最优化方法
多层神经网络的训练问题

西安交通大学, 数学与统计学院, 强基数学 002

吴天阳 [a], 马煜璇 [b], 白鹏楠 [c]

2204210460[a], 2204220461[b], 2204421343[c]

2022 年 4 月 18 日

目录

1 神经网络介绍

随着神经科学、认知科学的发展, 我们逐渐发现人类的智能行为和大脑的活动有关, 人类的大脑是一个能够产生意识、思想和情感的复杂器官. 受到人类大脑大脑的启发, 早期的神经科学家构造了一种模仿人脑神经系统的数学模型, 称为**人工神经网络** (Artificial Neural Network), 简称神经网络. 在机器学习领域, 神经网络是指由很多**人工神经元**构成的网络结构, 每个人工神经元之间的连接强度是可学习参数.

如图1是一个人工神经网络, 它由三层组成, 分别为**输入层**、**隐藏层**、**输出层**. 每一个圆圈结点代表一个人工神经元, 每一个箭头代表神经元之间从输入到输出方向的连接, 并包含一个权重值, 表示两个神经元之间的连接强度. 每一个各人工神经元可以视为一种有多个参数确定的函数, 先将来自前一层节点的信息经过相应的权重综合计算, 再输入到一个激活函数中, 从而获得当前节点的活性值.

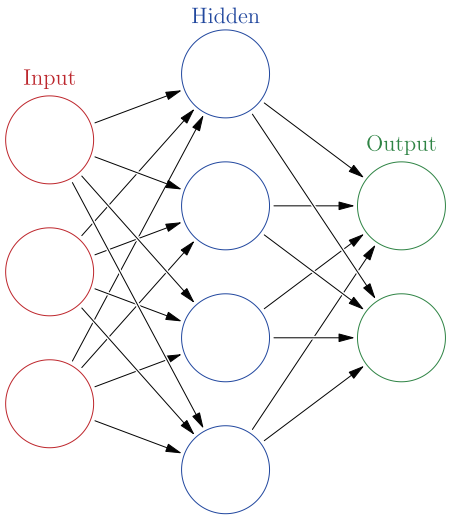


图 1: 神经网络结构[1]

1.1 模型

从数学的角度来看神经网络, 其本质就是一种由大量参数确定的复杂的非线性函数. 对于一个机器学习任务, 我们可以认为是给出一些样本, 每个**样本**由输入向量 $\boldsymbol{x}$ 和其对应的输出值 y 所决定, 记为 $(\boldsymbol{x}, y)$, 所有样本构成一个**样本空间** $\mathcal{X} \times \mathcal{Y}$, 即对于任意一个样本都有 $(\boldsymbol{x}, y) \in \mathcal{X} \times \mathcal{Y}$, 假定存在一个未知的**真实映射函数** $g : \mathcal{X} \to \mathcal{Y}$, 使得对于每一个样本 $(\boldsymbol{x}, y)$ 都有 $g(\boldsymbol{x}) = y$, 机器学习的目标就是去近似真实映射函数 $g(\boldsymbol{x})$.

由于不知道真实映射函数 $g(\boldsymbol{x})$ 的具体形式, 于是只能假设一个函数集合 $\mathcal{F}$, 称为**假设空间** (Hypothesis Space), 通过大量样本对网络的训练, 从中选取一个理想的**假设** (Hypothesis) $f^* \in \mathcal{F}$. 假设空间一般是一个参数化的函数族

$$\mathcal{F} = \{f(\boldsymbol{x}; \theta) : \theta \in \mathbb{R}^D\}, \tag{1.1}$$

其中 $f(\boldsymbol{x}; \theta)$ 是参数为 θ 的函数, 也称为**模型** (Model), D 为参数的数量.

1.1.1 线性模型

线性模型的假设空间为一个参数化的线性函数族, 即

$$f(\boldsymbol{x}; \theta) = f(\boldsymbol{x} : \boldsymbol{w}, b) = \boldsymbol{w}^T \boldsymbol{x} + b, \tag{1.2}$$

其中参数 θ 包含权重向量 $\boldsymbol{w}$ 和偏置 b, 下文中的感知器算法就是一种线性模型.

[1]图片来源: https://en.wikipedia.org/wiki/File:Colored_neural_network.svg

1.1.2 非线性模型

广义的非线性模型可以写成多个非线性**基函数** $\phi(\boldsymbol{x})$ 的线性组合

$$f(\boldsymbol{x};\theta) = \boldsymbol{w}^T\phi(\boldsymbol{x}) + b, \tag{1.3}$$

其中 $\phi(\boldsymbol{x}) = [\phi_1(\boldsymbol{x}), \phi_2(\boldsymbol{x}), \cdots, \phi_K(\boldsymbol{x})]^T$ 为 K 个非线性基函数组成的向量, 参数 θ 包含了权重向量 $\boldsymbol{w}$ 和偏置 b.

如果 $\phi(\boldsymbol{x})$ 本身也为可学习基函数, 即

$$\phi_k(\boldsymbol{x}) = h(\boldsymbol{w}_k^T\phi'(x) + b_k), \quad k = 1, 2, \cdots, K, \tag{1.4}$$

其中 $h(\cdot)$ 为非线性函数, $\phi'(\boldsymbol{x})$ 为另一组基函数, $\boldsymbol{w}_k$ 和 b_k 可学习参数, 则 $f(\boldsymbol{x};\theta)$ 就等价于**神经网络模型**.

如果我们将第 s 层中从上至下第 k 个人工神经元视为函数 $\phi_k^{(s)}(\boldsymbol{x})$, 那么一层中所有神经元构成函数向量 $\phi^{(s)}(\boldsymbol{x})$, 由于当前层的神经元 $\phi^{(s)}(\boldsymbol{x})$ 是由上一层的所有神经元 $\phi^{(s-1)}(\boldsymbol{x})$ 通过权重矩阵 $\boldsymbol{w}^{(s)}$ 计算加上偏置 $b^{(s)}$, 再经过一个非线性激活函数 $h(\cdot)$ 得到的, 即 $\phi^{(s)}(\boldsymbol{x}) = h(\boldsymbol{w}^{(s)T}\phi^{(s-1)}(\boldsymbol{x}) + b^{(1)})$ 满足式 (1.4), 所以可以看出神经网络的本质就是从输入空间到输出空间的非线性参数化函数.

1.2 神经网络的类型

感知器是最早具有机器学习思想的神经网络, 也是最简单的人工神经网络, 我们将从感知器算法为例, 阐述机器学习的基本概念和符号定义. 神经网络模型分为主要的三种, 分别为**前馈型神经网络**、**卷积神经网络和循环神经网络**, 还有一种更一般性的网络: **图网络**, 由于时间所限, 我们只研究了前馈型神经网络模型的理论基础及实际应用.

2 感知器算法

感知器 (Perceptron) 是 Frank Rosenblatt 于 1957 年所发明的, 它可以被视为一种最简单的前馈神经网络.

感知器是一种**二元线性分类器 (Binary Linear Classifier)**, 大概地来说, 感知器的输入为样本的**特征向量**, 输出为样本的类别, 以 $+1$ 表示正类, -1 表示负类, 利用**随机梯度下降法**对**损失函数**进行极小化, 求出可将样本进行划分的**分离超平面**.

2.1 感知器模型

我们通过一个例子引入: 预测学生能否通过课程的问题, 对于一名学生, 根据一批与这门课程相关的数据, 预测他是否可以通过某一门课程. 与这门课程相关的每一种数据称为**特征 (Feature)**, 我们用 n 维向量 $\boldsymbol{x}$ 进行存储, $\boldsymbol{x}$ 中的每一维对应一种特征, 称 $\boldsymbol{x}$ 为**特征向量 (Feature Vector)**. 比如: 第一个维度 x_1 为出勤率, 第二个维度 x_2 为自主学习时长等等. 我们将这些维度的数据加权求和, 得到一个综合得分 $\sum_{i=1}^n w_i x_i$, 如果该得分高于某个标准 b, 则预测他可以通过, 否则不能通过. 现在假设我们有大量该课程的历史学生

数据, 现在要求根据这些历史数据 (训练集), 建立上述分类模型, 以预测当前某一名学生 (测试集) 是否能够通过该课程.

我们称 $\boldsymbol{x} \in \mathbb{R}^n$ 为一个样本的**特征向量 (Feature Vector)**, **权重向量 (Weight Vector)** 为 $\boldsymbol{w} \in \mathbb{R}^n$, **偏置 (Bias)** 为 b. 于是我们可以将上述文字表述可以转化为以下数学表达式, 若 $\boldsymbol{w}^T\boldsymbol{x} \geqslant b$ 则分为正类 $+1$(可以通过), 若 $\boldsymbol{w}^T\boldsymbol{x} < b$ 则分为负类 -1(不能通过), 记符号函数

$$\text{sign}(x) = \begin{cases} 1, & x \geqslant 0, \\ -1, & x < 0. \end{cases} \tag{2.1}$$

则分类函数 $h(\boldsymbol{x}) = \text{sign}\left(\boldsymbol{w}^T\boldsymbol{x} - b\right)$, 若记 $x_0 = 1,\ w_0 = -b$, 则上式也可统一为

$$\begin{aligned} h(\boldsymbol{x}) &= \text{sign}\left(\sum_{i=1}^{n} w_i x_i - b\right) = \text{sign}\left(\sum_{i=1}^{n} w_i x_i + w_0 x_0\right) \\ &= \text{sign}\left(\sum_{i=0}^{n} w_i x_i\right) = \text{sign}(\widetilde{\boldsymbol{w}}^T\widetilde{\boldsymbol{x}}), \end{aligned} \tag{2.2}$$

其中 $\widetilde{\boldsymbol{w}} = (-b, w_1, \cdots, w_n)^T$, $\widetilde{\boldsymbol{x}} = (1, x_1, \cdots, x_n)^T$, 我们称 $\widetilde{\boldsymbol{w}}$ 为**增广权重向量**, $\widetilde{\boldsymbol{x}}$ 为**增广特征向量**.

下面给出感知器模型的几何解释, 由于线性方程 $\boldsymbol{w}^T\boldsymbol{x} + b = 0$ 对应特征空间 $\mathbb{R}^n$ 中的一个超平面 S, 其中 $\boldsymbol{w}$ 为 S 的法向量, b 为 S 的截距, 于是 S 将空间 $\mathbb{R}^n$ 分为两个部分, 位于两个部分中的点 (即样本的特征向量) 分为正、负两类, 我们将 S 称为**分离超平面 (Seperating Hyperplane)**. 当维度 $n = 2$ 时, **图2** 给出了一种情况下的分离超平面.

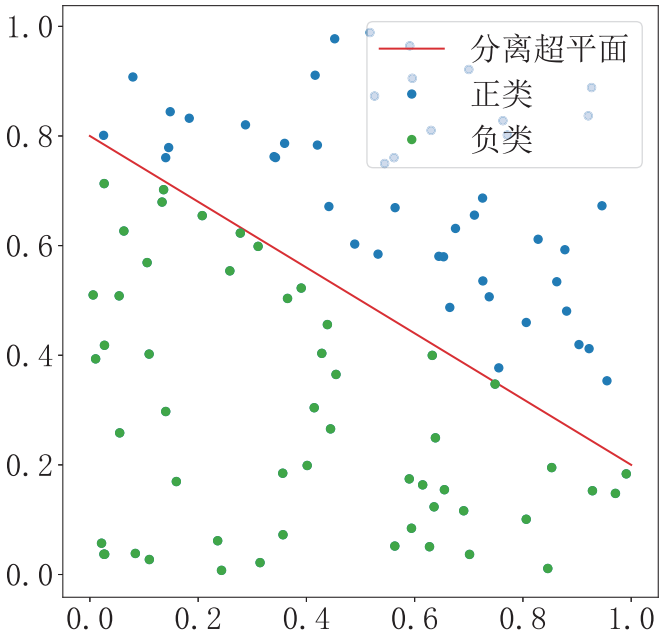


图 2: 分离超平面

定义 1 (线性可分性)**.** 给定一个数据集 $T \subset \mathbb{R}^n$, 如果存在超平面 $S : \boldsymbol{w}^T\boldsymbol{x} + b = 0$, 能够将数据集的正负样本点完全正确的划分到超平面的两侧, 则称该数据集线性可分, 否则线性不可分.

2.2 学习准则

为了描述一个样本集合, 我们定义二元组 $(\boldsymbol{x}, y)$ 为一个**样本 (Example)**, 也称为**实例 (Instance)**, $\boldsymbol{x} \in \mathbb{R}^n$ 称为该样本的**特征向量 (Feature Vector)**, 每一维表示一个**特征 (Feature)**, $y \in \{-1, 1\}$ 为该样本的**标签 (Label)**(在这里就是该样本对应的分类). 给定如下的一个大小为 N 的线性可分的样本集 T. (比如**图2** 中的样本集就是一个线性可分的)

$$\mathcal{T} = \{(\boldsymbol{x}^{(1)}, y^{(1)}), (\boldsymbol{x}^{(2)}, y^{(2)}), \cdots, (\boldsymbol{x}^{(\boldsymbol{N})}, y^{(N)}) : \boldsymbol{x}_i \in \mathbb{R}^n, y_i \in \{-1, 1\}, 1 \leqslant i \leqslant N\}, \quad (2.3)$$

我们将根据**训练集 (Training Set)**(样本集的子集, 用于训练模型) 寻找一个超平面 S : $\boldsymbol{w}^T\boldsymbol{x} + b = 0$, 即学习参数 $\boldsymbol{w}, b$ 的过程称为**训练 (Training)**, 使其将 T 中的数据完全正确的分离开. 因此我们需要制定一个与 $\boldsymbol{w}$ 和 b 有关的优化函数, 通过极小化该函数, 从而达到我们的目的, 这里的优化函数在机器学习中被称为**损失函数 (Loss Function)**, 其作用是量化模型预测与真实标签之间的差异, 在整个训练集 $\mathcal{T}$ 上, 我们还可以计算**经验风险 (Empirical Risk)**, 即全体损失函数的平均值. 一种很自然的选取方法是全体误分类点的总数, 但是这样构造的函数并不是连续的, 不易于优化. 感知器所选取的损失函数为全体误分类点到超平面 S 的距离之和.

我们知道对于任意的 $\boldsymbol{x} \in \mathbb{R}^n$, $\boldsymbol{x}$ 到 S 的距离可以表示为

$$d = \frac{|\boldsymbol{w}^T\boldsymbol{x} + b|}{||\boldsymbol{w}||}, \quad (2.4)$$

其中 $||\boldsymbol{w}|| = \sqrt{w_1^2 + w_2^2 + \cdots + w_n^2}$, 即为 2-范数.

对于超平面 S 的误分类点 $\boldsymbol{x}^{(i)}$, 有如下 2 种情况, 并且可以将它们归为一种:

$$\left.\begin{array}{ll} \text{if} & y^{(i)} = -1, \quad \boldsymbol{w}^T\boldsymbol{x}^{(i)} + b > 0 \\ \text{if} & y^{(i)} = 1, \quad \boldsymbol{w}^T\boldsymbol{x}^{(i)} + b < 0 \end{array}\right\} \; y^{(i)}(\boldsymbol{w}^T\boldsymbol{x}^{(i)} + b) < 0. \quad (2.5)$$

则 $|\boldsymbol{w}^T\boldsymbol{x}^{(i)} + b| = -y^{(i)}(\boldsymbol{w}^T\boldsymbol{x}^{(i)} + b)$, 因此 $\boldsymbol{x}^{(i)}$ 到 S 的距离为

$$d_i = -\frac{y^{(i)}(\boldsymbol{w}^T\boldsymbol{x}^{(i)} + b)}{||\boldsymbol{w}||}, \quad (2.6)$$

不考虑系数 $\frac{1}{||\boldsymbol{w}||}$, 得感知器损失函数与经验风险函数

$$\mathcal{L}(\boldsymbol{x}, y; \boldsymbol{w}, b) = \max(0, -y(\boldsymbol{w}^T\boldsymbol{x} + b)), \quad (2.7)$$

$$\mathcal{R}(\boldsymbol{w}, b) = \frac{1}{N}\sum_{i=1}^{N}\mathcal{L}(\boldsymbol{x}^{(\boldsymbol{i})}, y^{(i)}; \boldsymbol{w}, b). \quad (2.8)$$

这里不考虑 $\frac{1}{||\boldsymbol{w}||}$ 的原因, 我们认为有如下 2 个原因:

- 误分类主要由 $-y_i(\boldsymbol{w}^T\boldsymbol{x} + b)$ 的正负号来判断, 而 $\frac{1}{||\boldsymbol{w}||}$ 对符号没有影响.
- 便于计算偏导.

2.3 梯度下降法

根据损失函数 (2.7) 式可知, 可以将感知器学习算法转化为一下的无约束极小化问题

$$\min_{\boldsymbol{w}\in\mathbb{R}^n,\ b\in\mathbb{R}} \mathcal{L}(\boldsymbol{x}, y; \boldsymbol{w}, b) = \max(0, -y(\boldsymbol{w}^T\boldsymbol{x} + b)). \tag{2.9}$$

考虑使用最速梯度下降法作为该模型的优化算法, 即取优化函数的负梯度方向作为每次一迭代更新参数的下降方向

$$\frac{\partial \mathcal{L}(\boldsymbol{x}, y; \boldsymbol{w}, b)}{\partial \boldsymbol{w}} = \begin{cases} -y\boldsymbol{x}, & (\boldsymbol{x}, y) \in \mathcal{M}, \\ 0, & \texttt{otherwise}. \end{cases} \tag{2.10}$$

$$\frac{\partial \mathcal{L}(\boldsymbol{x}, y; \boldsymbol{w}, b)}{\partial b} = \begin{cases} -y, & (\boldsymbol{x}, y) \in \mathcal{M}, \\ 0, & \texttt{otherwise}. \end{cases} \tag{2.11}$$

其中 $\mathcal{M}$ 为 $\mathcal{T}$ 中所有误分类点集, 即 $\mathcal{M} := \{(\boldsymbol{x}, y) \in \mathcal{T} : y(\boldsymbol{w}^T\boldsymbol{x} + b) < 0\}$, 根据每次迭代更新所用的样本集大小, 可将最速下降法分为以下三种算法。

2.3.1 批量梯度下降法

批量梯度下降法 (Batch Gradient Descent, BGD), 每次使用整个训练集 $\mathcal{T}$ 的样本对参数 $\boldsymbol{w}, b$ 进行更新, 即直接对经验风险函数进行最小化:

$$\boldsymbol{w} \leftarrow \boldsymbol{w} + \frac{\eta}{N}\sum_{(\boldsymbol{x},y)\in\mathcal{M}} y\boldsymbol{x}, \quad b \leftarrow b + \frac{\eta}{N}\sum_{(\boldsymbol{x},y)\in\mathcal{M}} y. \tag{2.12}$$

其中 η 在机器学习中被称为**学习率 (Learning Rate)**, 也称为**步长 (Step Size)**, 它可以类比最速下降法中的线性步长因子 α.

优点: 全训练集所确定的方向能够较好的代表样本总体, 从而更准确地朝极小值所在的方向前进. 并且, 当目标函数为凸 (非凸) 函数时, 该算法一定能收敛到全局 (局部) 最小值点.

缺点: 每一次迭代需要遍历样本总体, 需花费大量时间, 而且更新是全体样本遍历后发生, 此时部分样本对参数的更新没有太大影响.

2.3.2 随机梯度下降法

随机梯度下降法 (Stochastic Gradient Descent, SGD), 不同于批量梯度下降法, 随机梯度下降法每次迭代只随机地从训练集 $\mathcal{T}$ 中选取一个样本 $(\boldsymbol{x}, y)$ 对参数进行更新:

$$\boldsymbol{w} \leftarrow \boldsymbol{w} + \eta y\boldsymbol{x}, \quad b \leftarrow b + \eta y, \qquad \text{if} \quad y(\boldsymbol{w}^T\boldsymbol{x} + b) < 0. \tag{2.13}$$

优点: 思路简单, 易于实现, 而且由于每次迭代只计算了一个样本上的梯度, 使得每一轮参数更新速度加快.

缺点: 不具有收敛性, 会在极小值附近震荡, 而且单个样本无法代表全体的趋势.

2.3.3 小批量梯度下降法

小批量梯度下降法 (Mini-batch Gradient Descent, MBGD), 介于上述两种算法之间一种算法, 每次迭代使用一个以上而又不是全部的训练样本, MBGD 先从训练集 $\mathcal{T}$ 中**随机**选出 B 个样本, 称 B **批次大小 (batch size)**, 构成集合 $\mathcal{B} = \{\boldsymbol{x}^{(i_1)}, \cdots, \boldsymbol{x}^{(i_B)}\} \subset \mathcal{T}$, 对参数进行更新:

$$\boldsymbol{w} \leftarrow \boldsymbol{w} + \frac{\eta}{B} \sum_{(\boldsymbol{x},y)\in\mathcal{B}\cap\mathcal{M}} y\boldsymbol{x}, \quad b \leftarrow b + \frac{\eta}{B} \sum_{(\boldsymbol{x},y)\in\mathcal{B}\cap\mathcal{M}} y. \tag{2.14}$$

优点: 计算速度比 BGD 快, 收敛速度快, 而且收敛结果更接近 BGD 的结果.
缺点: 最终结果会在极小值领域内震荡, 而且学习率和批次大小不易于选择.

2.4 算法实现

在感知器算法中, 由于参数较少, 损失函数易收敛, 我们选择简单易行的随机梯度下降法作为优化算法, 该算法流程如下:

算法 1 (感知器参数学习算法).

步 1. 设置最大迭代次数 N, 学习率 η, 随机初始化 $\boldsymbol{w}_0, b_0$, 记 $k := 0$.

步 2. 若达到最大迭代次数或训练集中没有误分类点, 停止.

步 3. 随机选取样本 $(\boldsymbol{x}^{(i)}, y^{(i)}) \in T$, 若 $y^{(i)}(\boldsymbol{w}^T\boldsymbol{x}^{(i)} + b) \leqslant 0$, 即该样本为误分类点, 计算 $\boldsymbol{w}_{k+1} \leftarrow \boldsymbol{w}_k + \eta y^{(i)}\boldsymbol{x}^{(i)}$, $b_{k+1} \leftarrow b_k + \eta y^{(i)}$.

步 4. $k := k + 1$, 转步 2.

下面给出感知器核心部分的 Python 代码, 完整代码请见**附录A.1**, 代码中为了方便存储优化参数, 使用了式 (2.2) 的简写方法, 即 $\boldsymbol{w}$ 为增广权重向量.

```python
import numpy as np
def perceptron(xn, yn, eta=0.7, max_iter=2000, w=np.zeros(3)):
    """
    Input
        xn: 样本的特征, Nx2 矩阵
        yn: 样本的标签, Nx1 矩阵
        eta: 学习率
        max_iter: 最大迭代次数
        w: 初始化参数
    Output
        w: 迭代结果, 最优分类曲线
    """
    f = lambda x: np.sign(w[0] + w[1] * x[0] + w[2] * x[1])  # 当前点 x 在直线的上方则返回 1
    for _ in range(max_iter):
        i = np.random.randint(N)  # 随机选取一个样本
```

```
        if yn[i] != f(xn[i, :]):  # 如果该样本为误分类点，则进行修正曲线
            w[0] += eta * yn[i]
            w[1] += eta * yn[i] * xn[i, 0]
            w[2] += eta * yn[i] * xn[i, 1]
    return w
```

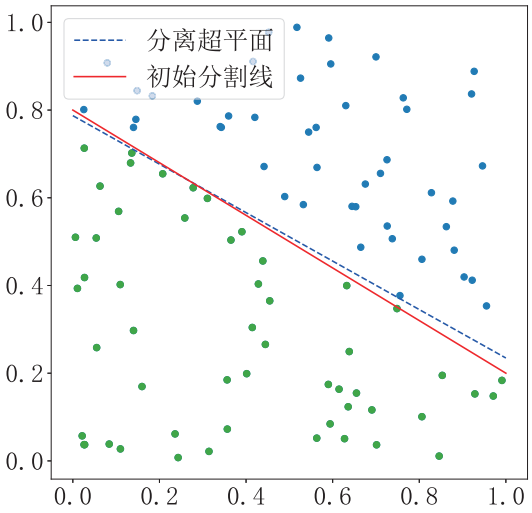


图 3: 对比图

为了验证该算法的正确性, 我们先在二维平面 $[0,1] \times [0,1]$ 上随机生成 100 个点, 然后先取定一条分割线, 将整个点集分为两半, 令分割线上方点对应为正类 $+1$, 下方点对应为负类 -1. 然后将这些点作为训练集, 对感知器训练, 学习过程请见**附录B**, 并将最后感知器计算的分离超平面与初始分割线进行对比 (如左图**图3**).

2.5 感知器的收敛性

对于线性可分的训练集 $\mathcal{T} = \left\{(\boldsymbol{x}^{(n)}, y^{(n)})\right\}_{n=1}^{N}$, 其中 $\boldsymbol{x}^{(n)}$ 为样本的增广特征向量, 则存在一个正常数 γ ($\gamma > 0$) 和增广权重向量 $\widetilde{\boldsymbol{w}}$ ($||\widetilde{\boldsymbol{w}}||^2 = 1$), 对于任意的 n 均有 $y^{(n)}\widetilde{\boldsymbol{w}}^T\boldsymbol{x}^{(n)} \geqslant \gamma$. Novikoff [2] 证明了如下定理.

定理 2.1 (感知器收敛性). 设 $\mathcal{T} = \left\{(\boldsymbol{x}^{(n)}, y^{(n)})\right\}_{n=1}^{N}$ 为线性可分集, γ 的定义如上, 令 R 为训练集中最大的特征向量的模, 即

$$R = \max_n ||x^{(n)}||.$$

则算法*1*权重更新次数不超过 $\dfrac{R^2}{\gamma^2}$.

证明. 感知器算法的更新方法为

$$\boldsymbol{w}_k = \boldsymbol{w}_{k-1} + y^{(k)}\boldsymbol{x}^{(k)}, \tag{2.15}$$

其中 $(\boldsymbol{x}^{(k)}, y^{(k)})$ 表示第 k 个分类错误的样本.

由于初始权重向量 $\boldsymbol{w}_0 = 0$, 所以在第 n 次更新权重向量时, 有

$$\boldsymbol{w}_n = \sum_{k=1}^{n} y^{(k)}\boldsymbol{x}^{(k)}. \tag{2.16}$$

分别估计 $||\boldsymbol{w}_k||^2$ 的上下界:

$$\begin{aligned}||\boldsymbol{w}_n||^2 &= ||\boldsymbol{w}_{n-1} + y^{(n)}\boldsymbol{x}^{(n)}||^2 = ||\boldsymbol{w}_{n-1}||^2 + ||y^{(n)}\boldsymbol{x}^{(n)}||^2 + 2y^{(n)}\boldsymbol{w}_{n-1}^T\boldsymbol{x}^{(n)} \\ &\leqslant ||\boldsymbol{w}_{n-1}||^2 + R^2 \leqslant ||\boldsymbol{w}_{n-2}||^2 + 2R^2 \leqslant \cdots \leqslant ||\boldsymbol{w}_0||^2 + nR^2 = nR^2. \end{aligned} \tag{2.17}$$

$$||\boldsymbol{w}_n||^2 = ||\widetilde{\boldsymbol{w}}||^2||\boldsymbol{w}_n||^2 \geqslant ||\widetilde{\boldsymbol{w}}^T\boldsymbol{w}_n||^2 = \left|\left|\sum_{k=1}^{n}\widetilde{\boldsymbol{w}}^T\left(y^{(k)}\boldsymbol{x}^{(k)}\right)\right|\right|^2 \geqslant n^2\gamma^2. \tag{2.18}$$

由公式 (2.17) 和公式 (2.18), 得到

$$n^2\gamma^2 \leqslant ||\boldsymbol{w}_n||^2 \leqslant nR^2 \;\Rightarrow\; n \leqslant \frac{R^2}{\gamma^2}$$

因此, 算法1会在 $\frac{R^2}{\gamma^2}$ 步内收敛. □

2.6 更多的线性模型

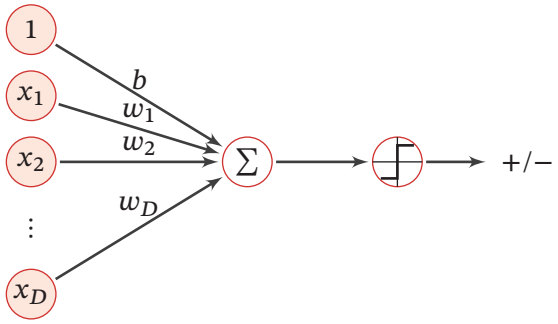


图 4: 二分类的线性模型[1]

线性分类模型一般是一个广义的线性函数, 即一个或多个**线性判别函数**加上一个**非线性激活函数**, 所谓线性是指决策边界是由一个或多个超平面组成, 常用的分类线性模型除了感知器, 还有支持向量机算法. 回归类问题也可以用于分类模型, 只需将标签值改为对应类别的 one-hot 向量表示即可, one-hot 是对应标签值为 1 其他值为 0 的一种特殊向量. 常用的回归模型有: 线性回归, Logistic 回归, Softmax 回归.

如果我们将感知器中的激活函数换成其他的非线性函数, 则可以把它视为一个单独的**神经元**, 再将它们进行堆叠、紧密连接在一起, 就可以形成**人工神经网络**模型, 该模型可以逼近更加复杂的函数, 对更复杂的数据进行分类, 下文的**前馈神经网络**就是人们最早发明的一种神经网络结构.

[1]图片来源: 邱锡鹏, 神经网络与深度学习 [1], 第 56 页

3 前馈神经网络

给定一组神经元, 我们可以将神经元作为结点来构造一个网络. 不同的神经网络模型有着不同的网络连接的拓扑结构, 一种比较直接的拓扑结构是前馈网络. **前馈神经网络 (Feedforward Neural Network, FNN)** 是最早发明的简单人工神经网络. 前馈神经网络也经常称为**多层感知器 (Multi-Layer Perceptron, MLP)**, 但细节上还是有写区别的, 前馈神经网络是由多层连续的非线性函数构成的, 而不是感知器中不连续的非线性函数构成.

在前馈型神经网络中, 每个神经元都被分到不同层中, 每一层的神经元可以接受到前一层所有神经元所输出的信号, 并产生信号输出到下一层. 我们将第 0 层称为**输入层**, 最后一层称为**输出层**, 其他中间层称为**隐藏层**. 整个网络无反馈, 信号从输入层单向传播到输出层, 整个前馈神经网络也可以用一个无环图来表示. 例如, 图5给出了一个三层的前馈神经网络 (计算层数时一般不考虑输入层).

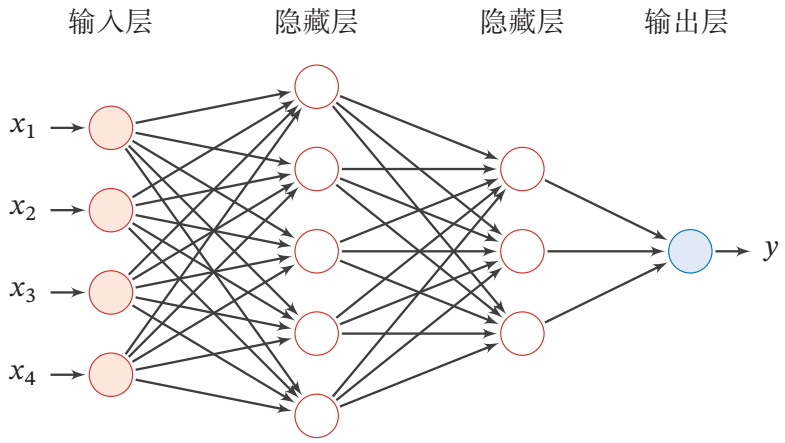


图 5: 前馈神经网络[1]

3.1 前馈传播形式

我们从一个更简单的前馈神经网络 (图6) 来看如何计算神经网络中一个神经元的输入信号. 对于第 1 层从上至下第 2 个神经元, 我们将它的净活性值 (也称净输入) 记为 $z_2^{(1)}$, 于是

$$z_2^{(1)} = x_1 w_{21}^{(1)} + x_2 w_{22}^{(1)} + \cdots + x_m w_{2n}^{(1)}, \tag{3.1}$$

我们将第 l 层的输出称为**活性值 (Activation)** $\boldsymbol{a}^{(l)} = \begin{bmatrix} a_1^{(l)} & \cdots & a_{N_l}^{(l)} \end{bmatrix}^T$, 第 0 层 (输入层) 的活性值定义为训练样本的特征向量 $\boldsymbol{x}$, 于是式 (3.1) 也可写为

$$z_2^{(1)} = a_1^{(0)} w_{21}^{(1)} + a_2^{(0)} w_{22}^{(1)} + \cdots + a_m^{(0)} w_{2n}^{(1)}. \tag{3.2}$$

如果我们记第 l 层的神经元数为 N_l , 第 l 层的偏置为 $\boldsymbol{b}^{(l)} = \begin{bmatrix} b_1^{(l)} & \cdots & b_{N_l}^{(l)} \end{bmatrix}^T$, 第 l 层

[1]图片来源: 邱锡鹏, 神经网络与深度学习 [1], 第 92 页

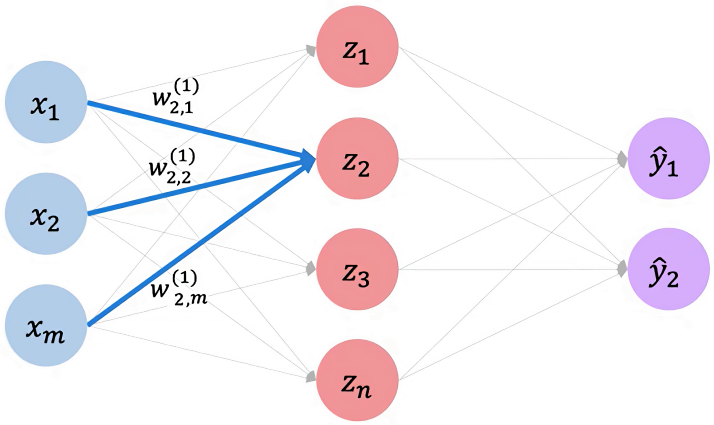


图 6: 计算单个神经元激活值[1]

的**净活性值 (Net Activation)** 为 $\boldsymbol{z}^{(l)} = \begin{bmatrix} z_1^{(l)} & \cdots & z_{N_l}^{(l)} \end{bmatrix}^T$, 第 $l-1$ 层到 l 层的权重矩阵为

$$\boldsymbol{W}^{(l)} = \begin{bmatrix} w_{11}^{(l)} & \cdots & w_{1,N_{l-1}}^{(l)} \\ \vdots & \ddots & \vdots \\ w_{N_l,1}^{(l)} & \cdots & w_{N_l,N_{l-1}}^{(l)} \end{bmatrix}_{(N_l \times N_{l-1})}$$

根据式 (3.2) 我们可以写出如下的前馈传播计算公式:

$$\boldsymbol{z}^{(l)} = \begin{bmatrix} w_{11}^{(l)} & \cdots & w_{1,N_{l-1}}^{(l)} \\ \vdots & \ddots & \vdots \\ w_{N_l,1}^{(l)} & \cdots & w_{N_l,N_{l-1}}^{(l)} \end{bmatrix} \begin{bmatrix} a_1^{(l-1)} \\ \vdots \\ a_{N_{l-1}}^{(l-1)} \end{bmatrix} + \begin{bmatrix} b_1^{(l)} \\ \vdots \\ b_{N_l}^{(l)} \end{bmatrix} = \boldsymbol{W}^{(l)}\boldsymbol{a}^{(l-1)} + \boldsymbol{b}^{(l)}, \tag{3.3}$$

如果令每一层的神经元的激活函数均为 $f_l(\cdot)$, 则

$$\boldsymbol{a}^{(l)} = f_l(\boldsymbol{z}^{(l)}), \tag{3.4}$$

首先根据第 $l-1$ 层神经元的**活性值** $\boldsymbol{a}^{(l-1)}$ 计算出第 l 层神经元的**净活性值** $\boldsymbol{z}^{(l)}$, 然后经过该层的激活函数 $f_l(\cdot)$ 得到第 l 层神经元的**活性值**.

公式 (3.3) 和公式 (3.4) 可以合并写为:

$$\boldsymbol{a}^{(l)} = f_l\left(\boldsymbol{W}^{(l)}\boldsymbol{a}^{(l-1)} + \boldsymbol{b}^{(l)}\right),$$

或

$$\boldsymbol{z}^{(l)} = \boldsymbol{W}^{(l)} f_{l-1}(\boldsymbol{z}^{(l-1)}) + \boldsymbol{b}^{(l)}. \tag{3.5}$$

这样, 前馈神经网经过逐层的信息传递, 得到最后的输出 $\boldsymbol{a}^{(L)}$. 整个网络可以看作一个复杂的参数化函数 $f(\boldsymbol{x}; \boldsymbol{W}, \boldsymbol{b})$, 将向量 $\boldsymbol{x}$ 作为第 0 层的输入 $\boldsymbol{a}^{(0)}$, 将第 L 层的输出 $\boldsymbol{a}^{(L)}$ 作为整个函数的输出. 信息的传递可以抽象为

$$\boldsymbol{x} = \boldsymbol{a}^{(0)} \to \boldsymbol{z}^{(1)} \to \boldsymbol{a}^{(1)} \to \boldsymbol{z}^{(2)} \to \cdots \to \boldsymbol{a}^{(L-1)} \to \boldsymbol{z}^{(L)} \to \boldsymbol{a}^{(L)} = f(\boldsymbol{x}; \boldsymbol{W}, \boldsymbol{b}). \tag{3.6}$$

其中 $\boldsymbol{W}, \boldsymbol{b}$ 表示网络中所有层的连接权重和偏置.

下表1给出了描述前馈神经网络的记号.

[1]图片来源: MIT-6S191 [3], Lecture 1 video 24'50"

表 1: 前馈神经网络的记号

记号	含义
L	神经网络的层数 (一般不考虑输入层)
N_l	第 l 层神经元的个数
$f_l(\cdot)$	第 l 层神经元的激活函数
$\boldsymbol{W}^{(l)} \in \mathbb{R}^{N_l \times N_{l-1}}$	第 $l-1$ 层到 l 层的权重矩阵
$\boldsymbol{b}^{(l)} \in \mathbb{R}^{N_l}$	第 $l-1$ 层到第 l 层的偏置
$\boldsymbol{z}^{(l)} \in \mathbb{R}^{N_l}$	第 l 层神经元的净输入 (净活性值)
$\boldsymbol{a}^{(l)} \in \mathbb{R}^{N_l}$	第 l 层神经元的输出 (活性值)

3.2 损失函数

损失函数 (Loss Function) 是一个用来量化模型预测与真实标签之间的误差, 所以也称为**误差函数 (Error Function)**. 下面介绍两种简单的损失函数. 由于模型都具有损失函数, 我们采用一般的参数化函数的记法, 即模型的预测输出为 $f(\boldsymbol{x};\theta)$.

3.2.1 平方损失函数

平方损失函数 (Quadratic Loss Funtion) 经常用在预测标签 y 为实数的情况, 定义为

$$\mathcal{L}(y, f(\boldsymbol{x};\theta)) = \frac{1}{2}(y - f(\boldsymbol{x};\theta))^2. \tag{3.7}$$

3.2.2 交叉熵损失函数

交叉熵损失函数 (Cross-Entropy Loss Function) 一般用于分类问题, 假设一共有 C 种标签, 即真实标签 $y \in \{1, \cdots, C\}$, 模型的输出 $f(\boldsymbol{x};\theta) \in [0,1]^C$ 为类别标签的条件概率分布, 即

$$p(y = c|\boldsymbol{x};\theta) = f_c(\boldsymbol{x};\theta), \tag{3.8}$$

其中 $f_c(\boldsymbol{x};\theta)$ 表示向量 $f(\boldsymbol{x};\theta)$ 的第 c 维分量, 并满足

$$f_c(\boldsymbol{x};\theta) \in [0,1], \qquad \sum_{c=1}^{C} f_c(\boldsymbol{x};\theta) = 1, \tag{3.9}$$

这表明我们的模型对第 c 中类别标签的预测概率大小为 $f_c(\boldsymbol{x};\theta)$.

我们用一个 C 维的 one-hot 向量 $\boldsymbol{y}$ 表示真实标签, 若真实标签为 c, 则只有 $\boldsymbol{y}$ 的第 c 维为 1, 其他元素均为 0. 则标签向量 $\boldsymbol{y}$ 可以看做样本标签的真实概率分布 $p_r(\boldsymbol{y}|\boldsymbol{x})$, 即样本属于第 c 类的概率为 1, 属于其他概率均为 0.

对于两个概率分布, 一般可用交叉熵衡量它们的差异. 标签的真实分布 $\boldsymbol{y}$ 和模型预测分布 $f(\boldsymbol{x};\theta)$ 之间的交叉熵为

$$\mathcal{L}(\boldsymbol{y}, f(\boldsymbol{x};\theta)) = -\boldsymbol{y} \log f(\boldsymbol{x};\theta) = -\sum_{c=1}^{C} y_c \log f_c(\boldsymbol{x};\theta). \tag{3.10}$$

例如一个五分类问题 ($C = 5$), 一个样本标签为 $y = 4$, 则其真实分布为 $\boldsymbol{y} = [0\ 0\ 0\ 1\ 0]^T$, 若模型预测分布为 $f(\boldsymbol{x};\theta) = [0.1\ 0.1\ 0.1\ 0.5\ 0.2]^T$, 则它们的交叉熵为

$$-\begin{bmatrix}0 & 0 & 0 & 1 & 0\end{bmatrix}\log\left(\begin{bmatrix}0.1 & 0.1 & 0.1 & 0.5 & 0.2\end{bmatrix}^T\right) = -\log(0.5) = f_c(\boldsymbol{x};\theta) \tag{3.11}$$

由于 $\boldsymbol{y}$ 为 one-hot 向量, 所以公式 (3.10) 也可写为

$$\mathcal{L}(y, f(\boldsymbol{x};\theta)) = -\log f_y(\boldsymbol{x};\theta). \tag{3.12}$$

3.3 优化算法

3.3.1 反向传播算法

假设采用随机梯度下降法进行神经网络学习, 给定一个样本 $(\boldsymbol{x}, \boldsymbol{y})$, 将其输入到神经网络模型中, 得到网络输出 $\widehat{\boldsymbol{y}}$. 假设损失函数是 $\mathcal{L}(\boldsymbol{y}, \widehat{\boldsymbol{y}}; \boldsymbol{W}, \boldsymbol{b})$, 要使用梯度下降法对参数进行学习就需要计算损失函数关于每个参数的偏导数.

直接单独地对每个参数求偏导数效率太低, 而且非常复杂, 在这里就需要用到神经网络中层与层直接的关系, 逐层求导. 为了更加直观的体现出层与层之间信息传播的关系, 也更加清楚的反映链导法则所涉及的变量, 我们绘制了图7.

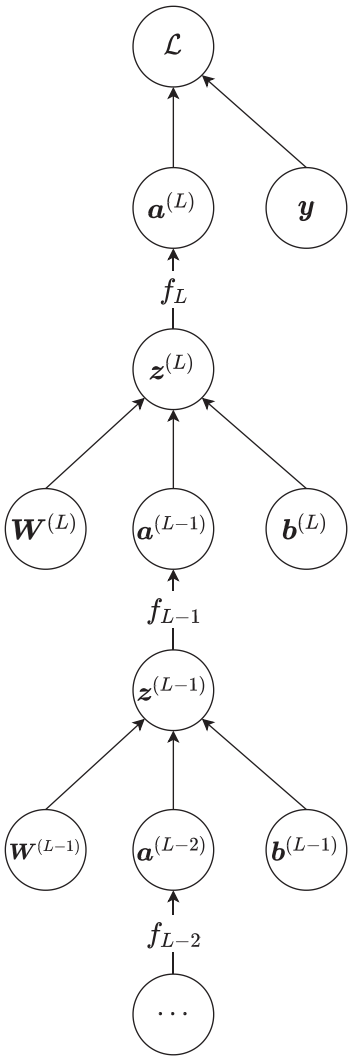


图 7: 信息传播关系

首先从最后一层, 也就是第 L 层开始求偏导, 我们先列出两层参数之间的关系式

$$\begin{cases} z_i^{(l)} = \sum_{j=1}^{N_{l-1}} w_{ij}^{(l)} a_j^{l-1} + b_i^{(l)}, \\ a_i^{(l)} = f_l(z_i^l), \end{cases} \tag{3.13}$$

其中 $w_{ij}^{(l)}$ 为 $\boldsymbol{W}^{(l)}$ 中第 (i,j) 位置的元素, $b_i^{(l)}$ 为 $\boldsymbol{b}^{(l)}$ 中第 i 个分量, $z_i^{(l)}$, $a_i^{(l)}$ 同理.

对第 L 层参数求偏导, 可得

$$\frac{\partial \mathcal{L}}{\partial b_i^{(L)}} = \frac{\partial \mathcal{L}}{\partial a_i^{(L)}}\frac{\partial a_i^{(L)}}{\partial z_i^{(L)}}\frac{\partial z_i^{(L)}}{\partial b_i^{(L)}} = \frac{\partial \mathcal{L}}{\partial a_i^{(L)}}\frac{\partial a_i^{(L)}}{\partial z_i^{(L)}}, \tag{3.14}$$

$$\frac{\partial \mathcal{L}}{\partial w_{ij}^{(L)}} = \frac{\partial \mathcal{L}}{\partial a_i^{(L)}}\frac{\partial a_i^{(L)}}{\partial z_i^{(L)}}\frac{\partial z_i^{(L)}}{\partial w_{ij}^{(L)}} = \frac{\partial \mathcal{L}}{\partial a_i^{(L)}}\frac{\partial a_i^{(L)}}{\partial z_i^{(L)}} a_j^{(L-1)}, \tag{3.15}$$

$$\frac{\partial \mathcal{L}}{\partial a_j^{(L-1)}} = \sum_{i=1}^{N_L}\frac{\partial \mathcal{L}}{\partial a_i^{(L)}}\frac{\partial a_i^{(L)}}{\partial z_i^{(L)}}\frac{\partial z_i^{(L)}}{\partial a_j^{(L-1)}} = \sum_{i=1}^{N_L}\frac{\partial \mathcal{L}}{\partial a_i^{(L)}} f_L'(z_i^{(L)}) w_{ij}^{(L)}, \tag{3.16}$$

我们将公式 (3.14), 公式 (3.15), 公式 (3.16) 写成向量的形式, 则

$$\begin{aligned}
\frac{\partial \mathcal{L}}{\partial \boldsymbol{b}^{(L)}} &= f_L'(\boldsymbol{z}^{(L)}) \circ \frac{\partial \mathcal{L}}{\partial \boldsymbol{a}^{(L)}}, \\
\frac{\partial \mathcal{L}}{\partial \boldsymbol{W}^{(L)}} &= \left(f_L'(\boldsymbol{z}^{(L)}) \circ \frac{\partial \mathcal{L}}{\partial \boldsymbol{a}^{(L)}} \right) \boldsymbol{a}^{(L-1)^T}, \\
\frac{\partial \mathcal{L}}{\partial \boldsymbol{a}^{(L-1)}} &= \boldsymbol{W}^{(L)^T} \left(f_L'(\boldsymbol{z}^{(L)}) \circ \frac{\partial \mathcal{L}}{\partial \boldsymbol{a}^{(L)}} \right),
\end{aligned} \tag{3.17}$$

其中 $\boldsymbol{A} \circ \boldsymbol{B}$ 表示求矩阵 $\boldsymbol{A}$, $\boldsymbol{B}$ 的 **Hadamard 积**, 也就将两个矩阵的对应项相乘. $f_l'(\boldsymbol{z}^{(l)})$ 为按位计算的函数, 即

$$f_l'(\boldsymbol{z}^{(l)}) = \begin{bmatrix} f_l'(z_1) & \cdots & f_l'(z_{N_l}) \end{bmatrix}^T, \tag{3.18}$$

$\dfrac{\partial \mathcal{L}}{\partial \boldsymbol{b}^{(l)}}$ 表示标量对向量的偏导中的**分母布局 (Denominator Layout)**, 即

$$\frac{\partial \mathcal{L}}{\partial \boldsymbol{b}^{(l)}} = \begin{bmatrix} \dfrac{\partial \mathcal{L}}{\partial b_1^{(l)}} & \cdots & \dfrac{\partial \mathcal{L}}{\partial b_{N_l}^{(l)}} \end{bmatrix}^T \in \mathbb{R}^{N_l \times 1}. \tag{3.19}$$

不难发现, 公式 (3.17) 中的 L 可以换成任意的 l $(2 \leqslant l \leqslant L)$, 我们称

$$\boldsymbol{\delta}^{(l)} := \frac{\partial \mathcal{L}}{\partial \boldsymbol{z}^{(l)}} = \frac{\partial \mathcal{L}}{\partial \boldsymbol{b}^{(l)}} = f_l'(\boldsymbol{z}^{(l)}) \circ \frac{\partial \mathcal{L}}{\partial \boldsymbol{a}^{(l)}}$$

为第 l 层神经元的**误差项 (Error)**, 它表示第 l 层神经元对最终损失的影响, 也反映了最终损失对第 l 层神经元的敏感程度.

公式 (3.17) 可以写为更一般的递推式

$$\begin{aligned}
\boldsymbol{\delta}^{(l)} &= f_L'(\boldsymbol{z}^{(l)}) \circ \frac{\partial \mathcal{L}}{\partial \boldsymbol{a}^{(l)}} = f_L'(\boldsymbol{z}^{(l)}) \boldsymbol{W}^{(l+1)^T} \boldsymbol{\delta}^{(l+1)}, \\
\frac{\partial \mathcal{L}}{\partial \boldsymbol{b}^{(l)}} &= \boldsymbol{\delta}^{(l)}, \\
\frac{\partial \mathcal{L}}{\partial \boldsymbol{W}^{(l)}} &= \boldsymbol{\delta}^{(l)} \boldsymbol{a}^{(l-1)^T}, \\
\frac{\partial \mathcal{L}}{\partial \boldsymbol{a}^{(l-1)}} &= \boldsymbol{W}^{(l)^T} \boldsymbol{\delta}^{(l)},
\end{aligned} \qquad (2 \leqslant l \leqslant L) \tag{3.20}$$

为了更好的理解公式 (3.20) 中定义参数的含义, 我们取 $L = 4$, 简记 $f_l' := f_l(z^{(l)})$, 可得如下关系式

$$\frac{\partial \mathcal{L}}{\partial \boldsymbol{a}^{(1)}} = \underbrace{\boldsymbol{W}^{(2)^T} \cdot f_2' \circ \underbrace{\boldsymbol{W}^{(3)^T} \cdot f_3' \circ \underbrace{\boldsymbol{W}^{(4)^T} \cdot f_4' \circ \overbrace{\frac{\partial \mathcal{L}}{\partial a^{(4)}}}^{\boldsymbol{\delta}^{(4)}}}_{\frac{\partial \mathcal{L}}{\partial \boldsymbol{a}^{(3)}}}}_{\boldsymbol{\delta}^{(3)}}}_{\frac{\partial \mathcal{L}}{\partial \boldsymbol{a}^{(2)}}}}_{\boldsymbol{\delta}^{(2)}}, \tag{3.21}$$

公式 (3.21) 中省略了括号, 计算顺序应为从右至左.

结合公式 (3.20) 我们给出**反向传播 (BackPropagation, BP)** 的递推形式:

$$\text{初始化：} \quad \boldsymbol{\delta}^{(L)} = f_L' \circ \frac{\partial \mathcal{L}}{\partial \boldsymbol{a}^{(L)}}, \tag{3.22}$$

$$\text{递推求解：} \quad \frac{\partial \mathcal{L}}{\partial \boldsymbol{b}^{(l)}} = \boldsymbol{\delta}^{(l)}, \qquad \frac{\partial \mathcal{L}}{\partial \boldsymbol{W}^{(l)}} = \boldsymbol{\delta}^{(l)} \boldsymbol{a}^{(l-1)^T}, \tag{3.23}$$

$$\boldsymbol{\delta}^{(l-1)} = f_{l-1}' \circ \left(\boldsymbol{W}^{(l)^T} \boldsymbol{\delta}^{(l)} \right). \quad (2 \leqslant l \leqslant L) \tag{3.24}$$

从公式 (3.24) 可以看出, 第 l 层的误差项可以通过第 $l+1$ 层的误差项计算得到, 这就是误差的**反向传播**. 其含义是: 第 l 层的一个神经元的误差项 (或最终损失对该神经元的敏感度) 是所有与该神经元相连的第 $l+1$ 层的神经元误差项的权重和, 然后乘上该神经元激活函数的梯度. 图8给出了一个 2 层神经网络结构的整个传播算法流程图.

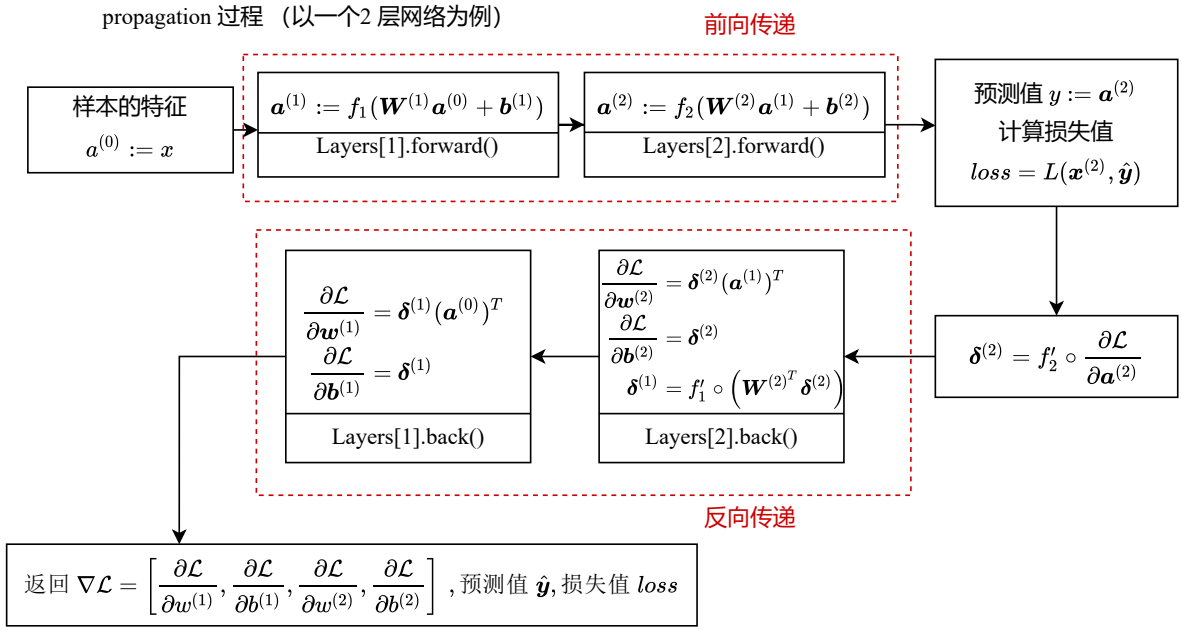


图 8: 传播算法流程图

```python
import numpy as np

class Layer:  # 神经网络中的每一层
    """
    Initialize
        当前层和上一层的权重矩阵: weight
        当期层的偏置: bias
        当前层所用的激活函数: activation

    """
    def forward(self, x_input):  # 向前传播, 计算当前层的激活值
        z = self.weight @ x_input + self.bias
```

```
13          return z, self.activation(z)
14
15      def back(self, a, z, delta):  # 反向传播
16          """
17          设当前层为 l，计算当前层的偏置和权重的梯度，并求出 l-1 层的 delta 值
18          Input
19              a: l-1 层的激活值
20              z: l-1 层的加权值 z = w*a+b
21              delta: l 层的 delta 值
22          Output
23              np.array, list: l-1 层的 delta 值和权重梯度和偏置梯度展成行向量的结果
24          """
25          grad_weight = (delta @ a.T)
26          grad_bias = delta
27          return self.activation(z, order=1) * (self.weight.T @ delta), [grad_weight, grad_bias]
28
29  class Model:  # 神经网络模型
30      """
31      Initialize
32          layers: 由 Layer 类为元素的 list
33      """
34      def propagation(self, x_input, output=None):
35          """
36          完成一次传播（包括前向传播和反向传播计算梯度），如果 output 没有输入则只用于做预测
37          Input
38              x_input: 输出参数
39              output: 期望输出
40          Output
41              总梯度
42          """
43          grad = []  # 用 list 存储总梯度
44          L = len(self.layers)  # 网络总层数
45          a = []  # 向前传播中，每一层的激活值，加权值
46          n = np.size(x_input)
47          x_input = x_input.reshape(n, 1)
48          a.append((x_input, 0))
```

```
        for i in range(L):  # 前向传播
            layer = self.layers[i]
            a.append(layer.forward(a[i][0]))
        y, layer = a[L][1], self.layers[L - 1]  # 预测值 y，最后一层 layer
        new_output = (y == np.max(y)).astype(int)  # 当前正规化预测值
        if output is None:
            return new_output
        m = np.size(output)
        output = output.reshape(m, 1)
        loss = self.loss(y, output)
        delta = layer.activation(a[L][0], order=1) * self.loss(y, output, order=1)  # 初始化误差值
        for i in range(L-1, -1, -1):  # 反向传播，计算梯度
            layer = self.layers[i]  # 倒着取值
            delta, g = layer.back(*a[i], delta)  # 计算每一层的梯度
            grad = g + grad
        return grad, new_output, loss
```

3.4 激活函数

在前馈神经网络中, 每一层的神经元都有一个激活函数, 为了增强网络的表示能力和学习能力, 激活函数需要具备以下几点性质:

(1) 连续可导 (只在一个零测集上不可导) 的非线性函数.

(2) 激活函数及其导数应该尽可能简单, 有利于提高网络的计算效率.

(3) 激活函数的导函数应在一个合适的区间内, 否则影响训练的效率和稳定性.

3.4.1 Sigmoid 型函数

Logistic 函数 Logistic 函数定义为

$$\sigma(x) = \frac{1}{1+e^{-x}}, \tag{3.25}$$

$$\sigma'(x) = \sigma(x)(1-\sigma(x)). \tag{3.26}$$

Logistic 函数可以看成一个" 挤压" 函数, 将实数域的驶入" 挤压" 到 $(0,1)$. 当输入值在 0 附近时, Sigmoid 型函数近似为线性函数, 当输入值靠近两端时, 对输入进行抑制. 这与生物神经元类似, 对一些输入会产生兴奋, 对另一些输入会产生抑制. 和感知器使用的阶跃函数相比, Logistic 函数是连续可导的, 其数学性质更好.

Tanh 函数 Tanh 函数也是一种 Sigmoid 型函数. 其定义为

$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}, \tag{3.27}$$

$$\tanh'(x) = 1 - \tanh^2(x). \tag{3.28}$$

Tanh 函数可以看作放大并平移的 Logistic 函数, 其值域是 $(-1, 1)$.

$$\tanh(x) = 2\sigma(2x) - 1. \tag{3.29}$$

图9给出了 Logistic 函数和 Tanh 函数的形状. Tanh 函数的输出是**零中心 (Zero-Centered)**, 而 Logistic 函数的输出恒大于 0. 非零中心化的输出可以使得下一层的神经元的输入发生**偏置偏移 (Bias Shift)**, 并进一步使得梯度收敛速度变慢.

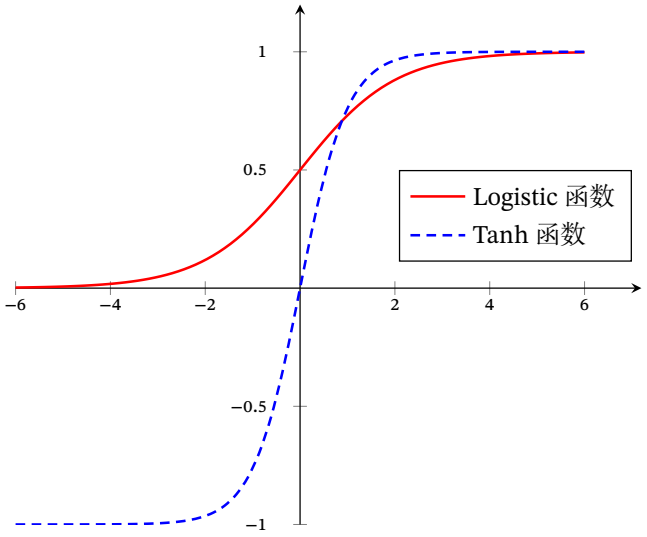


图 9: Logistic 函数和 Tanh 函数[1]

3.4.2 ReLU 函数

ReLU(Rectified Linear Unit, 修正线性单元), 也称线性整流函数, 是目前深度神经网络中经常使用的激活函数. ReLU 实际上是一个斜坡 (ramp) 函数, 定义为

$$\text{ReLU}(x) = \begin{cases} x, & x \geqslant 0 \\ 0, & x < 0 \end{cases} = \max(0, x). \tag{3.30}$$

优点 ReLU 函数形式简单, 只需进行加、乘和比较的操作, 计算上更加高效. ReLU 函数也被认为有一定的生物学原理 [4], 例如单侧抑制, 具有高兴奋程度的性质. 同时由于 ReLU 函数导数唯一, 所以在一定程度上缓解了神经网络的**梯度消失问题**, 加速了梯度下降的收敛速度.

[1]图片来源: 邱锡鹏, 神经网络与深度学习 [1], 第 84 页

缺点 在训练时, 如果参数在一次不恰当更新后, 某一个隐藏层的 ReLU 函数在训练时在所有训练数据上都不能被激活, 那么这个神经元的梯度永远是 0, 以后训练过程中都不会被更新, 这种现象称为**死亡 ReLU 问题 (Dying ReLU Problem)**. 为了避免这种问题, 引入了**带泄露的 ReLU(Leaky ReLU)**, 即当 $x < 0$ 时, 保持一个很小的梯度 γ, 定义如下

$$\text{LeakyReLU}(x) = \begin{cases} x, & x \geqslant 0 \\ \gamma x, & x < 0 \end{cases} = \max(0, x) + \gamma \min(0, x), \tag{3.31}$$

其中 γ 是一个很小的常数.

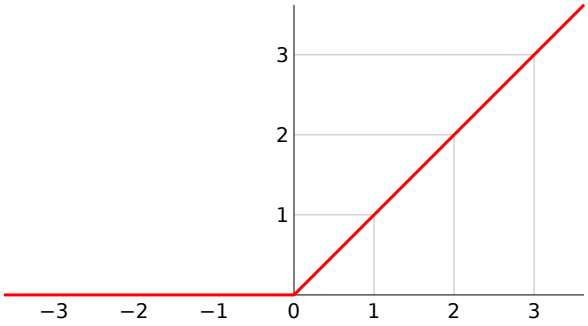


图 10: ReLU 函数[1]

3.5 算法实现

我们以识别手写数字作为前馈神经网络的应用, 使用小批量随机梯度下降法 (2.3.3), 使用 MNIST 数据集, 在读入 MNIST 数据集 Input.py中, 我们先将 MNIST 数据集文件进行解压, 转化为 Numpy 中 array 类. 在主程序 Main.py中, 先将输入的像素数据 (范围在 $[0, 255]$) 正规化处理为 $[0, 1]$ 作为样本的特征向量, 样本的标签处理为 $\mathbb{R}^{10\times 1}$ 的 one-hot 向量, 若样本的标签为 1, 则对应的 one-hot 向量为 $[0 \quad 1 \quad 0 \quad \cdots \quad 0]^T \in \mathbb{R}^{10\times 1}$, 即对应位为 1 其他位均为 0 的向量. 然后调用前馈神经网络核心代码 ANN.py中的 Model 类构建网络模型, 再使用 `Model.fit` 函数进入神经网络训练过程.

每次训练开始时, 将数据集打乱, 然后分为多个 Mini-Batch, 每次取出一个 Mini-Batch 进入到 `Model.Batch` 函数中, 开始处理每个 Mini-batch, 如果有多线程的情况 (参考 C++ 多线程代码[2], Python 由于功能限制无法实现真正的多线程) 可以同时取出多个训练样本, 同时执行 `Model.propagation` 函数 (该传播算法原理图请见图8, 其中会调用 `Layer.forward` 和 `Layer.back` 函数, 计算损失值). 再将一个 Mini-batch 中全部训练样本所返回的梯度求平均值, 对神经网络的 $\boldsymbol{W}, \boldsymbol{b}$ 参数进行一次更新, 返回到 `Model.fit` 函数中, 接着取出下一个 Mini-batch, 再进入到 `Model.Batch` 进行参数学习, 直到全部的 Mini-batch 取完, 一次训练结束.

[1]图片来源: https://zh.m.wikipedia.org/zh-hans/File:Ramp_function.svg

[2]吴天阳的博客: BP 神经网络算法的基本原理及 C++ 实现

最后使用验证集, 对神经网络的泛用性进行评估, 我们使用**准确率**的评价指标评判模型对测试集评价分数, **准确率 (Accuracy)** 的定义如下:

$$\mathcal{A} = \frac{1}{N}\sum_{n=1}^{N} I(\boldsymbol{y}^{(n)} = \hat{\boldsymbol{y}}^{(n)}), \tag{3.32}$$

其中 $I(x)$ 为指示函数, 当内部条件 x 为真时 $I(x) = 1$, 否则 $I(x) = 0$. 最后输出该神经网络的准确率, 结束程序. 我们制作了更加直观的前馈神经网络算法流程图便于理解算法的工作原理, 请见下图11.

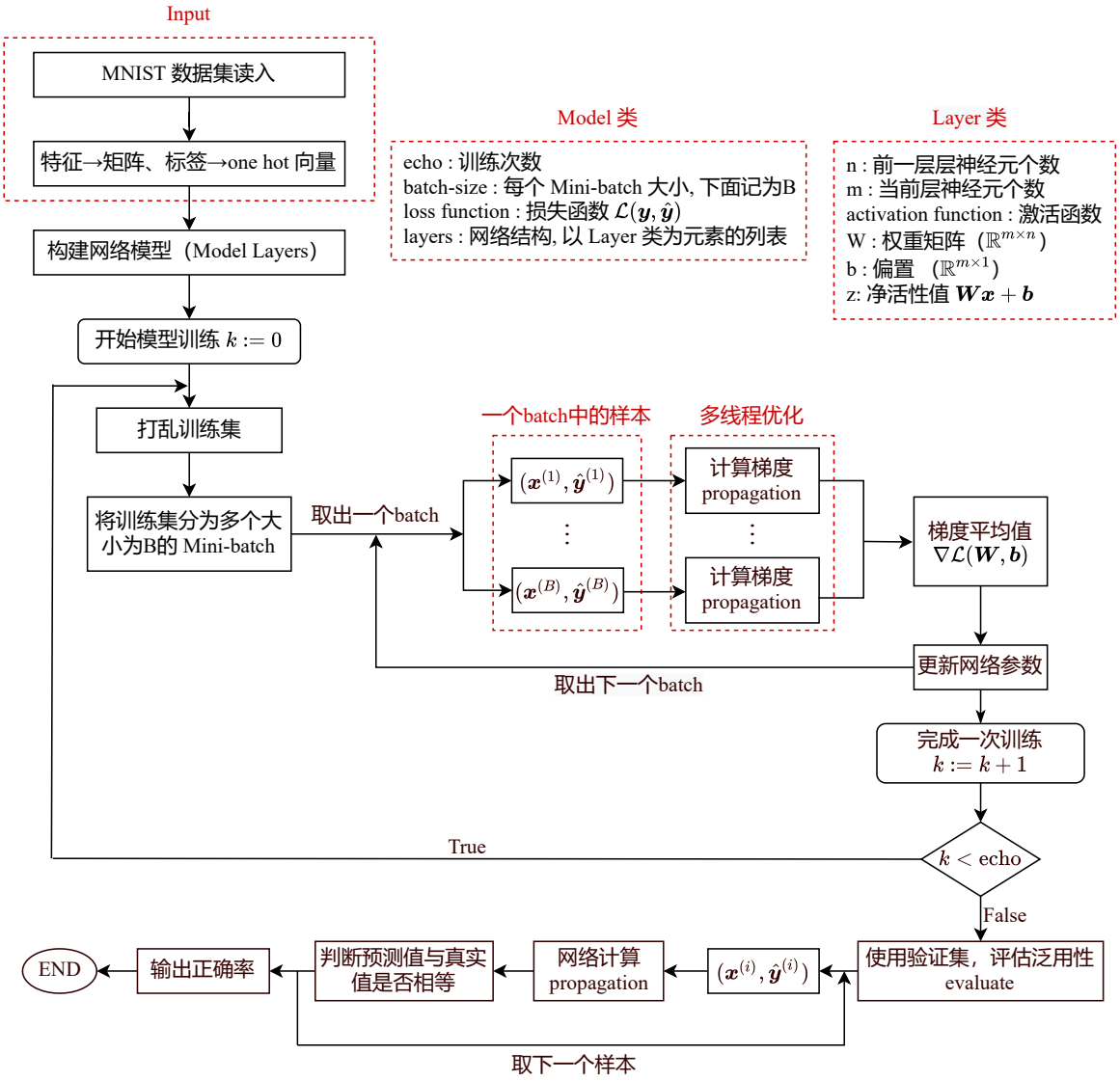


图 11: 前馈神经网络算法流程图

3.6 过拟合

使用梯度下降法的优化算法, 容易发生过拟合现象, 过拟合的定义如下

定义 2 (过拟合). 给定一个假设空间 $\mathcal{F}$, 一个假设 $f \in \mathcal{F}$, 如果存在其他假设 $g \in \mathcal{F}$, 使得在训练集上 f 的损失小于 g 的损失, 但是在整个样本空间上 g 的损失比 f 小, 那么则称假设 f 过度拟合训练数据集.

为解决过拟合现象, 有如下两种简单的方法.

3.6.1 提前停止

由于过拟合的原因, 在训练集上最优的假设 f, 并不一定是测试集上最优的假设. 因此, 除了训练集和测试集外, 有时会使用一个**验证集 (Validation Set)** 来测试模型是否发生了过拟合现象. 在每次迭代时, 将新的模型 $f(\boldsymbol{x};\theta)$ 在验证集上进行测试, 计算准确率, 如果在验证集上准确率不再上升, 就停止迭代. 这种策略叫**提前终止 (Early Stop)**. 验证集可以是从训练集中选取一小部分出来, 有时为了方便起见, 也可以将验证集直接取为测试集的一部分.

下图12是在识别手写数字中, 网络模型设置为 32 个节点的隐藏层, 使用 Sigmoid 作为激活函数, Batch Size 取为 100, 其训练次数与准确率的关系图 (这里将验证集随机取为测试集中 10% 的数据), 并且我们使用了一个 7 阶多项式对真实结果进行拟合, 拟合结果表明在第 51 次训练时准确率接近最大值达到了 0.93, 后面模型在验证集上的准确率开始缓慢下降, 所以根据提前停止策略应该在 50 次左右停止训练.

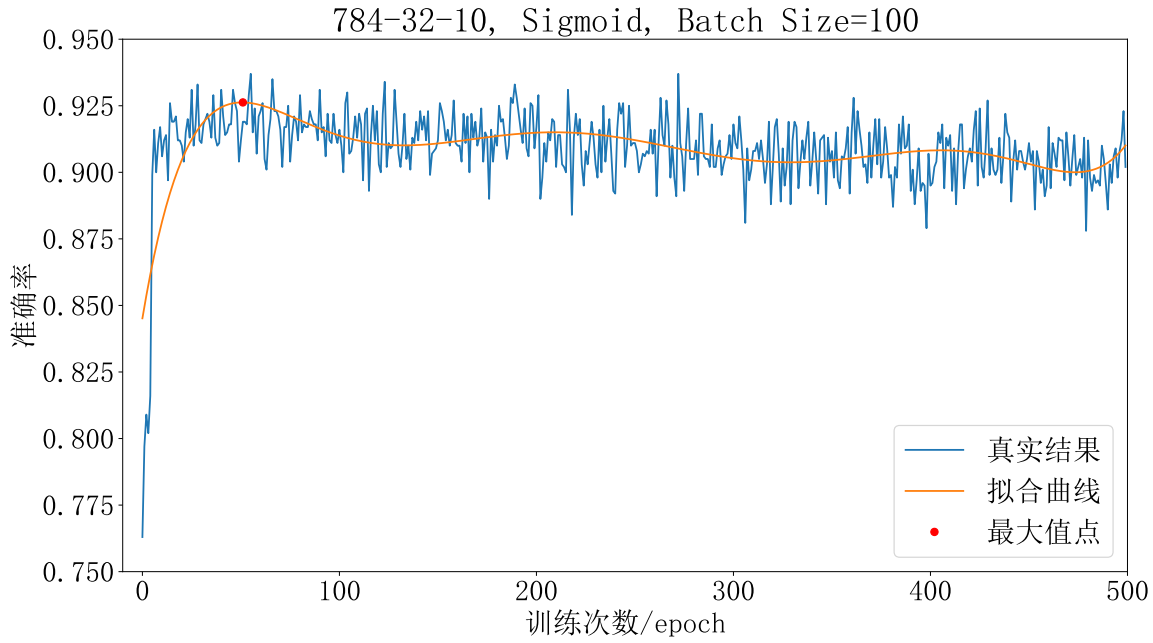
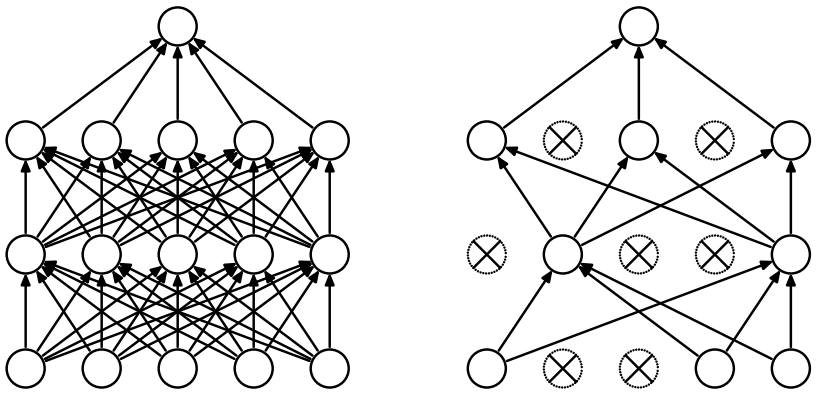


图 12: 提前终止

3.6.2 丢弃法

这里还介绍一种简单的方法来避免过拟合, 我们在每次训练网络时丢弃掉一部分的神经元 (同时丢弃相连的边), 这种方法称为**过拟合 (Dropout Method)**, 参考 Srivastava N, et al.[5]. 先设定一个概率 p, 对每一个神经元都以概率 p 判定是否被丢弃, 只需在计算每一层的活性值时乘上一个选择矩阵即可, 选择矩阵为一个对角阵, 对角线上元素满足概率为 p 的 Bernoulli 分布. 图13给出了一个网络应用丢弃法前后的对比图.

我们尝试将这个方法应用到数字识别中, 但发现效果并不明显, 可能是由于网络节点数目过少或者训练次数太少导致的 (也有可能是代码问题), 所以没有对其绘制图像. 根据 Hinton G E, et al.[6] 中的结果, 至少使用 800 个节点, 迭代 500 次以上才能看到明显区别.



(a) Standard Neural Net (b) After applying dropout.

图 13: 丢弃法[1]

3.7 模型参数选择

我们对手写数字识别的神经网络模型, 固定总训练次数为 500 次, 取 10% 的测试集作为验证集, 验证每一训练结果的准确率. 通过修改不同的 Batch Size 大小、网络结构、激活函数, 通过对比得出相对较好网络模型.

Batch Size 根据图14可以看出该模型使用 Batch 大小为 100 可以达到较好训练效果, 且训练速度较快.

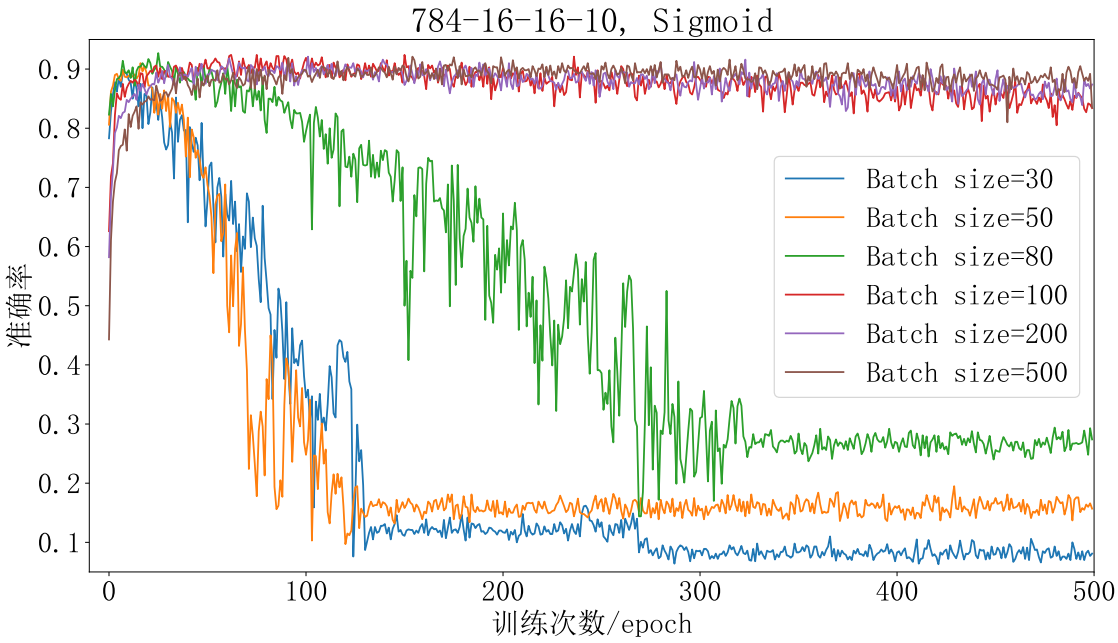


图 14: Batch Size 对比

[1]图片来源: Srivastava N[5], 第 2 页.

网络结构 根据图15可以看出该模型使用一个含有 32 个节点的隐藏层网络结构训练效果较好, 且节点数较少, 训练速度更快.

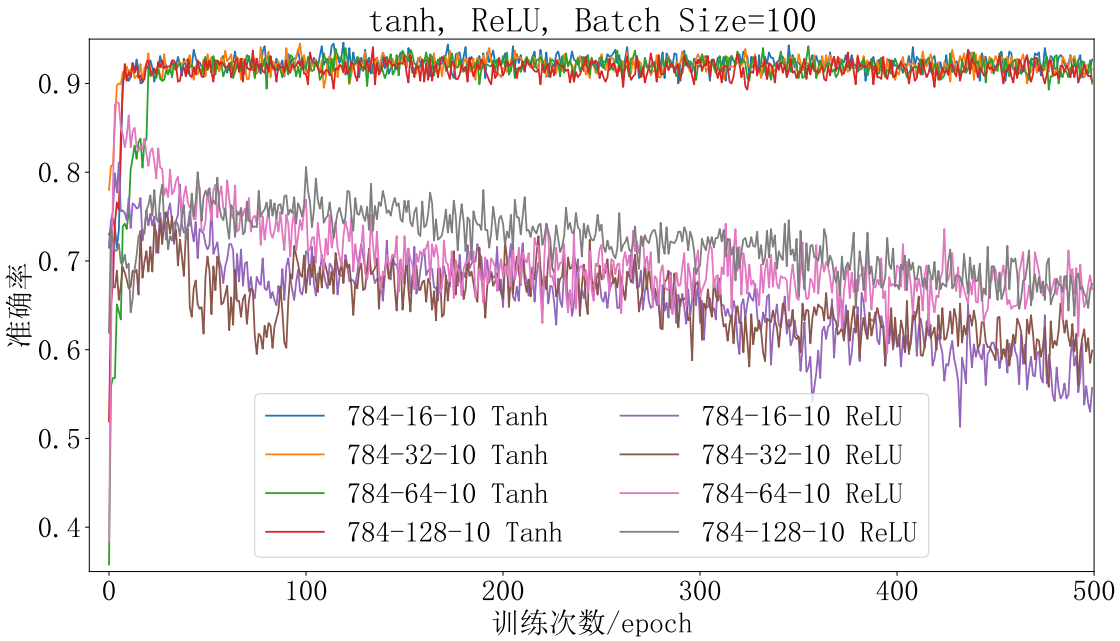


图 15: 网络结构对比

激活函数 根据图16可以看出该模型使用 Tanh 函数训练效果最好.

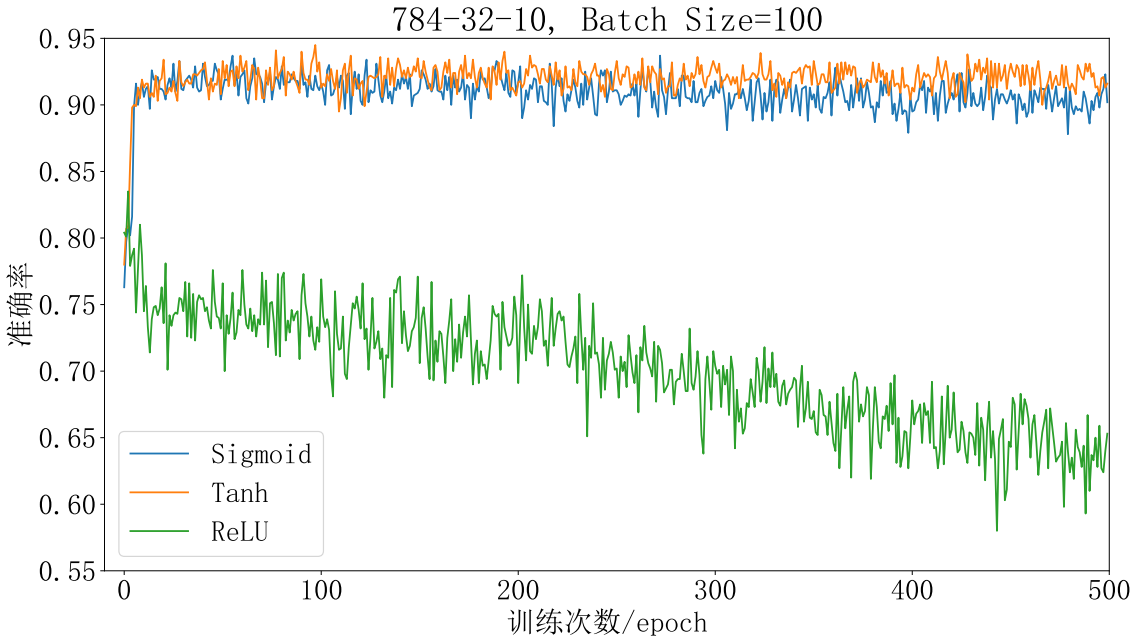


图 16: 激活函数对比

可以自行运行代码文件主程序 Main.py测试每次训练后前馈神经网络的准确率, 经过我们多次试验, 网络结构为 783-32-10, 激活函数为 Tanh, Batch Size=100, 训练 10 次左右已经可以达到 90% 以上的准确率.

参考文献

[1] 邱锡鹏. 神经网络与深度学习 [M]. 北京: 机械工业出版社, 2020.3.

[2] Novikoff A B, 1963. On convergence proofs for perceptron[R]. DTIC Document.

[3] Alexander Amini, Ava Soleimany, et al. MIT Introduction to Deep Learning 6.S191, 2022 [EB/OL]. http://introtodeeplearning.com/

[4] Xavier Glorot, Antoine Bordes and Yoshua Bengio. Deep sparse rectifier neural networks[C]. AISTATS. 2011.

[5] Srivastava N, Hinton G, Krizhevsky A, et al. Dropout: A simple way to prevent neural networks from overfitting[J]. The Journal of Machine Learning Research, 2014, 15(1): 1929-1958.

[6] Hinton G E, Srivastava N, Krizhevsky A, et al. Improving neural networks by preventing co-adaptation of feature detectors[J]. arXiv preprint arXiv:1207.0580, 2012.

A 完整代码

全部代码均已上传至 Github 仓库[1].

A.1 感知器算法

```python
# coding:utf-8
import numpy as np
import matplotlib.pyplot as plt

config = {  # 图像配置
    "figure.figsize": (8, 8),  # 固定图像大小
    "figure.dpi": 200,
    "font.family": 'serif',
    "font.size": 24,
    "savefig.bbox": 'tight',  # 减小边框大小
    "mathtext.fontset": "stix",
    "font.serif": ['SimSun'],
    "axes.unicode_minus": False,  # 用来正常显示负号
}
plt.rcParams.update(config)

```

[1]Github 仓库链接: https://github.com/wty-yy/Perceptron-and-FNN-by-Python

```
# 生成 100 个随机点
N = 100
xn = np.random.rand(N, 2)  # 第一列为横轴，第二列为纵轴
x = np.linspace(0, 1)  # 选取 [0,1] 上的线性分布

# 选取初始分割直线
a, b = -0.6, 0.8
f = lambda x: a * x + b
yn = np.zeros([N, 1])  # 样本的标签集，点在分割线上方为 1，反之为-1

def print_example(xn, yn):  # 打印数据集
    plt.plot(xn[:, 0], xn[:, 1], 'o', color='tab:blue')
    for i in range(N):
        if f(xn[i, 0]) > xn[i, 1]:  # 分割线下方
            yn[i] = 1
            plt.plot(xn[i, 0], xn[i, 1], 'o', color='tab:green')
        else:
            yn[i] = -1  # 分割线上方

plt.plot(x, f(x), 'tab:red')
print_example(xn, yn)
plt.legend(['分离超平面', '正类', '负类'])
plt.savefig('分离平面.pdf')
plt.show()

def print_hyper(xn, yn, w):  # 打印超平面
    if w[2] == 0:
        return
    y = lambda x: -w[0] / w[2] - w[1] / w[2] * x
    x = np.linspace(0, 1)
    plt.plot(x, y(x), 'tab:blue')
    print_example(xn, yn)

def perceptron(xn, yn, eta=0.7, max_iter=2000, w=np.random.rand(3)):
    """
    Input
        xn: 样本的特征，Nx2 矩阵
        yn: 样本的标签，Nx1 矩阵
        eta: 学习率
        max_iter: 最大迭代次数
```

```
        w: 初始化参数
    Output
        w: 迭代结果，最优分类曲线
    """
    f = lambda x: np.sign(w[0] + w[1] * x[0] + w[2] * x[1])  # 当前点 x 在直线的上方则返回 1
    for _ in range(max_iter):
        i = np.random.randint(N)  # 随机选取一个样本
        if yn[i] != f(xn[i, :]):  # 如果该样本为误分类点，则进行修正曲线
            w[0] += eta * yn[i]
            w[1] += eta * yn[i] * xn[i, 0]
            w[2] += eta * yn[i] * xn[i, 1]
        if _ + 1 == 1 or _ + 1 == 100 or _ + 1 == 500 or _ + 1 == 2000:
            print_hyper(xn, yn, w)
            plt.savefig('迭代' + str(_+1) + '.pdf')
            plt.show()
    return w

w = perceptron(xn, yn)  # 开始感知机学习算法

print_example(xn, yn)
y = lambda x: -w[0] / w[2] - w[1] / w[2] * x
plt.plot(x, y(x), 'b--', label='分离超平面')
plt.plot(x, f(x), 'r', label='初始分割线')
plt.legend()
plt.savefig('对比图.pdf')
plt.show()
```

A.2 前馈神经网络算法

读入 MNIST 数据集 Input.py

```
import os
import struct
import numpy as np
import matplotlib.pyplot as plt


def load_mnist(path, kind='train'):
```

```
8       """
9       用于读取 MNIST 数据集，代码参考:
        ↪ https://blog.csdn.net/simple_the_best/article/details/75267863
10      Args:
11          path: 读入文件的路径
12          kind: 读入数据类型，train 为训练数据，t10k 为测试数据
13      Returns: 像素矩阵 nx784 和标签向量 nx1
14      """
15      labels_path = os.path.join(path, '%s-labels.idx1-ubyte' % kind)
16      images_path = os.path.join(path, '%s-images.idx3-ubyte' % kind)
17      with open(labels_path, 'rb') as lbpath:  # 'rb'以二进制形式读入标
        ↪ 签文件
18          magic, n = struct.unpack('>II', lbpath.read(8))
19          labels = np.fromfile(lbpath, dtype=np.uint8)
20      with open(images_path, 'rb') as imgpath:
21          magic, num, rows, cols = struct.unpack('>IIII',
            ↪ imgpath.read(16))
22          images = np.fromfile(imgpath,
            ↪ dtype=np.uint8).reshape(len(labels), 784)  # 转化为图像矩
            ↪ 阵 28*28
23      return images, labels
24
25  def print_same(X, y, num=0, line=2):
26      """
27      Args:
28          X: 图像矩阵
29          y: 标签向量
30          num: 打印的数字
31          line: 打印行数
32      Returns: 打印图像矩阵中标签为 num 的前 line*5 个图片
33      """
34      fig, ax = plt.subplots(nrows=line, ncols=5, sharex=True,
        ↪ sharey=True)
35
36      ax = ax.flatten()
37      for i in range(line * 5):
38          # 将序号为 7 的图像切片取出，每一行都是一个 7 图片的数据，然后取
            ↪ 前 i 个即可
39          img = X[y == num][i].reshape(28, 28)
40          ax[i].imshow(img, cmap='Greys', interpolation='nearest')
```

```
41
42      ax[0].set_xticks([])
43      ax[0].set_yticks([])
44      plt.tight_layout()
45      plt.show()
46
47  if __name__ == '__main__':
48      X_train, y_train = load_mnist(os.getcwd())
49      print_same(X_train, y_train, num=6, line=3)
```

计时器 timer.py 用于计算代码运行所用时间.

```
1  # coding:utf-8
2  import time
3
4
5  class Timer:
6      def __init__(self):
7          self.start = time.time()
8
9      def get_time(self):
10         return time.time() - self.start
11
12     def show(self):
13         print('用时: %.4f s' % (time.time() - self.start))
```

前馈神经网络核心代码 ANN.py

```
1  import numpy as np
2  from timer import Timer
3
4
5  class Function:  # 存储各种函数及其导数
6      @staticmethod
7      def sigmoid(x, order=0):
8          """
9          sigmoid(x) = 1/(1+exp(-x))
10         Args:
11             x: 输入变量
```

```
            order: 导数的阶数
        Returns:
            1/(1+exp(-x))
        """
        assert order < 2  # 只处理 0 和 1 阶导
        if order == 0:
            return 1 / (1 + np.exp(-x))
        return Function.sigmoid(x) * (1 - Function.sigmoid(x))

    @staticmethod
    def tanh(x, order=0):
        """
        tanh(x) = (exp(x) - exp(-x)) / (exp(x) + exp(-x))
        """
        assert order < 2
        if order == 0:
            return (np.exp(x) - np.exp(-x)) / (np.exp(x) +
             ↪  np.exp(-x))
        return 1 - np.power(Function.tanh(x), 2)

    @staticmethod
    def ReLU(x, order=0):
        """
        ReLU(x) = max(0, x)
        """
        assert order < 2
        if isinstance(x, int):
            if order == 0:
                return max(x, 0)
            if x > 0:
                return 1
            return 0
        n = x.shape[0]
        if order == 0:
            for i in range(n):
                x[i] = max(x[i], 0)
            return x
        for i in range(n):
            if x[i] > 0:
                x[i] = 1
```

```
51          return x
52
53      @staticmethod
54      def sse(x, y=0, order=0):
55          """
56          平方损失函数 Quadratic Loss Function
57          误差平方和 Sum of Squared Errors
58          sse = sum((x-y)^2)
59          """
60          assert order < 2
61          if order == 0:
62              return np.sum((x - y) ** 2)
63          return 2 * (x - y)
64
65      @staticmethod
66      def cross_entropy(x, y=0, order=0):
67          """
68          交叉熵损失函数 Cross-Entropy Loss Function
69          -log(x[y])
70          """
71          assert order < 2
72          n = y.shape[0]
73          c = 0  # 样本的标签
74          for i in range(n):
75              if y[i] == 1:
76                  c = i
77          x = x / np.sum(x)  # 转成概率分布形式
78          if order == 0:
79              return -np.log(x[c])
80          ret = np.zeros(n).reshape(n, 1)
81          ret[c] = -1 / x[c]
82          return ret
83
84  # 参考下面文章，利用字典将字符串和静态函数相对应
85  #
    ↪ https://stackoverflow.com/questions/41921255/staticmethod-object-is-not-call
86  Function.switch = {
87      'sigmoid': Function.sigmoid,
88      'tanh': Function.tanh,
89      'ReLU': Function.ReLU,
```

```
    'sse': Function.sse,
    'cross_entropy': Function.cross_entropy,
}


class Layer:  # 神经网络中的每一层
    n, m = None, None  # n 为输入维数，m 为输出维数
    weight, bias = None, None  # 权重矩阵，偏置向量
    grad_weight, grad_bias = None, None  # 权重梯度，偏置梯度
    activation, regular = None, None  # 激活函数，正则化方法
    R = None  # 活跃矩阵，与 dropout 参数相关，计算每一层所使用的神经元

    def __init__(self, n, m, activation='sigmoid', dropout=0):
        self.n, self.m = n, m
        self.activation = Function.switch[activation]
        # 初始化偏置向量和权重矩阵
        self.bias = np.zeros((m, 1))
        self.weight = np.random.rand(m, n) - 0.5
        self.dropout = dropout  # 设定抛弃率

    def random_dropout(self):
        self.R = np.random.binomial(1, 1-self.dropout, self.m) *
        ↪ np.identity(self.m)  # 初始化活跃矩阵

    def forward(self, x_input):  # 向前传播，计算当前层的激活值
        z = self.R @ (self.weight @ x_input + self.bias)
        return z, self.activation(z)

    def back(self, a, z, delta):  # 反向传播
        """
        设当前层为 l，计算当前层的偏置和权重的梯度，并求出 l-1 层的 delta
    ↪ 值
        Args:
            a: l-1 层的激活值
            z: l-1 层的加权值 z = w*a+b
            delta: l 层的 delta 值
        Returns:
            delta, (np.array, list): l-1 层的 delta 值和权重梯度和偏置
    ↪ 梯度展成行向量的结果
        """
```

```
127         grad_weight = self.R @ delta @ a.T
128         grad_bias = self.R @ delta
129         return self.activation(z, order=1) * (self.weight.T @ (self.R
            ↪ @ delta)), [grad_weight, grad_bias]
130
131
132 class Model:  # 神经网络模型
133     tot_input, output = None, None
134
135     def __init__(self, *args, mu=1, loss='sse', test_input=None,
            ↪ test_output=None,
136                  test_rate=0.1):
137         self.layers = list(args)  # 神经网络结构
138         self.mu = mu  # 学习率
139         self.loss = Function.switch[loss]  # 损失函数
140         self.test_input, self.test_output = test_input, test_output
            ↪ # 测试集用于计算准确率，不用于训练
141         self.memory = False  # 判断是否需要记录，每次更新梯度次数时的准
            ↪ 确率
142         self.accuracy = []  # 用于存储准确率
143         self.test_rate = test_rate  # 如果存储每次梯度更新后的准确度，
            ↪ 计算准确度所需的测试集大小占比
144
145     def add(self, *args):  # 在末尾继续增加一层
146         self.layers.append(args)
147
148     def propagation(self, x_input, output=None):
149         """
150         完成一次传播（包括前向传播和反向传播计算梯度），如果 output 没有输
    ↪ 入则只用于做预测
151         Args:
152             x_input: 输出参数
153             output: 期望输出
154         Returns:
155             总梯度
156         """
157         grad = []  # 用 list 存储总梯度
158         L = len(self.layers)  # 网络总层数
159         a = []  # 向前传播中，每一层的激活值，加权值
160         n = np.size(x_input)
```

```
161             x_input = x_input.reshape(n, 1)
162             a.append((x_input, 0))
163             for i in range(L):  # 向前传播
164                 layer = self.layers[i]
165                 layer.random_dropout()  # 每次训练随机产生活跃矩阵
166                 a.append(layer.forward(a[i][0]))
167             y, layer = a[L][1], self.layers[L - 1]  # 预测值 y, 最后一层
                ↪ layer
168             new_output = (y == np.max(y)).astype(int)  # 当前正规化预测值
169             if output is None:
170                 return new_output
171             m = np.size(output)
172             output = output.reshape(m, 1)
173             loss = self.loss(y, output)
174             delta = layer.activation(a[L][0], order=1) * self.loss(y,
                ↪ output, order=1)  # 初始化误差值
175             for i in range(L-1, -1, -1):  # 反向传播, 计算梯度
176                 layer = self.layers[i]  # 倒着取值
177                 delta, g = layer.back(*a[i], delta)  # 计算每一层的梯度
178                 grad = g + grad
179             return grad, new_output, loss
180
181         def batch(self, arr, length):
182             """
183             计算每一个 Mini-batch, 并对网络参数进行调整
184             Args:
185                 arr: 用 list 存储当前 batch 中的训练编号
186                 length: arr 的长度
187             Returns:
188                 平均误差
189             """
190             L = len(self.layers)  # 总层数
191             grad, loss = None, None  # 当前 batch 的平均梯度, 和 loss 的平
                ↪ 均值
192             for i in arr:  # 开始训练 (此处可以加入多线程)
193                 x_input = np.asarray(self.tot_input[i]).T
194                 output = self.output[i]
195                 g, new_output, l = self.propagation(x_input, output)
196                 if grad is None:
197                     grad = g
```

```
198                 loss = l
199             else:
200                 for j in range(2 * L):
201                     grad[j] = grad[j] + g[j]  # 对每个数据的梯度进行累
                        ↪ 加
202                 loss += l
203         for i in range(L):  # 更新梯度
204             j = i * 2
205             grad[j] = grad[j] / length  # 对权重梯度求平均值
206             self.layers[i].weight = self.layers[i].weight - self.mu *
                ↪ grad[j]
207             grad[j + 1] = grad[j + 1] / length  # 对偏置梯度求平均值
208             self.layers[i].bias = self.layers[i].bias - self.mu *
                ↪ grad[j + 1]
209         return loss / length  # 返回平均误差
210
211     def fit(self, tot_input, output, epoch=5, batch=1, memory=False,
        ↪ tot_update=None):
212         """
213         Args:
214             tot_input: 总的输入训练数据
215             output: 总的输出训练数据
216             epoch: 训练数据集次数
217             batch: 每个 Mini-batch 的大小
218             memory: 是否需要记录准确值
219             tot_update: 总梯度更新次数 (如果不为 None, 则 epoch 参数无
    ↪ 效, 一直迭代直到梯度更新次数达到 tot_update)
220         """
221         update_counter = 0
222         if tot_update is not None:  # 如果总梯度更新次数不为 None
223             epoch = int(1e9)  # 设置 epoch 为最大值
224         self.memory = memory
225         timer = Timer()
226         self.tot_input, self.output = tot_input, output
227         n, m = np.shape(tot_input)  # n 为总数据量, m 为输入数据向量的
            ↪ 维数
228         B = n // batch  # Mini-batch 的总个数
229         for _ in range(epoch):
230             rand = np.random.permutation(n)
231             for i in range(B):
```

```
232                 st, en = i * batch, (i + 1) * batch  # 划分数据集
233                 loss = self.batch(rand[st:en], en - st)  # 用每个
                    ↪ batch 对网络参数进行调整
234                 update_counter += 1
235                 if update_counter == tot_update:
236                     break
237             if self.memory:
238                 self.accuracy.append(self.evaluate())
239                 # print('accuracy: {}'.format(self.accuracy[-1]))
240             print('epoch {}, 准确率: {}, '.format(_+1,
                ↪ self.accuracy[-1]), end='')
241             timer.show()
242             if update_counter == tot_update:
243                 break
244
245     def evaluate(self, test_rate=None):
246         """
247         评估模型的准确率
248         Args:
249             test_rate: 多大比例的测试样本 (随机选取),
250             如果传入是 None 默认使用 self.test_rate, 计算每次更新梯度后
    ↪ 的准确率
251         Returns:
252             准确率
253         """
254         if test_rate is None:
255             test_rate = self.test_rate
256         correct = 0
257         n, m = np.shape(self.test_input)
258         rand = np.random.permutation(n)  # 打乱测试样本
259         n = int(n * test_rate)  # 测试样本个数
260         for i in rand[:n]:
261             a_input = self.test_input[i].reshape(m, 1)
262             new_output = self.propagation(a_input)
263             if list(self.test_output[i]) == list(new_output):
264                 correct += 1
265         return correct / n
266
267
268 if __name__ == '__main__':
```

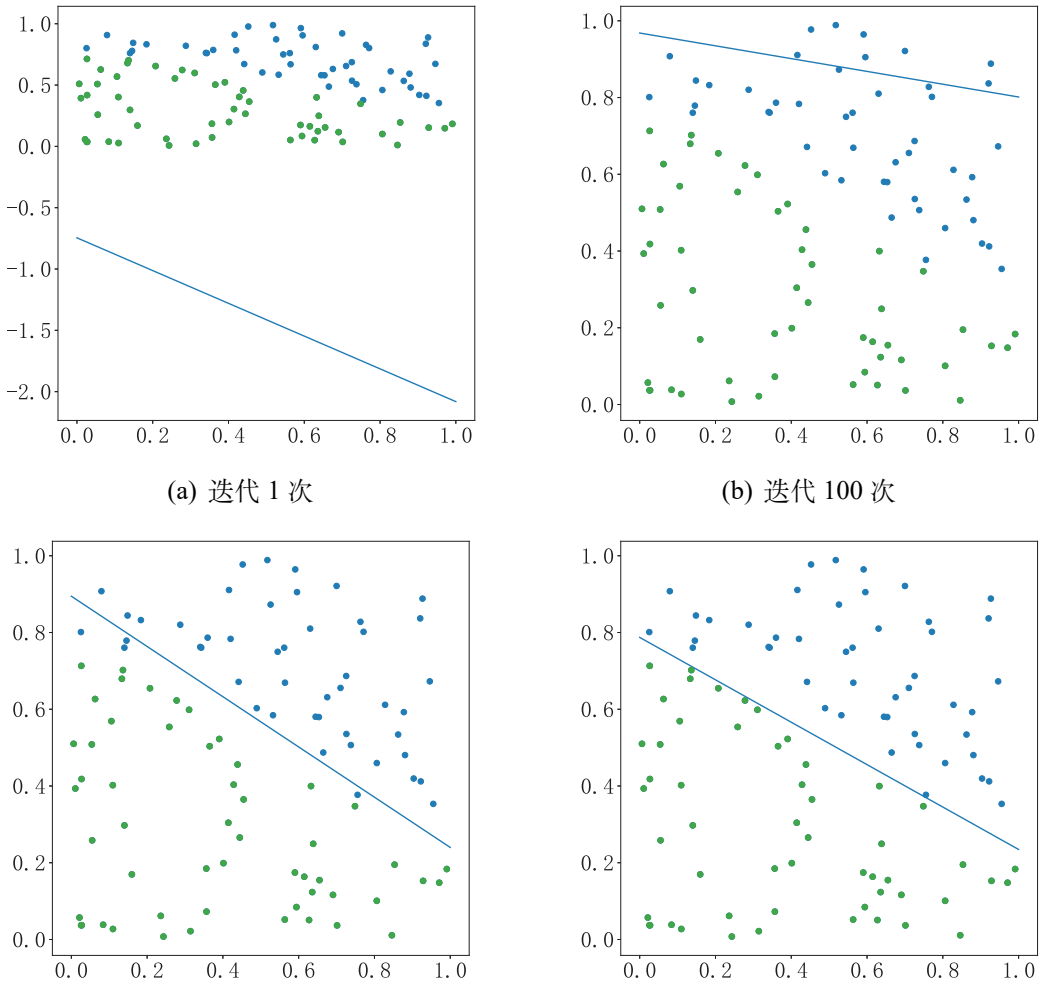
```
269     pass
```

主程序 Main.py

```
1  import os
2  import numpy as np
3  import Input  # 数据读入
4  import ANN  # 神经网络核心部分
5
6
7  def change_output(y_output):
8      #  转化输出为列向量
9      output = []
10     for i in y_output:
11         tmp = [0] * 10
12         tmp[i] = 1
13         output.append(tmp)
14     return np.array(output)
15
16 if __name__ == '__main__':
17     X_train, y_train = Input.load_mnist(os.getcwd())
18     X_check, y_check = Input.load_mnist(os.getcwd(), 't10k')
19     X_train = X_train / 255
20     X_check = X_check / 255
21     y_train = change_output(y_train)
22     y_check = change_output(y_check)
23
24     dropout = 0
25     #  创建神经网络模型
26     model = ANN.Model(
27         ANN.Layer(784, 32, activation='tanh', dropout=dropout),  # 神
           ↪  经网络结构
28         ANN.Layer(32, 10),
29         loss='sse',
30         test_input=X_check,  # 测试集用于计算准确率，不用于训练
31         test_output=y_check,
32         test_rate=0.1,  # 如果存储每次梯度更新后的准确度，即 memory=1,
           ↪  计算准确度所需的测试集大小占比
33     )
34     batch = 100  # batch_size
```

```
35      model.fit(X_train, y_train, epoch=10, batch=batch, memory=True)
36      end_accuracy = model.evaluate(test_rate=1)
37      print(end_accuracy)  # 输出最终准确率
38
39      if model.memory:  # 保存准确率
40          with open('diff_accuracy.txt', 'a', encoding='utf-8',
            ↪ errors='ignore') as file:
41              file.write('网络结构: 784-32-10, 使用 tanh 作为激活函数,
                ↪ Batch Size={}, dropout={}\n'.
42                         format(batch, dropout))
43              file.write('总训练次数:
                ↪ {}\n'.format(len(model.accuracy)))
44              for i in model.accuracy:
45                  file.write(str(i) + ' ')
46              file.write('\n最终准确率: {}\n\n'.format(end_accuracy))
47
48  """
49  Difference between numpy.array shape (R, 1) and (R,):
50  https://stackoverflow.com/questions/22053050/difference-between-numpy-array-shap
51  """
```

B 感知器迭代过程图



(a) 迭代 1 次

(b) 迭代 100 次

(c) 迭代 500 次

(d) 迭代 2000 次

图 17: 迭代过程图