting is analogous to decreasing the regularization penalty  $\lambda$ . If we modify boosting to allow some variable deletion steps (Zhao and Yu 2007), we can make it equivalent to the LARS

algorithm, which computes the full regularization path for the lasso problem. The same algorithm can be used for sparse logistic regression, by simply modifying the residual to be the appropriate negative gradient. Now consider a weak learner that is similar to the above, except it uses a smoothing spline instead of linear regression when mapping from  $x_j$  to the residual. The result is a sparse generalized additive model (see Section 16.3). It can obviously be extended to pick pairs of variables at a time. The resulting method often works much better than MARS (Buhlmann and Yu 2006).

#### 16.4.7 Multivariate adaptive regression trees (MART)

It is quite common to use CART models as weak learners. It is usually advisable to use a shallow tree, so that the variance is low. Even though the bias will be high (since a shallow tree is likely to be far from the “truth”), this will be compensated for in subsequent rounds of boosting. The height of the tree is an additional tuning parameter (in addition to  $M$ , the number of rounds of boosting, and  $\nu$ , the shrinkage factor). Suppose we restrict to trees with  $J$  leaves. If  $J = 2$ , we get a stump, which can only split on a single variable. If  $J = 3$ , we allow for two-variable interactions, etc. In general, it is recommended (e.g., in (Hastie et al. 2009, p363) and (Caruana and Niculescu-Mizil 2006)) to use  $J \approx 6$ . If we combine the gradient boosting algorithm with (shallow) regression trees, we get a model known as MART, which stands for “multivariate adaptive regression trees”. This actually includes a slight refinement to the basic gradient boosting algorithm: after fitting a regression tree to the residual (negative gradient), we re-estimate the parameters at the leaves of the tree to minimize the loss:

$$\gamma_{jm} = \operatorname{argmin}_{\gamma} \sum_{x_i \in R_{jm}} L(y_i, f_m - 1(x_i) + \gamma) \quad (16.58)$$

where  $R_{jm}$  is the region for leaf  $j$  in the  $m$ 'th tree, and  $\gamma_{jm}$  is the corresponding parameter (the mean response of  $y$  for regression problems, or the most probable class label for classification problems).

#### 16.4.8 Why does boosting work so well?

We have seen that boosting works very well, especially for classifiers. There are two main reasons for this. First, it can be seen as a form of  $L_1$  regularization, which is known to help prevent overfitting by eliminating “irrelevant” features. To see this, imagine pre-computing all possible

weak-learners, and defining a feature vector of the form  $\varphi(x) = [\varphi_1(x), \dots, \varphi_K(x)]$ . We could use L1 regularization to select a subset of these. Alternatively we can use boosting, where at each step, the weak learner creates a new  $\varphi_k$  on the fly. It is possible to combine boosting and L1 regularization, to get an algorithm known as L1-AdaBoost (Duchi and Singer 2009). Essentially this method greedily adds the best features (weak learners) using boosting, and then prunes off irrelevant ones using L1 regularization. Another explanation has to do with the concept of margin, which we introduced in Section 14.5.2.2. (Schapire et al. 1998; Ratsch et al. 2001) proved that AdaBoost maximizes the margin on the training data. (Rosset et al. 2004) generalized this to other loss functions, such as log-loss.

#### 16.4.9 A Bayesian view

So far, our presentation of boosting has been very frequentist, since it has focussed on greedily minimizing loss functions. A likelihood interpretation of the algorithm was given in (Neal and MacKay 1998; Meek et al. 2002). The idea is to consider a mixture of experts model of the form

$$p(y|x, \theta) = \sum_{m=1}^M \pi_m p(y|x, \theta_m) \quad (16.59)$$

where each expert  $p(y|x, \theta_m)$  is like a weak learner. We usually fit all  $M$  experts at once using EM, but we can imagine a sequential scheme, whereby we only update the parameters for one expert at a time. In the E step, the posterior responsibilities will reflect how well the existing experts explain a given data point; if this is a poor fit, these data points will have more influence on the next expert that is fitted. (This view naturally suggest a way to use a boosting-like algorithm for unsupervised learning: we simply sequentially fit mixture models, instead of mixtures of experts.) Notice that this is a rather “broken” MLE procedure, since it never goes back to update the parameters of an old expert. Similarly, if boosting ever wants to change the weight assigned to a weak learner, the only way to do this is to add the weak learner again with a new weight. This can result in unnecessarily large models. By contrast, the BART model (Chipman et al. 2006, 2010) uses a Bayesian version of backfitting to fit a small sum of weak learners (typically trees).

已思考 21s

下面分三块：逐字不遗漏的中文翻译 → 通俗解释 → 要点总结。

(注：原文个别地方存在排版小瑕疵，如 (16.31) 的分母应为  $p(\tilde{y} = -1 | x)$ 。翻译部分我忠实对应原文；在解释部分我会给出“更常见写法/理解”。)

---

## 翻译

Table 16.1 一些常用的损失函数、它们的梯度、它们的人群（总体）极小化器  $f^*$ ，以及一些用于最小化该损失的算法。对二分类问题，我们假设  $\tilde{y}_i \in \{-1, +1\}$ 、 $y_i \in \{0, 1\}$ ，并且  $\pi_i = \text{sigm}(2f(x_i))$ 。对回归问题，我们假设  $y_i \in \mathbb{R}$ 。

Name	Loss	Derivative	$f^*$	Algorithm
Squared error	$\frac{1}{2}(y_i - f(x_i))^2$	$y_i - f(x_i)$	$E[y   x_i]$	L2Boosting
Absolute error	$( y_i - f(x_i) )$			$\text{sgn}(y_i - f(x_i))$
Exponential loss	$\exp(-\tilde{y}_i f(x_i))$	$-\tilde{y}_i \exp(-\tilde{y}_i f(x_i))$	$\frac{1}{2} \log \frac{\pi_i}{1-\pi_i}$	AdaBoost
Logloss	$\log(1 + e^{-\tilde{y}_i f_i})$	$y_i - \pi_i$	$\frac{1}{2} \log \frac{\pi_i}{1-\pi_i}$	LogitBoost

Figure 16.8 在图 16.10 数据上，用决策树桩作为弱学习器的 AdaBoost 表现。训练（实线蓝色）与测试（虚线红色）错误率随迭代轮数的变化。

Figure 16.9 二分类里各种损失函数的示意。横轴是间隔  $y\eta$ ，纵轴是损失。对数损失 (log loss) 使用以 2 为底的对数。

Figure 16.10 在一个二分类数据集上，用决策树桩作为弱学习器的 AdaBoost 示例。黑色程度表示对红类的置信度；白色程度表示对蓝类的置信度；数据点大小表示其权重；黄色为决策边界。(a) 第 1 轮后；(b) 第 3 轮后；(c) 第 120 轮后。

---

## 16.4 提升 (Boosting)

Boosting (Schapire 和 Freund 2012) 是一种贪心算法，用来拟合式 (16.3) 形式的自适应基函数模型，其中  $\phi_m$  由称为弱学习器/基学习器的算法生成。算法的工作方式是：把弱学习器按顺序应用到加权的数据上，之前回合中被误分的样本会被赋予更大的权重。弱学习器可以是任意分类或回归算法，但常用 CART 模型。1998 年，已故的 Leo Breiman 把“弱

学习器为浅层决策树”的 boosting 称为“**全世界最好用的现成分类器**” (Hastie 等, 2009, 第 340 页)。这一点得到 (Caruana 和 Niculescu-Mizil 2006) 对 10 种分类器进行广泛经验对比的支持: **提升的决策树**在误分类率和概率校准 (用 ROC 曲线评判) 两方面都是最好的。(第二好的是 Breiman 发明的**随机森林**; 见第 16.2.5 节。) 相比之下, **单棵**决策树表现很差。

Boosting 最初在**计算学习理论**文献中推导 (Schapire 1990; Freund 和 Schapire 1996), 聚焦二分类。文中证明: 只要弱学习器**总能比瞎猜稍好**, 就能把 (在训练集上的) 性能提升到任意高。例如, 在图 16.8 中, 我们画出了图 16.10 那个二维数据集上“提升的树桩”的训练/测试误差。可见训练误差很快接近 0; 更令人惊讶的是**测试误差在训练误差已到 0 后仍继续下降** (尽管最终会回升)。因此 boosting 对过拟合非常耐受。(提升的树桩构成了一个非常成功的人脸检测器 (Viola 和 Jones 2001) 的基础, 它产生了图 1.6 的结果, 并被许多数码相机采用。)

鉴于其令人震惊的经验成功, 统计学家开始关注这种方法。Breiman (1998) 表明 boosting 可以解释为**函数空间的梯度下降**。Friedman 等 (2000) 进一步扩展, 展示了如何把 boosting 拓展到多种损失 (包括回归、稳健回归、泊松回归等)。本节将给出这种统计解释; 更多细节见 (Buhlmann 和 Hothorn 2007) 及 (Hastie 等 2009, 第 10 章)。

#### 16.4.1 前向分段可加建模 (Forward stagewise additive modeling)

Boosting 的目标是求解优化问题

$$\min_f \sum_{i=1}^N L(y_i, f(x_i)) \quad (16.25)$$

其中  $L(y, \hat{y})$  是某个损失函数,  $f$  被假设为式 (16.3) 的 ABM。常见损失见表 16.1。若用**平方误差损失**, 最优估计为

$$f^*(x) = \arg \min_{f(x)} \mathbb{E}_{Y|x}(Y - f(x))^2 = \mathbb{E}[Y | x] \quad (16.26)$$

这在实践中无法直接计算, 因为需要知道真实的  $p(y | x)$ , 因此称其为**总体极小化器** (频率学意义下的期望)。下面我们会看到 boosting 会去近似这个条件期望。

对二分类, 显然的损失是**0-1 损失**, 但它不可导; 常用**对数损失 (logloss)**, 它是 0-1 损失的一个**凸上界** (见 6.5.5)。在这种情况下, 可以证明最优估计为

$$f^*(x) = \frac{1}{2} \log \frac{p(\tilde{y} = 1 | x)}{p(\tilde{y} = -1 | x)} \quad (16.27)$$

其中  $\tilde{y} \in \{-1, +1\}$ 。多类情形可推广, 此处不展开。另一种凸上界是**指数损失**

$$L(\tilde{y}, f) = \exp(-\tilde{y}f) \quad (16.28)$$

其曲线见图 16.9。它相对 logloss 有一些计算优势（见后述）。该损失的最优估计同样是  $f^*(x) = \frac{1}{2} \log \frac{p(\tilde{y}=1|x)}{p(\tilde{y}=-1|x)}$ 。为看清这一点，对（给定  $x$  的）期望损失对  $f(x)$  求导并令其为 0：

$$\frac{\partial}{\partial f(x)} E[e^{-\tilde{y}f(x)}|x] = \frac{\partial}{\partial f(x)} [p(\tilde{y}=1|x)e^{-f(x)} + p(\tilde{y}=-1|x)e^{f(x)}] \quad (16.29)$$

$$= -p(\tilde{y}=1|x)e^{-f(x)} + p(\tilde{y}=-1|x)e^{f(x)} \quad (16.30)$$

$$= 0 \Rightarrow \frac{p(\tilde{y}=1|x)}{p(\tilde{y}=-1|x)} = e^{2f(x)} \quad (16.31)$$

因此两种情况下 boosting 都应当去近似（**一半的对数几率**）。

由于直接求最优  $f$  很难，我们采用**逐步**策略。初始化：

$$f_0(x) = \arg \min_{\gamma} \sum_{i=1}^N L(y_i, f(x_i; \gamma)) \quad (16.32)$$

例如平方误差可令  $f_0(x) = \bar{y}$ ；logloss 或指数损失可令

$f_0(x) = \frac{1}{2} \log \frac{\hat{\pi}}{1-\hat{\pi}}$ ，其中  $\hat{\pi} = \frac{1}{N} \sum_{i=1}^N \mathbf{I}(y_i = 1)$ 。也可用更强的基线模型（如 GLM）。

第  $m$  次迭代，计算

$$(\beta_m, \gamma_m) = \arg \min_{\beta, \gamma} \sum_{i=1}^N L(y_i, f_{m-1}(x_i) + \beta \phi(x_i; \gamma)) \quad (16.33)$$

然后更新

$$f_m(x) = f_{m-1}(x) + \beta_m \phi(x; \gamma_m) \quad (16.34)$$

关键点：**不回头改早先的参数**，因此称为**前向分段可加**。重复到固定迭代数  $M$ 。 $M$  是主要的调参量；常用**验证集**监控性能，一旦开始变差就停止（**早停**）。也可用 AIC/BIC 等准则。实践中，用

$$f_m(x) = f_{m-1}(x) + \nu \beta_m \phi(x; \gamma_m), \quad 0 < \nu \leq 1 \quad (16.35)$$

做“**部分更新/收缩 (shrinkage)**”（典型  $\nu = 0.1$ ）往往测试集表现更好。下面讨论 (16.33) 子问题的求解，这取决于损失形式，与弱学习器的形式无关。

## 16.4.2 L2Boosting

若用平方误差损失，则第  $m$  步的损失可写成

$$L(y_i, f_{m-1}(x_i) + \beta \phi(x_i; \gamma)) = (r_{im} - \phi(x_i; \gamma))^2 \quad (16.36)$$

其中  $r_{im} = y_i - f_{m-1}(x_i)$  为当前残差，并把  $\beta = 1$ （不失一般性）。因此用弱学习器去拟合残差即可得到新基函数。这称为 L2Boosting/最小二乘提升（Bühlmann 和 Yu 2003）。在 16.4.6 我们会看到，配合合适弱学习器，它与 LARS 等价，可做变量选择（见 13.4.2）。

### 16.4.3 AdaBoost

考虑二分类的指数损失。第  $m$  步需最小化

$$L_m(\phi) = \sum_{i=1}^N \exp[-\tilde{y}_i(f_{m-1}(x_i) + \beta\phi(x_i))] = \sum_{i=1}^N w_{i,m} e^{-\beta\tilde{y}_i\phi(x_i)} \quad (16.37)$$

其中  $w_{i,m} = e^{-\tilde{y}_i f_{m-1}(x_i)}$  是样本  $i$  的权重， $\tilde{y}_i \in \{-1, +1\}$ 。该目标可改写为

$$L_m = e^{-\beta} \sum_{\tilde{y}_i=\phi(x_i)} w_{i,m} + e^{\beta} \sum_{\tilde{y}_i \neq \phi(x_i)} w_{i,m} \quad (16.38)$$

$$= (e^\beta - e^{-\beta}) \sum_{i=1}^N w_{i,m} \mathbf{I}(\tilde{y}_i \neq \phi(x_i)) + e^{-\beta} \sum_{i=1}^N w_{i,m} \quad (16.39)$$

因此最优的新增函数为

$$\phi_m = \arg \min_{\phi} \sum_i w_{i,m} \mathbf{I}(\tilde{y}_i \neq \phi(x_i)) \quad (16.40)$$

也就是：对加权数据运行弱学习器。把  $\phi_m$  代入并对  $\beta$  求解得

$$\beta_m = \frac{1}{2} \log \frac{1 - \text{err}_m}{\text{err}_m} \quad (16.41)$$

其中

$$\text{err}_m = \frac{\sum_i w_{i,m} \mathbf{I}(\tilde{y}_i \neq \phi_m(x_i))}{\sum_i w_{i,m}} \quad (16.42)$$

整体更新为

$$f_m(x) = f_{m-1}(x) + \beta_m \phi_m(x) \quad (16.43)$$

下一轮的权重为

$$w_{i,m+1} = w_{i,m} e^{-\beta_m \tilde{y}_i \phi_m(x_i)} = w_{i,m} e^{\beta_m (2\mathbf{I}(\tilde{y}_i \neq \phi_m(x_i)) - 1)} = w_{i,m} e^{2\beta_m \mathbf{I}(\tilde{y}_i \neq \phi_m(x_i))} \quad (16.44)$$

由于归一化时  $e^{-\beta_m}$  会相消，可略去。得到 Algorithm 7 (Adaboost.M1)。其运行示例如图 16.10：迭代多次后可“雕刻”出复杂的决策边。AdaBoost 极慢过拟合，如图 16.8 所示（详见 16.4.8）。

#### 16.4.4 LogitBoost

指数损失的问题在于：它对误分样本施加**指教级**的巨大权重（见图 16.9 左侧），对**异常/错标**很敏感；且  $e^{-\tilde{y}f}$  不是任意  $\tilde{y} \in \{-1, +1\}$  的 pmf 的对数，因而**无法**从  $f(x)$  直接恢复概率。

自然的替代是**对数损失**：它对错误的惩罚是**线性的**（见图 16.9）；并且可以从学到的  $f$  中抽取概率：

$$p(y = 1 | x) = \frac{e^{f(x)}}{e^{-f(x)} + e^{f(x)}} = \frac{1}{1 + e^{-2f(x)}} \quad (16.47)$$

目标是最小化期望 logloss：

$$L_m(\phi) = \sum_{i=1}^N \log(1 + \exp(-2\tilde{y}_i(f_{m-1}(x_i) + \phi(x_i)))) \quad (16.48)$$

对该目标做**牛顿更新**（类似 IRLS）即可推导 Algorithm 8，称为 LogitBoost (Friedman 等, 2000)，可推广到多类。

#### 16.4.5 把 Boosting 看成函数梯度下降

不必对每种损失都重新推导一种 boosting，可以得到**通用版本**：**梯度提升** (gradient boosting) (Friedman 2001; Mason 等 2000)。设

$$\hat{f} = \arg \min_f L(f) \quad (16.49)$$

其中  $f = (f(x_1), \dots, f(x_N))$  看作“参数”。第  $m$  步，在  $f = f_{m-1}$  处的梯度为

$$g_{im} = \left. \frac{\partial L(y_i, f(x_i))}{\partial f(x_i)} \right|_{f=f_{m-1}} \quad (16.50)$$

（常见损失的梯度见表 16.1。）更新

$$f_m = f_{m-1} - \rho_m g_m \quad (16.51)$$

步长  $\rho_m$  由

$$\rho_m = \arg \min_\rho L(f_{m-1} - \rho g_m) \quad (16.52)$$

这称为**函数梯度下降**。直接这样做只在  $N$  个点上优化，无法泛化；因此改为：用弱学习器去**逼近负梯度信号**：

$$\gamma_m = \arg \min_\gamma \sum_{i=1}^N (-g_{im} - \phi(x_i; \gamma))^2 \quad (16.53)$$

整体算法见 Algorithm 6 (省略了并非必要的线搜索; 参见 Bühlmann 和 Hothorn 2007)。

对**平方损失**应用该法得到 L2Boosting; 对logloss得到 BinomialBoost (其优于 LogitBoost 的一点是**不需要加权拟合**: 直接把任何回归黑盒拟合到梯度向量上), 并易于扩展到多类。还可用于其他损失, 如Huber 损失 (比平方误差更抗异常)。

#### 16.4.6 稀疏提升 (Sparse boosting)

若弱学习器如下: 在所有变量  $j = 1:D$  中搜索, 使得它对**残差向量**拟合最好:

$$j^{(m)} = \arg \min_j \sum_{i=1}^N (r_{im} - \hat{\beta}_{jm} x_{ij})^2, \quad \hat{\beta}_{jm} = \frac{\sum_i x_{ij} r_{im}}{\sum_i x_{ij}^2}, \quad \phi_m(x) \in \hat{\beta}_{jm} x_{jm} \quad (16.54)$$

该方法称为**稀疏提升** (Bühlmann 和 Yu 2006), 与 13.2.3.1 节的**匹配追踪**相同。显然当  $M$  较小时会得到**稀疏估计**。把更新写为

$$\beta^{(m)} := \beta^{(m-1)} + v(0, \dots, 0, \hat{\beta}_{jm}, 0, \dots, 0) \quad (16.57)$$

(唯一非零项在  $j^{(m)}$  位置。) 这就是**前向分段线性回归** (Hastie 等 2009, 第 608 页); 当  $v \rightarrow 0$  时等价于 LAR (13.4.2)。增加步数  $m$  相当于**减小正则化**  $\lambda$ 。若允许变量删除 (Zhao 和 Yu 2007), 可使其等价于 LARS, 得到 lasso 的**整条正则化路径**。同一思路改为**逻辑回归**只需把残差替换为相应的**负梯度**。若把线性回归换成**一维平滑样条**, 就得到**稀疏 GAM** (见 16.3), 也可一次选择成对变量; 该方法往往**明显优于** MARS (Bühlmann 和 Yu 2006)。

#### 16.4.7 多元自适应回归树 (MART)

常用 CART 作为弱学习器。通常建议用**浅树**以降低方差; 虽然偏差会大, 但后续迭代会补偿。树高是额外的调参量 (除  $M$ 、 $v$  外)。若限制树有  $J$  片叶:  $J=2$  得到树桩 (只能分一个变量);  $J=3$  可捕捉二变量交互; 一般建议  $J \approx 6$  (Hastie 等 2009, 第 363 页; Caruana 和 Niculescu-Mizil 2006)。把**梯度提升**与 (浅) 回归树结合就得 MART (multivariate adaptive regression trees)。它在基本梯度提升上略作改进: 把树拟合到**残差/负梯度后, 在每个叶子上重新估计参数**以最小化损失:

$$\gamma_{jm} = \arg \min_{\gamma} \sum_{x_i \in R_{jm}} L(y_i, f_{m-1}(x_i) + \gamma) \quad (16.58)$$

其中  $R_{jm}$  是第  $m$  棵树的第  $j$  片叶对应的区域,  $\gamma_{jm}$  为对应参数 (回归时为均值, 分类时为最可能类别等)。

#### 16.4.8 为什么 boosting 效果这么好?

主要有两点。其一, 它可视作一种  $L_1$  正则化, 通过去掉“无关特征”来防过拟合。设把所有可能的弱学习器预先计算好, 得到特征向量  $\phi(x) = [\phi_1(x), \dots, \phi_K(x)]$ , 可用  $L_1$  选择其

子集；或用 boosting 逐步在线生成  $\phi_k$ 。还可把二者结合成 L1-Adaboost (Duchi 和 Singer 2009)：贪心加入最好特征，再用  $L_1$  剪掉无用特征。

其二，与间隔 (margin) 有关（见 14.5.2.2）。(Schapire 等 1998; Rätsch 等 2001) 证明 AdaBoost 在训练数据上最大化间隔；(Rosset 等 2004) 把这一点推广到 logloss 等其他损失。

### 16.4.9 一个贝叶斯视角

到目前为止，我们的表述偏频率学（贪心最小化损失）。(Neal 和 MacKay 1998; Meek 等 2002) 给出了似然解释：考虑一个专家混合模型

$$p(y | x, \theta) = \sum_{m=1}^M \pi_m p(y | x, \gamma_m) \quad (16.59)$$

每个专家  $p(y | x, \gamma_m)$  类似一个弱学习器。通常用 EM 同时拟合所有  $M$  个专家，但也可设想一个序贯方案：每次只更新一个专家的参数。E 步里，后验职责反映了“现有专家对某个样本解释得多好”；若很差，该样本将在下一位专家的拟合中权重更大。（这一视角自然地给了无监督的“提升式”思路：顺序地拟合混合模型，而不是专家混合。）注意这是一种“残缺的 MLE”——从不回头更新旧专家的参数；类似地，若 boosting 想改变某个弱学习器的权重，唯一办法是再加一次该弱学习器并用新权重，这可能导致过大的模型。相比之下，BART (Chipman 等 2006, 2010) 用贝叶斯版 backfitting 来拟合少量弱学习器（通常是树）。

## 解释（通俗易懂）

### 1) Boosting 到底干啥？

- 把最终模型写成很多“小模型”（弱学习器）的加和：  

$$f_M(x) = \sum_{m=1}^M \beta_m \phi_m(x).$$
- 第  $m$  轮专门去修正前  $m-1$  轮“搞不定”的样本：要么加权它们更重  
 (AdaBoost/LogitBoost)，要么直接拟合残差/负梯度 (L2/梯度提升)。

### 2) 三种常见口味

- L2Boosting (回归)：弱学习器拟合残差即可；像在做“把误差一点点吃掉”。
- AdaBoost (指数损失)：误分样本权重按指数级上升 → 下一轮更关注它们；输出的  $f$  近似对数几率的一半。优点：常常很强，甚至慢过拟合；缺点：对错标/异常敏感。
- LogitBoost (对数损失)：对错误的惩罚更温和，能直接输出概率  $p(y=1 | x) = 1/(1 + e^{-2f})$ 。

### 3) 梯度提升 (Gradient Boosting) = 通用作法

- 把“在函数上做梯度下降”离散化：**负梯度**当作“要学习的新目标”，让弱学习器去拟合它。
- 这就把不同损失统一起来：平方损失 → L2Boosting；logloss → BinomialBoost；Huber 等也可以。

### 4) 收缩与早停

- 每次只走**小步** ( $\nu \approx 0.1$ )，再配合**较多轮数与早停**，往往测试集更好、更稳。

### 5) 让模型“稀疏/可解释”

- 若弱学习器每次只挑**一个变量**做线性（或样条）拟合，就是**稀疏提升**：
  - 等价于**匹配追踪/前向分段回归**；
  - 细粒度下与 LAR/LARS、lasso 有对应关系；
  - 换样条就是**稀疏 GAM**，效果常优于 MARS。

### 6) 用树当弱学习器：MART / GBDT

- 用**浅树**（如 6 叶）+ 梯度提升  $\Rightarrow$  MART/GBDT。
- 先拟合负梯度得到一棵树，再在每个叶子上**重新估计叶值**以最小化当前损失。

### 7) 为什么强？

- 类似在做\*\* $L_1$  特征选择\*\*（一步步加最有用的基函数），能抑制过拟合。
- 同时它在训练集上**增大间隔** (margin)，理论上与泛化误差界有关。
- 代价：模型会变大，可解释性下降；若想“小而强”，可看 BART（贝叶斯少树求和）。

### 8) 读图提示

- **表 16.1：**不同损失的梯度/总体解/常见算法的一览。
- **图 16.8：**AdaBoost 训练误差迅速→0；测试误差在很长一段还持续下降（抗过拟合）。
- **图 16.9：**0–1 损失是“跳台”；指数/对数损失是**光滑上界**。指数左侧上升更猛→对误分更敏感。
- **图 16.10：**树桩 + AdaBoost：1、3、120 轮后决策边越来越复杂，样本权重（点大小）随误分而增大。

## 要点总结

- **Boosting：**把很多弱模型按顺序叠加；每轮针对“上一轮没搞好”的地方发力。

- **三条主线：**
  - L2Boosting (回归, 拟合残差) ;
  - AdaBoost (指数损失, 权重加倍关照误分, 强但怕异常) ;
  - Logit/Gradient Boosting (对数/通用梯度损失, 能输出概率, 鲁棒性更好) 。
- **训练细节：**前向阶段式更新; 步长  $\nu$  (收缩) + 轮数  $M$  (早停/验证集选择) 是关键超参。
- **稀疏提升**把 boosting 与 LAR/Lasso 接上, 能做**变量选择**; 换样条即**稀疏 GAM**。
- **树作弱学习器**  $\Rightarrow$  MART/GBDT: 浅树、叶值回归, 实用且强。
- **为何好用:** 等同于 $L_1$  特征选择+**增大间隔**; 实践中**抗过拟合**、预测准。
- **贝叶斯视角:** 像在顺序拟合“专家混合”; 想要“小而强”的树加和, 可看 BART。

这一小节主要在讲什么? 我不太懂

已思考 16s

没问题! 这小节 (16.4: Boosting) 在讲——**怎样把很多“很弱的小模型”按顺序叠加, 做成一个很强的大模型**, 以及几种常见做法、为什么有效、怎么用。

## 一句话版

Boosting=前向逐步地往模型里**加基函数** (弱学习器), 每一轮专治**上轮没学好的部分** (用加权或拟合残差/负梯度), 配合小步长 (收缩) 与早停, 得到强大又不易过拟合的模型。

## 核心思路 (通俗)

### 1. 模型形态:

$$f_M(x) = \sum_{m=1}^M \beta_m \phi_m(x)。 \phi_m \text{ 由弱学习器 (常用浅层树) 产生。}$$

### 2. 怎么迭代:

- 设损失  $L(y, f(x))$ , 目标是最小化  $\sum_i L(y_i, f(x_i))$ 。
- 第  $m$  轮只做**前向添加**: 在  $f_{m-1}$  的基础上, 加一个新基函数让损失降得最多; **不回头改旧参数** (叫“前向分段可加”)。
- 常配合**收缩** (学习率  $\nu \approx 0.1$ ) 和**早停** (看验证集)。

### 3. 几种口味 (跟损失有关)

- L2Boosting (平方误差, 回归) : 弱学习器去**拟合残差**。

- AdaBoost (指数损失, 二分类) : 给**误分样本加大权重**, 下一轮更关注它们; 学到的  $f$  近似**对数几率的一半**。很强但对**错标/离群敏感**。
  - LogitBoost (对数损失) : 惩罚更温和, 还能直接输出概率  $p(y=1|x) = 1/(1 + e^{-2f})$ 。
  - **梯度提升** (Gradient Boosting) : 通用版——让弱学习器去**拟合负梯度** (一切损失都能用)。树做弱学习器  $\Rightarrow$  MART/GBDT。
4. **稀疏提升**: 每轮只挑**一个变量** (线性或样条) 来拟合残差  $\rightarrow$  等价于前向逐步/匹配追踪; 和 LAR/Lasso 有对应, 能做**变量选择**; 换成样条就是**稀疏 GAM**。
5. **为什么好用**
- 像在做 $L_1$ 型特征选择: 逐步加最有用的基函数, 抑制过拟合。
  - **扩大间隔 (margin)** : 理论上与更好的泛化有关。
  - 代价: 模型可能很大、解释性下降 (可用小学习率+早停控制)。
- 

## 和图/表对应

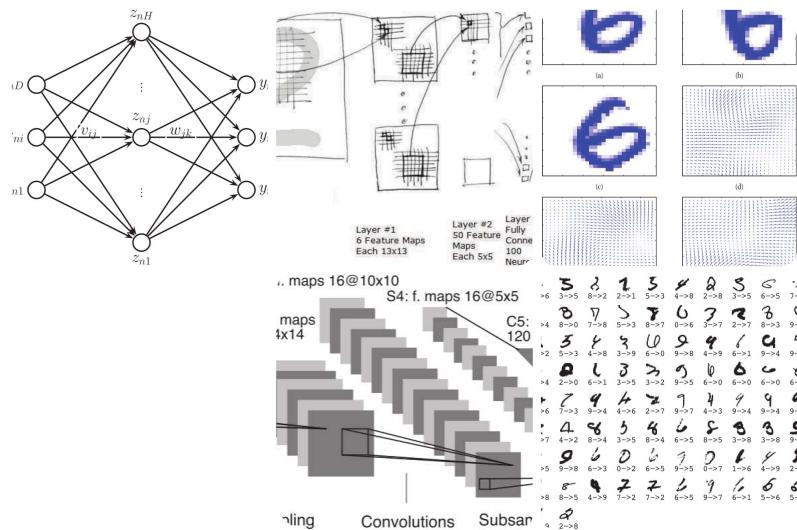
- **表16.1:** 不同损失的梯度与“理想解” $f^*$  (回归取条件期望; 绝对误差取条件中位数; 分类两种损失都对准**半对数几率**)。
  - **图16.8:** AdaBoost 训练误差很快 $\approx 0$ , 但测试误差还能继续降一段 (抗过拟合)。
  - **图16.9:** 0-1 损失不可导; 指数/对数损失是它的**平滑上界**, 指数左侧上升更陡 $\rightarrow$ 更“关照”误分点。
  - **图16.10:** 树桩 + AdaBoost: 1/3/120 轮后决策边界越来越复杂, 误分点权重 (点大小) 越来越大。
- 

## 实用小抄

- 做回归: 先试 GBDT (**平方损失**) ; 学习率 0.05–0.1, 树深 (叶子数) 小一些 (如深度 $\approx 3$ –6), 配**早停**。
  - 做分类:
    - 噪声/错标多  $\rightarrow$  用**对数损失** (Logit/Gradient Boosting)。
    - 干净数据、追求强判别  $\rightarrow$  AdaBoost 也很猛, 但留心异常点。
  - 变量很多想做选择  $\rightarrow$  **稀疏提升**或在 GBDT 上加**特征重要性/正则**。
- 

## 记住这几句

- Boosting = 逐轮拟合残差/负梯度 + 叠加弱学习器。
- AdaBoost 用指数损失、样本加权；Logit/Gradient Boosting 更稳还能给概率；GBDT/MART = 梯度提升 + 浅树。
- 小学习率 + 多轮 + 早停 是好用的默认配置。



翻译并解释以下内容，翻译不能有遗漏，解释请通俗易懂且详尽，因为我的数学基础不太好，最后总结要点。

Figure 16.11 A neural network with one hidden layer.

Figure 16.12 The convolutional neural network from (Simard et al. 2003).

Figure 16.13 Several synthetic warpings of a handwritten digit. Based on Figure 5.14 of (Bishop 2006a).

Figure 16.14 LeNet5, a convolutional neural net for classifying handwritten digits.

Figure 16.15 These are the 82 errors made by LeNet5 on the 10,000 test cases of MNIST. Below each image is a label of the form correct-label → estimated-label.

## 16.5 Feedforward neural networks (multilayer perceptrons)

A feedforward neural network, aka multi-layer perceptron (MLP), is a series of logistic regression models stacked on top of each other, with the final layer being either another logistic regression or a linear regression model, depending on whether we are solving a classification or regression problem. For example, if we have two layers, and we are solving a regression problem, the model has

the form

$$p(y|x, \theta) = N(y|w^T z(x), \sigma^2) \quad (16.60)$$

$$z(x) = g(Vx) = [g(v^T_1 x), \dots, g(v^T_H x)] \quad (16.61)$$

where  $g$  is a non-linear activation or transfer function (commonly the logistic function),  $z(x) = \varphi(x, V)$  is called the hidden layer (a deterministic function of the input),  $H$  is the number of hidden units,  $V$  is the weight matrix from the inputs to the hidden nodes, and  $w$  is the weight vector from the hidden nodes to the output. It is important that  $g$  be nonlinear, otherwise the whole model collapses into a large linear regression model of the form  $y = w^T(Vx)$ .

One can show that an MLP is a universal approximator, meaning it can model any suitably smooth function, given enough hidden units, to any desired level of accuracy (Hornik 1991). To handle binary classification, we pass the output through a sigmoid, as in a GLM:

$$p(y|x, \theta) = \text{Ber}(y|\text{sigm}(w^T z(x))) \quad (16.62)$$

We can easily extend the MLP to predict multiple outputs. For example, in the regression case, we have

$$p(y|x, \theta) = N(y|W \varphi(x, V), \sigma^2 I) \quad (16.63)$$

See Figure 16.11 for an illustration. If we add mutual inhibition arcs between the output units, ensuring that only one of them turns on, we can enforce a sum-to-one constraint, which can be used for multi-class classification.

The resulting model has the form

$$p(y|x, \theta) = \text{Cat}(y|S(Wz(x))) \quad (16.64)$$

### 16.5.1 Convolutional neural networks

The purpose of the hidden units is to learn non-linear combinations of the original inputs; this is called feature extraction or feature construction. These hidden features are then passed as input to the final GLM. This approach is particularly useful for problems where the original input features are not very individually informative. For example, each pixel in an image is not very informative; it is the combination of pixels that tells us what objects are present. Conversely, for a task such as document classification using a bag of words representation, each feature (word count) is informative on its own, so extracting "higher order" features is less important. Not surprisingly, then, much of the work in neural networks has been motivated by visual pattern recognition (e.g., (LeCun et al. 1989)), although they have also been applied to other types of data, including text (e.g., (Collobert and

Weston 2008)).

A form of MLP which is particularly well suited to 1d signals like speech or text, or 2d signals like images, is the convolutional neural network. This is an MLP in which the hidden units have local receptive fields (as in the primary visual cortex), and in which the weights are tied or shared across the image, in order to reduce the number of parameters. Intuitively, the effect of such spatial parameter tying is that any useful features that are "discovered" in some portion of the image can be re-used everywhere else without having to be independently learned. The resulting network then exhibits translation invariance, meaning it can classify patterns no matter where they occur inside the input image.

Figure 16.12 gives an example of a convolutional network, designed by Simard and colleagues (Simard et al. 2003), with 5 layers (4 layers of adjustable parameters) designed to classify  $29 \times 29$  gray-scale images of handwritten digits from the MNIST dataset (see Section 1.2.1.3). In layer 1, we have 6 feature maps each of which has size  $13 \times 13$ . Each hidden node in one of these feature maps is computed by convolving the image with a  $5 \times 5$  weight matrix (sometimes called a kernel), adding a bias, and then passing the result through some form of nonlinearity. There are therefore  $13 \times 13 \times 6 = 1014$  neurons in Layer 1, and  $(5 \times 5 + 1) \times 6 = 156$  weights. (The "+1" is for the bias.) If we did not share these parameters, there would be  $1014 \times 26 = 26,364$  weights at the first layer. In layer 2, we have 50 feature maps, each of which is obtained by convolving each feature map in layer 1 with a  $5 \times 5$  weight matrix, adding them up, adding a bias, and passing through a nonlinearity. There are therefore  $5 \times 5 \times 50 = 1250$  neurons in Layer 2,  $(5 \times 5 + 1) \times 6 \times 50 = 7800$  adjustable weights (one kernel for each pair of feature maps in layers 1 and 2), and  $1250 \times 26 = 32,500$  connections. Layer 3 is fully connected to layer 2, and has 100 neurons and  $100 \times (1250 + 1) = 125,100$  weights. Finally, layer 4 is also fully connected, and has 10 neurons, and  $10 \times (100 + 1) = 1010$  weights. Adding the above numbers, there are a total of 3,215 neurons, 134,066 adjustable weights, and 184,974 connections. This model is usually trained using stochastic gradient descent (see Section 16.5.4 for details). A single pass over

the data set is called an epoch. When Mike O'Neill did these experiments in 2006, he found that a single epoch took about 40 minutes (recall that there are 60,000 training examples in MNIST). Since it took about 30 epochs for the error rate to converge, the total training time was about 20 hours. Using this technique, he obtained a misclassification rate on the 10,000 test cases of about 1.40%.

To further reduce the error rate, a standard trick is to expand the training set by including distorted versions of the original data, to encourage the network to be invariant to small changes that don't affect the identity of the digit. These can be created by applying a random flow field to shift pixels around. See Figure 16.13 for some examples. (If we use online training, such as stochastic gradient descent, we can create these distortions on the fly, rather than having to store them.) Using this technique, Mike O'Neill obtained a misclassification rate on the 10,000 test cases of about 0.74%, which is close to the current state of the art.

Yann Le Cun and colleagues (LeCun et al. 1998) obtained similar performance using a slightly more complicated architecture shown in Figure 16.14. This model is known as LeNet5, and historically it came before the model in Figure 16.12. There are two main differences. First, LeNet5 has a subsampling layer between each convolutional layer, which either averages or computes the max over each small window in the previous layer, in order to reduce the size, and to obtain a small amount of shift invariance. The convolution and sub-sampling combination was inspired by Hubel and Wiesel's model of simple and complex cells in the visual cortex (Hubel and Wiesel 1962), and it continues to be popular in neurally-inspired models of visual object recognition (Riesenhuber and Poggio 1999). A similar idea first appeared in Fukushima's neocognitron (Fukushima 1975), though no globally supervised training algorithm was available at that time.

The second difference between LeNet5 and the Simard architecture is that the final layer is actually an RBF network rather than a more standard sigmoidal or softmax layer. This model gets a test error rate of about 0.95% when trained with no distortions, and 0.8% when

trained with distortions. Figure 16.15 shows all 82 errors made by the system. Some are genuinely ambiguous, but several are errors that a person would never make. A web-based demo of the LeNet5 can be found at <http://yann.lecun.com/exdb/lenet/index.html>.

Of course, classifying isolated digits is of limited applicability: in the real world, people usually write strings of digits or other letters. This requires both segmentation and classification. Le Cun and colleagues devised a way to combine convolutional neural networks with a model similar to a conditional random field (described in Section 19.6) to solve this problem. The system was eventually deployed by the US postal service. (See (LeCun et al. 1998) for a more detailed account of the system, which remains one of the best performing systems for this task.)

### 16.5.2 Other kinds of neural networks

Other network topologies are possible besides the ones discussed above. For example, we can have skip arcs that go directly from the input to the output, skipping the hidden layer; we can have sparse connections between the layers; etc. However, the MLP always requires that the weights form a directed acyclic graph. If we allow feedback connections, the model is known as a recurrent neural network; this defines a nonlinear dynamical system, but does not have a simple probabilistic interpretation. Such RNN models are currently the best approach for language modeling (i.e., performing word prediction in natural language) (Tomas et al. 2011), significantly outperforming the standard n-gram-based methods discussed in Section 17.2.2.

If we allow symmetric connections between the hidden units, the model is known as a Hopfield network or associative memory; its probabilistic counterpart is known as a Boltzmann machine (see Section 27.7) and can be used for unsupervised learning.

### 16.5.3 A brief history of the field

Neural networks have been the subject of great interest for many decades, due to the desire to understand the brain, and to build learning machines. It is not possible to review the entire history here. Instead, we just give a few “edited highlights”.

The field is generally viewed as starting with McCulloch and Pitts (McCullich and Pitts 1943), who devised a simple

mathematical model of the neuron in 1943, in which they approximated the output as a weighted sum of inputs passed through a threshold function,  $y = I(\sum_i w_i x_i > \theta)$ , for some threshold  $\theta$ . This is similar to a sigmoidal activation function. Frank Rosenblatt invented the perceptron learning algorithm in 1957, which is a way to estimate the parameters of a McCulloch-Pitts neuron (see Section 8.5.4 for details). A very similar model called the adaline (for adaptive linear element) was invented in 1960 by Widrow and Hoff.

In 1969, Minsky and Papert (Minsky and Papert 1969) published a famous book called “Perceptrons” in which they showed that such linear models, with no hidden layers, were very limited in their power, since they cannot classify data that is not linearly separable. This considerably reduced interest in the field.

In 1986, Rumelhart, Hinton and Williams (Rumelhart et al. 1986) discovered the backpropagation algorithm (see Section 16.5.4), which allows one to fit models with hidden layers. (The backpropagation algorithm was originally discovered in (Bryson and Ho 1969), and independently in (Werbos 1974); however, it was (Rumelhart et al. 1986) that brought the algorithm to people’s attention.) This spawned a decade of intense interest in these models.

In 1987, Sejnowski and Rosenberg (Sejnowski and Rosenberg 1987) created the famous NETtalk system, that learned a mapping from English words to phonetic symbols which could be fed into a speech synthesizer. An audio demo of the system as it learns over time can be found at

<http://www.cnl.salk.edu/ParallelNetsPronounce/nettalk.mp>

3. The system starts by “babbling” and then gradually learns to pronounce English words. NETtalk learned a distributed representation (via its hidden layer) of various sounds, and its success spawned a big debate in psychology between connectionism, based on neural networks, and computationalism, based on syntactic rules. This debate lives on to some extent in the machine learning community, where there are still arguments about whether learning is best performed using low-level, “neurallike” representations, or using more structured models.

In 1989, Yann Le Cun and others (LeCun et al. 1989) created the famous LeNet system described in Section 16.5.1.

In 1992, the support vector machine (see Section 14.5) was invented (Boser et al. 1992). SVMs provide similar prediction accuracy to neural networks while being considerably easier to train (since they use a convex objective function). This spawned a decade of interest in kernel methods in general.<sup>7</sup> Note, however, that SVMs do not use adaptive basis functions, so they require a fair amount of human expertise to design the right kernel function.

In 2002, Geoff Hinton invented the contrastive divergence training procedure (Hinton 2002), which provided a way, for the first time, to learn deep networks, by training one layer at a time in an unsupervised fashion (see Section 27.7.2.4 for details). This in turn has spawned renewed interest in neural networks over the last few years (see Chapter 28).

已思考 18s

下面按你的要求来：先做逐字无遗漏的中文翻译，再给出通俗解释，最后总结要点。

---

## 图注翻译

- 图 16.11：一层隐藏层的神经网络。
  - 图 16.12：来自 (Simard 等, 2003) 的卷积神经网络。
  - 图 16.13：对一个手写数字进行的几种合成形变。基于 (Bishop, 2006a) 图 5.14。
  - 图 16.14：LeNet5：用于手写数字分类的卷积神经网络。
  - 图 16.15：LeNet5 在 MNIST 的 10,000 个测试样本上产生的 82 个错误。每幅图下方的标签形如“正确标记 → 估计标记”。
- 

## 正文翻译

### 16.5 前馈神经网络（多层感知机）

前馈神经网络（亦称多层感知机，MLP）可以看作把若干个逻辑回归模型堆叠在一起；最后一层根据任务不同，要么还是一个逻辑回归（做分类），要么是线性回归（做回归）。

例如，对两层网络且是回归问题，模型形式为

$$p(y | x, \theta) = N(y | w^T z(x), \sigma^2) \quad (16.60)$$

$$z(x) = g(Vx) = [g(v_1^T x), \dots, g(v_H^T x)] \quad (16.61)$$

其中  $g$  是非线性激活/传递函数（常用 logistic 函数）， $z(x) = \phi(x, V)$  称为隐藏层（输入的确定性函数）， $H$  是隐藏单元数， $V$  是从输入到隐藏节点的权重矩阵， $w$  是从隐藏到输出的权重向量。重要的一点是  $g$  必须非线性，否则整个模型会退化为一个大的线性回归  $y = w^T(Vx)$ 。可以证明，MLP 是通用逼近器：给定足够多的隐藏单元，它能以任意精度逼近任何足够光滑的函数（Hornik 1991）。

处理二分类时，我们像 GLM 一样把输出通过 sigmoid：

$$p(y | x, \theta) = \text{Ber}(y | \text{sigm}(w^T z(x))) \quad (16.62)$$

我们可以很容易把 MLP 扩展到多输出。例如回归情形：

$$p(y | x, \theta) = N(y | W\phi(x, V), \sigma^2 I) \quad (16.63)$$

见图 16.11 的示意。如果在输出单元之间加入互抑制的连接，使得任一时刻只有一个单元“点亮”，我们就能强制一个“和为 1”的约束，这样可用于多类分类。得到的模型形式为

$$p(y | x, \theta) = \text{Cat}(y | S(Wz(x))) \quad (16.64)$$

（其中  $S$  表示把得分变为和为 1 的向量的变换，如 softmax。）

### 16.5.1 卷积神经网络

隐藏单元的目的是学习输入的非线性组合；这叫做特征提取/构造。这些隐藏特征随后作为最终 GLM 的输入。该方法在原始特征本身信息量不大的问题上尤其有用。例如图像中单个像素的信息很弱，像素的组合才告诉我们有什么物体。相反，在用词袋表示的文档分类中，每个特征（词频）本身就挺有信息，因此提取“高阶”特征的重要性相对小。不出所料，神经网络的大量工作都受视觉模式识别驱动（如 LeCun 等，1989），不过它们也被用于文本等其他数据（如 Collobert 与 Weston，2008）。

一种特别适合一维信号（语音或文本）或二维信号（图像）的 MLP 变体是卷积神经网络。这是这样一种 MLP：其隐藏单元具有局部感受野（类似初级视觉皮层），并且权重在空间上共享/绑定，以减少参数量。直观来说，这种空间参数共享的效应是：只要在图像某个局部“发现”了有用特征，就能在其它位置复用，而不必重新学习。由此网络呈现出平移不变性：无论模式出现在输入图像的何处都能分类。

图 16.12 给出了一个卷积网络的例子，由 Simard 及同事（2003）设计，共 5 层（其中 4 层含可调参数），用来对 MNIST（见 1.2.1.3 节）中的  $28 \times 28$  灰度手写数字图像分类。

- **第1层**: 有 6 个**特征图**, 每个大小  $13 \times 13$ 。每个隐藏节点由输入图像与一个  $5 \times 5$  的\*\*权重矩阵 (卷积核) \*\*进行卷积、加偏置、再经非线性得到。因此第 1 层共有  $13 \times 13 \times 6 = 1014$  个神经元, 权重数为  $(5 \times 5 + 1) \times 6 = 156$  ("+1"是偏置)。若不共享参数, 则第 1 层需要  $1014 \times 26 = 26,364$  个权重。
- **第2层**: 有 50 个特征图。其得到方式是: 将**第1层的每个特征图**分别与一个  $5 \times 5$  核卷积、再把这些结果**相加**、加偏置、过非线性。因此第 2 层有  $5 \times 5 \times 50 = 1250$  个神经元,  $(5 \times 5 + 1) \times 6 \times 50 = 7,800$  个可调权重 (层 1 与层 2 每对特征图对应一个核), 以及  $1250 \times 26 = 32,500$  条连接。
- **第3层**: 与第 2 层**全连接**, 有 100 个神经元和  $100 \times (1250 + 1) = 125,100$  个权重。
- **第4层**: 同样全连接, 有 10 个神经元和  $10 \times (100 + 1) = 1,010$  个权重。  
合计: 共有 3,215 个神经元、134,066 个可调权重、184,974 条连接。

该模型通常用**随机梯度下降训练** (细节见 16.5.4 节)。对数据的一次完整遍历称为一个 epoch。Mike O'Neill 在 2006 年做实验时发现, 单个 epoch 约需 40 分钟 (MNIST 有 60,000 个训练样本), 而错误率收敛约需 30 个 epoch, 总训练时间约 20 小时。用这种方法, 他在 10,000 个测试样本上得到约 1.40% 的误分类率。

为了进一步降错, 一个常见技巧是**扩充训练集**: 把原始数据做一些**形变**的拷贝, 从而鼓励网络对那些**不改变数字身份的小变化**保持不变性。可通过对像素施加**随机流场**来实现 (见图 16.13 示例)。若采用**在线/SGD 训练**, 这些形变可以**在线生成**, 无需保存。用此技巧, Mike O'Neill 在测试集上得到约 0.74% 的错误率, 接近当时的 SOTA。

Yann Le Cun 及其同事 (LeCun 等, 1998) 使用图 16.14 所示的、略更复杂的结构取得了类似性能。该模型称为 LeNet5, 按时间顺序其实先于图 16.12 的模型。两点主要差异:

1. **每个卷积层后有一次下采样** (subsampling), 在上层的小窗口内做**平均或最大值**, 以减小尺寸并获得一定的**平移不变性**。卷积 + 下采样的组合受 Hubel 与 Wiesel (1962) 关于视觉皮层**简单/复杂细胞**模型的启发, 并在神经启发的视觉识别模型中持续流行 (Riesenhuber 与 Poggio, 1999)。类似思想最早出现在 Fukushima 的 neocognitron (1975) 中, 但当时没有全局监督训练算法。
2. LeNet5 的**最终层是 RBF 网络**, 而不是更常见的 sigmoid 或 softmax 层。该模型在**不做形变训练**时测试错误率约 0.95%, **做形变时**约 0.8%。图 16.15 展示了系统犯下的所有 82 个错误: 有些确实**模糊歧义**, 但也有一些**人类不会犯**。LeNet5 的网页版演示在: <http://yann.lecun.com/exdb/lenet/index.html> ↗

当然, 识别**单个孤立数字**的适用性有限: 现实里人们常写数字或字母的串。这既需要**分割**也需要**分类**。Le Cun 及同事把卷积网络与一种类似**条件随机场** (见 19.6 节) 的模型结合

来解决该问题。该系统最终被**美国邮政**投入使用。（详见 LeCun 等, 1998；其仍是此任务表现最好的系统之一。）

### 16.5.2 其他类型的神经网络

除了上述结构，还存在其他拓扑。例如：可以有**跨层跳连**（直接从输入连到输出，跳过隐藏层）；可以让层间连接是**稀疏的**；等等。不过，MLP 总要求权重构成一个**有向无环图**。若允许**反馈连接**，模型称为**循环神经网络**（RNN）：这定义了一个**非线性动力系统**，但没有简单的概率解释。当前，RNN 是**语言建模**（即自然语言中的下一个词预测）的最佳方法（Tomas 等, 2011），显著优于 17.2.2 节讨论的标准 n-gram 方法。

若允许隐藏单元之间有**对称连接**，模型称为**Hopfield 网络/联想记忆**；其概率对应物为**玻尔兹曼机**（见 27.7 节），可用于**无监督学习**。

### 16.5.3 简史

由于人们既想理解**大脑**又想构建**学习机器**，神经网络几十年来一直备受关注。无法在此回顾全部历史，只列“剪影”：

- 1943 年，McCulloch 与 Pitts 提出了著名的**神经元数学模型**（McCullich & Pitts, 1943）：输出是输入加权和过一个阈值函数  $y = I(\sum_i w_i x_i > \theta)$ ，类似 sigmoid 激活。
- 1957 年，Frank Rosenblatt 发明了**感知机学习算法**（8.5.4 节）。非常相近的 adaline（自适应线性元）在 1960 年由 Widrow 与 Hoff 发明。
- 1969 年，Minsky 与 Papert 的名著《Perceptrons》指出：**无隐藏层的线性模型能力有限**，无法区分**非线性可分**的数据，这让该领域兴趣大减。
- 1986 年，Rumelhart、Hinton 与 Williams (1986) 让**反向传播算法**（见 16.5.4）家喻户晓，可拟合**有隐藏层**的模型。（该算法最早见于 Bryson & Ho, 1969；以及 Werbos, 1974；但 1986 年的工作让它被广泛关注。）由此掀起了十年的热潮。
- 1987 年，Sejnowski 与 Rosenberg 的 NETtalk 学会把英语单词映到**音素**，可送入语音合成器。系统从“咿呀学语”到逐渐会读（音频演示见 <http://www.cnl.salk.edu/ParallelNetsPronounce/nettalk.mp3>）。NETtalk 通过隐藏层学到**分布式表示**，在心理学中引发了**连接主义 vs. 计算主义**的争论；在机器学习里，这种争论部分延续为“底层神经式表示 vs. 结构化模型”。
- 1989 年，Yann Le Cun 等创建了本节 16.5.1 的 LeNet 系统。
- 1992 年，**支持向量机**（14.5 节）被提出（Boser 等, 1992）。SVM 在预测精度上与神经网络相近，但训练更容易（目标函数凸），从而引发了长达十年的**核方法**热潮。注意 SVM 不使用**自适应基函数**，因此需要人类投入设计合适的核函数。
- 2002 年，Geoff Hinton 发明了**对比散度训练**（Hinton, 2002），首次给出了**逐层无监督地学习深层网络**的方法（细节见 27.7.2.4），从而在随后的几年重新点燃了对神经网

络的兴趣（见第 28 章）。

---

## 通俗解释

- **MLP 是什么？**

就像把多次“逻辑回归/线性回归”**层层叠起来**：前面的层把原始特征“拧成”新的**非线性特征**，最后一层做常规回归/分类。只要隐藏层够多/够宽，理论上能逼近任意平滑函数。

- **为什么需要非线性  $g$ ？**

如果  $g$  是线性的，所有层叠加仍等价于**一层线性**，就没意义了。非线性让模型会“弯”。

- **CNN 的两件法宝**

1. **局部感受野**：只看小窗口，像人眼视皮层。

2. **权重共享**：同一组卷积核在整张图上“扫”，参数量大减，也带来**平移不变性**（猫在左上/右下都能认）。

再加上**下采样/池化**（平均或最大），既减尺寸又略带不变性。

- **为什么数据增强有用？**

人写“6”姿势多样，但**类别不变**。在训练时做随机形变，相当于告诉模型“这些变化都不应影响标签”，泛化自然更好。

- **LeNet5 与 Simard 架构的区别**

LeNet5 每个卷积层后都**下采样**；最终层用的是**RBF**（而不是 softmax）。两者在 MNIST 上都达到 1% 左右甚至更低的错误。

- **其它网络**

- **RNN**：允许回路，适合序列（语言建模表现突出）。

- **Hopfield / 玻尔兹曼机**：允许对称连接，能做无监督学习与记忆。

- **历史脉络**

感知机 → 发现线性模型局限 → 反向传播带来多层革命 → SVM 时代 → 对比散度与深度学习再兴。

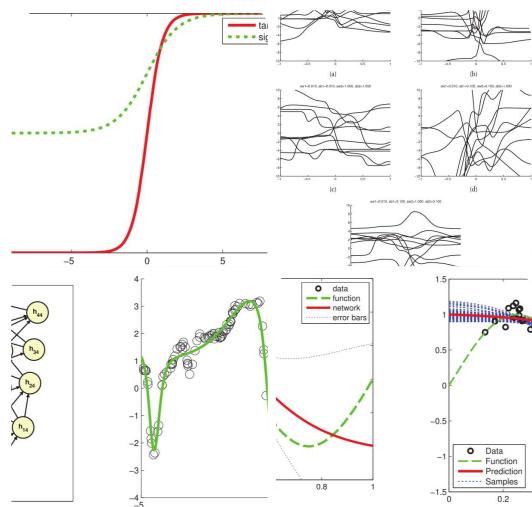
---

## 要点总结

- **MLP = 多层非线性变换 + 最后一层 GLM**；是**通用逼近器**。
- **分类**：输出经 sigmoid/softmax；**回归**：线性输出、高斯噪声。
- **CNN 通过局部连接 + 权重共享显著降参并带来平移不变性**；常配**池化**。

- **MNIST 实证**: 卷积网络+SGD 可达 ~1% 以内错误；**数据增强**进一步降低到 ~0.7–0.8%。
- **LeNet5**: 卷积/池化交替，末层 RBF；在当时属于 SOTA，并被邮政系统部署。
- **其他结构**: RNN (序列最强)，Hopfield/玻尔兹曼机 (无监督/联想记忆)。
- **历史线索**: 感知机 → 反向传播 → LeNet → SVM → 对比散度/深度学习复兴。

如果你希望，我可以把这些公式变成更直观的小例子或简单数值演算来帮你加深理解。



翻译并解释以下内容，翻译不能有遗漏，解释请通俗易懂且详尽，因为我的数学基础不太好，最后总结要点。

Figure 16.16 Two possible activation functions.  $\tanh$  maps  $\mathbb{R}$  to  $[-1, +1]$  and is the preferred nonlinearity for the hidden nodes.  $\text{sigm}$  maps  $\mathbb{R}$  to  $[0, 1]$  and is the preferred nonlinearity for binary nodes at the output layer.

Figure 16.17 The effects of changing the hyperparameters on an MLP. (a) Default parameter values  $\alpha_v = 0.01$ ,  $\alpha_b = 0.1$ ,  $\alpha_w = 1$ ,  $\alpha_c = 1$ . (b) Decreasing  $\alpha_v$  by factor of 10. (c) Decreasing  $\alpha_b$  by factor of 10. (d) Decreasing  $\alpha_w$  by factor of 10. (e) Decreasing  $\alpha_c$  by factor of 10.

Figure 16.18 (a) A deep but sparse neural network. The connections are pruned using 1 regularization. At each level, nodes numbered 0 are clamped to 1, so their outgoing weights correspond to the offset/bias terms. (b) Predictions made by the model on the training set.

Figure 16.19 The posterior predictive density for an MLP with 3 hidden nodes, trained on 16 data points. The dashed green line is the true function. (a) Result of using a

Laplace approximation, after performing empirical Bayes to optimize the hyperparameters. The solid red line is the posterior mean prediction, and the dotted blue lines are 1 standard deviation above and below the mean. Figure generated by `mlpRegEvidenceDemo`. (b) Result of using hybrid Monte Carlo, using the same trained hyperparameters as in (a). The solid red line is the posterior mean prediction, and the dotted blue lines are samples from the posterior predictive.

#### 16.5.4 The backpropagation algorithm

Unlike a GLM, the NLL of an MLP is a non-convex function of its parameters. Nevertheless, we can find a locally optimal ML or MAP estimate using standard gradient-based optimization methods. Since MLPs have lots of parameters, they are often trained on very large data sets. Consequently it is common to use first-order online methods, such as stochastic gradient descent (Section 8.5.2), whereas GLMs are usually fit with IRLS, which is a second-order offline method. We now discuss how to compute the gradient vector of the NLL by applying the chain rule of calculus. The resulting algorithm is known as backpropagation, for reasons that will become apparent. For notational simplicity, we shall assume a model with just one hidden layer. It is helpful to distinguish the pre- and post-synaptic values of a neuron, that is, before and after we apply the nonlinearity. Let  $x_n$  be the  $n$ 'th input,  $a_n = Vx_n$  be the pre-synaptic hidden layer, and  $z_n = g(a_n)$  be the post-synaptic hidden layer, where  $g$  is some transfer function. We typically use  $g(a) = \text{sigm}(a)$ , but we may also use  $g(a) = \tanh(a)$ : see Figure 16.16 for a comparison. (When the input to  $\text{sigm}$  or  $\tanh$  is a vector, we assume it is applied component-wise.) We now convert this hidden layer to the output layer as follows. Let  $b_n = Wz_n$  be the pre-synaptic output layer, and  $\hat{y}_n = h(b_n)$  be the post-synaptic output layer, where  $h$  is another nonlinearity, corresponding to the canonical link for the GLM. (We reserve the notation  $y_n$ , without the hat, for the output corresponding to the  $n$ 'th training case.) For a regression model, we use  $h(b) = b$ ; for binary classification, we use  $h(b) = [\text{sigm}(b_1), \dots, \text{sigm}(b_c)]$ ; for multi-class classification, we use  $h(b) = S(b)$ . We can write the overall model as follows:

$x_n V \rightarrow a_n g \rightarrow z_n W \rightarrow b_n h \rightarrow \hat{y}_n$  (16.65)

The parameters of the model are  $\theta = (V, W)$ , the first and second layer weight matrices. Offset or bias terms can be accommodated by clamping an element of  $x_n$  and  $z_n$  to 1. In the regression case, with  $K$  outputs, the NLL is given by the squared error:

$$J(\theta) = -\frac{1}{n} \sum_k (\hat{y}_{nk}(\theta) - y_{nk})^2 \quad (16.66)$$

In the classification case, with  $K$  classes, the NLL is given by the cross entropy

$$J(\theta) = -\frac{1}{n} \sum_k y_{nk} \log \hat{y}_{nk}(\theta) \quad (16.67)$$

Our task is to compute  $\nabla_{\theta} J$ . We will derive this for each  $n$  separately; the overall gradient is obtained by summing over  $n$ , although often we just use a mini-batch (see Section 8.5.2). Let us start by considering the output layer weights. We have

$$\nabla_{Wk} J_n = \partial J_n / \partial b_{nk} \quad \nabla_{Wkb_{nk}} = \partial J_n / \partial b_{nk} z_n \quad (16.68)$$

since  $b_{nk} = w^T k z_n$ . Assuming  $h$  is the canonical link function for the output GLM, then Equation 9.91 tells us that

$$\partial J_n / \partial b_{nk} = \delta_{Wk} = (\hat{y}_{nk} - y_{nk}) \quad (16.69)$$

which is the error signal. So the overall gradient is

$$\nabla_{Wk} J_n = \delta_{Wk} z_n \quad (16.70)$$

which is the pre-synaptic input to the output layer, namely  $z_n$ , times the error signal, namely  $\delta_{Wk}$ . For the input layer weights, we have

$$\nabla_{Vj} J_n = \partial J_n / \partial a_{nj} \quad \nabla_{Vj} a_{nj} = \delta_{Vj} n_j x_n \quad (16.71)$$

where we exploited the fact that  $a_{nj} = v^T j x_n$ . All that remains is to compute the first level error signal  $\delta_{Vj}$ . We have

$$\delta_{Vj} = \partial J_n / \partial a_{nj} = \sum_k \partial J_n / \partial b_{nk} \partial b_{nk} / \partial a_{nj} = \sum_k \delta_{Wk} z_n \partial b_{nk} / \partial a_{nj} \quad (16.72)$$

Now

$$\partial b_{nk} = \sum_j w_{kj} g(a_{nj}) \quad (16.73)$$

so

$$\partial b_{nk} / \partial a_{nj} = w_{kj} g'(a_{nj}) \quad (16.74)$$

where  $g'(a) = d/dx g(a)$ . For tanh units,  $g'(a) = d/dx \tanh(a) = 1 - \tanh^2(a) = \text{sech}^2(a)$ , and for sigmoid units,  $g'(a) = d/dx \sigma(a) = \sigma(a)(1 - \sigma(a))$ . Hence

$$\delta_{Vj} = \sum_k \delta_{Wk} z_n w_{kj} g'(a_{nj}) \quad (16.75)$$

Thus the layer 1 errors can be computed by passing the layer 2 errors back through the  $W$  matrix; hence the term “backpropagation”. The key property is that we can compute the gradients locally: each node only needs to

know about its immediate neighbors. This is supposed to make the algorithm “neurally plausible”, although this interpretation is somewhat controversial. Putting it all together, we can compute all the gradients as follows: we first perform a forwards pass to compute  $a_n$ ,  $z_n$ ,  $b_n$  and  $\hat{y}_n$ . We then compute the error for the output layer,  $\delta(2)$   $n = \hat{y}_n - y_n$ , which we pass backwards through  $W$  using Equation 16.75 to compute the error for the hidden layer,  $\delta(1) n$ . We then compute the overall gradient as follows:

$$\nabla \theta J(\theta) = n [\delta v n x_n, \delta w n z_n] \quad (16.76)$$

### 16.5.5 Identifiability

It is easy to see that the parameters of a neural network are not identifiable. For example, we can change the sign of the weights going into one of the hidden units, so long as we change the sign of all the weights going out of it; these effects cancel, since  $\tanh$  is an odd function, so  $\tanh(-a) = -\tanh(a)$ . There will be  $H$  such sign flip symmetries, leading to  $2H$  equivalent settings of the parameters. Similarly, we can change the identity of the hidden units without affecting the likelihood. There are  $H!$  such permutations. The total number of equivalent parameter settings (with the same likelihood) is therefore  $H!2^H$ . In addition, there may be local minima due to the non-convexity of the NLL. This can be a more serious problem, although with enough data, these local optima are often quite “shallow”, and simple stochastic optimization methods can avoid them. In addition, it is common to perform multiple restarts, and to pick the best solution, or to average over the resulting predictions. (It does not make sense to average the parameters themselves, since they are not identifiable.)

### 16.5.6 Regularization

As usual, the MLE can overfit, especially if the number of nodes is large. A simple way to prevent this is called early stopping, which means stopping the training procedure when the error on the validation set first starts to increase. This method works because we usually initialize from small random weights, so the model is initially simple (since the  $\tanh$  and  $\text{sigm}$  functions are nearly linear near the origin). As training progresses, the weights become larger, and the model becomes nonlinear. Eventually it will overfit. Another way to prevent overfitting, that is more in keeping with the approaches used elsewhere in this book,

is to impose a prior on the parameters, and then use MAP estimation. It is standard to use a  $N(0, \alpha^{-1}I)$  prior (equivalent to  $L_2$  regularization), where  $\alpha$  is the precision (strength) of the prior. In the neural networks literature, this is called weight decay, since it encourages small weights, and hence simpler models. The penalized NLL objective becomes

$$J(\theta) = -\sum_{n=1}^N \log p(y_n|x_n, \theta) + \frac{\alpha}{2} [\sum_{ij} v_i^2 + \sum_{jk} w_j^2]$$

(16.77)

(Note that we don't penalize the bias terms.) The gradient of the modified objective becomes

$$\nabla \theta J(\theta) = [n \delta v n x n + \alpha v, n \delta w n z n + \alpha w] \quad (16.78)$$

as in Section 8.3.6. If the regularization is sufficiently strong, it does not matter if we have too many hidden units (apart from wasted computation). Hence it is advisable to set  $H$  to be as large as you can afford (say 10–100), and then to choose an appropriate regularizer. We can set the  $\alpha$  parameter by cross validation or empirical Bayes (see Section 16.5.7.5). As with ridge regression, it is good practice to standardize the inputs to zero mean and unit variance, so that the spherical Gaussian prior makes sense.

#### 16.5.6.1 Consistent Gaussian priors

One can show (MacKay 1992) that using the same regularization parameter for both the first and second layer weights results in the lack of a certain desirable invariance property. In particular, suppose we linearly scale and shift the inputs and/or outputs to a neural network regression model. Then we would like the model to learn to predict the same function, by suitably scaling its internal weights and bias terms. However, the amount of scaling needed by the first and second layer weights to compensate for a change in the inputs and/or outputs is not the same. Therefore we need to use a different regularization strength for the first and second layer.

Fortunately, this is easy to do — we just use the following prior:

$$p(\theta) = N(W|0, 1/\alpha I)N(V|0, 1/\alpha I)N(b|0, 1/\alpha I)N(c|0, 1/\alpha I) \quad (16.79)$$

where  $b$  and  $c$  are the bias terms. To get a feeling for the effect of these hyper-parameters, we can sample MLP parameters from this prior and plot the resulting random functions. Figure 16.17 shows some examples. Decreasing

$\alpha_v$  allows the first layer weights to get bigger, making the sigmoid-like shape of the functions steeper. Decreasing  $\alpha_b$  allows the first layer biases to get bigger, which allows the center of the sigmoid to shift left and right more. Decreasing  $\alpha_w$  allows the second layer weights to get bigger, making the functions more “wiggly” (greater sensitivity to change in the input, and hence larger dynamic range). And decreasing  $\alpha_c$  allows the second layer biases to get bigger, allowing the mean level of the function to move up and down more. (In Chapter 15, we will see an easier way to define priors over functions.)

#### 16.5.6.2 Weight pruning

Since there are many weights in a neural network, it is often helpful to encourage sparsity. Various ad-hoc methods for doing this, with names such as “optimal brain damage”, were devised in the 1990s; see e.g., (Bishop 1995) for details.

However, we can also use the more principled sparsity-promoting techniques we discussed in Chapter 13. One approach is to use an  $L_1$  regularizer. See Figure 16.18 for an example. Another approach is to use ARD; this is discussed in more detail in Section 16.5.7.5.

#### 16.5.6.3 Soft weight sharing

Another way to regularize the parameters is to encourage similar weights to share statistical strength. But how do we know which parameters to group together? We can learn this, by using a mixture model. That is, we model  $p(\theta)$  as a mixture of (diagonal) Gaussians. Parameters that are assigned to the same cluster will share the same mean and variance and thus will have similar values (assuming the variance for that cluster is low). This is called soft weight sharing (Nowlan and Hinton 1992). In practice, this technique is not widely used. See e.g., (Bishop 2006a, p271) if you want to know the details.

#### 16.5.6.4 Semi-supervised embedding

An interesting way to regularize “deep” feedforward neural networks is to encourage the hidden layers to assign similar objects to similar representations. This is useful because it is often easy to obtain “side” information consisting of sets of pairs of similar and dissimilar objects. For example, in a video classification task, neighboring frames can be deemed similar, but frames that are distant in time can be deemed dis-similar (Mobahi et al. 2009).

Note that this can be done without collecting any labels. Let  $S_{ij} = 1$  if examples  $i$  and  $j$  are similar, and  $S_{ij} = 0$  otherwise. Let  $f(x_i)$  be some embedding of item  $x_i$ , e.g.,  $f(x_i) = z(x_i, \theta)$ , where  $z$  is the hidden layer of a neural network. Now define a loss function  $L(f(x_i), f(x_j), S_{ij})$  that depends on the embedding of two objects, and the observed similarity measure. For example, we might want to force similar objects to have similar embeddings, and to force the embeddings of dissimilar objects to be a minimal distance apart:

$$L(f_i, f_j, S_{ij}) = \|f_i - f_j\|^2 \text{ if } S_{ij} = 1 \quad \max(0, m - \|f_i - f_j\|^2) \text{ if } S_{ij} = 0 \quad (16.80)$$

where  $m$  is some minimal margin. We can now define an augmented loss function for training the neural network:  $\sum_{i \in L} NLL(f(x_i), y_i) + \lambda \sum_{i,j \in U} L(f(x_i), f(x_j), S_{ij}) \quad (16.81)$

where  $L$  is the labeled training set,  $U$  is the unlabeled training set, and  $\lambda \geq 0$  is some tradeoff parameter. This is called semi-supervised embedding (Weston et al. 2008). Such an objective can be easily optimized by stochastic gradient descent. At each iteration, pick a random labeled training example,  $(x_n, y_n)$ , and take a gradient step to optimize  $NLL(f(x_n), y_n)$ . Then pick a random pair of similar unlabeled examples  $x_i, x_j$  (these can sometimes be generated on the fly rather than stored in advance), and make a gradient step to optimize  $\lambda L(f(x_i), f(x_j), 1)$ . Finally, pick a random unlabeled example  $x_k$ , which with high probability is dissimilar to  $x_i$ , and make a gradient step to optimize  $\lambda L(f(x_i), f(x_k), 0)$ . Note that this technique is effective because it can leverage massive amounts of data. In a related approach, (Collobert and Weston 2008) trained a neural network to distinguish valid English sentences from invalid ones. This was done by taking all 631 million words from English Wikipedia ([en.wikipedia.org](http://en.wikipedia.org)), and then creating windows of length 11 containing neighboring words. This constitutes the positive examples. To create negative examples, the middle word of each window was replaced by a random English word (this is likely to be an “invalid” sentence — either grammatically and/or semantically — with high probability). This neural network was then trained over the course of 1 week, and its latent representation was then used as the input to a supervised semantic role labeling task, for which very little labeled training data is available.

(See also (Ando and Zhang 2005) for related work.)

### 16.5.7 Bayesian inference

Although MAP estimation is a successful way to reduce overfitting, there are still some good reasons to want to adopt a fully Bayesian approach to “fitting” neural networks:

- Integrating out the parameters instead of optimizing them is a much stronger form of regularization than MAP estimation.
- We can use Bayesian model selection to determine things like the hyper-parameter settings and the number of hidden units. This is likely to be much faster than cross validation, especially if we have many hyper-parameters (e.g., as in ARD).
- Modelling uncertainty in the parameters will induce uncertainty in our predictive distributions, which is important for certain problems such as active learning and risk-averse decision making.
- We can use online inference methods, such as the extended Kalman filter, to do online learning (Haykin 2001).

One can adopt a variety of approximate Bayesian inference techniques in this context. In this section, we discuss the Laplace approximation, first suggested in (MacKay 1992, 1995b). One can also use hybrid Monte Carlo (Neal 1996), or variational Bayes (Hinton and Camp 1993; Barber and Bishop 1998).

#### 16.5.7.1 Parameter posterior for regression

We start by considering regression, following the presentation of (Bishop 2006a, sec 5.7), which summarizes the work of (MacKay 1992, 1995b). We will use a prior of the form  $p(w) = N(w|0, (1/\alpha)I)$ , where  $w$  represents all the weights combined. We will denote the precision of the noise by  $\beta = 1/\sigma^2$ . The posterior can be approximated as follows:

$$p(w|D, \alpha, \beta) \propto \exp(-E(w)) \quad (16.82)$$

$$E(w) = \beta ED(w) + \alpha EW(w) \quad (16.83)$$

$$ED(w) = \frac{1}{2} \sum_{n=1}^N (y_n - f(x_n, w))^2 \quad (16.84)$$

$$EW(w) = \frac{1}{2} w^T w \quad (16.85)$$

where  $ED$  is the data error,  $EW$  is the prior error, and  $E$  is the overall error (negative log prior plus log likelihood).

Now let us make a second-order Taylor series approximation of  $E(w)$  around its minimum (the MAP

estimate)

$$E(w) \approx E(w_{MP}) + \frac{1}{2} (w - w_{MP})^T A (w - w_{MP}) \quad (16.86)$$

where  $A$  is the Hessian of  $E$ :

$$A = \nabla \nabla E(w_{MP}) = \beta H + \alpha I \quad (16.87)$$

where  $H = \nabla \nabla ED(w_{MP})$  is the Hessian of the data error.

This can be computed exactly in  $O(d^2)$  time, where  $d$  is the number of parameters, using a variant of backpropagation (see (Bishop 2006a, sec 5.4) for details). Alternatively, if we use a quasi-Newton method to find the mode, we can use its internally computed (low-rank) approximation to  $H$ . (Note that diagonal approximations of  $H$  are usually very inaccurate.) In either case, using this quadratic approximation, the posterior becomes Gaussian:

$$p(w|\alpha, \beta, D) \approx N(w|w_{MP}, A^{-1}) \quad (16.88)$$

#### 16.5.7.2 Parameter posterior for classification

The classification case is the same as the regression case, except  $\beta = 1$  and  $ED$  is a crossentropy error of the form

$$ED(w) = -\sum_{n=1}^N [y_n \ln f(x_n, w) + (1 - y_n) \ln (1 - f(x_n, w))] \quad (16.89)$$

(16.90)

#### 16.5.7.3 Predictive posterior for regression

The posterior predictive density is given by

$$p(y|x, D, \alpha, \beta) = N(y|f(x, w), 1/\beta)N(w|w_{MP}, A^{-1})dw \quad (16.91)$$

This is not analytically tractable because of the nonlinearity of  $f(x, w)$ . Let us therefore construct a first-order Taylor series approximation around the mode:

$$f(x, w) \approx f(x, w_{MP}) + g^T(w - w_{MP}) \quad (16.92)$$

where

$$g = \nabla w f(x, w)|_{w=w_{MP}} \quad (16.93)$$

We now have a linear-Gaussian model with a Gaussian prior on the weights. From Equation 4.126 we have

$$p(y|x, D, \alpha, \beta) \approx N(y|f(x, w_{MP}), \sigma^2(x)) \quad (16.94)$$

where the predictive variance depends on the input  $x$  as follows:

$$\sigma^2(x) = \beta^{-1} + g^T A^{-1} g \quad (16.95)$$

The error bars will be larger in regions of input space where we have little training data. See Figure 16.19 for an example.

#### 16.5.7.4 Predictive posterior for classification

In this section, we discuss how to approximate  $p(y|x, D)$  in the case of binary classification. The situation is similar to the case of logistic regression, discussed in Section 8.4.4, except in addition the posterior predictive mean is a non-linear function of  $w$ . Specifically, we have  $\mu = E$

$[y|x,w] = \text{sigm}(a(x,w))$ , where  $a(x,w)$  is the pre-synaptic output of the final layer. Let us make a linear approximation to this:

$$a(x,w) \approx aMP(x) + gT(w - wMP) \quad (16.96)$$

where  $aMP(x) = a(x,wMP)$  and  $g = \nabla_x a(x,wMP)$  can be found by a modified version of backpropagation. Clearly

$$p(a|x, D) \approx N(a(x,wMP), g(x)TA - 1g(x)) \quad (16.97)$$

Hence the posterior predictive for the output is

$$p(y=1|x, D) = \text{sigm}(a)p(a|x, D)da \approx \text{sigm}(\kappa(\sigma^2 a)bTwMP) \quad (16.98)$$

where  $\kappa$  is defined by Equation 8.70, which we repeat here for convenience:

$$\kappa(\sigma^2) = (1 + \pi\sigma^2/8)^{-1/2} \quad (16.99)$$

Of course, a simpler (and potentially more accurate) alternative to this is to draw a few samples from the Gaussian posterior and to approximate the posterior predictive using Monte Carlo. In either case, the effect of taking uncertainty of the parameters into account, as in Section 8.4.4, is to “moderate” the confidence of the output; the decision boundary itself is unaffected, however.

#### 16.5.7.5 ARD for neural networks

Once we have made the Laplace approximation to the posterior, we can optimize the marginal likelihood wrt the hyper-parameters  $\alpha$  using the same fixed-point equations as in Section 13.7.4.2. Typically we use one hyper-parameter for the weight vector leaving each node, to achieve an effect similar to group lasso (Section 13.5.1).

That is, the prior has the form

$$p(\theta) = \prod_{i=1}^D N(v_{:,i} | 0, 1/\alpha_{v,i}) \prod_{j=1}^H N(w_{:,j} | 0, 1/\alpha_{w,j}) \quad (16.100)$$

If we find  $\alpha_{v,i} = \infty$ , then input feature  $i$  is irrelevant, and its weight vector  $v_{:,i}$  is pruned out. Similarly, if we find  $\alpha_{w,j} = \infty$ , then hidden feature  $j$  is irrelevant. This is known as automatic relevancy determination or ARD, which was discussed in detail in Section 13.7. Applying this to neural networks gives us an efficient means of variable selection in non-linear models. The software package NETLAB contains a simple example of ARD applied to a neural network, called demand. This demo creates some data according to a nonlinear regression function  $f(x_1, x_2, x_3) = \sin(2\pi x_1) + \epsilon$ , where  $x_2$  is a noisy copy of  $x_1$ . We see that  $x_2$  and  $x_3$  are irrelevant for predicting the target.

However,  $x_2$  is correlated with  $x_1$ , which is relevant. Using ARD, the final hyper-parameters are as follows:  
 $\alpha = [0.2, 21.4, 249001.8]$  (16.101)  
This clearly indicates that feature 3 is irrelevant, feature 2 is only weakly relevant, and feature 1 is very relevant.

下面按你的要求来：先逐字不遗漏翻译（含公式/图注），再做通俗解释，最后给要点总结。

---

## 图注翻译

- **图16.16：**两种可能的激活函数。 $\tanh$  把实数域  $R$  映到区间  $[-1, +1]$ ，是隐藏节点的首选非线性； $\text{sigm}$  把  $R$  映到  $[0, 1]$ ，是输出层二值节点的首选非线性。
  - **图16.17：**改变超参数对 MLP 的影响。(a) 默认参数  $\alpha_v = 0.01$ ,  $\alpha_b = 0.1$ ,  $\alpha_w = 1$ ,  $\alpha_c = 1$ 。(b) 将  $\alpha_v$  降低 10 倍。(c) 将  $\alpha_b$  降低 10 倍。(d) 将  $\alpha_w$  降低 10 倍。(e) 将  $\alpha_c$  降低 10 倍。
  - **图16.18：**(a) 一个深而稀疏的神经网络。连接通过  $L_1$  正则被剪枝。每一层中编号为 0 的节点被钳制为 1，因此它们的输出权重对应偏置/截距项。(b) 该模型在训练集上的预测。
  - **图16.19：**在 16 个数据点上训练、含 3 个隐藏节点的 MLP 的后验预测密度。虚线绿为真实函数。(a) 先做经验贝叶斯优化超参数后，再用拉普拉斯近似的结果。红实线为后验均值预测，蓝点虚线为均值上下 1 个标准差。由 `m1pRegEvidenceDemo` 生成。(b) 使用混合蒙特卡洛 (HMC)，并采用与 (a) 同一组已训练超参数的结果。红实线为后验均值预测，蓝点虚线为后验预测的采样。
- 

### 16.5.4 反向传播算法（翻译）

不同于 GLM, MLP 的负对数似然 (NLL) 关于其参数是**非凸**函数。尽管如此，我们仍可用标准的基于梯度的优化法找到**局部**的 ML 或 MAP 估计。因为 MLP 的参数很多，且常在非常大的数据集上训练，因此通常使用**一阶在线方法**，如随机梯度下降（第8.5.2节），而 GLM 通常用二阶的离线方法 IRLS。下面我们用微积分的链式法则来计算 NLL 的梯度向量。得到的算法称为**反向传播**（原因随后可见）。为简化记号，假设模型只有一层隐藏层。把神经元在施加非线性**之前与之后**的值区分开来是有帮助的。令  $x_n$  为第  $n$  个输入， $a_n = Vx_n$  为隐藏层的“突触前 (pre-synaptic)”值， $z_n = g(a_n)$  为“突触后 (post-synaptic)”隐藏层，其中  $g$  是某个传递函数。我们通常用  $g(a) = \text{sigm}(a)$ ，也可以用  $g(a) = \tanh(a)$ ：比较见图16.16。（当  $\text{sigm}$  或  $\tanh$  的输入是向量时，默认逐分量作

用。) 现在把该隐藏层变成输出层: 令  $b_n = Wz_n$  为输出层的突触前值,  $\hat{y}_n = h(b_n)$  为突触后输出层, 其中  $h$  是另一个非线性, 对应于 GLM 的典则链接 (canonical link)。(不带帽号的  $y_n$  表示第  $n$  个训练样本的目标。) 回归模型用  $h(b) = b$ ; 二分类用  $h(b) = [\text{sigm}(b_1), \dots, \text{sigm}(b_C)]$ ; 多类分类用  $h(b) = S(b)$ 。整个模型可写成

$$x_n \xrightarrow{V} a_n \xrightarrow{g} z_n \xrightarrow{W} b_n \xrightarrow{h} \hat{y}_n \quad (16.65)$$

模型参数为  $\theta = (V, W)$ , 即第一层与第二层的权重矩阵。可通过把  $x_n$  与  $z_n$  的某个分量钳为 1 来容纳偏置项。

回归情形、 $K$  个输出时, NLL 为平方误差:

$$J(\theta) = -\sum_n \sum_k (\hat{y}_{nk}(\theta) - y_{nk})^2 \quad (16.66)$$

分类情形、 $K$  个类别时, NLL 为交叉熵:

$$J(\theta) = -\sum_n \sum_k y_{nk} \log \hat{y}_{nk}(\theta) \quad (16.67)$$

任务是计算  $\nabla_\theta J$ 。对每个  $n$  单独推导; 总体梯度对  $n$  求和 (常用小批量)。从输出层权重开始:

$$\nabla_{w_k} J_n = \frac{\partial J_n}{\partial b_{nk}} \nabla_{w_k} b_{nk} = \frac{\partial J_n}{\partial b_{nk}} z_n \quad (16.68)$$

因为  $b_{nk} = w_k^\top z_n$ 。若  $h$  为输出 GLM 的典则链接, 由式(9.91)可得

$$\frac{\partial J_n}{\partial b_{nk}} \equiv \delta_{nk}^w = (\hat{y}_{nk} - y_{nk}) \quad (16.69)$$

这是**误差信号**。因此

$$\nabla_{w_k} J_n = \delta_{nk}^w z_n \quad (16.70)$$

即输出层的**突触前输入**  $z_n$  乘误差信号  $\delta_{nk}^w$ 。

输入层权重:

$$\nabla_{v_j} J_n = \frac{\partial J_n}{\partial a_{nj}} \nabla_{v_j} a_{nj} = \delta_{nj}^v x_n \quad (16.71)$$

其中用到  $a_{nj} = v_j^\top x_n$ 。剩下只需计算第 1 层的误差信号  $\delta_{nj}^v$ :

$$\delta_{nj}^v = \frac{\partial J_n}{\partial a_{nj}} = \sum_{k=1}^K \frac{\partial J_n}{\partial b_{nk}} \frac{\partial b_{nk}}{\partial a_{nj}} = \sum_{k=1}^K \delta_{nk}^w \frac{\partial b_{nk}}{\partial a_{nj}} \quad (16.72)$$

而

$$b_{nk} = \sum_j w_{kj} g(a_{nj}) \quad (16.73)$$

所以

$$\frac{\partial b_{nk}}{\partial a_{nj}} = w_{kj} g'(a_{nj}) \quad (16.74)$$

其中  $g'(a) = \frac{d}{da}g(a)$ 。tanh 单元  $g'(a) = 1 - \tanh^2(a) = \operatorname{sech}^2(a)$ ; sigmoid 单元  $g'(a) = \sigma(a)(1 - \sigma(a))$ 。因此

$$\delta_{nj}^v = \sum_{k=1}^K \delta_{nk}^w w_{kj} g'(a_{nj}) \quad (16.75)$$

于是第 2 层的误差可通过矩阵  $W$  反传到第 1 层；因此称为“反向传播”。关键性质：梯度可局部计算——每个节点只需其相邻节点的信息。这使算法被认为“神经可行”，尽管这种解释颇有争议。综合起来，先前向计算  $a_n, z_n, b_n, \hat{y}_n$ 。然后计算输出层误差  $\delta_n^{(2)} = \hat{y}_n - y_n$ ，再用式(16.75)把误差通过  $W$  反传得到隐藏层误差  $\delta_n^{(1)}$ 。最终梯度为

$$\nabla_{\theta} J(\theta) = \sum_n [\delta_n^v x_n, \delta_n^w z_n] \quad (16.76)$$

## 16.5.5 可辨识性（翻译）

很容易看出神经网络的参数**不可辨识**。例如，只要把某个隐藏单元**输入**的权重整体变号，同时把其**输出**的权重也整体变号，效应会相互抵消，因为 tanh 是奇函数： $\tanh(-a) = -\tanh(a)$ 。由此产生  $H$  个符号翻转对称性，导致  $2^H$  组等价参数。同样，**置换**隐藏单元的身份也不影响似然，共有  $H!$  种置换。因此具有相同似然的等价参数设置共有  $H! 2^H$  种。此外，由于 NLL 非凸，可能存在**局部极小值**——这通常更麻烦；不过只要数据足够，这些局部极小通常较“浅”，简单的随机优化方法可以避开。还常做多次随机重启，选取最佳解，或对**预测**做平均。（直接平均参数没意义，因为参数不可辨识。）

## 16.5.6 正则化（翻译）

照例，MLE 会过拟合，尤其当节点很多时。一个简单的防止过拟合方法是**早停**：当验证集上的误差开始上升时停止训练。之所以有效，是因为我们通常从**小随机权重**初始化，起初模型很简单（tanh 与 sigm 在原点附近近似线性）；训练推进后权重变大，模型变得非线性，最终会过拟合。另一种更契合本书其他部分的方法是对参数施加先验，用 MAP 估

计。标准先验是  $N(0, \alpha^{-1}I)$  (等价于  $L_2$  正则) ,  $\alpha$  为先验精度 (强度) 。在神经网络文献中这叫**权重衰减**, 鼓励小权重、从而更简单的模型。带惩罚的 NLL:

$$J(\theta) = -\sum_{n=1}^N \log p(y_n|x_n, \theta) + \frac{\alpha}{2} [\sum_{ij} v_{ij}^2 + \sum_{jk} w_{jk}^2] \quad (16.77)$$

(不惩罚偏置。) 相应梯度为

$$\nabla_{\theta} J(\theta) = [ \sum_n \delta_n^v x_n + \alpha v, \sum_n \delta_n^w z_n + \alpha w ] \quad (16.78)$$

若正则足够强, 即使隐藏单元太多也无妨 (除了计算浪费) 。因此建议把  $H$  设为力所能及的较大值 (如 10–100), 再选合适正则。可用交叉验证或经验贝叶斯 (见16.5.7.5) 设置  $\alpha$ 。与岭回归一样, 最好把输入**标准化为零均值单位方差**, 使球形高斯先验合理。

### 16.5.6.1 一致的高斯先验

可证明 (MacKay 1992) : 对第一层与第二层权重使用**同一个**正则强度会失去某种理想的不变性。具体说, 若我们对回归网络的输入/输出做线性缩放和平移, 希望模型能通过适当缩放内部权重/偏置学到**同一个函数**。但为补偿输入/输出变化, 第一层与第二层权重所需的缩放**不相同**。因此需要为两层用**不同的**正则强度。幸运的是很容易:

$$p(\theta) = N(W|0, \frac{1}{\alpha_w} I) N(V|0, \frac{1}{\alpha_v} I) N(b|0, \frac{1}{\alpha_b} I) N(c|0, \frac{1}{\alpha_c} I) \quad (16.79)$$

其中  $b, c$  为偏置。为直观理解超参数的影响, 我们可从该先验中采样 MLP 参数并画出随机函数; 图16.17给出示例。减小  $\alpha_v$ : 允许第一层权重变大, 使函数的**S形更陡**。减小  $\alpha_b$ : 允许第一层偏置变大, 使 S 形的**中心左右平移**更自由。减小  $\alpha_w$ : 允许第二层权重变大, 使函数更“**起伏/抖动**” (对输入更敏感, 动态范围更大)。减小  $\alpha_c$ : 允许第二层偏置变大, 使函数**整体上下移动**更多。 (第15章我们会看到更容易的“对函数加先验”的做法。)

### 16.5.6.2 权重剪枝

由于网络中权重很多, 鼓励**稀疏**常很有用。90 年代有多种特设方法 (如“Optimal Brain Damage”), 见 Bishop(1995)。也可用第13章更有原则的稀疏技术: 一种是  $L_1$  正则 (见图16.18); 另一种是 ARD, 详见16.5.7.5。

### 16.5.6.3 软权重共享

另一种正则化方式是鼓励**相似权重共享统计强度**。但如何事先分组? 可用**混合模型**来学习: 把  $p(\theta)$  建模为若干 (对角) 高斯的混合。被分到同一簇的参数共享同一均值和方差 (若该簇方差小, 则参数会很接近)。称为**软权重共享** (Nowlan & Hinton, 1992)。实践中并不常用; 细节见 Bishop(2006a,p271)。

### 16.5.6.4 半监督嵌入

一种有趣的“深”前馈网络正则化方式是鼓励隐藏层把**相似对象映到相似表征**。这很有用，因为常常容易得到“**旁信息**”：若干对相似/不相似样本。例如视频分类里，相邻帧可认为相似，而时间上相距很远的帧可认为不相似 (Mobahi et al., 2009)；且无需任何标签。令  $S_{ij} = 1$  表示样本  $i, j$  相似， $S_{ij} = 0$  否则。令  $f(x_i)$  为某种嵌入 (如  $f(x_i) = z(x_i, \theta)$ ，即网络隐藏层)。定义基于两对象嵌入与观测到的相似性的损失：

$$L(f_i, f_j, S_{ij}) = \begin{cases} \|f_i - f_j\|^2, & S_{ij} = 1 \\ \max(0, m - \|f_i - f_j\|^2), & S_{ij} = 0 \end{cases} \quad (16.80)$$

其中  $m$  为最小间隔。由此定义用于训练网络的**增广损失**：

$$\sum_{i \in L} \text{NLL}(f(x_i), y_i) + \lambda \sum_{i,j \in U} L(f(x_i), f(x_j), S_{ij}) \quad (16.81)$$

$L$  为有标签集， $U$  为无标签集， $\lambda \geq 0$  为权衡系数。该目标可用 SGD 轻松优化：每次迭代，抽一个有标签样本做 NLL 的梯度步；再抽一对相似的无标签样本更新  $\lambda L(\cdot, \cdot, 1)$ ；最后抽一例  $x_k$  (与  $x_i$  大概率不相似) 更新  $\lambda L(\cdot, \cdot, 0)$ 。该技术有效是因为它能利用**海量数据**。相关工作：Collobert & Weston(2008) 让网络区分**有效与无效**英文句子：用维基百科 6.31 亿词生成长度 11 的滑窗作正例，负例由把中间词替换为随机英文单词得到（高概率无效）。该网络训练一周，其学到的潜在表示再作为输入去做监督的语义角色标注任务（可用标注极少）。另见 Ando & Zhang(2005)。

## 16.5.7 贝叶斯推断（翻译）

尽管 MAP 能减少过拟合，但采用**完全贝叶斯**方式来“拟合”神经网络仍有充分理由：

- 把参数**积分掉**而非优化，比 MAP 更强的正则化；
- 可用**贝叶斯模型选择**来决定超参数设置与隐藏单元数，尤其当超参数很多时（如 ARD），这可能比交叉验证**更快**；
- 对参数不确定性建模会诱导**预测分布的不确定性**，这在主动学习、风险规避决策等问题上很重要；
- 可用**在线推断**（如扩展卡尔曼滤波）做在线学习 (Haykin 2001)。

可以采用多种近似贝叶斯技术。本节讨论由 MacKay(1992, 1995b) 首提的**拉普拉斯近似**；也可用**混合蒙特卡洛** (Neal 1996) 或**变分贝叶斯** (Hinton & Camp 1993; Barber & Bishop 1998)。

### 16.5.7.1 回归的参数后验

按 Bishop(2006a, §5.7) 展示，汇总自 MacKay(1992, 1995b)。先验取  $p(w) = \mathcal{N}(w|0, (1/\alpha)I)$ ， $w$  表示所有权重的集合。记噪声精度为  $\beta = 1/\sigma^2$ 。后验近似为

$$p(w|D, \alpha, \beta) \propto \exp\{-E(w)\} \quad (16.82)$$

$$E(w) = \beta E_D(w) + \alpha E_W(w) \quad (16.83)$$

$$E_D(w) = \frac{1}{2} \sum_{n=1}^N (y_n - f(x_n, w))^2 \quad (16.84)$$

$$E_W(w) = \frac{1}{2} w^\top w \quad (16.85)$$

其中  $E_D$  为数据误差,  $E_W$  为先验误差,  $E$  为总误差 (负对数先验+负对数似然)。在 MAP 点  $w_{MP}$  处做二阶泰勒展开:

$$E(w) \approx E(w_{MP}) + \frac{1}{2}(w - w_{MP})^\top A(w - w_{MP}) \quad (16.86)$$

$A$  为  $E$  的海森矩阵:

$$A = \nabla \nabla E(w_{MP}) = \beta H + \alpha I \quad (16.87)$$

其中  $H = \nabla \nabla E_D(w_{MP})$  是数据误差的海森矩阵。可用反向传播的变体以  $O(d^2)$  时间精确计算 ( $d$  为参数数), 或在用拟牛顿法找极值时使用其内部的 (低秩)  $H$  近似。(对角近似通常很不准。) 在该二次近似下, 后验为高斯:

$$p(w|\alpha, \beta, D) \approx N(w | w_{MP}, A^{-1}) \quad (16.88)$$

### 16.5.7.2 分类的参数后验

分类与回归相同, 只是  $\beta = 1$ , 且  $E_D$  为交叉熵:

$$E_D(w) = \sum_{n=1}^N [y_n \ln f(x_n, w) + (1 - y_n) \ln (1 - f(x_n, w))] \quad (16.89-16.90)$$

### 16.5.7.3 回归的预测后验

预测后验为

$$p(y|x, D, \alpha, \beta) = \int N(y | f(x, w), 1/\beta) N(w | w_{MP}, A^{-1}) dw \quad (16.91)$$

由于  $f(x, w)$  的非线性, 不可解析。对  $w_{MP}$  做一阶泰勒近似:

$$f(x, w) \approx f(x, w_{MP}) + g^\top (w - w_{MP}) \quad (16.92)$$

$$g = \nabla_w f(x, w)|_{w=w_{MP}} \quad (16.93)$$

于是得到线性高斯模型 + 高斯先验。由式(4.126):

$$p(y|x, D, \alpha, \beta) \approx N(y | f(x, w_{MP}), \sigma^2(x)) \quad (16.94)$$

其中预测方差随  $x$  变化：

$$\sigma^2(x) = \beta^{-1} + g^\top A^{-1} g \quad (16.95)$$

在训练数据稀少的区域，误差条更宽。见图16.19。

#### 16.5.7.4 分类的预测后验

二分类时情形与第8.4.4节逻辑回归类似，但预测均值是  $w$  的非线性函数。设  $\mu = E[y|x, w] = \text{sigm}(a(x, w))$ ，其中  $a(x, w)$  为最后一层的突触前输出。作线性近似：

$$a(x, w) \approx a_{MP}(x) + g^\top (w - w_{MP}) \quad (16.96)$$

其中  $a_{MP}(x) = a(x, w_{MP})$ ， $g = \nabla_x a(x, w_{MP})$  可由改造的反向传播得到。显然

$$p(a|x, D) \approx N(a(x, w_{MP}), g(x)^\top A^{-1} g(x)) \quad (16.97)$$

故输出的预测后验为

$$p(y=1|x, D) = \int \text{sigm}(a) p(a|x, D) da \approx \text{sigm}(\kappa(\sigma_a^2) b^\top w_{MP}) \quad (16.98)$$

其中  $\kappa$  由式(8.70)给出（便于参考再记一次）：

$$\kappa(\sigma^2) = (1 + \pi\sigma^2/8)^{-1/2} \quad (16.99)$$

当然，更简单（且潜在更准确）的方法是从高斯后验抽若干样本，用蒙特卡洛近似预测后验。无论哪种方式，把参数不确定性纳入考虑会降低预测置信度（更保守），但决策边界本身不变。

#### 16.5.7.5 神经网络中的 ARD

一旦做了拉普拉斯近似，就可用第13.7.4.2节的不动点方程对超参数  $\alpha$  最大化边际似然。通常对每个节点外出的权重向量用一个超参数，效果类似组 lasso (§13.5.1)。即先验为

$$p(\theta) = \prod_{i=1}^D N(v_{:,i} | 0, \frac{1}{\alpha_{v,i}} I) \prod_{j=1}^H N(w_{:,j} | 0, \frac{1}{\alpha_{w,j}} I) \quad (16.100)$$

若得到  $\alpha_{v,i} = \infty$ ，则输入特征  $i$  不相关，其权重  $v_{:,i}$  被剪除；同理  $\alpha_{w,j} = \infty$  表示隐藏特征  $j$  不相关。这称为自动相关性判定 (ARD，详见第13.7节)，它为非线性模型提供了高效的变量选择。软件包 NETLAB 有 ARD+神经网络的简单例子 demand：数据由非线性回归  $f(x_1, x_2, x_3) = \sin(2\pi x_1) +$  生成，其中  $x_2$  是  $x_1$  的带噪拷贝。可见  $x_2, x_3$  对预测目标无关；但  $x_2$  与相关的  $x_1$  相关。使用 ARD，最终超参数为

$$\alpha = [0.2, 21.4, 249001.8] \quad (16.101)$$

显然特征 3 无关，特征 2 仅弱相关，特征 1 很相关。

---

## 通俗解释（重点难点拆开讲）

- **反向传播在干嘛？**

正向：一层层算输出。反向：把“**输出层的误差**”乘以**各层的导数**，一层层传回去，得到每条边该怎么改。输出层梯度=“该层输入” $\times$ “该层误差”；上一层误差=“下一层误差 $\times$ 权重 $\times$ 激活函数导数”。

- **tanh vs sigmoid (图16.16)**

tanh 输出在  $[-1, 1]$  居中，梯度更“均衡”，更适合**隐藏层**；sigmoid 在  $[0, 1]$ ，适合**二分类输出**。

- **参数不可辨识**

翻转某个隐藏单元的“进出”权重符号，或交换两个隐藏单元，不改变模型输出。于是同一模型有很多等价参数解；别指望“平均参数”，要**平均预测**或挑最好的一次。

- **正则化与超参数 (图16.17)**

不同层用不同的  $\alpha$ ：

$\downarrow \alpha_v \Rightarrow$  一层权重大  $\Rightarrow$  S 形更陡；

$\downarrow \alpha_b \Rightarrow$  一层偏置大  $\Rightarrow$  S 形中心左右移；

$\downarrow \alpha_w \Rightarrow$  二层权重大  $\Rightarrow$  函数更“抖”(高频)；

$\downarrow \alpha_c \Rightarrow$  二层偏置大  $\Rightarrow$  整体上下平移。

实操上配合**权重衰减、早停、标准化**最稳。

- **剪枝与稀疏**

用  $L_1$  或 ARD 自动把无关连接/特征置零，得到更小更稳的网络（图16.18）。

- **半监督嵌入**

给网络一个“同类要靠近、异类要分开”的辅助目标（用相似/不相似对），即便没标签也能用大量数据**塑造隐藏表示**，再用少量标签微调任务。

- **贝叶斯神经网络**

不再“只找一个最好权重”，而是承认权重有**不确定性**：

- **拉普拉斯近似**：在 MAP 点附近把后验近似成高斯；

- **预测分布**：均值 $\approx$ 用  $w_{MP}$  的预测；方差=噪声方差 + 参数不确定性经由梯度传播（式16.95）。数据少的地方**误差条更宽**（图16.19）；

- 二分类时也可得到更“保守”的概率（式16.98）；

- ARD 自动判断输入/隐藏特征是否相关，实现**变量选择**。

## 要点总结

- **反向传播**: 输出误差 → 乘导数 → 逐层反传; 梯度 = 局部输入 × 局部误差。
- **隐藏激活**: 优先用  $\tanh$ ; 输出二分类用  $\text{sigmoid}$ 。
- **不可辨识**: 同一函数很多等价参数; 重启/平均预测更靠谱。
- **正则化**: 早停 +  $L_2$  (权重衰减); 不同层用不同强度 ( $\alpha_v, \alpha_w, \alpha_b, \alpha_c$ ); 可用  $L_1/\text{ARD}$  剪枝; 可用半监督嵌入塑形。
- **贝叶斯视角**: 拉普拉斯/采样近似后验; 预测有不确定度 (误差条随  $x$  变); 用边际似然调超参, ARD 做**非线性变量选择**。

需要, 我可以把反向传播的每一步用一个小数例算一遍, 或给出 PyTorch 版的最小实现帮助你上手。

16.5节主要在讲什么? 我不太懂

已思考 10s

下面把 16.5 节的主线用“大白话”理一遍, 你抓住这些就明白了。

## 这节在讲啥 (一句话)

讲的是: **前馈神经网络 (MLP)** —— 它长什么样、怎么训 (反向传播)、有哪些变体 (卷积网等)、如何防过拟合, 以及还能用贝叶斯方法给它“带不确定性”。

## 核心脉络 (按小节)

### 1. 什么是 MLP (前馈神经网络)

- 把很多个“**小的逻辑回归/线性单元**”一层层叠起来。
- 形式: 先算隐藏层  $z = g(Vx)$ , 再算输出  $\hat{y} = h(Wz)$ 。
  - $g$ : 隐藏层激活 (常用  $\tanh$ ), 学“特征”。
  - $h$ : 输出层激活 (回归用线性; 二分类用  $\text{sigmoid}$ ; 多类用  $\text{softmax}$ )。
- MLP 是“通用逼近器”: 隐藏单元够多就能逼近任意平滑函数。

### 2. 卷积神经网络 (CNN) 是 MLP 的特殊形态

- 对图像/序列更友好:

- 局部感受野（只看邻域）、权重共享（同一卷积核到处用）→ 参数少、平移不变性强。
- 在 MNIST 手写体上：按层做卷积/池化/全连接，可用数据增强（形变、扰动）进一步降错。经典结构：LeNet-5。

### 3. 还有别的网络拓扑

- 可以有跳连、稀疏连；允许反馈就成了 RNN（做语言建模很强）。对称连接是 Hopfield/Boltzmann（无监督）。

### 4. 一点点历史

- 感知机→被证明只能分线性可分→低潮；
- 反向传播（1986）带回高潮；
- 90s 末 SVM/核方法很火；
- 2000s 中期开始，深层网络（预训练/对比散度）又复兴。

### 5. 反向传播（怎么训）

- 用链式法则算梯度：
  - 前向算各层激活；
  - 后向把输出层误差  $\delta$  乘以导数一层层“反传”；
  - 每条权重的梯度 = “该层输入 × 该层误差”。
- 大数据上一般用 \*\*随机梯度下降（SGD）\*\* 或小批次训练。

### 6. 可辨识性（为什么同一函数有很多组参数）

- 例如把某隐藏单元的入/出权重同时取反、或交换两个隐藏单元，输出不变 ⇒ 很多等价解。
- 目标非凸有局部极小；常用多次随机初始化、平均预测而不是平均参数。

### 7. 正则化与控制复杂度

- 早停：验证误差上升就停。
- 权重衰减 ( $L_2$ )：给权重加高斯先验，鼓励小权重；不同层用不同强度（第一层、第二层、偏置分开调）。
- 稀疏：用  $L_1$  或 ARD 自动把不重要的连接/特征“剪掉”。
- 半监督嵌入：给隐藏表示加“相似靠近、不相似分开”的辅助损失，能利用大量无标签数据。

### 8. 贝叶斯视角（给预测带上“置信度”）

- 不只找一个最优权重，而是近似整个后验分布（如 拉普拉斯近似、HMC、变分）。
- 预测时能得到均值 + 方差（哪儿数据少，误差条更宽）。
- 用边际似然还能自动挑超参数；ARD 做非线性变量选择。

## 如果只记住 5 句话

1. MLP = “学特征的层”  $z = g(Vx) +$  “做预测的层”  $h(Wz)$ 。
2. 卷积网通过**局部+共享**大幅降参数、带平移不变性，是处理图像/序列的首选。
3. **反向传播**= 前向算值、后向传误差、每层本地算梯度。
4. 防过拟合靠 **早停 /  $L_2 / L_1 / \text{ARD} / \text{数据增强}$**  等。
5. **贝叶斯神经网络**能输出置信区间，并用证据最大化自动调超参。

你要是想，我可以举一个极小的数值例子（两层、手算一次反向传播）让公式落地，或者用 PyTorch 给你跑一个 30 行的小 MLP/CNN 演示。

Class	$C_1$	$C_2$	$C_3$	$C_4$	$C_5$	$C_6$	...	$C_{15}$
0	1	1	0	0	0	0	...	1
1	0	0	1	1	1	1	...	0
							⋮	
9	0	1	1	1	0	0	...	0

翻译并解释以下内容，翻译不能有遗漏，解释请通俗易懂且详尽，因为我的数学基础不太好，最后总结要点。

Table 16.2 Part of a 15-bit error-correcting output code for a 10-class problem. Each row defines a two-class problem. Based on Table 16.1 of (Hastie et al. 2009).

### 16.6 Ensemble learning

Ensemble learning refers to learning a weighted combination of base models of the form

$$f(y|x, \pi) = \sum_{m \in M} w_m f_m(y|x) \quad (16.102)$$

where the  $w_m$  are tunable parameters. Ensemble learning is sometimes called a committee method, since each base model  $f_m$  gets a weighted “vote”. Clearly ensemble learning is closely related to learning adaptive-basis function models. In fact, one can argue that a neural net is an ensemble method, where  $f_m$  represents the  $m^{\text{th}}$  hidden unit, and  $w_m$  are the output layer weights. Also, we can think of boosting as kind of ensemble learning, where the weights on the base models are determined sequentially. Below we describe some other forms of ensemble learning.

### 16.6.1 Stacking

An obvious way to estimate the weights in Equation 16.102 is to use

$$\hat{w} = \operatorname{argmin}_w \sum_{i=1}^N L(y_i, \sum_{m=1}^M w_m f_m(x)) \quad (16.103)$$

However, this will result in overfitting, with  $w_m$  being large for the most complex model. A simple solution to this is to use cross-validation. In particular, we can use the LOOCV estimate

$$\hat{w} = \operatorname{argmin}_w \sum_{i=1}^N L(y_i, \sum_{m=1}^M \hat{w}_{-i} f_m(x)) \quad (16.104)$$

where  $\hat{f}_{-i}(x)$  is the predictor obtained by training on data excluding  $(x_i, y_i)$ . This is known as stacking, which stands for "stacked generalization" (Wolpert 1992). This technique is more robust to the case where the "true" model is not in the model class than standard BMA (Clarke 2003). This approach was used by the Netflix team known as "The Ensemble", which tied the submission of the winning team (BellKor's Pragmatic Chaos) in terms of accuracy (Sill et al. 2009). Stacking has also been used for problems such as image segmentation and labeling.

### 16.6.2 Error-correcting output codes

An interesting form of ensemble learning is known as error-correcting output codes or ECOC (Dietterich and Bakiri 1995), which can be used in the context of multi-class classification. The idea is that we are trying to decode a symbol (namely the class label) which has  $C$  possible states. We could use a bit vector of length  $B = \lceil \log_2 C \rceil$  to encode the class label, and train  $B$  separate binary classifiers to predict each bit. However, by using more bits, and by designing the codewords to have maximal Hamming distance from each other, we get a method that is more resistant to individual bit-flipping errors (misclassification). For example, in Table 16.2, we use  $B = 15$  bits to encode a  $C = 10$  class problem. The minimum Hamming distance between any pair of rows is 7. The decoding rule is

$$\hat{c}(x) = \min_c \sum_{b=1}^B |C_{cb} - \hat{p}_b(x)| \quad (16.105)$$

where  $C_{cb}$  is the  $b$ 'th bit of the codeword for class  $c$ .

(James and Hastie 1998) showed that a random code worked just as well as the optimal code: both methods work by averaging the results of multiple classifiers, thereby reducing variance.

### 16.6.3 Ensemble learning is not equivalent to Bayes model averaging

In Section 5.3, we discussed Bayesian model selection. An alternative to picking the best model, and then using this to make predictions, is to make a weighted average of the predictions made by each model, i.e., we compute

$$p(y|x, D) = \sum_{m \in M} p(y|x, m, D)p(m|D) \quad (16.106)$$

This is called Bayes model averaging (BMA), and can sometimes give better performance than using any single model (Hoeting et al. 1999). Of course, averaging over all models is typically computationally infeasible (analytical integration is obviously not possible in a discrete space, although one can sometimes use dynamic programming to perform the computation exactly, e.g., (Meila and Jaakkola 2006)). A simple approximation is to sample a few models from the posterior. An even simpler approximation (and the one most widely used in practice) is to just use the MAP model.

It is important to note that BMA is not equivalent to ensemble learning (Minka 2000c). This latter technique corresponds to enlarging the model space, by defining a single new model which is a convex combination of base models, as follows:

$$p(y|x, \pi) = \sum_{m \in M} \pi_m p(y|x, m) \quad (16.107)$$

In principle, we can now perform Bayesian inference to compute  $p(\pi|D)$ ; we then make predictions using  $p(y|x, D) = p(y|x, \pi)p(\pi|D)d\pi$ . However, it is much more common to use point estimation methods for  $\pi$ , as we saw above.

已思考 10s

下面按你的要求来：先逐字不遗漏翻译（含公式/表注），再做通俗解释，最后给要点小结。

## 翻译

**表16.2：**一个 10 类问题的 15 比特纠错输出码 (ECOC) 的部分内容。每一行定义了一个二分类问题。改编自 (Hastie 等, 2009) 的表16.1。

(表意：最左列是真实类别标签 0...9；后面的  $C_1, \dots, C_{15}$  是 15 个“子分类器”的目标位。每行是一串 0/1 码字。)

## 16.6 集成学习 (Ensemble learning)

集成学习指的是学习一组基模型的**加权组合**, 形式为

$$f(y | x, \pi) = \sum_{m \in M} w_m f_m(y | x) \quad (16.102)$$

其中  $w_m$  是可调参数。集成学习有时也叫**委员会方法**, 因为每个基模型  $f_m$  都投一张按权重计的“票”。显然, 集成学习与**自适应基函数模型**密切相关。实际上, 可以认为**神经网络**本质上也是一种集成: 把每个隐藏单元看作  $f_m$ , 输出层权重就是  $w_m$ 。另外, **boosting** 也可看作一种集成, 只是它的基模型权重是**按顺序**确定的。下面介绍其它几种集成学习形式。

### 16.6.1 堆叠 (Stacking)

估计式(16.102)中权重的直接方法是

$$\hat{w} = \arg \min_w \sum_{i=1}^N L(y_i, \sum_{m=1}^M w_m f_m(x_i)) \quad (16.103)$$

但这样会**过拟合**: 复杂的模型会拿到很大的  $w_m$ 。一个简单解法是用**交叉验证**。特别地, 用**留一法**(LOOCV) 估计:

$$\hat{w} = \arg \min_w \sum_{i=1}^N L(y_i, \sum_{m=1}^M w_m \hat{f}_m^{-i}(x_i)) \quad (16.104)$$

其中  $\hat{f}_m^{-i}$  表示在去掉第  $i$  个样本  $(x_i, y_i)$  训练得到的第  $m$  个预测器。这叫**堆叠 (stacked generalization)** (Wolpert, 1992)。与标准的**贝叶斯模型平均**(BMA) 相比, 它在“真模型不在候选集里”时更健壮 (Clarke, 2003)。该方法被 Netflix 竞赛中名为“The Ensemble”的队伍使用 (其成绩与获胜队 BellKor's Pragmatic Chaos 持平) (Sill 等, 2009)。堆叠也被用于图像分割与标注等任务。

### 16.6.2 纠错输出码 (ECOC)

一种有趣的集成形式是**纠错输出码** (ECOC; Dietterich & Bakiri, 1995), 用于**多类**分类。

思路: 我们要解码一个有  $C$  个取值 (类别) 的符号。最少只需  $B = \log_2 C$  位来编码, 并训练  $B$  个二分类器分别预测每一位。但如果使用**更多位**, 并把各类别的码字设计成**海明距离**尽量大, 那么就能对单个比特的错误 (误分类) **更鲁棒**。例如表16.2中, 用  $B = 15$  位表示  $C = 10$  类问题; 任意两行 (两个类别) 之间的最小海明距离为 7。解码规则为

$$\hat{c}(x) = \arg \min_c \sum_{b=1}^B |C_{cb} - \hat{p}_b(x)| \quad (16.105)$$

其中  $C_{cb}$  是类别  $c$  的第  $b$  位码字;  $\hat{p}_b(x)$  是第  $b$  个二分类器对“该位为 1”的预测 (概率或决策)。James 与 Hastie (1998) 证明: **随机码与最优码**效果相当; 两者的作用本质上都是**对多个分类器做平均, 从而降低方差**。

### 16.6.3 集成学习 ≠ 贝叶斯模型平均

第5.3节讨论过贝叶斯模型选择。与“挑一个最佳模型后再预测”不同，**模型平均**是对各模型的预测做**后验加权平均**：

$$p(y | x, D) = \sum_{m \in M} p(y | x, m, D) p(m | D) \quad (16.106)$$

这叫**贝叶斯模型平均 (BMA)**，有时比任何单一模型都好 (Hoeting 等, 1999)。当然，穷举所有模型通常不可行（离散空间难以解析积分；偶尔能用动态规划精确计算，如 Meila & Jaakkola, 2006）。一种近似是从后验中采样少量模型；实践中更常见的更简单近似是直接用**MAP 模型**。

需要注意的是：**BMA 并不等同于集成学习** (Minka, 2000)。后者相当于**扩展模型空间**，定义了一个由基模型构成的**凸组合**新模型：

$$p(y | x, \pi) = \sum_{m \in M} \pi_m p(y | x, m) \quad (16.107)$$

理论上可以对  $\pi$  做贝叶斯推断得到  $p(\pi | D)$ ，再用  $\int p(y | x, \pi) p(\pi | D) d\pi$  预测；但实践中更常见的是像前述那样对  $\pi$  做**点估计**。

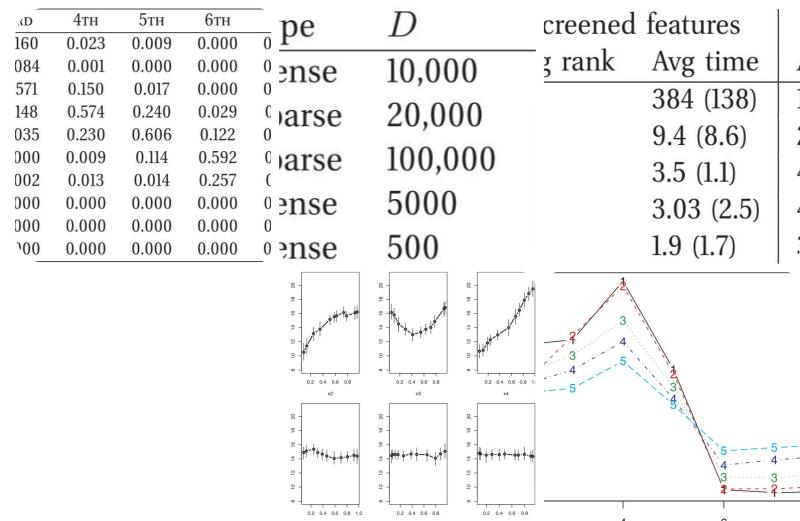
---

## 通俗解释

- **集成**=“多人投票”：把若干不同的模型的预测按权重相加/平均，通常会比任何一个单模更稳更准。
  - **Stacking** 为了防止“在训练集上调权重时偏向最会背题的模型”，它用**交叉验证的出折预测**来学权重（相当于只用“没见过当前样本时的预测”），因此不容易过拟合。
  - **ECOC 多类转多二**：把每个类别编码成长位串；每一位交给一个二分类器去判“这一位是 1 还是 0”。预测时，看哪一类的码字与“15个二分类器的输出”**最像**（与式16.105等价）。位数越多、类别间码字越远（海明距离大），就越能**纠错**；最小距离为  $d_{\min}$  时，理论上可纠正  $\lfloor \frac{d_{\min}-1}{2} \rfloor$  位错误。
  - **BMA vs 集成**：
    - **BMA**是在“模型集合”上按**后验概率**做平均——“相信哪个模型，就给它更大权重”；
    - **集成**是把若干模型**并到一个更大的模型里**（混合/凸组合），权重通常通过优化损失来学。两者思想相近，但**不是同一件事**。
-

## 要点小结

- 集成学习:  $f = \sum w_m f_m$ 。能降方差、提精度。
- Stacking: 用出折预测学组合权重，显著减少过拟合。
- ECOC: 把多类问题拆成多次二分类；用冗余码（大海明距）实现纠错；解码选与码字距离最小的类别。
- 与 BMA 的区别:** BMA 是“在模型集合上做贝叶斯平均”；集成是“把模型混成一个更大的参数化模型并学权重”。两者方法论不同。
- 直觉：**多模型求同存异**通常更稳健，尤其当单个模型各有偏差时。



翻译并解释以下内容，翻译不能有遗漏，解释请通俗易懂且详尽，因为我的数学基础不太好，最后总结要点。

Table 16.3 Fraction of time each method achieved a specified rank, when sorting by mean performance across 11 datasets and 8 metrics. Based on Table 4 of (Caruana and Niculescu-Mizil 2006).

Table 16.4 Summary of the data used in the NIPS 2003 “feature selection” challenge. For the Dorothea datasets, the features are binary. For the others, the features are real-valued.

Table 16.5 Performance of different methods on the NIPS 2003 “feature selection” challenge. (HMC stands for hybrid Monte Carlo; see Section 24.5.4.) We report the average rank (lower is better) across the 5 datasets. We also report the average training time in minutes (standard error in brackets). The MCMC and bagged MLPs use two hidden

layers of 20 and 8 units. The boosted MLPs use one hidden layer with 2 or 4 hidden units. The boosted trees used depths between 2 and 9, and shrinkage between 0.001 and 0.1. Each tree was trained on 80% of the data chosen at random at each step (so-called stochastic gradient boosting). From Table 11.3 of (Hastie et al. 2009). Figure 16.20 Partial dependence plots for the 10 predictors in Friedman's synthetic 5-dimensional regression problem.

Figure 16.21 Average usage of each variable in a BART model fit to data where only the first 5 features are relevant. The different coloured lines correspond to different numbers of trees in the ensemble.

## 16.7 Experimental comparison

We have described many different methods for classification and regression. Which one should you use? That depends on which inductive bias you think is most appropriate for your domain. Usually this is hard to assess, so it is common to just try several different methods, and see how they perform empirically. Below we summarize two such comparisons that were carefully conducted (although the data sets that were used are relatively small). See the website [mlcomp.org](http://mlcomp.org) for a distributed way to perform large scale comparisons of this kind. Of course, we must always remember the no free lunch theorem (Section 1.4.9), which tells us that there is no universally best learning method.

### 16.7.1 Low-dimensional features

In 2006, Rich Caruana and Alex Niculescu-Mizil (Caruana and Niculescu-Mizil 2006) conducted a very extensive experimental comparison of 10 different binary classification methods, on 11 different data sets. The 11 data sets all had 5000 training cases, and had test sets containing ~ 10, 000 examples on average. The number of features ranged from 9 to 200, so this is much lower dimensional than the NIPS 2003 feature selection challenge. 5-fold cross validation was used to assess average test error. (This is separate from any internal CV a method may need to use for model selection.)

The methods they compared are as follows (listed in roughly decreasing order of performance, as assessed by Table 16.3):

- BST-DT: boosted decision trees
- RF: random forest
- BAG-DT: bagged decision trees
- SVM: support vector machine
- ANN: artificial neural network
- KNN: K-nearest neighbors
- BST-STMP: boosted stumps
- DT: decision tree
- LOGREG: logistic regression
- NB: naive Bayes

They used 8 different performance measures, which can be divided into three groups. Threshold metrics just require a point estimate as output. These include accuracy, F-score (Section 5.7.2.3), etc. Ordering/ ranking metrics measure how well positive cases are ordered before the negative cases. These include area under the ROC curve (Section 5.7.2.1), average precision, and the precision/recall break even point. Finally, the probability metrics included cross-entropy (log-loss) and squared error,  $(y - \hat{p})^2$ . Methods such as SVMs that do not produce calibrated probabilities were post-processed using Platt's logistic regression trick (Section 14.5.2.3), or using isotonic regression. Performance measures were standardized to a 0:1 scale so they could be compared.

Obviously the results vary by dataset and by metric. Therefore just averaging the performance does not necessarily give reliable conclusions. However, one can perform a bootstrap analysis, which shows how robust the conclusions are to such changes. The results are shown in Table 16.3. We see that most of the time, boosted decision trees are the best method, followed by random forests, bagged decision trees, SVMs and neural networks.

However, the following methods all did relatively poorly: KNN, stumps, single decision trees, logistic regression and naive Bayes.

These results are generally consistent with conventional wisdom of practitioners in the field. Of course, the conclusions may change if there the features are high dimensional and/ or there are lots of irrelevant features (as in Section 16.7.2), or if there is lots of noise, etc.

## 16.7.2 High-dimensional features

In 2003, the NIPS conference ran a competition where the goal was to solve binary classification problems with large

numbers of (mostly irrelevant) features, given small training sets. (This was called a “feature selection” challenge, but performance was measured in terms of predictive accuracy, not in terms of the ability to select features.) The five datasets that were used are summarized in Table 16.4. The term probe refers to artificial variables that were added to the problem to make it harder. These have no predictive power, but are correlated with the original features.

Results of the competition are discussed in (Guyon et al. 2006). The overall winner was an approach based on Bayesian neural networks (Neal and Zhang 2006). In a follow-up study (Johnson 2009), Bayesian neural nets (MLPs with 2 hidden layers) were compared to several other methods based on bagging and boosting. Note that all of these methods are quite similar: in each case, the prediction has the form

$$\hat{f}(x^*) = \sum_{m=1}^M w_m E[y|x^*, \theta_m] \quad (16.108)$$

The Bayesian MLP was fit by MCMC (hybrid Monte Carlo), so we set  $w_m = 1/M$  and set  $\theta_m$  to a draw from the posterior. In bagging, we set  $w_m = 1/M$  and  $\theta_m$  is estimated by fitting the model to a bootstrap sample from the data. In boosting, we set  $w_m = 1$  and the  $\theta_m$  are estimated sequentially.

To improve computational and statistical performance, some feature selection was performed. Two methods were considered: simple uni-variate screening using T-tests, and a method based on MLP+ARD. Results of this follow-up study are shown in Table 16.5. We see that Bayesian MLPs are again the winner. In second place are either random forests or boosted MLPs, depending on the preprocessing. However, it is not clear how statistically significant these differences are, since the test sets are relatively small.

In terms of training time, we see that MCMC is much slower than the other methods. It would be interesting to see how well deterministic Bayesian inference (e.g., Laplace approximation) would perform. (Obviously it will be much faster, but the question is: how much would one lose in statistical performance?)

## 16.8 Interpreting black-box models

Linear models are popular in part because they are easy to interpret. However, they often are poor predictors,

which makes them a poor proxy for “nature’s mechanism”. Thus any conclusions about the importance of particular variables should only be based on models that have good predictive accuracy (Breiman 2001b). (Interestingly, many standard statistical tests of “goodness of fit” do not test the predictive accuracy of a model.)

In this chapter, we studied black-box models, which do have good predictive accuracy. Unfortunately, they are hard to interpret directly. Fortunately, there are various heuristics we can use to “probe” such models, in order to assess which input variables are the most important.

As a simple example, consider the following non-linear function, first proposed (Friedman 1991) to illustrate the power of MARS:

$f(x) = 10 \sin(\pi x_1 x_2) + 20(x_3 - 0.5)^2 + 10x_4 + 5x_5 + (16.109)$   
where  $\sim N(0, 1)$ . We see that the output is a complex function of the inputs. By augmenting the  $x$  vector with additional irrelevant random variables, all drawn uniform on  $[0, 1]$ , we can create a challenging feature selection problem. In the experiments below, we add 5 extra dummy variables.

One useful way to measure the effect of a set  $s$  of variables on the output is to compute a partial dependence plot (Friedman 2001). This is a plot of  $f(xs)$  vs  $xs$ , where  $f(xs)$  is defined as the response to  $xs$  with the other predictors averaged out:

$$f(xs) = \frac{1}{N} \sum_{i=1}^N f(x_i, xi, -s) \quad (16.110)$$

Figure 16.20 shows an example where we use sets corresponding to each single variable. The data was generated from Equation 16.109, with 5 irrelevant variables added. We then fit a BART model (Section 16.2.5) and computed the partial dependence plots. We see that the predicted response is invariant for  $s \in \{6, \dots, 10\}$ , indicating that these variables are (marginally) irrelevant. The response is roughly linear in  $x_4$  and  $x_5$ , and roughly quadratic in  $x_3$ . (The error bars are obtained by computing empirical quantiles of  $f(x, \theta)$  based on posterior samples of  $\theta$ ; alternatively, we can use bootstrap.)

Another very useful summary computes the relative importance of predictor variables. This can be thought of as a nonlinear, or even “model free”, way of performing variable selection, although the technique is restricted to

ensembles of trees. The basic idea, originally proposed in (Breiman et al. 1984), is to count how often variable  $j$  is used as a node in any of the trees. In particular, let  $v_j = \frac{1}{M} \sum_{m=1}^M I(j \in T_m)$  be the proportion of all splitting rules that use  $x_j$ , where  $T_m$  is the  $m$ 'th tree. If we can sample the posterior of trees,  $p(T_1:M|D)$ , we can easily compute the posterior for  $v_j$ . Alternatively, we can use bootstrap.

Figure 16.21 gives an example, using BART. We see that the five relevant variables are chosen much more than the five irrelevant variables. As we increase the number  $M$  of trees, all the variables are more likely to be chosen, reducing the sensitivity of this method, but for small  $M$ , the method is fairly diagnostic. usage 0.00 0.05 0.10 0.15 0.20 0.25

已思考 15s

下面按你的要求来：先逐字不遗漏翻译（含公式/图表说明），再给出通俗解释，最后总结要点。

---

## 翻译

**表16.3：**在 11 个数据集、8 个评测指标上，以“平均表现”排序时，各方法获得某个名次的时间占比。改编自 (Caruana and Niculescu-Mizil, 2006) 的表4。

**表16.4：**NIPS 2003“特征选择”挑战赛所用数据的摘要。对 Dorothea 数据集，特征为二值型；其余数据集的特征为实值型。

**表16.5：**在 NIPS 2003“特征选择”挑战赛上的不同方法表现。（HMC 表示混合蒙特卡洛；见第24.5.4节。）报告 5 个数据集上的**平均名次**（越小越好），以及**平均训练时间**（分钟，括号内为标准误）。MCMC 与 bagging 的 MLP 使用两层隐藏层（20 与 8 个单元）；boosted MLP 使用一层隐藏层（2 或 4 个单元）。Boosted trees 的树深在 2–9 之间，缩减系数在 0.001–0.1 之间。每棵树在每一步都用 80% 的样本训练（所谓**随机梯度提升**）。改编自 (Hastie 等, 2009) 的表11.3。

**图16.20：**在 Friedman 的 5 维合成回归问题中，对 10 个预测变量的**部分依赖图**。

**图16.21：**把只有前 5 个特征真正相关的数据用 BART 模型拟合后，各变量的**平均使用频率**。不同颜色的线对应集成中不同的树数目。

---

## 16.7 实验性比较

我们已经介绍了许多分类与回归方法。该用哪一个？这取决于你认为最适合你领域的**归纳偏置**。通常这很难判断，所以大家常见的做法是尝试多种方法，看其经验表现。下面总结两次精心进行的比较（尽管所用数据集相对较小）。可参见网站 [mlcomp.org](http://mlcomp.org) 去以分布式方式做大规模比较。当然，务必记住\*\*“没有免费的午餐”定理\*\*（第1.4.9节）：不存在对所有问题都最好的学习方法。

### 16.7.1 低维特征

2006 年，Rich Caruana 与 Alex Niculescu-Mizil 对 10 种二分类方法在 11 个数据集上进行了广泛比较（每个数据集有 5000 个训练样本，测试集平均约 1 万个样本；特征数 9–200，明显低于 NIPS 2003 挑战赛）。使用 5 折交叉验证来评估平均测试误差（与方法内部用于模型选择的交叉验证无关）。

比较的方法（按照表16.3 评估的“由好到差”的大致顺序）：

- **BST-DT**: 提升的决策树 (boosted decision trees)
- **RF**: 随机森林
- **BAG-DT**: 装袋的决策树 (bagging)
- **SVM**: 支持向量机
- **ANN**: 人工神经网络
- **KNN**: K 近邻
- **BST-STMP**: 提升的树桩
- **DT**: 单棵决策树
- **LOGREG**: 逻辑回归
- **NB**: 朴素贝叶斯

采用 8 个性能度量，分三类：

**阈值型**（只需点估计，如准确率、F 分数）；

**排序/排名型**（正样本排在负样本之前的程度，如 ROC 曲线下面积、平均精度、P/R 平衡点）；

**概率型**（交叉熵/对数损失与平方误差  $(y - \hat{p})^2$ ）。

对于不产生校准概率的方法（如 SVM），用 Platt 的逻辑回归（第14.5.2.3节）或等距回归做后处理。为便于比较，把各评测标准化到 0–1。

由于结果会随数据集和指标而变，仅做简单平均并不可靠；因此作者做了**自助法**

**（bootstrap）分析以检验结论的稳健性。表16.3 的结果显示：**

**大多数情况下，提升的决策树效果最佳，其次是随机森林、装袋树、SVM 与神经网络。**

**而 KNN、树桩、单树、逻辑回归、朴素贝叶斯整体表现较弱。**

这些发现与业界“经验规律”基本一致。当然，当特征维度很高/无关特征很多（见 16.7.2），或噪声很大时，结论可能变化。

## 16.7.2 高维特征

2003 年，NIPS 会议举办了比赛：目标是在**大量（多为无关）特征且训练样本很少**的条件下做二分类。（虽然称为“特征选择”挑战，但评分依据是**预测准确率**，不是选特征本身的好坏。）5 个数据集见表 16.4。术语 **probe** 指的是人工加入的变量以增加难度——它们**没有预测力，但与原始特征相关**。

比赛结果见 (Guyon 等, 2006)。总冠军是基于**贝叶斯神经网络**的方法 (Neal & Zhang, 2006)。后续研究 (Johnson, 2009) 将\*\*贝叶斯 MLP (两层隐藏层) \*\*与多种 bagging/boosting 方法进行了比较。需要注意，这些方法形式上都很相似：其预测都可写为

$$\hat{f}(x^{*}) = \sum_{m=1}^M w_m E[y | x^{*}, \theta_m] \quad (16.108)$$

其中：

\*\*贝叶斯 MLP (MCMC) \*\*取  $w_m = 1/M$ ,  $\theta_m$  是从后验抽样得到；

**bagging** 取  $w_m = 1/M$ ,  $\theta_m$  是在自助采样集上拟合得到；

**boosting** 取  $w_m = 1$ ,  $\theta_m$  依次序列化估计。

为改善统计与计算性能，做了特征筛选：一是简单的**单变量 t 检验筛选**；二是基于 MLP+ARD 的方法。后续研究的结果如表 16.5：**贝叶斯 MLP**再度胜出；亚军因预处理不同而在**随机森林**与 boosted MLP 之间。由于测试集较小，差异的统计显著性尚不明确。从训练时间看，MCMC 显著慢于其他方法。一个有趣问题是：若用**确定性的贝叶斯近似**（如**拉普拉斯近似**），速度会快很多，统计性能会损失多少？

## 16.8 解释黑盒模型

线性模型部分流行的原因在于其**易解释**；但它们常常**预测力差**，难以反映“自然机制”，因此关于变量重要性的结论应建立在**预测准确**的模型之上 (Breiman, 2001b)。本章研究了若干**黑盒模型**，它们预测准确但难以直接解释；庆幸的是有一些启发式手段可以“探测”模型，评估输入变量的重要性。

考虑 Friedman (1991) 为展示 MARS 威力而提出的非线性函数：

$$f(x) = 10 \sin(\pi x_1 x_2) + 20(x_3 - 0.5)^2 + 10x_4 + 5x_5 + \varepsilon \quad (16.109)$$

其中  $\varepsilon \sim N(0, 1)$ 。输出是输入的复杂函数。若用 5 个额外的无关均匀随机变量扩充  $x$ , 就得到一个具有挑战性的特征选择问题。

衡量一组变量  $s$  对输出影响的一个有用方式是部分依赖图 (Friedman, 2001)。它绘制  $f(x_s)$  对  $x_s$  的响应, 其中

$$f(x_s) = \frac{1}{N} \sum_{i=1}^N f(x_s, x_{i,-s}) \quad (16.110)$$

即固定  $x_s$  而把其它变量对模型输出的影响取平均。图 16.20 给出一个例子: 用式(16.109)造数据, 再加 5 个无关变量, 拟合 BART (第 16.2.5 节), 并画部分依赖图。可以看到: 对  $s \in \{6, \dots, 10\}$  的响应基本不变, 说明这些变量 (边际上) 无关; 对  $x_4, x_5$  近似线性; 对  $x_3$  近似二次。 (误差条可由对参数  $\theta$  的后验样本的经验分位数得到; 或者用 bootstrap。)

另一个非常有用的总结是计算预测变量的相对重要性。可把它视为一种非线性/“无模型”的变量选择方式, 但该技巧仅适用于树的集成。基本想法 (Breiman 等, 1984) : 统计变量  $j$  在所有树的分裂中被使用的频率。具体地, 令

$$v_j = \frac{1}{M} \sum_{m=1}^M \mathbf{1}(j \in T_m)$$

即在第  $m$  棵树  $T_m$  中是否使用了  $x_j$  的比例。若能从后验  $p(T_{1:M} | D)$  采样, 就能得到  $v_j$  的后验; 或用 bootstrap。

图 16.21 给出一个 BART 例子: 前 5 个相关变量的被选频率明显高于 5 个无关变量。随着树的数量  $M$  增加, 所有变量被选的概率都上升, 降低了此法的分辨率, 但在较小  $M$  下诊断性较强。

## 通俗解释

- **要不要用哪个模型?** 没有放之四海而皆准的答案 (“没有免费的午餐”)。最靠谱的办法: **多跑几种**, 用交叉验证等统一的指标比较。
- **现实对比 (低维)**: 在几十到几百维、样本几千到上万时, **提升树/随机森林/装袋树** 经常赢; SVM、神经网络也不错; KNN、单树、逻辑回归、朴素贝叶斯总体偏弱。
- **高维稀疏/无关特征多时: 贝叶斯神经网络** (用 MCMC) 表现最好; 随机森林、提升的 MLP 也很强。它们在形式上都是**把多个基学习器的预测加权平均** (式 16.108), 差别在于权重/参数的产生方式 (MCMC 抽样、bagging 重采样、boosting 逐步拟合)。代价: **MCMC 很慢**。

- **如何解释黑盒？**

- **部分依赖图**：把不关心的变量平均掉，看“某个（组）变量变化”对预测的影响。
  - **变量使用频率**（树类模型）：看每个变量在所有树的分裂中被用的比例，高者更重要。
  - 这类“解释”应建立在模型**预测力强**的前提下，否则解释没有意义。
- 

## 要点小结

- **比较结论**：在常见低维任务上，Boosted Trees > Random Forests > Bagged Trees ≈ SVM/ANN；KNN/单树/LogReg/NB 较弱。
- **高维小样本**：\*\*贝叶斯 MLP (MCMC) \*\*最强，其次 RF/Boosted MLP；但 MCMC 训练时间代价大。
- 多种强方法本质上都在做： $\hat{f}(x) = \sum_m w_m E[y | x, \theta_m]$ 。差异在于  $w_m, \theta_m$  的获得方式。
- **解释黑盒**：用**部分依赖图**看“边际效应”，用**树的使用频率/重要性**衡量相关性；可配合后验或 bootstrap 得到不确定性条。
- **实践建议**：**多方法对比 + 统一评测 + 交叉验证**；若模型不透明，用上面的**可解释性工具**辅助分析。

这一节主要在讲什么？我不太懂

已思考 12s

这段主要分两块：16.7 实验比较 + 16.8 黑盒模型的解释。一句话版：

先用多组公开数据把常见算法“真刀真枪”比了一遍，告诉你在不同场景谁更稳；再教你几招，如何给准确但不透明的模型做“体检”和解释。

## 16.7 在真实数据上比较方法（到底用谁？）

- **做了什么**：作者用多个数据集和多种评测指标，系统地对比了 10 类常见模型（提升树、随机森林、装袋树、SVM、神经网络、KNN、单棵树、逻辑回归、朴素贝叶斯等）。
- **低维（几十到几百维）结论**：

- 最常赢：**提升的决策树**（Boosted Trees）；
- 其后：**随机森林、装袋树、SVM、神经网络**；
- 较弱：**KNN、树桩、单棵树、逻辑回归、朴素贝叶斯**。
- 表16.3 把各方法拿到第1、2、...名的比例列了出来，结论更稳健（做了bootstrap）。
- **高维小样本（成千上万特征、多数无关）结论：**
  - NIPS 2003 挑战的结果（表16.4&16.5）：**贝叶斯神经网络**综合最强；**随机森林或提升的 MLP**次之。
  - 但**贝叶斯 MCMC 很耗时**；若用确定性近似（如拉普拉斯近似）可能快很多但精度要再评估。
- **一个统一视角**：这些强方法本质上都是把**很多基模型的预测加权平均**（式16.108），差别只是权重/参数怎么来（抽样、重采样、逐步拟合）。
- **提醒**：没有放之四海皆准的赢家（“没有免费的午餐”）；实际要**多方法对比 + 交叉验证**。

## 16.8 如何解释“黑盒”模型（准但不透明）

- **为何要解释**：线性模型好懂但常不准；黑盒（如集成树、神经网）准但难懂。要在**保证预测准确的前提下**评估变量影响。
- **两把好用的“探针”：**
  1. **部分依赖图（PDP）**：把其他变量的影响“平均掉”，只看某一变量变动时预测怎样变化（式16.110）。
    - 图16.20 示例：能看出哪个变量基本无关（曲线水平），哪个近似线性或二次。
  2. **变量重要性（树类专用）**：统计一个变量在所有树里被用来分裂的**频率**（或带权重版本）。
    - 图16.21：真正相关的前5个变量使用频率明显更高；树越多，区分度会被稀释一些。
- **不确定性**：PDP 可配**误差条**（来自后验或自助法），更稳健地解读。

## 快速要点

- 常见场景里，**提升树/随机森林/装袋树**通常表现最好；**SVM/神经网络**也强；**KNN/单树/LogReg/NB**常落后。
- 高维小样本、无关特征多时，**贝叶斯神经网络**领先，但**训练慢**；**RF/Boosted MLP**是实用强选手。

- 解释黑盒的两招：部分依赖图看边际影响，变量使用频率看重要性；先保证模型够准再谈解释。
- 实践建议：多模型+统一评测（交叉验证）选法；选完后用 PDP/重要性图理解与汇报模型。

基于本对话里我贴过的整章原文，输出：①知识点总结  
(分点+关键词+用途) ②子章节关联 (依赖/对比/递进+关系句，附Mermaid可选) ③本章总结。

已思考 22s

太好了！我把整章（16 章：自适应基函数模型）的核心内容整理成你要的三部分 ↘

---

## ① 知识点总结（分点 | 关键词 | 用途）

- 自适应基函数模型 (ABM)**
  - 关键词：** $f(x) = w_0 + \sum_m w_m \phi_m(x; v_m)$ 、学习  $\phi_m$  与权重、非线性但非参数线性。
  - 用途：**统一看待“学特征+做线性”的一大类方法；当线性/固定核不够用时，用数据自动学“特征”。
- CART 决策树 (16.2)**
  - 关键词：**轴对齐划分、区域  $R_m$ 、回归 SSE、分类指标（误分类率/熵/Gini）、贪心生长、剪枝、纯度。
  - 用途：**可解释、能混合离散/连续特征、自动选特征；适合规则提取与快速原型。
- 随机森林 & BART (16.2.5)**
  - 关键词：**Bagging、特征子采样、方差下降、高准确；BART=贝叶斯树集成。
  - 用途：**提升树稳定性与精度；BART 还能给不确定性。
- 与分层专家混合 HME 的对比 (16.2.6)**
  - 关键词：**软划分、线性边界组合、EM。
  - 用途：**当轴对齐切分受限、希望“平均多个专家”而非“落在一个叶子”。
- 广义加性模型 GAM (16.3)**
  - 关键词：** $f(x) = \alpha + \sum_j f_j(x_j)$ 、平滑样条、Backfitting、权重回归 (GLM)。
  - 用途：**在“每个特征单独非线性、少交互”时，给出可解释的非线性回归/分类。

- **MARS (多元自适应回归样条, 16.3.3)**
  - **关键词:** 分段线性基  $(x - t)_+$ ,  $(t - x)_+$ 、贪心前向+后向剪枝、交互的张量积。
  - **用途:** 当确实需要少量交互但仍要可解释且高效时; 与 CART 生长策略近似。
- **Boosting 提升法 (16.4)**
  - **关键词:** 前向分段加和、弱学习器、损失最小化、学习率  $\nu$ 、早停。
  - **常见变体:**
    - L2Boosting: 平方损失 → 拟合残差。
    - AdaBoost: 指数损失 → 样本权重再分配, 强抗过拟合。
    - LogitBoost / 梯度提升: logloss 或任意可微损失的“函数空间梯度下降”。
    - Sparse Boosting/LARS: 逐步选择变量, 得到稀疏解。
    - MART: 弱学习器用浅树 (典型 6 叶), 叶值二次优化。
  - **用途:** 通用强基线; 小调参即可拿到很强的分类/回归效果; 还能做变量选择。
- **为什么 Boosting 好 (16.4.8–9)**
  - **关键词:** 等价于  $L_1$  型正则 (逐步加特征)、最大化间隔 (分类)、与混合专家的关系 (贝叶斯视角)。
  - **用途:** 理解其抗过拟合与强泛化的原因。
- **前馈神经网络/MLP (16.5)**
  - **关键词:** 隐藏层  $z = g(Vx)$ 、输出层  $h(Wz)$ 、通用逼近、反向传播、非凸、早停/权重衰减。
  - **用途:** 当需要“学到复杂特征组合”且数据量较大时。
- **卷积神经网络 CNN (16.5.1)**
  - **关键词:** 局部感受野、权重共享、平移不变、下采样/池化、数据增强 (形变)。
  - **用途:** 一维/二维信号 (语音/图像/文档) 的强特征抽取器; MNIST 等任务的经典方案。
- **BP 与正则化 (16.5.4–16.5.6)**
  - **关键词:** 链式求导、层级误差信号、可辨识性问题、早停、 $L_2$  (权重衰减)、分层一致高斯先验、 $L_1$ /ARD 剪枝、软共享、半监督嵌入。
  - **用途:** 稳定训练、避免过拟合、做变量/连接稀疏化。
- **神经网络的贝叶斯推断 (16.5.7)**
  - **关键词:** 拉普拉斯近似、HMC、预测后验、回归/分类的误差条、ARD 超参数学习。
  - **用途:** 给出不确定性、自动相关性判断 (变量相关性/隐藏单元相关性)。
- **集成学习 (16.6)**

- **关键词**: 堆叠 Stacking (CV 上学权重)、ECOC (编码多分类, 纠错)、与 BMA 区别 (扩展模型族 vs 后验加权)。
  - **用途**: 组合多模型、稳健化; 多分类更抗“单个分类器出错”。
  - **实验比较 (16.7)**
    - **关键词**: 低维: 提升树≈最强, 其次 RF/Bagging/SVM/NN; 高维小样本: 贝叶斯 MLP 领先但耗时。
    - **用途**: 选型参考与预期; 认识时间—精度的权衡。
  - **黑盒模型解释 (16.8)**
    - **关键词**: PDP 部分依赖图、树系变量重要性 (分裂频率/增益)、BART 后验不确定性。
    - **用途**: 在不牺牲精度的前提下解释变量作用, 支持汇报与决策。
- 

## ② 子章节关联 (依赖 | 对比 | 递进)

- 16.1 → **(奠基)** → 16.2/16.3/16.4/16.5  
ABM 统一框架提出; 后续各节是“如何选  $\phi_m$  与怎么学”的四条路线: **树、可加/样条、提升、神经网络**。
- 16.2 ↔ 16.3 (**对比+互通**)  
决策树=分段常数/线性在网格上的极端做法; MARS 用反射样条形成“软的分段线性”, 且其**贪心生长**≈CART。  
用途对比: 树可解释、边界轴对齐; MARS 可线性+少量交互, 边界更平滑。
- 16.4 (**递进/总括**)  
从“固定弱学习器”出发, 通过**损失驱动的前向加和**把 ABM 做成强学习器; MART = 16.2 (树) + 16.4 (梯度提升)。
- 16.5 (**另一条路**)  
与 16.2/16.3/16.4 的“分段/加和”不同, MLP/CNN 直接学连续可微的  $\phi_m$ ; 背后同样是“特征学习 + 简单输出层”。
- 16.6 (**汇合**)  
把前面任一基学习器组合起来 (Stacking/编码/与 BMA 区分) ——方法论的“胶水”。
- 16.7 (**证据**)  
用系统实验证 16.2–16.6 的经验强弱与适用场景。
- 16.8 (**落地**)  
解决 16.4/16.5/16.6 这类“黑盒强模型”的可解释性与变量重要度评估。

**(可选) Mermaid 关系图**

```

flowchart TD
    A[16.1 ABM框架] --> B[16.2 CART/树]
    A --> C[16.3 GAM/MARS]
    A --> D[16.4 Boosting]
    A --> E[16.5 神经网络 MLP/CNN]
    B <--> C
    D --> F[MART=梯度提升树]
    B --> F
    C --> F
    B --> G[16.6 集成学习]
    C --> G
    D --> G
    E --> G
    G --> H[16.7 实验比较]
    F --> H
    E --> H
    H --> I[16.8 解释(PDP/重要性)]
    F --> I
    E --> I
  
```

## 关系句小结

- “**统一**”——16.1 给出 ABM 总方程；
- “**分流**”——16.2/16.3/16.5 提供不同的  $\phi$  与学习方式；
- “**增强**”——16.4 用优化视角把弱学习器迭代成强模型；
- “**组合**”——16.6 把多模型拼成更稳的系统；
- “**验证+解释**”——16.7 比赛与实验；16.8 让黑盒可读。

## ③ 本章总结（面向实践）

- **一条主线：**把“**学到好的特征  $\phi(x)$  + 在其上做简单预测**”作为统一范式。树、样条、提升、神经网络、集成，都是在这条线上做权衡：
  - **精度：**Boosting/MART、RF、CNN 常最强；
  - **可解释：**单树、GAM/MARS、PDP/重要性图；
  - **数据规模/计算：**树/提升调参少、收敛快；NN 需更多样本与算力；贝叶斯方法最稳但最慢；
  - **稳健性/方差：**集成 (Bagging/RF/Boosting/BART) 降方差显著；
  - **场景：**
    - 低维/结构混杂 → 提升树或 RF；
    - 高维小样本、特征多且稀疏 → 贝叶斯 MLP / 带 ARD 的方法；
    - 图像/序列 → CNN/RNN；
    - 需要可解释 → GAM/MARS/单树 + PDP/重要性。
- **方法选型建议**
  1. 先拿 **梯度提升树/随机森林** 做强基线；
  2. 若任务是视觉/文本序列，直接上 CNN/RNN；
  3. 特征极多且样本少，考虑 **稀疏化/ARD/贝叶斯**；
  4. 训练后务必做 **交叉验证与解释性分析 (PDP/重要性)**；
  5. 有多种模型时，用 Stacking/Ensemble 做稳健融合。

**结论：**第16章不是“又多了几种模型”，而是给了一个**统一视角与实践路线：以 ABM 为核心，把特征学习、损失驱动的前向加和、集成与解释串成一体**，帮助你在不同数据与

约束下，拿到“准、稳、可用、可讲”的模型。

请用通俗易懂的话总结本章，使我能够理解

已思考 18s

下面把第 16 章用“日常话”捋一遍，尽量不靠公式。

---

## 一句话版

本章讲：别死盯一个固定特征/函数，改成把很多“简单小块”学出来再拼起来。这些小块叫“基函数/弱学习器”。树、提升法、神经网络、样条……看似不同，其实都是在学特征 + 线性组合这条路上各显身手。

---

## 核心思路（像搭乐高）

- 目标函数 = **许多小积木的加和**：

$$f(x) \approx \sum w_m \phi_m(x).$$

重点是： $\phi_m$  不是人手写，而是从数据里学出来（所以叫“自适应基函数”）。

---

## 几条经典“搭积木”的路

### 1. 决策树 / CART

- 做法：不停用“是否小于阈值”把空间切成小方块，每块给个预测值。
- 优点：直观好解释、混合特征都能上；
- 缺点：单棵树**不稳、容易过拟合**；
- 解法：剪枝；用很多树平均：Bagging/随机森林（降方差），或做贝叶斯树集成 BART（带不确定性）。

### 2. 广义加性模型 GAM / MARS

- 做法：把整体拆成“各特征的平滑曲线相加”，必要时再加少量交互。
- MARS 用“**折线（hinge）基函数**”贪心地加、再剪，像软版的树。
- 适合：每个变量**单独非线性**、交互不多、还想看得懂。

### 3. Boosting 提升法

- 做法：从一个弱模型开始，**每次修前一次的错**，一块一块往上加；
- 代表：**AdaBoost**（指数损失）、**梯度提升/LogitBoost**（把“损失的梯度”当新目标去学）、**MART**（弱学习器用浅树，常用）。
- 特点：调参不多、**效果强还不过拟合**（有学习率/早停）；还能做**稀疏变量选择**。

#### 4. 神经网络 (MLP/CNN)

- 做法：多层把原始输入**变成更有用的特征**（隐藏层），最后再线性/softmax 输出。
- **BP 反向传播算梯度；早停、权重衰减(L2) 防过拟合；**
- CNN靠“局部感受野+权重共享+池化”，特别适合**图像/语音/文本；数据增强常见**。
- 还可以做**贝叶斯 NN拿到不确定性和自动相关性判断 (ARD)**，但更慢。

#### 5. 集成学习

- **Stacking**：用交叉验证的预测再学一个加权器；
  - **ECOC**：多分类先“编码成多次二分类”，能**纠错**；
  - 注意它与**贝叶斯模型平均(BMA)不同**：一个是建一个“**加权大模型**”，一个是在**模型集上做后验平均**。
- 

### 实验给的“择术指南”

- **低维、中等样本**：**提升树** ≈ **最强**，其后是**随机森林/Bagging**，再是**SVM/神经网**；单树、朴素贝叶斯、KNN 往往靠后。
  - **高维、小样本、特征多且多数无关**：**贝叶斯 MLP (带 ARD)** 表现最好，但训练最慢。
  - **图像/序列**：**CNN / RNN**是常规首选。
- 

### 黑盒也能解释

- **部分依赖图 (PDP)**：看某个变量变动时，平均预测怎么变。
  - **树系的重要性**：变量被用来分裂的频率/增益。
  - **BART等还能给误差条 (不确定性)**。
- 

### 落地清单 (照着做就行)

1. 先用**梯度提升树**（如 XGBoost/LightGBM）或**随机森林**做强基线；

2. 设**小学习率+早停**, 用交叉验证选超参;
  3. 看 PDP/**重要性**, 排查变量、和业务对齐;
  4. 若是**图像/文本/语音**或有大量数据 → 上 CNN/RNN;
  5. 特征极多且样本少 → 用**稀疏化/ARD/贝叶斯**思路;
  6. 多个好模型 → Stacking/简单加权 再抬一档。
- 

## 记住这 3 点就不亏

- **统一观**: 本章方法本质都是“**学特征 + 线性拼接**”。
- **强基线**: 提升树/随机森林几乎任何结构化数据都好使。
- **可解释**: 用 PDP + **变量重要性**, 黑盒也能讲清楚。

这一套理解住, 你就能在不同任务下迅速选模型、稳妥调参、还能把结果讲明白。