Machine Learning

题目：**中山大学**

学习小组 05 机器学习作业

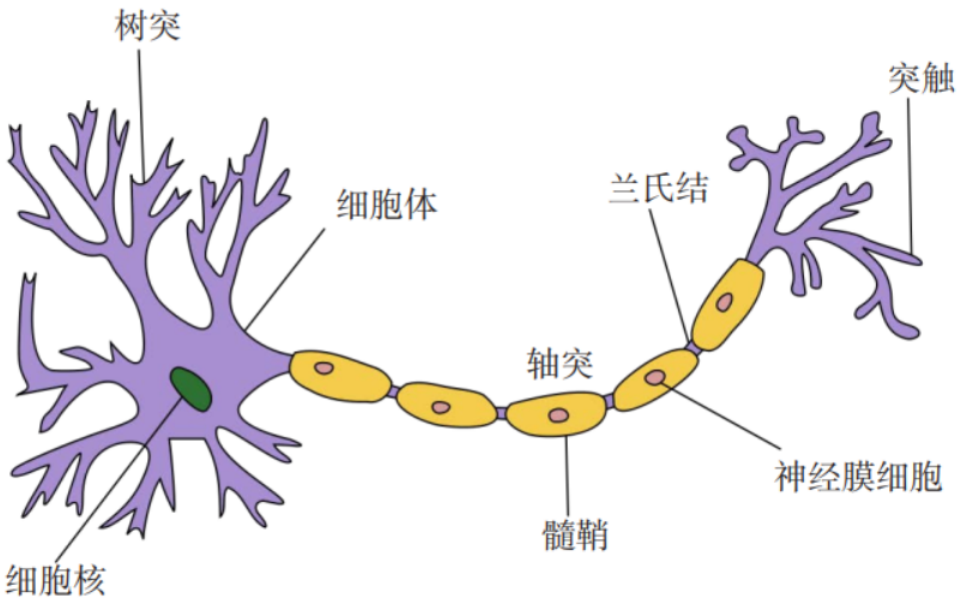
姓　　名	方桂安，刘玥，周敏
学　　号	20354027,20354229,20354187
院　　系	智能工程学院
专　　业	智能科学与技术
指导教师	彭卫文 (副教授)

2021 年 11 月 19 日

一、描述所使用的神经网络模型

1.1 人脑神经网络

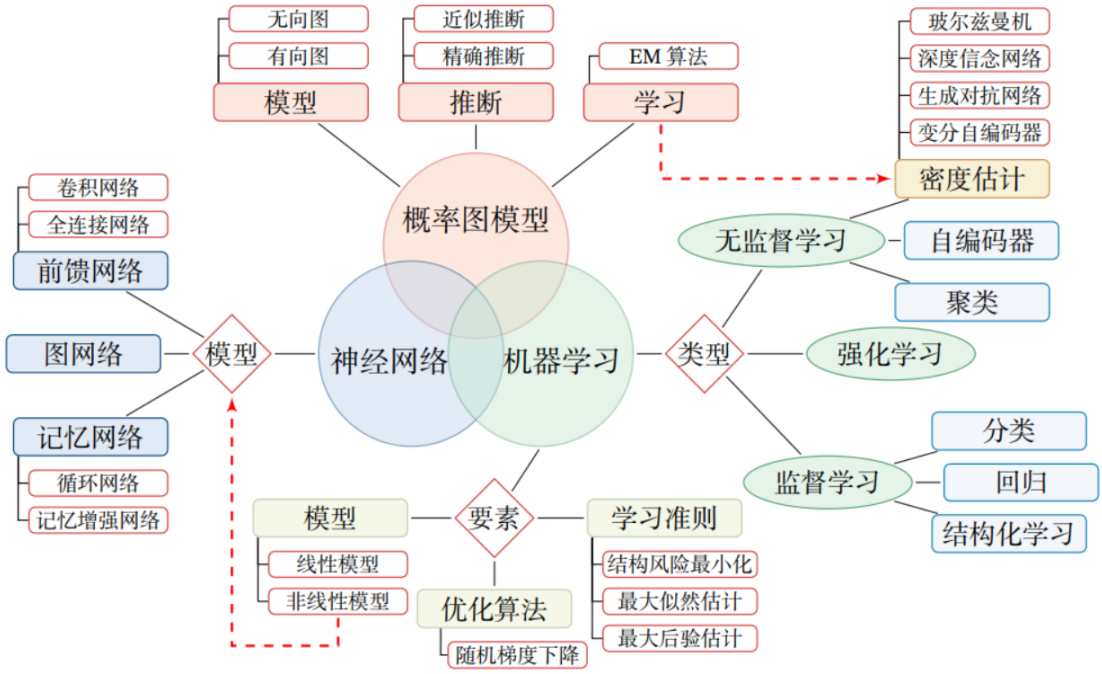
人类大脑是人体最复杂的器官，由神经元、神经胶质细胞、神经干细胞和血管组成．其中，神经元（Neuron），也叫神经细胞（NerveCell），是携带和传输信息的细胞，是人脑神经系统中最基本的单元．人脑神经系统是一个非常复杂的组织，包含近860亿个神经元，每个神经元有上千个突触和其他神经元相连接．这些神经元和它们之间的连接形成巨大的复杂网络，其中神经连接的总长度可达数千公里．我们人造的复杂网络，比如全球的计算机网络，和大脑神经网络相比要“简单”得多．



1.2 人工神经网络

人工神经网络是为模拟人脑神经网络而设计的一种计算模型， 它从结构、实现机理和功能上模拟人脑神经网络． 人工神经网络与生物神经元类似， 由多个节点（ 人工神经元） 互相连接而成， 可以用来对数据之间的复杂关系进行建模． 不同节点之间的连接被赋予了不同的权重， 每个权重代表了一个节点对另一个节点的影响大小． 每个节点代表一种特定函数， 来自其他节点的信息经过其相应的权重综合计算，输入到一个激活函数中并得到一个新的活性值（ 兴奋或抑制）．从系统观点看， 人工神经元网络是由大量神经元通过极其丰富和完善的连接而构成的自适应非线性动态系统．
虽然我们可以比较容易地构造一个人工神经网络， 但是如何让人工神经网络具有学习能力并不是一件容易的事情． 早期的神经网络模型并不具备学习能力． 首个可学习的人工神经网络是赫布网络， 采用一种基于赫布规则的无监督学习方法． 感知器是最早的具有机器学习思想的神经网络， 但其学习方法无法扩展到多层的神经网络上． 直到 1980 年左右， 反向传播算法才有效地解决了多层神经网络的学习问题， 并成为最为流行的神经网络学习算法．

人工神经网络诞生之初并不是用来解决机器学习问题． 由于人工神经网络可以用作一个通用的函数逼近器（ 一个两层的神经网络可以逼近任意的函数），因此我们可以将人工神经网络看作一个可学习的函数， 并将其应用到机器学习中． 理论上， 只要有足够的训练数据和神经元数量， 人工神经网络就可以学到很多复杂的函数． 我们可以把一个人工神经网络塑造复杂函数的能力称为网络容量（ Network Capacity）， 这与可以被储存在网络中的信息的复杂度以及数量相关．



1.3 前馈神经网络

在本次作业中， 我们主要采用误差反向传播来进行学习的神经网络， 即作为一种机器学习模型的神经网络．从机器学习的角度来看， 神经网络一般可以看作一个非线性模型， 其基本组成单元为具有非线性激活函数的神经元， 通过大量神经元之间的连接， 使得神经网络成为一种高度非线性的模型． 神经元之间的连接权重就是需要学习的参数， 可以在机器学习的框架下通过梯度下降方法来进行学习．

1.3.1 神经元

1943 年， 心理学家 McCulloch 和数学家 Pitts 根据生物神经元的结构， 提出了一种非常简单的神经元模型， MP神经元． 现代神经网络中的神经元和 MP 神经元的结构并无太多变化． 不同的是， MP 神经元中的激活函数f 为0或1的阶跃函数， 而现代神经元中的激活函数通常要求是连续可导的函数．

假设一个神经元接收 D 个输入$x_1, x_2, \cdots, x_D$，令向量$\boldsymbol{x} = [x_1; x_2; \cdots; x_D]$来表示这组输入，并用净输入 (Net Input) $z \in \mathbb{R}$表示一个神经元所获得的输入信号 x 的加权和，

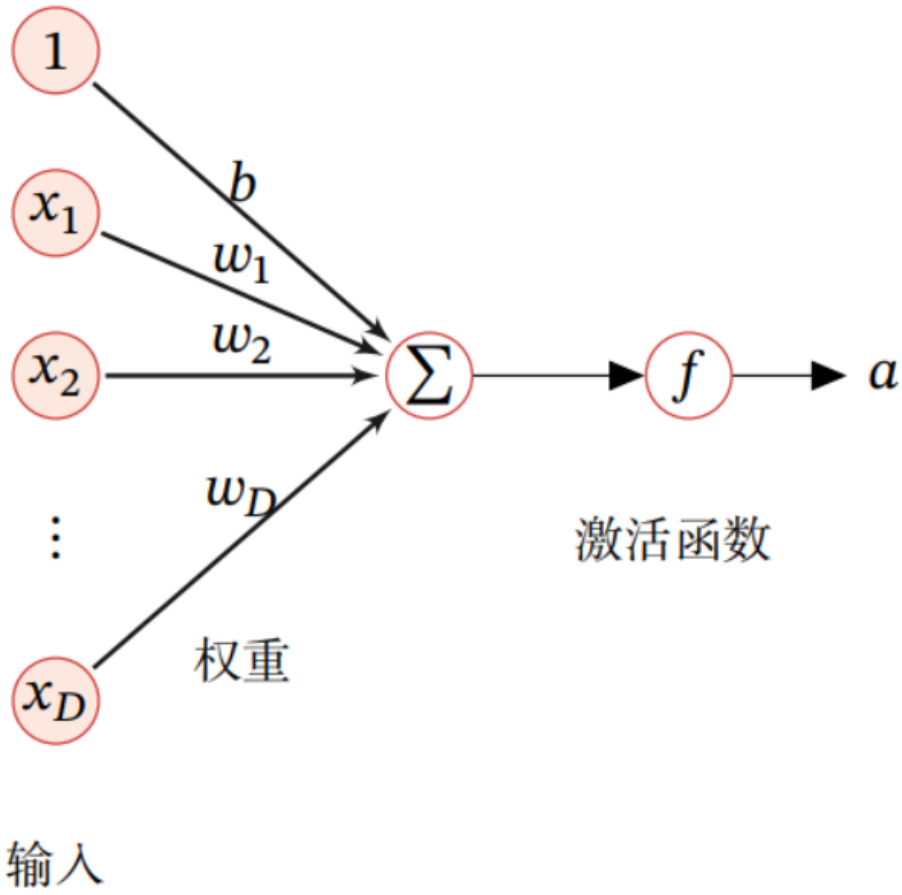
$$\begin{aligned} z &= \sum_{d=1}^{D} w_d x_d + b \\ &= \boldsymbol{w}^\top \boldsymbol{x} + b \end{aligned}$$

其中 $\boldsymbol{w} = [w_1; w_2; \cdots; w_D] \in \mathbb{R}^D$ 是 D 维的权重向量，$b \in \mathbb{R}$ 是偏置.

净输入 **z** 在经过一个非线性函数 $f(\cdot)$ 后, 得到神经元的活性值 (Activation) **a** ,

$$a = f(z)$$

其中非线性函数 $f(\cdot)$ 称为激活函数 (Activation Function).



1.3.2 激活函数

激活函数在神经元中非常重要的。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. 连续并可导(允许少数点上不可导)的非线性函数.可导的激活函数可以直接利用数值优化的方法来学习网络参数.
2. 激活函数及其导函数要尽可能的简单,有利于提高网络计算效率.
3. 激活函数的导函数的值域要在一个合适的区间内,不能太大也不能太小,否则会影响训练的效率和稳定性.

下面介绍几种在神经网络中常用的激活函数.

1.3.2.1 Sigmoid型函数

Sigmoid型函数是指一类S型曲线函数,为两端饱和函数.常用的Sigmoid型函数有Logistic 函数和Tanh 函数.

1.3.2.1.1 Logistic

Logistic 函数定义为

$$\sigma(x) = \frac{1}{1 + \exp(-x)}$$

Logistic 函数可以看成是一个“挤压” 函数， 把一个实数域的输入“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． 输入越小， 越接近于 0；输入越大， 越接近于 1． 这样的特点也和生物神经元类似， 对一些输入会产生兴奋（ 输出为1）， 对另一些输入产生抑制（ 输出为0）． 和感知器使用的阶跃激活函数相比， Logistic函数是连续可导的，其数学性质更好．因为Logistic函数的性质， 使得装备了Logistic激活函数的神经元具有以下两点性质:

1. 其输出直接可以看作概率分布， 使得神经网络可以更好地和统计学习模型进行结合．

2. 其可以看作一个软性门（ Soft Gate）， 用来控制其他神经元输出信息的数量.

1.3.2.1.2 Tanh

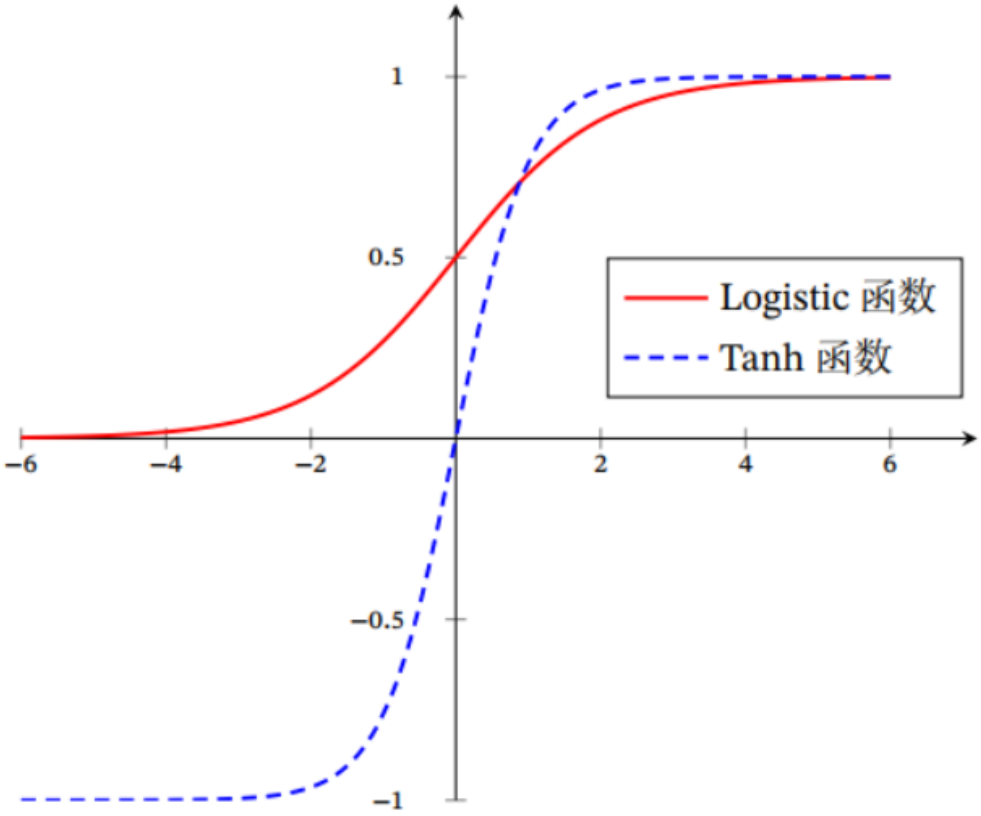
Tanh 函数也是一种 Sigmoid 型函数. 其定义为

$$\tanh(x) = \frac{\exp(x) - \exp(-x)}{\exp(x) + \exp(-x)}$$

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

$$\tanh(x) = 2\sigma(2x) - 1$$

下图给出了 Logistic 函数和 Tanh 函数的形状． Tanh 函数的输出是零中心化的（ Zero-Centered），而 Logistic 函数的输出恒大于 0． 非零中心化的输出会使得其后一层的神经元的输入发生偏置偏移（ Bias Shift）， 并进一步使得梯度下降的收敛速度变慢 .



Logistic函数和Tanh函数都是Sigmoid型函数， 具有饱和性， 但是计算开销较大． 因为这两个函数都是在中间（ 0附近） 近似线性， 两端饱和． 因此， 这两个函数可以通过分段函数来近似 .

以 Logistic 函数 $\sigma(x)$ 为例, 其导数为 $\sigma'(x) = \sigma(x)(1 - \sigma(x))$. Logistic 函数 在 0 附近的一阶泰勒展开 (Taylor expansion) 为

$$\begin{aligned} g_l(x) &\approx \sigma(0) + x \times \sigma'(0) \\ &= 0.25x + 0.5 \end{aligned}$$

这样 Logistic 函数可以用分段函数 hard-logistic（x）来近似.

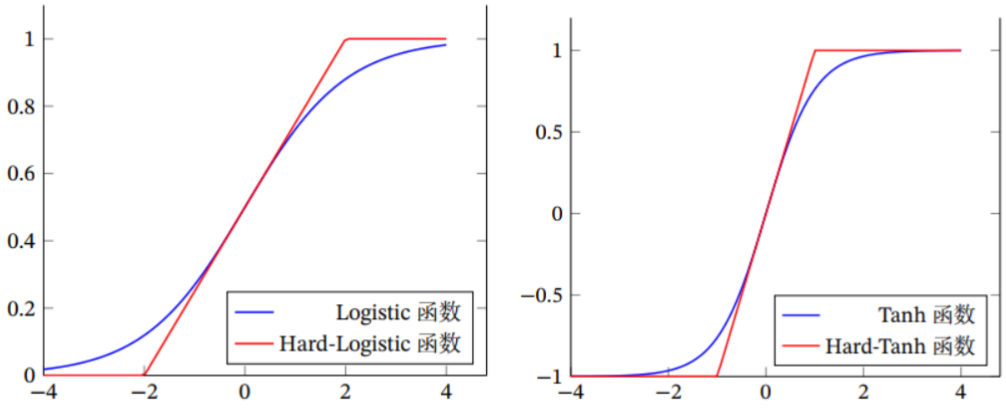
$$\begin{aligned} \mathrm{hard-logistic}(x) &= \begin{cases} 1 & g_l(x) \geq 1 \\ g_l & 0 < g_l(x) < 1 \\ 0 & g_l(x) \leq 0 \end{cases} \\ &= \max(\min(g_l(x), 1), 0) \\ &= \max(\min(0.25x + 0.5, 1), 0) \end{aligned}$$

同样, Tanh 函数在 0 附近的一阶泰勒展开为

$$\begin{aligned} g_t(x) &\approx \tanh(0) + x \times \tanh'(0) \\ &= x \end{aligned}$$

这样 Tanh 函数也可以用分段函数$\mathrm{hard}-\tanh(x)$来近似.

$$\begin{aligned} \mathrm{hard}-\tanh(x) &= \max(\min(g_t(x), 1), -1) \\ &= \max(\min(x, 1), -1) \end{aligned}$$



(a) Hard Logistic 函数　　(b) Hard Tanh 函数

1.3.2.2 ReLU函数

ReLU（ Rectified Linear Unit， 修正线性单元）， 也叫Rectifier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

$$\begin{aligned} \mathrm{ReLU}(x) &= \begin{cases} x & x \geq 0 \\ 0 & x < 0 \end{cases} \\ &= \max(0, x) \end{aligned}$$

优点 采用 ReLU 的神经元只需要进行加、 乘和比较的操作， 计算上更加高效．ReLU 函数也被认为具有生物学合理性（ Biological Plausibility）， 比如单侧抑制、宽兴奋边界（ 即兴奋程度可以非常高）．
在生物神经网络中， 同时处于兴奋状态的神经元非常稀疏． 人脑中在同一时刻大概只有 1% ~ 4% 的神经元处于活跃状态． Sigmoid 型激活函数会导致一个非稀疏的神经网络， 而 ReLU 却具有很好的稀疏性， 大约50%的神经元会处于激活状态．
在优化方面， 相比于Sigmoid型函数的两端饱和， ReLU函数为左饱和函数，且在 $x > 0$ 时导数为 1， 在一定程度上缓解了神经网络的梯度消失问题， 加速梯度下降的收敛速度．

缺点 ReLU 函数的输出是非零中心化的， 给后一层的神经网络引入偏置偏移，会影响梯度下降的效率．此外， ReLU 神经元在训练时比较容易“死亡”． 在训练时， 如果参数在一次不恰当的更新后， 第一个隐藏层中的某个 ReLU 神经元在所有的训练数据上都不能被激活， 那么这个神经元自身参数的梯度永远都会是0， 在以后的训练过程中永远不能被激活． 这种现象称为死亡 ReLU 问题（ Dying ReLU Problem),并且也有可能会发生在其他隐藏层.

在实际使用中,为了避免上述情况,有几种 ReLU的变种也会被广泛使用.

1.3.2.2.1 带泄露的ReLU

带泄露的ReLU（ Leaky ReLU） 在输入 $x < 0$时， 保持一个很小的梯度γ． 这样当神经元非激活时也能有一个非零的梯度可以更新参数， 避免永远不能被激活[Maas et al., 2013]． 带泄露的ReLU的定义如下：

$$\begin{aligned}\text{LeakyReLU}(x) &= \begin{cases} x & \text{if } x > 0 \\ \gamma x & \text{if } x \leq 0 \end{cases} \\ &= \max(0, x) + \gamma \min(0, x)\end{aligned}$$

其中 γ是一个很小的常数， 比如0.01． 当γ < 1时， 带泄露的ReLU也可以写为

$$\text{LeakyReLU}(x) = \max(x, \gamma x)$$

相当于是一个比较简单的maxout单元 .

1.3.2.2.2 带参数的ReLU

带参数的 ReLU（ Parametric ReLU， PReLU） 引入一个可学习的参数， 不同神经元可以有不同的参数． 对于第 i 个神经元， 其 PReLU 的定义为

$$\begin{aligned}\text{PReLU}_i(x) &= \begin{cases} x & \text{if } x > 0 \\ \gamma_i x & \text{if } x \leq 0 \end{cases} \\ &= \max(0, x) + \gamma_i \min(0, x)\end{aligned}$$

其中 γi 为 $x \leq 0$ 时函数的斜率． 因此， PReLU 是非饱和函数． 如果 γi = 0， 那么PReLU就退化为ReLU． 如果γi 为一个很小的常数， 则PReLU可以看作带泄露的ReLU． PReLU 可以允许不同神经元具有不同的参数， 也可以一组神经元共享一个参数．

1.3.2.2.3 ELU函数

ELU（ Exponential Linear Unit， 指数线性单元）是一个近似的零中心化的非线性函数， 其定义为

$$\begin{aligned}\text{ELU}(x) &= \begin{cases} x & \text{if } x > 0 \\ \gamma(\exp(x) - 1) & \text{if } x \leq 0 \end{cases} \\ &= \max(0, x) + \min(0, \gamma(\exp(x) - 1))\end{aligned}$$

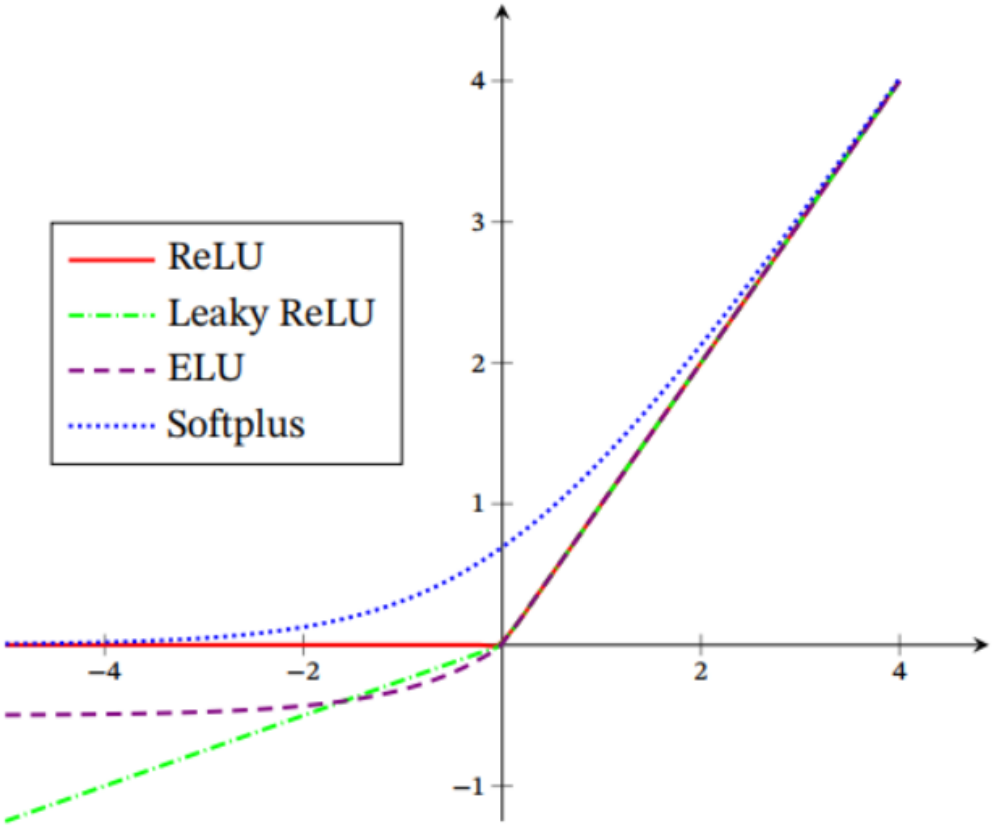
其中 $\gamma \geq 0$是一个超参数， 决定$x \leq 0$时的饱和曲线， 并调整输出均值在0附近．

1.3.2.2.4 Softplus函数

Softplus 函数 可以看作 Rectifier 函数的平滑版本， 其定义为

$$\text{Softplus}(x) = \log(1 + \exp(x))$$

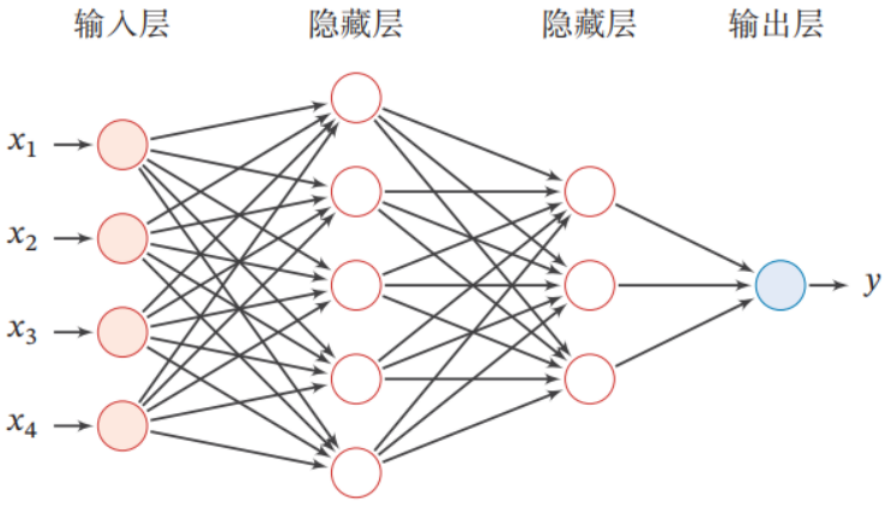
Softplus函数其导数刚好是Logistic函数． Softplus函数虽然也具有单侧抑制、宽兴奋边界的特性， 却没有稀疏激活性 .



1.3.3 多层前馈神经网络模型

给定一组神经元， 我们可以将神经元作为节点来构建一个网络． 不同的神经网络模型有着不同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） ． 但多层感知器的叫法并不是十分合理， 因为前馈神经网络其实是由多层的 Logistic 回归模型（ 连续的非线性函数） 组成， 而不是由多层的感知器（ 不连续的非线性函数） 组成 .

前馈神经网络中， 各神经元分别属于不同的层． 每一层的神经元可以接收前一层神经元的信号， 并产生信号输出到下一层． 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． 整个网络中无反馈， 信号从输入层向输出层单向传播， 可用一个有向无环图表示 .



对于本次作业中的多分类问题 $y \in \{1, \cdots, C\}$, 使用 Softmax 回归分类器, 相当于网络 最后一层设置 C 个神经元, 其激活函数为 Softmax 函数. 网络最后一层 (第 L 层) 的输出可以作为每个类的条件概率, 即

$$\hat{\boldsymbol{y}} = \text{softmax}\left(\boldsymbol{z}^{(L)}\right)$$

其中 $\boldsymbol{z}^{(L)} \in \mathbb{R}^C$为第 L 层神经元的净输入; $\hat{\boldsymbol{y}} \in \mathbb{R}^C$为第 L 层神经元的活性值, 每 一维分别表示不同类别标签的预测条件概率.

故采用交叉熵损失函数, 对于样本 $(\boldsymbol{x}, y)$, 其损失函数为

$$\mathcal{L}(\boldsymbol{y}, \hat{\boldsymbol{y}}) = -\boldsymbol{y}^\top \log \hat{\boldsymbol{y}},$$

其中 $\boldsymbol{y} \in \{0,1\}^C$为标签 y 对应的 one-hot 向量表示.
给定训练集为 $\mathcal{D} = \left\{\left(\boldsymbol{x}^{(n)}, \boldsymbol{y}^{(n)}\right)\right\}_{n=1}^N$, 将每个样本 $\boldsymbol{x}^{(n)}$输入给前馈神经网络，得到网络输出为$\hat{\boldsymbol{y}}^{(n)}$, 其在数据集 $\mathcal{D}$上的结构化风险函数为

$$\mathcal{R}(\boldsymbol{W}, \boldsymbol{b}) = \frac{1}{N}\sum_{n=1}^N \mathcal{L}\left(\boldsymbol{y}^{(n)}, \hat{\boldsymbol{y}}^{(n)}\right) + \frac{1}{2}\lambda \|\boldsymbol{W}\|_F^2$$

其中$\boldsymbol{W}$ 和 $\boldsymbol{b}$分别表示网络中所有的权重矩阵和偏置向量; $\|\boldsymbol{W}\|_F^2$是正则化项, 用 来防止过拟合; $\lambda > 0$ 为超参数. λ 越大, $\boldsymbol{W}$ 越接近于 0 . 这里的 $\|\boldsymbol{W}\|_F^2$一般使用 Frobenius 范数:

$$\|\boldsymbol{W}\|_F^2 = \sum_{l=1}^L \sum_{i=1}^{M_l} \sum_{j=1}^{M_{l-1}} \left(w_{ij}^{(l)}\right)^2$$

有了学习准则和训练样本, 网络参数可以通过梯度下降法来进行学习. 在梯度下降方法的每次迭代中,第 l 层的参数 $\boldsymbol{W}^{(l)}$ 和 $\boldsymbol{b}^{(l)}$参数更新方式为

$$\begin{aligned}
\boldsymbol{W}^{(l)} &\leftarrow \boldsymbol{W}^{(l)} - \alpha \frac{\partial \mathcal{R}(\boldsymbol{W}, \boldsymbol{b})}{\partial \boldsymbol{W}^{(l)}} \\
&= \boldsymbol{W}^{(l)} - \alpha \left(\frac{1}{N}\sum_{n=1}^N \left(\frac{\partial \mathcal{L}\left(\boldsymbol{y}^{(n)}, \hat{\boldsymbol{y}}^{(n)}\right)}{\partial \boldsymbol{W}^{(l)}} \right) + \lambda \boldsymbol{W}^{(l)} \right) \\
\boldsymbol{b}^{(l)} &\leftarrow \boldsymbol{b}^{(l)} - \alpha \frac{\partial \mathcal{R}(\boldsymbol{W}, \boldsymbol{b})}{\partial \boldsymbol{b}^{(l)}} \\
&= \boldsymbol{b}^{(l)} - \alpha \left(\frac{1}{N}\sum_{n=1}^N \frac{\partial \mathcal{L}\left(\boldsymbol{y}^{(n)}, \hat{\boldsymbol{y}}^{(n)}\right)}{\partial \boldsymbol{b}^{(l)}} \right)
\end{aligned}$$

其中 α 为学习率.

梯度下降法需要计算损失函数对参数的偏导数, 如果通过链式法则逐一对每个参数进行求偏导比较低效. 在神经网络的训练中经常使用**反向传播算法**来高效地计算梯度.

二、描述训练模型所使用的算法

假设采用随机梯度下降进行神经网络参数学习, 给定一个样本$(\boldsymbol{x}, \boldsymbol{y})$, 将其输入到神经网络模型中, 得到网络输出为 $\hat{\boldsymbol{y}}$. 假设损失函数为$\mathcal{L}(\boldsymbol{y}, \hat{\boldsymbol{y}})$, 要进行参数学习就需要计算损失函数关于每个参数的导数.

不失一般性, 对第 l 层中的参数$\boldsymbol{W}^{(l)}$和 $\boldsymbol{b}^{(l)}$计算偏导数. 因为$\frac{\partial \mathcal{L}(\boldsymbol{y}, \hat{\boldsymbol{y}})}{\partial \boldsymbol{W}^{(l)}}$的计算 涉及向量对矩阵的微分, 十分繁琐, 因此我们先计算$\mathcal{L}(\boldsymbol{y}, \hat{\boldsymbol{y}})$关于参数矩阵中每个元素的偏导数 $\frac{\partial \mathcal{L}(\boldsymbol{y}, \hat{\boldsymbol{y}})}{\partial w_{ij}^{(l)}}$. 根据链式法则,

$$\begin{aligned}
\frac{\partial \mathcal{L}(\boldsymbol{y}, \hat{\boldsymbol{y}})}{\partial w_{ij}^{(l)}} &= \frac{\partial \boldsymbol{z}^{(l)}}{\partial w_{ij}^{(l)}} \frac{\partial \mathcal{L}(\boldsymbol{y}, \hat{\boldsymbol{y}})}{\partial \boldsymbol{z}^{(l)}} \\
\frac{\partial \mathcal{L}(\boldsymbol{y}, \hat{\boldsymbol{y}})}{\partial \boldsymbol{b}^{(l)}} &= \frac{\partial \boldsymbol{z}^{(l)}}{\partial \boldsymbol{b}^{(l)}} \frac{\partial \mathcal{L}(\boldsymbol{y}, \hat{\boldsymbol{y}})}{\partial \boldsymbol{z}^{(l)}}
\end{aligned}$$

以上两个公式中的第二项都是目标函数关于第 l 层的神经元 $\boldsymbol{z}^{(l)}$的偏导数,称为误差项, 可以一次计算得到. 这样我们只需要计算三个偏导数, 分别为 $\frac{\partial \boldsymbol{z}^{(l)}}{\partial w_{ij}^{(l)}}$, $\frac{\partial \boldsymbol{z}^{(l)}}{\partial \boldsymbol{b}^{(l)}}$ 和 $\frac{\partial \mathcal{L}(\boldsymbol{y}, \hat{\boldsymbol{y}})}{\partial \boldsymbol{z}^{(l)}}$.

2.1 偏导数计算

下面分别来计算这三个偏导数:

(1) 计算偏导数 $\frac{\partial z^{(l)}}{\partial w_{ij}^{(l)}}$因 $z^{(l)}=\boldsymbol{W}^{(l)}\boldsymbol{a}^{(l-1)}+\boldsymbol{b}^{(l)}$, 偏导数

$$\begin{aligned}\frac{\partial \boldsymbol{z}^{(l)}}{\partial w_{ij}^{(l)}} &= [\frac{\partial z_1^{(l)}}{\partial w_{ij}^{(l)}},\cdots,\frac{\partial z_i^{(l)}}{\partial w_{ij}^{(l)}}\cdots,\frac{\partial z_{M_l}^{(l)}}{\partial w_{ij}^{(l)}}] \\ &= [0,\cdots,\frac{\partial\left(\boldsymbol{w}_{i:}^{(l)}\boldsymbol{a}^{(l-1)}+b_i^{(l)}\right)}{\partial w_{ij}^{(l)}},\cdots,0] \\ &= \left[0,\cdots,a_j^{(l-1)},\cdots,0\right] \\ &\triangleq \mathbb{I}_i\left(a_j^{(l-1)}\right) \quad \in \mathbb{R}^{1\times M_l}\end{aligned}$$

其中 $\boldsymbol{w}_{i:}^{(l)}$为权重矩阵 $\boldsymbol{W}^{(l)}$ 的第 i 行,$_i\left(a_j^{(l-1)}\right)$ 表示第 i 个元素为 $a_j^{(l-1)}$, 其余为 0 的行向量.

(2) 计算偏导数 $\frac{\partial z^{(l)}}{\partial b^{(l)}}$ 因为 $z^{(l)}$和 $\boldsymbol{b}^{(l)}$ 的函数关系为 $\boldsymbol{z}^{(l)}=\boldsymbol{W}^{(l)}\boldsymbol{a}^{(l-1)}+\boldsymbol{b}^{(l)}$, 因此偏导数

$$\frac{\partial \boldsymbol{z}^{(l)}}{\partial \boldsymbol{b}^{(l)}}=\boldsymbol{I}_{M_l} \quad \in \mathbb{R}^{M_l\times M_l}$$

为 $M_l\times M_l$的单位矩阵.

(3) 计算偏导数 $\frac{\partial \mathcal{L}(\boldsymbol{y},\hat{\boldsymbol{y}})}{\partial \boldsymbol{z}^{(l)}}$ 偏导数$\frac{\partial \mathcal{L}(\boldsymbol{y},\hat{\boldsymbol{y}})}{\partial \boldsymbol{z}^{(l)}}$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, 因此一般称为第 l 层神经 元的误差项,用 $\delta^{(l)}$ 来表示.

$$\delta^{(l)}\triangleq\frac{\partial \mathcal{L}(\boldsymbol{y},\hat{\boldsymbol{y}})}{\partial \boldsymbol{z}^{(l)}}\in\mathbb{R}^{M_l}$$

2.2 误差项

误差项 $\delta^{(l)}$也间接反映了不同神经元对网络能力的贡献程度, 从而比较好地解决 了贡献度分配问题 (Credit Assignment Problem,CAP).

根据 $\boldsymbol{z}^{(l+1)}=\boldsymbol{W}^{(l+1)}\boldsymbol{a}^{(l)}+\boldsymbol{b}^{(l+1)}$, 有

$$\frac{\partial \boldsymbol{z}^{(l+1)}}{\partial \boldsymbol{\alpha}^{(l)}}=\left(\boldsymbol{W}^{(l+1)}\right)^{\top} \quad \in \mathbb{R}^{M_l\times M_{l+1}}$$

根据 $\boldsymbol{a}^{(l)}=f_l\left(\boldsymbol{z}^{(l)}\right)$, 其中 $f_l(\cdot)$为按位计算的函数, 因此有

$$\begin{aligned}\frac{\partial \boldsymbol{\alpha}^{(l)}}{\partial \boldsymbol{z}^{(l)}} &= \frac{\partial f_l\left(\boldsymbol{z}^{(l)}\right)}{\partial \boldsymbol{z}^{(l)}} \\ &= \operatorname{diag}\left(f_l'\left(\boldsymbol{z}^{(l)}\right)\right) \quad \in \mathbb{R}^{M_l\times M_l}\end{aligned}$$

因此,根据链式法则,第 l 层的误差项为

$$
\begin{aligned}
\delta^{(l)} &\triangleq \frac{\partial \mathcal{L}(\boldsymbol{y}, \hat{\boldsymbol{y}})}{\partial \boldsymbol{z}^{(l)}} \\
&= \frac{\partial \boldsymbol{a}^{(l)}}{\partial \boldsymbol{z}^{(l)}} \cdot \frac{\partial \boldsymbol{z}^{(l+1)}}{\partial \boldsymbol{a}^{(l)}} \cdot \frac{\partial \mathcal{L}(\boldsymbol{y}, \hat{\boldsymbol{y}})}{\partial \boldsymbol{z}^{(l+1)}} \\
&= \operatorname{diag}\left(f_l'\left(\boldsymbol{z}^{(l)}\right)\right) \cdot \left(\boldsymbol{W}^{(l+1)}\right)^{\top} \cdot \delta^{(l+1)} \\
&= f_l'\left(\boldsymbol{z}^{(l)}\right) \odot \left(\left(\boldsymbol{W}^{(l+1)}\right)^{\top} \delta^{(l+1)}\right) \quad \in \mathbb{R}^{M_l},
\end{aligned}
$$

其中 ⊙是向量的 Hadamard 积运算符, 表示每个元素相乘.
从上面的公式可以看出,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 (BackPropagation, BP). 反向传播算法的含义是: 第 l 层的一个神经元的误差项 (或敏感性) 是所有与该神经元相连的第 l+1 层 的神经元的误差项的权重和. 然后, 再乘上该神经元激活函数的梯度.
在计算出上面三个偏导数之后,

$$
\begin{aligned}
\frac{\partial \mathcal{L}(\boldsymbol{y}, \hat{\boldsymbol{y}})}{\partial w_{ij}^{(l)}} &= l_i\left(a_j^{(l-1)}\right) \delta^{(l)} \\
&= \left[0, \cdots, a_j^{(l-1)}, \cdots, 0\right]\left[\delta_1^{(l)}, \cdots, \delta_i^{(l)}, \cdots, \delta_{M_l}^{(l)}\right]^{\top} \\
&= \delta_i^{(l)} a_j^{(l-1)}
\end{aligned}
$$

其中 $\delta_i^{(l)} a_j^{(l-1)}$相当于向量$\delta^{(l)}$ 和向量 $\boldsymbol{a}^{(l-1)}$的外积的第 i, j 个元素. 上式可以进 一步写为

$$
\left[\frac{\partial \mathcal{L}(\boldsymbol{y}, \hat{\boldsymbol{y}})}{\partial \boldsymbol{W}^{(l)}}\right]_{ij} = \left[\delta^{(l)}\left(\boldsymbol{a}^{(l-1)}\right)^{\top}\right]_{ij}
$$

因此, $\mathcal{L}(\boldsymbol{y}, \hat{\boldsymbol{y}})$关于第 l 层权重 $\boldsymbol{W}^{(l)}$的梯度为

$$
\frac{\partial \mathcal{L}(\boldsymbol{y}, \hat{\boldsymbol{y}})}{\partial \boldsymbol{W}^{(l)}} = \delta^{(l)}\left(\boldsymbol{a}^{(l-1)}\right)^{\top} \quad \in \mathbb{R}^{M_l \times M_{l-1}}
$$

同理, $\mathcal{L}(\boldsymbol{y}, \hat{\boldsymbol{y}})$关于第 l 层偏置 $\boldsymbol{b}^{(l)}$的梯度为

$$
\frac{\partial \mathcal{L}(\boldsymbol{y}, \hat{\boldsymbol{y}})}{\partial \boldsymbol{b}^{(l)}} = \delta^{(l)} \quad \in \mathbb{R}^{M_l}
$$

2.3 实现步骤与伪代码

在计算出每一层的误差项之后, 我们就可以得到每一层参数的梯度. 因此, 使用误差反向传播算法的前馈神经网络训练过程可以分为以下三步:

1. 前馈计算每一层的净输入 $\boldsymbol{z}^{(l)}$和激活值 $\boldsymbol{a}^{(l)}$, 直到最后一层;
2. 反向传播计算每一层的误差项 δ(l) ;
3. 计算每一层参数的偏导数, 并更新参数.

使用反向传播算法的随机梯度下降训练过程:

输入: 训练集 $\mathcal{D}=\{(\boldsymbol{x}^{(n)},y^{(n)})\}_{n=1}^{N}$, 验证集 $\mathcal{V}$, 学习率 α, 正则化系数 λ, 网络层数 L, 神经元数量 $M_l, 1\leq l\leq L$.

随机初始化 $\boldsymbol{W},\boldsymbol{b}$;

repeat

  对训练集 $\mathcal{D}$ 中的样本随机重排序;

  for $n=1\cdots N$ **do**

    从训练集 $\mathcal{D}$ 中选取样本 $(\boldsymbol{x}^{(n)},y^{(n)})$;

    前馈计算每一层的净输入 $\boldsymbol{z}^{(l)}$ 和激活值 $\boldsymbol{a}^{(l)}$, 直到最后一层;

    反向传播计算每一层的误差 $\delta^{(l)}$;

    // 计算每一层参数的导数

    $\forall l,\quad \frac{\partial \mathcal{L}(y^{(n)},\hat{y}^{(n)})}{\partial \boldsymbol{W}^{(l)}}=\delta^{(l)}(\boldsymbol{a}^{(l-1)})^{\mathrm{T}}$;

    $\forall l,\quad \frac{\partial \mathcal{L}(y^{(n)},\hat{y}^{(n)})}{\partial \boldsymbol{b}^{(l)}}=\delta^{(l)}$;

    // 更新参数

    $\boldsymbol{W}^{(l)}\leftarrow \boldsymbol{W}^{(l)}-\alpha(\delta^{(l)}(\boldsymbol{a}^{(l-1)})^{\mathrm{T}}+\lambda \boldsymbol{W}^{(l)})$;

    $\boldsymbol{b}^{(l)}\leftarrow \boldsymbol{b}^{(l)}-\alpha\delta^{(l)}$;

  end

until 神经网络模型在验证集 $\mathcal{V}$ 上的错误率不再下降;

输出: $\boldsymbol{W},\boldsymbol{b}$

三、描述模型超参数确定的过程，分析模型训练结果

在神经网络中， 除了可学习的参数之外， 还存在很多超参数. 这些超参数对网络性能的影响也很大. 不同的机器学习任务往往需要不同的超参数. 常见的超参数有以下三类:

1. 网络结构,包括神经元之间的连接关系、层数、每层的神经元数量、激活函数的类型等.
2. 优化参数,包括优化方法、学习率、小批量的样本数量等.
3. 正则化系数.

超参数优化（ Hyperparameter Optimization） 主要存在两方面的困难:

- 超参数优化是一个组合优化问题， 无法像一般参数那样通过梯度下降方法来优化， 也没有一种通用有效的优化方法;
- 评估一组超参数配置（ Configuration）的时间代价非常高， 从而导致一些优化方法（比如演化算法（ Evolution Algorithm）） 在超参数优化中难以应用

对于超参数的配置， 比较简单的方法有网格搜索、随机搜索、贝叶斯优化、动态资源分配和神经架构搜索， 这次我们介绍最后两种。

3.1 动态资源分配

在超参数优化中， 每组超参数配置的评估代价比较高. 如果我们可以在较早的阶段就估计出一组配置的效果会比较差， 那么我们就可以中止这组配置的评估， 将更多的资源留给其他配置. 这个问题可以归结为多臂赌博机问题的一个泛化问题： **最优臂问题（ Best-Arm Problem）**， 即在给定有限的机会次数下， 如何玩这些赌博机并找到收益最大的臂. 和多臂赌博机问题类似， 最优臂问题也是在利用和探索之间找到最佳的平衡.

由于目前神经网络的优化方法一般都采取随机梯度下降， 因此我们可以通过一组超参数的学习曲线来预估这组超参数配置是否有希望得到比较好的结果． 如果一组超参数配置的学习曲线不收敛或者收敛比较差， 我们可以应用**早期停止（Early-Stopping）** 策略来中止当前的训练．

动态资源分配的关键是将有限的资源分配给更有可能带来收益的超参数组合． 一种有效方法是**逐次减半（Successive Halving）** 方法， 将超参数优化看作一种非随机的最优臂问题． 假设要尝试 N 组超参数配置， 总共可利用的资源预算（摇臂的次数） 为B， 我们可以通过T = [log2(N)] - 1轮逐次减半的方法来选取最优的配置.

3.1.1 实现步骤与伪代码

输入: 预算 B，N 个超参数配置 $\{\boldsymbol{x}_n\}_{n=1}^N$

$T \leftarrow \lceil \log_2(N) \rceil - 1$;

随机初始化 $\mathcal{S}_0 = \{\boldsymbol{x}_n\}_{n=1}^N$;

for $t \leftarrow 1$ **to** T **do**

  $r_t \leftarrow \lfloor \frac{B}{|\mathcal{S}_t| \times T} \rfloor$;

  给 S_t 中的每组配置分配 r_t 的资源;

  运行 S_t 所有配置，评估结果为 $\boldsymbol{y}_t$;

  根据评估结果，选取 $|S_t|/2$ 组最优的配置 $\mathcal{S}_t \leftarrow \arg\max(\mathcal{S}_t, \boldsymbol{y}_t, |\mathcal{S}_t|/2)$;

  // $\arg\max(\mathcal{S}, \boldsymbol{y}, m)$ 为从集合 $\mathcal{S}$ 中选取 m 个元素，对应最优的 m 个评估结果.

end

输出: 最优配置 $\mathcal{S}_K$

在逐次减半方法中， 尝试的超参数配置数量N 十分关键． 如果N 越大， 得到最佳配置的机会也越大，但每组配置分到的资源就越少， 这样早期的评估结果可能不准确． 反之， 如果N 越小， 每组超参数配置的评估会越准确， 但有可能无法得到最优的配置． 因此， 如何设置 N 是平衡“利用-探索” 的一个关键因素． 一种改进的方法是**HyperBand方法**， 通过尝试不同的N 来选取最优参数．

3.2 神经架构搜索

上面介绍的超参数优化方法都是在固定（或变化比较小） 的超参数空间 $\mathcal{X}$中进行最优配置搜索， 而最重要的神经网络架构一般还是需要由有经验的专家来进行设计．神经架构搜索（Neural Architecture Search， NAS） 是一个新的比较有前景的研究方向， 通过神经网络来自动实现网络架构的设计． 一个神经网络的架构可以用一个变长的字符串来描述． 利用元学习的思想， 神经架构搜索利用一个控制器来生成另一个子网络的架构描述． 控制器可以由一个循环神经网络来实现． 控制器的训练可以通过强化学习来完成， 其奖励信号为生成的子网络在开发集上的准确率 ．

3.3 结果分析

Hyper-parameter	Binary classification	Multilabel binary classification	Multiclass classification
Input and hidden layers	Same as regression	Same as regression	Same as regression
# output neurons	1	1 per label	1 per class
Output layer activation	Logistic	Logistic	Softmax
Loss function	Cross entropy	Cross entropy	Cross entropy

在两次报告中我们一共介绍了5种超参数调优方式，也都尝试过应用到实验中，但并没有获得很大的提升。且结合数据量较小，网络结构较简单的实际，我们依旧选择了根据一些"经验结论"来进行网格搜索。

默认参数下，五次五折交叉验证结果：

Accuracy: 0.95

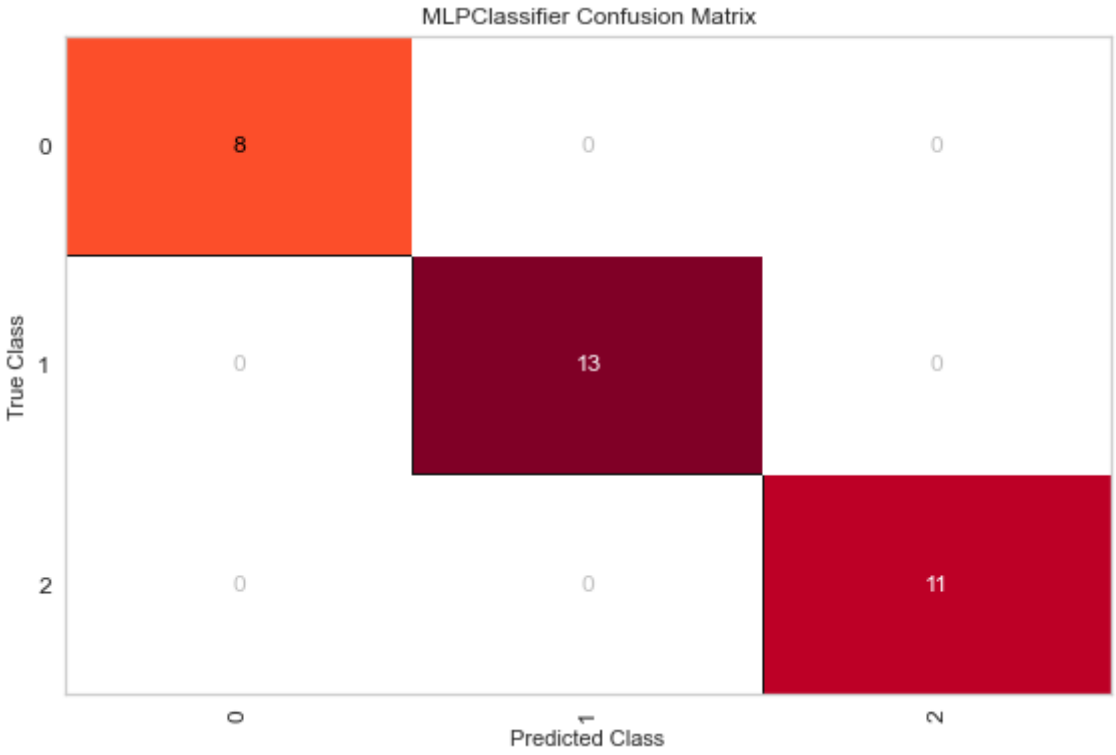
使用代码如下：

```
from sklearn.neural_network import MLPClassifier
from sklearn.model_selection import cross_val_score
import warnings
warnings.filterwarnings('ignore')
mlp = MLPClassifier(hidden_layer_sizes=(100), activation='relu', solver='adam',
alpha=0.0001, batch_size='auto', learning_rate='constant',
learning_rate_init=0.001, power_t=0.5, max_iter=200, shuffle=True,
random_state=None, tol=0.0001, verbose=False, warm_start=False, momentum=0.9,
nesterovs_momentum=True, early_stopping=False, validation_fraction=0.1,
beta_1=0.9, beta_2=0.999, epsilon=1e-08, n_iter_no_change=10, max_fun=15000) #均
以官网默认参数设置
mlp.fit(X_std, Y)
ACC = cross_val_score(mlp, X_std, Y, cv=rkf,scoring='accuracy').mean()
print('Accuracy:',ACC)
```

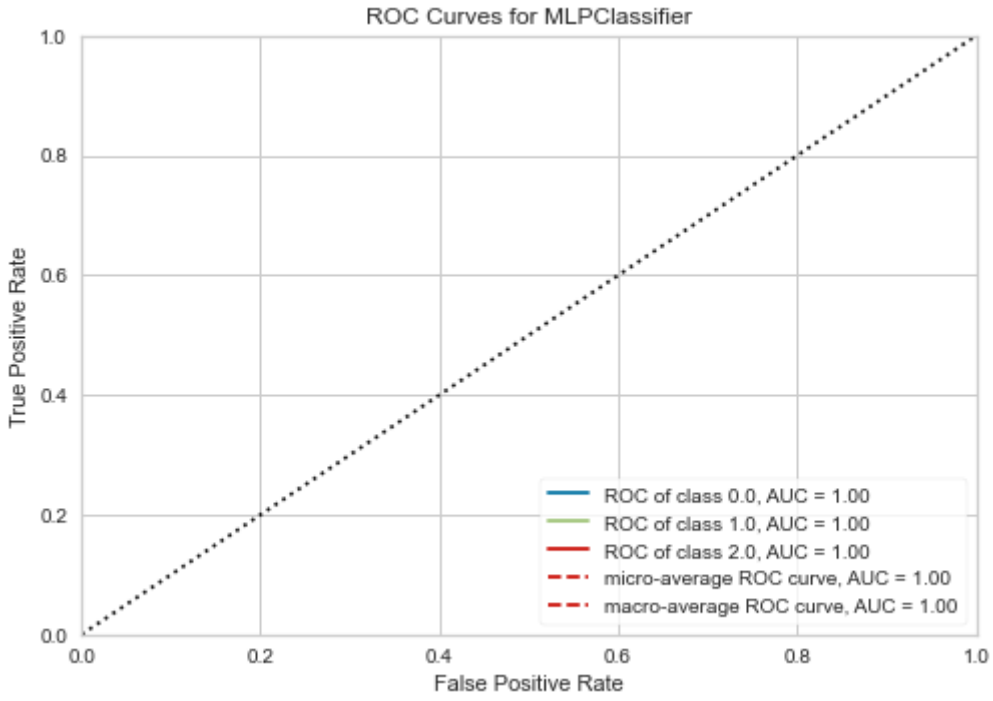
网格搜索最优参数的结果为：

Accuracy: 0.96125

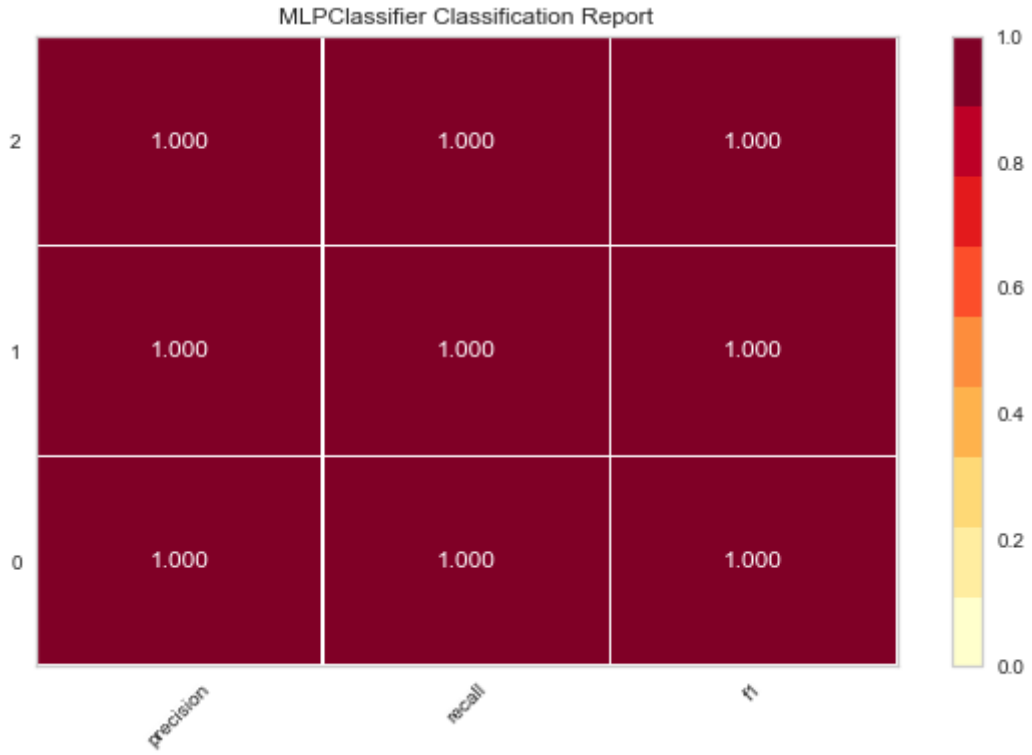
混淆矩阵如图：



ROC曲线如图：



评价指标（精确率，召回率和F1值）：



从上面可以看出，我们的模型分类结果完全正确，甚至有可能出现了过拟合。为了避免这种情况，我们继续修改参数，发现即使不设置任何超参数也可以达到这样的准确率，也从另一方面证明了神经网络拟合能力的强悍。

网格搜索代码如下：

```
# 超参数调优
import warnings
warnings.filterwarnings('ignore')
from sklearn.model_selection import GridSearchCV
parameters = {'hidden_layer_sizes': [(10,10,10,10,10),(20,20,20,20,20),
(30,30,30,30,30),(40,40,40,40,40),(50,50,50,50,50),(60,60,60,60,60),
(70,70,70,70,70),(80,80,80,80,80),(90,90,90,90,90),(100,100,100,100,100)],
                'activation': ['identity', 'logistic','tanh', 'relu'],
                'solver': ['adam','lbgfs','sgd'],
                'alpha': [0.0001, 0.001, 0.01, 0.1, 1,10,100],
                'learning_rate': ['constant', 'invscaling', 'adaptive']}
grid = GridSearchCV(mlp, parameters, cv=rkf,
scoring='neg_mean_absolute_percentage_error',n_jobs=-1)
grid.fit(X_std, Y)
print('最优参数：',grid.best_params_)
print('最优模型得分：',grid.best_score_)
```

四、总结模型训练过程中的收获

4.1 优化与正则化

神经网络的优化和正则化是既对立又统一的关系． 一方面我们希望优化算法能找到一个全局最优解（或较好的局部最优解）， 另一方面我们又不希望模型优化到最优解， 这可能陷入过拟合． 优化和正则化的统一目标是期望风险最小化．

4.1.1 优化

在优化方面， 训练神经网络时的主要难点是非凸优化以及梯度消失问题． 在深度学习技术发展的初期，我们通常需要利用预训练和逐层训练等比较低效的
方法来辅助优化．随着深度学习技术的发展， 我们目前通常可以高效地、 端到端地训练一个深度神经网络． 这些提高训练效率的方法通常分为以下 3 个方面：

1. 修改网络模型来得到更好的优化地形， 比如使用逐层归一化、 残差连接以及ReLU激活函数等；
2. 使用更有效的优化算法， 比如动态学习率以及梯度估计修正等；
3. 使用更好的参数初始化方法

4.1.2 泛化

在泛化方面， 传统的机器学习中有一些很好的理论可以帮助我们在模型的表示能力、复杂度和泛化能力之间找到比较好的平衡， 但是这些理论无法解释深度神经网络在实际应用中的泛化能力表现． 根据通用近似定理，神经网络的表示能力十分强大． 从直觉上， 一个过度参数化的神经网络很容易产生过拟合现象， 因为它的容量足够记住所有训练数据． 但是实验表明， 神经网络在训练过程中依然优先记住训练数据中的一般模式（ Pattern）， 即具有高泛化能力的模式 ． 但目前， 神经网络的泛化能力还没有很好的理论支持． 在传统机器学习模型上比较有效的$\ell 1$ 或$\ell 2$ 正则化在深度神经网络中作用也比较有限，而一些经验的做法（ 比如小的批量大小、大的学习率、提前停止、丢弃法、数据增强） 会更有效 。

4.2 与回归的差异

和回归问题不同， 分类问题中的目标标签 y 是离散的类别标签， 因此分类问题中的决策函数需要输出离散值或是标签的后验概率． 线性分类模型一般是一个广义线性函数， 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”是指决策边界由一个或多个超平面组成．

线性模型	激活函数	损失函数	优化方法
线性回归	-	$(y-\boldsymbol{w}^{\mathrm{T}}\boldsymbol{x})^2$	最小二乘、梯度下降
Logistic 回归	$\sigma(\boldsymbol{w}^{\mathrm{T}}\boldsymbol{x})$	$\boldsymbol{y}\log\sigma(\boldsymbol{w}^{\mathrm{T}}\boldsymbol{x})$	梯度下降
Softmax 回归	$\mathrm{softmax}(\boldsymbol{W}^{\mathrm{T}}\boldsymbol{x})$	$\boldsymbol{y}\log\mathrm{softmax}(\boldsymbol{W}^{\mathrm{T}}\boldsymbol{x})$	梯度下降
感知器	$\mathrm{sgn}(\boldsymbol{w}^{\mathrm{T}}\boldsymbol{x})$	$\max(0,-y\boldsymbol{w}^{\mathrm{T}}\boldsymbol{x})$	随机梯度下降
支持向量机	$\mathrm{sgn}(\boldsymbol{w}^{\mathrm{T}}\boldsymbol{x})$	$\max(0,1-y\boldsymbol{w}^{\mathrm{T}}\boldsymbol{x})$	二次规划、SMO 等

4.3 对深度学习框架的初步了解

在深度学习中， 一般通过误差反向传播算法来进行参数学习． 采用手工方式来计算梯度再写代码实现的方式会非常低效， 并且容易出错． 此外， 深度学习模型需要的计算机资源比较多， 一般需要在 CPU 和 GPU 之间不断进行切换， 开发难度也比较大． 因此， 一些支持自动梯度计算、无缝CPU和GPU切换等功能的深度学习框架就应运而生．比较有代表性的框架包括： Theano、Caffe、TensorFlow、Pytorch、飞桨（ PaddlePaddle）、Chainer和MXNet等。

因此本次作业我们没有局限在scikit-learn中，我们了解并学习了TensorFlow，pytorch等框架的使用，搭建了简易的神经网络对本次作业的数据进行分析。虽然最终得到的效果不及sklearn，但是在体验的过程加深了对神经网络的理解。