第 5 章

深度学习基础

深度学习是目前比较成功的表示学习方法，是机器学习的一个分支。本章重点讨论深度学习的基本原理和神经网络结构。5.1 节介绍神经网络最简单的结构感知机。5.2 节介绍神经网络的基本结构及函数、模型、算法。5.3 节介绍深度学习的雏形三层神经网络。5.4 节介绍深度学习在自然语言处理中的应用。

1min

5.1 感知机

深度学习的灵感来源于人脑过滤信息的方式，其初衷是通过模拟人脑运行模式来做些不一样的事情。人脑中有大约 1 亿个神经元。每个神经元都与另外约 10 万个同类相连，人类现在正试图将类似的工作原理应用到机器上。

人脑中的神经元分为胞体(Body)、树突(Dendrites)和轴突(Axon)。神经元发出的信号经由轴突传送到下一个神经元的树突。这种信号连接被称作神经元突触(Synapse)。单个的神经元其实没多大用处，但如果有很多神经元互相合作就能发挥奇效。这就是深度学习算法背后的理念。从观察中获得输入数据，再对输入数据进行筛选，这类筛选产生的输出数据就是下一层筛选的输入数据，这个过程不断进行，直到获得最后的输出信号。

最简单的人工神经网络：只有一个神经元的单层神经网络，即感知机。它可以完成简单的线性分类任务。

为了形象地说明感知机的工作机理，可以用神经网络中的感知机模型来描述，如图 5-1 所示。

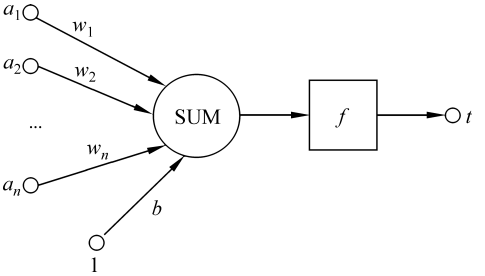

图 5-1　感知机的模型图

图 5-1 中，n 维向量$(a_1,a_2,\cdots,a_n)^{\mathrm{T}}$ 作为感知机的输入，$(w_1,w_2,\cdots,w_n)^{\mathrm{T}}$ 为输入分量连接到感知机的权重(weight)，b 为偏置(bias)，f 为激活函数，t 为感知机的输出。t 的数学表示如式(5-1)所示。

$$t=f\left(\sum_{i=1}^{n}w_ix_i+b\right)=f(\boldsymbol{W}^{\mathrm{T}}\boldsymbol{X}) \tag{5-1}$$

另外，这里的 f 用的是符号函数，如式(5-2)所示。

$$f(x)=\begin{cases}+1, & x\geqslant 0\\ -1, & x\leqslant 0\end{cases} \tag{5-2}$$

符号函数的函数图像如图 5-2 所示。

可以看出，符号函数是非连续的，不光滑的，这只是激活函数的一种。

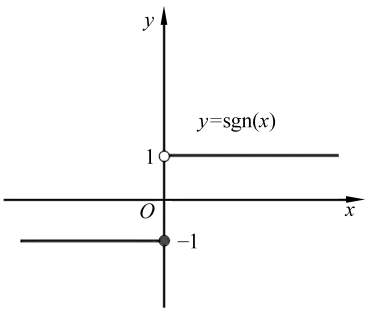

图 5-2　符号函数的函数图像

5.2　简单神经网络

在机器学习和认知科学领域，人工神经网络(简称神经网络或类神经网络)是一种模仿生物神经网络的结构和功能的计算模型，用于对函数进行估计或近似。神经网络是一种机器学习算法，经过 70 多年的发展，逐渐成为人工智能的主流。

5.2.1　简单神经网络框架


3min

神经网络可以被理解为一种分层的“神经元”网络结构。预测变量(或输入)构成底层，响应变量(或输出)构成顶层，还可能存在包含“隐藏神经元”的中间层。

最简单的网络不包含中间的隐藏层，等价于线性回归，如图 5-3 所示，展示了等价于线性回归且包含 4 个预测变量的神经网络。这些预测变量对应的系数称为“权重”，响应变量由输入项的线性组合得到。在神经网络框架中，通过“学习算法”最小化，诸如 MSE 等“损失函数”，从而确定权重大小。在这个简单的案例中，可以使用线性回归，这是一种更有效的训练模型的方法。

如果增加一个包含隐藏神经元的中间层，神经网络就成为非线性形式。一个简单的例子如图 5-4 所示。

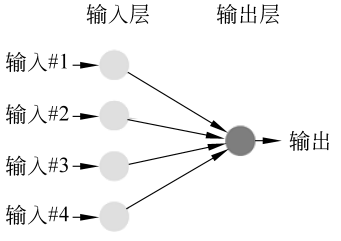

图 5-3　与线性回归等价的简单神经网络

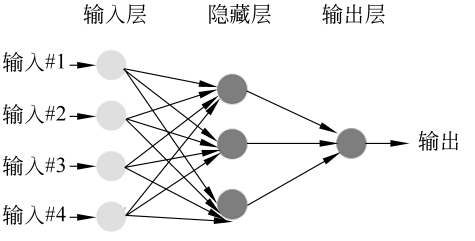

图 5-4　包含 4 个输入及 1 个隐藏层的神经网络，其中隐藏层中包含 3 个隐藏神经元

这是一个多层前馈网络，其中每层节点都接收来自先前层的输入，每层节点的输出是下一层的输入，每一节点接收输入后会对它们进行加权线性组合。在输出之前，用一个非线性函数对结果进行修改。如式(5-3)所示，对第 j 个隐藏神经元的输入进行线性组合。

$$z_j = b_j + \sum_{i=1}^{4} w_{i,j} x_i \tag{5-3}$$

在隐藏层，使用非线性函数(如 Sigmoid)对其进行修改，见式(5-4)：

$$s(z) = \frac{1}{1+e^{-z}} \tag{5-4}$$

这往往会降低极端输入值的影响，从而使神经网络对异常值具有稳健性。通常需要对权重大小进行限制以防其过大。限制权重的参数称为“衰减参数”，通常设置为 0.1。权重最初取随机值，然后根据观察到的数据进行更新，因此，根据神经网络产生的预测中存在的随机性因素，通常选取不同的随机起点进行多次训练，并对得到的结果进行平均。

在神经网络中，必须提前确定隐藏层的个数及每个隐藏层中的节点数。

5.2.2 激活函数

7min

激活函数可以为神经网络提供非线性特征，对神经网络的功能影响很大。20 世纪 70 年代，神经网络研究一度陷入低谷的主要原因是，明斯基证明了当时的神经网络由于没有 Sigmoid 这类非线性的激活函数，无法解决非线性可分问题，例如异或问题。从某种意义上讲，非线性激活函数拯救了神经网络。

激活函数对于人工神经网络模型去学习、理解非常复杂和非线性的函数来讲具有十分重要的作用。它们将非线性特性引入网络中。在神经元中，输入(Inputs)通过加权和求和后，还被作用在一个函数上，这个函数就是激活函数，如图 5-5 所示。

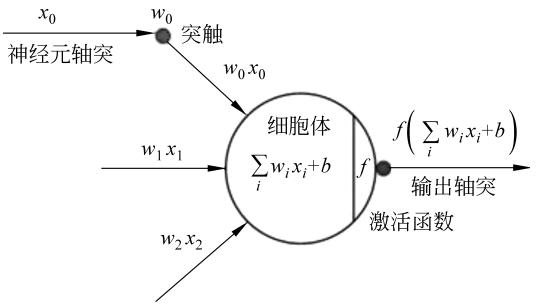

图 5-5 激活函数作用在神经元上

在实际选择激活函数时，通常要求激活函数是可微的、输出值的范围是有限的。由于基于反向传播的神经网络训练算法使用梯度下降法来进行优化训练，所以激活函数必须是可微的。激活函数的输出决定了下一层神经网络的输入，如果激活函数的输出范围是有限的，则特征表示受到有限权重的影响会更显著，基于梯度的优化方法就会更稳定；如果激活函数的输出范围是无限的，则神经网络的训练速度可能会很快，但必须选择一个合适的学习率

(Learning Rate)。

常用的激活函数：Sigmoid、Tanh、ReLU、Leaky ReLU、PReLU、ELU、Maxout、SELU。

1. Sigmoid 函数

Sigmoid 函数又称 Logistic 函数，用于隐含层神经元输出，取值范围为(0,1)，可以用来做二分类。Sigmoid 函数表达式如式(5-5)所示。

$$f(x)=\frac{1}{1+e^{-x}} \tag{5-5}$$

优点：Sigmoid 函数的输出在(0,1)之间，输出范围有限，优化稳定，可以用作输出层；连续函数，便于求导。

缺点：Sigmoid 函数在变量取绝对值非常大的正值或负值时会出现饱和现象，意味着函数会变得很平，并且对输入的微小改变会变得不敏感。在反向传播时，当梯度接近于 0 时，权重基本不会更新，很容易就会出现梯度消失的情况，从而无法完成深层网络的训练。Sigmoid 函数的输出不是 0 均值的会导致后层的神经元的输入是非 0 均值的信号，这会对梯度产生影响。计算复杂度高，因为 Sigmoid 函数是指数形式。

2. Tanh 函数

Tanh 函数也称为双曲正切函数，取值范围为[－1,1]。Tanh 函数的定义如式(5-6)所示。

$$f(x)=\frac{1-e^{-2x}}{1+e^{-2x}} \tag{5-6}$$

Tanh 函数与 Sigmoid 函数相比，输出均值为 0，这就使其收敛速度要比 Sigmoid 快，从而可以减少迭代次数。缺点就是同样具有软饱和性，会造成梯度消失。

3. ReLU 函数

整流线性单元(Rectified Linear Unit，ReLU)是现代神经网络中最常用的激活函数，是大多数前馈神经网络默认使用的激活函数。ReLU 函数的定义如式(5-7)所示。

$$f(x)=\begin{cases}x, & x\geqslant 0\\ 0, & x<0\end{cases}$$

$$f(x)=\max(0,x) \tag{5-7}$$

它在 $x>0$ 时不存在饱和问题，保持梯度不衰减，从而解决了梯度消失问题。能直接以监督的方式训练深度神经网络，而无须依赖无监督的逐层预训练。随着训练的推进，部分输入会落入硬饱和区，导致对应权重无法更新，这种现象称为“神经元死亡”。

与 Sigmoid 类似，ReLU 的输出均值也大于 0，所以偏移现象和神经元死亡共同影响网络的收敛性。

4. Leaky ReLU 函数

渗漏整流线性单元(Leaky ReLU)，为了解决 Dead ReLU 现象。用一个类似 0.01 的小值来初始化神经元，从而使 ReLU 在负数区域更偏向于激活而不是死掉。这里的斜率都是确定的。

Leaky ReLU 激活函数是 ReLU 的衍变版本，主要就是为了解决 ReLU 输出为 0 的问题。在输入小于 0 时，虽然输出值很小，但是值不为 0。Leaky ReLU 激活函数的一个缺点就是它有些近似线性，导致在复杂分类中效果不好，如式(5-8)所示。

$$f(x)=\begin{cases}x, & x\geqslant 0\\ \alpha x, & x<0\end{cases} \tag{5-8}$$

5. ELU 指数线性单元

具有 ReLU 的优势，没有 Dead ReLU 问题，输出均值接近 0，实际上 PReLU 和 Leaky ReLU 都有这一优点。有负数饱和区域，从而对噪声有一些稳健性。可以看作介于 ReLU 和 Leaky ReLU 之间的一个函数。这个函数也需要计算 EXP，从而计算量上更大一些。它结合了 Sigmoid 和 ReLU 函数，左侧软饱和，右侧无饱和。右侧线性部分使 ELU 能缓解梯度消失，而左侧软饱和能让 ELU 对输入变化或噪声更稳健。由于 ELU 的输出均值接近于 0，所以收敛速度更快。如式(5-9)所示。

$$f(x)=\begin{cases}x, & x\geqslant 0\\ \alpha(e^{x}-1), & x<0\end{cases} \tag{5-9}$$

8min

5.2.3 损失函数

一言以蔽之，损失函数(Loss Function)就是用来度量模型的预测值 $f(x)$ 与真实值 Y 的差异程度的运算函数，它是一个非负实值函数，通常使用 $L(Y,f(x))$ 来表示，损失函数越小，模型的稳健性就越好。

损失函数主要使用在模型的训练阶段，每个批次的训练数据送入模型后，通过前向传播输出预测值，然后损失函数会计算出预测值和真实值之间的差异值，也就是损失值。得到损失值之后，模型通过反向传播去更新各个参数，以此来降低真实值与预测值之间的损失，使模型生成的预测值往真实值方向靠拢，从而达到学习的目的。

1. 均方误差损失函数(Mean Square Error，MSE)

均方误差损失函数如式(5-10)所示。

$$L(Y\mid f(x))=\frac{1}{n}\sum_{i=1}^{n}(Y_i-f(x_i))^2 \tag{5-10}$$

在回归问题中，MSE 用于度量样本点到回归曲线的距离，通过最小化平方损失使样本点可以更好地拟合回归曲线。MSE 的值越小，表示预测模型描述的样本数据具有越好的精确度。由于无参数、计算成本低和具有明确物理意义等优点，MSE 已成为一种优秀的距离度量方法。尽管 MSE 在图像和语音处理方面表现较弱，但它仍是评价信号质量的标准，在回归问题中，MSE 常被作为模型的经验损失或算法的性能指标。

2. L1 损失函数

L1 损失函数的公式如式(5-11)所示。

$$L(Y\mid f(x))=\sum_{i=1}^{n}\mid Y_i-f(x_i)\mid \tag{5-11}$$

L1 损失又称为曼哈顿距离，表示残差的绝对值之和。L1 损失函数对离群点有很好的稳健性，但它在残差为 0 处却不可导。另一个缺点是更新的梯度始终相同，也就是说，即使很小的损失值，梯度也很大，这样不利于模型的收敛。针对它的收敛问题，一般的解决办法是在优化算法中使用变化的学习率，在损失接近最小值时降低学习率。

3. L2 损失函数

L2 损失函数的公式如式(5-12)所示。

$$L(Y \mid f(x)) = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(Y_i - f(x_i))^2} \tag{5-12}$$

L2 损失又被称为欧氏距离，是一种常用的距离度量方法，通常用于度量数据点之间的相似度。由于 L2 损失具有凸性和可微性，并且在独立、同分布的高斯噪声情况下，它能提供最大似然估计，使它成为回归问题、模式识别、图像处理中最常使用的损失函数。

4. Smooth L1 损失函数

Smooth L1 损失函数的公式如式(5-13)所示。

$$L(Y \mid f(x)) = \begin{cases} \dfrac{1}{2}(Y - f(x))^2, & \mid Y - f(x) \mid < 1 \\ \mid Y - f(x) \mid - \dfrac{1}{2}, & \mid Y - f(x) \mid \geqslant 1 \end{cases} \tag{5-13}$$

Smooth L1 损失是由格尔希克在 Fast R-CNN 中提出的，主要用在目标检测中以防止梯度爆炸。

5. Huber 损失函数

Huber 损失函数的公式如式(5-14)所示。

$$L(Y \mid f(x)) = \begin{cases} \dfrac{1}{2}(Y - f(x))^2, & \mid Y - f(x) \mid \leqslant \delta \\ \delta \mid Y - f(x) \mid - \dfrac{1}{2}\delta^2, & \mid Y - f(x) \mid > \delta \end{cases} \tag{5-14}$$

Huber 损失是平方损失和绝对损失的综合，它克服了平方损失和绝对损失的缺点，不仅使损失函数具有连续的导数，而且利用 MSE 梯度随误差减小的特性，可取得更精确的最小值。尽管 Huber 损失对异常点具有更好的稳健性，但是，它不仅引入了额外的参数，而且选择合适的参数比较困难，这也增加了训练和调试的工作量。

6. KL 散度函数(相对熵)

KL 散度函数(相对熵)的公式如式(5-15)所示。

$$L(Y \mid f(x)) = \sum_{i=1}^{n} Y_i \times \log\left(\frac{Y_i}{f(x_i)}\right) \tag{5-15}$$

式中，Y 代表真实值，$f(x)$代表预测值。

KL 散度(Kullback-Leibler Divergence)也被称为相对熵，是一种非对称度量方法，常用于度量两个概率分布之间的距离。KL 散度也可以衡量两个随机分布之间的距离，两个随机分布的相似度越高，它们的 KL 散度越小，当两个随机分布的差别增大时，它们的 KL 散

度也会增大，因此 KL 散度可以用于比较文本标签或图像的相似性。基于 KL 散度的演化损失函数有 JS 散度函数(Jensen-Shannon Divergence)。JS 散度也称 JS 距离，用于衡量两个概率分布之间的相似度，它是基于 KL 散度的一种变形，消除了 KL 散度非对称的问题，与 KL 散度相比，它使相似度判别更加准确。相对熵是恒大于或等于 0 的。当且仅当两分布相同时，相对熵才等于 0。

7. 交叉熵

交叉熵损失的公式如式(5-16)所示。

$$L(Y \mid f(x)) = -\sum_{i=1}^{n} Y_i \log f(x_i) \tag{5-16}$$

交叉熵是信息论中的一个概念，最初用于估算平均编码长度，引入机器学习后，用于评估当前训练得到的概率分布与真实分布的差异情况。为了使神经网络的每层输出从线性组合转换为非线性逼近，以提高模型的预测精度，在以交叉熵为损失函数的神经网络模型中一般选用 Tanh、Sigmoid、Softmax 或 ReLU 作为激活函数。

交叉熵损失函数刻画了实际输出概率与期望输出概率之间的相似度，也就是交叉熵的值越小，两个概率分布就越接近，特别是在正负样本不均衡的分类问题中，常用交叉熵作为损失函数。目前，交叉熵损失函数是卷积神经网络中最常使用的分类损失函数，它可以有效地避免梯度消散。在二分类情况下也叫作对数损失函数。

8. Softmax 损失函数

Softmax 损失函数的公式如式(5-17)所示。

$$L(Y \mid f(x)) = -\frac{1}{n}\sum_{i=1}^{n} \log \frac{e^{fY_i}}{\sum_{j=1}^{c} e^{f_j}} \tag{5-17}$$

从标准形式上看，Softmax 损失函数应归到对数损失的范畴，在监督学习中，由于它被广泛使用，所以单独形成一个类别。Softmax 损失函数本质上是逻辑回归模型在多分类任务上的一种延伸，常作为 CNN 模型的损失函数。Softmax 损失函数的本质是将一个 k 维的任意实数值向量 x 映射成另一个 k 维的实数值向量，其中，输出向量中的每个元素的取值范围都是(0,1)，即 Softmax 损失函数输出每个类别的预测概率。由于 Softmax 损失函数具有类间可分性，被广泛用于分类、分割、人脸识别、图像自动标注和人脸验证等问题中，其特点是类间距离的优化效果非常好，但类内距离的优化效果比较差。

Softmax 损失函数具有类间可分性，在多分类和图像标注问题中，常用它解决特征分离问题。在基于卷积神经网络的分类问题中，一般使用 Softmax 损失函数作为损失函数，但是 Softmax 损失函数学习到的特征不具有足够的区分性，因此它常与对比损失或中心损失组合使用，以增强区分能力。

5.2.4 梯度法

梯度算法是一种常用的优化算法，广泛应用于机器学习和深度学习领域。它通过不断

调整参数来最小化或最大化一个目标函数，以达到优化的目的。

梯度算法的核心思想是基于目标函数的梯度信息来决定参数的更新方向和步长。梯度是一个向量，表示函数在某一点上的变化率。对于一个多元函数，其梯度是一个向量，包含了各个自变量的偏导数。函数在某一点的梯度是这样一个向量，它的方向与取得最大方向导数的方向一致，而它的模为方向导数的最大值。以山为例，就是坡度最陡的地方，梯度值就是描述坡度有多陡的。

1. 梯度下降法

梯度下降法沿着负梯度方向逐步更新优化参数，函数在某一点的梯度是在该方向单位步长上升最快的向量。梯度下降法是利用待优化变量，沿着负梯度方向不断迭代寻找最优值，如图 5-6 所示。

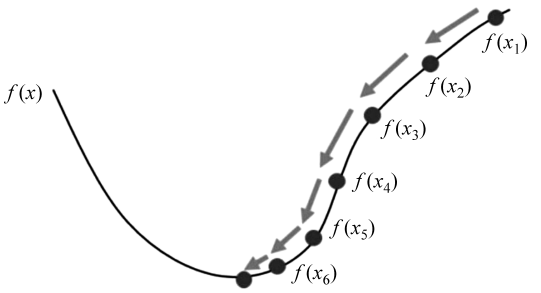

图 5-6　梯度下降法

梯度下降法算法流程见表 5-1。

表 5-1　梯度下降法算法流程

Algorithm1：梯度下降法
Require：步长 α，初始参数 x_0 repeat： 梯度计算：$\nabla f(x_i)$ 参数更新：$x_{i+1}=x_i-\alpha\nabla f(x_i)$ until：达到收敛条件

以下是使用 Python 编写的一个简单的梯度下降法代码示例：

```
//第 5 章/GradientDescent.python
import numpy as np
#定义函数及其导数
def f(x):
    return x ** 2 + 2 * x + 1
def df(x):
    return 2 * x + 2
#定义梯度下降函数
def gradient_descent(x, learning_rate, num_iterations):
```

```
    for i in range(num_iterations):
        grad = df(x)
        x = x - learning_rate * grad
    return x
#初始化参数
x0 = np.random.randn(1)
#调用梯度下降函数
learning_rate = 0.1
num_iterations = 100
x_min = gradient_descent(x0, learning_rate, num_iterations)
#打印最小值
print('Minimum value: ', f(x_min))
```

2. 最优梯度法

最优梯度法利用梯度计算步长，减小在谷底的来回振动，梯度法设置固定步长，可能出现的情况是在谷底左右来回波动难以收敛。最优梯度法根据梯度模长设置步长，越接近最优点，步长越短。相比梯度下降法，最优梯度法的核心在于利用梯度计算步长，见表 5-2。

表 5-2 最优梯度法算法流程

Algorithm2：最优梯度下降法
Require：初始步长 α_0，初始参数 x_0 repeat： 梯度计算：$\nabla f(x_i)$、$\|\nabla f(x_i)\|$ 计算搜索方向：$\theta_i=\nabla f(x_i)/\|\nabla f(x_i)\|$ 步长计算：$\alpha=-\nabla f(x_i)\theta/\theta_i^T\nabla\nabla f(x_i)\theta_i$ 参数更新：$x_{i+1}=x_i-\alpha\theta_i$ until：达到收敛条件

以下是使用 Python 编写的最优梯度法代码示例：

```
//第 5 章/GradientDescent.python
def gradient_descent(x, learning_rate, num_iterations):
    for _ in range(num_iterations):
        gradient = 2 * x                        #函数 f(x) = x^2 的导数为 2x
        x = x - learning_rate * gradient
    return x
#初始化参数
x0 = 1.0
learning_rate = 0.1
num_iterations = 100
#调用最优梯度法
x_min = gradient_descent(x0, learning_rate, num_iterations)
#打印最小值
```

```
print('Minimum value: ', x_min)
print('Function minimum: ', x_min ** 2)
```

3. 共轭梯度法

共轭梯度法每次搜索方向与上次方向共轭，理论上 k 维变量经过 k 次迭代可找到最优解。共轭梯度法对最优梯度法进行了修正，搜索方向为共轭方向，将负梯度方向旋转了一个角度，每次最优方向需要在负梯度方向进行修正，见表 5-3。

表 5-3　共轭梯度法算法流程

Algorithm3：共轭梯度法

Require：初始参数 x_0

确定初始优化方向 $P_0=-\nabla f(x_0)$：

repeat：

步长计算：$\lambda_i=\dfrac{-\nabla f(x_i)P_i}{P_I^T\ \nabla^2 f(x_i)P_i}$

参数更新：$x_{i+1}=x_i+\lambda_i P_i$

梯度计算：$g_i=\nabla f(x_i)$，$g_{i+1}=\nabla f(x_{i+1})$

步长计算：$\alpha_i=\|g_{i+1}\|^2/\|g_i\|^2$

方向更新：$P_{i+1}=-g_{i+1}+\alpha_i P_i$

until：达到收敛条件

以下是使用 Python 编写的共轭梯度法代码示例：

```
//第 5 章/ConjugateGradient.python
import numpy as np
#定义函数及其导数
def f(x):
    return x ** 2 + 2 * x + 1
def df(x):
    return 2 * x + 2
#定义共轭梯度法
def conjugate_gradient(x, num_iterations):
    r = -df(x)
    p = r
    rs_old = np.dot(r, r)
    for i in range(num_iterations):
        Ap = df(p)
        alpha = rs_old / np.dot(p, Ap)
        x = x + alpha * p
        r = r - alpha * Ap
        rs_new = np.dot(r, r)
        if np.sqrt(rs_new) < 1e-8:
            break
        p = r + (rs_new / rs_old) * p
```