一份不太专业的多层感知机介绍

Dey Z

目录

摘要

多层感知机可用于解决分类与回归问题，它是两个向量集之间的一个映射。根据通用近似定理，多层感知机隐藏层的输出的加权和可被视为某个多元函数的值。经过参数优化，该多元函数能在一定程度上拟合诸多连续函数。

小批量梯度下降法是一种机器学习领域常见的参数优化方法，它可被用于训练多层感知机。梯度下降的难点在于计算损失函数对各层参数的偏导数，而基于链式法则实现的反向传播算法则较好地解决了这一问题。

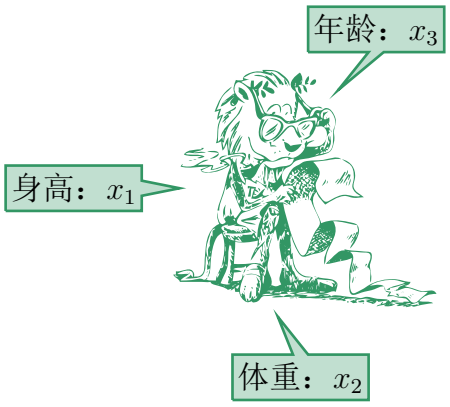
关键词　多层感知机; 神经网络; 通用近似定理; 梯度下降; 反向传播

1　模型结构

1.1　样本与数据

样本是具体或抽象的事物，是研究对象。分类（Classification）任务是确定给定样本所属的类别，是一种定性的任务，如鸢尾花品种分类。One-hot 向量是仅有一个分量为 1，其余分量为 0 的向量。在样本被划分为 $n(n \geqslant 2)$ 个类别的多分类任务中，不同的 n 维 One-hot 向量表示不同的类别。从样本的角度进一步看，样本的 One-hot 向量的不同分量实际上对应不同的类别。一个分量可被视为一个真值或概率值，它表明样本是否属于该分量对应的类别。当一个分量为 1 时，样本属于该分量对应的类别，否则不属于。回归（Regression）任务是确定给定样本在特定情形下对应的具体数值或数值组，是一种定量的任务，如波士顿房价预测。

特征（Feature）是样本的属性，其值是对样本本身的描述。特征向量是样本全体共有的一组给定特征的值按给定顺序组成的向量，它表征了样本。标签（Label）是样本全体被赋予的具有意义的属性或属性组，其值由样本的特征向量决定。标签向量是标签值的数学抽象。在分类任务中，样本标签值是样本所属的类别，标签向量是 One-hot 向量。在回归任务中，标签值是若干有序实数，标签向量是分量为这些实数的实向量。若标签值为一个实数，则标签向量为 1 维实向量。



样本序号	特征向量 $[x_1, x_2, x_3]^{\mathrm{T}}$	标签值	标签向量
1	$[171, 52, 58]^{\mathrm{T}}$	正常	$[0, 1, 0]^{\mathrm{T}}$
2	$[165, 45, 17]^{\mathrm{T}}$	偏轻	$[1, 0, 0]^{\mathrm{T}}$
3	$[179, 91, 25]^{\mathrm{T}}$	超重	$[0, 0, 1]^{\mathrm{T}}$
4	$[175, 61, 37]^{\mathrm{T}}$	正常	$[0, 1, 0]^{\mathrm{T}}$

图 1　分类任务中的样本数据示例[1]

[1]图中 TeX Lion 为 Duane Bibby 原创，后文不再说明。

特征空间是所有可能的特征向量的集合，标签空间是所有可能的标签向量的集合。存在一个真实的映射，它是一个从特征空间到标签空间的映射，记为 f_t。由此可知，每个特征向量都有唯一一个与之对应的标签向量。样本空间是所有可能的特征向量及其对应的标签向量的组合的集合。f_t 未必是已知的，于是对给定的特征向量，其对应的标签向量是存在的，但可能是未知的。

1.2 预测与学习

多层感知机（Multi-layer Perceptron，MLP）是一种机器学习（Machine Learning）模型，它常用于分类任务与回归任务中。通俗而言，多层感知机的任务是根据给定样本的特征向量预测其对应的标签向量。在针对一个样本的预测任务中，多层感知机的输入是该样本的特征向量，输出是该样本的标签向量预测值。

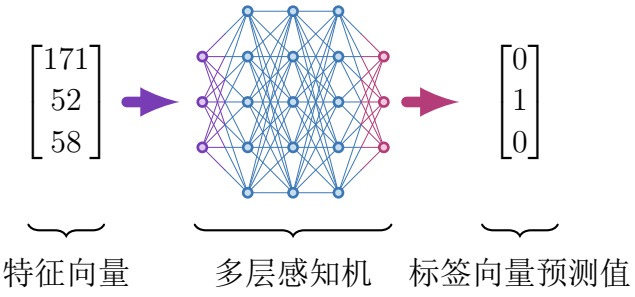


图 2　多层感知机的输入与输出示例

在简单任务中，样本特征向量和标签向量之间的关系 f_t 是容易被人为确定并设计为固定算法的，而复杂任务中则不然。复杂任务要解决的是诸如手写数字识别等不易发现 f_t 的问题，这些问题是传统的算法逻辑基本固定的确定性程序难以应付的。而多层感知机能较好地解决这些问题，它能从数据中学习，从而构建出接近或吻合 f_t 的关系。

多层感知机模型内部有大量参数，它们参与了处理特征向量从而生成标签向量预测值的过程。学习的本质是利用大量已知特征向量及其对应标签向量的样本数据来调整模型内部参数，这一过程称为训练（Training），这些用于训练的已知样本的集合称为训练集（Training Set）。训练集可以认为是样本空间的一个很小的子集。在训练的某一过程中，多层感知机接收某个训练集子集的特征向量组及其对应的标签向量组，同时计算出标签向量预测值组。之后，它采用特定的参数优化方法，根据标签向量组和标签向量预测值组调整内部参数，从而提高预测的准确率。

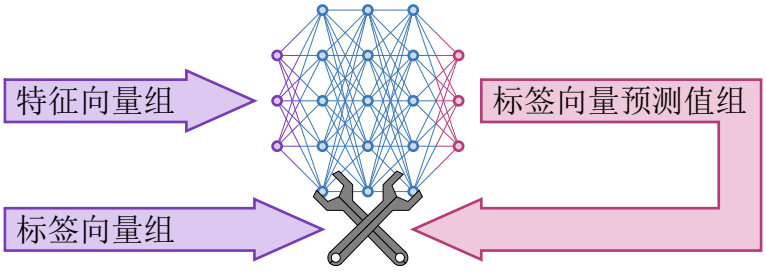


图 3　多层感知机的参数调整

1.3 数学模型

1.3.1 神经网络

多层感知机是一种人工神经网络（Artificial Neural Network）模型，其设计思想最初源于生物学意义上的神经网络[1][2]。从任务和计算角度看，多层感知机是从特征空间到另一个向量集的映射，记为 $\widehat{f}$。

假设在一个任务中，所有样本独立同分布（Independent and Identically Distributed），样本的特征向量是 $a(a \geqslant 1)$ 维的，标签向量是 $q(q \geqslant 1)$ 维的。现针对任务，分别给出 $M(M \geqslant 2)$ 个 $n^{(m)} \times n^{(m-1)}$ 矩阵 $\boldsymbol{W}^{(m)}$、$n^{(m)} \times 1$ 列向量 $\boldsymbol{b}^{(m)}$ 和函数 $\varphi^{(m)}$（其中，$m = 1, 2, \cdots, M$，$n^{(0)} = a$，$n^{(M)} = q$，$n^{(m)} \geqslant 1$）。方便起见，记 $\boldsymbol{W}^{(m)}$ 第 $i(1 \leqslant i \leqslant n^{(m)})$ 行的转置为 $n^{(m-1)} \times 1$ 列向量 $\boldsymbol{w}_i^{(m)}$，而 $\boldsymbol{w}_i^{(m)}$ 的第 $j(1 \leqslant j \leqslant n^{(m-1)})$ 个分量为 $w_{i,j}^{(m)}$。又记 $\boldsymbol{b}^{(m)}$ 的第 $i(1 \leqslant i \leqslant n^{(m)})$ 个分量为 $b_i^{(m)}$。现给定 $s(s \geqslant 1)$ 个样本，并声明如下几个变量。

符号	格式	意义
$\boldsymbol{x}$	$n^{(0)} \times 1$ 列向量	特征向量
x_j	标量	特征的值，$\boldsymbol{x}$ 的第 $j(0 \leqslant j \leqslant n^{(0)})$ 个分量
$\boldsymbol{y}$	$n^{(M)} \times 1$ 列向量	标签向量
y_j	标量	$\boldsymbol{y}$ 的第 $j(0 \leqslant j \leqslant n^{(M)})$ 个分量
$\widehat{\boldsymbol{y}}$	$n^{(M)} \times 1$ 列向量	标签向量预测值
$\widehat{y}_j$	标量	$\widehat{\boldsymbol{y}}$ 的第 $j(0 \leqslant j \leqslant n^{(M)})$ 个分量
$\boldsymbol{X}$	$s \times n^{(0)}$ 矩阵	特征矩阵，多层感知机的输入，每一行为一个样本特征向量的转置
$\boldsymbol{x}_i$	$n^{(0)} \times 1$ 列向量	特征向量，$\boldsymbol{X}$ 第 $i(1 \leqslant i \leqslant s)$ 行的转置
$x_{i,j}$	标量	特征的值，$\boldsymbol{x}_i$ 的第 $j(0 \leqslant j \leqslant n^{(0)})$ 个分量
$\boldsymbol{Y}$	$s \times n^{(M)}$ 矩阵	标签矩阵，每一行为一个样本标签向量的转置
$\boldsymbol{y}_i$	$n^{(M)} \times 1$ 列向量	标签向量，$\boldsymbol{Y}$ 第 $i(1 \leqslant i \leqslant s)$ 行的转置
$y_{i,j}$	标量	$\boldsymbol{y}_i$ 的第 $j(0 \leqslant j \leqslant n^{(M)})$ 个分量
$\widehat{\boldsymbol{Y}}$	$s \times n^{(M)}$ 矩阵	多层感知机的输出，每一行为一个样本标签向量预测值的转置
$\widehat{\boldsymbol{y}}_i$	$n^{(M)} \times 1$ 列向量	标签向量预测值，$\widehat{\boldsymbol{Y}}$ 第 $i(1 \leqslant i \leqslant s)$ 行的转置
$\widehat{y}_{i,j}$	标量	$\widehat{\boldsymbol{y}}_i$ 的第 $j(0 \leqslant j \leqslant n^{(M)})$ 个分量
$\boldsymbol{X}^{(m)}$	$s \times n^{(m)}$ 矩阵	
$x_{i,j}^{(m)}$	标量	$\boldsymbol{X}^{(m)}$ 第 $i(1 \leqslant i \leqslant s)$ 行、第 $j(1 \leqslant j \leqslant n^{(m)})$ 列元素
$\boldsymbol{x}^{(m)}$	$n^{(m)} \times 1$ 列向量	$s = 1$ 时的 ${\boldsymbol{X}^{(m)}}^{\mathrm{T}}$
$x_j^{(m)}$	标量	$\boldsymbol{x}^{(m)}$ 的第 $j(1 \leqslant j \leqslant n^{(m)})$ 个分量
$\boldsymbol{Y}^{(m)}$	$s \times n^{(m)}$ 矩阵	
$y_{i,j}^{(m)}$	标量	$\boldsymbol{Y}^{(m)}$ 第 $i(1 \leqslant i \leqslant s)$ 行、第 $j(1 \leqslant j \leqslant n^{(m)})$ 列元素
$\boldsymbol{y}^{(m)}$	$n^{(m)} \times 1$ 列向量	$s = 1$ 时的 ${\boldsymbol{Y}^{(m)}}^{\mathrm{T}}$
$y_j^{(m)}$	标量	$\boldsymbol{y}^{(m)}$ 的第 $j(1 \leqslant j \leqslant n^{(m)})$ 个分量

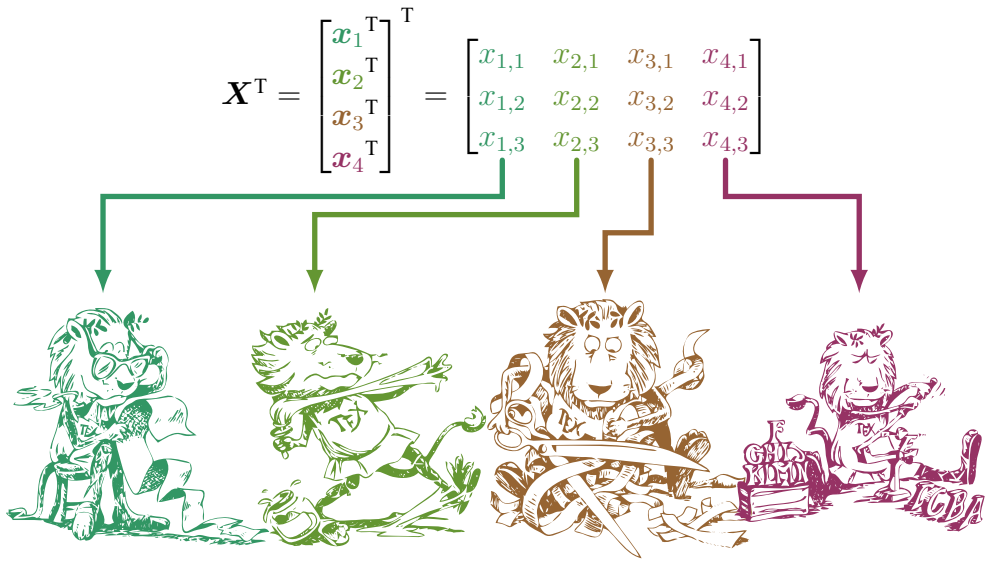


图 4　4×3 特征矩阵 $\boldsymbol{X}$ 的转置示例

现给出 M 个每一列均为 $\boldsymbol{b}^{(m)}$ 的 $n^{(m)} \times s$ 矩阵 $\boldsymbol{B}^{(m)} = [\boldsymbol{b}^{(m)}, \cdots, \boldsymbol{b}^{(m)}]$，那么多层感知机的输出 $\widehat{\boldsymbol{Y}} = \widehat{f}(\boldsymbol{X})$ 可计算如下

$$\begin{cases} \boldsymbol{X}^{(m)\mathrm{T}} = \boldsymbol{W}^{(m)} \boldsymbol{Y}^{(m-1)\mathrm{T}} + \boldsymbol{B}^{(m)} \\ \boldsymbol{Y}^{(m)} = \varphi^{(m)}(\boldsymbol{X}^{(m)}) \\ \boldsymbol{Y}^{(0)} = \boldsymbol{X} \\ \boldsymbol{Y}^{(M)} = \widehat{\boldsymbol{Y}} \end{cases} \tag{1-1}$$

其中，$\boldsymbol{W}^{(m)}$ 被称为权重（Weight）矩阵，$\boldsymbol{b}^{(m)}$ 被称为偏置（Bias）列向量，$\varphi^{(m)}$ 被称为激活函数（Activation Function），且约定 $(\varphi^{(m)}(\boldsymbol{X}^{(m)}))_{i,j} = \varphi^{(m)}(x_{i,j}^{(m)})$。$m$ 是多层感知机的层号，第 0 层是输入层，第 M 层是输出层，其余层是隐藏层。

当 $s = 1$ 时，由(1-1)式，有

$$y_j^{(m)} = \varphi^{(m)}(\boldsymbol{w}_j^{(m)\mathrm{T}} \boldsymbol{y}^{(m-1)} + b_j^{(m)}) \tag{1-2}$$

称(1-2)式为多层感知机第 m 层的第 j 个神经元（Neuron）。

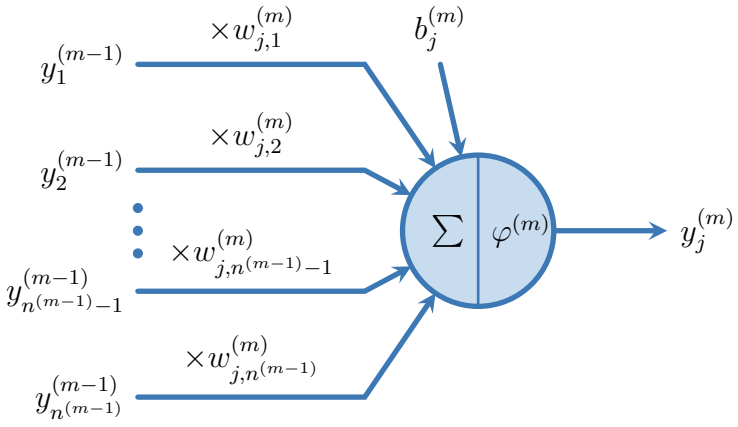


图 5　多层感知机第 m 层的第 j 个神经元结构

神经元是多层感知机的基本单位。由于多层感知机的神经元分层排列，每个神经元只接收前一层神经元的输出作为输入，各层神经元之间不存在反馈，故多层感知机属于前馈神经网络（Feedforward Neural Network）。由于每个神经元均接收前一层所有神经元的输出，故这种神经网络结构又是全连接的（Fully Connected）。图 6所示的神经元连接线上的数据为多层感知机第 m 层的第 $n^{(m)}$ 个神经元上待求和的加权输入。其中 $y_1^{(m-1)}$、$y_2^{(m-1)}$、$y_{n^{(m-1)}-1}^{(m-1)}$ 和 $y_{n^{(m-1)}}^{(m-1)}$ 分别为多层感知机第 $m-1$ 层的第 1、2、$n^{(m-1)}-1$ 和 $n^{(m-1)}$ 个神经元的输出。

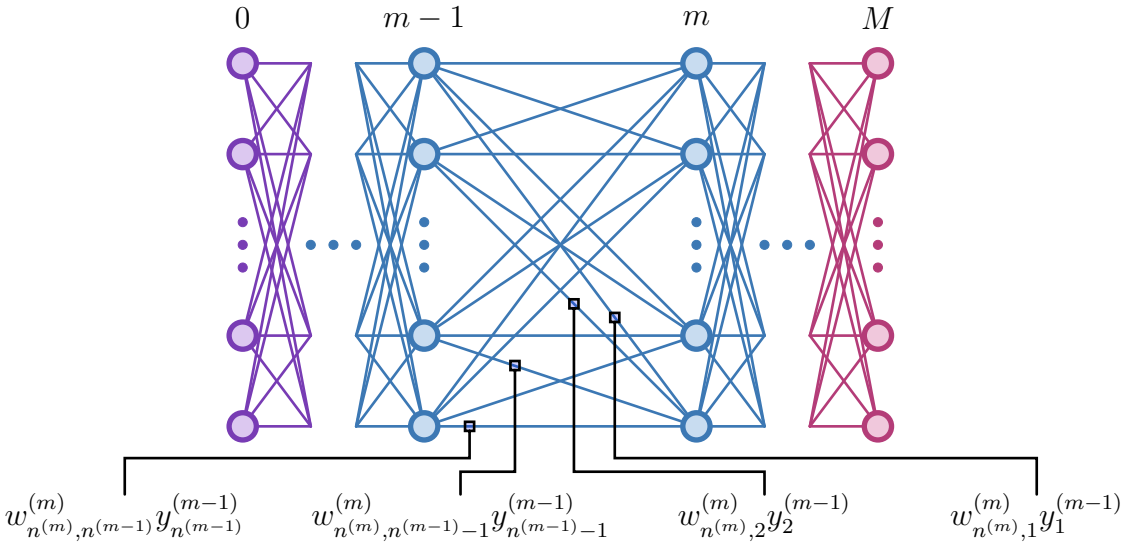


图 6　多层感知机中第 m 层的第 $n^{(m)}$ 个神经元的加权输入

1.3.2　损失函数

损失函数（Loss Function）用于衡量多层感知机的输出与实际的样本标签向量之间的差异程度，其值为非负实数。一般而言，损失函数是一个关于标签向量预测值组 $\widehat{\boldsymbol{Y}}$ 和标签向量组 $\boldsymbol{Y}$ 的函数，记为 $\mathcal{L}$。令 $L \in \mathbb{R}^*$ 为损失函数值，有

$$L = \mathcal{L}(\widehat{\boldsymbol{Y}}, \boldsymbol{Y}) \tag{2-1}$$

由于多层感知机是映射 $\widehat{f}$，且 $\widehat{\boldsymbol{Y}} = \widehat{f}(\boldsymbol{X})$，于是有

$$L = \mathcal{L}(\widehat{f}(\boldsymbol{X}), \boldsymbol{Y}) \tag{2-2}$$

由此可知，损失函数是一个从样本空间到 $\mathbb{R}^*$ 的函数。对一个较好的多层感知机模型，它在整个特征空间上的输出与对应的标签向量之间的差异应该尽可能小，换言之，损失函数在整个样本空间上的值应该尽可能小。从损失函数的角度看，训练就是调整参数 $\boldsymbol{W}^{(m)}$ 和 $\boldsymbol{b}^{(m)}$，从而找到一个使得 $\mathcal{L}$ 在整个样本空间上的函数值尽可能小的映射 $\widehat{f}$。

一般地，训练只能基于样本规模十分有限的训练集。因此，可能出现损失函数在训练集上的值很小，而在训练集以外的部分样本空间上的值较大的现象，这种现象一般为过拟合（Overfitting）。而损失函数在整个样本空间上的值均较大的现象一般为欠拟合（Underfitting）。

1.3.2.1　交叉熵损失函数

在分类任务中，常用交叉熵（Cross Entropy）作为损失函数。对前面给定的 $\boldsymbol{y}$ 和 $\widehat{\boldsymbol{y}}$，$\widehat{\boldsymbol{y}}$ 一般是一个属于空间 $(0,1)^{n^{(M)}}$ 的向量，其第 $j(1 \leqslant j \leqslant n^{(M)})$ 个分量可被视为样本属于第 j 个类

别的概率，于是交叉熵损失为

$$l_{CE} = \sum_{j=1}^{n^{(M)}} y_j \ln \widehat{y}_j$$

对于(1-1)式，有交叉熵损失

$$\mathcal{L}_{CE} = \frac{1}{s} \sum_{i=1}^{s} \sum_{j=1}^{n^{(M)}} y_{i,j} \ln \widehat{y}_{i,j} \tag{2-3-1}$$

$\mathcal{L}_{CE}$ 对元素 $\widehat{y}_{p,q}$ 的偏导数为

$$\frac{\partial}{\partial \widehat{y}_{p,q}} \mathcal{L}_{CE} = -\frac{1}{s} \frac{y_{p,q}}{\widehat{y}_{p,q}} \tag{2-3-2}$$

1.3.2.2 均方差损失函数

均方误差（Mean Squared Error，MSE），简称均方差。在回归任务中，常用均方差作为损失函数。对同属实向量的 $\boldsymbol{y}$ 和 $\widehat{\boldsymbol{y}}$，均方差损失为

$$l_{MSE} = \frac{1}{n^{(M)}} \sum_{j=1}^{n^{(M)}} (\widehat{y}_j - y_j)^2$$

对于(1-1)式，有均方差损失

$$\mathcal{L}_{MSE} = \frac{1}{n^{(M)} s} \sum_{i=1}^{s} \sum_{j=1}^{n^{(M)}} (\widehat{y}_{i,j} - y_{i,j})^2 \tag{2-4-1}$$

$\mathcal{L}_{MSE}$ 对元素 $\widehat{y}_{p,q}$ 的偏导数为

$$\frac{\partial}{\partial \widehat{y}_{p,q}} \mathcal{L}_{MSE} = \frac{2}{n^{(M)} s} (\widehat{y}_{p,q} - y_{p,q}) \tag{2-4-2}$$

1.3.3 激活函数

激活函数一般指非线性激活函数，其主要作用是为多层感知机引入非线性能力或控制神经元输出范围。

在(1-1)式中，若所有的激活函数 $\varphi^{(m)}(x) = x$，则

$$\begin{aligned} \widehat{\boldsymbol{Y}}^{\mathrm{T}} =& \boldsymbol{W}^{(M)} (\cdots \boldsymbol{W}^{(2)} (\boldsymbol{W}^{(1)} \boldsymbol{X}^{\mathrm{T}} + \boldsymbol{B}^{(1)}) + \boldsymbol{B}^{(2)} \cdots) + \boldsymbol{B}^{(M)} \\ =& \boldsymbol{U} \boldsymbol{X}^{\mathrm{T}} + \boldsymbol{V} \end{aligned}$$

其中

$$\boldsymbol{U} = \boldsymbol{W}^{(M)} \cdots \boldsymbol{W}^{(2)} \boldsymbol{W}^{(1)}$$

$$\boldsymbol{V} = \boldsymbol{W}^{(M)} \cdots \boldsymbol{W}^{(2)} \boldsymbol{B}^{(1)} + \boldsymbol{W}^{(M)} \cdots \boldsymbol{W}^{(3)} \boldsymbol{B}^{(2)} + \cdots + \boldsymbol{W}^{(M)} \boldsymbol{B}^{(M-1)} + \boldsymbol{B}^{(M)}$$

可见，当多层感知机的所有神经元均不被施以非线性激活函数时，该多层感知机实际上是一个仿射变换（Affine Transformation）。无论如何调整参数 $\boldsymbol{W}^{(m)}$ 和 $\boldsymbol{b}^{(m)}$，它始终与不属于仿射变换的非线性变换（Nonlinear Transformation）存在本质区别。换言之，它无法更好地拟合非线性变换，故需要依靠非线性激活函数引入非线性能力。

在训练过程中，随着参数 $\boldsymbol{W}^{(m)}$ 和 $\boldsymbol{b}^{(m)}$ 的不断调整，神经元的输出可能超出计算机数据类型表示范围，即可能发生数据溢出现象。此时应当适当“挤压”输出数据，将其映射到某个有界区间上。而在分类任务中，输出层神经元的输出通常也被控制在区间 $(0,1)$ 上，并被视为样本属于某一类别的概率。故此，多层感知机需要依靠非线性激活函数控制神经元输出范围。

非线性激活函数种类较多，常见的有以 Logistic 和 tanh 等为代表的 Sigmoid 型和以 ReLU、Leaky ReLU 和 Parametric ReLU 等为代表的 ReLU 型。激活函数有各自的优缺点，对训练速度和模型精度等方面的影响各不相同。例如 ReLU 型计算简单，但容易造成数据溢出，而 Sigmoid 型反之。下面给出几个较为常见的激活函数及其导数的计算式。

1.3.3.1 Logistic 函数

Logistic 函数是一个经典的激活函数，其函数图像形如“S”，故常被称为 Sigmoid 函数。以下二式分别为 Logistic 及其导数的计算方法。

$$\varphi_{lg} = \frac{1}{1+\mathrm{e}^{-x}} \tag{3-1-1}$$

$$\varphi'_{lg} = \varphi_{lg}(1-\varphi_{lg}) \tag{3-1-2}$$

1.3.3.2 tanh 函数

tanh 即双曲正切函数（Hyperbolic Tangent Function），它与 Logistic 同属 Sigmoid 型函数，但其输出以 0 为均值。以下二式分别为 tanh 及其导数的计算方法。

$$\varphi_{th} = \frac{1-\mathrm{e}^{-2x}}{1+\mathrm{e}^{-2x}} \tag{3-2-1}$$

$$\varphi'_{th} = 1-\varphi_{th}^2 \tag{3-2-2}$$

1.3.3.3 ReLU 函数

ReLU 即修正线性单元（Rectified Linear Unit）。以下二式分别为 ReLU 及其导数的计算方法。

$$\varphi_r = \max(0,x) \tag{3-3-1}$$

$$\varphi'_r = \begin{cases} 1, & x \geqslant 0 \\ 0, & x < 0 \end{cases} \tag{3-3-2}$$

1.3.3.3 Leaky ReLU 函数

Leaky ReLU 在 ReLU 的基础上保留了负方向的信息。令 $\alpha > 0$，则以下二式分别为 Leaky ReLU 及其导数的计算方法。

$$\varphi_{kr} = \begin{cases} x, & x \geqslant 0 \\ \alpha x, & x < 0 \end{cases} \tag{3-4-1}$$

$$\varphi'_{kr} = \begin{cases} 1, & x \geqslant 0 \\ \alpha, & x < 0 \end{cases} \tag{3-4-2}$$

一般地，取 α 为 0.01 左右。需要指明的是，Leaky ReLU 和 ReLU 在 $x=0$ 时均不可导，因而(3-4-2)和(3-3-2)在 $x=0$ 处的值均是人为规定的。

1.3.3.2 Softmax 函数

Softmax 函数是一个比较特殊的激活函数，常用在分类任务中，一般为多层感知机输出层的激活函数。在(1-1)式中，对 $i(1\leqslant i\leqslant s)$、$p(1\leqslant p\leqslant n^{(M)})$ 以及 $q(1\leqslant q\leqslant n^{(M)},q\neq p)$，有 Softmax 及其偏导数的计算方法如下

$$\widehat{y}_{i,p}=\frac{\mathrm{e}^{x_{i,p}^{(M)}}}{\sum\limits_{j=1}^{n^{(M)}}\mathrm{e}^{x_{i,j}^{(M)}}} \tag{3-5-1}$$

$$\frac{\partial}{\partial x_{i,p}^{(M)}}\widehat{y}_{i,p}=\widehat{y}_{i,p}(1-\widehat{y}_{i,p}) \tag{3-5-2}$$

$$\frac{\partial}{\partial x_{i,q}^{(M)}}\widehat{y}_{i,p}=-\widehat{y}_{i,p}\widehat{y}_{i,q} \tag{3-5-3}$$

1.4 通用近似定理

通用近似定理（Universal Approximation Theorem）[3][4][5]是多层感知机的理论依据，它表明多层感知机对连续函数具有拟合能力。下面是通用近似定理的一种表述。

令 $\varphi(\bullet)$ 为一个连续、有界、单调递增（区别于严格单调递增）的非常数函数，$\boldsymbol{I}_D$ 为 $D(D\geqslant 1)$ 维单位超立方体 $[0,1]^D$，$C(\boldsymbol{I}_D)$ 为定义在 $\boldsymbol{I}_D$ 上的连续函数的集合。对给定的任意实数 $\varepsilon>0$ 和任意函数 $g\in C(\boldsymbol{I}_D)$，存在一个自然数 $N\in\mathbb{N}^+$，一组实数 $v_i\in\mathbb{R}$、$b_i\in\mathbb{R}$ 和实向量 $\boldsymbol{w}_i\in\mathbb{R}^m$（其中 $i=1,2,\cdots,N$），以至于对任意 $\boldsymbol{x}\in\boldsymbol{I}_D$，可以定义

$$F(\boldsymbol{x})=\sum_{i=1}^{N}v_i\varphi(\boldsymbol{w}_i^{\mathrm{T}}\boldsymbol{x}+b_i)$$

使得

$$|F(\boldsymbol{x})-g(\boldsymbol{x})|<\varepsilon$$

换言之，形如 $F(\boldsymbol{x})$ 的函数在 $C(\boldsymbol{I}_D)$ 中是稠密的。

回顾(1-2)的神经元模型，令多层感知机第 $m+1(1\leqslant m\leqslant M-1)$ 层的第 $k(0\leqslant k\leqslant n^{(m+1)})$ 个神经元的激活函数的输入为 $x_k^{(m+1)}$，则

$$\begin{aligned}x_k^{(m+1)}&=\boldsymbol{w}_k^{(m+1)^{\mathrm{T}}}\boldsymbol{y}^{(m)}+b_k^{(m+1)}\\&=\sum_{j=1}^{n^{(m)}}w_{k,j}^{(m+1)}y_j^{(m)}+b_k^{(m+1)}\\&=\sum_{j=1}^{n^{(m)}}w_{k,j}^{(m+1)}\varphi^{(m)}(\boldsymbol{w}_j^{(m)^{\mathrm{T}}}\boldsymbol{y}^{(m-1)}+b_j^{(m)})+b_k^{(m+1)}\end{aligned}$$

假设 $\boldsymbol{y}^{(m-1)}\in[-C,C]^{n^{(m-1)}}(C\in\mathbb{R}^+)$，令

$$\boldsymbol{y}^*=\frac{1}{2C}\boldsymbol{y}^{(m-1)}+[\frac{1}{2},\cdots,\frac{1}{2}]^{\mathrm{T}}$$

则 $\boldsymbol{y}^* \in [0,1]^{n^{(m-1)}}$，于是有

$$x_k^{(m+1)} = \sum_{j=1}^{n^{(m)}} w_{k,j}^{(m+1)} \varphi^{(m)}(2C\boldsymbol{w}_j^{(m)\mathrm{T}}\boldsymbol{y}^* + (b_j^{(m)} - \boldsymbol{w}_j^{(m)\mathrm{T}}[C,\cdots,C]^{\mathrm{T}})) + b_k^{(m+1)} \qquad (4^*)$$

现令 $g(\bullet)$ 是定义在 $[-C,C]^{n^{(m-1)}}$ 上的连续函数，则

$$g^*(\bullet) = g(2C(\bullet) - [C,\cdots,C]^{\mathrm{T}})$$

是定义在 $[0,1]^{n^{(m-1)}}$ 上的连续函数。忽略偏置 $b_k^{(m+1)}$，易知(4*)式符合通用近似定理，即对连续函数 $g^*(\bullet)$，只要参数 $n^{(m)}$、$w_{k,j}^{(m+1)}$、$\boldsymbol{w}_j^{(m)\mathrm{T}}$ 和 $b_j^{(m)}$ 恰当，(4*)式就能在某种程度上拟合连续函数 $g^*(\bullet)$。由前文已知，参数 $w_{k,j}^{(m+1)}$、$\boldsymbol{w}_j^{(m)\mathrm{T}}$ 和 $b_j^{(m)}$ 可通过训练不断调整，而多层感知机第 m 个隐藏层的神经元个数 $n^{(m)}$ 则不能调整。$n^{(m)}$ 一般是一个固定的常数，它在多层感知机构建之初便确定下来，而其具体值的确定是一项经验性工作。

在激活函数中，ReLU 型函数并不是有界函数，这在直觉上不符合通用近似定理。但事实上，一个输出层为线性激活层（$\varphi^{(M)}(x) = x$），隐藏层为 ReLU 激活层的二层感知机已经具有极强的拟合能力。一种观点认为，这是因为仅由 ReLU 便可以构造出一个新的连续、有界且单调递增的非常数函数。令 $p < q (p \in \mathbb{R}, q \in \mathbb{R})$，则该新函数如下

$$\begin{aligned}
\varphi_{p,q} &= \varphi_r(x-p) - \varphi_r(x-q) \\
&= \max(0, x-p) - \max(0, x-q) \\
&= \begin{cases} 0, & x < p \\ x, & p \leqslant x < q \\ q-p, & x \geqslant q \end{cases}
\end{aligned}$$

2 优化方法

2.1 梯度下降法

回顾(2-2)式，现要求在 $\boldsymbol{X}$ 为训练集特征向量组和 $\boldsymbol{Y}$ 为对应的训练集标签向量组的情况下，找出使得 $\mathcal{L}$ 值尽可能小的权重矩阵参数 $\boldsymbol{W}^{(m)}$ 和偏置列向量参数 $\boldsymbol{b}^{(m)}$。由此，视 $\mathcal{L}$ 为一个关于权重参数组 $w_{1,1}^{(1)}, w_{1,2}^{(1)}, \cdots, w_{n^{(M)},n^{(M-1)}}^{(M)}$ 和偏置参数组 $b_1^{(1)}, b_2^{(1)}, \cdots, b_{n^{(M)}}^{(M)}$ 的多元函数

$$L = \mathcal{L}(w_{1,1}^{(1)}, w_{1,2}^{(1)}, \cdots, w_{n^{(M)},n^{(M-1)}}^{(M)}, b_1^{(1)}, b_2^{(1)}, \cdots, b_{n^{(M)}}^{(M)}) = \mathcal{L}(\boldsymbol{p})$$

对多元函数 $\mathcal{L}$ 而言，其极值可能为最值，而其所有偏导数均为 0 的点可能为极值点。于是存在一种方法为，计算出使得 $\mathcal{L}$ 的偏导数均为 0 的权重组和偏置组，即解算方程组

$$\frac{\partial}{\partial \boldsymbol{p}}\mathcal{L} = \boldsymbol{0}$$

但事实上，这种方法在计算上几乎不可行，故引入梯度下降法（Gradient Descent Method）。

根据函数在其梯度的反方向上的方向导数最小的原理可知，函数在该方向上的值下降最快。对 $\mathcal{L}$ 而言，只要顺着其梯度反方向前进，就能在局部上找到使得 $\mathcal{L}$ 尽可能小的权重组和

偏置组。在一次梯度下降的迭代过程中，设当前权重组和偏置组为点 $\boldsymbol{p}$，则从当前位置向梯度反方向前进一小段距离，并更新 $\boldsymbol{p}$ 到下一个位置，即

$$\boldsymbol{p} \leftarrow \boldsymbol{p} - \eta \frac{\partial}{\partial \boldsymbol{p}} \mathcal{L} \tag{5-1}$$

其中，$\eta(0 < \eta < 1)$ 称为学习率，它决定了前进的幅度。重复多轮执行(5-1)，$\mathcal{L}$ 在点 $\boldsymbol{p}$ 上的值在总体上将呈现为一个下降的趋势。针对(1-1)式，(5-1)式也可以具体拆分为

$$\begin{cases} \boldsymbol{W}^{(m)} \leftarrow \boldsymbol{W}^{(m)} - \eta \dfrac{\partial}{\partial \boldsymbol{W}^{(m)}} \mathcal{L} \\ \boldsymbol{b}^{(m)} \leftarrow \boldsymbol{b}^{(m)} - \eta \dfrac{\partial}{\partial \boldsymbol{b}^{(m)}} \mathcal{L} \end{cases} \tag{5-2}$$

梯度下降法并不能保证得到全局最优解。图 7所示为两个梯度下降算法和学习率均相同，但参数初始值不同的梯度下降过程。以圆形标记为起点的绿色路径找到了全局最优解，而以十字标记为起点的红色路径则陷入了局部最优。解决局部最优的方案较多，可以从自适应学习率、动量梯度下降、参数初始化等方面入手。

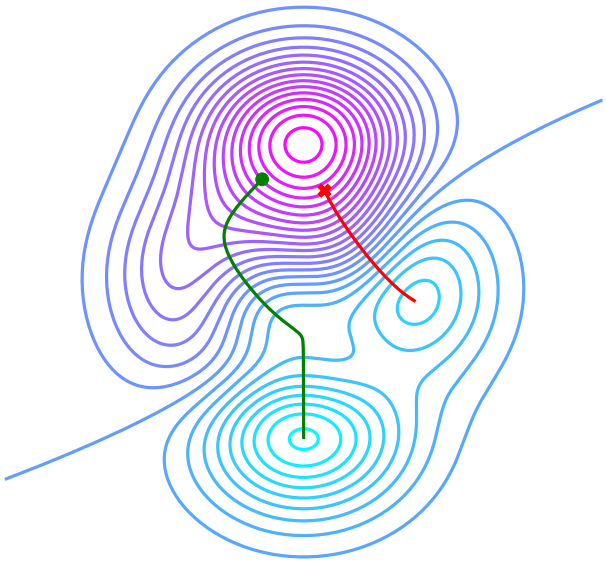


图 7　全局最优与局部最优示例

在上文中已交代，$\boldsymbol{X}$ 和 $\boldsymbol{Y}$ 分别是整个训练集的特征向量组和标签向量组，这种每次迭代过程都基于整个训练集来更新参数的梯度下降称为批量梯度下降（Batch Gradient Descent，BGD）。对计算机而言，整个训练集的样本数据是相对庞大的，因而批量梯度下降较为突出的缺点之一是计算开销大。不同于批量梯度下降，随机梯度下降（Stochastic Gradient Descent，SGD）的每次迭代过程只随机采取一个样本进行参数更新。换言之，此时的 $\boldsymbol{X}$ 和 $\boldsymbol{Y}$ 均为行向量，而非具有多行的矩阵。然而，单个样本不足以代表整个训练集，随机梯度下降具有一定盲目性，容易陷入局部最优。综上，可以考虑在批量梯度下降和随机梯度下降之间进行折中处理，引入小批量梯度下降（Mini-batch Gradient Descent，MBGD），它的每次迭代过程基于训练集中的小部分样本来更新参数。此时，矩阵 $\boldsymbol{X}$ 和 $\boldsymbol{Y}$ 分别为训练集小部分样本的特征向量组和标签向量组。

2.2 反向传播公式

如(5-2)式，梯度下降法的难点在于计算 $\mathcal{L}$ 对各层参数 $\boldsymbol{W}^{(m)}$ 和 $\boldsymbol{b}^{(m)}$ 的偏导数，故而引入误差反向传播（Backward Propagation of Errors）算法，也称为反向传播（Backpropagation，BP）算法[6][7]。反向传播算法基于链式法则实现，它利用被称为误差项的 $\partial\mathcal{L}/\partial\boldsymbol{X}^{(m)}$ 由后至前逐层计算出损失函数 $\mathcal{L}$ 对权重参数和偏置参数的偏导数，这一过程即为反向传播过程[8]。

现针对前文(1-1)式及(2-1)式，给出以下几个公式及其证明。

2.2.0.1 损失函数对神经元输出的偏导数

$$\frac{\partial\mathcal{L}}{\partial\boldsymbol{Y}^{(m)}}=\frac{\partial\mathcal{L}}{\partial\boldsymbol{X}^{(m+1)}}\boldsymbol{W}^{(m+1)} \tag{6-1}$$

其中，$1\leqslant m\leqslant M-1$。现给出如下证明。

首先，计算 $x_{i,v}^{(m+1)}$ 对 $y_{i,j}^{(m)}$ 的偏导数如下

$$\begin{aligned}\frac{\partial x_{i,v}^{(m+1)}}{\partial y_{i,j}^{(m)}}&=\frac{\partial}{\partial y_{i,j}^{(m)}}\left(\sum_{k=1}^{n^{(m)}}w_{v,k}^{(m+1)}y_{i,k}^{(m)}+b_v^{(m+1)}\right)\\&=w_{v,j}^{(m+1)}\end{aligned}$$

由上式，再依据链式法则，有

$$\begin{aligned}\left(\frac{\partial\mathcal{L}}{\partial\boldsymbol{Y}^{(m)}}\right)_{i,j}&=\frac{\partial\mathcal{L}}{\partial y_{i,j}^{(m)}}\\&=\sum_{u=1}^{s}\sum_{v=1}^{n^{(m+1)}}\frac{\partial x_{u,v}^{(m+1)}}{\partial y_{i,j}^{(m)}}\frac{\partial\mathcal{L}}{\partial x_{u,v}^{(m+1)}}\\&=\sum_{u\neq i}\sum_{v=1}^{n^{(m+1)}}\frac{\partial x_{u,v}^{(m+1)}}{\partial y_{i,j}^{(m)}}\frac{\partial\mathcal{L}}{\partial x_{u,v}^{(m+1)}}+\sum_{v=1}^{n^{(m+1)}}\frac{\partial x_{i,v}^{(m+1)}}{\partial y_{i,j}^{(m)}}\frac{\partial\mathcal{L}}{\partial x_{i,v}^{(m+1)}}\\&=0+\sum_{v=1}^{n^{(m+1)}}w_{v,j}^{(m+1)}\frac{\partial\mathcal{L}}{\partial x_{i,v}^{(m+1)}}\\&=\sum_{v=1}^{n^{(m+1)}}\left(\frac{\partial\mathcal{L}}{\partial\boldsymbol{X}^{(m+1)}}\right)_{i,v}(\boldsymbol{W}^{(m+1)})_{v,j}\end{aligned}$$

由此，(6-1)式成立。

2.2.0.2 损失函数对权重的偏导数

$$\frac{\partial\mathcal{L}}{\partial\boldsymbol{W}^{(m)}}=\left(\frac{\partial\mathcal{L}}{\partial\boldsymbol{X}^{(m)}}\right)^{\mathrm{T}}\boldsymbol{Y}^{(m-1)} \tag{6-2}$$

现给出如下证明。

首先，计算 $x_{v,i}^{(m)}$ 对 $w_{i,j}^{(m)}$ 的偏导数如下

$$\begin{aligned}\frac{\partial x_{v,i}^{(m)}}{\partial w_{i,j}^{(m)}}&=\frac{\partial}{\partial w_{i,j}^{(m)}}\left(\sum_{k}^{n^{(m-1)}}w_{i,k}^{(m)}y_{v,k}^{(m-1)}+b_i^{(m)}\right)\\&=y_{v,j}^{(m-1)}\end{aligned}$$

由上式，再依据链式法则，有

$$\begin{aligned}\left(\frac{\partial \mathcal{L}}{\partial \boldsymbol{W}^{(m)}}\right)_{i,j} &= \frac{\partial \mathcal{L}}{\partial w_{i,j}^{(m)}} \\ &= \sum_{u=1}^{n^{(m)}} \sum_{v=1}^{s} \frac{\partial \mathcal{L}}{\partial x_{v,u}^{(m)}} \frac{\partial x_{v,u}^{(m)}}{\partial w_{i,j}^{(m)}} \\ &= \sum_{u \neq i} \sum_{v=1}^{s} \frac{\partial \mathcal{L}}{\partial x_{v,u}^{(m)}} \frac{\partial x_{v,u}^{(m)}}{\partial w_{i,j}^{(m)}} + \sum_{v=1}^{s} \frac{\partial \mathcal{L}}{\partial x_{v,i}^{(m)}} \frac{\partial x_{v,i}^{(m)}}{\partial w_{i,j}^{(m)}} \\ &= 0 + \sum_{v=1}^{s} \frac{\partial \mathcal{L}}{\partial x_{v,i}^{(m)}} y_{v,j}^{(m-1)} \\ &= \sum_{v=1}^{s} \left(\left(\frac{\partial \mathcal{L}}{\partial \boldsymbol{X}^{(m)}} \right)^{\mathrm{T}} \right)_{i,v} (\boldsymbol{Y}^{(m-1)})_{v,j}\end{aligned}$$

由此，(6-2)式成立。

2.2.0.3 损失函数对偏置的偏导数

$$\left(\frac{\partial \mathcal{L}}{\partial \boldsymbol{b}^{(m)}}\right)_{i} = \sum_{v=1}^{s} \left(\frac{\partial \mathcal{L}}{\partial \boldsymbol{X}^{(m)}}\right)_{v,i} \tag{6-3}$$

换言之，$\partial \mathcal{L}/\partial \boldsymbol{b}^{(m)}$ 的第 i 个分量是 $\partial \mathcal{L}/\partial \boldsymbol{X}^{(m)}$ 的第 i 列之和。现给出如下证明。

首先，计算 $x_{v,i}^{(m)}$ 对 $b_i^{(m)}$ 的偏导数如下

$$\begin{aligned}\frac{\partial x_{v,i}^{(m)}}{\partial b_i^{(m)}} &= \frac{\partial}{\partial b_i^{(m)}} \left(\sum_{k=1}^{n^{(m-1)}} w_{i,k}^{(m)} y_{v,k}^{(m-1)} + b_i^{(m)} \right) \\ &= 1\end{aligned}$$

由上式，再依据链式法则，有

$$\begin{aligned}\left(\frac{\partial \mathcal{L}}{\partial \boldsymbol{b}^{(m)}}\right)_{i} &= \frac{\partial \mathcal{L}}{\partial b_i^{(m)}} \\ &= \sum_{u=1}^{n^{(m)}} \sum_{v=1}^{s} \frac{\partial \mathcal{L}}{\partial x_{v,u}^{(m)}} \frac{\partial x_{v,u}^{(m)}}{\partial b_i^{(m)}} \\ &= \sum_{u \neq i} \sum_{v=1}^{s} \frac{\partial \mathcal{L}}{\partial x_{v,u}^{(m)}} \frac{\partial x_{v,u}^{(m)}}{\partial b_i^{(m)}} + \sum_{v=1}^{s} \frac{\partial \mathcal{L}}{\partial x_{v,i}^{(m)}} \frac{\partial x_{v,i}^{(m)}}{\partial b_i^{(m)}} \\ &= \sum_{v=1}^{s} \frac{\partial \mathcal{L}}{\partial x_{v,i}^{(m)}} \\ &= \sum_{v=1}^{s} \left(\frac{\partial \mathcal{L}}{\partial \boldsymbol{X}^{(m)}}\right)_{v,i}\end{aligned}$$

2.2.0.4 隐藏层误差项

$$\frac{\partial \mathcal{L}}{\partial \boldsymbol{X}^{(m)}} = \varphi^{(m)\prime}(\boldsymbol{X}^{(m)}) \odot \left(\frac{\partial \mathcal{L}}{\partial \boldsymbol{X}^{(m+1)}} \cdot \boldsymbol{W}^{(m+1)} \right) \tag{6-4}$$

其中，$1 \leqslant m \leqslant M-1$，$\odot$ 是哈达玛积（Hadamard Product），$\varphi^{(m)\prime}$ 是 $\varphi^{(m)}$ 的导数且约定 $(\varphi^{(m)\prime}(\boldsymbol{X}^{(m)}))_{i,j}=\varphi^{(m)\prime}(x_{i,j}^{(m)})$。同时，需要注意，第 m 层任意一个神经元的输出 $y_{u,v}^{(m)}$ 仅与 $x_{u,v}^{(m)}$ 相关，即 $y_{u,v}^{(m)}=\varphi^{(m)}(x_{u,v}^{(m)})$ 仅有唯一变量 $x_{u,v}^{(m)}$。现给出如下证明。

首先，由链式法则，有

$$\begin{aligned}\left(\frac{\partial\mathcal{L}}{\partial\boldsymbol{X}^{(m)}}\right)_{i,j}&=\frac{\partial\mathcal{L}}{\partial x_{i,j}^{(m)}}\\&=\sum_{u=1}^{s}\sum_{v=1}^{n^{(m)}}\frac{\partial y_{u,v}^{(m)}}{\partial x_{i,j}^{(m)}}\frac{\partial\mathcal{L}}{\partial y_{u,v}^{(m)}}\\&=\sum_{u\neq i}\sum_{v=1}^{n^{(m)}}\frac{\partial y_{u,v}^{(m)}}{\partial x_{i,j}^{(m)}}\frac{\partial\mathcal{L}}{\partial y_{u,v}^{(m)}}+\sum_{v=1}^{n^{(m)}}\frac{\partial y_{i,v}^{(m)}}{\partial x_{i,j}^{(m)}}\frac{\partial\mathcal{L}}{\partial y_{i,v}^{(m)}}\\&=\sum_{v=1}^{n^{(m)}}\frac{\partial y_{i,v}^{(m)}}{\partial x_{i,j}^{(m)}}\frac{\partial\mathcal{L}}{\partial y_{i,v}^{(m)}}\end{aligned}\tag{6*}$$

已知当 $v\neq j$ 时，$y_{i,v}^{(m)}$ 与 $x_{i,j}^{(m)}$ 不相关，于是由(6*)，有

$$\begin{aligned}\left(\frac{\partial\mathcal{L}}{\partial\boldsymbol{X}^{(m)}}\right)_{i,j}&=\sum_{v\neq j}\frac{\partial y_{i,v}^{(m)}}{\partial x_{i,j}^{(m)}}\frac{\partial\mathcal{L}}{\partial y_{i,v}^{(m)}}+\frac{\partial y_{i,j}^{(m)}}{\partial x_{i,j}^{(m)}}\frac{\partial\mathcal{L}}{\partial y_{i,j}^{(m)}}\\&=0+\varphi^{(m)\prime}(x_{i,j}^{(m)})\frac{\partial\mathcal{L}}{\partial y_{i,j}^{(m)}}\\&=\varphi^{(m)\prime}(\boldsymbol{X}^{(m)})_{i,j}\left(\frac{\partial\mathcal{L}}{\partial\boldsymbol{Y}^{(m)}}\right)_{i,j}\end{aligned}$$

再结合(6-1)式，可知(6-4)式成立。

2.2.0.5 基于交叉熵损失的 Softmax 输出层误差项

$$\frac{\partial\mathcal{L}}{\partial\boldsymbol{X}^{(M)}}=\frac{1}{s}(\widehat{\boldsymbol{Y}}-\boldsymbol{Y})\tag{6-5}$$

(6-5)式适用于损失函数为交叉熵、输出层激活函数为 Softmax 的用于解决分类问题的多层感知机。现给出如下证明。

首先，由(6*)式，有

$$\begin{aligned}\left(\frac{\partial\mathcal{L}}{\partial\boldsymbol{X}^{(M)}}\right)_{i,j}&=\sum_{v=1}^{n^{(M)}}\frac{\partial y_{i,v}^{(M)}}{\partial x_{i,j}^{(M)}}\frac{\partial\mathcal{L}}{\partial y_{i,v}^{(M)}}\\&=\sum_{v\neq j}\frac{\partial y_{i,v}^{(M)}}{\partial x_{i,j}^{(M)}}\frac{\partial\mathcal{L}}{\partial y_{i,v}^{(M)}}+\frac{\partial y_{i,j}^{(M)}}{\partial x_{i,j}^{(M)}}\frac{\partial\mathcal{L}}{\partial y_{i,j}^{(M)}}\\&=\sum_{v\neq j}\frac{\partial\widehat{y}_{i,v}}{\partial x_{i,j}^{(M)}}\frac{\partial\mathcal{L}}{\partial\widehat{y}_{i,v}}+\frac{\partial\widehat{y}_{i,j}}{\partial x_{i,j}^{(M)}}\frac{\partial\mathcal{L}}{\partial\widehat{y}_{i,j}}\end{aligned}$$

同时，由于 $\boldsymbol{Y}$ 各行均为 One-hot 向量，于是有

$$\sum_{v=1}^{n^{(M)}}y_{i,v}=1$$

再由(2-3-2)式、(3-5-2)式和(3-5-3)式，有

$$\begin{aligned}\left(\frac{\partial \mathcal{L}}{\partial \boldsymbol{X}^{(M)}}\right)_{i,j} &= \sum_{v\neq j}(-\widehat{y}_{i,v}\widehat{y}_{i,j})\left(-\frac{1}{s}\frac{y_{i,v}}{\widehat{y}_{i,v}}\right)+\widehat{y}_{i,j}(1-\widehat{y}_{i,j})\left(-\frac{1}{s}\frac{y_{i,j}}{\widehat{y}_{i,j}}\right)\\ &= \sum_{v\neq j}\frac{1}{s}\widehat{y}_{i,j}y_{i,v}+\frac{1}{s}\widehat{y}_{i,j}y_{i,j}-\frac{1}{s}y_{i,j}\\ &= \sum_{v=1}^{n^{(M)}}\frac{1}{s}\widehat{y}_{i,j}y_{i,v}-\frac{1}{s}y_{i,j}\\ &= \frac{1}{s}\widehat{y}_{i,j}\left(\sum_{v=1}^{n^{(M)}}y_{i,v}\right)-\frac{1}{s}y_{i,j}\\ &= \frac{1}{s}\widehat{y}_{i,j}-\frac{1}{s}y_{i,j}\\ &= \frac{1}{s}((\widehat{\boldsymbol{Y}})_{i,j}-(\boldsymbol{Y})_{i,j})\end{aligned}$$

由此，(6-5)式成立。

2.2.0.6 基于均方差损失的线性激活输出层误差项

$$\frac{\partial \mathcal{L}}{\partial \boldsymbol{X}^{(M)}}=\frac{2}{n^{(M)}s}(\widehat{\boldsymbol{Y}}-\boldsymbol{Y}) \tag{6-6}$$

(6-6)式适用于损失函数为均方差、输出层激活函数为线性函数 $\varphi^{(M)}(x)=x$ 的用于解决分类问题的多层感知机。现给出如下证明。

由于 $\widehat{\boldsymbol{Y}}=\varphi^{(M)}(\boldsymbol{X}^{(M)})=\boldsymbol{X}^{(M)}$，于是依据(2-4-2)式，有

$$\begin{aligned}\left(\frac{\partial \mathcal{L}}{\partial \boldsymbol{X}^{(M)}}\right)_{i,j} &= \left(\frac{\partial \mathcal{L}}{\partial \widehat{\boldsymbol{Y}}}\right)_{i,j}\\ &= \frac{\partial \mathcal{L}}{\partial \widehat{y}_{i,j}}\\ &= \frac{2}{n^{(M)}s}(\widehat{y}_{i,j}-y_{i,j})\\ &= \frac{2}{n^{(M)}s}((\widehat{\boldsymbol{Y}})_{i,j}-(\boldsymbol{Y})_{i,j})\end{aligned}$$

由此，(6-6)成立。

3 训练过程

一般地，机器学习需要经过数据采集、数据分析、数据处理、模型选择、模型训练和模型评估等流程，现简化为数据准备、模型准备和模型训练。

数据准备的主要工作是特征工程（Feature Engineering）。简而言之，特征工程是将原始数据转化为能够更好地表征实体并且适用于特定机器学习模型的特征的过程。例如，在对 28×28 像素灰度图上的手写数字的识别任务中，每张图片由 28×28 矩阵表征，该矩阵的元素为像素灰度值，是 $[0,255]$ 上的整数。由于多层感知机的输入是 1 维的，故需要将 28×28 矩阵转化为含有 784 个分量的向量。此外，为避免出现收敛困难、数据溢出等问题，还需要对数据进行归一化处理。如前例，所有像素灰度值应被映射到 $[0,1]$ 上。

模型准备的主要工作是超参数（Hyperparameter）选择。一般而言，超参数是用于定义模型结构和优化方法的一类参数，如多层感知机层数、各层神经元个数、各层激活函数以及梯度下降学习率等。超参数很难通过优化方法自动调整，通常需要人为设置。综上，超参数选择是一项经验性极强的工作。

一般地，在基于小批量梯度下降的训练过程中，对模型进行一次参数调整的一个过程称为一次迭代（Iteration），用于一次迭代的一小批样本数据称为一个批量（Batch），整个训练集全部数据均已被用于参数调整的一个完整过程称为一个时期（Epoch）。合理设置批量大小（Batch Size）、迭代次数和时期次数也是训练的关键环节。现给出一种训练方案的主要过程，见如下算法。

算法: 基于误差反向传播的小批量梯度下降训练主过程

输入: $\mathcal{D}=\{\boldsymbol{x}_i,\boldsymbol{y}_i\}_{i=1}^{|\mathcal{D}|}$,
$n_{ep}(n_{ep}\geqslant 1)$,
$s(s\leqslant|\mathcal{D}|)$

随机或按特定方法初始化参数组 $\mathcal{P}=\{\boldsymbol{W}^{(m)},\boldsymbol{b}^{(m)}\}_{m=1}^{M}$;
for $e\leftarrow 1$ **to** n_{ep} **do** // n_{ep}: epoch number
  $n_{it}\leftarrow\lceil|\mathcal{D}|/s\rceil$; // n_{it}: iteration number; s: batch size
  对 $\mathcal{D}$ 中的所有有序对 $(\boldsymbol{x}_i,\boldsymbol{y}_i)$ 进行随机排序;
  按 s 将 $\mathcal{D}$ 顺序划分为 n_{it} 批数据 $\{\boldsymbol{X}_i,\boldsymbol{Y}_i\}_{i=1}^{n_{it}}$;
  for $i\leftarrow 1$ **to** n_{it} **do**
    $\boldsymbol{Y}^{(0)}\leftarrow\boldsymbol{X}_i$;
    for $m\leftarrow 1$ **to** M **do**
      依据(1-1)式计算 $\boldsymbol{X}^{(m)}$ 和 $\boldsymbol{Y}^{(m)}$;
    end
    $\boldsymbol{Y}\leftarrow\boldsymbol{Y}_i$;
    依据(6-5)式或(6-6)式计算输出层误差项 $\partial\mathcal{L}/\partial\boldsymbol{X}^{(M)}$;
    $\boldsymbol{W}\leftarrow\boldsymbol{W}^{(M)}$;
    依据(6-2)式计算 $\partial\mathcal{L}/\partial\boldsymbol{W}^{(M)}$;
    依据(6-3)式计算 $\partial\mathcal{L}/\partial\boldsymbol{b}^{(M)}$;
    依据(5-2)式更新 $\boldsymbol{W}^{(M)}$ 和 $\boldsymbol{b}^{(M)}$;
    for $m\leftarrow(M-1)$ **to** 1 **do**
      依据(6-4)式和更新前的第 $m+1$ 层权重矩阵 $\boldsymbol{W}$ 计算误差项 $\partial\mathcal{L}/\partial\boldsymbol{X}^{(m)}$;
      $\boldsymbol{W}\leftarrow\boldsymbol{W}^{(m)}$;
      依据(6-2)式计算 $\partial\mathcal{L}/\partial\boldsymbol{W}^{(m)}$;
      依据(6-3)式计算 $\partial\mathcal{L}/\partial\boldsymbol{b}^{(m)}$;
      依据(5-2)式更新 $\boldsymbol{W}^{(m)}$ 和 $\boldsymbol{b}^{(m)}$;
    end
  end
end

输出: $\mathcal{P}=\{\boldsymbol{W}^{(m)},\boldsymbol{b}^{(m)}\}_{m=1}^{M}$

参考文献

[1] Warren S McCulloch,Walter Pitts.A Logical Calculus of the Ideas Immanent in Nervous Activity[J].Bulletin of Mathematical Biophysics,1943,5(1):115‑133.

[2] F Rosenblatt.The Perceptron: A Probabilistic Model for Information Storage and Organization in the Brain[J].Psychological Review,1958,65(6):386‑408.

[3] G Cybenko.Approximation by Superpositions of a Sigmoidal Function[J].Mathematics of Control, Signals, and Systems,1989,2(1):303-314.

[4] Kurt Hornik.Multilayer Feedforward Networks are Universal Approximators[J].Neural Networks, 1989,2(5):359-366.

[5] Kurt Hornik.Approximation Capabilities of Multilayer Feedforward Networks[J].Neural Networks,1991,4(2):251-257.

[6] Paul J Werbos.Beyond Regression: New Tools for Prediction and Analysis in the Behavioral Sciences[D].Cambridge, MA:Harvard University,1974.

[7] David E Rumelhart,Geoffrey E Hinton,Ronald J Williams.Learning Representations by Back-propagating Errors[J].Nature,1986,323(9):533-536.

[8] 邱锡鹏. 神经网络与深度学习 [M]. 北京: 机械工业出版社,2020:92-95.

代码

神经网络

```python
# -*- coding: UTF-8 -*-
"""
File Name: neural_network.py
"""
# Python 3.8.16

import numpy as np  # 1.18.5


class FullyConnectedNN(object):
    """Fully connected neural network"""

    @property
    def num_layers(self):
        """Number of layers"""
        return self.__num_lyr

    def __init__(self, layer_dims, activ_funcs):
        # Number of layers
        self.__num_lyr = len(layer_dims)
        # Input matrices of hidden and output layers
        self.lyr_in = dict()
        # Output matrices of all layers
        self.lyr_out = {0: np.array([])}
        # Weight matrices of hidden and output layers
        self.weight = dict()
        # Bias column vectors of hidden and output layers
        self.bias = dict()
        # Activation functions of hidden and output layers
        self.__ac_func = dict()
        # Initialization
        for i in range(1, self.__num_lyr, 1):
            self.lyr_in[i] = np.array([])
            self.lyr_out[i] = np.array([])
            self.weight[i] = np.random.uniform(
                -1, 1, (layer_dims[i], layer_dims[i - 1])
            )
            self.bias[i] = np.random.uniform(-1, 1, (layer_dims[i], 1))
            self.__ac_func[i] = activ_funcs[i - 1]
```

```
        # Error matrix of the backward propagation process
        self.bp_err = np.array([])
        # Weight matrix stored for the backward propagation process before updating
        self.bp_wgt = np.array([])

    def grad_loss_out(self):
        """The partial derivatives of the loss function
        with respect to elements of an output matrix"""
        return self.bp_err @ self.bp_wgt

    def grad_loss_weight(self, layer_id):
        """The partial derivatives of the loss function
        with respect to elements of a weight matrix"""
        return self.bp_err.T @ self.lyr_out[layer_id - 1]

    def grad_loss_bias(self):
        """The partial derivatives of the loss function
        with respect to elements of a bias column vector"""
        err0_sum = self.bp_err.sum(0)
        return err0_sum.reshape((err0_sum.shape[0], 1))

    def forward(self, features):
        """Forward propagation"""
        self.lyr_out[0] = features
        for i in range(1, self.__num_lyr, 1):
            self.lyr_in[i] = (
                    self.weight[i] @ self.lyr_out[i - 1].T + self.bias[i]
            ).T
            self.lyr_out[i] = self.__ac_func[i](self.lyr_in[i])


class BackpropagationNN(FullyConnectedNN):
    """Back propagation (backward propagation of errors) neural network"""

    def set_learning_rate(self, rate):
        """Initialize the learning rate."""
        self.__lr = rate if 0 < rate < 1 else 0.01

    def cal_out_err(self):
        """Calculate the error matrix of the output layer."""
        pass

    def __init__(self, layer_dims, activ_funcs, hid_grad_activ_funcs):
```

```
        super().__init__(layer_dims, activ_funcs)
        # Label matrix
        self.label = None
        # Learning rate
        self.__lr = 0.01
        # Derivatives of all hidden layer activation functions
        self.__hid_grd_acf = dict()
        # Output layer id, a key of lyr_out
        self.max_lid = self.num_layers - 1
        for i in range(1, self.max_lid, 1):
            self.__hid_grd_acf[i] = hid_grad_activ_funcs[i - 1]

    def __adjust_weight(self, layer_id):
        self.weight[layer_id] -= self.__lr * self.grad_loss_weight(layer_id)

    def __adjust_bias(self, layer_id):
        self.bias[layer_id] -= self.__lr * self.grad_loss_bias()

    def __cal_hid_err(self, layer_id):
        """Calculate an error matrix of a hidden layer."""
        self.bp_err = self.__hid_grd_acf[layer_id](
            self.lyr_in[layer_id]) * self.grad_loss_out()
        self.bp_wgt = self.weight[layer_id]

    def __backward(self):
        self.bp_err = self.cal_out_err()
        self.bp_wgt = self.weight[self.max_lid]
        self.__adjust_weight(self.max_lid)
        self.__adjust_bias(self.max_lid)
        for i in range(self.max_lid - 1, 0, -1):
            self.__cal_hid_err(i)
            self.__adjust_weight(i)
            self.__adjust_bias(i)

    def train_one_batch(self, features, labels):
        """Train one batch."""
        self.forward(features)
        self.label = labels
        self.__backward()

    def train_one_epoch(self, features, labels, batch_size):
        """Train one epoch."""
        num_sp = labels.shape[0]
```

```
126         iteration = (num_sp + batch_size - 1) // batch_size
127         index = np.arange(0, num_sp, 1)
128         # Disrupt the entire training set order.
129         # And this is actually not mandatory to execute.
130         np.random.shuffle(index)
131         ite_idx = 0
132         for i in range(iteration):
133             sub_idx = index[ite_idx:ite_idx + batch_size:1]
134             sub_features = features[sub_idx]
135             sub_labels = labels[sub_idx]
136             self.train_one_batch(sub_features, sub_labels)
137             ite_idx += batch_size
138
139
140 class SoftmaxClassifier(BackpropagationNN):
141     """Classifier based on cross-entropy & softmax"""
142
143     @staticmethod
144     def softmax(np2d_arr):
145         """Softmax function"""
146         row_max = np.amax(np2d_arr, axis=1)
147         exp_matrix = np.exp(np2d_arr - row_max.reshape((row_max.shape[0], 1)))
148         row_sum = np.sum(exp_matrix, axis=1)
149         return exp_matrix / row_sum.reshape((row_sum.shape[0], 1))
150
151     @staticmethod
152     def ce_loss(outputs, labels):
153         """Cross-entropy loss function"""
154         return -np.sum(labels * np.log(
155             np.where(outputs == 0, 0.001, outputs))) / labels.shape[0]
156
157     def cal_out_err(self):
158         """Calculate the error matrix of the output layer."""
159         return (self.lyr_out[self.max_lid] - self.label) / self.label.shape[0]
160
161     def __init__(self, layer_dims, hid_activ_funcs, hid_grad_activ_funcs):
162         """
163         Parameters
164         --
165         layer_dims: tuple or list
166             Numbers of neurons in each layer
167         hid_activ_funcs: tuple or list
168             Activation functions of all hidden layers
```

```
        hid_grad_activ_funcs: tuple or list
            Derivatives of all hidden layer activation functions
        """
        activ_funcs = [func for func in hid_activ_funcs]
        activ_funcs.append(self.softmax)
        super().__init__(layer_dims, activ_funcs, hid_grad_activ_funcs)
        self.loss = []
        self.accuracy = []

    def train_mul_epoch(self, features, labels, batch_size, epoch_size,
                        test_features=None, test_labels=None):
        """Train multiple epochs."""
        test_x = features if test_features is None else test_features
        test_y = labels if test_labels is None else test_labels
        for i in range(epoch_size):
            self.train_one_epoch(features, labels, batch_size)
            #
            # ** **
            _features = test_x
            _labels = test_y
            self.forward(_features)
            loss = self.ce_loss(self.lyr_out[self.max_lid], _labels)
            outputs = np.argmax(self.lyr_out[self.max_lid], axis=1)
            _labels = np.argmax(_labels, axis=1)
            accu = np.sum(outputs == _labels) / _labels.shape[0]
            print('{: <6}'.format(i), end='')
            print('{: <15.6f}'.format(loss), end='')
            print('{:.6f}'.format(accu))
            self.loss.append(loss)
            self.accuracy.append(accu)
            # ** **
            #

    def predict(self, features):
        """
        Returns
        --
        k_id: numpy.ndarray
            The ordinal numbers that distinguish their corresponding sample labels
        """
        self.forward(features)
        return np.argmax(self.lyr_out[self.max_lid], axis=1)

```

```
212     def test_accuracy(self, features, labels):
213         """
214         Returns
215         --
216         num: integer
217             The total number of samples whose prediction results matched their labels
218         """
219         self.forward(features)
220         outputs = np.argmax(self.lyr_out[self.max_lid], axis=1)
221         labels = np.argmax(labels, axis=1)
222         return np.sum(outputs == labels)
223 
224 
225 class MSERegressor(BackpropagationNN):
226     """Regressor based on mean squared error"""
227 
228     @staticmethod
229     def ms_loss(outputs, labels):
230         """Mean squared loss function"""
231         delta = outputs - labels
232         return np.sum(delta * delta) / (outputs.shape[0] * outputs.shape[1])
233 
234     def cal_out_err(self):
235         """Calculate the error matrix of the output layer."""
236         outs = self.lyr_out[self.max_lid]
237         return 2 * (outs - self.label) / (outs.shape[0] * outs.shape[1])
238 
239     def __init__(self, layer_dims, hid_activ_funcs, hid_grad_activ_funcs):
240         """
241         Parameters
242         --
243         layer_dims: tuple or list
244             Numbers of neurons in each layer
245         hid_activ_funcs: tuple or list
246             Activation functions of all hidden layers
247         hid_grad_activ_funcs: tuple or list
248             Derivatives of all hidden layer activation functions
249         """
250         activ_funcs = [func for func in hid_activ_funcs]
251         activ_funcs.append(lambda arr: arr)
252         super().__init__(layer_dims, activ_funcs, hid_grad_activ_funcs)
253         self.loss = []
254 
```

```
    def train_mul_epoch(self, features, labels, batch_size, epoch_size,
                        test_features=None, test_labels=None):
        """Train multiple epochs."""
        test_x = features if test_features is None else test_features
        test_y = labels if test_labels is None else test_labels
        for i in range(epoch_size):
            self.train_one_epoch(features, labels, batch_size)
            #
            # ** **
            _features = test_x
            _labels = test_y
            self.forward(_features)
            loss = self.ms_loss(self.lyr_out[self.max_lid], _labels)
            print('{: <6}'.format(i), end='')
            print('{: <15.6f}'.format(loss))
            self.loss.append(loss)
            # ** **
            #

    def predict(self, features):
        """
        Returns
        --
        k_id: numpy.ndarray
        """
        self.forward(features)
        return self.lyr_out[self.max_lid]
```

激活函数

```
# -*- coding: UTF-8 -*-
"""
File Name: activation_function.py
"""
# Python 3.8.16

import numpy as np  # 1.18.5


def sigmoid(np2d_arr):
    """Sigmoid function"""
    return 1 / (1 + np.exp(-np2d_arr))
```

```
13
14
15  def d_sigmoid(np2d_arr):
16      """The partial derivative of the sigmoid function"""
17      sgm = sigmoid(np2d_arr)
18      return sgm * (1 - sgm)
19
20
21  def relu(np2d_arr):
22      """Rectified linear unit function"""
23      return np.maximum(np2d_arr, 0)
24
25
26  def d_relu(np2d_arr):
27      """The partial derivative of the rectified linear unit function"""
28      return np.where(np2d_arr > 0, 1, 0)
29
30
31  def leaky_relu(np2d_arr):
32      """Leaky rectified linear unit function"""
33      return np.minimum(np2d_arr, 0) * 0.01 + np.maximum(np2d_arr, 0)
34
35
36  def d_leaky_relu(np2d_arr):
37      """The partial derivative of the leaky rectified linear unit function"""
38      mtx = np.where(np2d_arr < 0, np2d_arr, 1)
39      return np.where(mtx > 0, mtx, 0.01)
40
41
42  def tan_h(np2d_arr):
43      """Hyperbolic tangent function"""
44      exp = np.exp(-2 * np2d_arr)
45      return (1 - exp) / (1 + exp)
46
47
48  def d_tan_h(np2d_arr):
49      """The partial derivative of the hyperbolic tangent function"""
50      th = tan_h(np2d_arr)
51      return 1 - th * th
```

MNIST 测试

```python
# -*- coding: UTF-8 -*-
"""
File Name: test_mnist.py
"""
# Python 3.8.16

import numpy as np  # 1.18.5
from tensorflow.keras.datasets import mnist  # tensorflow-gpu 2.3.0
import activation_function as af
import neural_network as nn


def get_one_hot(one_dim_digit):
    """Represent a digit with a One-hot."""
    one_hot = np.zeros(shape=(10,))
    one_hot[one_dim_digit[0]] = 1
    return one_hot


# DATA

(train_features, train_labels), (test_features, test_labels) = mnist.load_data()
feature_dim = train_features.shape[1] * train_features.shape[2]
# train data
train_num = train_features.shape[0]
train_features = train_features.reshape((train_num, feature_dim)) / 255
train_labels = np.apply_along_axis(
    get_one_hot, axis=1, arr=train_labels.reshape((train_num, 1))
)
# test data
test_num = test_features.shape[0]
test_features = test_features.reshape((test_num, feature_dim)) / 255
test_labels = np.apply_along_axis(
    get_one_hot, axis=1, arr=test_labels.reshape((test_num, 1))
)

# MODEL

# There are 81 neurons in the hidden layer.
classifier = nn.SoftmaxClassifier((feature_dim, 81, 10),
                                  (af.sigmoid,),
                                  (af.d_sigmoid,))

```

```
# TRAINING

classifier.set_learning_rate(0.55)
# batch_size = 9, epoch_size = 11
classifier.train_mul_epoch(train_features, train_labels, 9, 11,
                           test_features, test_labels)
classifier.set_learning_rate(0.1)
# batch_size = 27, epoch_size = 9
classifier.train_mul_epoch(train_features, train_labels, 27, 9,
                           test_features, test_labels)
classifier.set_learning_rate(0.001)
# batch_size = 31, epoch_size = 13
classifier.train_mul_epoch(train_features, train_labels, 31, 13,
                           test_features, test_labels)

# TEST

print('\naccuracy = ', end='')
print(classifier.test_accuracy(test_features, test_labels) / test_num)
```

整理不易，转载请注明作者及出处。

作者：Deng Z
联系：3030299946
出处：binisbull@GitHub