摘　要

本文依次介绍了神经元、单层感知机与多层感知机模型，梳理了神经网络的基本知识。在详细推导并阐述了神经网络训练过程中，链式求导法则的应用与基于梯度优化的基本原理之后，本文列举了常见的神经网络的类型。

在任务一中，我们首先运行基本代码，发现损失函数没有收敛到 0. 经过分析得出结论：**线性不可分的数据集无法使单层感知机收敛**。调整数据集后，最终我们成功让单层神经网络模型的损失函数收敛至 0. 在任务二中，我们在基础代码上进行添加与修改，为单层感知机添加了一层具有 20 个节点的隐藏层，使其成为多层感知机。以**万能近似定理**为理论依据，最终成功让多层感知机完美拟合原始数据集。在任务三中，我们通过矩阵变换，推导将偏置融入权重的方程模型，并通过编程实现，最终得到了与任务一、二相同的结果。

关键词：Python；神经网络；链式求导法则；梯度下降

目录

第 1 章　神经网络简介

神经网络 (neural networks) 的研究很早就已出现。今天，“神经网络”已是一个相当大的、多学科交叉的学科领域，各相关学科对神经网络的定义多种多样。这里我们给出目前使得最广泛的一种定义，即**“神经网络是由具有适应性的简单单元组成的广泛并行互连的网络，它的组织能够模拟生物神经系统对真实世界物体所作出的交互反应”**。

1.1　神经元模型

神经网络中最基本的成分是**神经元** (neuron) 模型，即上述定义中的“简单、单元”。在生物神经网络中，每个神经元与其他神经元相连，当它“兴奋”时，就会向相连的神经元发送化学物质，从而改变这些神经元内的电位; 如果某神经元的电位超过了一个**“阈值”** (threshold)，那么它就会被激活，即“兴奋”起来，向其他神经元发送化学物质。把许多个这样的神经元按一定的层次结构连接起来，就得到了神经网络。

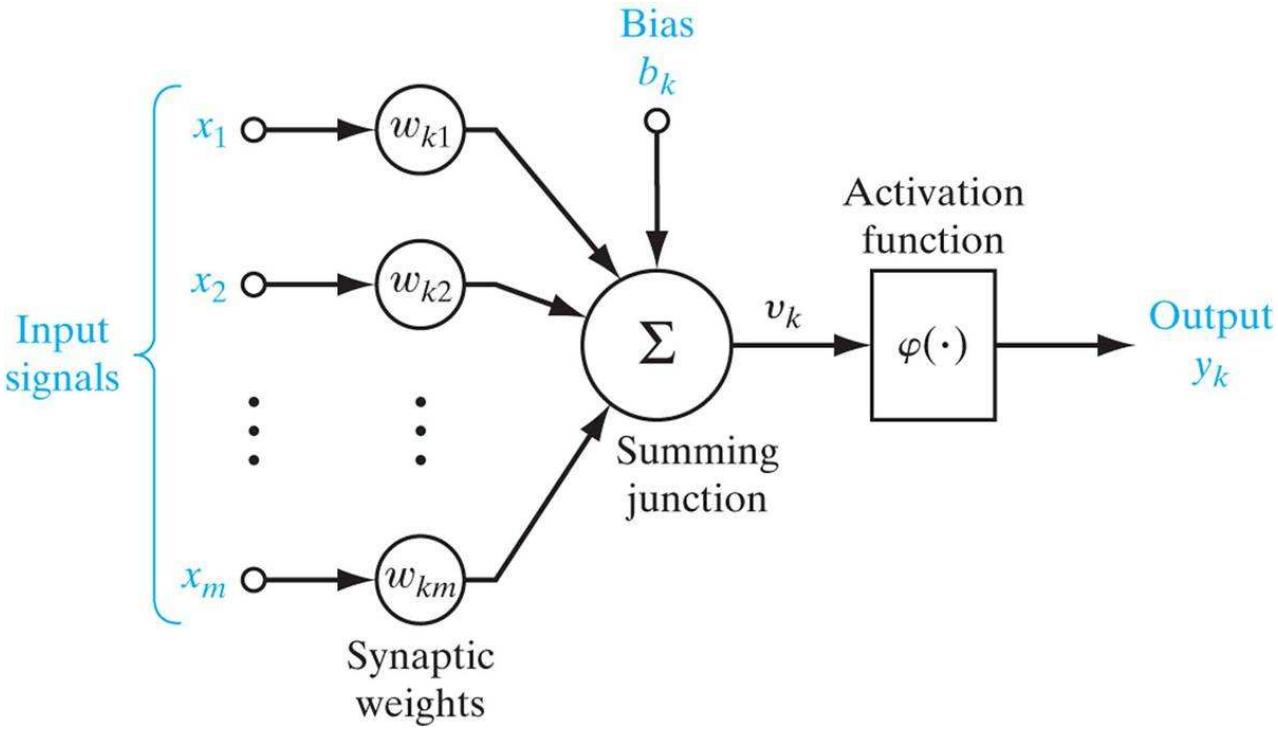

图 1: 神经元模型示意图

事实上，从计算机科学的角度看，我们可以先不考虑神经网络是否真的模拟了生物神经网络，只需将一个神经网络视为包含了许多参数的的数学模型。这个模型由若干个函数嵌套而得，例如将 $y_j = f\left(\sum_i w_i x_i - \theta_j\right)$ 相互代入。有效的神经网络学习算法大多以更加详细严格的数学证明为支撑。

1.2　单层感知机模型

感知机 (Perceptron)，也叫感知器，它是二分类的线性模型，在模式识别算法的历史上占有重要的地位。感知机的输入为样本的特征向量，输出为样本的类别，取和二值。具体方法为：给样本的每一维特征引入一个相乘的权重来表达每个特征的重要程度，然后对乘积求和后加上偏置项。将结果送入激活函数，利用激活函数的二值特性将样本划分为两类。

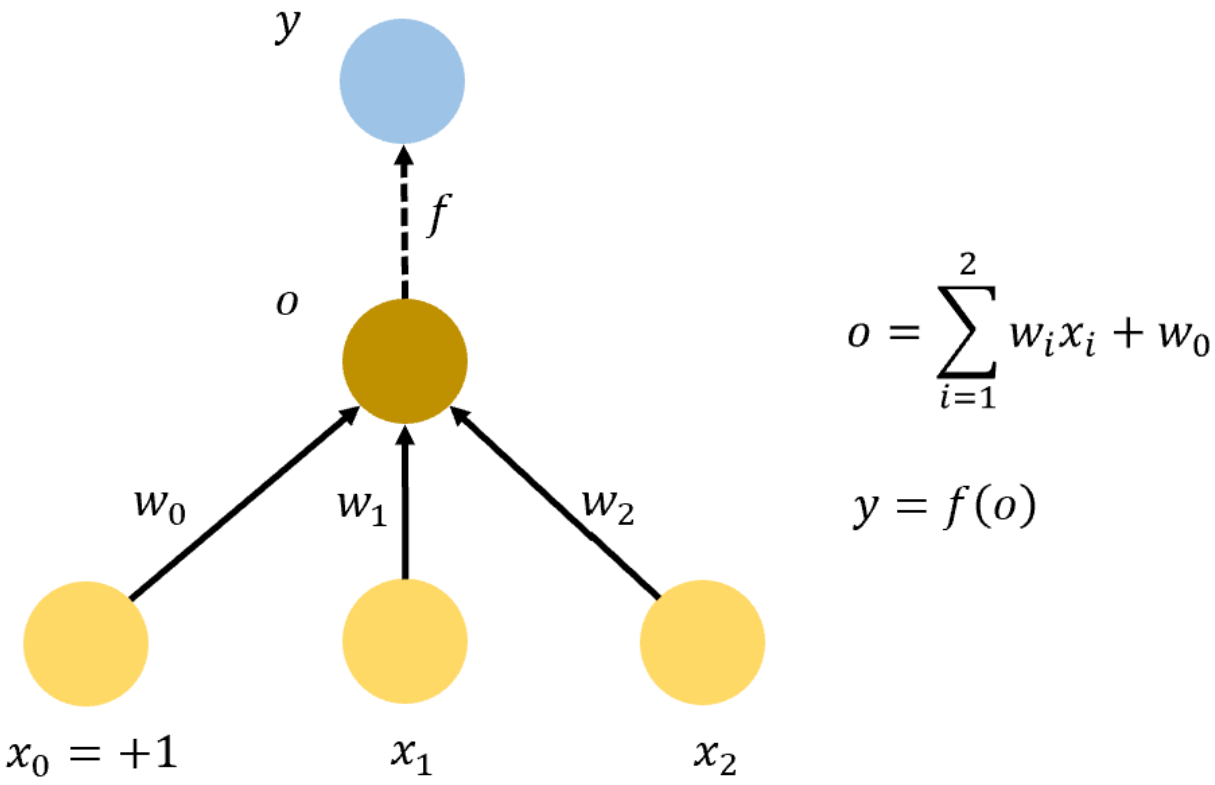

图 2: 感知机模型示意图

训练感知机的目标可以概括为：寻找合适的权值和偏置，使得激活函数能够将样本尽量准确地分成两类。需注意的是，感知机只有输出层神经元进行激活函数处理，即只拥有一层**功能神经元** (functional neuron)，因此其学习能力非常有限。事实上，对于与、或、非这类**线性可分** (linearly separable) 的问题，可以证明，一定存在一个线性超平面能将它们分开。也就是说，一定能求得适当的权向量 $\boldsymbol{w} = [w_1\ w_2\ \cdots\ w_{n+1}]$，使得感知机的学习过程**收敛** (converge)。而对于异或这样简单的非线性可分问题，感知机学习过程将会发生**振荡** (fuctuation)，$\boldsymbol{w}$ 难以稳定下来，不能求得合适解。

1.3　多层感知机模型

多层感知机 (multilayer perceptron, MLP)，也叫**深度前馈网络** (deep feedforward network)，是典型的深度学习模型。多层感知机的目标是近似某个函数 f^*。例如，对于分类器，$y = f^*(\boldsymbol{x})$ 将输入 $\boldsymbol{x}$ 映射到一个类别 y。前馈网络定义了一个映射 $\boldsymbol{y} = f(\boldsymbol{x}; \boldsymbol{\theta})$，并且学习参数 θ 的值，使它能够得到最佳的函数近似。

多层感知机被称作网络 (network)，是因为它们通常用许多不同函数复合在一起来表示。该模型与一个有向无环图相关联，而图描述了函数是如何复合在一起的。例如，我们有三个函数 $f^{(1)}$，$f^{(2)}$ 和 $f^{(3)}$ 连接在一个链上形成 $f(\boldsymbol{x}) = f^{(3)}(f^{(2)}(f^{(1)}(\boldsymbol{x})))$。这些链式结构是神经网络中最常用的结构。在这种情况下，$f^{(1)}$ 被称为网络的**第一层** (first layer)，$f^{(2)}$ 被称为网络的**第二层** (second layer)，以此类推。链的全长称为模型的**深度** (depth)。正是因为这个术语才出现了“深度学习”这个名字。多层感知机的最后一层被称为**输出层** (output layer)。

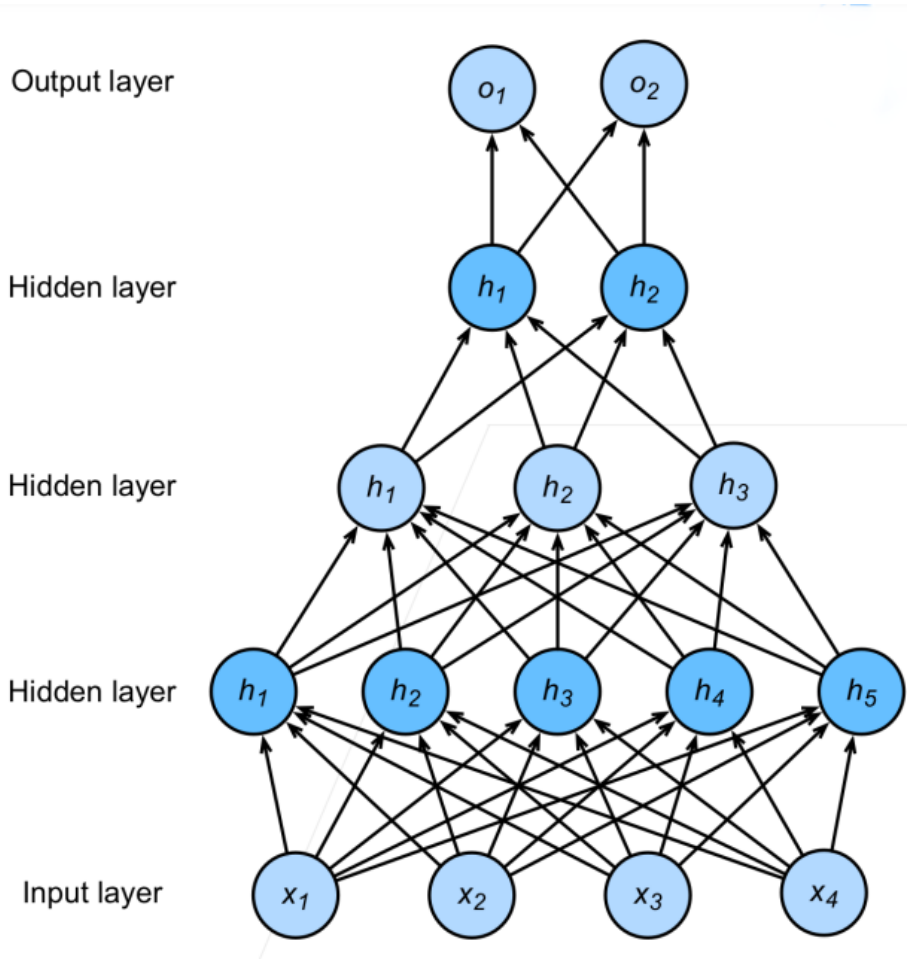

图 3: 多层感知机模型示意图

最后，这些网络被称为神经网络是因为它们或多或少地受到神经科学的启发。网络中的每个隐藏层通常都是向量值的。这些隐藏层的维数决定了模型的**宽度** (width)。向量的每个元素都可以被视为起到类似一个神经元的作用。除了将层想象成向量到向量的单个函数，我们也可以把层想象成由许多并行操作的**单元** (unit) 组成，每个单元表示一个向量到标量的函数。每个单元在某种意义上类似一个神经元，它接收的输入来源于许多其他的单元，并计算它自己的激活值。

使用多层向量值表示的想法来源于神经科学。用于计算这些表示的函数 $f^i(\boldsymbol{x})$ 的选择，也或多或少地受到神经科学观测的指引，这些观测是关于生物神经元计算功能的。然而，现代的神经网络研究受到更多的是来自许多数学和工程学科的指引，并且神经网络的目标并不是完美地给大脑建模。我们最好将前馈神经网络想成是为了实现统计泛化而设计出的函数近似机，它偶尔从我们了解的大脑中提取灵感，但并不是大脑功能的模型。

要解决非线性可分问题，就需要考虑使用多层感知机。例如图3中这个简单的两层感知机，就能解决异或问题。在图3中，输出层与输入层之之间的一层神经元，被称为**隐藏层** (hidden layer)，隐含层和输出层神经元都是拥有激活函数的功能神经元。

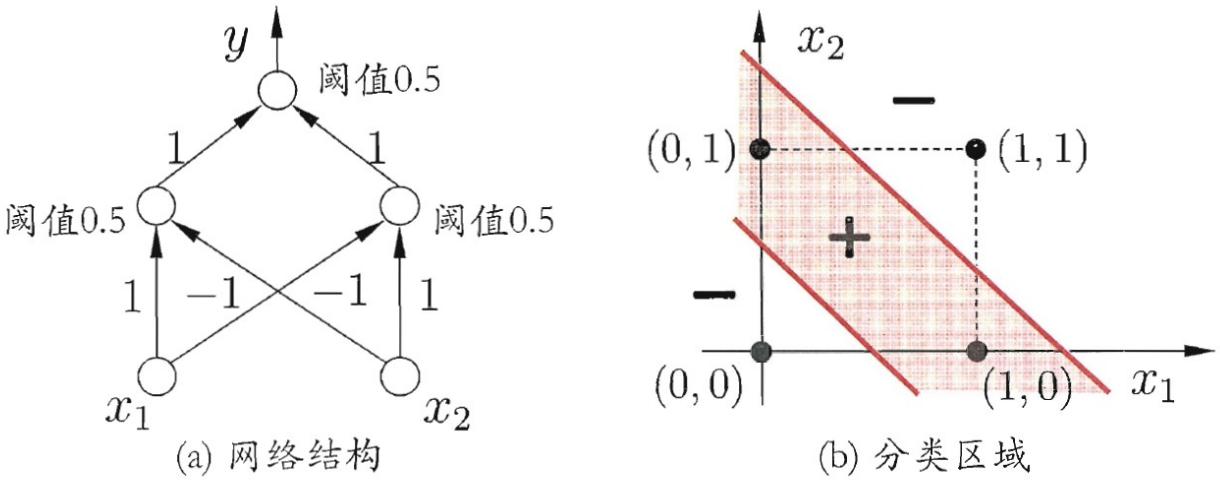

图 4: 能解决异或问题的两层感知机

第 2 章　神经网络的训练

2.1　链式求导法则

链式法则 (chain rule) 应用广泛，例如神经网络中用于训练的反向传播算法，就是以链式法则为基础演变的。链式法则指的是，两个函数组合起来的复合函数，导数等于内函数代入外函数值的导数，再乘以内函数的导数。例如，对于 $y = f(g(x))$，有链式求导法则如下：

$$\frac{dy}{dx} = \frac{dy}{du}\frac{du}{dx} = f'\left(g\left(x\right)\right)g'\left(x\right) \tag{1}$$

有了链式求导法则这一工具，下面我们将推导基于梯度的训练神经网络的算法。

2.2　基于梯度的优化方法

大多数深度学习算法都涉及某种形式的优化。优化指的是改变 $\boldsymbol{x}$ 以最小化或最大化某个函数 $f(x)$ 的任务。我们通常以最小化 $f(x)$ 指代大多数最优化问题。最大化可经由最小化算法最小化 $-f(x)$ 来实现。

我们把要最小化或最大化的函数称为**目标函数** (objective function) 或**准则** (criterion)。当我们对其进行最小化时，我们也把它称为**代价函数** (cost function)、**损失函数** (loss function) 或**误差函数** (error function)

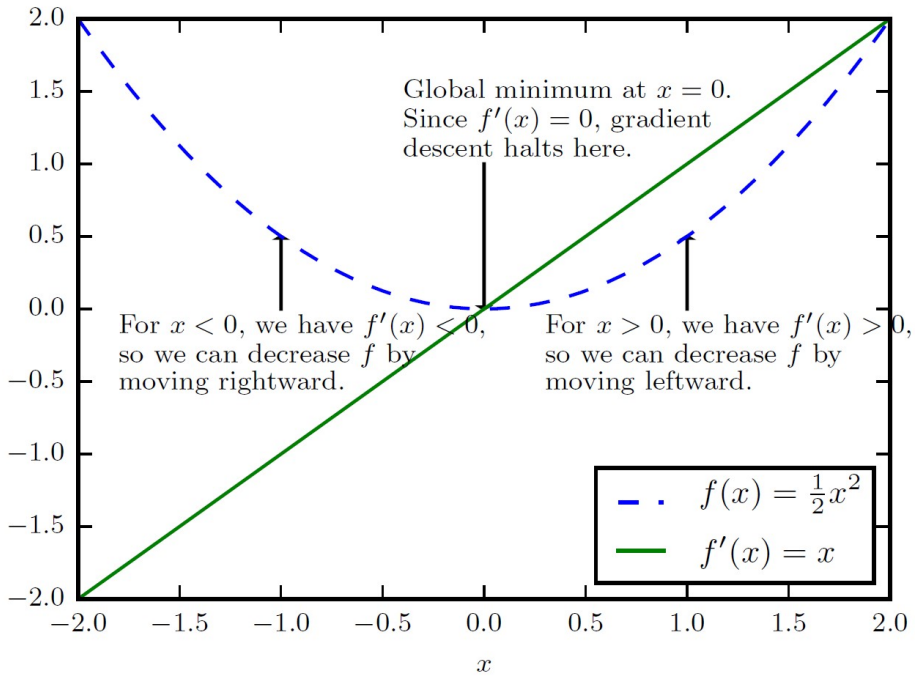

图 5: 梯度下降算法如何使用函数导数的示意图

假设我们有一个函数 $y = f(x)$，其中 x 和 y 是实数。这个函数的**导数** (derivative) 记为 $f'(x)$ 或 $\frac{dy}{dx}$。导数 $f'(x)$ 代表 $f(x)$ 在点 x 处的斜率。换句话说，它表明如何缩放输入的小变化才能在输出获得相应的变化：$f(x+\epsilon) \approx f(x) + \epsilon f'(x)$。

导数对于最小化一个函数很有用，因为它告诉我们如何更改 x 来略微地改善 y。例如，我们知道对于足够小的 ϵ 来说，$f(x - \epsilon sign(f'(x)))$ 是比 $f(x)$ 小的。因此我们可以将 x 往导数的反方向移动一小步来减小 $f(x)$。这种技术被称为**梯度下降** (gradient descent)。

当 $f'(x) = 0$，导数无法提供往哪个方向移动的信息。$f'(x) = 0$ 的点称为**临界点** (critical point) 或**驻点** (stationary point)。一个**局部极小点** (local minimum) 意味着这个点的 $f(x)$ 小于所有邻近点，因此不可能通过移动无穷小的步长来减小 $f(x)$。一个**局部极大点** (local maximum) 意味着这个点的 $f(x)$ 大于所有邻近点，因此不可能通过移动无穷小的步长来增大 $f(x)$。有些临界点既不是最小点也不是最大点。这些点被称为**鞍点** (saddle point)。见图6给出的各种临界点的例子。

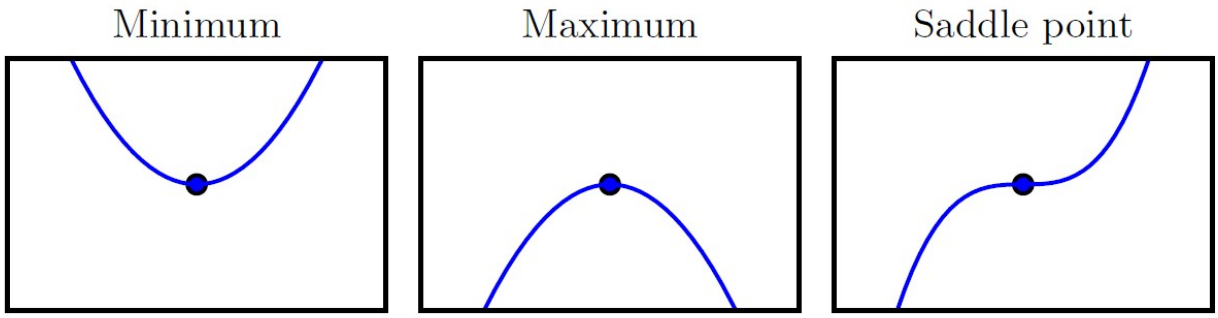

图 6: 临界点的类型

使 $f(x)$ 取得绝对的最小值（相对所有其他值）的点是**全局最小点** (global minimum)。函数可能只有一个全局最小点或存在多个全局最小点，还可能存在不是全局最优的局部极小点。在深度学习的背景下，我们要优化的函数可能含有许多不是最优的局部极小点，或者还有很多处于非常平坦的区域内的鞍点。尤其是当输入是多维的时候，所有这些都将使优化变得困难。因此，我们通常寻找使 $f(x)$ 非常小的点，但这在任何形式意义下并不一定是最小, 例如图7给出的例子.

我们经常最小化具有多维输入的函数：$f : R^n \to R$。为了使“最小化”的概念有意义，输出必须是一维的 (标量)。

针对具有多维输入的函数，我们需要用到**偏导数** (partial derivative) 的概念。偏导数

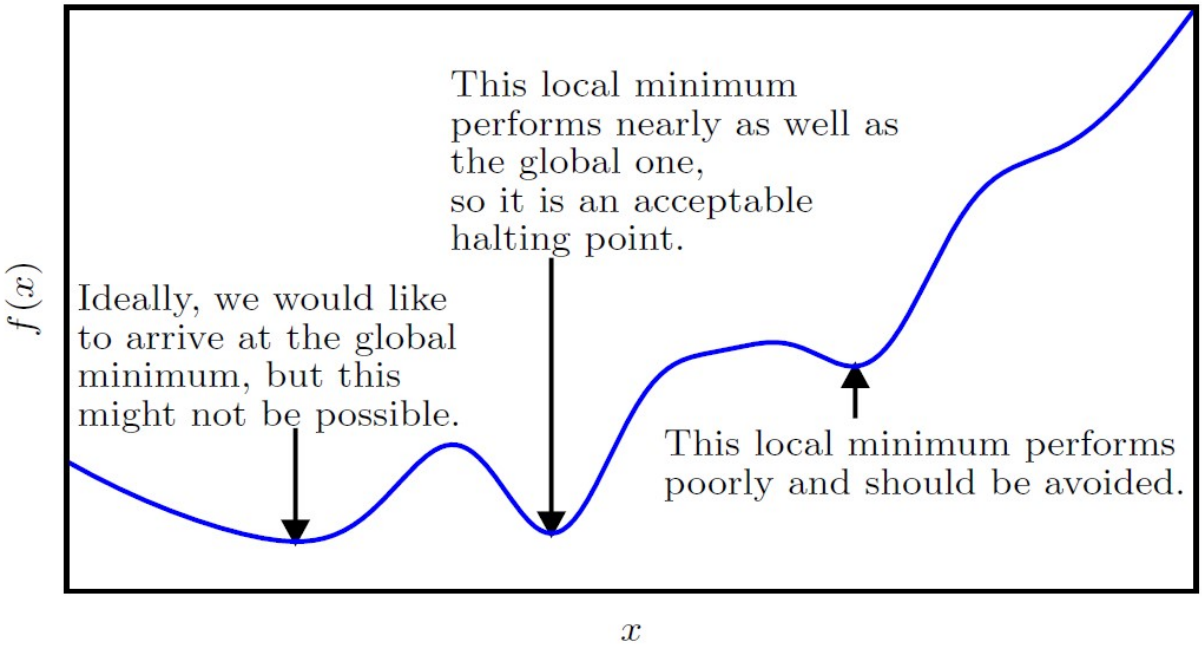

图 7: 优化过程陷入局部最优解的示意图

$\frac{\partial}{\partial x_i} f(\boldsymbol{x})$ 衡量点 $\boldsymbol{x}$ 处只有 x_i 增加时 $f(\boldsymbol{x})$ 如何变化。**梯度** (gradient) 是相对一个向量求导的导数：f 的导数是包含所有偏导数的向量，记为 $\nabla_x f(\boldsymbol{x})$。梯度的第 i 个元素是 f 关于 x_i 的偏导数。在多维情况下，临界点是梯度中所有元素都为零的点。

在 $\boldsymbol{u}$ 方向的方向导数 (directional derivative) 是函数 f 在 $\boldsymbol{u}$ 方向的斜率。换句话说，方向导数是函数 $f(\boldsymbol{x}+\alpha\boldsymbol{u})$ 关于 α 的导数（在 $\alpha=0$ 时取得）。使用链式法则，我们可以看到当 $\alpha=0$ 时，$f(\boldsymbol{x}+\alpha\boldsymbol{u})=\boldsymbol{u}^T\nabla_x f(\boldsymbol{x})$。

为了最小化 f，我们希望找到使 f 下降得最快的方向。计算方向导数：

$$\min_{u,u^\top u=1} \boldsymbol{u}^\top \nabla_x f(\boldsymbol{x}) = \min_{u,u^\top u=1} \|\boldsymbol{u}\|_2 \|\nabla_x f(\boldsymbol{x})\|_2 \cos\theta \tag{2}$$

其中 θ 是 u 与梯度的夹角。将 $\|\boldsymbol{u}\|_2=1$ 代入，并忽略与 u 无关的项，就能简化得到 $\min_u \cos\theta$。这在 u 与梯度方向相反时取得最小。换句话说，梯度向量指向上坡，负梯度向量指向下坡。我们在负梯度方向上移动可以减小 f。这被称为**最速下降法** (method of steepest descent) 或**梯度下降法** (gradient descent)。

最速下降建议新的点为:

$$\boldsymbol{x}' = \boldsymbol{x} - \epsilon \nabla_x f(\boldsymbol{x}) \tag{3}$$

其中 ϵ 为**学习率** (learning rate)，是一个确定步长大小的正标量。我们可以通过几种不同的方式选择 ϵ。普遍的方式是选择一个小常数。有时我们通过计算，选择使方向导

数消失的步长。还有一种方法是根据几个 ϵ 计算 $f(\boldsymbol{x}-\epsilon\nabla_x f(\boldsymbol{x}))$，并选择其中能产生最小目标函数值的 ϵ。这种策略被称为线搜索。

最速下降在梯度的每一个元素为零时收敛（或在实践中，很接近零时）。在某些情况下，我们也许能够避免运行该迭代算法，并通过解方程 $\nabla_x f(\boldsymbol{x})=0$ 直接跳到临界点。

虽然梯度下降被限制在连续空间中的优化问题，但不断向更好的情况移动一小步（即近似最佳的小移动）的一般概念可以推广到离散空间。

2.3　误差逆传播算法

多层网络的学习能力比单层感知机强得多。欲训练多层网络，简单感知机学习规则显然不够了, 需要更强大的学习算法。**误差逆传播** (error BackPropagation, BP) 算法就是其中最杰出的代表, 它是迄今最成功的神经网络学习算法。现实任务中使用神经网络时, 大多是在使用 BP 算法进行训练。值得指出的是，BP 算法不仅可用于多层前馈神经网络, 还可用于其他类型的神经网络。但通常说“BP 神经网络”时, 一般是指用 BP 算法训练的多层前馈神经网络。

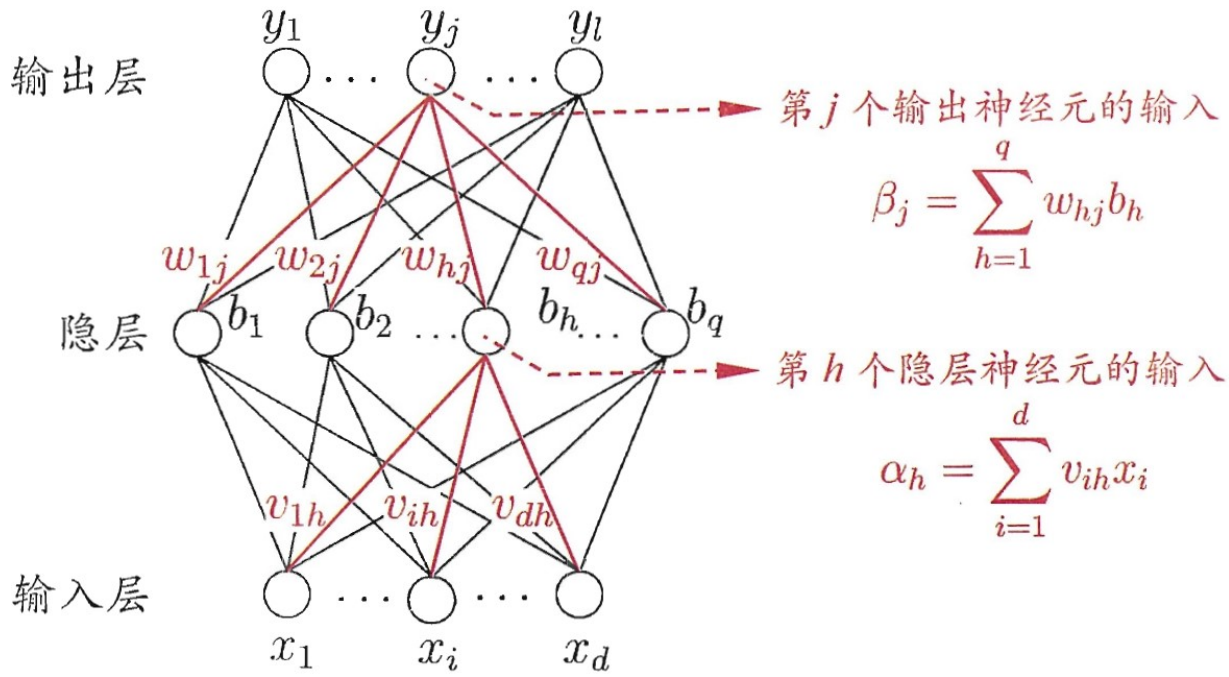

图 8: BP 网络及算法中的变量符号

给定训练集 $D=(\boldsymbol{x}_1,\boldsymbol{y}_1),(\boldsymbol{x}_2,\boldsymbol{y}_2),\cdots(\boldsymbol{x}_m,\boldsymbol{y}_m),\boldsymbol{x}_i\in R^d,\boldsymbol{y}_i\in R^l$，即输入示例由 d 个属性描述，输出 l 维实值向量。为便于讨论，图8给出了一个拥有 d 个输入神经元、l 个输出神经元、q 个隐层神经元的多层前馈网络结构，其中输出层第 j 个神经元的阈值用

θ_j 表示，隐层第 h 个神经元的阈值用 γ_h 表示。输入层第 i 个神经元与隐层第 h 个神经元之间的连接权为 v_{ih}，隐层第 h 个神经元与输出层第 j 个神经元之间的连接权为 w_{hj}。记隐层第 h 个神经元接收到的输入为 $\alpha_h = \sum_{i=1}^{d} v_{ih}x_i$，输出层第 j 个神经元接收到的输入为 $\beta_j = \sum_{h=1}^{q} w_hjb_h$，其中 b_h 为隐层第 h 个神经元的输出。假设隐层和输出层神经元都使用 $Sigmoid$ 函数。

对训练例 $(\boldsymbol{x}_k, \boldsymbol{y}_k)$，假定神经网络的输出为 $\hat{\boldsymbol{y}}_k = \left(\hat{y}_1^k, \hat{y}_2^k, \ldots, \hat{y}_l^k\right)$，即

$$\hat{y}_j^k = f\left(\beta_j - \theta_j\right) \tag{4}$$

则网络在 $(\boldsymbol{x}_k, \boldsymbol{y}_k)$ 上的均方误差为

$$E_k = \frac{1}{2}\sum_{j=1}^{l}\left(\hat{y}_j^k - y_j^k\right)^2 \tag{5}$$

图8的网络中有 $(d+l+1)q+l$ 个参数需确定：输入层到隐层的 $d \times q$ 个权值、隐层到输出层的 $q \times l$ 个权值、q 个隐层神经元的阈值、l 个输出层神经元的阈值。BP 是一个迭代学习算法，在迭代的每一轮中采用广义的感知机学习规则对参数进行更新估计，任意参数 v 的更新估计式为：

$$v \leftarrow v + \Delta v \tag{6}$$

下面我们以图8中隐层到输出层的连接权 w_{hj} 为例来进行推导。

BP 算法基于梯度下降策略，以目标的负梯度方向对参数进行调整。对均方误差 E_k，给定学习率 η，有：

$$\Delta w_{hj} = -\eta\frac{\partial E_k}{\partial w_{hj}} \tag{7}$$

注意到 w_{hj} 先影响到第 j 个输出层神经元的输入值 β_j，再影响到其输出值 $\hat{y}_j^k$，然后影响到 E_k，有：

$$\frac{\partial E_k}{\partial w_{hj}} = \frac{\partial E_k}{\partial \hat{y}_j^k}\cdot\frac{\partial \hat{y}_j^k}{\partial \beta_j}\cdot\frac{\partial \beta_j}{\partial w_{hj}} \tag{8}$$

根据 β_j 的定义，显然有

$$\frac{\partial \beta_j}{\partial w_{hj}} = b_h \tag{9}$$

由于 $Sigmoid$ 函数有一个很好的性质：

$$f'(x) = f(x)(1 - f(x)) \tag{10}$$

于是：

$$\begin{aligned} g_j &= -\frac{\partial E_k}{\partial \hat{y}_j^k} \cdot \frac{\partial \hat{y}_j^k}{\partial \beta_j} \\ &= -\left(\hat{y}_j^k - y_j^k\right) f'\left(\beta_j - \theta_j\right) \\ &= \hat{y}_j^k\left(1 - \hat{y}_j^k\right)\left(y_j^k - \hat{y}_j^k\right) \end{aligned} \tag{11}$$

将式 (9)、(11) 代入 (8)，就得到了 BP 算法中关于 w_{hj} 的更新公式

$$\Delta w_{hj} = \eta g_j b_h \tag{12}$$

类似可得：

$$\begin{aligned} \Delta\theta_j &= -\eta g_j \\ \Delta v_{ih} &= \eta e_h x_i \\ \Delta\gamma_h &= -\eta e_h \end{aligned} \tag{13}$$

其中：

$$\begin{aligned} e_h &= -\frac{\partial E_k}{\partial b_h} \cdot \frac{\partial b_h}{\partial \alpha_h} \\ &= -\sum_{j=1}^{l} \frac{\partial E_k}{\partial \beta_j} \cdot \frac{\partial \beta_j}{\partial b_h} f'\left(\alpha_h - \gamma_h\right) \\ &= \sum_{j=1}^{l} w_{hj} g_j f'\left(\alpha_h - \gamma_h\right) \\ &= b_h\left(1 - b_h\right) \sum_{j=1}^{l} w_{hj} g_j. \end{aligned} \tag{14}$$

误差逆传播算法的算法流程如下：

算法 1 误差逆传播算法

输入： 训练集 $D = (\boldsymbol{x}_1, \boldsymbol{y}_1), (\boldsymbol{x}_2, \boldsymbol{y}_2), \cdots (\boldsymbol{x}_m, \boldsymbol{y}_m), \boldsymbol{x}_i \in R^d, \boldsymbol{y}_i \in R^l$；学习率 η

输出： 连接权与阈值确定的多层前馈神经网络

1: 在 $(0, 1)$ 范围内随机初始化网络中所有连接权和阈值
2: repeat
3: **for** $all(\boldsymbol{x}_k, \boldsymbol{y}_k)$ **do**
4:　　根据当前参数和式 (4) 计算当前样本的输出 $\hat{\boldsymbol{y}}_k$
5:　　根据式 (11) 计算输出层神经元的梯度项 g_j
6:　　根据式 (14) 计算隐层神经元的梯度项 e_h
7:　　根据式 (12)、(13) 更新连接权 w_{hj}, v_{ih} 与阈值 θ_j, γ_h
8: **end for**

第 3 章　神经网络的种类及应用

3.1　RBF 网络

RBF(Radial Basis Function，径向基函数) 网络是一种单隐层前馈神经网络，它使用径向基函数作为隐层神经元激活函数，而输出层则是对隐层神经元输出的线性组合。假定输入为 d 维向量 x，输出为实值，则 RBF 网络可表示为

$$\varphi(\boldsymbol{x}) = \sum_{i=1}^{q} w_i \rho(\boldsymbol{x}, \boldsymbol{c}_i) \tag{15}$$

其中 q 为隐层神经元个数，$\boldsymbol{c}_i$ 和 w_i 分别是第 i 个隐层神经元所对应的中心和权重，$\rho(\boldsymbol{x}, \boldsymbol{c}_i)$ 是径向基函数，这是某种沿径向对称的标量函数，通常定义为样本 $\boldsymbol{x}$ 到数据中心 $\boldsymbol{c}_i$ 之间欧氏距离的单调函数。常用的高斯径向基函数形如：

$$\rho(\boldsymbol{x}, \boldsymbol{c}_i) = e^{-\beta_i \|\boldsymbol{x} - \boldsymbol{c}_i\|^2} \tag{16}$$

通常采用两步过程来训练 RBF 网络。第一步，确定神经元中心 $\boldsymbol{c}_i$，常用的方式包括随机采样、聚类等；第二步，利用 BP 算法等来确定参数 w_i 和 β_i

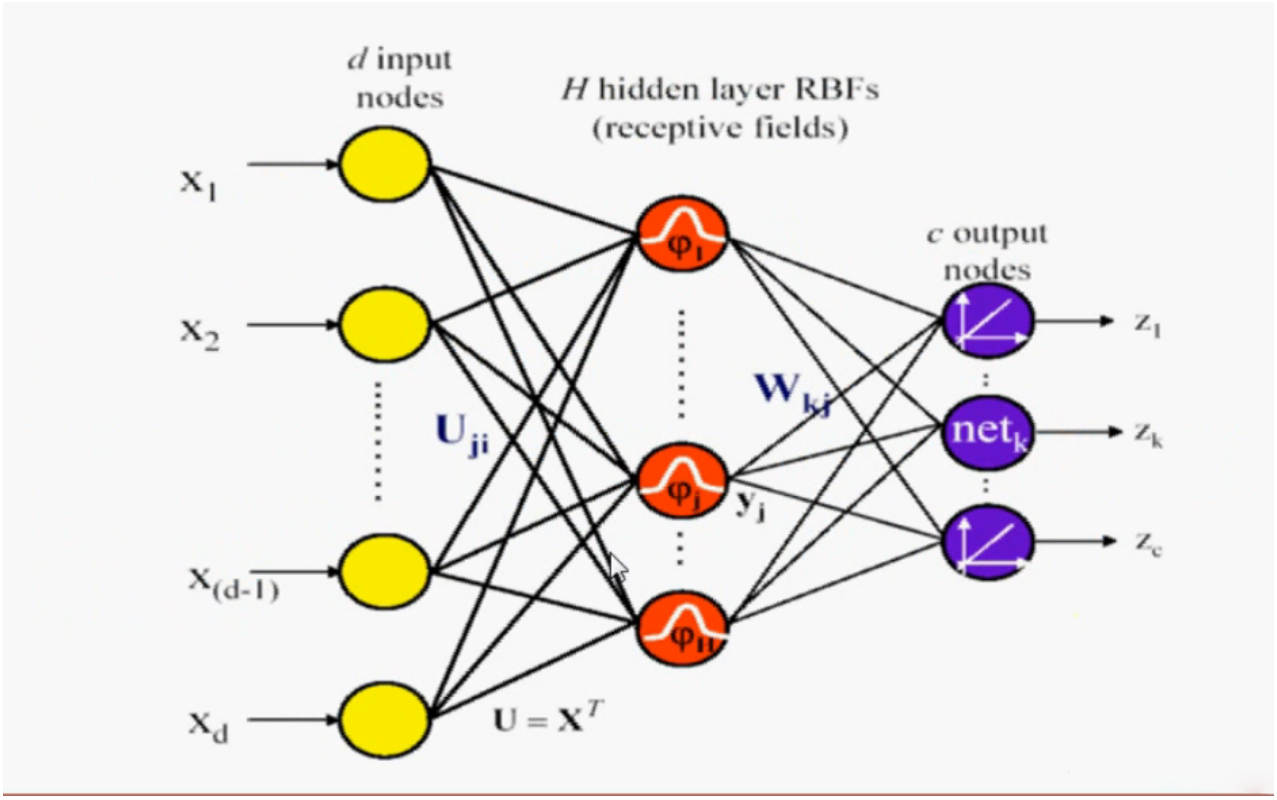

图 9: RBF 网络示意图

3.2　ART 网络

竞争型学习 (competitive learning) 是神经网络中一种常用的无监督学习策略，在使用该策略时，网络的输出神经元相互竞争，每一时刻仅有一个竞争获胜的神经元被激活，其他神经元的状态被抑制。这种机制亦称“胜者通吃”(winner–take-all) 原则。

ART(Adaptive Resonance Theory, 自适应谐振理论) 网络是竞争型学习的重要代表。该网络由比较层、识别层、识别阈值和重置模块构成。其中，比较层负责接收输入样本，并将其传递给识别层神经元。识别层每个神经元对应一个模式类，神经元数目可在训练过程中动态增长以增加新的模式类。

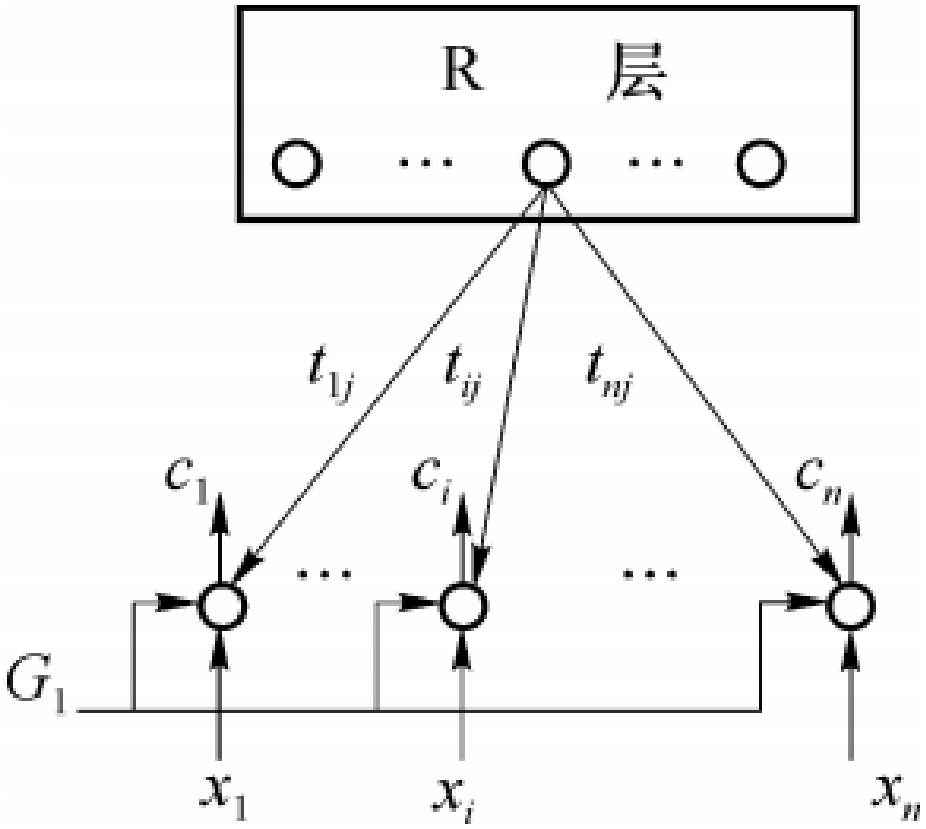

图 10: ART 网络示意图

在接收到比较层的输入信号后，识别层神经元之间相互竞争以产生获胜神经元，竞争的最简单方式是，计算输入向量与每个识别层神经元所对应的模式类的代表向量之间的距离，距离最小者胜。获胜神经元将向其他识别层神经元发送信号，抑制其激。若输入向量与获胜神经元所对应的代表向量之间的相似度大于识别阈值，则当前输入样本将被归为该代表向量所属类别，同时，网络连接权将会更新，使得以后在接收到相似输入

样本时该模式类会计算出更大的相似度，从而使该获胜神经元有更大可能获胜；若相似度不大于识别阈值，则重置模块将在识别层增设一个新的神经元，其代表向量就设置为当前输入向量。

显然, 识别阈值对 ART 网络的性能有重要影响，当识别阈值较高时，输入样本将会被分成比较多、比较精细的模式类，而如果识别阅值较低，则会产生比较少、比较粗略的模式类。

ART 比较好地缓解了竞争型学习中的“可塑性-稳定性窘境”(stability-plasticity dilemma)，可塑性是指神经网络要有学习新知识的能力，而稳定性则是指神经网络在学习新知识时要保持对旧知识的记忆，这就使得 ART 网络具有个很重要的优点：可进行增量学习 (incremental learning) 或在线学习 (online learning)。

早期的 ART 网络只能处理布尔型输入数据，此后 ART 发展成了一个算法族，包括能处理实值输入的 ART2 网络、结合模糊处理的 FuzzyART 网络，以及可进行监督学习的 ARTMAP 网络等。

3.3　SOM 网络

SOM(Self-Organizing Map, 自组织映射) 网络是一种竞争学习型的无监督神经网络，它能将高维输入数据映射到低维空间 (通常为二维)，同时保持输入数据在高维空间的拓扑结构，即将高维空间中相似的样本点映射到网络输出层中的邻近神经元.

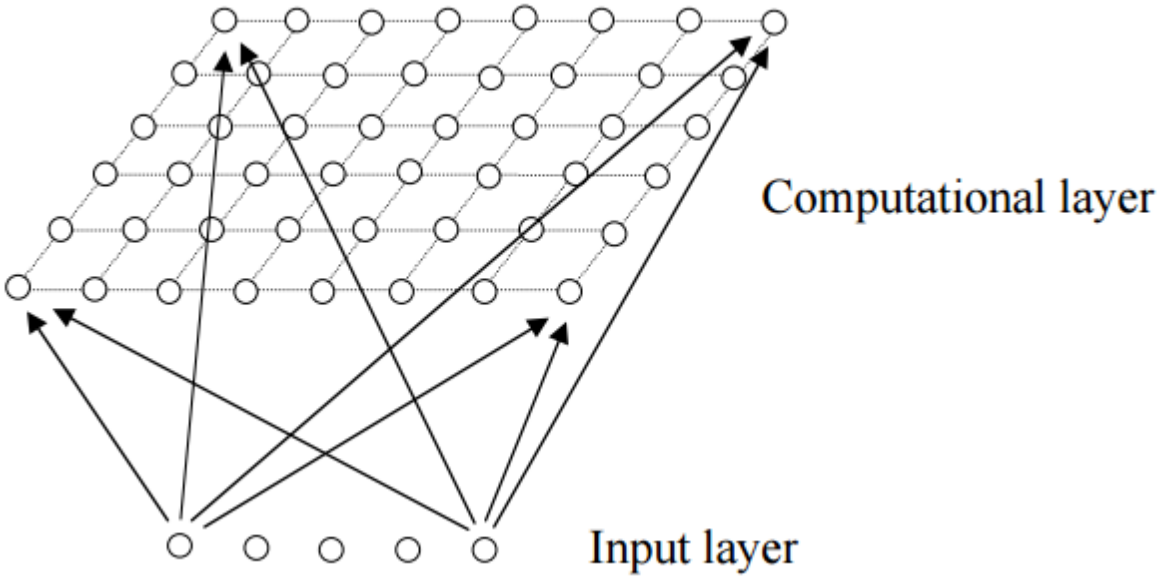

图 11: SOM 网络示意图

如图11所示，SOM 网络中的输出层神经元以矩阵方式排列在二维空间中，每个神经元都拥有一个权向量，网络在接收输入向量后，将会确定输出层获胜神经元，它决定了该输入向量在低维空间中的位置。SOM 的训练目标就是为每个输出层神经元找到合适的权向量，以达到保持拓扑结构的目的。

SOM 的训练过程很简单：在接收到一个训练样本后，每个输出层神经元会计算该样本与自身携带的权向量之间的距离，距离最近的神经元成为竞争获胜者，称为最佳匹配单元 (best matching unit)。然后，最佳匹配单元及其邻近神经元的权向量将被调整，以使得这些权向量与当前输入样本的距离缩小。这个过程不断迭代，直至收敛。

3.4　卷积神经网络

卷积网络 (convolutional network)，也叫做卷积神经网络 (con-volutional neural network, CNN)，是一种专门用来处理具有类似网格结构的数据的神经网络。例如时间序列数据可以认为是在时间轴上有规律地采样形成的一维网格）和图像数据（可以看作是二维的像素网格）。卷积网络在诸多应用领域都表现优异。“卷积神经网络”一词表明该网络使用了卷积 (convolution) 这种数学运算。卷积是一种特殊的线性运算。卷积网络是指那些至少在网络的一层中使用卷积运算来替代一般的矩阵乘法运算的神经网络。

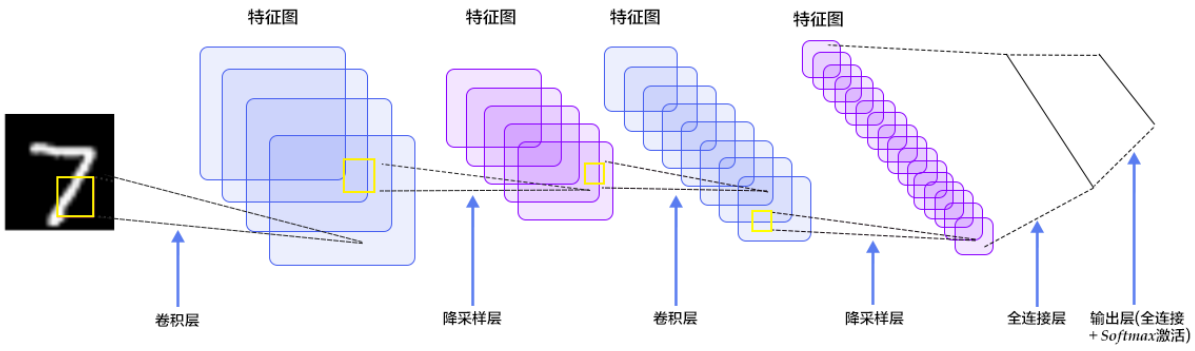

图 12: 卷积神经网络示意图

3.5　循环神经网络

循环神经网络 (recurrent neural networkm RNN) 是一类用于处理序列数据的神经网络。就像卷积网络是专门用于处理网格化数据的神经网络，循环神经网络是专门用于处理序列的神经网络。正如卷积网络可以很容易地扩展到具有很大宽度和高度的图像，以

及处理大小可变的图像，循环网络可以扩展到更长的序列（比不基于序列的特化网络长得多）。大多数循环网络也能处理可变长度的序列。

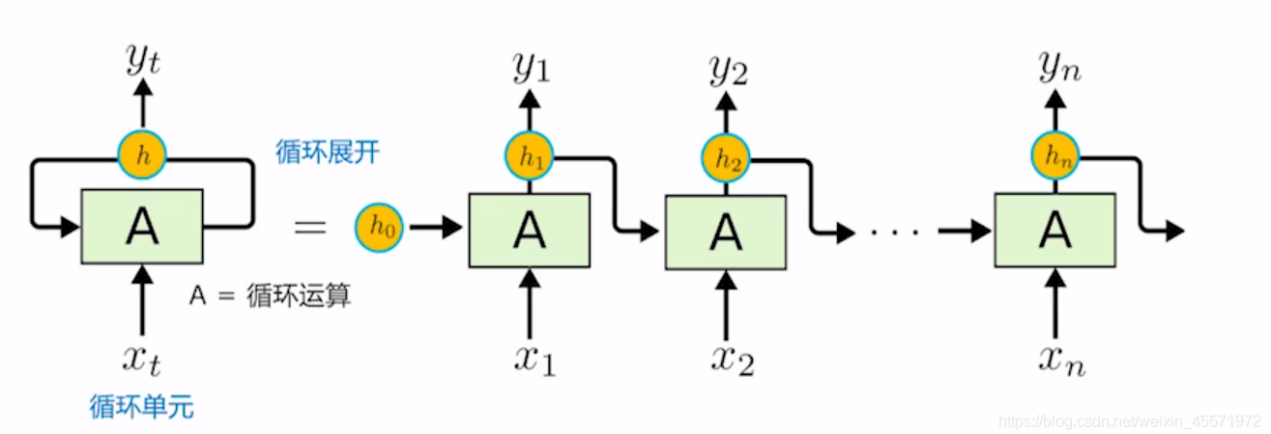

图 13: 循环神经网络示意图

第 4 章　实训结果与分析

4.1　任务一的结果与原因分析

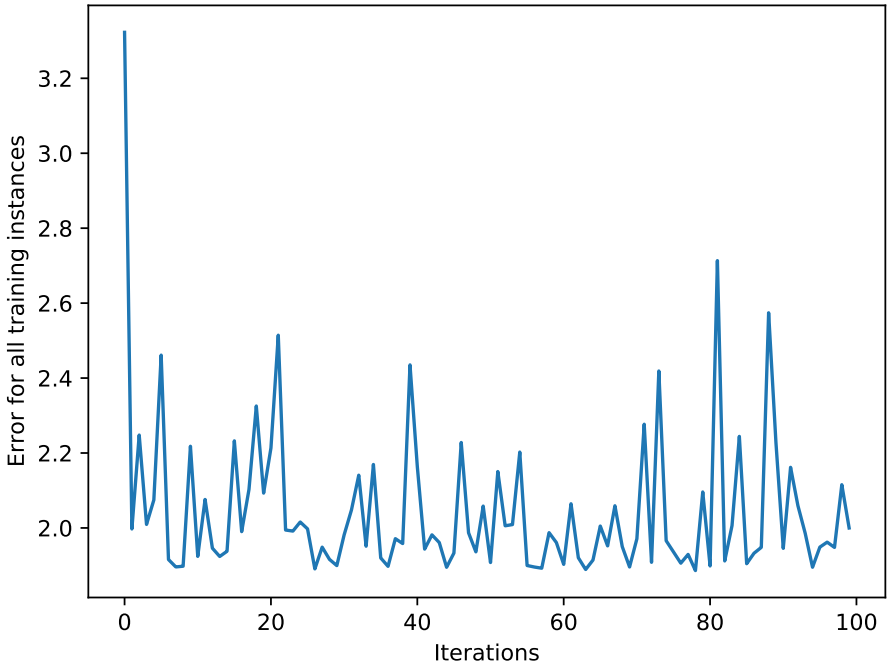

图 14: 任务一基础代码的损失函数

任务一的主要内容是通过 Python 自编程，实现单层感知机算法。在基础代码中，训练集由 8 个二维数据点和 8 个相应的标签值组成。其中标签值取值为 0 或 1. 初始的学习率为 0.1，迭代轮数为 10000 轮，每迭代 100 轮记录一次损失函数的值，并绘制损失函数的图像。运行基础代码，最终损失函数的如图14所示。

显然，损失函数始终保持在一定范围内上下浮动，说明这一训练过程是不收敛的。我们猜测，**训练集提供的八个数据点是线性不可分的**。因此，我们绘制原始数据的散点分布图进行探索性分析：

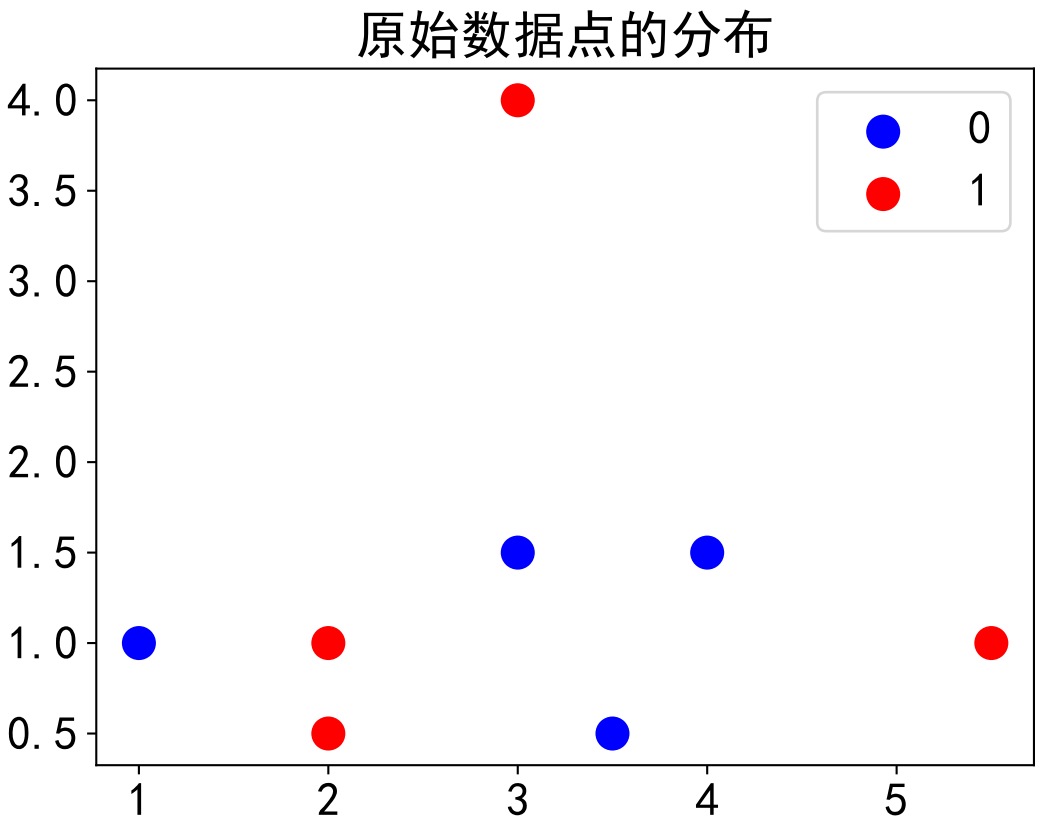

图 15: 训练集的散点分布图

很明显，原始数据按分类无明显聚集特征，而要想让单层感知机收敛，原始数据集必须线性可分。经过分析，我们认为数据点 $(1,1)$ 与 $(5.5,1)$ 为噪声数据。若删去噪声数据点，则剩余部分将称为线性可分数据集。因此，在原始代码的基础上，我们对数据集进行处理后重新运行程序，得到的损失函数如图16所示：

可见，当数据集为线性可分数据集时，损失函数成功收敛至 0。

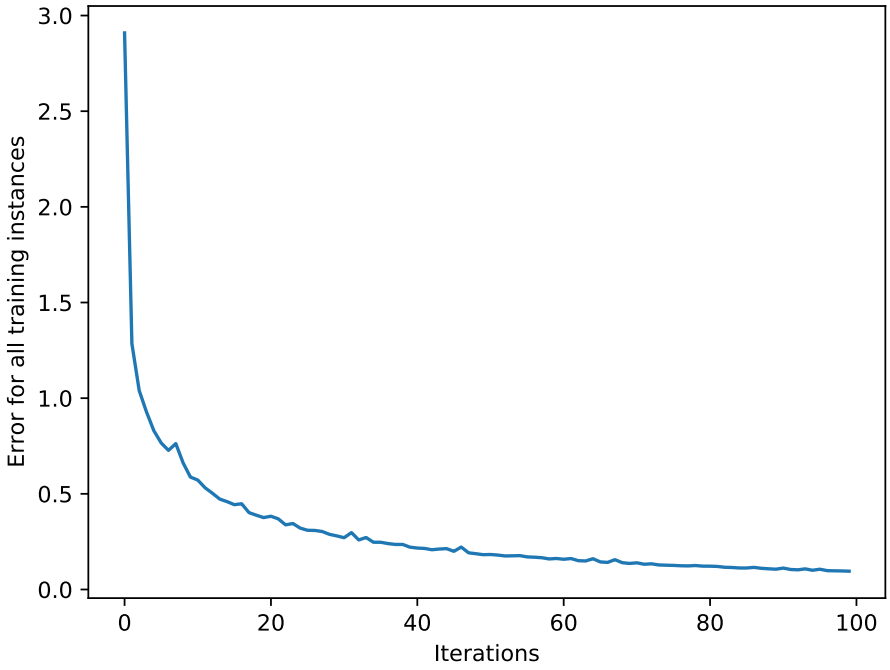

图 16: 任务一调整后的损失函数图像

4.2　任务二的实现

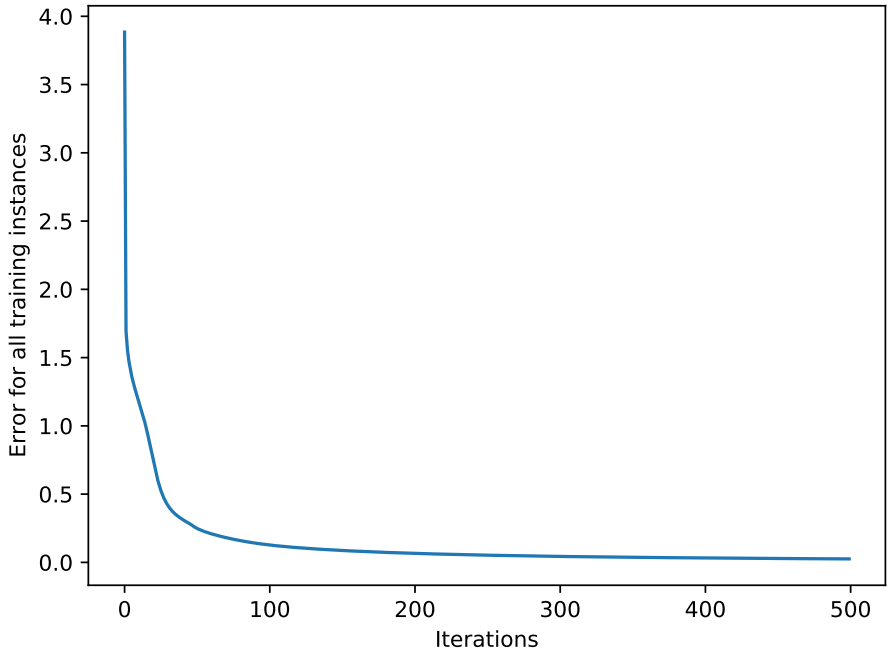

图 17: 任务二的损失函数图像

任务二要求我们添加一层神经网络，也就是在任务一的基础上构建多层感知机。根据**万能近似定理** (universal approximation theorem)，一个前馈神经网络如果具有线性输出层和至少一层具有任何一种“挤压”性质的激活函数（例如 logistic sigmoid 激活函数）的隐藏层，只要给予网络足够数量的隐藏单元，它可以以任意的精度来近似任何从一个有限维空间到另一个有限维空间的 Borel 可测函数。因此，通过添加隐藏层并调整隐藏层节点个数以及训练参数，一定能将原始数据集精确划分。

我们在原始代码的基础上，进行了一些改动（具体的数学原理以及计算过程见 2.3 误差逆传播算法）。最终，我们添加了一层含有 20 个节点的隐藏层，并设置学习率为 0.1，迭代轮数为 50000. 最终的损失函数图像如图17所示。

4.3　任务三的实现

任务三要求我们重新设计神经网络，将偏置作为权重融入神经网络中。

以单层感知机模型为例，对于原始模型 $y = \boldsymbol{w}^T\boldsymbol{x} + b$，令数据集 $\tilde{\boldsymbol{x}} = (1, x_1, x_2)$，权重 $\tilde{\boldsymbol{w}} = (w_0, w_1, w_2)$，则方程等价于 $y = \tilde{\boldsymbol{w}}^T\tilde{\boldsymbol{x}}$。经过等价，偏置融入了权重向量中，偏置与权重的得到了统一，编程实现得到了简化（具体编程过程见附录中任务三的代码）。输出的损失函数图像基本与任务一、二中的一致，说明编程过程没有出现问题。

第 5 章　思考与结论

5.1　实训中的问题与思考

本次实训最大的难点在于，我们需要通过自编程构建神经网络，而不能借助目前已有的神经网络框架编程。在自编程的过程中，训练神经网络的全部细节均需要自行完成，这对公式推导和编程能力提出了很高的要求。

在代码调试的过程中，本人遇到最大的问题主要来自问题二的多层感知机的训练。由于权重的初值为随机初始化，每次绘制出的损失函数图像往往不同，且只有少部分为收敛的图像。通过对学习率和迭代轮数的不断调整，最终本人得到了较为完整的实验结果。

5.2　实训总结

本次实训中我们学习了神经网络的基本原理，训练了使用 Python 自编程实现神经网络模型。在编程实践的过程中，我掌握了构建神经网络模型的基本能力。

在编程实践的过程中，本人遇到了许多的难以解决的小问题。在查阅资料以及对问题的反思整理的过程中，本人培养了解决问题的方法与能力。

本次实训任务量适中，难度较大。经过本次实训，本人在加深理论知识学习的同时，强化了基于 Python 自编程的神经网络实践，收获了许多宝贵的理论知识与实践经验。

参考文献

[1] 周志华. 机器学习 [M]. 清华大学出版社，2016.

[2] Ian Goodfellow, Yoshua Bengio, Aaron Courville. 深度学习 [M]. 人民邮电出版社，2017.

[3] TravelingLight77. 链式法则总结, https://blog.csdn.net/weixin_44010756/article/details/109662171 [EB/OL].2020-11-12.

[4] SongGu1996. 感知机（Perceptron）,https://blog.csdn.net/SongGu1996/article/details/100849768 [EB/OL].2020-07-23.

附录

任务一的代码

```
# %%
# 任务1：此处函数不收敛，调试寻找原因
# 1. 检查数据，如果数据不包含特征，神经网络无法收敛。
# 2: 调整学习率，检查收敛速率
# 3: 找到合适的收敛参数
import numpy as np
import matplotlib.pyplot as plt


class NeuralNetwork:
def __init__(self, learning_rate):
# 随机产生权重与偏置
# 学习步长由用户设置
self.weights = np.array([np.random.randn(), np.random.randn()])
self.bias = np.random.randn()
self.learning_rate = learning_rate

@staticmethod
def _sigmoid(x):
# 激活函数——sigmoid
return 1 / (1 + np.exp(-x))

def _sigmoid_deriv(self, x):
# 激活函数的导数
return self._sigmoid(x) * (1 - self._sigmoid(x))

def predict(self, input_vector):
# f1=x*w+b
layer_1 = input_vector @ self.weights + self.bias
# f2=sigmoid(f1)
```

```
31 layer_2 = self._sigmoid(layer_1)
32 prediction = layer_2
33
34 return prediction
35
36 def _compute_gradients(self, input_vector, target):
37 # 根据链式法则计算梯度
38 # f1=x*w+b
39 layer_1 = input_vector @ self.weights + self.bias
40 # f2=sigmoid(f1)
41 layer_2 = self._sigmoid(layer_1)
42
43 prediction = layer_2
44
45 derror_dprediction = 2 * (prediction - target)
46 dprediction_dlayer1 = self._sigmoid_deriv(layer_1)
47 dlayer1_dbias = 1
48 dlayer1_dweights = (0 * self.weights) + (1 * input_vector)
49
50 derror_dbias = (derror_dprediction * dprediction_dlayer1 * dlayer1_dbias)
51 derror_dweights = (derror_dprediction * dprediction_dlayer1 * dlayer1_dweights)
52
53 return derror_dbias, derror_dweights
54
55 def _update_parameters(self, derror_dbias, derror_dweights):
56 # 更新偏置
57 self.bias = self.bias - (derror_dbias * self.learning_rate)
58 # 更新权重
59 self.weights = self.weights - (derror_dweights * self.learning_rate)
60
61 def train(self, input_vectors, targets, iterations):
62 cumulative_errors = []
63 for current_iteration in range(iterations):
64 # 随机选择数据
```

```
random_data_index = np.random.randint(len(input_vectors))
input_vector = input_vectors[random_data_index]
target = targets[random_data_index]
# 计算梯度并更新权重
derror_dbias, derror_dweights = self._compute_gradients(input_vector, target)

self._update_parameters(derror_dbias, derror_dweights)

# 测量累计误差
if current_iteration % 100 == 0:
cumulative_error = 0
# 考虑所有数据
for data_instance_index in range(len(input_vectors)):
data_point = input_vectors[data_instance_index]
target = targets[data_instance_index]

prediction = self.predict(data_point)
error = np.square(prediction - target)

cumulative_error = cumulative_error + error
cumulative_errors.append(cumulative_error)

return cumulative_errors

# 原始数据集
# input_vectors = np.array([[3, 1.5], [2, 1], [4, 1.5], [3, 4], [3.5, 0.5],
    [2, 0.5], [5.5, 1], [1, 1]])
# targets = np.array([0, 1, 0, 1, 0, 1, 1, 0])

# 修正数据集
input_vectors = np.array([[3, 1.5], [2, 1], [4, 1.5], [3, 4], [3.5, 0.5], [2,
    0.5]])
targets = np.array([0, 1, 0, 1, 0, 1])

```

```
# 学习率
learning_rate = 0.1

# 迭代轮数
iteration = 10000

# 实例化
neural_network = NeuralNetwork(learning_rate)
training_error = neural_network.train(input_vectors, targets, iteration)
# 绘制训练误差折线图
plt.plot(training_error)
plt.xlabel("Iterations")
plt.ylabel("Error for all training instances")
plt.show()
# plt.savefig("cumulative_error.png")
```

任务二的代码

```
# 任务2：添加一层神经网络，并评估训练效果，须做神经网络设计和理论推导
import numpy as np
import matplotlib.pyplot as plt

class NeuralNetwork:
# 初始化函数
def __init__(self, inputnodes, hiddennodes, outputnodes, learning_rate):
self.innodes = inputnodes
self.hinodes = hiddennodes
self.outnodes = outputnodes
self.learning_rate = learning_rate

# 随机产生权重与偏置
self.winhi = np.random.randn(self.hinodes, self.innodes)
self.whiout = np.random.randn(self.outnodes, self.hinodes)
```

```
16
17  @staticmethod
18  def _sigmoid(x):
19  # 激活函数——sigmoid
20  return 1 / (1 + np.exp(-x))
21
22  def train(self, input_vectors, targets_vectors, iterations):
23  cumulative_errors = []
24  # 把输入列表转化为np型的2维
25  inputs = np.array(input_vectors, ndmin=2).T
26  targets = np.array(targets_vectors, ndmin=2).T
27  for current_iteration in range(iterations):
28  # 计算隐含层输出
29  hidden_inputs = np.dot(self.winhi, inputs)
30  hidden_outputs = self._sigmoid(hidden_inputs)
31  # 计算输出层输出
32  final_inputs = np.dot(self.whiout, hidden_outputs)
33  final_outputs = self._sigmoid(final_inputs)
34  # 计算输出层的误差
35  output_errors = targets.T - final_outputs
36  hidden_errors = np.dot(self.whiout.T, output_errors)
37  # 更新链接权重
38  self.whiout += self.learning_rate * np.dot((output_errors * final_outputs * (1
        - final_outputs)),
39  np.transpose(hidden_outputs))
40  # final_outputs已经经过了sigmoid运算，所以可以直接调用。
41  # 对输入层和隐含层之间的权重进行更新
42  self.winhi += self.learning_rate * np.dot((hidden_errors * hidden_outputs * (1
        - hidden_outputs)),
43  np.transpose(inputs))
44  # 测量累计误差
45  if current_iteration % 100 == 0:
46  cumulative_error = 0
47  # 考虑所有数据
```

```
48 for data_instance_index in range(len(input_vectors)):
49 data_point = input_vectors[data_instance_index]
50 target = targets_vectors[data_instance_index]
51
52 prediction = self.predict(data_point)
53 error = np.square(prediction - target)
54
55 cumulative_error = cumulative_error + error
56 cumulative_errors.append(cumulative_error)
57 return cumulative_errors
58
59 # 输入的是一个数列例如[1,2,3,4]
60 def predict(self, inputs):
61 hidden_inputs = np.dot(self.winhi, inputs)
62 # 经过隐含层的激活函数
63 hidden_outputs = self._sigmoid(hidden_inputs)
64 # 信号从隐含层传递到输出层
65 final_inputs = np.dot(self.whiout, hidden_outputs)
66 # 信号经过输出层的激活函数
67 final_outputs = self._sigmoid(final_inputs)
68 return final_outputs
69
70 input_vectors = np.array([[3, 1.5], [2, 1], [4, 1.5], [3, 4], [3.5, 0.5], [2,
    0.5], [5.5, 1], [1, 1]])
71 targets = np.array([0, 1, 0, 1, 0, 1, 1, 0])
72 learning_rate = 0.1
73 iteration = 50000
74 # 实例化
75 neural_network = NeuralNetwork(inputnodes=2, hiddennodes=20, outputnodes=1,
    learning_rate=learning_rate)
76 training_error = neural_network.train(input_vectors, targets,
    iterations=iteration)
77 # 绘制训练误差折线图
78 plt.plot(training_error)
```

```
plt.xlabel("Iterations")
plt.ylabel("Error for all training instances")
plt.show()
```

任务三的代码

```
# 任务3（可选）：将偏置融入权重，重新设计设计网络代码。
import numpy as np
import matplotlib.pyplot as plt


class NeuralNetwork:
def __init__(self, learning_rate):
# 随机产生权重与偏置
# 学习步长由用户设置
self.weights = np.array([np.random.randn(), np.random.randn(),
    np.random.randn()])
self.learning_rate = learning_rate

@staticmethod
def _sigmoid(x):
# 激活函数——sigmoid
return 1 / (1 + np.exp(-x))

def _sigmoid_deriv(self, x):
# 激活函数的导数
return self._sigmoid(x) * (1 - self._sigmoid(x))

def predict(self, input_vector):
# f1=x*w+b
layer_1 = input_vector @ self.weights
# f2=sigmoid(f1)
layer_2 = self._sigmoid(layer_1)
```

```
27 prediction = layer_2
28
29 return prediction
30
31 def _compute_gradients(self, input_vector, target):
32 # 根据链式法则计算梯度
33 # f1=x*w+b
34 layer_1 = input_vector @ self.weights
35 # f2=sigmoid(f1)
36 layer_2 = self._sigmoid(layer_1)
37
38 prediction = layer_2
39
40 derror_dprediction = 2 * (prediction - target)
41 dprediction_dlayer1 = self._sigmoid_deriv(layer_1)
42 dlayer1_dweights = (0 * self.weights) + (1 * input_vector)
43
44 derror_dweights = (derror_dprediction * dprediction_dlayer1 * dlayer1_dweights)
45
46 return derror_dweights
47
48 def _update_parameters(self, derror_dweights):
49 # 更新权重
50 self.weights = self.weights - (derror_dweights * self.learning_rate)
51
52 def train(self, input_vectors, targets, iterations):
53 cumulative_errors = []
54 input_vectors = np.insert(input_vectors, input_vectors.shape[1],
      np.ones(len(input_vectors)), axis=1)
55 for current_iteration in range(iterations):
56 # 随机选择数据
57 random_data_index = np.random.randint(len(input_vectors))
58 input_vector = input_vectors[random_data_index]
59 target = targets[random_data_index]
```

```
60 # 计算梯度并更新权重
61 derror_dweights = self._compute_gradients(input_vector, target)
62
63 self._update_parameters(derror_dweights)
64
65 # 测量累计误差
66 if current_iteration % 100 == 0:
67 cumulative_error = 0
68 # 考虑所有数据
69 for data_instance_index in range(len(input_vectors)):
70 data_point = input_vectors[data_instance_index]
71 target = targets[data_instance_index]
72
73 prediction = self.predict(data_point)
74 error = np.square(prediction - target)
75
76 cumulative_error = cumulative_error + error
77 cumulative_errors.append(cumulative_error)
78
79 return cumulative_errors
80
81 input_vectors = np.array([[3, 1.5], [2, 1], [4, 1.5], [3, 4], [3.5, 0.5], [2,
      0.5], [5.5, 1], [1, 1]])
82 targets = np.array([0, 1, 0, 1, 0, 1, 1, 0])
83 learning_rate = 0.1
84 iteration = 10000
85 # 实例化
86 neural_network = NeuralNetwork(learning_rate)
87 training_error = neural_network.train(input_vectors, targets, iteration)
88 # 绘制训练误差折线图
89 plt.plot(training_error)
90 plt.xlabel("Iterations")
91 plt.ylabel("Error for all training instances")
92 plt.show()
```

```
93 # plt.savefig("cumulative_error.png")
```