深度前馈网络

朱明超
deityrayleigh@gmail.com

1 深度前馈网络

深度前馈网络 (Deep Feedforward Network，DFN) 也叫前馈神经网络 (Feedforward Neural Network，FNN) 或多层感知机 (Multilayer Perceptron，MLP)，是最典型的深度学习模型。目标是拟合一个函数，如有一个分类器 $y = f^*(x)$ 将输入 x 映射到输出类别 y 。深度前馈网将这个映射定义为 $f(x, \theta)$ 并学习这个参数 θ 的值来得到最好的函数拟合。

深度前馈网络中信息从 x 流入，通过中间 f 的计算，最后到达输出 y。如图 1 所示，假设有 $f^{(1)}, f^{(2)}, f^{(3)}$ 这三个函数链式连接，这个链式连接可以表示为 $f(x) = f^{(3)}(f^{(2)}(f^{(1)}(x)))$，这种链式结构是神经网络最为常用的结构。$f^{(1)}, f^{(2)}$ 被称为神经网络的第一层，第二层，也为网络的隐藏层 (Hidden Layer)，深度前馈网络最后一层 $f^{(3)}$ 就是输出层 (Output Layer)。这个链的长度就是神经网络的深度，输入向量的每个元素均视作一个神经元。

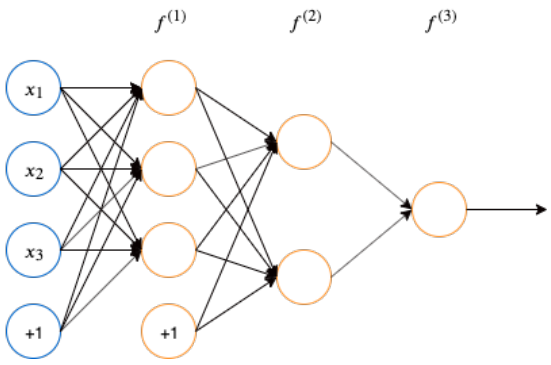


图 1. 深度前馈网络示意图

2 DFN 相关设计

DFN 内部的神经网络层可以分为三类，输入层，隐藏层和输出层。为了构造一个可训练的 DFN，我们需要考虑几个方面，包括隐藏单元，输出单元，和代价函数 (损失函数)。

2.1 隐藏单元

层与层之间是全连接的，也就是说，第 i 层的任意一个神经元一定与第 $i+1$ 层的任意一个神经元相连。如下图所示，大多数隐藏单元 (Hidden Unit) 都可以描述为接受输入向量 $\boldsymbol{x}$，计算仿射变换 $\boldsymbol{z} = \boldsymbol{W}^\top \boldsymbol{x} + \boldsymbol{b}$，然后使用一个逐元素的非线性函数 $g(\boldsymbol{z})$ 得到隐藏单元的输出 $\boldsymbol{a}$。而大多数隐藏单元的区别仅仅在于激活函数 $g(\boldsymbol{z})$ 的形式。(**这里增加激活函数，是为了提高模型的表达能力；如果不引入激活函数，可以验证，无论多少层神经网络，输出都是输入的线性组合**)

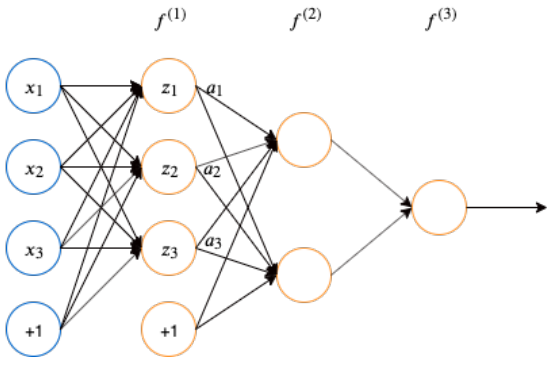


图 2. 深度前馈网络及第一层隐藏单元示意图

如图 2 所示，假设激活函数 $g(\boldsymbol{z})$ 为 σ，于是 $f^{(1)}$ 层的隐藏单元可以描述为：

$$\begin{cases} a_1 = \sigma(z_1) = \sigma(w_{11}x_1 + w_{12}x_2 + w_{13}x_3 + b_1) \\ a_2 = \sigma(z_2) = \sigma(w_{21}x_1 + w_{22}x_2 + w_{23}x_3 + b_2) \\ a_3 = \sigma(z_3) = \sigma(w_{31}x_1 + w_{32}x_2 + w_{33}x_3 + b_3) \end{cases} \tag{1}$$

选择隐藏单元实际上就是要选择一个合适的激活函数。常见的有如下几种激活函数：

- 整流线性单元 (ReLU)：$g(z) = \max\{0, z\}$。优点是易于优化，二阶导数几乎处处为 0，**处于激活状态时一阶导数处处为 1**，也就是其相比于引入二阶效应的激活函数，梯度方向对学习更有用。如果使用 ReLU，第一步做线性变换 $\boldsymbol{W}^\top \boldsymbol{x} + \boldsymbol{b}$ 时的 $\boldsymbol{b}$ 一般设置成小的正值。缺陷是不能通过基于梯度的方法学习那些使单元激活为 0 的样本。

  ReLU 函数的梯度为 $g'(z) = \begin{cases} 1 & z > 0 \\ 0 & z \leq 0 \end{cases}$

- sigmoid 函数 (常写作 σ) 或双曲正切函数 tanh。两者之间有一个联系：$\tanh(z) = 2\sigma(2z) - 1$。两者都比较容易饱和，仅当 z 接近 0 时才对输入强烈敏感，因此使得基于梯度的学习变得非常困难，不适合做前馈网络中的隐藏单元。**如果必须要使用这两种中的一个，那么 tanh 通常表现更好**，因为在 0 附近其类似于单位函数。即，如果网络的激活能一直很小，训练 $\hat{y} = w^\top \tanh(\mathrm{U}^\top \tanh(\mathrm{V}^\top x))$ 类似于训练一个线性模型 $\hat{y} = w^\top \mathrm{U}^\top \mathrm{V}^\top x$。RNN 和一些自编码器有一些额外的要求，因此不能使用分段激活函数，此时这种类 sigmoid 单元更合适。

  sigmoid 函数写作 $g(z) = \sigma(z) = \frac{1}{1+e^{-z}}$，梯度为 $g'(z) = \sigma(z)(1 - \sigma(z))$

  双曲正切函数写作 $g(z) = \tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$，梯度为 $g'(z) = 1 - \tanh^2(z)$

[1]:
```
from abc import ABC, abstractmethod
import numpy as np
import time
import re
from collections import OrderedDict
```

[2]:
```
class ActivationBase(ABC):

    def __init__(self, **kwargs):
        super().__init__()

    def __call__(self, z):
        if z.ndim == 1:
            z = z.reshape(1, -1)
        return self.forward(z)

    @abstractmethod
    def forward(self, z):
        """
        函数作用：前向传播，通过激活函数获得 a
        """
        raise NotImplementedError

    @abstractmethod
    def grad(self, x, **kwargs):
        """
        函数作用：反向传播，获得梯度
        """
        raise NotImplementedError


class ReLU(ActivationBase):
    """
    整流线性单元。
    """
    def __init__(self):
        super().__init__()

    def __str__(self):
        return "ReLU"

    def forward(self, z):
        return np.clip(z, 0, np.inf)

    def grad(self, x):
        return (x > 0).astype(int)
```

```
class Sigmoid(ActivationBase):
    """
    sigmoid 激活函数。更多激活函数见 ../method 文件夹。
    """
    def __init__(self):
        super().__init__()

    def __str__(self):
        return "Sigmoid"

    def forward(self, z):
        return 1 / (1 + np.exp(-z))

    def grad(self, x):
        return self.forward(x) * (1 - self.forward(x))

class Tanh(ActivationBase):
    """
    双曲正切函数。
    """
    def __init__(self):
        super().__init__()

    def __str__(self):
        return "Tanh"

    def forward(self, z):
        return np.tanh(z)

    def grad(self, x):
        return 1 - np.tanh(x) ** 2

class Affine(ActivationBase):
    """
    affine 激活函数， 即仿射变换。 输出 slope*z + intercept。当 slope=1 且 intercept=0 表示不做变换。
    """
    def __init__(self, slope=1, intercept=0):
        self.slope = slope
        self.intercept = intercept
        super().__init__()

    def __str__(self):
        return "Affine(slope={}, intercept={})".format(self.slope, self.intercept)

    def forward(self, z):
        return self.slope * z + self.intercept

    def grad(self, x):
        return self.slope * np.ones_like(x)

class ActivationInitializer(object):

    def __init__(self, acti_name='sigmoid'):
        self.acti_name = acti_name

    def __call__(self):
        if self.acti_name.lower() == 'sigmoid':
            acti_fn = Sigmoid()
        elif self.acti_name.lower() == 'relu':
```

```
            acti_fn = ReLU()
        elif "affine" in self.acti_name.lower():
            r = r"affine\(slope=(.*), intercept=(.*)\)"
            slope, intercept = re.match(r, self.acti_name.lower()).groups()
            acti_fn = Affine(float(slope), float(intercept))
        return acti_fn
```

2.2 输出单元

假设前面已经使用若干隐藏层提供了一组隐藏特征 $\boldsymbol{h}$，输出层是要对这些特征做一些额外变换来完成任务。

常见的有如下几种输出单元：

- 用于高斯输出分布的线性单元，即对隐藏特征不做非线性变换，直接产生 $\hat{\boldsymbol{y}} = \boldsymbol{z} = \boldsymbol{W}^\top \boldsymbol{h} + \boldsymbol{b}$。其用来产生条件高斯分布的均值：$p(\boldsymbol{y} \mid \boldsymbol{x}) = N(\boldsymbol{y}; \hat{\boldsymbol{y}}, \boldsymbol{I})$
- 用于伯努利输出分布的 sigmoid 单元，即对隐藏特征先用线性层求 $\boldsymbol{z} = \boldsymbol{W}^\top \boldsymbol{h} + \boldsymbol{b}$，然后对这个值做一个 sigmoid 变换 $\sigma(z)$ 将其映射到 $[0, 1]$ 区间，转化成一个**概率值**，即 $\hat{\boldsymbol{y}} = \sigma(\boldsymbol{z}) = \sigma(\boldsymbol{W}^\top \boldsymbol{h} + \boldsymbol{b})$
- 用于多元伯努利输出分布的 softmax 单元，可以看作是伯努利输出分布的 sigmoid 单元在多分类问题上的推广。此时**输出标签空间是一个离散的，多类别的集合**。假如一共有 K 个类别，则标签空间 $\mathcal{Y} = \{0, 1, 2, \ldots, \mathrm{K}-1\}$。对隐藏特征做线性变换 $\boldsymbol{z} = \boldsymbol{W}^\top \boldsymbol{h} + \boldsymbol{b}$ 后，通过 softmax 函数得到一个向量 $\mathrm{softmax}(\boldsymbol{z})$，这个向量的每个维度可以看做是输入样本属于对应类别标签的概率，因此有 $\forall i \in \{0, 1, \ldots, \mathrm{K}-1\}, z_i \in [0, 1] \wedge \sum_i z_i = 1$。softmax 函数的具体形式为：

$$\mathrm{softmax}(\boldsymbol{z})_i = \frac{\exp(z_i)}{\sum_j \exp(z_j)} \tag{2}$$

[3]:
```
def sigmoid(x):
    return 1 / (1 + np.exp(-x))


def softmax(x):
    e_x = np.exp(x - np.max(x, axis=-1, keepdims=True))
    return e_x / e_x.sum(axis=-1, keepdims=True)
```

2.3 代价函数

任何能够衡量模型预测值与真实值之间的差异的函数都可以叫做代价函数。如果有多个样本，则可以将所有代价函数的取值求均值，记作 $J(\theta)$。当我们确定了模型后，再要做的是训练模型的参数 θ (如 $\boldsymbol{W}$，$\boldsymbol{b}$)。**训练参数的过程就是误差反向传递，不断调整 θ ，从而得到更小的 $J(\theta)$ 的过程。**理想情况下，当我们取到代价函数 J 的最小值时，就得到了最优的参数 θ ，记为：$\min_\theta J(\theta)$ 。常见的代价函数主要有：

- 二次代价函数，具体形式为：

$$J(\theta) = \frac{1}{2m} \sum_{i=1}^{m} (\hat{\boldsymbol{y}}^{(i)} - \boldsymbol{y}^{(i)})^2 \tag{3}$$

  梯度的计算：以单个样本为例，假设输出单元 $\hat{\boldsymbol{y}} = g(\boldsymbol{z})$，$g$ 为输出单元的激活函数，$\boldsymbol{z}$ 为 θ 的函数，则此时代价函数为 $\frac{1}{2}(g(\boldsymbol{z}) - \boldsymbol{y})^2$ 。可以计算梯度：

$$\frac{\partial J(\theta)}{\partial \boldsymbol{z}} = (g(\boldsymbol{z}) - \boldsymbol{y}) g'(\boldsymbol{z}) \tag{4}$$

  可以验证，当输出单元激活函数采用 sigmoid 或 tanh 的 S 型激活函数，二次代价函数在梯度收敛至 0 时 (如果真实值为 0)，存在收敛速度慢而导致的训练速度慢的问题。
- 最大 (对数) 似然代价函数或者最小负 (对数) 似然代价函数，具体形式为：

$$\operatorname*{argmin}_\theta \; -\mathcal{L}(\theta \mid \boldsymbol{y}, \hat{\boldsymbol{y}}) \tag{5}$$

  似然 $\mathcal{L}(\theta \mid \boldsymbol{y}, \hat{\boldsymbol{y}})$ 可以表示成联合概率：

$$P(t, \boldsymbol{z} \mid \theta) = P(t \mid \boldsymbol{z}, \theta) P(\boldsymbol{z} \mid \theta) \tag{6}$$

  - 如果输出单元是用于伯努利输出分布的 sigmoid 单元，可以得到对应的代价函数：

$$J(\theta) = \operatorname*{argmin}_\theta \; -\mathcal{L}(\theta \mid \boldsymbol{y}, \hat{\boldsymbol{y}}) = -\frac{1}{m} \sum_{i=1}^{m} \left(\boldsymbol{y}^{(i)} \log \hat{\boldsymbol{y}}^{(i)} + (1 - \boldsymbol{y}^{(i)}) \log(1 - \hat{\boldsymbol{y}}^{(i)}) \right) \tag{7}$$

    梯度的计算：以单个样本为例，假设输出单元 $\hat{\boldsymbol{y}} = g(\boldsymbol{z})$ ，g 为输出单元的激活函数，$\boldsymbol{z}$ 为 θ 的函数。可以计算梯度：

$$\frac{\partial J(\theta)}{\partial \boldsymbol{z}} = -\left(\frac{\boldsymbol{y}}{g(\boldsymbol{z})} - \frac{(1 - \boldsymbol{y})}{1 - g(\boldsymbol{z})} \right) g'(\boldsymbol{z}) \tag{8}$$

    如果 g 为 sigmoid 函数，可以进一步化简：

$$J(\theta) = -\boldsymbol{y}\boldsymbol{z} + \log(1 + e^{\boldsymbol{z}}) \tag{9}$$

$$\frac{\partial \mathrm{J}(\theta)}{\partial \boldsymbol{z}} = \sigma(\boldsymbol{z}) - \boldsymbol{y} \tag{10}$$

$\sigma(\boldsymbol{z}) - \boldsymbol{y}$ 表示真实值与输出值之间的误差。当误差越大时，梯度就越大，θ 的调整就越快，训练速度就越快。**当输出神经元的激活函数是线性时 (如 ReLU 函数)，二次代价函数是一种合适的选择；当输出神经元的激活函数是 S 型函数时 (如 sigmoid 、tanh 函数)，选择交叉熵代价函数则比较合理**。

– 如果输出单元是用于多元伯努利输出分布的 softmax 单元，可以得到对应的代价函数：

$$J(\theta) = \underset{\theta}{\operatorname{argmin}} \ -\mathcal{L}(\theta \mid \boldsymbol{y}, \hat{\boldsymbol{y}}) = -\frac{1}{m}\sum_{i=1}^{m}(\boldsymbol{y}^{(i)} \log \hat{\boldsymbol{y}}^{(i)}) \tag{11}$$

其中对每一个 $\hat{\boldsymbol{y}}$ 有 $\hat{y}_k = \frac{\exp(z_k)}{\sum_j \exp(z_j)}$。因此可以进一步展开为：

$$J(\theta) = -\frac{1}{m}\sum_{i=1}^{m}\sum_{k=0}^{\mathrm{K}-1}(y_k^{(i)} \log \hat{y}_k^{(i)}) \tag{12}$$

梯度的计算：以单个样本为例，假设输出单元 $\hat{\boldsymbol{y}} = g(\boldsymbol{z})$ ，g 为输出单元的激活函数，则此时代价函数为 $J(\theta) = -\sum_{k=0}^{\mathrm{K}-1}(y_k \log \hat{y}_k)$。
首先，对 softmax 函数求导：

1. 当 $j = i$ 时：

$$\begin{aligned}
\frac{\partial \hat{y}_j}{\partial z_i} &= \frac{\partial}{\partial z_i}\left(\frac{e^{z_j}}{\sum_k e^{z_k}}\right) \\
&= \frac{(e^{z_j})' \cdot \sum_k e^{z_k} - e^{z_j} \cdot e^{z_j}}{(\sum_k e^{z_k})^2} \\
&= \frac{e^{z_j}}{\sum_k e^{z_k}} - \frac{e^{z_j}}{\sum_k e^{z_k}} \cdot \frac{e^{z_j}}{\sum_k e^{z_k}} \\
&= \hat{y}_j(1 - \hat{y}_j) \\
&= \hat{y}_j - \hat{y}_j\hat{y}_j
\end{aligned} \tag{13}$$

2. 当 $j \neq i$ 时：

$$\begin{aligned}
\frac{\partial \hat{y}_j}{\partial z_i} &= \frac{\partial}{\partial z_i}\left(\frac{e^{z_j}}{\sum_k e^{z_k}}\right) \\
&= \frac{0 \cdot \sum_k e^{z_k} - e^{z_j} \cdot e^{z_i}}{(\sum_k e^{z_k})^2} \\
&= -\frac{e^{z_j}}{\sum_k e^{z_k}} \cdot \frac{e^{z_i}}{\sum_k e^{z_k}} \\
&= -\hat{y}_j\hat{y}_i \\
&= 0 - \hat{y}_j\hat{y}_i
\end{aligned} \tag{14}$$

然后，进一步计算：

$$\begin{aligned}
\frac{\partial J(\theta)}{\partial z_j} &= -\sum_k y_k \frac{\partial}{\partial z_j}(\log \hat{y}_k) \\
&= -\sum_k y_k \frac{1}{\hat{y}_k}\frac{\partial \hat{y}_k}{\partial z_j}(\text{拆分求导}) \\
&= -y_j \frac{1}{\hat{y}_j} \cdot \hat{y}_j(1 - \hat{y}_j) - \sum_{k \neq j} y_k \frac{1}{\hat{y}_k} \cdot (-\hat{y}_j\hat{y}_k) \\
&= -y_j(1 - \hat{y}_j) + \hat{y}_j \sum_{k \neq j} y_k(\text{合并}) \\
&= -y_j + \hat{y}_j \sum_k y_k \\
&= \hat{y}_j - y_j
\end{aligned} \tag{15}$$

2.4 架构设计

架构 (`Architecture`) 一词指网络的整体结构：它应该具有多少单元，以及这些单元应该如何连接。
在实践中，神经网络具有多样性。
用于计算机视觉的卷积神经网络的特殊架构在第九章中介绍。前馈网络也可以推广到序列处理的循环神经网络，但也有它们自己的架构考虑，将在第十章中介绍。

3 反向传播算法

3.1 单个神经元的训练

单个神经元的结构如图 3 所示，假设训练样本 $(\boldsymbol{x}, y)$，在图 3 左，$\boldsymbol{x}$ 是输入向量，通过一个激励函数 $h_{\boldsymbol{w},b(\boldsymbol{x})}$ 得到一个输出 a，a 再通过代价函数得到 J。激励函数以使用 sigmoid 为例：

$$\begin{aligned}
f(\boldsymbol{W}, b, \boldsymbol{x}) = a &= \text{sigmoid}(\sum_i x_i w_i + b) \\
J(\boldsymbol{W}, b, \boldsymbol{x}, y) &= \frac{1}{2}\|y - a\|^2
\end{aligned} \tag{16}$$

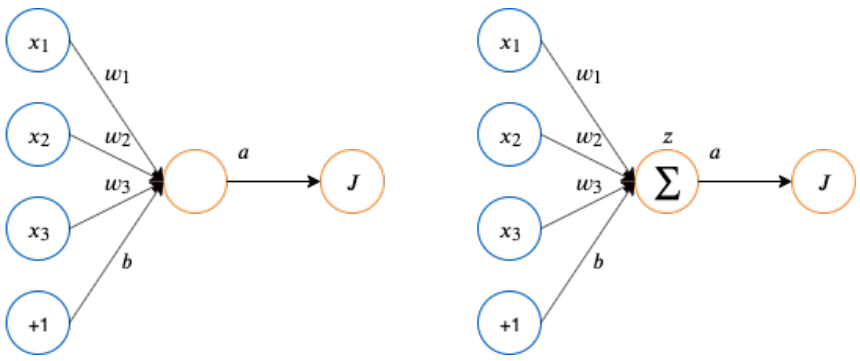


图 3. 单个神经元用于训练的示意图

将激励函数拆成两部分，第一部分通过仿射求和得到 $z=\sum_i x_i w_i + b = \sum_i w_i x_i + b$，第二部分通过 sigmoid 得到 a。

1. 训练过程中，要求代价函数 J 关于 $\boldsymbol{w}$ 和 b 的偏导数。先求 J 关于中间变量 a 和 z 的偏导：

$$\begin{aligned}\delta^{(a)} &= \frac{\partial}{\partial a} J(\boldsymbol{w}, b, \boldsymbol{x}, y) = -(y-a) \\ \delta^{(z)} &= \frac{\partial}{\partial z} J(\boldsymbol{w}, b, \boldsymbol{x}, y) = \frac{\partial J}{\partial a}\frac{\partial a}{\partial z} = \delta^{(a)} a(1-a)\end{aligned} \tag{17}$$

其中 sigmoid 的导数为 $a(1-a)$；

2. 再根据链导法则，可以求得 J 关于 $\boldsymbol{w}$ 和 b 的偏导数，即得 $\boldsymbol{w}$ 和 b 的梯度。

$$\begin{aligned}\nabla_{\boldsymbol{w}} J(\boldsymbol{w}, b, \boldsymbol{x}, y) &= \frac{\partial}{\partial w} J = \frac{\partial J}{\partial z}\frac{\partial z}{\partial \boldsymbol{w}} = \boldsymbol{x}\delta^{(z)} \\ \nabla_b J(\boldsymbol{w}, b, \boldsymbol{x}, y) &= \frac{\partial}{\partial b} J = \frac{\partial J}{\partial z}\frac{\partial z}{\partial b} = \delta^{(z)}\end{aligned} \tag{18}$$

在这个过程中，先求 $\frac{\partial J}{\partial a}$，进一步求 $\frac{\partial J}{\partial z}$，最后求得 $\frac{\partial J}{\partial \boldsymbol{w}}$ 和 $\frac{\partial J}{\partial b}$。结合上图及链导法则，可以看出这是一个将代价函数的增量 ∂J 自后向前传播的过程，因此称为反向传播 (`Back Propagation`)。

3.2 多层神经网络的训练

在描述多层神经网络时，我们不再仔细描述某一层内部节点，而是描述层与层之间的关系。如图 4，为方便标记，在描述多层网络时，我们将第 i 层记作下标 $_i$。在本书中，第 i 层用 $\boldsymbol{a}_l$ 和 a^l 两种方式表示，视情况而用。假设第 $l+1$ 层的输入和输出分别是 $\boldsymbol{a}_l$ 和 $\boldsymbol{a}_{l+1}$，参数为 $\boldsymbol{W}_l$ 和 b_l，其中 $\boldsymbol{W}_l$ 的维数为 (n_{in}, n_{out})。仿射结果为中间变量 $\boldsymbol{z}_{l+1}$。其中第一层的输出 $\boldsymbol{a}_1 = \boldsymbol{x}$，为整个网络的输入，最后一层的输出 $\boldsymbol{a}_L$ 是代价函数的输入。

$$\begin{aligned}\boldsymbol{z}_{l+1} &= \boldsymbol{W}_l^\top \boldsymbol{a}_l + \boldsymbol{b}_l \\ \boldsymbol{a}_{l+1} &= \text{sigmoid}(\boldsymbol{z}_{l+1})\end{aligned} \tag{19}$$

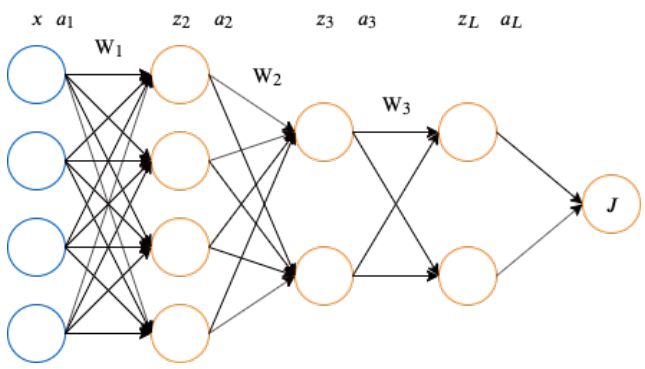


图 4. 多层神经网络训练的示意图

在实际神经网络中，网络层的输入 $\boldsymbol{a}_l$ 的维数通常表示为 (n_{samp}, n_{in})，其中第一维 n_{samp} 为输入样本的数量。于是，计算公式可以写作：

$$\boldsymbol{z}_{l+1} = \boldsymbol{a}_l \boldsymbol{W}_l + \boldsymbol{b}_l \tag{20}$$

对多层网络的训练需要计算代价函数 J 对每一层的参数求偏导。后向传播过程变为：

1. 求 J 对 $\boldsymbol{a}_L$ 的偏导 $\delta_L^{(a)}$；

2. 在第 $l+1$ 层，将误差信号从 $\boldsymbol{a}_{l+1}$ 传递到 $\boldsymbol{z}_{l+1}$；

$$\frac{\partial \boldsymbol{a}_{l+1}}{\partial \boldsymbol{z}_{l+1}} = \boldsymbol{a}_{l+1}(1-\boldsymbol{a}_{l+1}) \tag{21}$$

3. 将误差信号从第 $l+1$ 层向第 l 层传播；

4. 对第 l 层计算得到 J 对 $\boldsymbol{a}_l$ 和 $\boldsymbol{z}_l$ 的偏导数；

$$\begin{aligned}\delta_l^{(a)} = \frac{\partial J}{\partial \boldsymbol{a}_l} &= \begin{cases} -(\boldsymbol{y}-\boldsymbol{a}_l), & \text{if } l = L \\ \frac{\partial J}{\partial \boldsymbol{z}_{l+1}}\frac{\partial \boldsymbol{z}_{l+1}}{\partial \boldsymbol{a}_l} = \delta_{l+1}^{(z)}(\boldsymbol{W}_l)^\top, & \text{otherwise} \end{cases} \\ \delta_l^{(z)} = \frac{\partial J}{\partial \boldsymbol{z}_l} &= \frac{\partial J}{\partial \boldsymbol{a}_l}\frac{\partial \boldsymbol{a}_l}{\partial \boldsymbol{z}_l} = \delta_l^{(a)} \boldsymbol{a}_{l+1}(1-\boldsymbol{a}_{l+1})\end{aligned} \tag{22}$$

5. 对第 l 层计算得到 J 对参数 $\boldsymbol{W}_l$ 和 $\boldsymbol{b}_l$ 的梯度。

$$\begin{aligned}\nabla_{\boldsymbol{W}_l} J(\boldsymbol{W}, \boldsymbol{b}, \boldsymbol{x}, \boldsymbol{y}) &= \frac{\partial}{\partial \boldsymbol{W}_l} J = \frac{\partial J}{\partial \boldsymbol{z}_{l+1}}\frac{\partial \boldsymbol{z}_{l+1}}{\partial \boldsymbol{W}_l} = (\boldsymbol{a}_l)^\top \delta_{l+1}^{(z)} \\ \nabla_{\boldsymbol{b}_l} J(\boldsymbol{W}, \boldsymbol{b}, \boldsymbol{x}, \boldsymbol{y}) &= \frac{\partial}{\partial \boldsymbol{b}_l} J = \frac{\partial J}{\partial \boldsymbol{z}_{l+1}}\frac{\partial \boldsymbol{z}_{l+1}}{\partial \boldsymbol{b}_l} = \delta_{l+1}^{(z)}\end{aligned} \tag{23}$$

为具体实现深度前馈网络，需要首先定义以下模块：

3.2.1 定义权重初始化方法

[4]:
```
class std_normal:
    """
    标准正态初始化
    """
    def __init__(self, gain=0.01):
        self.gain = gain

    def __call__(self, weight_shape):
        return self.gain * np.random.randn(*weight_shape)

class he_uniform:
    """
    He 初始化， 通过 Uniform(-b, b) 初始化权重矩阵 W， 这里的 b=sqrt(6 / n_in)
    （更多实现会在第 8 章展开）
    """
    def __init__(self):
        pass

    def __call__(self, weight_shape):
        n_in, n_out = weight_shape
        b = np.sqrt(6 / n_in)
        return np.random.uniform(-b, b, size=weight_shape)

class WeightInitializer(object):

    def __init__(self, mode="he_uniform"):
        self.mode = mode  # 更多初始化方法在第 8 章展开
        r = r"([a-zA-Z]*)=([^,)]*)"
        mode_str = self.mode.lower()
        kwargs = dict([(i, eval(j)) for (i, j) in re.findall(r, mode_str)])
        if "std_normal" in mode_str:
            self.init_fn = std_normal(**kwargs)
        elif "he_uniform" in mode_str:
            self.init_fn = he_uniform(**kwargs)

    def __call__(self, weight_shape):
        W = self.init_fn(weight_shape)
        return W
```

3.2.2 定义激活函数 (见前文描述)

3.2.3 定义优化方法

[5]:
```
"""
这里采用 sgd 优化方法，更多实现细节和优化方法见第 8 章
"""
class OptimizerBase(ABC):

    def __init__(self):
        pass

    def __call__(self, params, params_grad, params_name):
        """
        参数说明:
        params: 待更新参数， 如权重矩阵 W;
        params_grad: 待更新参数的梯度;
        params_name: 待更新参数名;
```

```
        """
        return self.update(params, params_grad, params_name)

    @abstractmethod
    def update(self, params, params_grad, params_name):
        raise NotImplementedError

class SGD(OptimizerBase):
    """
    sgd 优化方法
    """
    def __init__(self, lr=0.01):
        super().__init__()
        self.lr = lr

    def __str__(self):
        return "SGD(lr={})".format(self.hyperparams["lr"])

    def update(self, params, params_grad, params_name):
        update_value = self.lr * params_grad
        return params - update_value

    @property
    def hyperparams(self):
        return {
            "op": "SGD",
            "lr": self.lr
        }

class OptimizerInitializer(ABC):

    def __init__(self, opti_name="sgd"):
        self.opti_name = opti_name

    def __call__(self):
        r = r"([a-zA-Z]*)=([^,)]*)"
        opti_str = self.opti_name.lower()
        kwargs = dict([(i, eval(j)) for (i, j) in re.findall(r, opti_str)])
        if "sgd" in opti_str:
            optimizer = SGD(**kwargs)
        return optimizer
```

3.2.4 定义网络层的框架

[6]:
```
class LayerBase(ABC):

    def __init__(self, optimizer=None):
        self.X = []   # 网络层输入
        self.gradients = {}  # 网络层待梯度更新变量
        self.params = {}  # 网络层参数变量
        self.acti_fn = None   # 网络层激活函数
        self.optimizer = OptimizerInitializer(optimizer)()  # 网络层优化方法

    @abstractmethod
    def _init_params(self, **kwargs):
        """
        函数作用：初始化参数
        """
        raise NotImplementedError
```

```
    @abstractmethod
    def forward(self, X, **kwargs):
        """
        函数作用：前向传播
        """
        raise NotImplementedError

    @abstractmethod
    def backward(self, out, **kwargs):
        """
        函数作用：反向传播
        """
        raise NotImplementedError

    def flush_gradients(self):
        """
        函数作用：重置更新参数列表
        """
        self.X = []
        for k, v in self.gradients.items():
            self.gradients[k] = np.zeros_like(v)

    def update(self):
        """
        函数作用：更新参数
        """
        for k, v in self.gradients.items():
            if k in self.params:
                self.params[k] = self.optimizer(self.params[k], v, k)
```

[7]:
```
class FullyConnected(LayerBase):
    """
    定义全连接层，实现 a=g(x*W+b)，前向传播输入 x，返回 a；反向传播输入
    """
    def __init__(self, n_out, acti_fn, init_w, optimizer=None):
        """
        参数说明：
        acti_fn：激活函数， str 型
        init_w：权重初始化方法， str 型
        n_out：隐藏层输出维数
        optimizer：优化方法
        """
        super().__init__(optimizer)
        self.n_in = None  # 隐藏层输入维数， int 型
        self.n_out = n_out  # 隐藏层输出维数， int 型
        self.acti_fn = ActivationInitializer(acti_fn)()
        self.init_w = init_w
        self.init_weights = WeightInitializer(mode=init_w)
        self.is_initialized = False  # 是否初始化， bool 型变量

    def _init_params(self):
        b = np.zeros((1, self.n_out))
        W = self.init_weights((self.n_in, self.n_out))
        self.params = {"W": W, "b": b}
        self.gradients = {"W": np.zeros_like(W), "b": np.zeros_like(b)}
        self.is_initialized = True

    def forward(self, X, retain_derived=True):
```

```
        """
        全连接网络的前向传播，原理见上文 反向传播算法 部分。

        参数说明：
        X：输入数组，为（n_samples, n_in），float 型
        retain_derived：是否保留中间变量，以便反向传播时再次使用，bool 型
        """
        if not self.is_initialized:  # 如果参数未初始化，先初始化参数
            self.n_in = X.shape[1]
            self._init_params()
        W = self.params["W"]
        b = self.params["b"]
        z = X @ W + b
        a = self.acti_fn.forward(z)
        if retain_derived:
            self.X.append(X)
        return a

    def backward(self, dLda, retain_grads=True):
        """
        全连接网络的反向传播，原理见上文 反向传播算法 部分。

        参数说明：
        dLda：关于损失的梯度，为（n_samples, n_out），float 型
        retain_grads：是否计算中间变量的参数梯度，bool 型
        """
        if not isinstance(dLda, list):
            dLda = [dLda]
        dX = []
        X = self.X
        for da, x in zip(dLda, X):
            dx, dw, db = self._bwd(da, x)
            dX.append(dx)
            if retain_grads:
                self.gradients["W"] += dw
                self.gradients["b"] += db
        return dX[0] if len(X) == 1 else dX

    def _bwd(self, dLda, X):
        W = self.params["W"]
        b = self.params["b"]
        Z = X @ W + b
        dZ = dLda * self.acti_fn.grad(Z)
        dX = dZ @ W.T
        dW = X.T @ dZ
        db = dZ.sum(axis=0, keepdims=True)
        return dX, dW, db

    @property
    def hyperparams(self):
        return {
            "layer": "FullyConnected",
            "init_w": self.init_w,
            "n_in": self.n_in,
            "n_out": self.n_out,
            "acti_fn": str(self.acti_fn),
            "optimizer": {
                "hyperparams": self.optimizer.hyperparams,
            },
```

```
            "components": {
                k: v for k, v in self.params.items()
            }
        }
```

[8]:
```
class Softmax(LayerBase):
    """
    定义 Softmax 层
    """
    def __init__(self, dim=-1, optimizer=None):
        super().__init__(optimizer)
        self.dim = dim
        self.n_in = None
        self.is_initialized = False

    def _init_params(self):
        self.params = {}
        self.gradients = {}
        self.is_initialized = True

def forward(self, X, retain_derived=True):
    """
    Softmax 的前向传播， 原理见上文 代价函数 部分。
    """
    if not self.is_initialized:
        self.n_in = X.shape[1]
        self._init_params()
    Y = self._fwd(X)
    if retain_derived:
        self.X.append(X)
    return Y

def _fwd(self, X):
    e_X = np.exp(X - np.max(X, axis=self.dim, keepdims=True))
    return e_X / e_X.sum(axis=self.dim, keepdims=True)

def backward(self, dLdy):
    """
    Softmax 的反向传播，原理见上文 代价函数 部分。
    """
    if not isinstance(dLdy, list):
        dLdy = [dLdy]
    dX = []
    X = self.X
    for dy, x in zip(dLdy, X):
        dx = self._bwd(dy, x)
        dX.append(dx)
    return dX[0] if len(X) == 1 else dX

def _bwd(self, dLdy, X):
    dX = []
    for dy, x in zip(dLdy, X):
        dxi = []
        for dyi, xi in zip(*np.atleast_2d(dy, x)):
            yi = self._fwd(xi.reshape(1, -1)).reshape(-1, 1)
            dyidxi = np.diagflat(yi) - yi @ yi.T
            dxi.append(dyi @ dyidxi)
        dX.append(dxi)
    return np.array(dX).reshape(*X.shape)
```

```
    @property
    def hyperparams(self):
        return {
            "layer": "SoftmaxLayer",
            "n_in": self.n_in,
            "n_out": self.n_in,
            "optimizer": {
                "hyperparams": self.optimizer.hyperparams,
            },
        }
```

3.2.5 定义代价函数

[9]:
```
class ObjectiveBase(ABC):

    def __init__(self):
        super().__init__()

    @abstractmethod
    def loss(self, y_true, y_pred):
        """
        函数作用：计算损失
        """
        raise NotImplementedError

    @abstractmethod
    def grad(self, y_true, y_pred, **kwargs):
        """
        函数作用：计算代价函数的梯度
        """
        raise NotImplementedError


class SquaredError(ObjectiveBase):
    """
    二次代价函数。
    """
    def __init__(self):
        super().__init__()

    def __call__(self, y_true, y_pred):
        return self.loss(y_true, y_pred)

    def __str__(self):
        return "SquaredError"

    @staticmethod
    def loss(y_true, y_pred):
        """
        参数说明：
        y_true：训练的 n 个样本的真实值， 形状为 (n,m) 数组；
        y_pred：训练的 n 个样本的预测值， 形状为 (n,m) 数组；
        """
        (n, _) = y_true.shape
        return 0.5 * np.linalg.norm(y_pred - y_true) ** 2 / n

    @staticmethod
    def grad(y_true, y_pred, z, acti_fn):
```

```
        (n, _) = y_true.shape
        return (y_pred - y_true) * acti_fn.grad(z) / n


class CrossEntropy(ObjectiveBase):
    """
    交叉熵代价函数。
    """
    def __init__(self):
        super().__init__()

    def __call__(self, y_true, y_pred):
        return self.loss(y_true, y_pred)

    def __str__(self):
        return "CrossEntropy"

    @staticmethod
    def loss(y_true, y_pred):
        """
        参数说明:
        y_true: 训练的 n 个样本的真实值， 要求形状为 (n,m) 二进制（每个样本均为 one-hot 编码);
        y_pred: 训练的 n 个样本的预测值， 形状为 (n,m);
        """
        (n, _) = y_true.shape
        eps = np.finfo(float).eps  # 防止 np.log(0)
        cross_entropy = -np.sum(y_true * np.log(y_pred + eps)) / n
        return cross_entropy

    @staticmethod
    def grad(y_true, y_pred):
        (n, _) = y_true.shape
        grad = (y_pred - y_true) / n
        return grad
```

3.2.6 定义深度前馈网络

[10]:
```
def minibatch(X, batchsize=256, shuffle=True):
    """
    函数作用：将数据集分割成 batch， 基于 mini batch 训练，具体可见第 8 章。
    """
    N = X.shape[0]
    idx = np.arange(N)
    n_batches = int(np.ceil(N / batchsize))
    if shuffle:
        np.random.shuffle(idx)
    def mb_generator():
        for i in range(n_batches):
            yield idx[i * batchsize : (i + 1) * batchsize]
    return mb_generator(), n_batches
```

[11]:
```
class DFN(object):

    def __init__(
        self,
        hidden_dims_1=None,
        hidden_dims_2=None,
        optimizer="sgd(lr=0.01)",
        init_w="std_normal",
        loss=CrossEntropy()
```

```
    ):
        self.optimizer = optimizer
        self.init_w = init_w
        self.loss = loss
        self.hidden_dims_1 = hidden_dims_1
        self.hidden_dims_2 = hidden_dims_2
        self.is_initialized = False

    def _set_params(self):
        """
        函数作用：模型初始化
        FC1 -> Sigmoid -> FC2 -> Softmax
        """
        self.layers = OrderedDict()
        self.layers["FC1"] = FullyConnected(
            n_out=self.hidden_dims_1,
            acti_fn="sigmoid",
            init_w=self.init_w,
            optimizer=self.optimizer
        )
        self.layers["FC2"] = FullyConnected(
            n_out=self.hidden_dims_2,
            acti_fn="affine(slope=1, intercept=0)",
            init_w=self.init_w,
            optimizer=self.optimizer
        )
        self.is_initialized = True

    def forward(self, X_train):
        Xs = {}
        out = X_train
        for k, v in self.layers.items():
            Xs[k] = out
            out = v.forward(out)
        return out, Xs

    def backward(self, grad):
        dXs = {}
        out = grad
        for k, v in reversed(list(self.layers.items())):
            dXs[k] = out
            out = v.backward(out)
        return out, dXs

    def update(self):
        """
        函数作用：梯度更新
        """
        for k, v in reversed(list(self.layers.items())):
            v.update()
        self.flush_gradients()

    def flush_gradients(self, curr_loss=None):
        """
        函数作用：更新后重置梯度
        """
        for k, v in self.layers.items():
            v.flush_gradients()
```

```
    def fit(self, X_train, y_train, n_epochs=20, batch_size=64, verbose=False):
        """
        参数说明:
        X_train: 训练数据
        y_train: 训练数据标签
        n_epochs: epoch 次数
        batch_size: 每次 epoch 的 batch size
        verbose: 是否每个 batch 输出损失
        """
        self.verbose = verbose
        self.n_epochs = n_epochs
        self.batch_size = batch_size
        if not self.is_initialized:
            self.n_features = X_train.shape[1]
            self._set_params()
        prev_loss = np.inf
        for i in range(n_epochs):
            loss, epoch_start = 0.0, time.time()
            batch_generator, n_batch = minibatch(X_train, self.batch_size, shuffle=True)
            for j, batch_idx in enumerate(batch_generator):
                batch_len, batch_start = len(batch_idx), time.time()
                X_batch, y_batch = X_train[batch_idx], y_train[batch_idx]
                out, _ = self.forward(X_batch)
                y_pred_batch = softmax(out)
                batch_loss = self.loss(y_batch, y_pred_batch)
                grad = self.loss.grad(y_batch, y_pred_batch)
                _, _ = self.backward(grad)
                self.update()
                loss += batch_loss
                if self.verbose:
                    fstr = "\t[Batch {}/{}] Train loss: {:.3f} ({:.1f}s/batch)"
                    print(fstr.format(j + 1, n_batch, batch_loss, time.time() - batch_start))
            loss /= n_batch
            fstr = "[Epoch {}] Avg. loss: {:.3f}  Delta: {:.3f} ({:.2f}m/epoch)"
            print(fstr.format(i + 1, loss, prev_loss - loss, (time.time() - epoch_start) / 60.0))
            prev_loss = loss

    def evaluate(self, X_test, y_test, batch_size=128):
        acc = 0.0
        batch_generator, n_batch = minibatch(X_test, batch_size, shuffle=True)
        for j, batch_idx in enumerate(batch_generator):
            batch_len, batch_start = len(batch_idx), time.time()
            X_batch, y_batch = X_test[batch_idx], y_test[batch_idx]
            y_pred_batch, _ = self.forward(X_batch)
            y_pred_batch = np.argmax(y_pred_batch, axis=1)
            y_batch = np.argmax(y_batch, axis=1)
            acc += np.sum(y_pred_batch == y_batch)
        return acc / X_test.shape[0]

    @property
    def hyperparams(self):
        return {
            "init_w": self.init_w,
            "loss": str(self.loss),
            "optimizer": self.optimizer,
            "hidden_dims_1": self.hidden_dims_1,
            "hidden_dims_2": self.hidden_dims_2,
            "components": {k: v.params for k, v in self.layers.items()}
        }
```

用自定义 DFN 实现，测试 MNIST 数据集

[12]:
```
def load_data(path="../data/mnist/mnist.npz"):
    f = np.load(path)
    X_train, y_train = f['x_train'], f['y_train']
    X_test, y_test = f['x_test'], f['y_test']
    f.close()
    return (X_train, y_train), (X_test, y_test)

(X_train, y_train), (X_test, y_test) = load_data()
y_train = np.eye(10)[y_train.astype(int)]
y_test = np.eye(10)[y_test.astype(int)]
X_train = X_train.reshape(-1, X_train.shape[1]*X_train.shape[2]).astype('float32')
X_test = X_test.reshape(-1, X_test.shape[1]*X_test.shape[2]).astype('float32')
print(X_train.shape, y_train.shape)
N = 20000   # 取 20000 条数据用以训练
indices = np.random.permutation(range(X_train.shape[0]))[:N]
X_train, y_train = X_train[indices], y_train[indices]
print(X_train.shape, y_train.shape)
X_train /= 255
X_train = (X_train - 0.5) * 2
X_test /= 255
X_test = (X_test - 0.5) * 2
```

```
(60000, 784) (60000, 10)
(20000, 784) (20000, 10)
```

[13]:
```
model = DFN(hidden_dims_1=200, hidden_dims_2=10)
```

[14]:
```
model.fit(X_train, y_train, n_epochs=20)
```

```
[Epoch 1] Avg. loss: 2.283  Delta: inf (0.02m/epoch)
[Epoch 2] Avg. loss: 2.204  Delta: 0.078 (0.02m/epoch)
[Epoch 3] Avg. loss: 1.986  Delta: 0.219 (0.02m/epoch)
[Epoch 4] Avg. loss: 1.628  Delta: 0.357 (0.02m/epoch)
[Epoch 5] Avg. loss: 1.292  Delta: 0.336 (0.02m/epoch)
[Epoch 6] Avg. loss: 1.051  Delta: 0.242 (0.02m/epoch)
[Epoch 7] Avg. loss: 0.886  Delta: 0.165 (0.02m/epoch)
[Epoch 8] Avg. loss: 0.772  Delta: 0.114 (0.02m/epoch)
[Epoch 9] Avg. loss: 0.691  Delta: 0.081 (0.02m/epoch)
[Epoch 10] Avg. loss: 0.631  Delta: 0.060 (0.02m/epoch)
[Epoch 11] Avg. loss: 0.585  Delta: 0.046 (0.02m/epoch)
[Epoch 12] Avg. loss: 0.548  Delta: 0.037 (0.02m/epoch)
[Epoch 13] Avg. loss: 0.519  Delta: 0.029 (0.02m/epoch)
[Epoch 14] Avg. loss: 0.494  Delta: 0.025 (0.02m/epoch)
[Epoch 15] Avg. loss: 0.474  Delta: 0.020 (0.02m/epoch)
[Epoch 16] Avg. loss: 0.456  Delta: 0.018 (0.02m/epoch)
[Epoch 17] Avg. loss: 0.441  Delta: 0.015 (0.02m/epoch)
[Epoch 18] Avg. loss: 0.428  Delta: 0.013 (0.02m/epoch)
[Epoch 19] Avg. loss: 0.417  Delta: 0.011 (0.02m/epoch)
[Epoch 20] Avg. loss: 0.406  Delta: 0.011 (0.02m/epoch)
```

[15]:
```
print(model.evaluate(X_test, y_test))
```

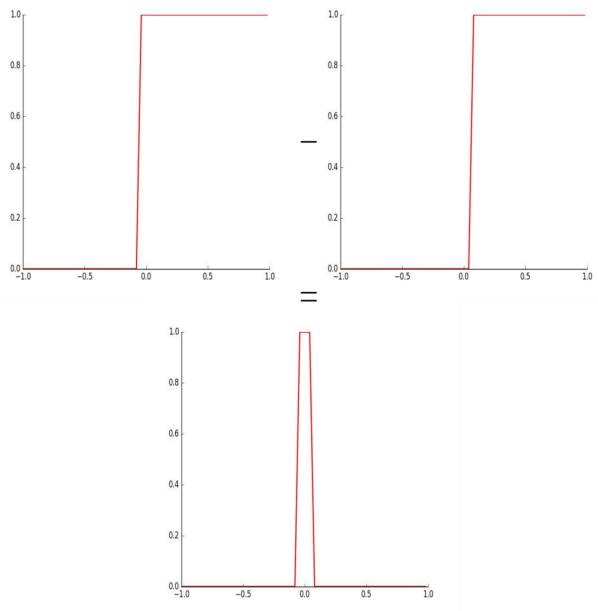
```
0.894
```

4 神经网络的万能近似定理

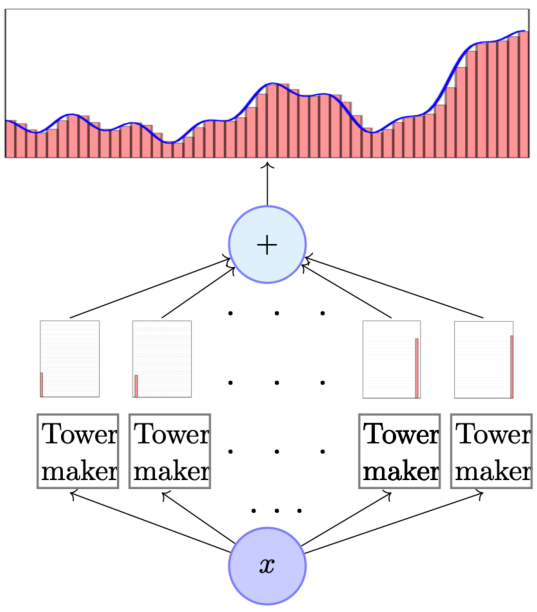
万能近似定理：一个前馈神经网络如果具有线性层和至少一层具有“挤压”性质的激活函数（如 sigmoid 等），给定网络足够数量的隐藏单元，它可以以任意精度来近似任何从一个有限维空间到另一个有限维空间的 borel 可测函数。

sigmoid 函数的万能近似可视化证明 (以一元函数为例)：

1. 我们可以通过两个 sigmoid 函数 ($\boldsymbol{y} = \text{sigmoid}(\boldsymbol{w}^\top \boldsymbol{x} + \boldsymbol{b})$) 生成一个 *tower*，如图：



2. 我们构造多个这样的 *tower* 近似任意函数：



可视化证明可参考：http://neuralnetworksanddeeplearning.com/chap4.html

或者 https://www.cse.iitm.ac.in/~miteshk/CS7015/Slides/Teaching/pdf/Lecture3.pdf

5 实例：学习 XOR

我们的网络是 $f(\boldsymbol{x};\boldsymbol{W},c,\boldsymbol{w},b)=\boldsymbol{w}^{\top}\max\{\boldsymbol{0},\boldsymbol{W}^{\top}\boldsymbol{x}+\boldsymbol{c}\}+b$

[16]:
```python
class XOR(object):


    def __init__(
        self,
        hidden_dims_1=None,
        hidden_dims_2=None,
        optimizer="sgd(lr=1.0)",
        init_w="std_normal(gain=1.0)",
        loss=SquaredError()
    ):
        self.optimizer = optimizer
        self.init_w = init_w
        self.loss = loss
        self.hidden_dims_1 = hidden_dims_1
        self.hidden_dims_2 = hidden_dims_2
        self.is_initialized = False

    def _set_params(self):
        """
        函数作用: 模型初始化
        FC1 -> Relu -> FC2
        """
        self.layers = OrderedDict()
```

```
        self.layers["FC1"] = FullyConnected(
            n_out=self.hidden_dims_1,
            acti_fn="relu",
            init_w=self.init_w,
            optimizer=self.optimizer
        )
        self.layers["FC2"] = FullyConnected(
            n_out=self.hidden_dims_2,
            acti_fn="affine(slope=1, intercept=0)",
            init_w=self.init_w,
            optimizer=self.optimizer
        )
        self.is_initialized = True

    def forward(self, X_train):
        Xs = {}
        out = X_train
        for k, v in self.layers.items():
            Xs[k] = out
            out = v.forward(out)
        return out, Xs

    def backward(self, grad):
        dXs = {}
        out = grad
        for k, v in reversed(list(self.layers.items())):
            dXs[k] = out
            out = v.backward(out)
        return out, dXs

    def update(self):
        """
        函数作用：梯度更新
        """
        for k, v in reversed(list(self.layers.items())):
            v.update()
        self.flush_gradients()

    def flush_gradients(self, curr_loss=None):
        """
        函数作用：更新后重置梯度
        """
        for k, v in self.layers.items():
            v.flush_gradients()

    def fit(self, X_train, y_train, n_epochs=20001, batch_size=4):
        """
        参数说明:
        X_train: 训练数据
        y_train: 训练数据标签
        n_epochs: epoch 次数
        batch_size: 每次 epoch 的 batch size
        """
        self.n_epochs = n_epochs
        if not self.is_initialized:
            self.n_features = X_train.shape[1]
            self._set_params()
        prev_loss = np.inf
        for i in range(n_epochs):
```

```
            loss, epoch_start = 0.0, time.time()
            out, _ = self.forward(X_train)
            anti_fn = Affine()
            y_pred = anti_fn.forward(out)
            loss = self.loss(y_train, y_pred)
            grad = self.loss.grad(y_train, y_pred, out, anti_fn)
            _, _ = self.backward(grad)
            self.update()
            if not i%5000:
                fstr = "[Epoch {}] Avg. loss: {:.3f}  Delta: {:.3f} ({:.2f}m/epoch)"
                print(fstr.format(i + 1, loss, prev_loss - loss, (time.time() - epoch_start) / 60.0))
                prev_loss = loss

    @property
    def hyperparams(self):
        return {
            "init_w": self.init_w,
            "loss": str(self.loss),
            "optimizer": self.optimizer,
            "hidden_dims_1": self.hidden_dims_1,
            "hidden_dims_2": self.hidden_dims_2,
            "components": {k: v.params for k, v in self.layers.items()}
        }
```

[17]:
```
X_train = np.array([[0.0, 0.0], [0.0, 1.0], [1.0, 0.0], [1.0, 1.0]])
y_train = np.array([[0.0], [1.0], [1.0], [0.0]])
```

[20]:
```
model = XOR(hidden_dims_1=2, hidden_dims_2=1)
```

[21]:
```
model.fit(X_train, y_train)
```

```
[Epoch 1] Avg. loss: 0.086  Delta: inf (0.00m/epoch)
[Epoch 5001] Avg. loss: 0.000  Delta: 0.086 (0.00m/epoch)
[Epoch 10001] Avg. loss: 0.000  Delta: 0.000 (0.00m/epoch)
[Epoch 15001] Avg. loss: 0.000  Delta: 0.000 (0.00m/epoch)
[Epoch 20001] Avg. loss: 0.000  Delta: 0.000 (0.00m/epoch)
```

[22]:
```
print(model.hyperparams)
```

```
{'init_w': 'std_normal(gain=1.0)', 'loss': 'SquaredError', 'optimizer':
'sgd(lr=1.0)', 'hidden_dims_1': 2, 'hidden_dims_2': 1, 'components': {'FC1':
{'W': array([[-1.35300239,  0.95949967],
       [ 0.88153178, -0.95209678]]), 'b': array([[ 3.76300323e-17,
-7.40288739e-03]])}, 'FC2': {'W': array([[1.13438905],
       [1.0503134 ]]), 'b': array([[-1.1389201e-17]])}}}
```

[23]:
```
import numpy
import re

print("numpy:", numpy.__version__)
print("re:", re.__version__)
```

```
numpy: 1.14.5
re: 2.2.1
```