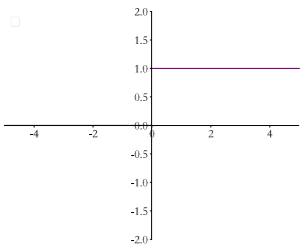
激活函数

# 阶跃函数

阶跃函数(Step function)的函数图像如下图所示，阶跃函数的输出只有 0/1 两种数值，当𝑧 < 0时输出 0，代表类别0；当𝑧 ≥ 0时输出 1，代表类别 1，即：





阶跃函数在𝑧 = 0处是不连续的，其他位置导数为0，无法利用梯度下降算法进行参数优化。

# 符号函数

符合函数的表达式如下所示：





符号函数在𝑧 = 0处也是不连续的，其他位置导数为0，无法利用梯度下降算法进行参数优化。

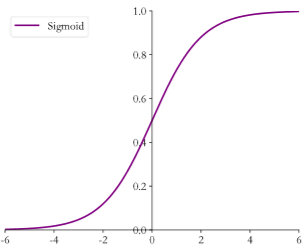
# Sigmoid

Sigmoid 函数也叫 Logistic 函数，定义为

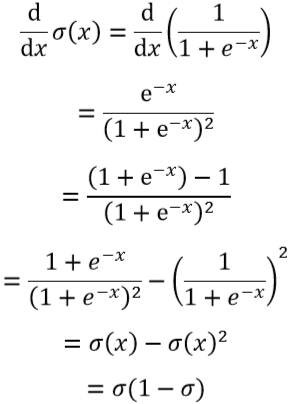
它的一个优良特性就是能够把𝑥 ∈ 𝑅的输入“压缩”到𝑥 ∈ (0,1)区间，这个区间的数值在机 器学习常用来表示以下意义：

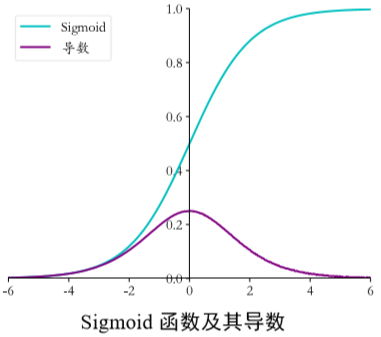
* 概率分布：(0,1)区间的输出和概率的分布范围[0,1]契合，可以通过Sigmoid函数将输出转译为概率输出；
* 信号强度：一般可以将0~1理解为某种信号的强度，如像素的颜色强度，1代表当前通 道颜色最强，0代表当前通道无颜色；抑或代表门控值(Gate)的强度，1代表当前门控全部开放，0代表门控关闭。

Sigmoid函数连续可导，如下图所示，可以直接利用梯度下降算法优化网络参数，应用的非常广泛。



Sigmoid函数的导数表达式推导如下：





在TensorFlow中，可以通过 tf.nn.sigmoid 实现Sigmoid函数，代码如下：

|  |
| --- |
| def set\_plt\_ax():  # get current axis 获得坐标轴对象  ax = plt.gca()  ax.spines['right'].set\_color('none')  # 将右边 上边的两条边颜色设置为空 其实就相当于抹掉这两条边  ax.spines['top'].set\_color('none')  ax.xaxis.set\_ticks\_position('bottom')  # 指定下边的边作为 x 轴，指定左边的边为 y 轴  ax.yaxis.set\_ticks\_position('left')  # 指定 data 设置的bottom(也就是指定的x轴)绑定到y轴的0这个点上  ax.spines['bottom'].set\_position(('data', 0))  ax.spines['left'].set\_position(('data', 0))  x = tf.linspace(-6., 6., 10)  # 通过 Sigmoid 函数  sigmoid\_y = tf.nn.sigmoid(x)  set\_plt\_ax()  plt.plot(x, sigmoid\_y, color='C4', label='Sigmoid')  plt.xlim(-6, 6)  plt.ylim(0, 1)  plt.legend(loc=2)  plt.show() |

通过 Numpy 实现 Sigmoid 函数的导数，代码如下：

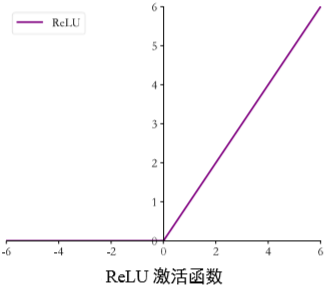
|  |
| --- |
| import numpy as np # 导入 numpy 库  def sigmoid(x): # 实现 sigmoid 函数  return 1 / (1 + np.exp(-x))    def derivative(x): # sigmoid 导数的计算  # sigmoid 函数的表达式由手动推导而得  return sigmoid(x)\*(1-sigmoid(x)) |

# ReLU

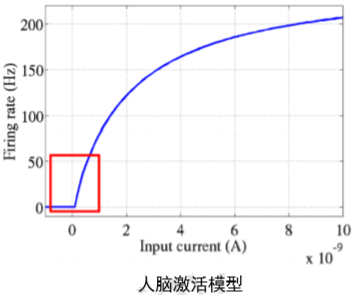
在ReLU(REctified Linear Unit，修正线性单元)激活函数提出之前，Sigmoid函数通常是神经网络的激活函数首选。但是Sigmoid函数在输入值较大或较小时容易出现梯度值接近于0 的现象，称为梯度弥散现象。出现梯度弥散现象时，网络参数长时间得不到更新，导致训练不收敛或停滞不动的现象发生，较深层次的网络模型中更容易出现梯度弥散现象。2012年提出的8层AlexNet 模型采用了一种名叫ReLU的激活函数，使得网络层数达到了8层，自此ReLU函数应用的越来越广泛。ReLU 函数定义为：

ReLU(𝑥) ≜ max(0,𝑥)

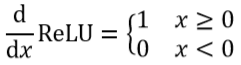
ReLU函数曲线如下图所示。



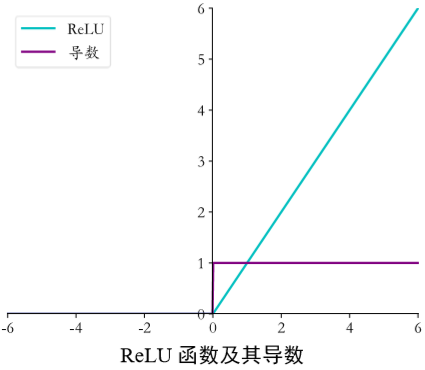
可以看到，ReLU对小于0的值全部抑制为0；对于正数则直接输出，这种单边抑制特性来源于生物学。2001年，神经科学家Dayan和Abott模拟得出更加精确的脑神经元激活模型，如下图所示，它具有单侧抑制、相对宽松的兴奋边界等特 性，ReLU 函数的设计与之非常类似



ReLU函数的导数如下：



可以看到，ReLU 函数的导数计算简单，x 大于等于零的时候，导数值恒为 1，在反向传播 过程中，它既不会放大梯度，造成梯度爆炸(Gradient exploding)现象；也不会缩小梯度，造成梯度弥散(Gradient vanishing)现象。



|  |
| --- |
| tf.nn.relu(x) # 采用tensorflow实现relu函数 |

除了可以使用函数式接口tf.nn.relu实现ReLU函数外，还可以像Dense层一样将ReLU函数作为一个网络层添加到网络中，对应的类为layers.ReLU()类。一般来说，激活函数类并不是主要的网络运算层，不计入网络的层数。 ReLU函数的设计源自神经科学，函数值和导数值的计算均十分简单，同时有着优良 的梯度特性，在大量的深度学习应用中被验证非常有效，是应用最广泛的激活函数之一。

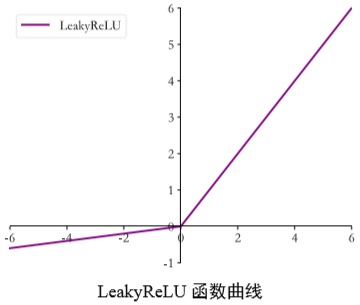
在 ReLU 函数被广泛应用之前，神经网络中激活函数采用 Sigmoid 居多，但是 Sigmoid 函数容易出现梯度弥散现象，当网络的层数增加后，较前层的参数由于梯度值非常微小， 参数长时间得不到有效更新，无法训练较深层的神经网络，导致神经网络的研究一直停留 在浅层。随着 ReLU 函数的提出，很好地缓解了梯度弥散的现象，神经网络的层数能够地 达到较深层数。

通过 Numpy，我们可以方便地实现 ReLU函数导数的代码如下：

|  |
| --- |
| def derivative(x): # ReLU 函数的导数  d = np.array(x, copy=True) # 用于保存梯度的张量  d[x < 0] = 0 # 元素为负的导数为 0  d[x >= 0] = 1 # 元素为正的导数为 1  return d |

# LeakyReLU

ReLU 函数在𝑥 < 0时导数值恒为 0，也可能会造成梯度弥散现象，为了克服这个问题，LeakyReLU 函数被提出，其函数图像如下图所示：

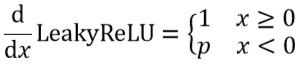


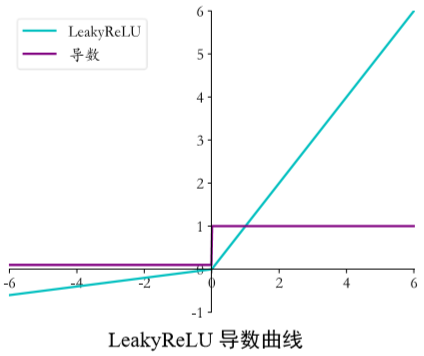
LeakyReLU 的表达式为:



其中𝑝为用户自行设置的某较小数值的超参数，如 0.02 等。当𝑝 = 0时，LeayReLU 函数退化为 ReLU 函数；当𝑝 ≠ 0时，𝑥 < 0处能够获得较小的导数值𝑝，从而避免出现梯度弥散现象。

LeakyReLU函数的导数为：





在 TensorFlow 中，可以通过 tf.nn.leaky\_relu 实现 LeakyReLU 函数，代码如下：

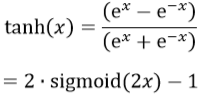
|  |
| --- |
| # 其中 alpha 参数代表𝑝。tf.nn.leaky\_relu 对应的类为 layers.LeakyReLU，可以通过 LeakyReLU(alpha)创建 LeakyReLU 网络层，并设置𝑝参数，像 Dense 层一样将 LeakyReLU 层放置在网络的合适  tf.nn.leaky\_relu(x, alpha=0.1) |

通过 Numpy 实现 LeakyReLU 函数的导数，代码如下：

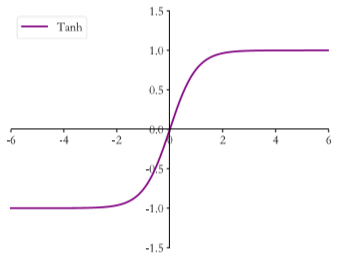
|  |
| --- |
| # 其中 p 为 LeakyReLU 的负半段斜率，为超参数  def derivative(x, p):  dx = np.ones\_like(x) # 创建梯度张量，全部初始化为 1  dx[x < 0] = p # 元素为负的导数为 p  return d |

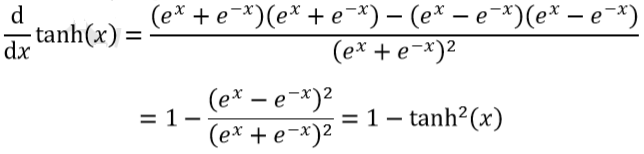
# Tanh

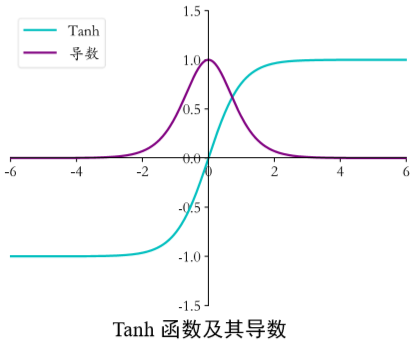
Tanh 函数能够将𝑥 ∈ 𝑅的输入“压缩”到(−1,1)区间，定义为：



可以看到 tanh 激活函数可通过 Sigmoid 函数缩放平移后实现，函数曲线下图所示。







在 TensorFlow 中，可以通过 tf.nn.tanh 实现 tanh 函数，代码如下：

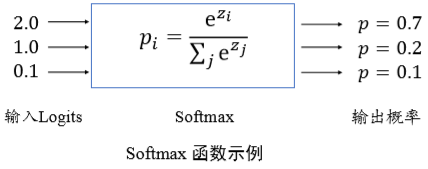
|  |
| --- |
| tf.nn.tanh(x) # 通过 tanh 激活函数 |

在 Numpy 中，借助于 Sigmoid 函数实现 Tanh 函数的导数，代码如下：

|  |
| --- |
| def sigmoid(x): # sigmoid 函数实现  return 1 / (1 + np.exp(-x))    def tanh(x): # tanh 函数实现  return 2\*sigmoid(2\*x) - 1    def derivative(x): # tanh 导数实现  return 1-tanh(x)\*\*2 |

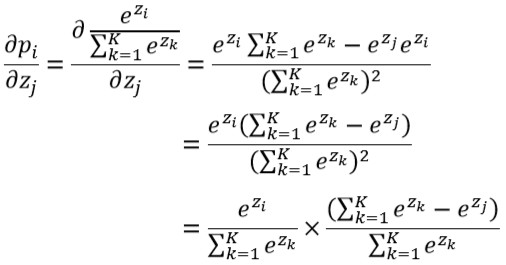
# Softmax

Softmax函数的输出值𝑜𝑖 ∈ [0,1]，且所有输出值之和为 1。其函数定义为：

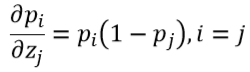


对于 Softmax 函数，，下面我们根据𝑖 = 𝑗和𝑖 ≠ 𝑗来分别推导 Softmax 函数的梯度。

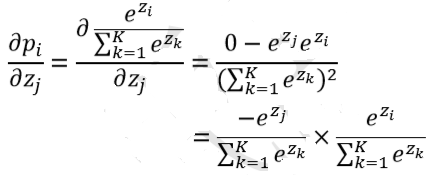
当𝑖 = 𝑗时。Softmax 函数的偏导数𝜕𝑝𝑖/𝜕𝑧，可以展开为：



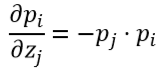
可以看到，上式是概率值𝑝𝑖和1 − 𝑝 的相乘，同时满足𝑝𝑖 = 𝑝j 。因此𝑖 = 𝑗时，Softmax 函数 的偏导数𝜕𝑝𝑖/ 𝜕𝑧 为：



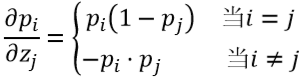
当𝑖 ≠ 𝑗时。Softmax 函数的偏导数𝜕𝑝𝑖/𝜕𝑧，可以展开为：



即：



基于上面的推导，Softmax偏导数表达式如下：



在 TensorFlow 中，可以通过 tf.nn.softmax 实现 Softmax 函数，代码如下：

|  |
| --- |
| z = tf.constant([2.,1.,0.1])  tf.nn.softmax(z) # 通过 Softmax 函数 |

在 Softmax 函数的数值计算过程中，容易因输入值偏大发生数值溢出现象；在计算交 叉熵时，也会出现数值溢出的问题。为了数值计算的稳定性，TensorFlow 中提供了一个统一的接口，将 Softmax 与交叉熵损失函数同时实现，同时也处理了数值不稳定的异常，一般推荐使用这些接口函数，避免分开使用 Softmax 函数与交叉熵损失函数。函数式接口为 tf.keras.losses.categorical\_crossentropy(y\_true, y\_pred, from\_logits=False)，其中 y\_true 代表了 One-hot 编码后的真实标签，y\_pred 表示网络的预测值，当 from\_logits 设置为 True 时， y\_pred 表示须为未经过 Softmax 函数的变量 z；当 from\_logits 设置为 False 时，y\_pred 表示 为经过 Softmax 函数的输出。为了数值计算稳定性，一般设置 from\_logits 为 True，此时 tf.keras.losses.categorical\_crossentropy 将在内部进行 Softmax 函数计算，所以不需要在模型 中显式调用 Softmax 函数，例如。

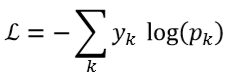
|  |
| --- |
| z = tf.random.normal([2,10]) # 构造输出层的输出  y\_onehot = tf.constant([1,3]) # 构造真实值  y\_onehot = tf.one\_hot(y\_onehot, depth=10) # one-hot 编码  # 输出层未使用 Softmax 函数，故 from\_logits 设置为 True  # 这样 categorical\_crossentropy 函数在计算损失函数前，会先内部调用 Softmax 函数  loss = keras.losses.categorical\_crossentropy(y\_onehot,z,from\_logits=True)  loss = tf.reduce\_mean(loss) # 计算平均交叉熵损失 |

除了函数式接口，也可以利用 losses.CategoricalCrossentropy(from\_logits)类方式同时实 现 Softmax 与交叉熵损失函数的计算，from\_logits 参数的设置方式相同。例如：

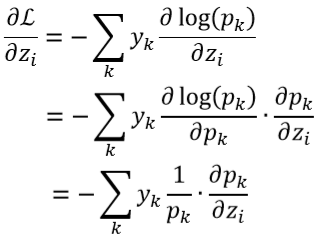
|  |
| --- |
| # 创建 Softmax 与交叉熵计算类，输出层的输出 z 未使用 softmax  criteon = keras.losses.CategoricalCrossentropy(from\_logits=True)  loss = criteon(y\_onehot,z) # 计算损失 |

# 交叉熵

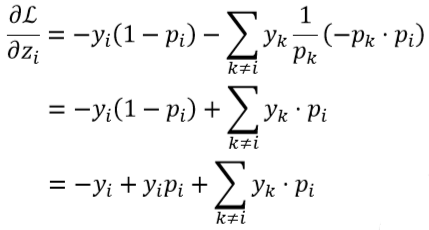
交叉熵损失函数通常用于分类问题，其公式化描述如下：



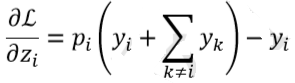
交叉熵损失函数求导过程如下：



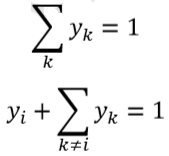
其中𝜕𝑝𝑘/𝜕z𝑖，即为上面推导的 Softmax 函数的偏导数。将求和符号拆分为𝑘 = 𝑖以及𝑘≠𝑖的两种情况，并代入𝜕𝑝𝑘/ 𝜕z𝑖求解的公式，可得



提供公共项𝑝𝑖，可得：



特别地，对于分类问题中标签𝑦通过 One-hot 编码的方式，则有如下关系：



因此交叉熵的偏导数可以进一步简化为：

