

# 尚硅谷大模型技术之深度学习

(作者: 尚硅谷研究院)

版本: V1.2.0

## 第 1 章 深度学习概述

### 1.1 深度学习简介

#### 人工智能 artificial intelligence

人工智能是一门让机器表现出类似人类智能行为的学科，旨在使机器能够执行诸如推理、学习、规划、决策和语言理解等任务。

#### 机器学习 machine learning

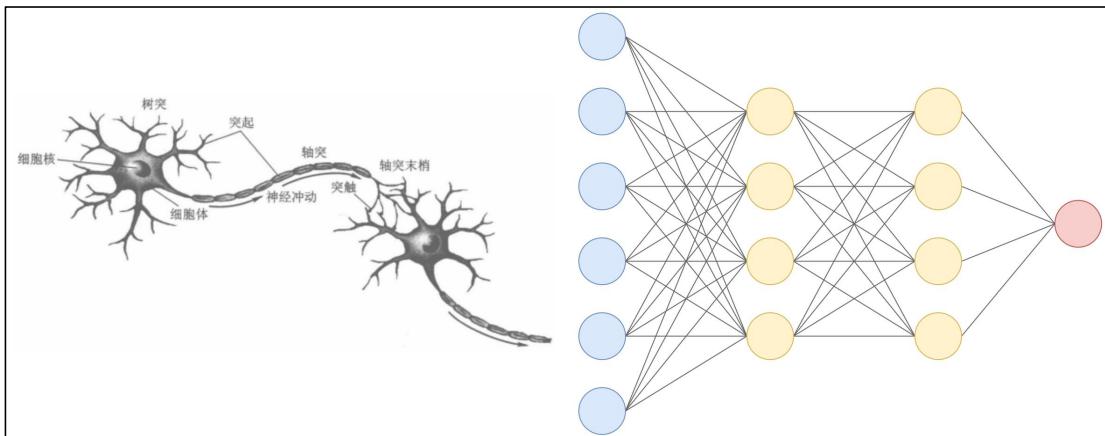
机器学习是人工智能的一个子领域，强调通过数据训练模型，使其在没有明确编程规则的情况下，学会从数据中识别模式并做出预测或决策。

#### 深度学习 deep learning

深度学习是机器学习的一个子领域，主要关注使用多层神经网络（深度神经网络）来从大量数据中提取特征和进行复杂任务处理。

深度学习作为机器学习的一个分支，专注于使用多层神经网络（深度神经网络）来建模和解决问题。

人脑中有很多相互连接的神经元，当大脑处理信息时，这些神经元之间通过电信号和化学物质相互作用，在大脑的不同区域之间传递信息。神经网络使用人工神经元模仿这种生物现象，这些人工神经元由称为节点的软件模块构成，使用数值计算来进行通信和传递信息。



## 1.2 深度学习的特点

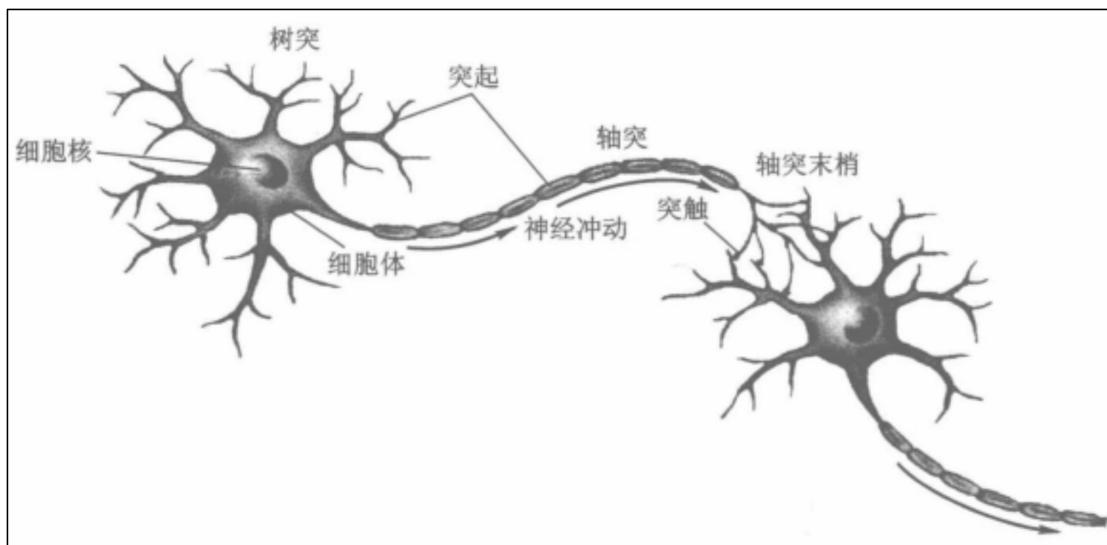
- 使用多层神经网络，能够自动提取数据的多层次特征。
- 适合处理非结构化数据，如图像、音频、文本等。
- 依赖大量数据和计算资源，训练时间较长。
- 模型复杂，通常被视为“黑箱”，解释性较差。

## 1.3 深度学习应用场景

- 金融服务：预测分析可推荐股票的算法交易，评估贷款审批的业务风险，检测欺诈行为，并帮助管理信贷和投资组合。
- 媒体和娱乐：从网上购物到流媒体服务，跟踪用户活动及开发个性化推荐也应用到了深度学习。
- 客户服务：聊天机器人、虚拟助手和拨入式客户服务门户网站利用语音识别等工具。
- 医疗卫生：通过图像识别应用和深度学习技术，医学影像专家可以利用数字化的医疗记录和医学影像数据来支持和改进医学诊断过程，提供更精确和高效的医疗服务。
- 工业自动化：在工厂和仓库中，深度学习应用可以自动检测人或物体何时处于机器的安全距离之外，或协助质量控制及预测性维护。
- 自动驾驶汽车：汽车行业研究员使用深度学习来训练汽车检测停车标志、红绿灯、人行横道和行人等对象。
- 航空航天和军事：深度学习技术可以用来在监控的大片地理区域中检测物体，从远处识别需要关注的区域，并为部队验证安全或不安全区域。
- 执法：语音识别、计算机视觉和自然语言处理（NLP）有助于分析大量数据，从而节省时间和资源。

## 第 2 章 神经网络基础

### 2.1 神经网络的构成



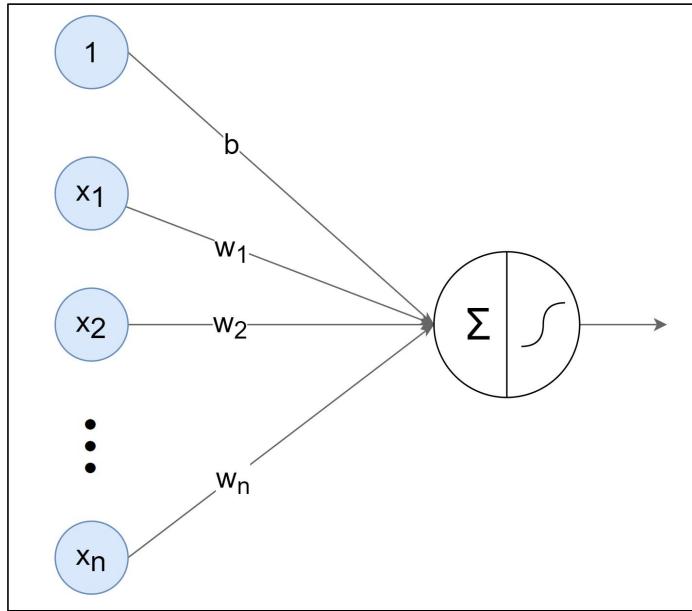
在生物体中，**神经元** 是神经系统最基本的结构和功能单位。

神经元从树突接收其它神经元细胞发出的电化学刺激脉冲，这些脉冲叠加后，一旦强度达到临界值，这个神经元就会产生动作电位，沿着轴突发送电信号。

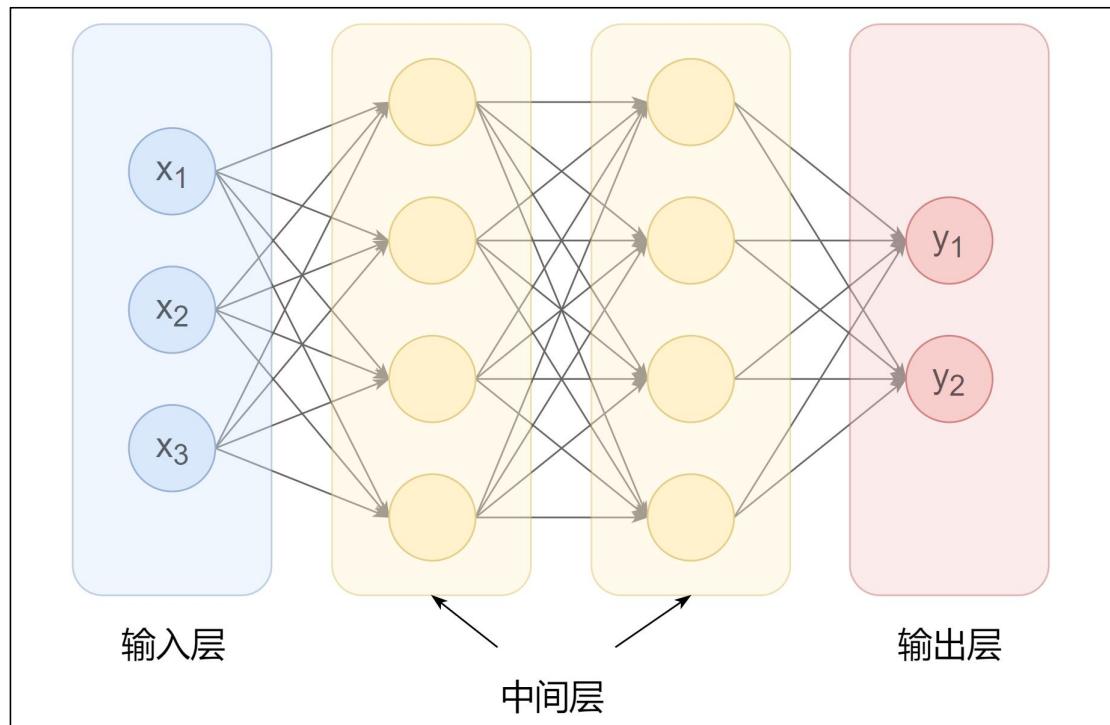
轴突将刺激传到末端的突触，电信号触发突触上面的电压敏感蛋白，把一个内含神经递质的小泡（突触小体）推到突触的膜上，从而释放出突触小体中的神经递质。这些化学物质会扩散到其它神经元的树突或轴突上。

#### 2.1.1 基本概念和结构

人工神经网络（Artificial Neural Network, ANN）简称 **神经网络（NN）**，是一种模仿生物神经网络结构和功能的计算模型。大多数情况下人工神经网络能在外界信息的基础上改变内部结构，是一种自适应系统(adaptive system)，通俗地讲就是具备学习功能。



人工神经网络中的神经元，一般可以对多个输入进行加权求和，再经过特定的“激活函数”转换后输出。



使用多个神经元就可以构建多层神经网络，最左边的一列神经元都表示输入，称为 **输入层**；最右边一列表示网络的输出，称为 **输出层**；输入层与输出层之间的层统称为 **中间层（隐藏层）**。

相邻层的神经元相互连接(图中下一层每个神经元都与上一层所有神经元连接，称为 **全连接**)，每个连接都会有一个 **权重**。

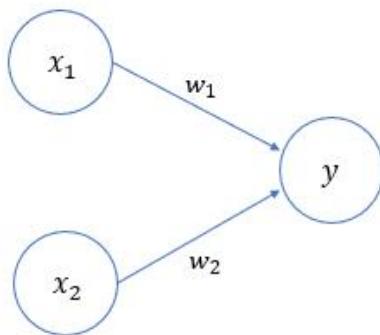
神经元中的信息逐层传递（一般称为 **前向传播 forward**），上一层神经元的输出作为下一层神经元的输入。

### 2.1.2 复习感知机

我们先复习一下在机器学习部分学习过的感知机。

感知机（Perceptron）是二分类模型，接收多个信号，输出一个信号。感知机的信号只有 0、1 两种取值。

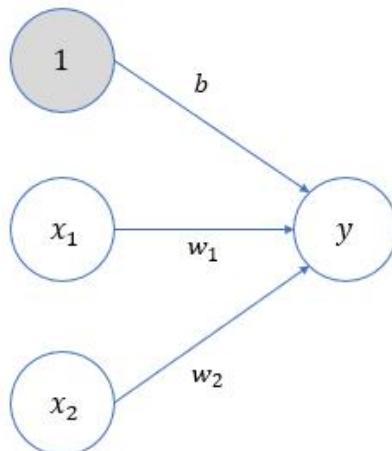
下图是一个接收两个输入信号的感知机的例子：



$x_1, x_2$  是输入信号， $y$  是输出信号， $w_1, w_2$  是权重， $\circ$  称为神经元或节点。输入信号被送往神经元时，会分别乘以固定的权重。神经元会计算传送过来的信号的总和，只有当这个总和超过某个界限值时才会输出 1，也称之为神经元被激活。

$$y = \begin{cases} 0 & (b + w_1x_1 + w_2x_2 \leq 0) \\ 1 & (b + w_1x_1 + w_2x_2 > 0) \end{cases} \quad (2.1)$$

这里将界限的阈值设为 0。除了权重  $w_1, w_2$  之外，还可以增加一个参数  $b$ ，被称为 **偏置**。



感知机的多个输入信号都有各自的权重，这些权重发挥着控制各个信号的重要性的作用，权重越大，对应信号的重要性越高。偏置则可以用来控制神经元被激活的容易程度。

### 2.1.3 引入激活函数

可以看出，式（2.1）中包含了两步处理操作：首先按照输入各自的权重及偏置，计算出一个加权总和；然后再根据这个总和与阈值的大小关系，决定输出 0 还是 1。

如果定义一个函数：

$$h(x) = \begin{cases} 0 & (x \leq 0) \\ 1 & (x > 0) \end{cases} \quad (2.2)$$

那么式（2.1）就可以简化为：

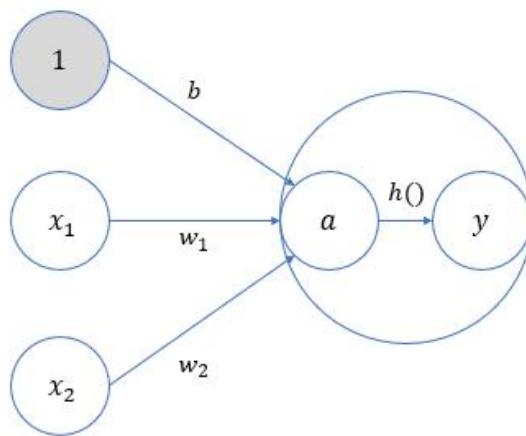
$$y = h(b + w_1x_1 + w_2x_2) \quad (2.3)$$

为了更明显地表示出两个处理步骤，可以进一步写成：

$$a = b + w_1x_1 + w_2x_2 \quad (2.4)$$

$$y = h(a) \quad (2.5)$$

这里，我们将输入信号和偏置的加权总和，记作  $a$ 。



$h(x)$  可以将输入信号的加权总和转换为输出信号，起到“激活神经元”的作用，所以被称为 **激活函数**。

## 2.2 激活函数

## 2.2.1 激活函数的作用

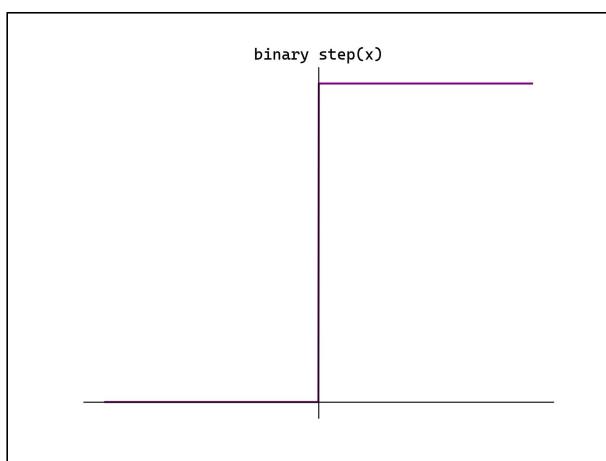
激活函数是连接感知机和神经网络的桥梁，在神经网络中起着至关重要的作用。

如果没有激活函数，整个神经网络就等效于单层线性变换，不论如何加深层数，总是存在与之等效的“无隐藏层的神经网络”。激活函数必须是非线性函数，也正是激活函数的存在为神经网络引入了非线性，使得神经网络能够学习和表示复杂的非线性关系。

## 2.2.2 阶跃（Binary step）函数

之前的感知机中，式（2.2）表示的  $h(x)$  就是最简单的激活函数，它可以为输入设置一个“阈值”；一旦超过这个阈值，就切换输出（0 或者 1）。这种函数被称为“阶跃函数”。

$$f(x) = \begin{cases} 0, & x < 0 \\ 1, & x \geq 0 \end{cases}, f'(x) = 0$$



阶跃函数的导数恒为 0。

阶跃函数可以用代码实现如下：

```
def step_function(x):
    if x > 0:
        return 1
    else:
        return 0
```

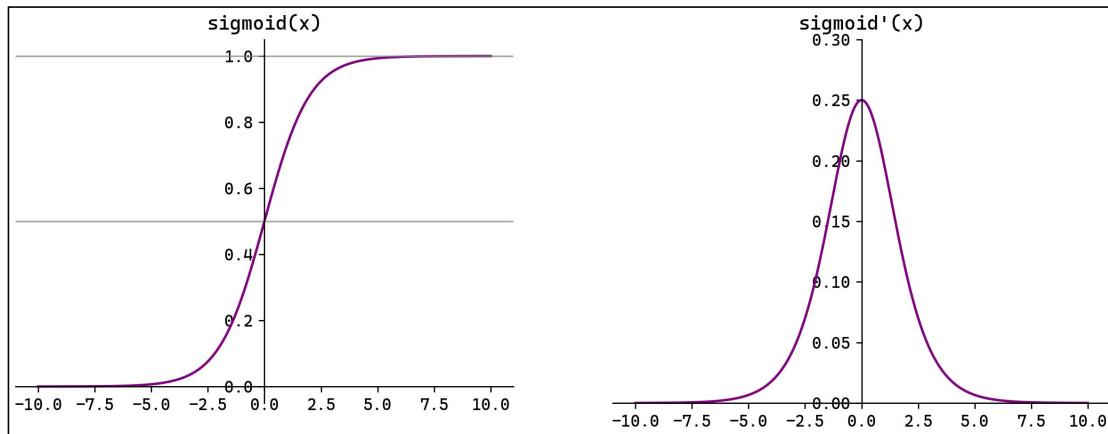
这里的  $x$  只能取一个数值（浮点数）。如果我们希望直接传入 Numpy 数组进行批量化操作，可以改进如下：

```
def step_function(x):
    return np.array(x > 0, dtype=int)
```

### 2.2.3 Sigmoid 函数

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

$$f'(x) = \frac{1}{1 + e^{-x}} \left(1 - \frac{1}{1 + e^{-x}}\right) = f(x)(1 - f(x))$$



Sigmoid（也叫 Logistic 函数）是平滑的、可微的，能将任意输入映射到区间(0,1)。常用于二分类的输出层。但因其涉及指数运算，计算量相对较高。

Sigmoid 的输入在[-6,6]之外时，其输出值变化很小，可能导致信息丢失。

Sigmoid 的输出并非以 0 为中心，其输出值均>0，导致后续层的输入始终为正，可能影响后续梯度更新方向。

Sigmoid 的导数范围为(0,0.25)，梯度较小。当输入在[-6,6]之外时，导数接近 0，此时网络参数的更新将会极其缓慢。使用 Sigmoid 作为激活函数，可能出现梯度消失（在逐层反向传播时，梯度会呈指数级衰减）。

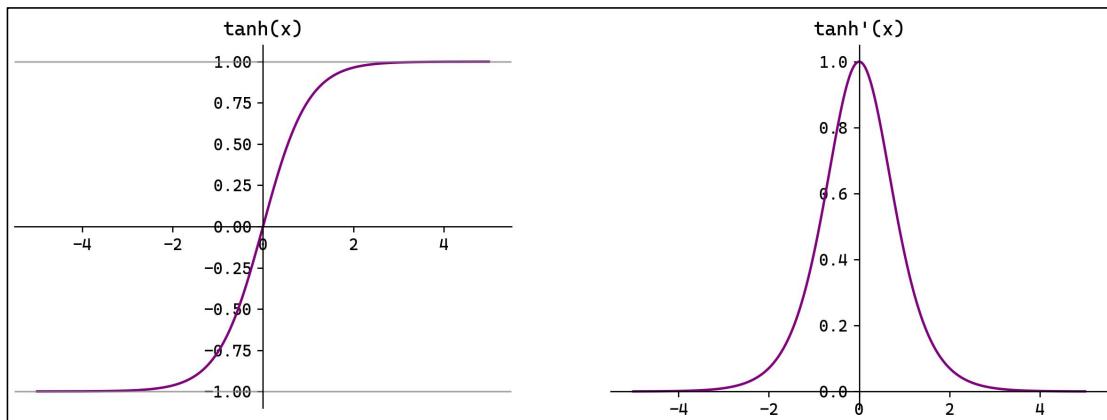
Sigmoid 函数可以用代码实现如下：

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

### 2.2.4 Tanh 函数

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

$$f'(x) = 1 - \left(\frac{1 - e^{-2x}}{1 + e^{-2x}}\right)^2 = 1 - f^2(x)$$



Tanh（双曲正切）将输入映射到区间(-1,1)。其关于原点中心对称。常用在隐藏层。

输入在[-3,3]之外时，Tanh 的输出值变化很小，此时其导数接近 0。

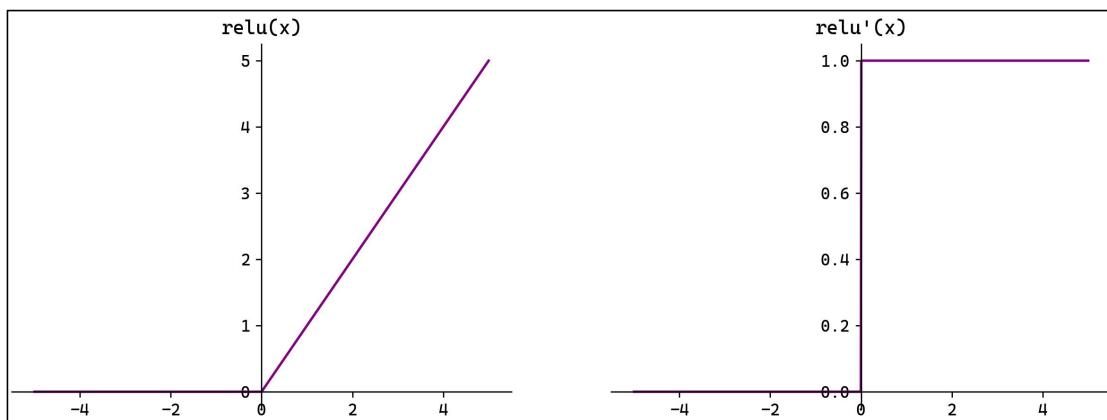
Tanh 的输出以 0 为中心，且其梯度相较于 Sigmoid 更大，收敛速度相对更快。但同样也存在梯度消失现象。

## 2.2.5 ReLU 函数

$$f(x) = \max(0, x) = \begin{cases} 0, & x \leq 0 \\ x, & x > 0 \end{cases}$$

$$f'(x) = \begin{cases} 0, & x \leq 0 \\ 1, & x > 0 \end{cases}$$

注意：x=0 时 ReLU 函数不可导，此时我们默认使用左侧的函数。



ReLU（Rectified Linear Unit，修正线性单元）会将小于 0 的输入转换为 0，大于等于 0 的输入则保持不变。ReLU 定义简单，计算量小。常用于隐藏层。

ReLU 作为激活函数不存在梯度消失。当输入小于 0 时，ReLU 的输出为 0，这意味着在神经网络中，ReLU 激活的节点只有部分是“活跃”的，这种稀疏性有助于减少计算量和提高模型的效率。

当神经元的输入持续为负数时，ReLU 的输出始终为 0。这意味着神经元可能永远不会被激活，从而导致“神经元死亡”问题。这会影响模型的学习能力，特别是如果大量的神经元都变成了“死神经元”。为解决此问题，可使用 Leaky ReLU 来代替 ReLU 作为激活函数。

Leaky ReLU ( $f(x) = \begin{cases} \alpha x, & x \leq 0 \\ x, & x > 0 \end{cases}$ ，其中  $\alpha$  是一个很小的常数) 在负数区域引入一个小的斜率来解决“神经元死亡”问题。

ReLU 函数可以用代码实现如下：

```
def relu(x):
    return np.maximum(0, x)
```

## 2.2.6 Softmax 函数

$$y_k = \frac{e^{x_k}}{\sum_{i=1}^n e^{x_i}}, k = 1 \sim n$$

$$\frac{\partial y_k}{\partial x_i} = \begin{cases} y_k(1 - y_i), & k = i \\ -y_k y_i, & k \neq i \end{cases}$$

Softmax 将一个任意的实数向量转换为一个概率分布，确保输出值的总和为 1，是二分类激活函数 Sigmoid 在多分类上的推广。Softmax 常用于多分类问题的输出层，用来表示类别的预测概率。

Softmax 会放大输入中较大的值，使得最大输入值对应的输出概率较大，其他较小的值会被压缩。即在类别之间起到了一定的区分作用。

$$\begin{bmatrix} 1 \\ 3 \\ 5 \\ 10 \end{bmatrix} \xrightarrow{\text{Softmax}} \begin{bmatrix} 0.00012246 \\ 0.00090485 \\ 0.006686 \\ 0.99229 \end{bmatrix}$$

Softmax 函数可以用代码实现如下：

```
def softmax(x):
    return np.exp(x) / np.sum(np.exp(x))
```

考虑到  $x$  较大时，指数函数的值会非常大，容易溢出，可以改进为：

```
def softmax(x):
    x = x - np.max(x) # 溢出对策
    return np.exp(x) / np.sum(np.exp(x))
```

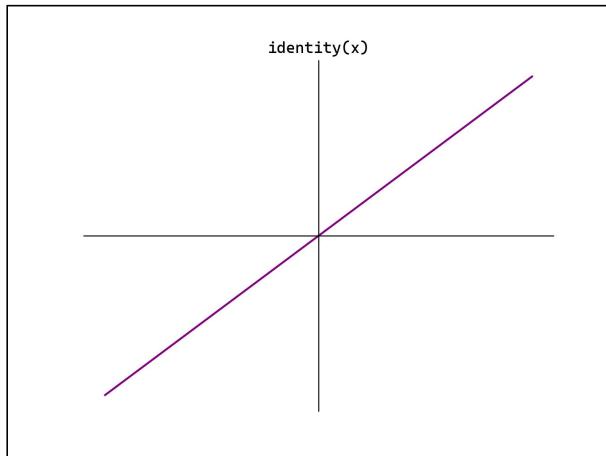
考虑到  $x$  为二维数组（矩阵）的情况，可以进一步写为：

```
def softmax(x):  
    if x.ndim == 2:  
        x = x.T  
        x = x - np.max(x, axis=0)  
        y = np.exp(x) / np.sum(np.exp(x), axis=0)  
        return y.T  
  
    x = x - np.max(x) # 溢出对策  
    return np.exp(x) / np.sum(np.exp(x))
```

## 2.2.7 其他常见激活函数

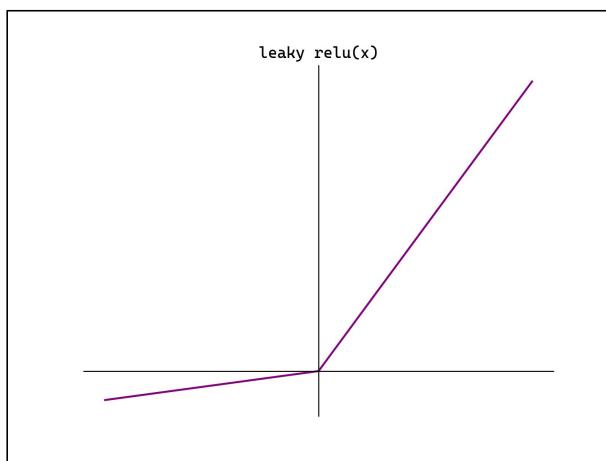
### 1) Identity (恒等函数)

$$f(x) = x, f'(x) = 1$$



### 2) Leaky ReLU (Leaky Rectified Linear Unit)

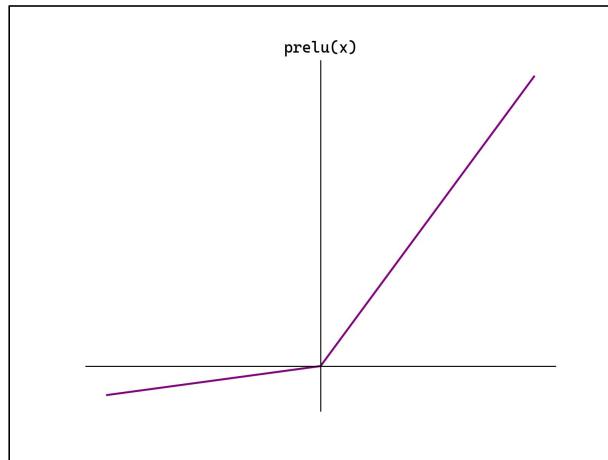
$$f(x) = \begin{cases} \alpha x, & x \leq 0 \\ x, & x > 0 \end{cases}, \quad f'(x) = \begin{cases} \alpha, & x \leq 0 \\ 1, & x > 0 \end{cases}$$



### 3) PReLU (Parametric Rectified Linear Unit)

$$f(x) = \begin{cases} \alpha x, & x \leq 0 \\ x, & x > 0 \end{cases}, \quad f'(x) = \begin{cases} \alpha, & x \leq 0 \\ 1, & x > 0 \end{cases}$$

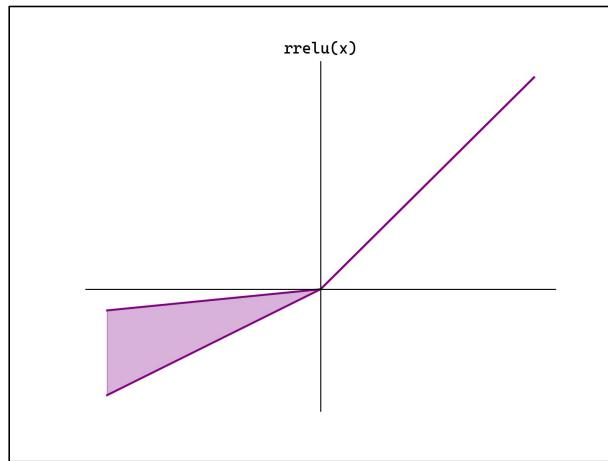
这里 $\alpha$ 是一个可训练的参数，而非固定的常数。



### 4) RReLU (Randomized Leaky ReLU)

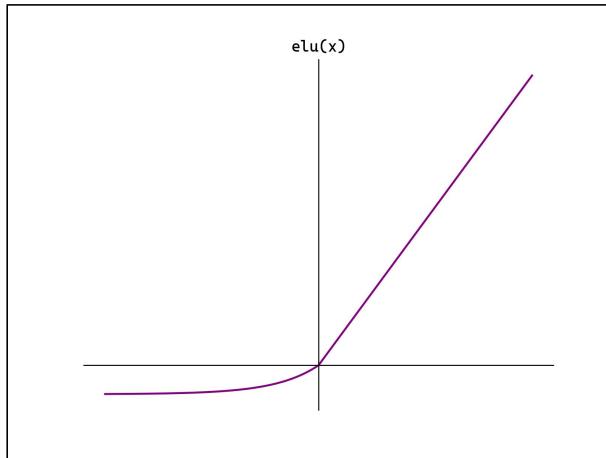
$$f(x) = \begin{cases} \alpha x, & x \leq 0 \\ x, & x > 0 \end{cases}, \quad f'(x) = \begin{cases} \alpha, & x \leq 0 \\ 1, & x > 0 \end{cases}$$

这里 $\alpha$ 是一个在训练时从一个均匀分布中随机选择的参数。



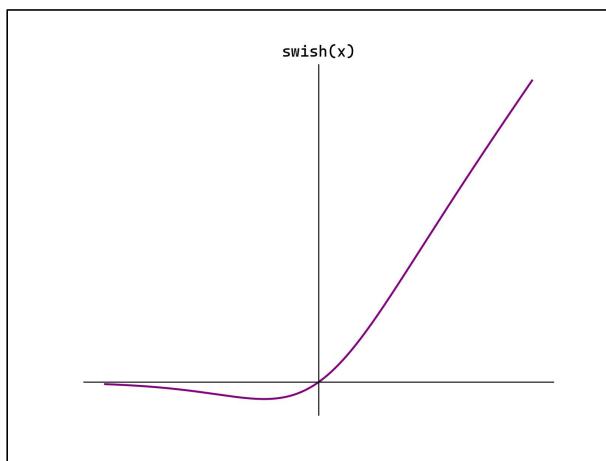
### 5) ELU (Exponential Linear Unit)

$$f(x) = \begin{cases} \alpha(e^x - 1), & x \leq 0 \\ x, & x > 0 \end{cases}, \quad f'(x) = \begin{cases} \alpha e^x, & x \leq 0 \\ 1, & x > 0 \end{cases}$$



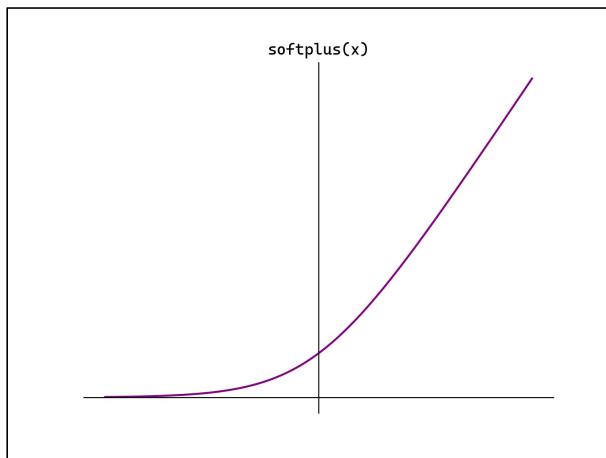
#### 6) Swish (也称 Sigmoid Linear Unit, SiLU)

$$f(x) = \frac{x}{1 + e^{-x}}, \quad f'(x) = \frac{1 + e^{-x} + xe^{-x}}{(1 + e^{-x})^2}$$



#### 7) Softplus

$$f(x) = \ln(1 + e^x), \quad f'(x) = \frac{1}{1 + e^{-x}}$$



## 2.2.8 如何选择激活函数

### 1) 隐藏层

- 首选 ReLU，如果效果不好可尝试 Leaky ReLU 等。
- Sigmoid 在隐藏层易导致梯度消失，应尽量避免。
- Tanh 的输出均值为 0，对中心化数据更友好，但仍可能引发梯度消失，仅适用于浅层网络。

### 2) 输出层

- 二分类选择 Sigmoid。
- 多分类选择 Softmax。
- 回归默认选择 Identity。

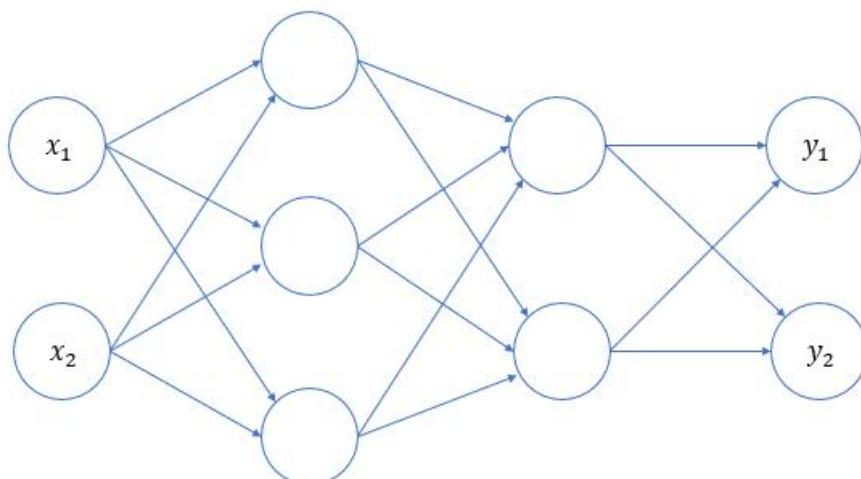
## 2.3 神经网络的简单实现

深度神经网络由多个层（layer）组成，通常将其称之为 **模型（Model）**。整个模型接受原始 **输入（特征）**，生成 **输出（预测）**，并包含一些 **参数**。而在模型内部，每个单独的层都会接受一些输入（由前一层提供），生成输出（到下一层的输入），并包含一组参数；层层向下传递，就可以得到最终的输出值。

神经网络中的参数，就是每一层的权重和偏置。

### 2.3.1 三层神经网络

我们这里以一个三层神经网络为例，实现从输入到输出的处理计算，这个过程就是 **前向传播（forward）**。

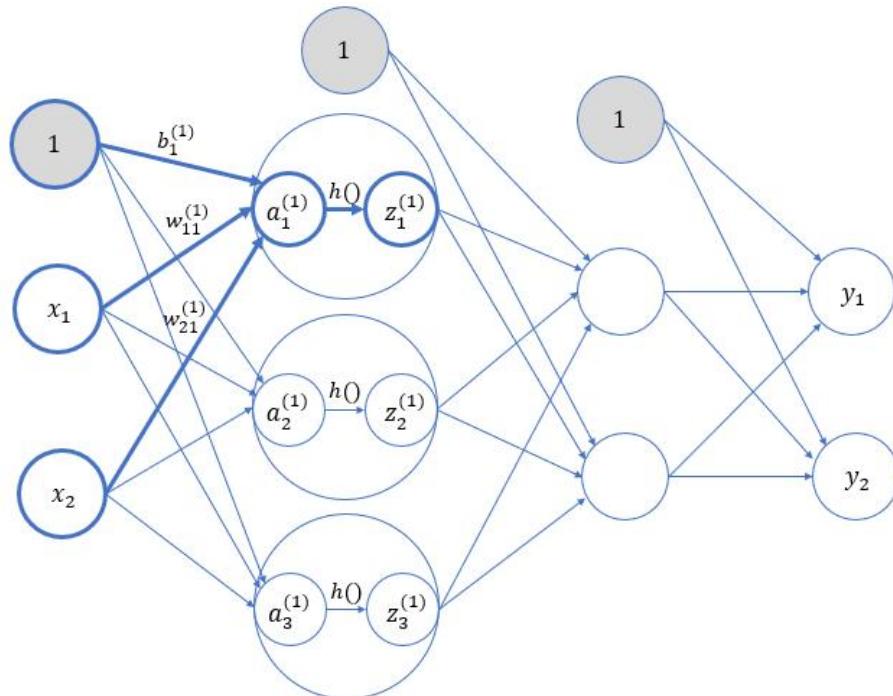


简单起见，我们的输入层（第 0 层）有 2 个神经元；第 1 个隐藏层（第 1 层）有 3 个神经元；第 2 个隐藏层（第 2 层）有 2 个神经元；输出层（第 3 层）有 2 个神经元。

### 2.3.2 各层之间的信号传递

上面只是三层网络的示意图，实际上每层还应该有偏置，各输入信号加权总和还要经过激活函数的处理。接下来逐层进行分析，考察信号在各层之间传递的过程。

#### 1) 输入层（第 0 层）→ 第 1 层



权重和神经元的上标 (1) 表示网络层号。而下标对于神经元来说，就是这一层内的“索引号”；对于权重来说则包含两个数字，分别代表前一层和后一层神经元的索引号。所以， $w_{21}^{(1)}$  就表示这是第 1 层的权重（输入层到第 1 层），并且是从第 2 个输入节点到第 1 层第 1 个节点。偏置的下标只有 1 个，因为前一层的偏置节点只有一个。

所以，可以利用式 (2.4) (2.5) 得到：

$$a_1^{(1)} = w_{11}^{(1)}x_1 + w_{21}^{(1)}x_2 + b_1^{(1)} \quad (2.6)$$

$$z_1^{(1)} = h(a_1^{(1)}) \quad (2.7)$$

同样，对于第 1 层的第 2 个、第 3 个神经元，有：

$$a_2^{(1)} = w_{12}^{(1)}x_1 + w_{22}^{(1)}x_2 + b_2^{(1)}$$

$$a_3^{(1)} = w_{13}^{(1)}x_1 + w_{23}^{(1)}x_2 + b_3^{(1)}$$

我们可以直接写成矩阵乘法的形式：

$$A^{(1)} = XW^{(1)} + B^{(1)} \quad (2.8)$$

其中，

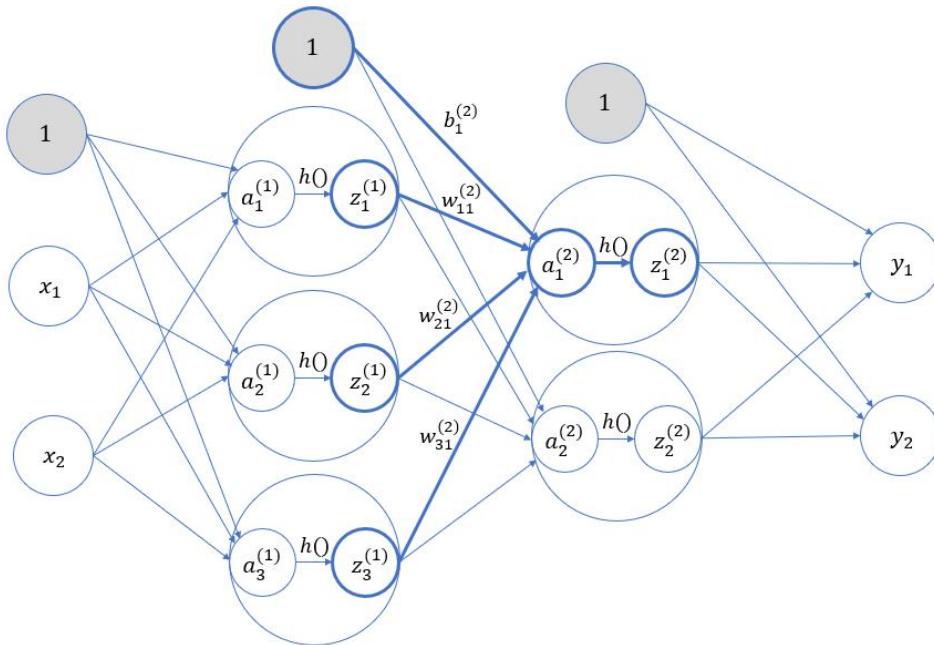
$$A^{(1)} = \begin{pmatrix} a_1^{(1)} & a_2^{(1)} & a_3^{(1)} \end{pmatrix}, X = (x_1 \ x_2), B^{(1)} = \begin{pmatrix} b_1^{(1)} & b_2^{(1)} & b_3^{(1)} \end{pmatrix},$$

$$W^{(1)} = \begin{pmatrix} w_{11}^{(1)} & w_{12}^{(1)} & w_{13}^{(1)} \\ w_{21}^{(1)} & w_{22}^{(1)} & w_{23}^{(1)} \end{pmatrix}$$

可以看到，由于有 2 个输入节点、3 个第 1 层节点，所以全连接层的权重  $W$  就应该是一个  $2 \times 3$  的矩阵。这样，我们就可以很容易地利用 Numpy 中的矩阵乘法计算出输出信号值了。

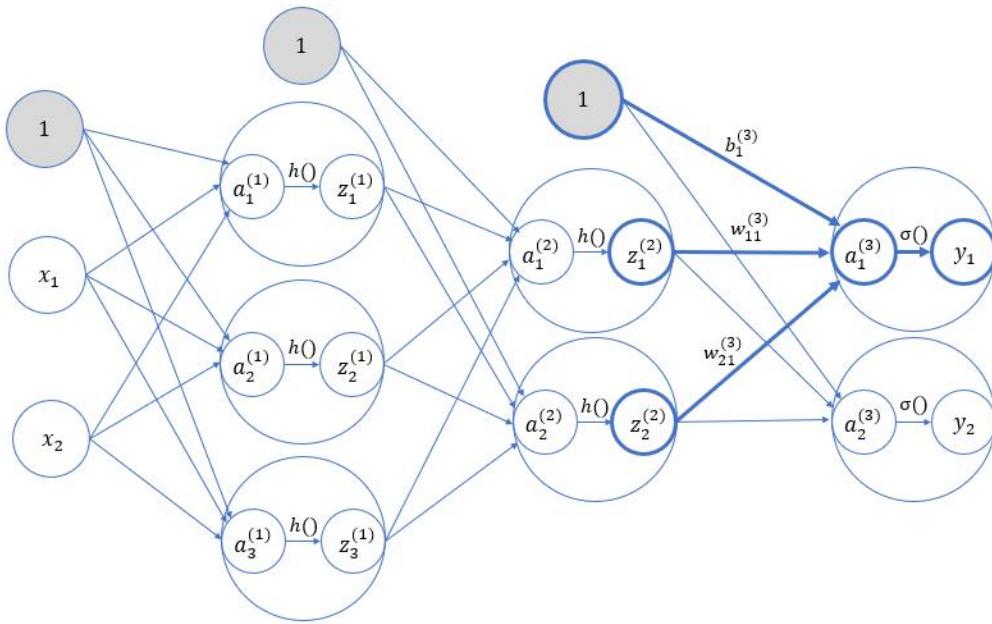
## 2) 第 1 层 → 第 2 层

第 1 层到第 2 层的处理类似，权重参数应该是一个  $3 \times 2$  的矩阵。



## 3) 第 2 层 → 输出层（第 3 层）

第 2 层到输出层的处理也类似，权重参数为  $2 \times 2$  的矩阵；不过输出层的激活函数一般与隐藏层是不同的，这里用  $\sigma()$  表示。



### 2.3.3 代码实现

我们可以将神经网络的所有参数（每一层的权重  $w$  和偏置  $b$ ），保存在一个字典 `network` 中；并定义函数：

- `init_network()`: 对参数进行初始化，每一个权重参数都是一个矩阵（二维），每一个偏置参数则是一个数组（一维）；
- `forward()`: 前向传播，将输入信号转换为输出信号的处理操作。

这里的激活函数，隐藏层用 sigmoid，输出层用 identity（恒等函数）。

具体代码如下：

```
import numpy as np
from common.functions import sigmoid, identity_function

def init_network():
    network = {}
    network['W1'] = np.array([[0.1, 0.3, 0.5], [0.2, 0.4, 0.6]])
    network['b1'] = np.array([0.1, 0.2, 0.3])
    network['W2'] = np.array([[0.1, 0.4], [0.2, 0.5], [0.3, 0.6]])
    network['b2'] = np.array([0.1, 0.2])
    network['W3'] = np.array([[0.1, 0.3], [0.2, 0.4]])
    network['b3'] = np.array([0.1, 0.2])

    return network
```

```
def forward(network, x):
    w1, w2, w3 = network['W1'], network['W2'], network['W3']
    b1, b2, b3 = network['b1'], network['b2'], network['b3']

    a1 = np.dot(x, w1) + b1
    z1 = sigmoid(a1)
    a2 = np.dot(z1, w2) + b2
    z2 = sigmoid(a2)
    a3 = np.dot(z2, w3) + b3
    y = identity_function(a3)

    return y

network = init_network()
x = np.array([1.0, 0.5])
y = forward(network, x)

print(y)
```

## 2.4 应用案例：手写数字识别

我们依然使用 Digit Recognizer 数据集来进行手写数字识别：

<https://www.kaggle.com/competitions/digit-recognizer>。

文件 train.csv 中包含手绘数字（从 0 到 9）的灰度图像，每张图像为  $28 \times 28$  像素，共 784 像素。每个像素有一个 0 到 255 的值表示该像素的亮度。

文件第 1 列为标签，之后 784 列分别为 784 个像素的亮度值。

我们的任务，就是要搭建一个神经网络，实现它的前向传播；也就是要根据输入的数据 ( $28 \times 28 = 784$  数据点表示的图像)，推断出它到底是哪个数字，这个过程也被称为“**推理**”。

这里，我们构建的也是一个三层神经网络，输入层应该有 784 个神经元，输出层有 10 个神经元（表示 0~9 的分类结果）；中间设置 2 个隐藏层，第一个隐藏层有 50 个神经元，第二个隐藏层有 100 个神经元。这里的参数是需要 **学习** 得到的；我们假设已经学习完毕，直接从保存好的文件 nn\_sample 中进行读取即可。

具体代码如下：

```
import numpy as np
import pandas as pd
import joblib
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import MinMaxScaler
from common.functions import sigmoid, softmax

def get_data():
    # 加载数据集
    data = pd.read_csv("../data/train.csv")
    # 划分训练集和测试集
    X = data.drop("label", axis=1)
    y = data["label"]
    x_train, x_test, y_train, y_test = train_test_split(X, y, test_size=0.3)
    # 归一化
    preprocessor = MinMaxScaler()
    x_train = preprocessor.fit_transform(x_train)
    x_test = preprocessor.transform(x_test)
    return x_test, t_test

def init_network():
    # 加载模型
    network = joblib.load("../data/nn_sample")
    return network

def predict(network, x):
    w1, w2, w3 = network['W1'], network['W2'], network['W3']
    b1, b2, b3 = network['b1'], network['b2'], network['b3']

    a1 = np.dot(x, w1) + b1
    z1 = sigmoid(a1)
    a2 = np.dot(z1, w2) + b2
    z2 = sigmoid(a2)
```

```
a3 = np.dot(z2, w3) + b3
y = softmax(a3)

return y

x, t = get_data()
network = init_network()

batch_size = 100 # 批数量
accuracy_cnt = 0

for i in range(0, len(x), batch_size):
    x_batch = x[i:i+batch_size]
    y_batch = predict(network, x_batch)
    p = np.argmax(y_batch, axis=1)
    accuracy_cnt += np.sum(p == t[i:i+batch_size])

print("Accuracy:" + str(float(accuracy_cnt) / len(x)))
```

## 第 3 章 神经网络的学习

神经网络的主要特点，就是可以从数据中进行“学习”。这个学习的过程，就是让训练数据自动决定最优的权重参数。

神经网络（深度学习）也是机器学习的一种；跟传统机器学习方法相比，神经网络不需要人工设置 **特征量**（如 SIFT、HOG 等），这样就可以用同样的流程直接处理所有问题了。

### 3.1 损失函数

神经网络中，需要以某个指标为线索来寻找最优权重参数；这个指标就是 **损失函数**（loss function）。

#### 3.1.1 常见损失函数

##### 1) 均方误差 (MSE)

均方误差 (Mean Squared Error , MSE)，也称 L2 Loss:

$$L = \frac{1}{n} \sum_{i=1}^n (y_i - t_i)^2$$

其中， $y_i$  表示神经网络的输出， $t_i$  表示监督数据的标签（正确的解标签）， $n$  则是数据的“维度”。对于固定维度的网络，前面的系数  $n$  不重要，因此公式有时也可以写成：

$$L = \frac{1}{2} \sum_{i=1}^n (y_i - t_i)^2$$

L2 Loss 对异常值敏感，遇到异常值时易发生梯度爆炸。

代码实现如下：

```
def mean_squared_error(y, t):
    return 0.5 * np.sum((y-t)**2)
```

## 2) 交叉熵误差

除均方误差之外，交叉熵误差（Cross Entropy Error）也经常被用作损失函数：

$$L = - \sum_{i=1}^n t_i \log y_i$$

其中， $\log$  表示自然对数， $y_i$  表示神经网络的输出， $t_i$  表示正确解标签；而且， $t_i$  中只有正确解标签对应的值为 1，其它均为 0（one-hot 表示）。

代码实现如下：

```
def cross_entropy_error(y, t):
    if y.ndim == 1:
        t = t.reshape(1, t.size)
        y = y.reshape(1, y.size)

    # 监督数据是 one-hot 向量的情况下，转换为正确解标签的索引
    if t.size == y.size:
        t = t.argmax(axis=1)

    batch_size = y.shape[0]
    return -np.sum(np.log(y[np.arange(batch_size), t] + 1e-7)) / batch_size
```

### 3.1.2 分类任务损失函数

#### 1) 二分类任务损失函数

二分类任务常用二元交叉熵损失函数（Binary Cross-Entropy Loss）。

$$L = -\frac{1}{n} \sum_{i=1}^n (y_i \log \hat{y}_i + (1 - y_i) \log (1 - \hat{y}_i))$$

其中：

- $y_i$  为真实值（通常为 0 或 1）
- $\hat{y}_i$  为预测值（表示样本  $i$  为 1 的概率）

#### 2) 多分类任务损失函数

多分类任务常用多类交叉熵损失函数（Categorical Cross-Entropy Loss）。它是对每个类别的预测概率与真实标签之间差异的加权平均。

$$L = -\frac{1}{n} \sum_{i=1}^n \sum_{c=1}^C y_{i,c} \log \hat{y}_{i,c}$$

其中：

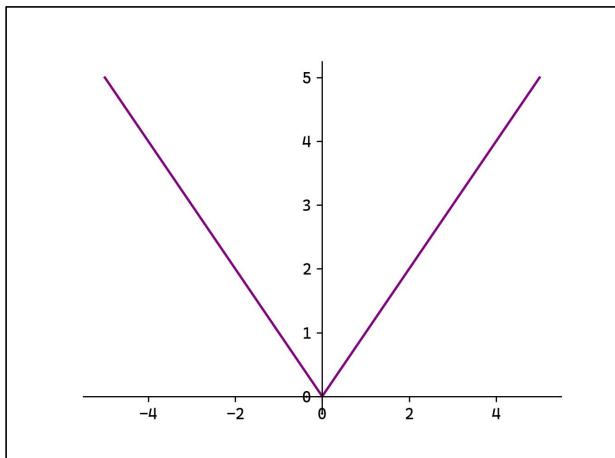
- $C$  是类别数
- $y_{i,c}$  为真实值（表示  $y_i$  是否为类别  $c$ ，通常为 0 或 1）
- $\hat{y}_{i,c}$  为预测值（表示样本  $i$  为类别  $c$  的概率）

### 3.1.3 回归任务损失函数

#### 1) MAE

平均绝对误差（Mean Absolute Error, MAE），也称 L1 Loss:

$$L = \frac{1}{n} \sum_{i=1}^n |y_i - \hat{y}_i|$$

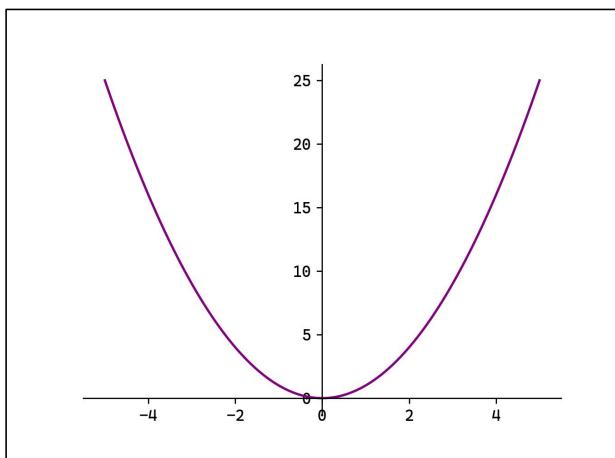


L1 Loss 对异常值鲁棒，但在 0 点处不可导。

## 2) MSE

均方误差 (Mean Squared Error , MSE) , 也称 L2 Loss:

$$L = \frac{1}{n} \sum_{i=1}^n (y_i - \hat{y}_i)^2$$

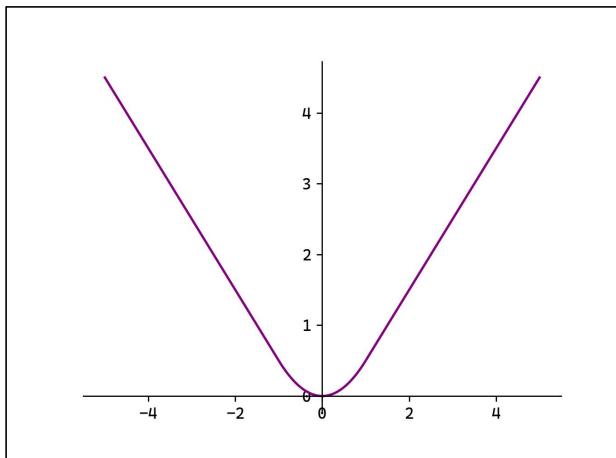


L2 Loss 对异常值敏感，遇到异常值时易发生梯度爆炸。

## 3) Smooth L1

平滑 L1:

$$\text{Smooth L1} = \begin{cases} \frac{1}{2}(y_i - \hat{y}_i)^2, & |y_i - \hat{y}_i| < 1 \\ |y_i - \hat{y}_i| - \frac{1}{2}, & |y_i - \hat{y}_i| \geq 1 \end{cases}$$



当误差较小时( $|y_i - \hat{y}_i| < 1$ )使用 L2 Loss, 使得损失函数平滑可导。当误差较大时( $|y_i - \hat{y}_i| \geq 1$ ) 使用 L1 Loss 降低异常值的影响。

## 3.2 数值微分

损失函数的值越小, 代表我们选取的参数越适合; 想要求得损失函数的最小值, 最基本的想法就是对函数求导, 解出导数值为 0 的点, 并判断它是否为极小值/最小值。

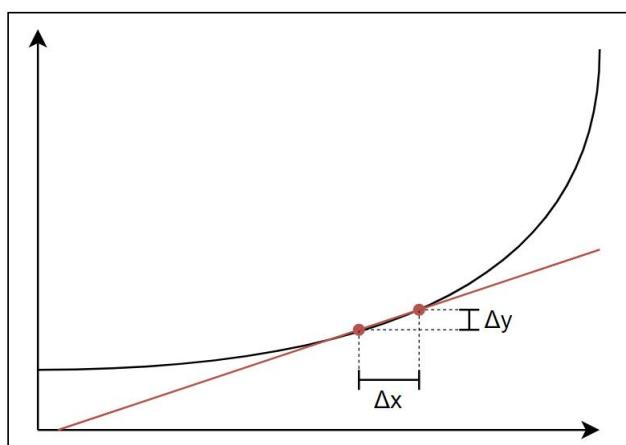
然而, 实际的函数直接求导, 不容易得到解析解。这时可以用数值微分的方式来求某点处的导数, 这在工程上应用非常广泛。

### 3.2.1 导数和数值微分

在数学上, 导数被定义为

$$f'(x) = \frac{df(x)}{dx} = \lim_{h \rightarrow 0} \frac{f(x + h) - f(x)}{h}$$

这个定义中表达出了导数的本质。当  $x$  发生一个微小的变化  $h$  (或者 $\Delta x$ ) 时, 函数值 $f(x)$ 也会发生变化; 当  $h$  趋近于 0 时, 此时 $f(x)$ 的“变化率”就是 $x$ 这一点的导数值。



利用这个定义，我们可以直接以数值计算的方式，利用微小的差分来求函数某点处的导数值，这种方法称为 **数值微分**。

数值微分可以用代码实现非常方便地实现：

```
def numerical_diff(f, x):
    h = 1e-4 # 微小值 0.0001
    return (f(x+h) - f(x-h)) / (2 * h)
```

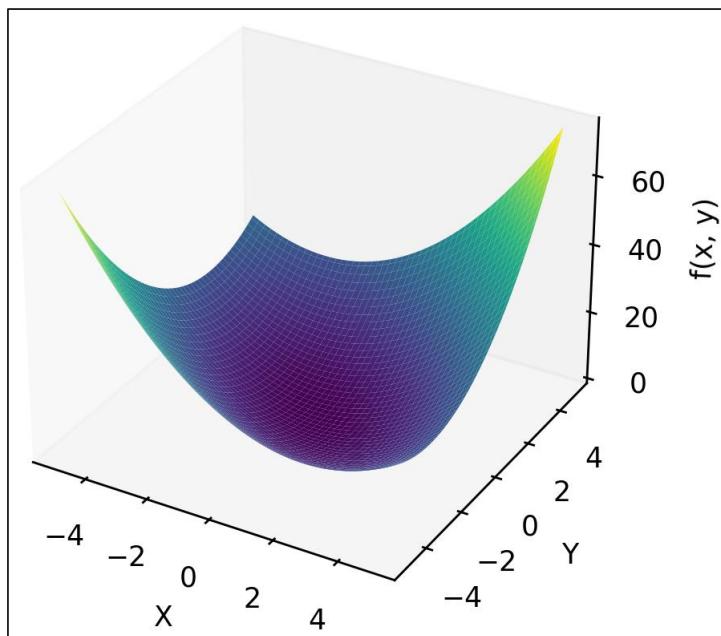
在这里，我们以  $x$  为中心，计算它两边各发生微小变化后的差分，可以避免只计算单向增大时的误差。这种方法称为 **中心差分**。

另外，取微小值  $h$  时不能太小，这会导致计算机浮点数表示的精度不够，出现舍入误差。

### 3.2.2 偏导数

如果函数  $f$  的自变量并非单个元素，而是多个元素，例如：

$$f(x, y) = x^2 + xy + y^2$$



可将其中一个元素  $x$  看作参数，此时  $f$  可看作关于另一元素  $y$  的函数。

$$f_x(y) = x^2 + xy + y^2$$

在  $x = a$  固定的情况下，可计算  $f_x$  关于  $y$  的导数。

$$f_a'(y) = a + 2y$$

这种导数称为偏导数，一般记作：

$$\frac{\partial f}{\partial y}(x, y) = x + 2y$$

更一般地来说，一个多元函数  $f(x_1, x_2, \dots, x_n)$  在点  $(a_1, a_2, \dots, a_n)$  处对  $x_i$  的偏导数定义为：

$$\frac{\partial f}{\partial x_i}(a_1, a_2, \dots, a_n) = \lim_{\Delta x_i \rightarrow 0} \frac{f(a_1, \dots, a_i + \Delta x_i, \dots, a_n) - f(a_1, \dots, a_i, \dots, a_n)}{\Delta x_i}$$

偏导数同样也可以用数值微分的方式来求，即只改变一个自变量、其它不变，做差分考察函数值的变化率。

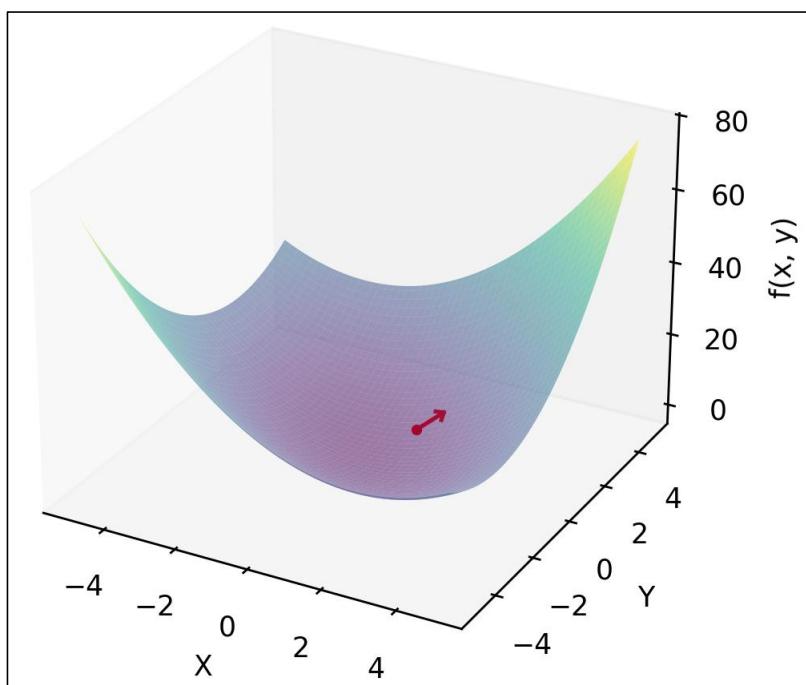
### 3.2.3 梯度

多元函数  $f(x_1, \dots, x_n)$  关于每个变量  $x_i$  都有偏导数  $\frac{\partial f}{\partial x_i}$ ，在点  $a$  处，这些偏导数定义了一个向量。

$$\nabla f(a) = \left[ \frac{\partial f}{\partial x_1}(a), \dots, \frac{\partial f}{\partial x_n}(a) \right]$$

这个向量称为  $f$  在点  $a$  的梯度。

例如：  $f(x, y) = x^2 + xy + y^2$  在  $(1, 1)$  处的梯度为  $[3, 3]$ 。



在函数的极小值、极大值和鞍点处，梯度为 0。

需要注意的是，梯度代表的其实是函数值增大最快的方向；在实际应用中，我们需要寻找损失函数的最小值，所以一般选择 **负梯度** 向量。同样地，负梯度代表的是函数值减小最快的方向，并不一定直接指向函数图像的最低点。

利用数值微分，我们可以在代码中实现梯度的计算：

```
def _numerical_gradient(f, x):
    h = 1e-4 # 0.0001
    grad = np.zeros_like(x)

    for idx in range(x.size):
        tmp_val = x[idx]
        x[idx] = float(tmp_val) + h
        fxh1 = f(x) # f(x+h)

        x[idx] = tmp_val - h
        fxh2 = f(x) # f(x-h)
        grad[idx] = (fxh1 - fxh2) / (2*h)

        x[idx] = tmp_val # 还原值

    return grad
```

### 3.3 神经网络的梯度计算

在神经网络的学习中，梯度的计算非常重要。神经网络中的梯度，指的就是损失函数关于权重参数的梯度。

我们以一个单层的简单网络为例，形状为  $2 \times 3$ ，权重参数为  $W$ ，损失函数记为  $L$ 。那么它的权重参数和梯度为：

$$W = \begin{pmatrix} w_{11} & w_{12} & w_{13} \\ w_{21} & w_{22} & w_{23} \end{pmatrix}$$

$$\frac{\partial L}{\partial W} = \begin{pmatrix} \frac{\partial L}{\partial w_{11}} & \frac{\partial L}{\partial w_{12}} & \frac{\partial L}{\partial w_{13}} \\ \frac{\partial L}{\partial w_{21}} & \frac{\partial L}{\partial w_{22}} & \frac{\partial L}{\partial w_{23}} \end{pmatrix}$$

这里，梯度  $\frac{\partial L}{\partial W}$  也是一个  $2 \times 3$  的矩阵，其中各个元素由  $L$  关于  $W$  中各元素的偏导数构成。

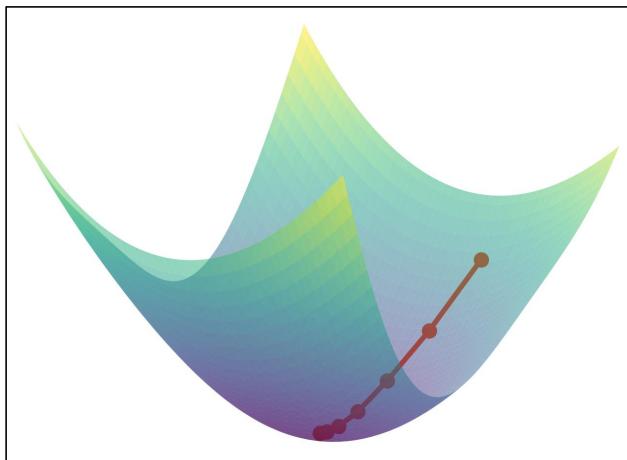
计算这个简单网络的梯度，可以用代码实现如下：

```
class simpleNet:  
    def __init__(self):  
        self.W = np.random.randn(2,3)  
  
    def predict(self, x):  
        return np.dot(x, self.W)  
  
    def loss(self, x, t):  
        z = self.predict(x)  
        y = softmax(z)  
        loss = cross_entropy_error(y, t)  
  
        return loss  
  
x = np.array([0.6, 0.9])  
t = np.array([0, 0, 1])  
  
net = simpleNet()  
  
f = lambda w: net.loss(x, t)  
dW = numerical_gradient(f, net.W)  
  
print(dW)
```

## 3.4 随机梯度下降法 (SGD)

### 3.4.1 梯度下降法

**梯度下降法 (Gradient Descent)** 是一种用于最小化目标函数的迭代优化算法。核心是沿着目标函数（如损失函数）的负梯度方向逐步调整参数，从而逼近函数的最小值。梯度方向指示了函数增长最快的方向，因此负梯度方向是函数下降最快的方向。



具体来说，我们初始找到函数 $f(x_1, x_2)$ 的一个点 $(x_1, x_2)$ ，按下式进行更新：

$$\begin{aligned} \dot{x}_1 &= x_1 - \eta \frac{\partial f}{\partial x_1} \\ \dot{x}_2 &= x_2 - \eta \frac{\partial f}{\partial x_2} \end{aligned}$$

这样就可以沿着负梯度方向，找到一个新的点 $(\dot{x}_1, \dot{x}_2)$ ，让函数值更小。

这里的 $\eta$ 表示每次的更新量，在神经网络的学习过程中，就代表了一次学习的步长（一次学习多少、多大程度去更新参数），称为 **学习率 (learning rate)**。学习率需要预先设定好，过大或过小都会导致学习效果不佳。

梯度下降法可以代码实现如下：

```
def gradient_descent(f, init_x, lr=0.01, step_num=100):
    x = init_x
    x_history = []

    for i in range(step_num):
        x_history.append( x.copy() )

        grad = numerical_gradient(f, x)
        x -= lr * grad

    return x, np.array(x_history)
```

### 3.4.2 模型训练相关概念

#### 1) Epoch

1 个 Epoch 表示模型完整遍历一次整个训练数据集的过程。例如，训练 10 个 Epoch 表示模型将整个数据集反复学习 10 次。

模型需要多次遍历数据集（多个 Epoch）才能逐步学习数据中的模式，单次遍历数据集（1 个 Epoch）通常不足以让模型收敛，多次遍历可以逐步优化模型参数。

## 2) Batch Size

Batch Size 是每次训练时输入的样本数量。例如，Batch Size=32 表示每次用 32 个样本计算一次梯度并更新模型参数。

小批量数据计算梯度比单样本（Batch Size=1）更稳定，比全批量（Batch Size=全体数据）更高效。并且较小的 Batch Size 可能带来更多噪声，有助于模型泛化。

## 3) Iteration

一次 Iteration 表示完成一个 Batch 数据的正向传播（预测）和反向传播（更新参数）的过程。

例如，数据集现有 2000 个样本，对其训练 10 个 Epoch，选择 Batch Size=64：

Batch 个数为  $2000//64+1=31+1=32$  个（最后一个 Batch 仅有 16 个样本）。

每个 Epoch 中迭代次数 Iteration=32 次。

总迭代次数为  $10 \times 32 = 320$  次。

总训练样本数为  $10 \times 2000 = 20000$ 。

### 3.4.3 SGD

在神经网络的学习过程中，可以使用梯度下降法来更新参数，目标就是减小损失函数的值。

实际操作时，一般会从训练数据中随机选择一个小批量数据（mini-batch），然后用梯度下降法迭代多个轮次（iteration）；这种“对随机选择的数据进行的梯度下降法”，被称作 **随机梯度下降法（stochastic gradient descent, SGD）**。

具体过程如下：

#### 1) 随机选择批数据（mini-batch）

从训练数据中随机选出一部分数据，学习的目标就是要减少这个 mini-batch 数据的损失函数值。

#### 2) 计算梯度

对当前的各权重参数，计算出梯度的值，负梯度就表示了损失函数减小最多的方向。

### 3) 更新参数

按照 3.4.1 节中梯度下降法的公式，对权重参数沿负梯度方向进行微小更新。

### 4) 重复迭代

重复上面的步骤 1) 2) 3) ， 直到完成预定的总迭代次数。

## 3.5 综合代码实现

### 应用案例：手写数字识别

我们还是考察之前手写数字识别的案例。方便起见，这次只实现一个 2 层的神经网络(中间只有 1 个隐藏层、后面是输出层)，利用之前的数据集来进行学习，学习方法采用 SGD。

首先我们先来实现一个 TwoLayerNet 类：

```
from common.functions import *
from common.gradient import numerical_gradient

class TwoLayerNet:

    def __init__(self, input_size, hidden_size, output_size,
weight_init_std=0.01):
        # 初始化权重
        self.params = {}
        self.params['W1'] = weight_init_std * np.random.randn(input_size,
hidden_size)
        self.params['b1'] = np.zeros(hidden_size)
        self.params['W2'] = weight_init_std * np.random.randn(hidden_size,
output_size)
        self.params['b2'] = np.zeros(output_size)

    def predict(self, x):
        W1, W2 = self.params['W1'], self.params['W2']
        b1, b2 = self.params['b1'], self.params['b2']

        a1 = np.dot(x, W1) + b1
        z1 = sigmoid(a1)
        a2 = np.dot(z1, W2) + b2
        y = softmax(a2)
```

```
    return y

# x: 输入数据, t: 监督数据
def loss(self, x, t):
    y = self.predict(x)

    return cross_entropy_error(y, t)

def accuracy(self, x, t):
    y = self.predict(x)
    y = np.argmax(y, axis=1)
    t = np.argmax(t, axis=1)

    accuracy = np.sum(y == t) / float(x.shape[0])
    return accuracy

# x: 输入数据, t: 监督数据
def numerical_gradient(self, x, t):
    loss_W = lambda W: self.loss(x, t)

    grads = {}
    grads['W1'] = numerical_gradient(loss_W, self.params['W1'])
    grads['b1'] = numerical_gradient(loss_W, self.params['b1'])
    grads['W2'] = numerical_gradient(loss_W, self.params['W2'])
    grads['b2'] = numerical_gradient(loss_W, self.params['b2'])

    return grads
```

然后我们再利用 SGD 对神经网络进行训练学习：

```
import numpy as np
import matplotlib.pyplot as plt
from two_layer_net import TwoLayerNet

# 读入数据
x_train, x_test, t_train, t_test = get_data()

network = TwoLayerNet(input_size=784, hidden_size=50, output_size=10)
```

更多 Java -大数据 -前端 -python 人工智能资料下载，可百度访问：尚硅谷官网

```
iters_num = 10000 # 适当设定循环的次数
train_size = x_train.shape[0]
batch_size = 100
learning_rate = 0.1

train_loss_list = []
train_acc_list = []
test_acc_list = []

iter_per_epoch = max(train_size / batch_size, 1)

for i in range(iters_num):
    batch_mask = np.random.choice(train_size, batch_size)
    x_batch = x_train[batch_mask]
    t_batch = t_train[batch_mask]

    # 计算梯度
    grad = network.numerical_gradient(x_batch, t_batch)

    # 更新参数
    for key in ('W1', 'b1', 'W2', 'b2'):
        network.params[key] -= learning_rate * grad[key]

    loss = network.loss(x_batch, t_batch)
    train_loss_list.append(loss)

    if i % iter_per_epoch == 0:
        train_acc = network.accuracy(x_train, t_train)
        test_acc = network.accuracy(x_test, t_test)
        train_acc_list.append(train_acc)
        test_acc_list.append(test_acc)
        print("train acc, test acc | " + str(train_acc) + ", " + str(test_acc))

# 绘制图形
markers = {'train': 'o', 'test': 's'}
x = np.arange(len(train_acc_list))
```

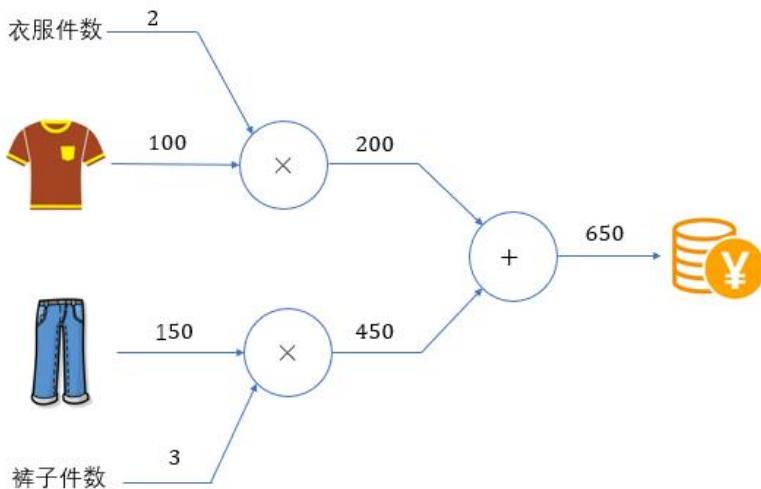
```
plt.plot(x, train_acc_list, label='train acc')
plt.plot(x, test_acc_list, label='test acc', linestyle='--')
plt.xlabel("epochs")
plt.ylabel("accuracy")
plt.ylim(0, 1.0)
plt.legend(loc='lower right')
plt.show()
```

## 第 4 章 反向传播算法

反向传播（Backward Propagation 或 Back Propagation，BP 算法）指的是计算神经网络参数梯度的方法。简言之，该方法根据微积分中的链式法则，按相反的顺序从输出层到输入层遍历网络。该算法存储了计算某些参数梯度时所需的任何中间变量。

### 4.1 计算图

计算图将计算过程用图表示出来。这里说的图是数据结构中的图，通过多个节点和边表示（连接节点的直线称为边）。

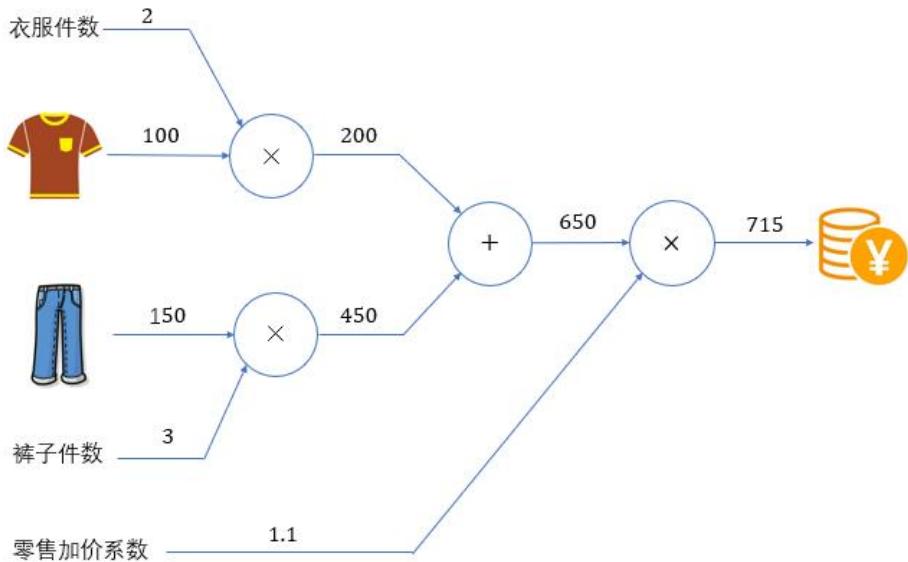


如上就是  $100 \times 2 + 150 \times 3 = 650$  的计算图表示。

计算图的基本计算原则，就是从输入出发、按照箭头方向，从左到右依次进行计算，最终得到输出结果。这个过程，其实就是 **前向传播（forward）**。

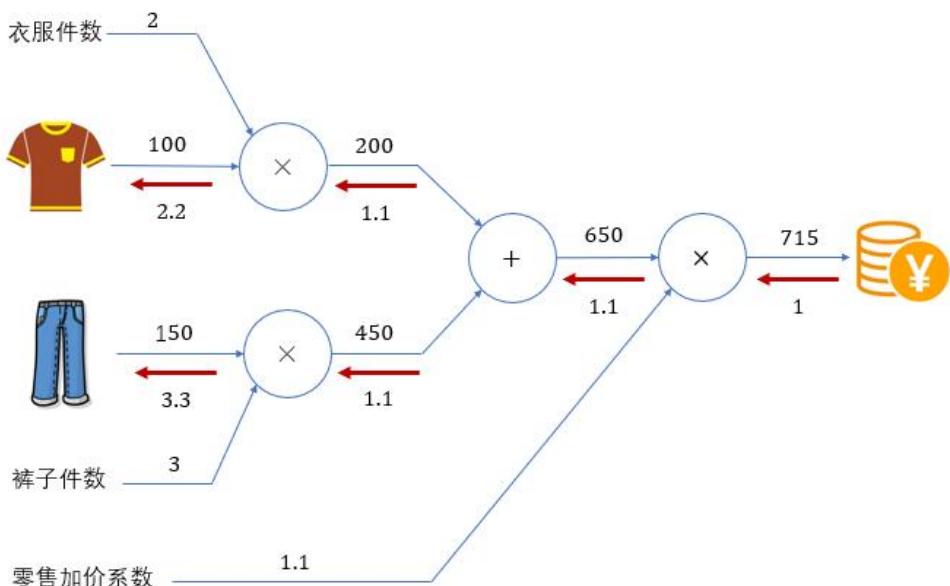
计算图的特点是可以通过传递“局部计算”获得最终结果。即只需根据与自己相关的信息输出接下来的结果。无论全局的计算有多么复杂，各个节点所要做的就是进行局部计算并传递计算结果，最终得出全局的复杂计算的结果。

如果增加更多的计算环节，比如再乘以一个“零售加价系数”，计算图如下所示。



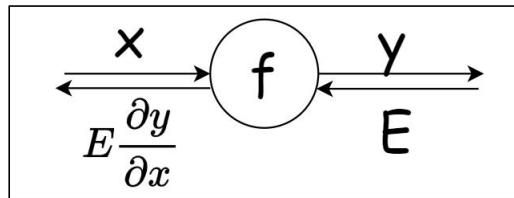
如果我们进一步考虑，当衣服的价格上涨（输入变化）时，会多大程度上影响最后要支付的金额（输出结果）？

将输入的衣服价格记为  $x$ ，输出的支付金额记为  $L$ ，这其实就是要要求导数值  $\frac{\partial L}{\partial x}$ 。在计算图上，我们可以利用反向（从右到左）的传递来方便地计算导数。这个过程，就可以叫做**反向传播（backward）**。



## 4.2 链式法则

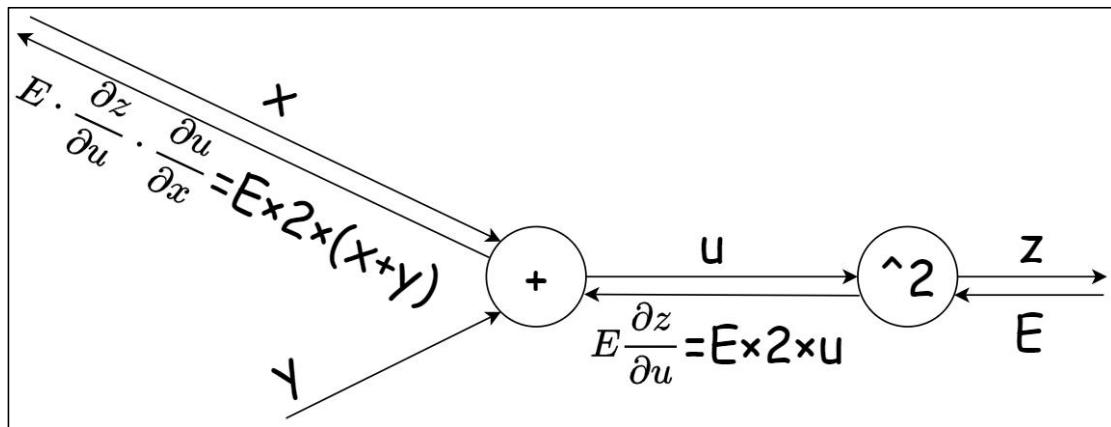
反向传播将局部导数向反方向传递，传递的原理基于链式法则。反向传播时将信号乘以节点的局部导数然后传递给下一个节点。



对于复合函数  $z = (x + y)^2$ , 令  $u = x + y$ , 则

$$\frac{\partial z}{\partial x} = \frac{\partial z}{\partial u} \frac{\partial u}{\partial x} = 2u \times 1 = 2(x + y)$$

现用计算图表示：

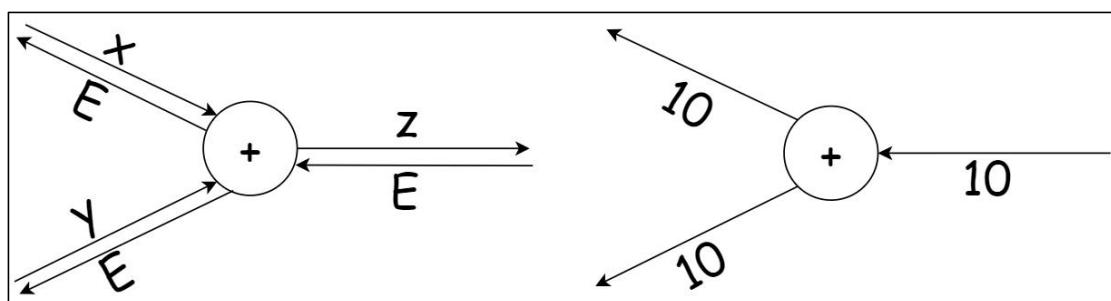


## 4.3 反向传播

利用计算图的反向传播，可以很容易地计算出输出关于输入的偏导数。

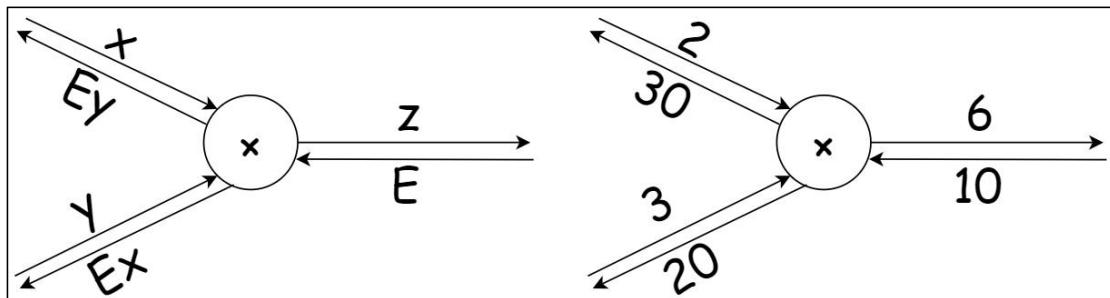
### 4.3.1 加法节点的反向传播

对于  $z = x + y$ ,  $\frac{\partial z}{\partial x} = 1$ ,  $\frac{\partial z}{\partial y} = 1$ 。因此加法的反向传播会将上游传来的值原样向下游传递。



### 4.3.2 乘法节点的反向传播

对于  $z = xy$ ,  $\frac{\partial z}{\partial x} = y$ ,  $\frac{\partial z}{\partial y} = x$ 。因此乘法的反向传播会将上游传来的值乘以输入的翻转向下游传递。



## 4.4 激活层的反向传播和实现

现在将计算图应用到神经网络中。神经网络中一步重要的计算操作就是激活函数，下面就来讨论各种激活函数的反向传播，基于这些我们就可以用代码实现激活层的完整功能了。

### 4.4.1 ReLU 的反向传播

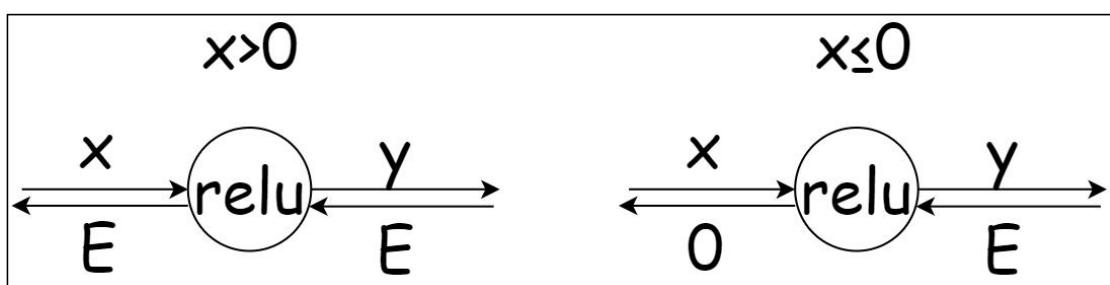
对于 ReLU 函数：

$$f(x) = \max(0, x) = \begin{cases} 0, & x \leq 0 \\ x, & x > 0 \end{cases}$$

其导数为：

$$f'(x) = \begin{cases} 0, & x \leq 0 \\ 1, & x > 0 \end{cases}$$

分为  $x \leq 0$  和  $x > 0$  两种情况分别讨论，反向传播的计算图如下：



ReLU 层可以在代码中实现为一个类 Relu:

```
class Relu:
    def __init__(self):
        self.mask = None
```

```
def forward(self, x):
    self.mask = (x <= 0)
    out = x.copy()
    out[self.mask] = 0

    return out

def backward(self, dout):
    dout[self.mask] = 0
    dx = dout

    return dx
```

#### 4.4.2 Sigmoid 的反向传播

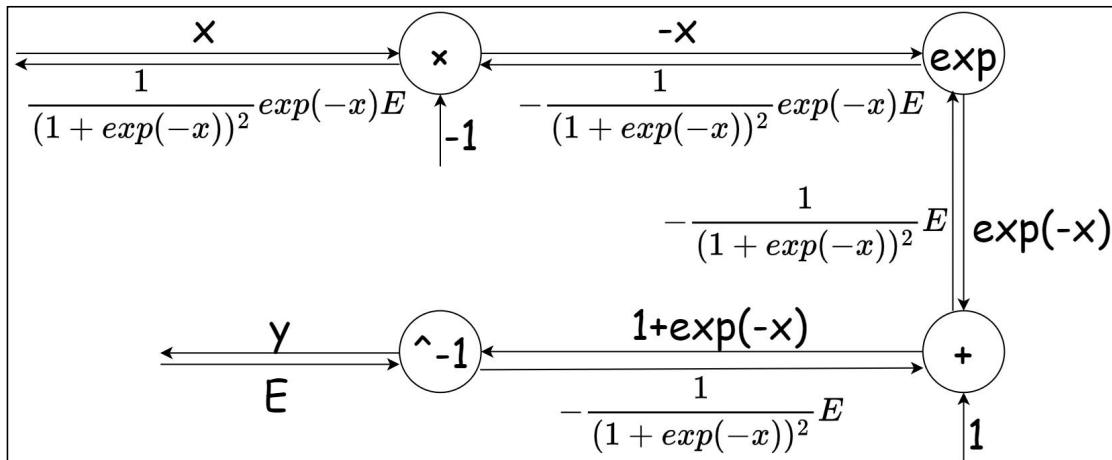
对于 Sigmoid 函数：

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

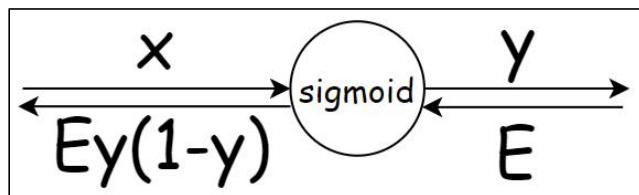
其导数为：

$$\begin{aligned} f'(x) &= -\left(\frac{1}{1 + e^{-x}}\right)^2 \cdot e^{-x} \cdot (-1) = \frac{e^{-x}}{(1 + e^{-x})^2} \\ &= \frac{1}{1 + e^{-x}} \left(1 - \frac{1}{1 + e^{-x}}\right) \\ &= f(x)(1 - f(x)) \end{aligned}$$

利用计算图的反向传播，也可以得到相同的结果：



简化得：



Sigmoid 层可以在代码中实现为一个类 Sigmoid:

```
class Sigmoid:
    def __init__(self):
        self.out = None

    def forward(self, x):
        out = sigmoid(x)
        self.out = out
        return out

    def backward(self, dout):
        dx = dout * (1.0 - self.out) * self.out

        return dx
```

## 4.5 Affine 的反向传播和实现

在全连接层（Fully Connected Layer, Dense Layer）中，每个输入节点与输出节点相连，通过权重矩阵和偏置进行线性变换，这种操作在几何领域称为仿射变换（Affine

transformation, 几何中, 仿射变换包括一次线性变换和一次平移, 分别对应神经网络的加权求和运算与加偏置运算)。

考虑 N 个数据一起进行正向传播的情况, 写成矩阵计算形式:

$$Y = XW + B$$

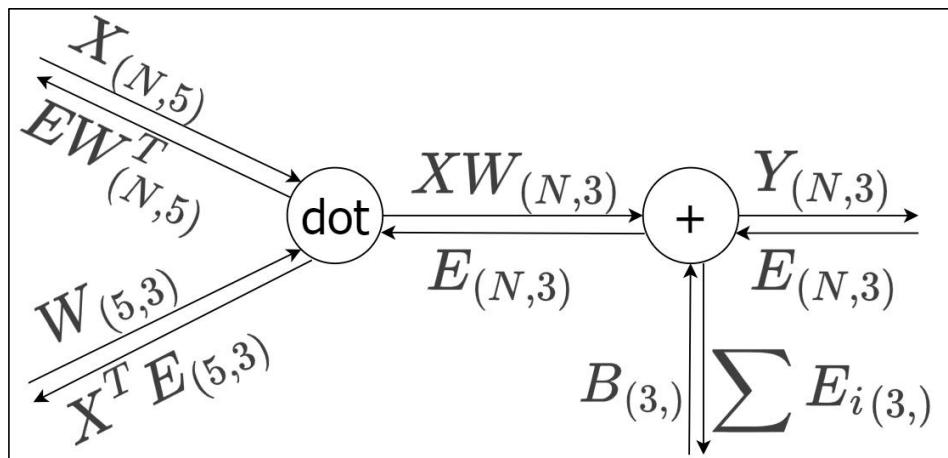
这里的  $X$  是形状为  $N \times m$  的矩阵,  $m$  就是 Affine 层输入神经元的个数; 而  $W$  是形状为  $m \times n$  的权重矩阵,  $n$  就是 Affine 层输出神经元的个数。

根据矩阵求导的运算法则, 可以得到损失函数  $L$  关于  $X$ 、 $W$  的偏导数:

$$\frac{\partial L}{\partial X} = \frac{\partial L}{\partial Y} \cdot W^T$$

$$\frac{\partial L}{\partial W} = X^T \cdot \frac{\partial L}{\partial Y}$$

用计算图的反向传播计算如下:



这里令  $E = \frac{\partial L}{\partial Y}$ , 需要注意矩阵的形状要满足矩阵乘法的要求。

Affine 层可以在代码中实现为一个类 Affine:

```
class Affine:
    def __init__(self, w, b):
        self.W = w
        self.b = b

        self.x = None
        self.original_x_shape = None
        # 权重和偏置参数的导数
```

```
self.dW = None
self.db = None

def forward(self, x):
    # 对应张量
    self.original_x_shape = x.shape
    x = x.reshape(x.shape[0], -1)
    self.x = x

    out = np.dot(self.x, self.W) + self.b

    return out

def backward(self, dout):
    dx = np.dot(dout, self.W.T)
    self.dW = np.dot(self.x.T, dout)
    self.db = np.sum(dout, axis=0)

    dx = dx.reshape(*self.original_x_shape)

    return dx
```

## 4.6 输出层的反向传播和实现

在输出层，我们一般使用 Softmax 作为激活函数。

对于 Softmax 函数：

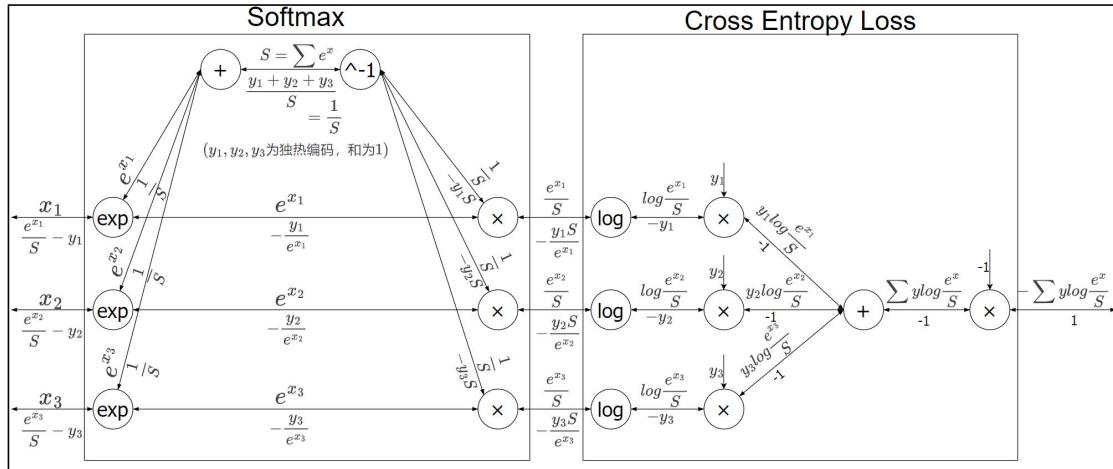
$$y_k = \frac{e^{x_k}}{\sum_{i=1}^n e^{x_i}}, k = 1 \sim n$$

其偏导数为：

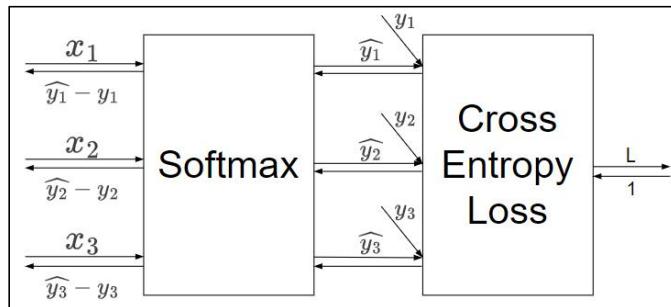
$$\frac{\partial y_k}{\partial x_i} = \begin{cases} y_k(1 - y_i), & k = i \\ -y_k y_i, & k \neq i \end{cases}$$

而对于输出层，一般会直接将结果代入损失函数的计算。对于我们之前介绍的分类问题，这里选择交叉熵误差(Cross Entropy Error)作为损失函数，就可以得到一个 Softmax-with-Loss 层，它包含了 Softmax 和 Cross Entropy Loss 两部分。

导数的计算会比较复杂，可以用计算图表示如下：



简化得：



在代码中可以实现为一个类 SoftmaxWithLoss:

```
class SoftmaxWithLoss:
    def __init__(self):
        self.loss = None
        self.y = None # softmax 的输出
        self.t = None # 监督数据

    def forward(self, x, t):
        self.t = t
        self.y = softmax(x)
        self.loss = cross_entropy_error(self.y, self.t)

    return self.loss

    def backward(self, dout=1):
```

```
batch_size = self.t.shape[0]
if self.t.size == self.y.size: # 监督数据是 one-hot-vector 的情况
    dx = (self.y - self.t) / batch_size
else:
    dx = self.y.copy()
    dx[np.arange(batch_size), self.t] -= 1
    dx = dx / batch_size

return dx
```

## 4.7 反向传播的综合代码实现

现在可以在之前手写数字识别案例的基础上，对 SGD 的计算过程进行优化。核心就是使用误差的反向传播法来计算梯度，而不是使用差分数值计算；这将大大提升学习的效率。

对于二层网络 TwoLayerNet，隐藏层由一个 Affine 层和一个 ReLU 层组成，输出层则由一个 Affine 层和一个 Softmax-with-Loss 层组成。由于之前已经实现了各层的类，现在只要用“搭积木”的方式将它们拼接在一起就可以了。

将 TwoLayerNet 类的代码实现改进如下：

```
import numpy as np
from common.layers import *
from common.gradient import numerical_gradient
from collections import OrderedDict

class TwoLayerNet:

    def __init__(self, input_size, hidden_size, output_size,
weight_init_std = 0.01):
        # 初始化权重
        self.params = {}
        self.params['W1'] = weight_init_std * np.random.randn(input_size,
hidden_size)
        self.params['b1'] = np.zeros(hidden_size)
        self.params['W2'] = weight_init_std * np.random.randn(hidden_size,
output_size)
        self.params['b2'] = np.zeros(output_size)
```

```
# 生成层
self.layers = OrderedDict()
self.layers['Affine1'] = Affine(self.params['W1'],
self.params['b1'])
self.layers['Relu1'] = Relu()
self.layers['Affine2'] = Affine(self.params['W2'],
self.params['b2'])

self.lastLayer = SoftmaxWithLoss()

def predict(self, x):
    for layer in self.layers.values():
        x = layer.forward(x)

    return x

# x:输入数据, t:监督数据
def loss(self, x, t):
    y = self.predict(x)
    return self.lastLayer.forward(y, t)

def accuracy(self, x, t):
    y = self.predict(x)
    y = np.argmax(y, axis=1)
    if t.ndim != 1 : t = np.argmax(t, axis=1)

    accuracy = np.sum(y == t) / float(x.shape[0])
    return accuracy

# x:输入数据, t:监督数据
def numerical_gradient(self, x, t):
    loss_W = lambda W: self.loss(x, t)

    grads = {}
    grads['W1'] = numerical_gradient(loss_W, self.params['W1'])
    grads['b1'] = numerical_gradient(loss_W, self.params['b1'])
    grads['W2'] = numerical_gradient(loss_W, self.params['W2'])
```

```
grads['b2'] = numerical_gradient(loss_W, self.params['b2'])

return grads

def gradient(self, x, t):
    # forward
    self.loss(x, t)

    # backward
    dout = 1
    dout = self.lastLayer.backward(dout)

    layers = list(self.layers.values())
    layers.reverse()
    for layer in layers:
        dout = layer.backward(dout)

    # 设定
    grads = {}
    grads['W1'], grads['b1'] = self.layers['Affine1'].dW,
self.layers['Affine1'].db
    grads['W2'], grads['b2'] = self.layers['Affine2'].dW,
self.layers['Affine2'].db

return grads
```

接下来的训练过程，跟之前是一样的，只需要调用新的梯度计算函数即可。

## 第 5 章 学习的技巧

### 5.1 深度神经网络及其问题

#### 5.1.1 深度学习

当神经网络的层数加深时，往往可以有效地提高识别精度；特别是对于大规模的复杂问题，通常都需要用更多的层来分层次传递信息，而且可以有效减少网络的参数数量，从而使得学习更加高效。使用深度神经网络进行的学习就被称为 **深度学习（Deep Learning）**。

ILSVRC 是近年来机器视觉领域最受追捧且最具权威的学术竞赛之一，代表了图像领域的最高水平。它包含多个测试项目，其中最有名的就是“类别分类”（Classification），该项目会进行 1000 个类别的分类，比试识别精度。自从 2012 年以后的 ILSVRC 比赛，优胜队伍采用的方法基本上都是以深度学习为基础的。

可以说，深度学习代表了目前人工智能领域的发展方向，正以超乎想象的速度发展并改变我们的生活。

### 5.1.2 梯度消失和梯度爆炸

在某些神经网络中，随着网络深度的增加，梯度在隐藏层反向传播时倾向于变小。这就意味着，前面隐藏层中的神经元要比后面的学习起来更慢。这种现象被称为“**梯度消失**”。

与之对应，如果我们进行一些特殊的调整（比如初始权重很大），可以让梯度反向传播时不会明显减小，从而解决梯度消失的问题；然而这样一来，前面层的梯度又会变得非常大，引起网络不稳定，无法再从训练数据中学习。这种现象被称为“**梯度爆炸**”。

基于梯度学习的深度神经网络中，梯度本身是不稳定的，前面层中的梯度可能“消失”，也可能“爆炸”。

对于梯度消失和梯度爆炸，一个容易理解的解释是：当反向传播进行很多层的时候，每一层都对前一层梯度乘以了一个系数；因此当这个系数比较小（小于 1）时，越往前传递，梯度就会越小、训练越慢，导致梯度消失；而如果这个系数比较大，则越往前传递梯度就会越大，导致梯度爆炸。

所以，深度神经网络的训练是比较复杂的，会有一系列的问题。研究表明，激活函数的选择、权重的初始化，甚至学习算法的实现方式都是影响因素；另外，网络的架构和其它一些超参数也有重要影响。

为了让深度神经网络的学习更加稳定、高效，我们需要考虑进一步改进寻找最优参数的方法，以及如何设置参数初始值、如何设定超参数；此外还应该解决过拟合的问题。

## 5.2 更新参数方法的优化

对于深度神经网络的学习，之前更新参数的方法是随机梯度下降法（SGD）。这种方法比较简单，也非常经典，但在很多具体问题中并不是最高效的。

### 5.2.1 SGD 的缺点

SGD 的更新方式可以写为：

$$W \leftarrow W - \eta \frac{\partial L}{\partial W}$$

这里  $W$  是要更新的权重参数（矩阵），损失函数  $L$  关于  $W$  的梯度就是  $\frac{\partial L}{\partial W}$ ；而  $\eta$  表示 **学习率**。

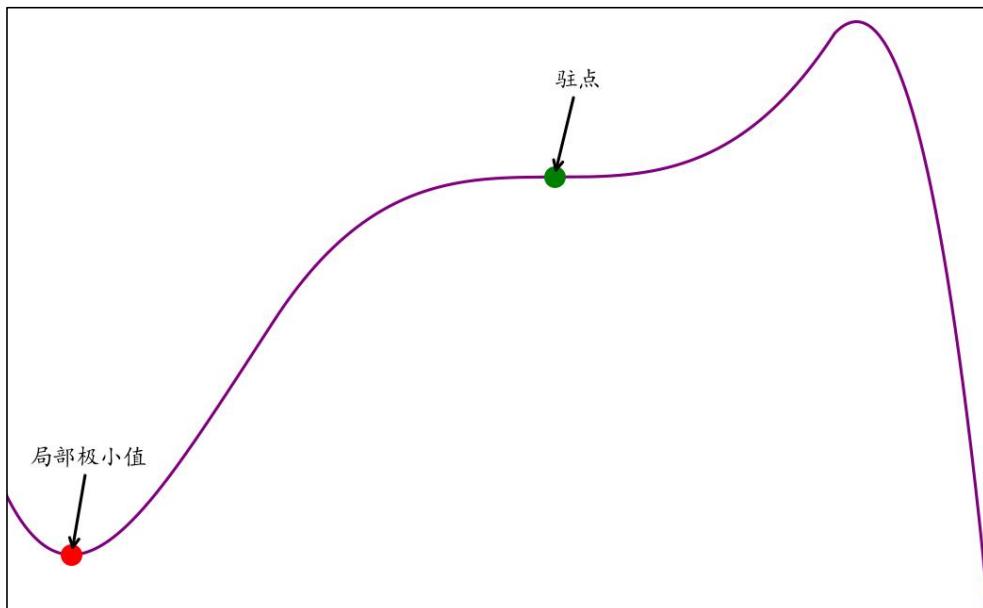
我们可以将 SGD 单独包装成一个类：

```
class SGD:
    def __init__(self, lr=0.01):
        self.lr = lr

    def update(self, params, grads):
        for key in params.keys():
            params[key] -= self.lr * grads[key]
```

SGD 有以下问题：

- 局部最优解：陷入局部最优，尤其在非凸函数中，难以找到全局最优解。
- 鞍点：陷入鞍点，梯度为 0，导致训练停滞。
- 收敛速度慢：高维或非凸函数中，收敛速度较慢。
- 学习率选择：学习率过大导致震荡或不收敛，过小则收敛速度慢。



## 5.2.2 Momentum

原始的梯度下降法直接使用当前梯度来更新参数：

$$W \leftarrow W - \eta \nabla$$

而 Momentum (动量法) 会保存历史梯度并给予一定的权重，使其也参与到参数更新中：

$$v \leftarrow \alpha v - \eta \nabla$$

$$W \leftarrow W + v$$

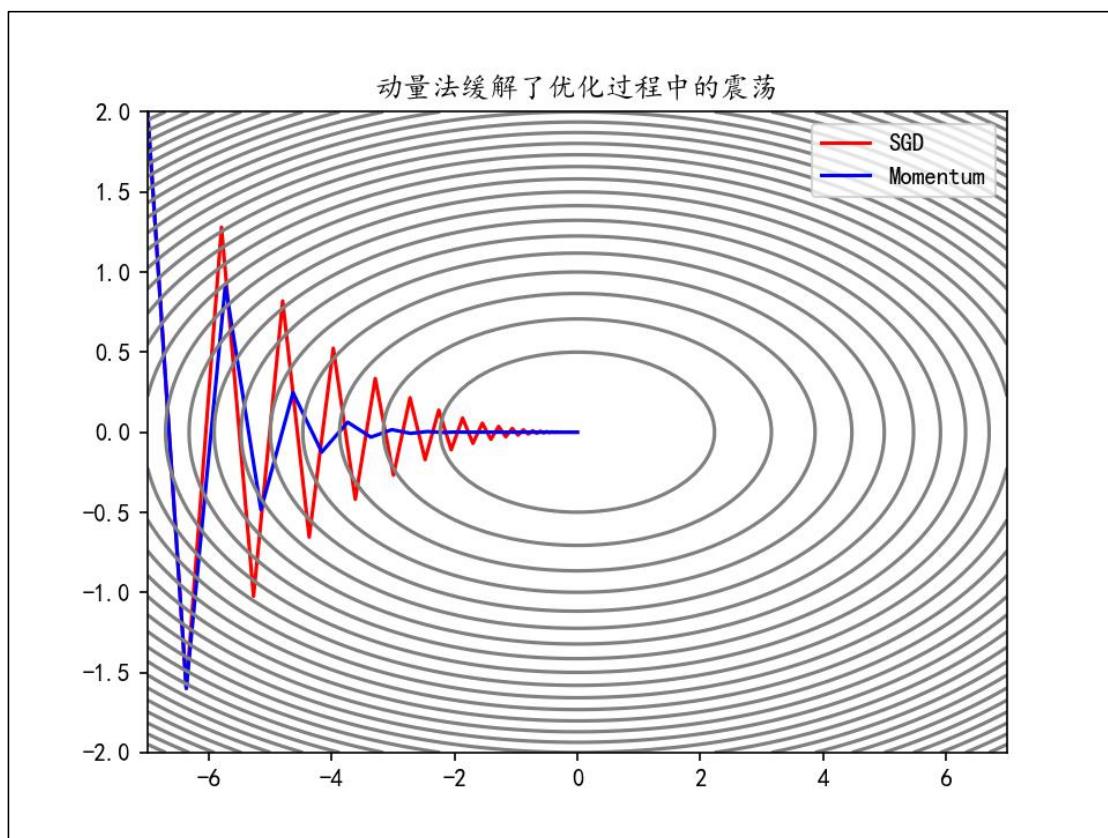
➤  $V$ : 历史(负)梯度的加权和

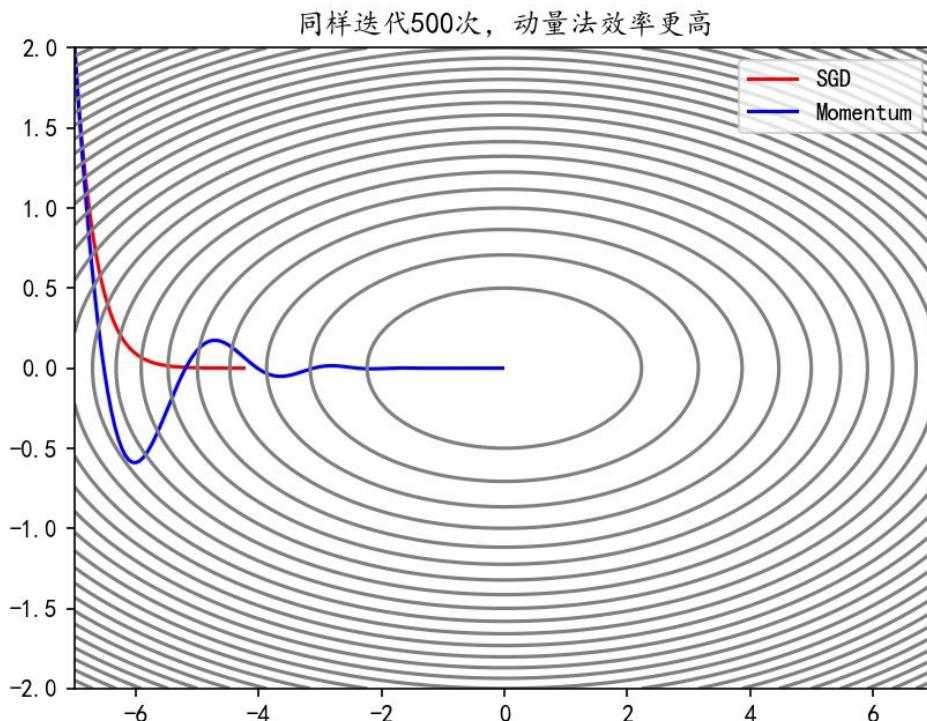
➤  $\alpha$ : 历史梯度的权重

➤  $\nabla$ : 当前梯度，即  $\frac{\partial L}{\partial W}$

➤  $\eta$ : 学习率

动量法有时能够减缓优化过程中的震荡，加快优化的速度。因为其会累计历史梯度，也可以有效避免鞍点问题。





Momentum 的代码实现如下：

```
class Momentum:

    def __init__(self, lr=0.01, momentum=0.9):
        self.lr = lr
        self.momentum = momentum
        self.v = None

    def update(self, params, grads):
        if self.v is None:
            self.v = {}
            for key, val in params.items():
                self.v[key] = np.zeros_like(val)

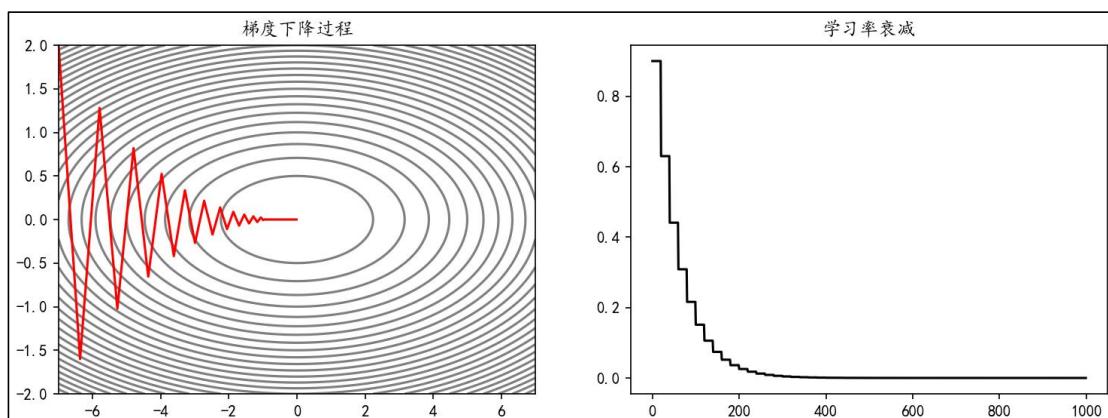
        for key in params.keys():
            self.v[key] = self.momentum*self.v[key] - self.lr*grads[key]
            params[key] += self.v[key]
```

### 5.2.3 学习率衰减

深度学习模型训练中调整最频繁的当属学习率，好的学习率可以使模型逐渐收敛并获得更好的精度。较大的学习率可以加快收敛速度，但可能在最优解附近震荡或不收敛；较小的学习率可以提高收敛的精度，但训练速度慢。学习率衰减是一种平衡策略，初期使用较大学习率快速接近最优解，后期逐渐减小学习率，使参数更稳定地收敛到最优解。

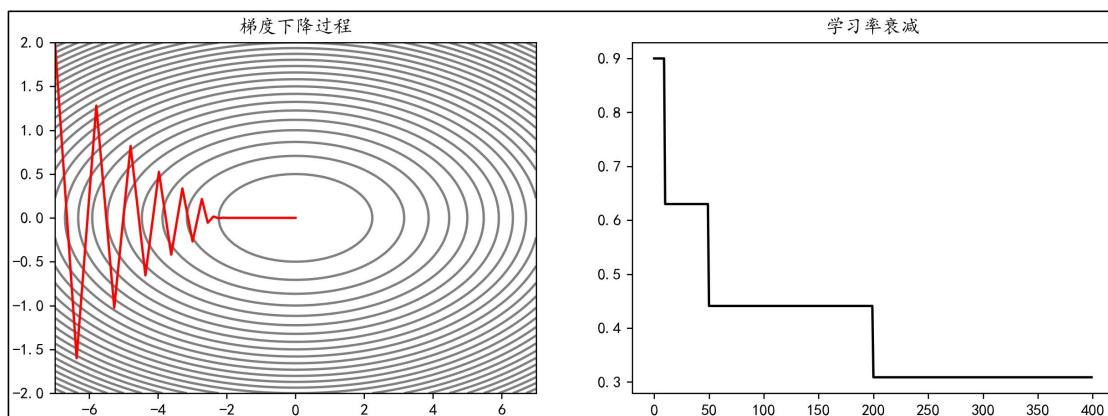
#### 1) 等间隔衰减

每隔固定的训练周期（epoch），学习率按一定的比例下降，也称为“步长衰减”。例如，使学习率每隔 20 epoch 衰减为之前的 0.7：



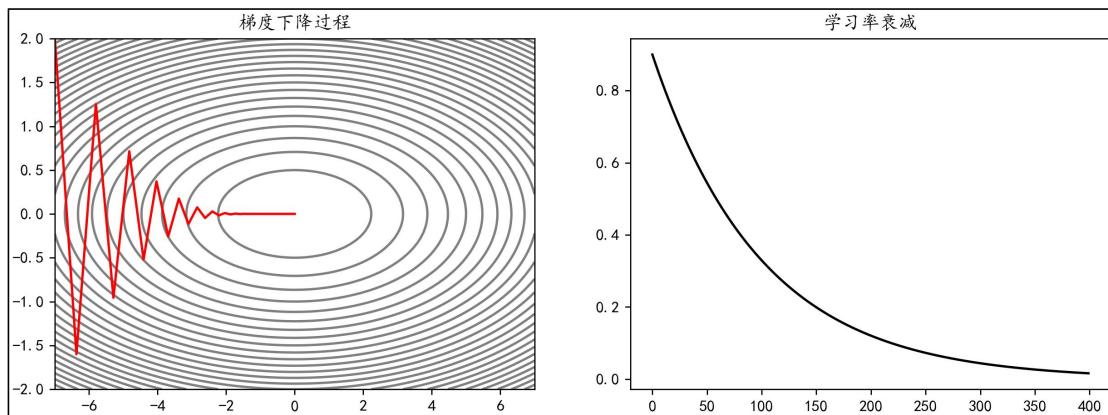
#### 2) 指定间隔衰减

在指定的 epoch，让学习率按照一定的系数衰减。例如，使学习率在 epoch 达到 [10,50,200] 时衰减为之前的 0.7：



#### 3) 指数衰减

学习率按照指数函数  $f(x) = a^x$ ,  $a < 1$  进行衰减。例如，使学习率以 0.99 为底数，epoch 为指数衰减：



### 5.2.4 AdaGrad

AdaGrad (Adaptive Gradient, 自适应梯度) 会为每个参数适当地调整学习率，并且随着学习的进行，梯度会逐渐减小。

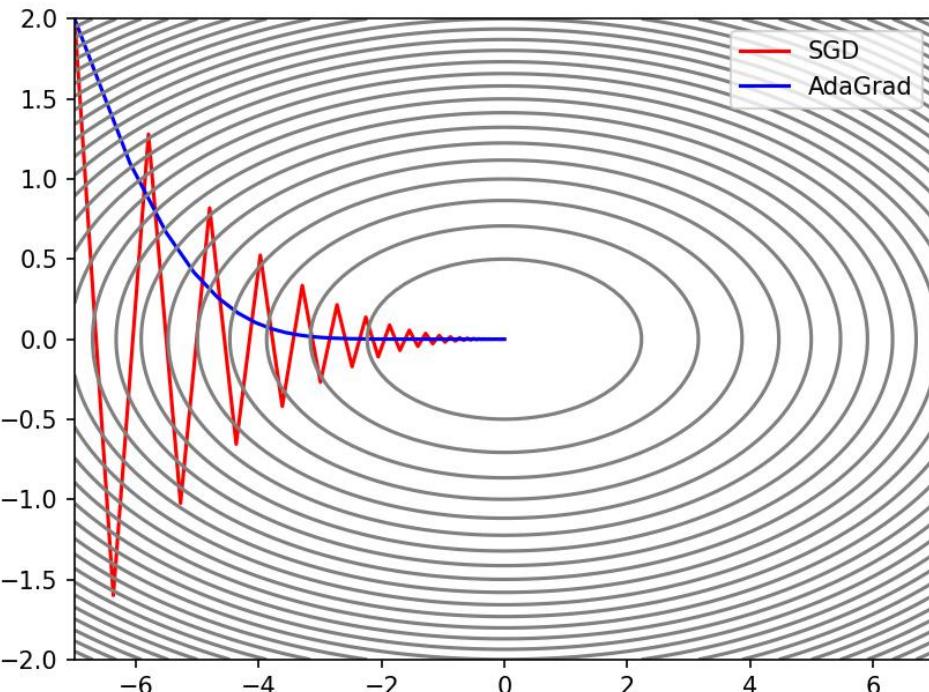
$$h \leftarrow h + \nabla^2$$

$$W \leftarrow W - \eta \frac{1}{\sqrt{h}} \nabla$$

➤  $h$ : 历史梯度的平方和

这里  $\nabla^2$  就表示了梯度的平方和，即  $\frac{\partial L}{\partial W} \odot \frac{\partial L}{\partial W}$ ，这里的  $\odot$  表示对应矩阵元素的乘法。

使用 AdaGrad 时，学习越深入，更新的幅度就越小。如果无止境地学习，更新量就会变为 0，完全不再更新。



AdaGrad 的代码实现如下：

```
class AdaGrad:

    def __init__(self, lr=0.01):
        self.lr = lr
        self.h = None

    def update(self, params, grads):
        if self.h is None:
            self.h = {}
        for key, val in params.items():
            self.h[key] = np.zeros_like(val)

        for key in params.keys():
            self.h[key] += grads[key] * grads[key]
            params[key] -= self.lr * grads[key] / (np.sqrt(self.h[key]) +
1e-7)
```

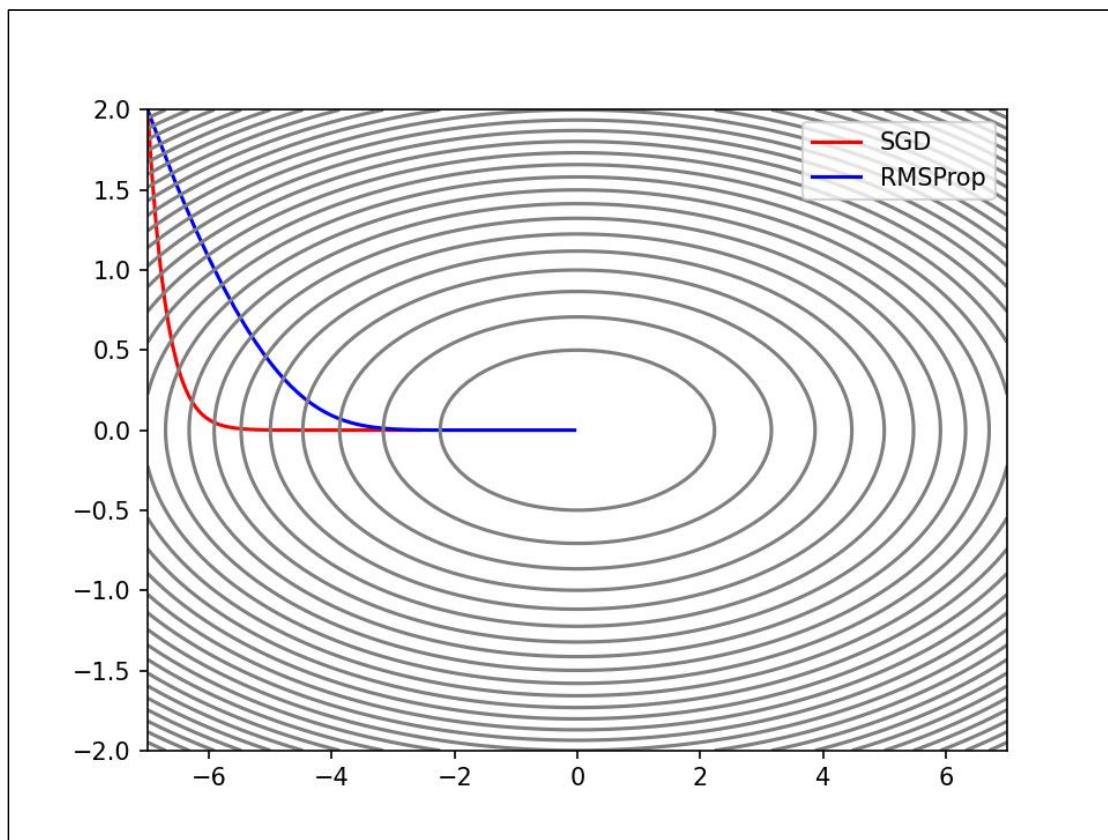
### 5.2.5 RMSProp

RMSProp (Root Mean Square Propagation, 均方根传播) 是在 AdaGrad 基础上的改进，它并非将过去所有梯度一视同仁的相加，而是逐渐遗忘过去的梯度，采用指数移动加权平均，呈指数地减小过去梯度的尺度。

$$h \leftarrow \alpha h + (1 - \alpha)\nabla^2$$

$$W \leftarrow W - \eta \frac{1}{\sqrt{h}} \nabla$$

- $h$ : 历史梯度平方和的指数移动加权平均
- $\alpha$ : 权重



### 5.2.6 Adam

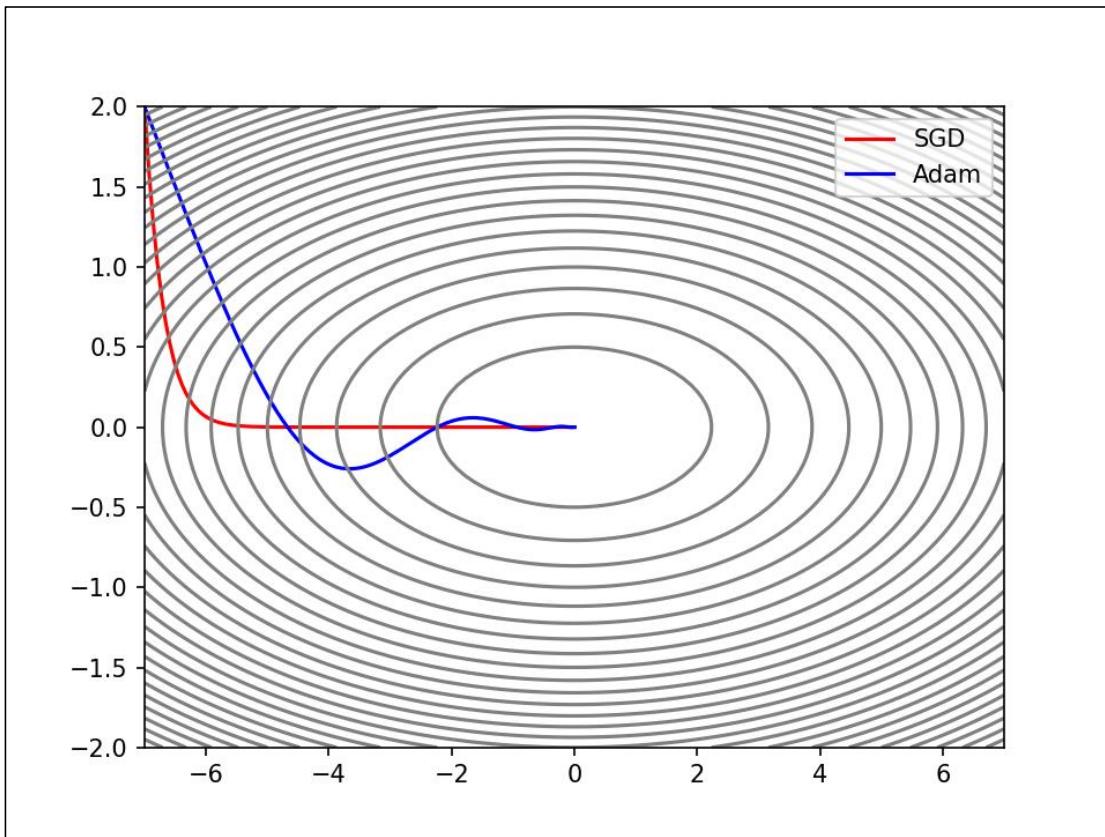
Adam (Adaptive Moment Estimation, 自适应矩估计) 融合了 Momentum 和 AdaGrad 的方法。

$$v \leftarrow \alpha_1 v + (1 - \alpha_1)\nabla$$

$$h \leftarrow \alpha_2 h + (1 - \alpha_2)\nabla^2$$

$$\hat{v} = \frac{v}{1 - \alpha_1^t}$$
$$\hat{h} = \frac{h}{1 - \alpha_2^t}$$
$$W \leftarrow W - \eta \frac{\hat{v}}{\sqrt{\hat{h}}}$$

- $\eta$ : 学习率
- $\alpha_1$ 、 $\alpha_2$ : 一次动量系数和二次动量系数
- $t$ : 迭代次数, 从 1 开始



## 5.3 参数初始化

参数初始化方案的选择在神经网络学习中起着举足轻重的作用, 它对保持数值稳定性至关重要。此外, 这些初始化方案的选择可以与激活函数的选择有趣的结合在一起。我们选择哪个激活函数以及如何初始化参数, 可以决定优化算法收敛的速度有多快; 糟糕选择可能会导致我们在训练时遇到梯度爆炸或梯度消失。

### 5.3.1 常数初始化

所有权重参数初始化为一个常数, 即

$$W = k \cdot J$$

这里  $J$  为全 1 矩阵,  $k$  为初始化的常数。

注意: 将权重初始值设为 0 将无法正确进行学习。严格地说, 不能将权重初始值设成一样的值。因为这意味着反向传播时权重全部都会进行相同的更新, 被更新为相同的值(对称的值)。这使得神经网络拥有许多不同的权重的意义丧失了。为了防止“权重均一化”(瓦解权重的对称结构), 必须随机生成初始值。

### 5.3.2 秩初始化

权重参数初始化为单位矩阵, 即

$$W = I$$

这里  $I$  为单位矩阵, 即主对角线上元素为 1, 其它元素为 0。

### 5.3.3 正态分布初始化

权重参数按指定均值  $\mu$  与标准差  $\sigma$  正态分布初始化。因为不能直接将权重初始化为相同的常数, 所以需要对参数进行随机初始化。最常见的随机分布就是 **正态分布**(也叫 **高斯分布**), 记作  $X \sim N(\mu, \sigma^2)$ 。

其概率密度函数为:

$$f(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$

### 5.3.4 均匀分布初始化

权重参数在指定区间内均匀分布初始化。均匀分布一般记作  $X \sim U(a, b)$ 。

其概率密度函数为:

$$f(x) = \begin{cases} \frac{1}{b-a}, & a < x < b \\ 0, & x \leq a \text{ or } x \geq b \end{cases}$$

### 5.3.5 Xavier 初始化(也叫 Glorot 初始化)

Xavier 初始化根据输入和输出的神经元数量调整权重的初始范围, 确保每一层的输出方差与输入方差相近。

Xavier 正态分布初始化: 均值为 0, 标准差为  $\sqrt{\frac{2}{n_{in}+n_{out}}}$  的正态分布。

Xavier 均匀分布初始化: 区间  $\left[-\sqrt{\frac{6}{n_{in}+n_{out}}}, \sqrt{\frac{6}{n_{in}+n_{out}}}\right]$  内均匀分布。

其中  $n_{in}$  表示输入数， $n_{out}$  表示输出数。

Xavier 初始化参数适用于 Sigmoid 和 Tanh 等激活函数，能有效缓解梯度消失或爆炸问题。

### 5.3.6 He 初始化（也叫 Kaiming 初始化）

He 初始化根据输入的神经元数量调整权重的初始范围。

He 正态分布初始化：均值为 0，标准差为  $\sqrt{\frac{2}{n_{in}}}$  的正态分布。

He 均匀分布初始化：区间  $[-\sqrt{\frac{6}{n_{in}}}, \sqrt{\frac{6}{n_{in}}}]$  内均匀分布。

其中  $n_{in}$  表示输入数。

He 初始化参数主要适用于 ReLU 及其变体（如 Leaky ReLU）激活函数。

## 5.4 正则化

机器学习的问题中，**过拟合** 是一个很常见的问题。

过拟合指的是能较好拟合训练数据，但不能很好地拟合不包含在训练数据中的其他数据。

机器学习的目标是提高泛化能力，希望即便是不包含在训练数据里的未观测数据，模型也可以进行正确的预测。因此可以通过 **正则化** 方法来抑制过拟合。

常用的正则化方法有 Batch Normalization、权值衰减、Dropout、早停法等。

### 5.4.1 Batch Normalization 批量标准化

Batch Normalization 最重要的目的，其实是调整各层的激活值分布使其拥有适当的广度，BN 层通常放在线性层（全连接层/卷积层）之后，激活函数之前。它有着以下优点：

- 可以使学习快速进行（允许更高的学习率）
- 不那么依赖初始值（对于初始值不用那么神经质）
- 抑制过拟合（降低 Dropout 等的必要性）

Batch Normalization 会先对数据进行标准化，再对数据进行缩放和平移：

$$\text{均值: } \mu = \frac{1}{n} \sum x$$

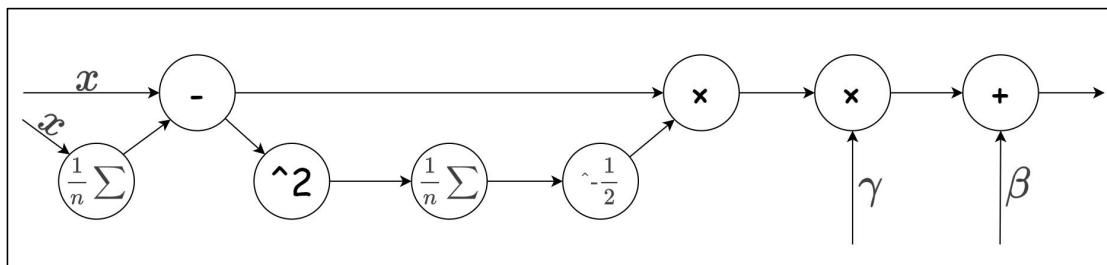
$$\text{方差: } \sigma^2 = \frac{1}{n} \sum (x - \mu)^2$$

$$\text{标准化: } \hat{x} = \frac{x - \mu}{\sqrt{\sigma^2 + \epsilon}} \quad (\epsilon \text{ 为一个微小值, 防止分母为 0})$$

$$\text{缩放平移: } y = \gamma \hat{x} + \beta$$

- $\epsilon$ : 一个微小值，防止分母为 0
- $\gamma$ : 系数，可通过学习调整
- $\beta$ : 偏置，可通过学习调整

下图为 Batch Normalization 的计算图：



### 5.4.2 权值衰减

通过在学习的过程中对大的权重进行“惩罚”，可以有效地抑制过拟合，这种方法被称为 **权值衰减**。因为很多过拟合产生的原因，就是权重参数取值过大。

一般会对损失函数加上一个权重的范数；最常见的就是 L2 范数的平方：

$$L' = L + \frac{1}{2} \cdot \lambda \cdot \|W\|^2$$

- $\|W\|$ : 表示权重  $W = (w_1, w_2, \dots, w_n)$  的 L2 范数，即  $\sqrt{w_1^2 + w_2^2 + \dots + w_n^2}$
- $\lambda$ : 控制正则化强度的超参数。

惩罚项  $\frac{1}{2} \cdot \lambda \cdot \|W\|^2$  求导之后得到  $\lambda W$ ；所以在求权重梯度时，需要为之前误差反向传播法的结果，再加上  $\lambda W$ 。

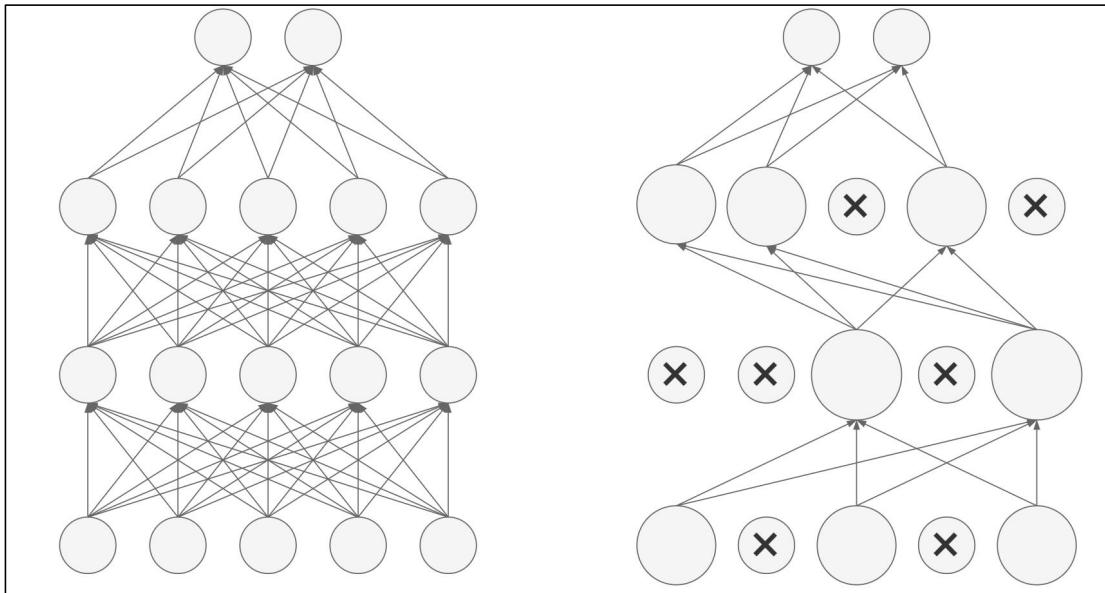
### 5.4.3 Dropout 随机失活

Dropout（随机失活，暂退法）是一种在学习的过程中随机关闭神经元的方法。

训练时以概率  $p$  随机关闭神经元，迫使网络不依赖特定神经元，增强鲁棒性，同时未被关闭的神经元的输出值以  $\frac{1}{(1-p)}$  的比例进行缩放，以保持期望值不变；而测试时通常不使用 Dropout，即所有神经元保持激活状态并且不进行缩放。

Dropout 会有隐式集成的效果（每次迭代训练不同的子网络，测试时近似集成效果）。

Dropout 在全连接层和卷积层均适用，尤其对大规模网络效果显著。Dropout 通常放在激活函数之后，线性层（全连接层/卷积层）之前。



## 第 6 章 PyTorch 简介

### 6.1 什么是 PyTorch

PyTorch 是一个开源的 Python 机器学习库，基于 Torch 库（一个有大量机器学习算法支持的科学计算框架，有着与 Numpy 类似的张量（Tensor）操作，采用的编程语言是 Lua），底层由 C++ 实现，应用于人工智能领域，如计算机视觉和自然语言处理。

PyTorch 主要有两大特征：

- 类似于 NumPy 的张量计算，能在 GPU 或 MPS 等硬件加速器上加速。
- 基于带自动微分系统的深度神经网络。

PyTorch 官网：<https://pytorch.org/>。

### 6.2 PyTorch 安装

PyTorch 分为 CPU 和 GPU 版本。

PyTorch 选择安装版本页面：<https://pytorch.org/get-started/locally/>。

## START LOCALLY

Select your preferences and run the install command. Stable represents the most currently tested and supported version of PyTorch. This should be suitable for many users. Preview is available if you want the latest, not fully tested and supported, builds that are generated nightly. Please ensure that you have **met the prerequisites below (e.g., numpy)**, depending on your package manager. You can also [install previous versions of PyTorch](#). Note that LibTorch is only available for C++.

**NOTE:** Latest PyTorch requires Python 3.9 or later.

| PyTorch Build     | Stable (2.6.0)  |           | Preview (Nightly) |            |
|-------------------|---|-----------|-------------------|------------|
| Your OS           | Linux   | Mac       | Windows           |            |
| Package           | Conda   | Pip       | LibTorch          | Source     |
| Language          | Python  |           |                   | C++ / Java |
| Compute Platform  | CUDA 11.8   | CUDA 12.4 | CUDA 12.6         | ROCM 6.2.4 |
| Run this Command: | <pre>pip3 install torch torchvision torchaudio --index-url https://download.pytorch.org/whl/cu118</pre> |           |                   |            |

### 6.2.1 CPU 版本 PyTorch 安装

直接通过 pip 命令安装即可: `pip3 install torch torchvision torchaudio`。

若需要离线安装，可以考虑下载 whl 包然后自行安装。下载 whl 的链接：

<https://download.pytorch.org/whl/torch/>。

手动下载 whl 时，需要注意 PyTorch 与 torchvision 之间版本对应关系。可以到 <https://github.com/pytorch/vision> 或 <https://pytorch.org/get-started/previous-versions/> 查看。

| <b>torch</b>   | <b>torchvision</b> | <b>Python</b>  |
|----------------|--------------------|----------------|
| main / nightly | main / nightly     | >=3.9 , <=3.12 |
| 2.5            | 0.20               | >=3.9 , <=3.12 |
| 2.4            | 0.19               | >=3.8 , <=3.12 |
| 2.3            | 0.18               | >=3.8 , <=3.12 |
| 2.2            | 0.17               | >=3.8 , <=3.11 |
| 2.1            | 0.16               | >=3.8 , <=3.11 |
| 2.0            | 0.15               | >=3.8 , <=3.11 |

## 6.2.2 GPU 版本 PyTorch 安装

绝大多数情况下我们会安装 GPU 版本的 PyTorch。目前 PyTorch 不仅支持 NVIDIA 的 GPU，还支持 AMD 的 ROCm 的 GPU。

安装 GPU 版本的 PyTorch 步骤：

- 根据 NVIDIA 驱动程序版本和要安装的 PyTorch 版本，确定安装哪个版本的 CUDA。
- 根据 CUDA 版本，安装对应版本的 cuDNN。

### 1) GPU 计算能力要求

对于 N 卡，需要计算能力（compute capability） $\geq 3.0$ 。

#### torch

The torch package contains data structures for multi-dimensional tensors and defines mathematical operations over these tensors. Additionally, it provides many utilities for efficient serialization of Tensors and arbitrary types, and other useful utilities.

It has a CUDA counterpart, that enables you to run your tensor computations on an NVIDIA GPU with compute capability  $\geq 3.0$ .

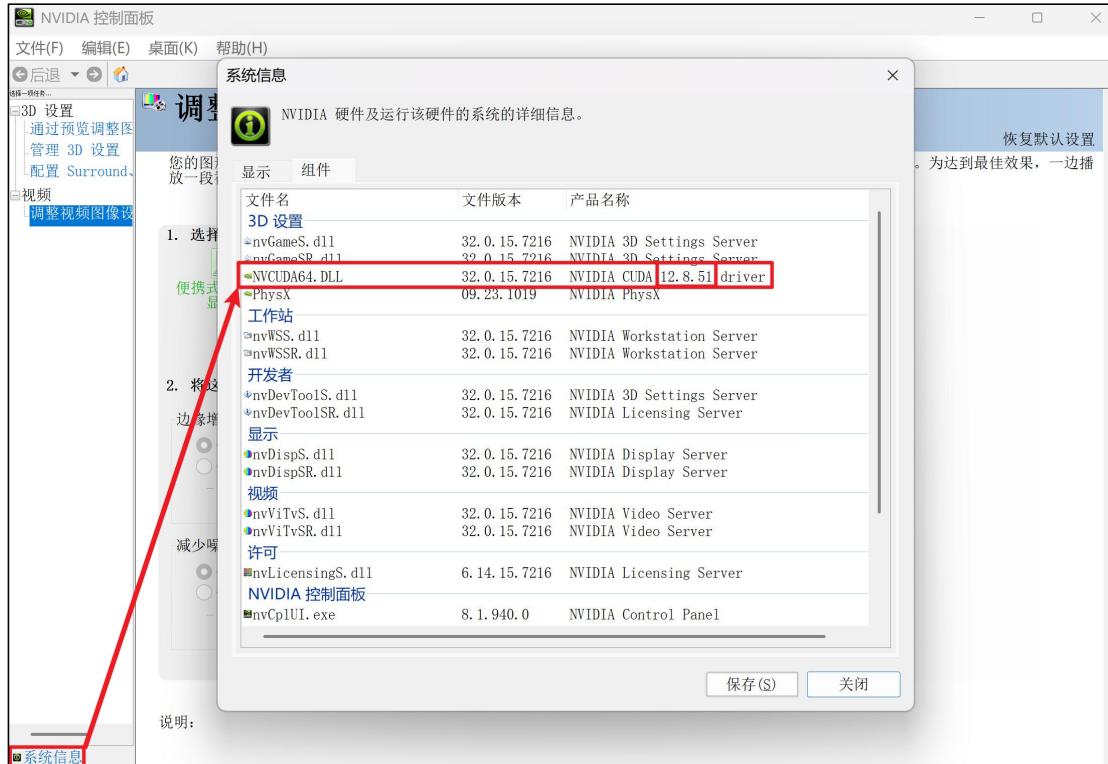
可在 <https://developer.nvidia.cn/cuda-gpus#compute> 查看 GPU 计算能力。

### 2) CUDA 版本选择

CUDA（Compute Unified Device Architecture）是 NVIDIA 开发的并行计算平台和编程平台，允许开发者利用 NVIDIA GPU 的强大计算能力进行通用计算。CUDA 不仅用于图形渲染，还广泛应用于科学计算、深度学习、金融建模等领域。

- (1) 根据 NVIDIA 驱动程序版本确定支持的最高 CUDA 版本

打开 NVIDIA 控制面板 → 系统信息 → 组件，查看 NVCUDA64.DLL 的产品名称栏，可查看驱动程序支持的最高 CUDA 版本。

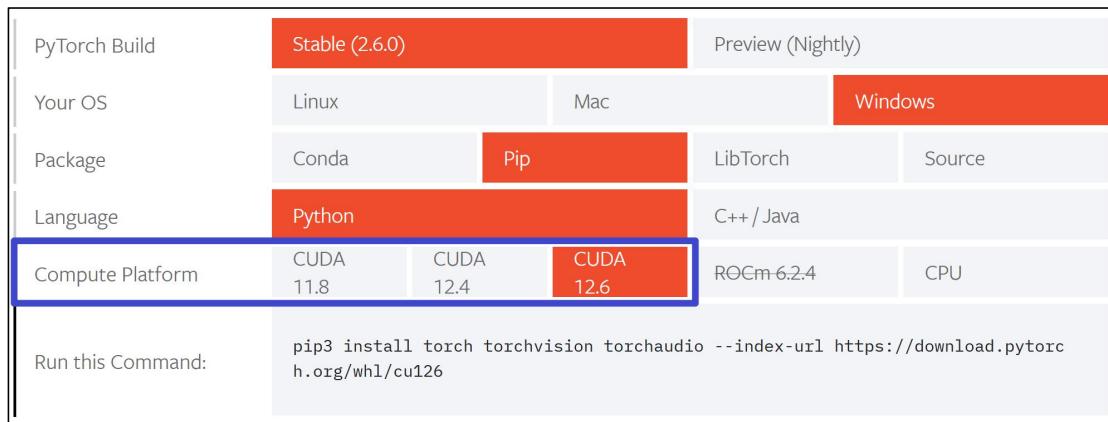


或在命令行中输入 `nvidia-smi`，在 CUDA Version 栏查看支持的最高 CUDA 版本。

```
NVIDIA-SMI 572.16      Driver Version: 572.16      CUDA Version: 12.8
+-----+-----+-----+
| GPU  Name        | Driver-Model | Bus-Id   | Disp.A    | Volatile | Uncorr. | ECC |
| Fan  Temp        | Pwr:Usage/Cap |          | Memory-Usage | GPU-Util | Compute M. |
|      Perf        | Cap          |          |             |          |          | MIG M. |
+-----+-----+-----+
| 0  NVIDIA GeForce RTX 4080 ... | WDDM          | 00000000:01:00.0 | Off       | 0MiB / 12282MiB | 0%        | N/A |
| N/A  51C          | 16W / 70W     |          |             |          |          | Default |
|          P0          |              |          |             |          |          | N/A |
+-----+-----+-----+
Processes:
+-----+-----+-----+-----+-----+
| GPU ID | GI ID | CI ID | PID | Type | Process name | GPU Memory Usage |
+-----+-----+-----+-----+-----+
No running processes found
```

## (2) 根据 PyTorch 版本选择 CUDA 版本

需要安装特定版本的 CUDA 版本，才能使用特定版本的 PyTorch。在 PyTorch 下载页面可查看该版本 PyTorch 支持的 CUDA 版本。



或在 <https://download.pytorch.org/whl/torch/> 查看过往版本 PyTorch 支持的 CUDA 版本。

例如：

`torch-2.6.0+cu126-cp313-cp313-win_amd64.whl`

此处的 cu126 表示支持 CUDA12.6 版本。

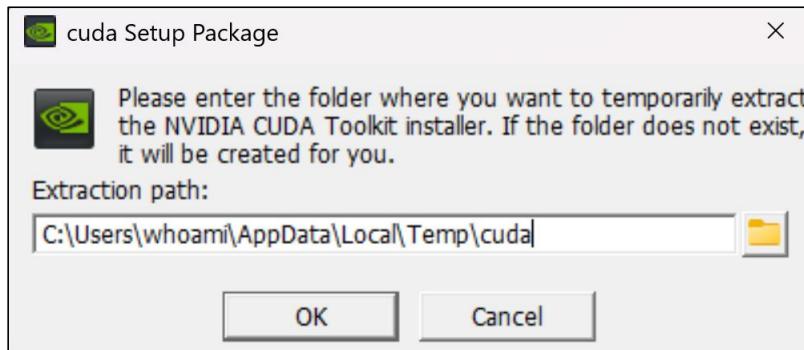
### 3) CUDA 安装

NVIDIA 官网通常只展示最新的 CUDA 版本，过往 CUDA 版本可在 <https://developer.nvidia.com/cuda-toolkit-archive> 下载。

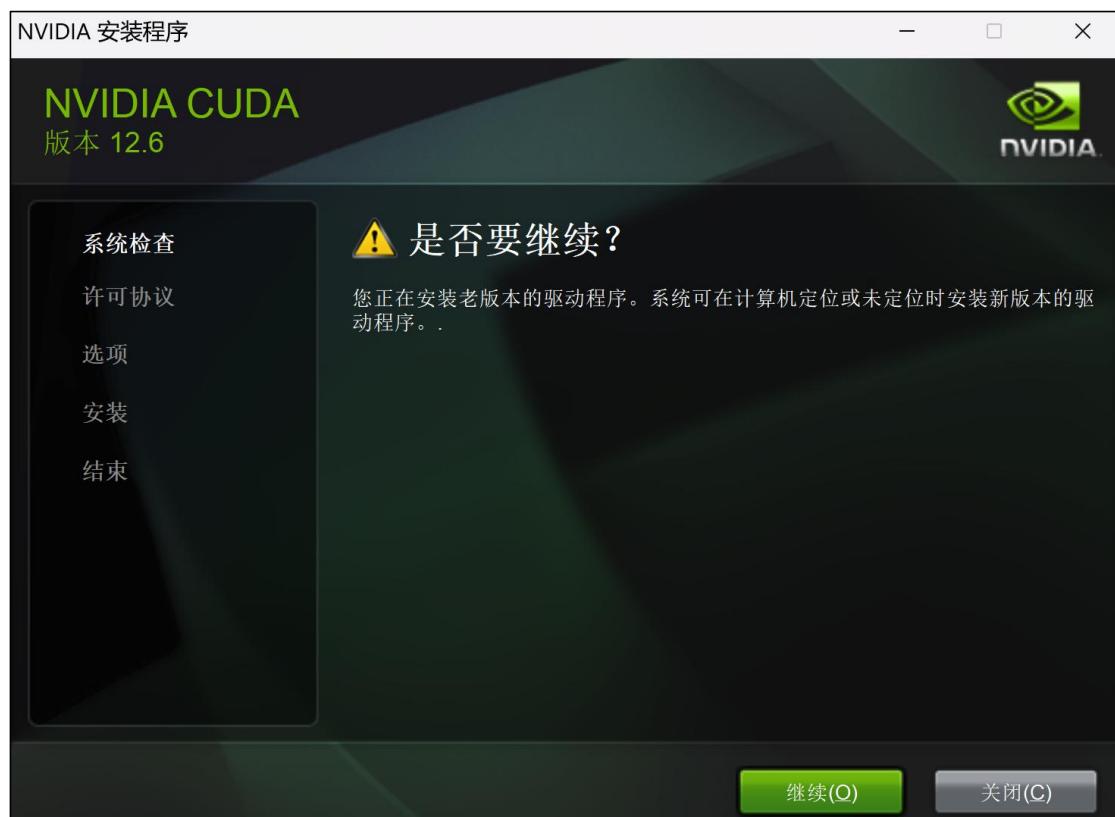
选择相应 CUDA 版本后，选择要安装的平台，Installer Type 安装方式选择 exe(local)本地安装。

The screenshot shows the 'Select Target Platform' page of the NVIDIA CUDA Toolkit installer download page. It includes fields for Operating System (Linux, Windows), Architecture (x86\_64), Version (10, 11, Server 2022), and Installer Type (exe (local) selected, exe (network)). A green bar at the bottom indicates the download link for 'Download Installer for Windows 11 x86\_64'.

双击.exe 文件进行安装，首先需要输入临时解压路径，临时解压路径在安装结束后会自动被删除，保持默认即可。点击 OK。



若在系统检查环节提示“您正在安装老版本的驱动程序...”，说明安装包中包含的驱动程序版本比当前已安装的驱动程序的版本旧，可忽略。点击继续。



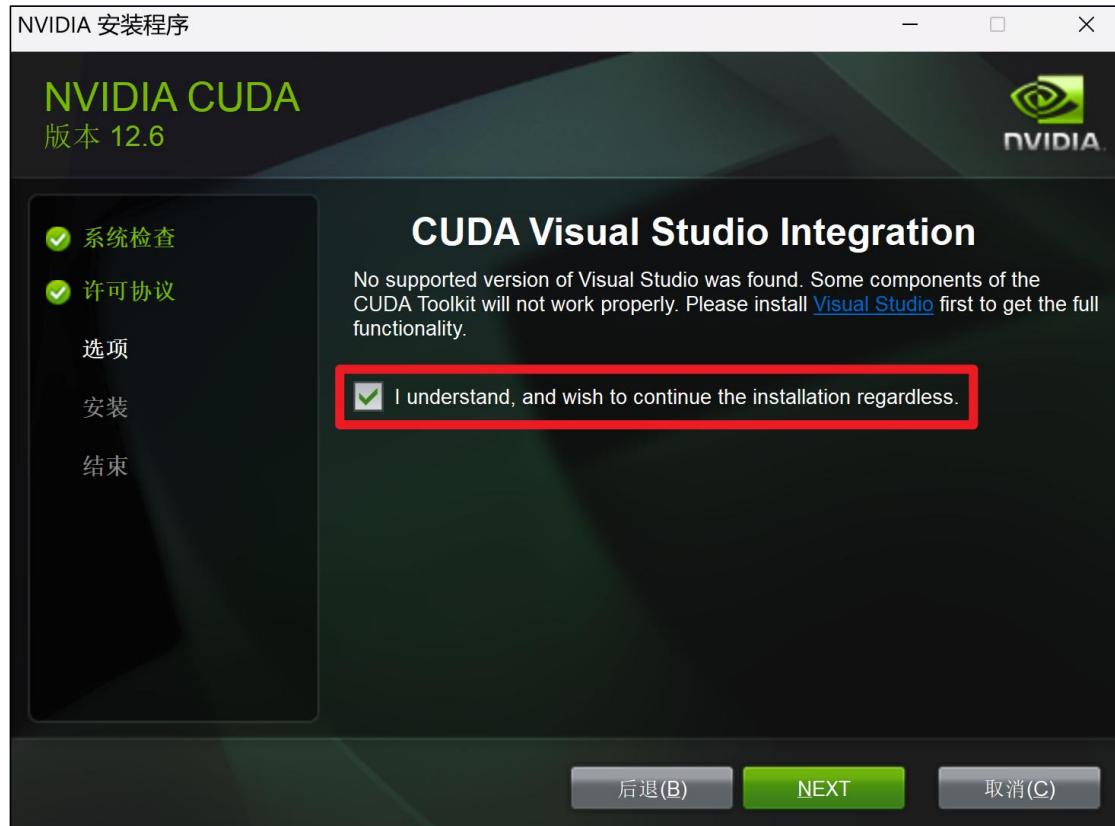
同意安装协议并继续。



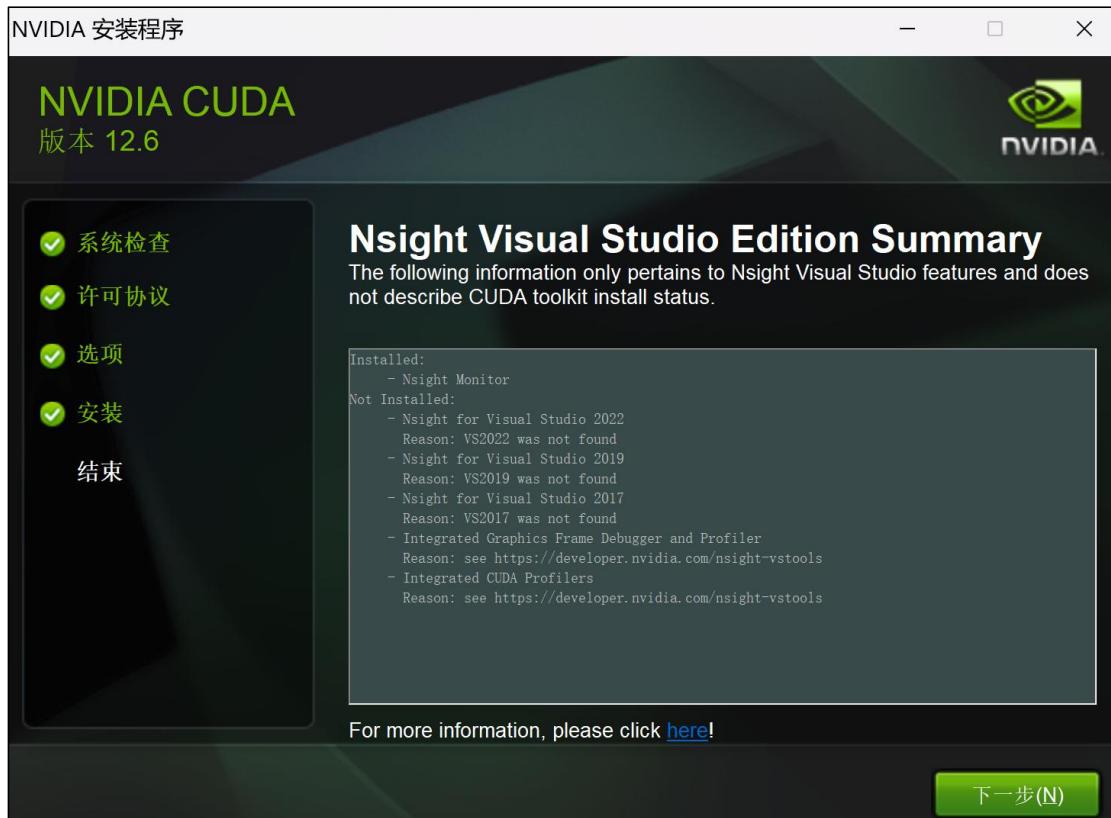
选择精简，会安装所有组件并覆盖现有驱动程序。点击下一步。



如果出现以下提示，表明缺少 Visual Studio，部分组件不能正常工作。不用在意，选择 I understand...。点击 Next。



点击下一步。



安装完成，点击关闭。



可在命令行使用 `nvcc --version` 查看 CUDA 版本信息。

#### 4) cuDNN 安装

cuDNN (NVIDIA CUDA Deep Neural Network library, 深度神经网络库) 是用于深度神经网络的 GPU 加速原语库。cuDNN 为标准例程 (如前向和反向卷积、注意力、matmul、池化和归一化) 提供了高度调优的实现。

cuDNN 下载地址: <https://developer.nvidia.com/cudnn-archive>。

cuDNN 下载后是一个压缩包, 解压后包含 bin、include、lib 三个文件夹。

找到 CUDA 安装目录, 默认在 C:\Program Files\NVIDIA GPU Computing Toolkit\CUDA\v12.6。

分别将 cuDNN 的 bin、include、lib\x64 文件夹中的文件拷贝到 CUDA 的 bin、include、lib\x64 文件夹中。

下面验证 cuDNN 是否安装成功。进入 C:\Program Files\NVIDIA GPU Computing Toolkit\CUDA\v12.6\extras\demo\_suite 目录下, 打开命令行, 分别执行 deviceQuery.exe 与 bandwidthTest.exe。如果出现类似下图的输出则说明安装成功。

```
> ./deviceQuery.exe
C:\Program Files\NVIDIA GPU Computing Toolkit\CUDA\v12.6\extras\demo_suite\deviceQuery.exe Starting ...
CUDA Device Query (Runtime API) version (CUDART static linking)

Detected 1 CUDA Capable device(s)

Device 0: "NVIDIA GeForce RTX 4080 Laptop GPU"
  CUDA Driver Version / Runtime Version      12.8 / 12.6
  CUDA Capability Major/Minor version number:  8.9
  Total amount of global memory:            12282 MBytes (12878086144 bytes)
  MapSMtoCores for SM 8.9 is undefined. Default to use 128 Cores/SM
  MapSMtoCores for SM 8.9 is undefined. Default to use 128 Cores/SM
  (58) Multiprocessors, (128) CUDA Cores/MP:    7424 CUDA Cores
  GPU Max Clock rate:                      1350 MHz (1.35 GHz)
  Memory Clock rate:                       9001 Mhz
  Memory Bus Width:                        192-bit
  L2 Cache Size:                          50331648 bytes
  Maximum Texture Dimension Size (x,y,z):  1D=(131072), 2D=(131072, 65536), 3D=(16384, 16384, 16384)
  Maximum Layered 1D Texture Size, (num) layers  1D=(32768), 2048 layers
  Maximum Layered 2D Texture Size, (num) layers  2D=(32768, 32768), 2048 layers
  Total amount of constant memory:          zu bytes
  Total amount of shared memory per block:   zu bytes
  Total number of registers available per block: 65536
  Warp size:                            32
  Maximum number of threads per multiprocessor: 1536
  Maximum number of threads per block:       1024
  Max dimension size of a thread block (x,y,z): (1024, 1024, 64)
  Max dimension size of a grid size      (x,y,z): (2147483647, 65535, 65535)
  Maximum memory pitch:                  zu bytes
```

```
> .\bandwidthTest.exe
[CUDA Bandwidth Test] - Starting ...
Running on ...

Device 0: NVIDIA GeForce RTX 4080 Laptop GPU
Quick Mode

Host to Device Bandwidth, 1 Device(s)
PINNED Memory Transfers
Transfer Size (Bytes)      Bandwidth(MB/s)
33554432                  12123.5

Device to Host Bandwidth, 1 Device(s)
PINNED Memory Transfers
Transfer Size (Bytes)      Bandwidth(MB/s)
33554432                  12849.3

Device to Device Bandwidth, 1 Device(s)
PINNED Memory Transfers
Transfer Size (Bytes)      Bandwidth(MB/s)
33554432                  318467.4

Result = PASS

NOTE: The CUDA Samples are not meant for performance measurements. Results may vary when GPU Boost is enabled.
```

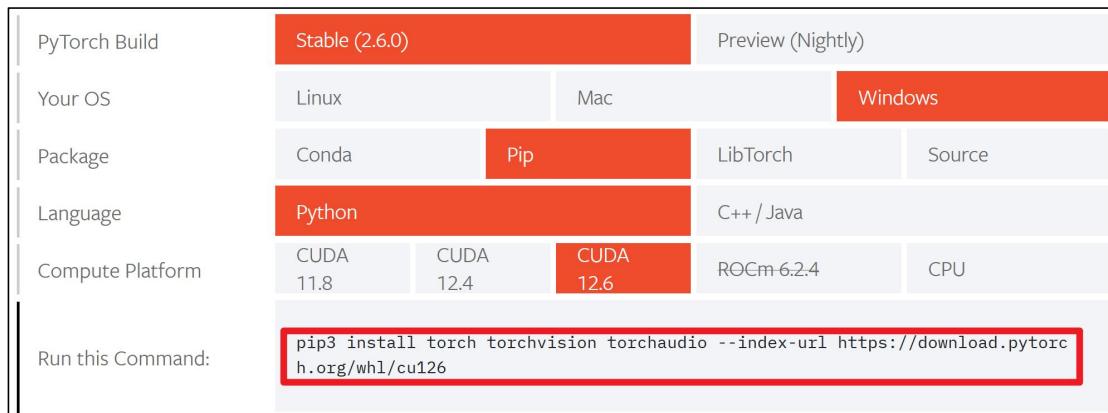
## 5) PyTorch 安装

新建一个虚拟环境来安装 PyTorch。

在命令行输入 `conda create -n pytorch-2.6.0-gpu python=3.12` 创建一个环境名为 pytorch-2.6.0-gpu，Python 版本为 3.12 的虚拟环境。

使用 `conda activate pytorch-2.6.0-gpu` 激活 pytorch-2.6.0-gpu 虚拟环境。

在官网 <https://pytorch.org/get-started> 选择要安装的版本，复制命令，在命令行中执行以安装 PyTorch。



若安装速度较慢或安装失败，可配置 pip 的国内镜像源 `pip config set global.index-url https://pypi.tuna.tsinghua.edu.cn/simple`。

要在新的虚拟环境中使用 Jupyter Notebook，需使用 `conda install jupyter notebook` 安装。

编写代码时需在 IDE 中选择新创建的虚拟环境作为 Python 解释器。

## 6.3 张量创建

Tensor（张量）是 PyTorch 的核心数据结构。张量在不同学科中有不同的意义，在深度学习中张量表示一个多元数组，是标量、向量、矩阵的拓展。如一个 RGB 图像的数组就是一个三维张量，第 1 维是图像的高，第 2 维是图像的宽，第 3 维是图像的颜色通道。

### 6.3.1 基本张量创建

#### 1) `torch.tensor(data)` 创建指定内容的张量

```
import torch
import numpy as np

# 创建标量张量
tensor1 = torch.tensor(10)
print(tensor1)

# 使用列表创建张量
tensor2 = torch.tensor([1, 2, 3])
print(tensor2)

# 使用 numpy 创建张量
tensor3 = torch.tensor(np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]]))
print(tensor3)
```

#### 2) `torch.Tensor(size)` 创建指定形状的张量

```
import torch

# 创建指定形状的张量，默认类型为 float32
tensor1 = torch.Tensor(3, 2, 4)
print(tensor1)
print(tensor1.dtype)

# 也可以用来创建指定内容的张量
tensor2 = torch.Tensor([[1, 2, 3, 4], [5, 6, 7, 8]])
print(tensor2)
```

#### 3) 创建指定类型的张量

可通过 `torch.IntTensor()`、`torch.FloatTensor()` 等创建。

或在 `torch.tensor()` 中通过 `dtype` 参数指定类型。

```
import torch

# 创建 int32 类型的张量
tensor1 = torch.IntTensor(2, 3)
tensor2 = torch.tensor([1, 2, 3], dtype=torch.int32)
print(tensor1)
```

```
print(tensor2)

# 元素类型不匹配则会进行类型转换
tensor1 = torch.IntTensor([1.1, 2.2, 3.6])
tensor2 = torch.tensor([3.1, 2.2, 1.6], dtype=torch.int32)
print(tensor1)
print(tensor2)

# 创建 int64 类型的张量
tensor1 = torch.LongTensor([1, 2, 3])
tensor2 = torch.tensor([1, 2, 3], dtype=torch.int64)
print(tensor1, tensor1.dtype)
print(tensor2, tensor1.dtype)

# 创建 int16 类型的张量
tensor1 = torch.ShortTensor(2, 2)
tensor2 = torch.tensor([1, 2, 3], dtype=torch.int16)
print(tensor1, tensor1.dtype)
print(tensor2, tensor1.dtype)

# 创建 float32 类型的张量
tensor1 = torch.FloatTensor([9, 8, 7])
tensor2 = torch.tensor([1, 2, 3], dtype=torch.float32)
print(tensor1, tensor1.dtype)
print(tensor2, tensor1.dtype)

# 创建 float64 类型的张量
tensor1 = torch.DoubleTensor(2, 3, 1)
tensor2 = torch.tensor([1, 2, 3], dtype=torch.float64)
print(tensor1)
print(tensor2)
```

### 6.3.2 指定区间的张量创建

1) `torch.arange(start, end, step)` 在区间内按步长创建张量

```
import torch

# torch.arange(start, end, step) 在区间[start,end)中创建步长为 step 的张量
tensor1 = torch.arange(10, 30, 2)
print(tensor1)

# torch.arange(end) 创建区间为[0,end), 步长为 1 的张量
tensor2 = torch.arange(6)
print(tensor2)
```

2) `torch.linspace(start, end, steps)`在区间内按元素数量创建张量

```
import torch

# torch.linspace(start, end, steps) 在区间按元素数量创建张量
tensor1 = torch.linspace(10, 30, 5)
print(tensor1)
```

3) `torch.logspace(start, end, steps, base)`在指数区间内按指定底数创建张量

```
import torch

# torch.logspace(start, end, steps, base) 在区间[start,end]之间生成 steps
# 个数，并以 base 为底，区间内的数为指数创建张量
tensor1 = torch.logspace(1, 3, 3, 2)
print(tensor1)
```

### 6.3.3 按数值填充张量

- `torch.zeros(size)`创建指定形状的全 0 张量
- `torch.ones(size)`创建指定形状的全 1 张量
- `torch.full(size)`创建指定形状的按指定值填充的张量
- `torch.empty(size)`创建指定形状的未初始化的张量
- `torch.zeros_like(input)`创建与给定张量形状相同的全 0 张量
- `torch.ones_like(input)`创建与给定张量形状相同的全 1 张量
- `torch.full_like(input)`创建与给定张量形状相同的按指定值填充的张量
- `torch.empty_like(input)`创建与给定张量形状相同的未初始化的张量

```
import torch

# torch.zeros(size) 创建指定形状的全 0 张量
tensor1 = torch.zeros(2, 3)
print(tensor1)

# torch.ones_like(input) 创建与给定张量形状相同的全 1 张量
tensor2 = torch.ones_like(tensor1)
print(tensor2)

# torch.full(size, fill_value) 创建指定形状的按指定值填充的张量
tensor1 = torch.full((2, 3), 6)
print(tensor1)

# torch.empty_like(input) 创建与给定张量形状相同的未初始化的张量
tensor2 = torch.empty_like(tensor3)
```

```
print(tensor2)
    ➤ torch.eye(n, [m]) 创建单位矩阵

import torch

# torch.eye(n) 创建 n*n 的单位矩阵
tensor1 = torch.eye(3)
print(tensor1)

# torch.eye(n, m) 按指定的行和列创建
tensor2 = torch.eye(3, 4)
print(tensor2)
```

### 6.3.4 随机张量创建

- torch.rand(size) 创建在[0,1]上均匀分布的，指定形状的张量
- torch.randint(low, high, size) 创建在[low,high)上均匀分布的，指定形状的张量
- torch.randn(size) 创建标准正态分布的，指定形状的张量
- torch.normal(mean, std, size) 创建自定义正态分布的，指定形状的张量
- torch.rand\_like(input) 创建在[0,1]上均匀分布的，与给定张量形状相同的张量
- torch.randint\_like(input, low, high) 创建在[low,high)上均匀分布的，与给定张量形状相同的张量
- torch.randn\_like(input) 创建标准正态分布的，与给定张量形状相同的张量

```
import torch

# torch.rand(size) 创建在[0,1)上均匀分布的，指定形状的张量
tensor1 = torch.rand(2, 3)
print(tensor1)

# torch.rand_like(input) 创建在[0,1)上均匀分布的，与给定张量形状相同的张量
tensor2 = torch.randint_like(tensor1, 1, 10)
print(tensor2)

# torch.randn(size) 创建标准正态分布的，指定形状的张量
tensor1 = torch.randn(4, 2)
print(tensor1)

# torch.normal(mean, std, size) 创建自定义正态分布的，指定形状的张量。mean 为均值，std 为标准差
tensor2 = torch.normal(5, 1, tensor1.shape)
```

```
print(tensor2)
    ➤ torch.randperm(n)生成从 0 到 n-1 的随机排列，类似洗牌
import torch

# torch.randperm(n) 生成从 0 到 n-1 的随机排列
tensor1 = torch.randperm(10)
print(tensor1)

    ➤ torch.random.initial_seed()查看随机数种子
    ➤ torch.manual_seed(seed)设置随机数种子
import torch

# 查看随机数种子
print(torch.random.initial_seed())
# 设置随机数种子
torch.manual_seed(42)
print(torch.random.initial_seed())
```

## 6.4 张量转换

### 6.4.1 张量元素类型转换

#### 1) `Tensor.type(dtype)`修改张量的类型

```
import torch

tensor1 = torch.tensor([1, 2, 3])
print(tensor1, tensor1.dtype)

# 使用 type 方法修改张量的类型
tensor1 = tensor1.type(torch.float32)
print(tensor1, tensor1.dtype)
```

#### 2) `Tensor.double()`等修改张量的类型

```
import torch

tensor1 = torch.tensor([1, 2, 3])
print(tensor1, tensor1.dtype)

# 使用 double 方法修改张量的类型
tensor1 = tensor1.double()
print(tensor1)
# 使用 long 方法修改张量的类型
tensor1 = tensor1.long()
print(tensor1, tensor1.dtype)
```

## 6.4.2 Tensor 与 ndarray 转换

1) `Tensor.numpy()` 将 Tensor 转换为 ndarray，共享内存。使用 `copy()` 避免共享内存

```
import torch

# 使用 numpy() 方法将 Tensor 转换为 ndarray，共享内存
tensor1 = torch.rand(3, 2)
numpy_array = tensor1.numpy()
print(tensor1)
print(numpy_array)
print(type(tensor1), type(numpy_array))
print()
tensor1[:, 0] = 4
print(tensor1)
print(numpy_array)
print()

# 使用 copy() 方法避免共享内存
numpy_array = tensor1.numpy().copy()
tensor1[:, 0] = -1
print(tensor1)
print(numpy_array)
```

2) `Tensor.from_numpy(ndarray)` 将 ndarray 转换为 Tensor，共享内存。使用 `copy()` 避免共享内存

```
import torch
import numpy as np

# 使用 from_numpy() 方法将 ndarray 转换为 Tensor，共享内存
numpy_array = np.random.randn(3)
tensor1 = torch.from_numpy(numpy_array)
print(numpy_array)
print(tensor1)
print()
numpy_array[0] = 100
print(numpy_array)
print(tensor1)
print()

# 使用 copy() 方法避免共享内存
tensor1 = torch.from_numpy(numpy_array.copy())
numpy_array[0] = -1
print(numpy_array)
```

```
print(tensor1)
```

3) `torch.tensor(ndarray)`将 ndarray 转换为 Tensor, 不共享内存

```
import torch
import numpy as np

# 使用 torch.tensor() 将 ndarray 转换为 Tensor
numpy_array = np.random.randn(3)
tensor1 = torch.tensor(numpy_array)
print(numpy_array)
print(tensor1)
print()
numpy_array[0] = 100
print(numpy_array)
print(tensor1)
```

### 6.4.3 Tensor 与标量转换

若张量中只有 1 个元素, `Tensor.item()`可提取张量中元素为标量。

```
import torch

tensor1 = torch.tensor(1)
print(tensor1)
print(tensor1.item())
```

## 6.5 张量数值计算

### 6.5.1 基本运算

#### 1) 四则运算

- `+`、`-`、`*`、`/`加减乘除
- `add()`、`sub()`、`mul()`、`div()`加减乘除, 不改变原数据
- `add_()`、`sub_()`、`mul_()`、`div_()`加减乘除、修改原数据

```
import torch

tensor1 = torch.randint(1, 9, (2, 3))
print(tensor1)
print(tensor1 + 10)
print()

# add(), 不修改原数据
print(tensor1.add(10))
print(tensor1)
print()
```

```
# add_(), 修改原数据
print(tensor1.add_(10))
print(tensor1)
```

#### 2) -、neg()、neg\_()取负

```
import torch

tensor1 = torch.tensor([1, 2, 3])
print(-tensor1)
print()

print(tensor1.neg())
print(tensor1)
print()

print(tensor1.neg_())
print(tensor1)
```

#### 3) \*\*、pow()、pow\_()求幂

```
import torch

tensor1 = torch.tensor([1, 2, 3])
print(tensor1**2)
print()

print(tensor1.pow(2))
print(tensor1)
print()

print(tensor1.pow_(2))
print(tensor1)
```

#### 4) sqrt()、sqrt\_()求平方根

```
import torch

tensor1 = torch.tensor([1.0, 2.0, 3.0])
print(tensor1.sqrt())
print(tensor1)
print()

print(tensor1.sqrt_())
print(tensor1)
```

#### 5) exp()、exp\_()以 e 为底数求幂

```
import torch
```

```
tensor1 = torch.tensor([1.0, 2.0, 3.0])
print(2.71828183**tensor1)
print()

print(tensor1.exp())
print(tensor1)
print()

print(tensor1.exp_())
print(tensor1)
```

6) `log()`、`log_()`以 e 为底求对数

```
import torch

tensor1 = torch.tensor([1.0, 2.0, 3.0])
print(tensor1.log())
print(tensor1)
print()

print(tensor1.log_())
print(tensor1)
```

## 6.5.2 哈达玛积（元素级乘法）

两个矩阵对应位置元素相乘称为哈达玛积（Hadamard product）。

使用`*`、`mul()`实现两个形状相同的张量之间对位相乘。

```
import torch

tensor1 = torch.tensor([[1, 2], [3, 4]])
tensor2 = torch.tensor([[1, 2], [3, 4]])
print(tensor1 * tensor2)
print(tensor1.mul(tensor2))
```

## 6.5.3 矩阵乘法运算

`mm()`严格用于二维矩阵相乘。

`@、matmul()`支持多维张量，按最后两个维度做矩阵乘法，其他维度广播。

```
import torch

# 2 维矩阵的矩阵乘法
tensor1 = torch.tensor([[1, 2, 3], [4, 5, 6]])
tensor2 = torch.tensor([[1, 2], [3, 4], [5, 6]])
print(tensor1)
```

```
print(tensor2)
print(tensor1.mm(tensor2))
print(tensor1 @ tensor2)
print(tensor1.matmul(tensor2))
print()

# 3 维张量的矩阵乘法
tensor1 = torch.tensor([[[1, 2, 3], [4, 5, 6]], [[6, 5, 4], [3, 2, 1]]])
tensor2 = torch.tensor([[[1, 2], [3, 4], [5, 6]], [[6, 5], [4, 3], [2, 1]]])
print(tensor1)
print(tensor2)
print(tensor1 @ tensor2)
print(tensor1.matmul(tensor2))
```

## 6.5.4 节省内存

运行一些操作时可能导致为新的结果分配内存，例如 `X=X@Y`，发现 `id(X)` 会指向另一个位置，这是因为 Python 首先计算 `X@Y`，为结果分配新的内存，再令 `X` 指向内存中的新位置。

```
import torch

X = torch.randint(1, 9, (3, 2, 4))
Y = torch.randint(1, 9, (3, 4, 1))
print(id(X))
X = X @ Y
print(id(X), end="\n\n")
```

如果后续 `X` 不再重复使用，可以使用 `X[:] = X @ Y` 来减少内存开销。

```
import torch

X = torch.randint(1, 9, (3, 2, 4))
Y = torch.randint(1, 9, (3, 4, 1))
print(id(X))
X[:] = X @ Y
print(id(X))
```

## 6.6 张量运算函数

常见运算函数：

`sum()`求和

`mean()`求均值

`max() / min()`求最大/最小值及其索引

`argmax()/argmin()`求最大值/最小值的索引

`std()`求标准差

`unique()`去重

`sort()`排序

```
import torch

tensor1 = torch.randint(1, 9, (3, 2, 4))
tensor1 = tensor1.float()
print(tensor1)
print()

# sum() 求和
print("求和")
print(tensor1.sum())
print("按第 0 个维度求和")
print(tensor1.sum(dim=0))
print()

# mean() 求均值
print("求均值")
print(tensor1.mean())
print("按第 1 个维度求均值")
print(tensor1.mean(dim=1))
print()

# max() 求最大值
print("求最大值")
print(tensor1.max())
print("按第 2 个维度求最大值与索引")
print(tensor1.max(dim=2))
print()

# argmin() 求最小值索引
print("求最小值索引")
print(tensor1.argmin())
print()

# std() 求标准差
print("求标准差")
print(tensor1.std())
print()
```

```
# unique() 去重
print("去重")
print(tensor1.unique())
print()

# sort() 排序
print("排序")
print(tensor1.sort())
```

## 6.7 张量索引操作

### 6.7.1 简单索引

```
import torch

tensor1 = torch.randint(1, 9, (3, 5, 4))
print(tensor1)
print()

# 取 第 0 维第 0
print(tensor1[0])
print()

# 取 第 0 维所有, 第 1 维第 1
print(tensor1[:, 1])
print()

# 取 第 0 维所有, 第 1 维第 1, 第 2 维第 3
print(tensor1[2, 1, 3])
```

### 6.7.2 列表索引

```
import torch

tensor1 = torch.randint(1, 9, (3, 5, 4))
print(tensor1)
print()

# 取 第 0 维第 0, 第 1 维第 1 和 第 0 维第 1, 第 1 维第 2
print(tensor1[[0, 1], [1, 2]])
print()

# 取 第 0 维第 0, 第 1 维第 1、2 和 第 0 维第 1, 第 1 维第 1、2
print(tensor1[[[0], [1]], [1, 2]])
```

### 6.7.3 范围索引

```
import torch

tensor1 = torch.randint(1, 9, (3, 5, 4))
print(tensor1)
print()

# 取 第 0 维第 1 到最后
print(tensor1[1:])
print()

# 取 第 0 维最后, 第 1 维 1 到 3(包含 3), 第 2 维 0 到 2(包含 2)
print(tensor1[-1:, 1:4, 0:3])
print()
```

## 6.7.4 布尔索引

```
import torch

tensor1 = torch.randint(1, 9, (3, 5, 4))
print(tensor1)
print()

# 取 第 2 维第 0 大于 5 的, 返回(dim0,dim1)形状的索引
print(tensor1[:, :, 0] > 5)
print(tensor1[tensor1[:, :, 0] > 5])
print()

# 取 第 1 维第 1 大于 5 的, 返回(dim0,dim2)形状的索引
mask = tensor1[:, 1, :] > 5
print(mask)
tensor2 = tensor1.permute(0, 2, 1) # 转换维度为(dim0,dim2,dim1)
print(tensor2[mask])
tensor2 = tensor2[mask].permute(1, 0) # 转换维度为(dim1,?)
print(tensor2)
print()

# 取 第 1 维第 1, 第 2 维第 2 大于 5 的, 返回(dim0)形状的索引
print(tensor1[:, 1, 2] > 5)
print(tensor1[tensor1[:, 1, 2] > 5])
```

## 6.8 张量形状操作

### 6.8.1 交换维度

#### 1) `transpose()`交换两个维度

```
import torch

tensor1 = torch.randint(1, 9, (2, 3, 6))
print(tensor1)
print(tensor1.transpose(1, 2)) # 交换第 1 维和第 2 维
```

## 2) `permute()`重新排列多个维度

```
import torch

tensor1 = torch.randint(1, 9, (2, 3, 6))
print(tensor1)
print(tensor1.permute(2, 0, 1)) # (2, 3, 6)->(6, 2, 3)
```

## 6.8.2 调整形状

### 1) `reshape()`调整张量的形状

```
import torch

tensor1 = torch.randint(1, 9, (3, 5, 4))
print(tensor1)
print(tensor1.reshape(6, 10))
print(tensor1.reshape(3, -1))
```

### 2) `view()`调整张量的形状，需要内存连续。共享内存

`is_contiguous()`判断是否内存连续

`contiguous()`转换为内存连续

```
import torch

tensor1 = torch.randint(1, 9, (3, 5, 4))
print(tensor1)
print(tensor1.is_contiguous()) # is_contiguous()判断是否内存连续
print(tensor1.view(-1, 10))

tensor1 = tensor1.T
print(tensor1.is_contiguous()) # is_contiguous()判断是否内存连续
print(tensor1.contiguous().view(-1)) # contiguous()强制内存连续
```

## 6.8.3 增加或删除维度

### 1) `unsqueeze()`在指定维度上增加 1 个维度

```
import torch

tensor1 = torch.tensor([1, 2, 3, 4, 5])
print(tensor1)
```

```
# 在 0 维上增加一个维度  
print(tensor1.unsqueeze(dim=0))  
# 在 1 维上增加一个维度  
print(tensor1.unsqueeze(dim=1))  
# 在 -1 维上增加一个维度  
print(tensor1.unsqueeze(dim=-1))
```

## 2) `squeeze()`删除大小为 1 的维度

```
import torch  
  
tensor1 = torch.tensor([1, 2, 3, 4, 5])  
print(tensor1.unsqueeze_(dim=0))  
print(tensor1.squeeze())
```

## 6.9 张量拼接操作

### 1) `torch.cat()`张量拼接，按已有维度拼接。除拼接维度外，其他维度大小须相同

```
import torch  
  
tensor1 = torch.randint(1, 9, (2, 2, 5))  
tensor2 = torch.randint(1, 9, (2, 1, 5))  
print(tensor1)  
print(tensor2)  
print(torch.cat([tensor1, tensor2], dim=1))
```

### 2) `torch.stack()`张量堆叠，按新维度堆叠。所有张量形状必须一致

```
import torch  
  
torch.manual_seed(42)  
tensor1 = torch.randint(1, 9, (3, 1, 5))  
tensor2 = torch.randint(1, 9, (3, 1, 5))  
print(tensor1)  
print(tensor2)  
tensor3 = torch.stack([tensor1, tensor2], dim=2)  
print(tensor3)  
print(tensor3.shape)
```

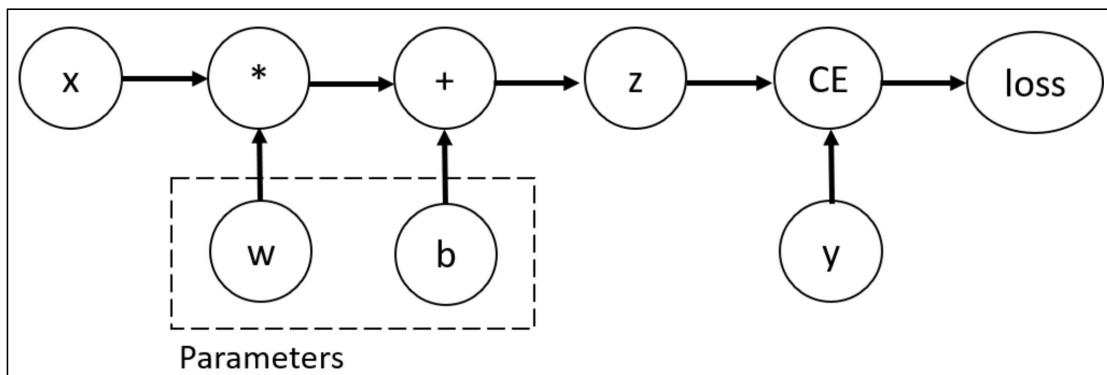
## 6.10 自动微分模块

训练神经网络时，框架会根据设计好的模型构建一个计算图（computational graph），来跟踪计算是哪些数据通过哪些操作组合起来产生输出，并通过反向传播算法来根据给定参数的损失函数的梯度调整参数（模型权重）。

PyTorch 具有一个内置的微分引擎 `torch.autograd` 以支持计算图的梯度自动计算。

考虑最简单的单层神经网络，具有输入 `x`、参数 `w`、偏置 `b` 以及损失函数：

更多 Java - 大数据 - 前端 - python 人工智能资料下载，可百度访问：[尚硅谷官网](#)



```

import torch

# 输入 x
x = torch.tensor(10.0)
# 目标值 y
y = torch.tensor(3.0)

# 初始化权重 w
w = torch.rand(1, 1, requires_grad=True)
# 初始化偏置 b
b = torch.rand(1, 1, requires_grad=True)
z = w * x + b
# 设置损失函数
loss = torch.nn.MSELoss()
loss_value = loss(z, y)
# 反向传播
loss_value.backward()
# 打印 w,b 的梯度
print("w 的梯度:\n", w.grad)
print("b 的梯度:\n", b.grad)
  
```

该计算图中 x、w、b 为叶子节点，即最基础的节点。叶子节点的数据并非由计算生成，因此是整个计算图的基石，叶子节点张量不可以执行 in-place 操作。而最终的 loss 为根节点。

可通过 is\_leaf 属性查看张量是否为叶子节点：

```

print(x.is_leaf) # True
print(w.is_leaf) # True
print(b.is_leaf) # True
print(z.is_leaf) # False
print(y.is_leaf) # True
print(loss_value.is_leaf) # False
  
```

自动微分的关键就是记录节点的数据与运算。数据记录在张量的 data 属性中，计算记录在张量的 grad\_fn 属性中。

计算图根据搭建方式可分为静态图和动态图，PyTorch 是动态图机制，在计算的过程中逐步搭建计算图，同时对每个 Tensor 都存储 grad\_fn 供自动微分使用。当计算到根节点后，在根节点调用 backward()方法即可反向传播计算图中所有节点的梯度。

非叶子节点的梯度在反向传播之后会被释放掉（除非设置参数 retain\_grad=True）。而叶子节点的梯度在反向传播之后会保留（累积）。通常需要使用 optimizer.zero\_grad()清零参数的梯度。

若设置张量参数 requires\_grad=True，则 PyTorch 会追踪所有基于该张量的操作，并在反向传播时计算其梯度。依赖于叶子节点的节点，requires\_grad 默认为 True。

有时我们希望将某些计算移动到计算图之外，可以使用 Tensor.detach()返回一个新的变量，该变量与原变量具有相同的值，但丢失计算图中如何计算原变量的信息。换句话说，梯度不会在该变量处继续向下传播。例如：

```
import torch

x = torch.ones(2, 2, requires_grad=True)
y = x * x
# 分离 y 来返回一个新变量 u
u = y.detach()
z = u * x
# 梯度不会向后流经 u 到 x
z.sum().backward()
# 反向传播函数计算 z=u*x 关于 x 的偏导数时将 u 作为常数处理，而不是 z=x*x*x 关于 x 的偏导数
x.grad == u
# tensor([[True, True],
#         [True, True]])
```

## 6.11 机器学习案例：线性回归

在此虚拟环境中没有 matplotlib，先安装：`pip install matplotlib`。

通过 PyTorch 训练一个模型一般分为以下 4 个步骤：

准备数据→构建模型→定义损失函数与优化器→模型训练

接下来，构建数据集并使用 PyTorch 实现线性回归：

```
import torch
import matplotlib.pyplot as plt
from torch import nn, optim # 模型、损失函数和优化器
from torch.utils.data import TensorDataset, DataLoader # 数据集和数据加载器
```

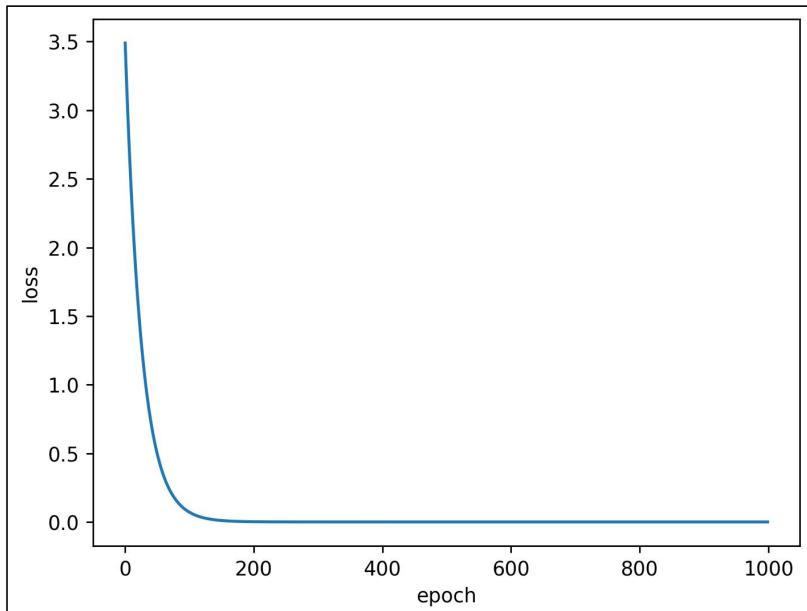
```
# 构建数据集
X = torch.randn(100, 1) # 输入
w = torch.tensor([2.5]) # 权重
b = torch.tensor([5.2]) # 偏置
noise = torch.randn(100, 1) * 0.1 # 噪声
y = w * X + b + noise # 目标
dataset = TensorDataset(X, y) # 构造数据集对象
dataloader = DataLoader(
    dataset, batch_size=10, shuffle=True
) # 构造数据加载器对象, batch_size 为每次训练的样本数, shuffle 为是否打乱数据

# 构造模型
model = nn.Linear(in_features=1, out_features=1) # 线性回归模型, 1 个输入,
1 个输出

# 损失函数和优化器
loss = nn.MSELoss() # 均方误差损失函数
optimizer = optim.SGD(model.parameters(), lr=1e-3) # 随机梯度下降, 学习率
0.001

# 模型训练
loss_list = []
for epoch in range(1000):
    total_loss = 0
    train_num = 0
    for x_train, y_train in dataloader:
        # 每次训练一个 batch 大小的数据
        y_pred = model(x_train) # 模型预测
        loss_value = loss(y_pred, y_train) # 计算损失
        total_loss += loss_value.item()
        train_num += len(y_train)
        optimizer.zero_grad() # 梯度清零
        loss_value.backward() # 反向传播
        optimizer.step() # 更新参数
    loss_list.append(total_loss / train_num)

print(model.weight, model.bias) # 打印权重和偏置
plt.plot(loss_list)
plt.xlabel("epoch")
plt.ylabel("loss")
plt.show()
```



## 第 7 章 用 PyTorch 进行深度学习

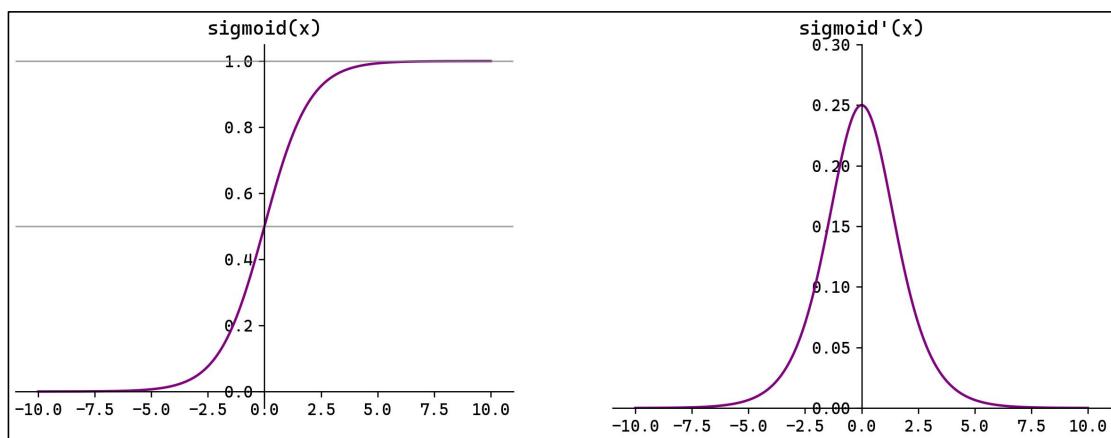
### 7.1 激活函数

PyTorch 中已经实现了神经网络中可能用到的各种激活函数，我们在代码中只要直接调用即可。

#### 7.1.1 Sigmoid 函数

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

$$f'(x) = \frac{1}{1 + e^{-x}} \left(1 - \frac{1}{1 + e^{-x}}\right) = f(x)(1 - f(x))$$



Sigmoid 函数与其导数图像绘制代码：

```
import torch
import matplotlib.pyplot as plt
```

```

x = torch.linspace(-10, 10, 1000, requires_grad=True)
fig, ax = plt.subplots(1, 2)
fig.set_size_inches(12, 4)

ax[0].plot(x.data, torch.sigmoid(x).data, "purple")
ax[0].set_title("sigmoid(x)")
ax[0].spines["top"].set_visible(False)
ax[0].spines["right"].set_visible(False)
ax[0].spines["left"].set_position("zero")
ax[0].spines["bottom"].set_position("zero")
ax[0].axhline(0.5, color="gray", alpha=0.7, linewidth=1)
ax[0].axhline(1, color="gray", alpha=0.7, linewidth=1)

torch.sigmoid(x).sum().backward() # 反向传播计算梯度
ax[1].plot(x.data, x.grad, "purple")
ax[1].set_title("sigmoid'(x)")
ax[1].spines["top"].set_visible(False)
ax[1].spines["right"].set_visible(False)
ax[1].spines["left"].set_position("zero")
ax[1].spines["bottom"].set_position("zero")
ax[1].set_ylim(0, 0.3)

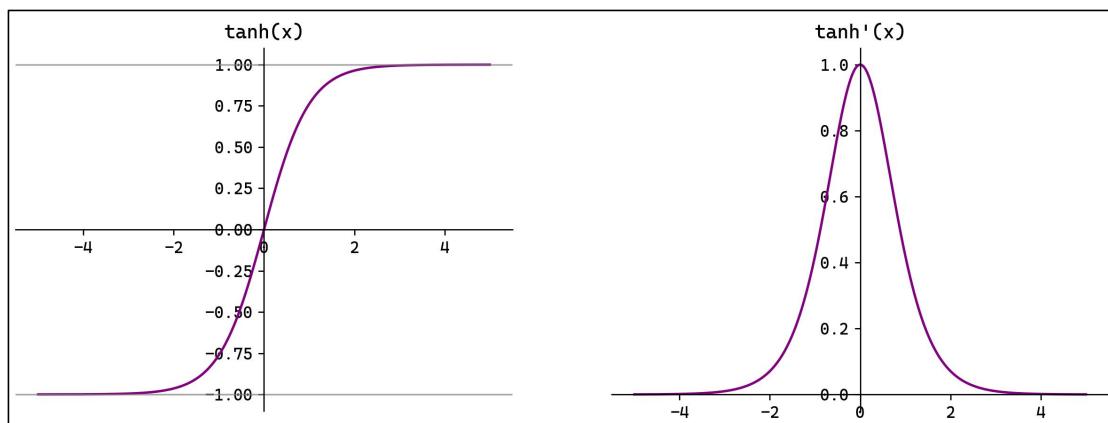
plt.show()

```

### 7.1.2 Tanh 函数

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

$$f'(x) = 1 - \left(\frac{1 - e^{-2x}}{1 + e^{-2x}}\right)^2 = 1 - f^2(x)$$



Tanh 函数与其导数图像绘制代码:

```
import torch
```

```

import matplotlib.pyplot as plt

x = torch.linspace(-5, 5, 1000, requires_grad=True)
fig, ax = plt.subplots(1, 2)
fig.set_size_inches(12, 4)

ax[0].plot(x.data, torch.tanh(x).data, "purple")
ax[0].set_title("tanh(x)")
ax[0].spines["top"].set_visible(False)
ax[0].spines["right"].set_visible(False)
ax[0].spines["left"].set_position("zero")
ax[0].spines["bottom"].set_position("zero")
ax[0].axhline(-1, color="gray", alpha=0.7, linewidth=1)
ax[0].axhline(1, color="gray", alpha=0.7, linewidth=1)

torch.tanh(x).sum().backward() # 反向传播计算梯度
ax[1].plot(x.data, x.grad, "purple")
ax[1].set_title("tanh'(x)")
ax[1].spines["top"].set_visible(False)
ax[1].spines["right"].set_visible(False)
ax[1].spines["left"].set_position("zero")
ax[1].spines["bottom"].set_position("zero")

plt.show()

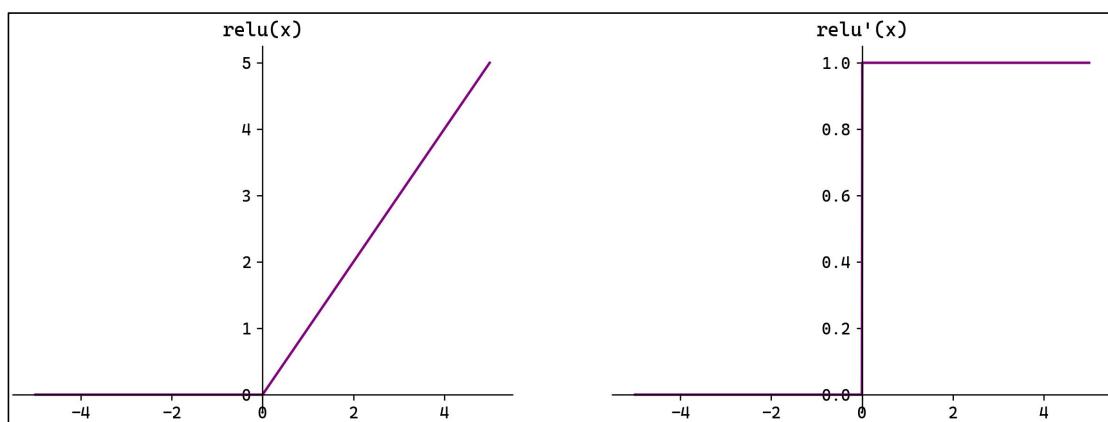
```

### 7.1.3 ReLU 函数

$$f(x) = \max(0, x)$$

$$f'(x) = \begin{cases} 0, & x \leq 0 \\ 1, & x > 0 \end{cases}$$

注意：x=0 时 ReLU 函数不可导，此时我们默认使用左侧的函数。



ReLU 函数与其导数图像绘制代码：

```
import torch
import matplotlib.pyplot as plt

x = torch.linspace(-5, 5, 1000, requires_grad=True)
fig, ax = plt.subplots(1, 2)
fig.set_size_inches(12, 4)

ax[0].plot(x.data, torch.relu(x).data, "purple")
ax[0].set_title("relu(x)")
ax[0].spines["top"].set_visible(False)
ax[0].spines["right"].set_visible(False)
ax[0].spines["left"].set_position("zero")
ax[0].spines["bottom"].set_position("zero")

torch.relu(x).sum().backward() # 反向传播计算梯度
ax[1].plot(x.data, x.grad, "purple")
ax[1].set_title("relu'(x)")
ax[1].spines["top"].set_visible(False)
ax[1].spines["right"].set_visible(False)
ax[1].spines["left"].set_position("zero")
ax[1].spines["bottom"].set_position("zero")

plt.show()
```

#### 7.1.4 Softmax 函数

$$f(x_i) = \frac{e^{z_i}}{\sum_{j=1}^n e^{z_j}}$$
$$\frac{\partial f(x_i)}{\partial z_j} = \begin{cases} f(x_i) (1 - f(x_j)), & i = j \\ -f(x_i)f(x_j), & i \neq j \end{cases}$$

Softmax 函数直接调用即可，代码略。

### 7.2 搭建神经网络

#### 7.2.1 自定义模型

在神经网络框架中，由多个层组成的组件称之为 **模块（Module）**。

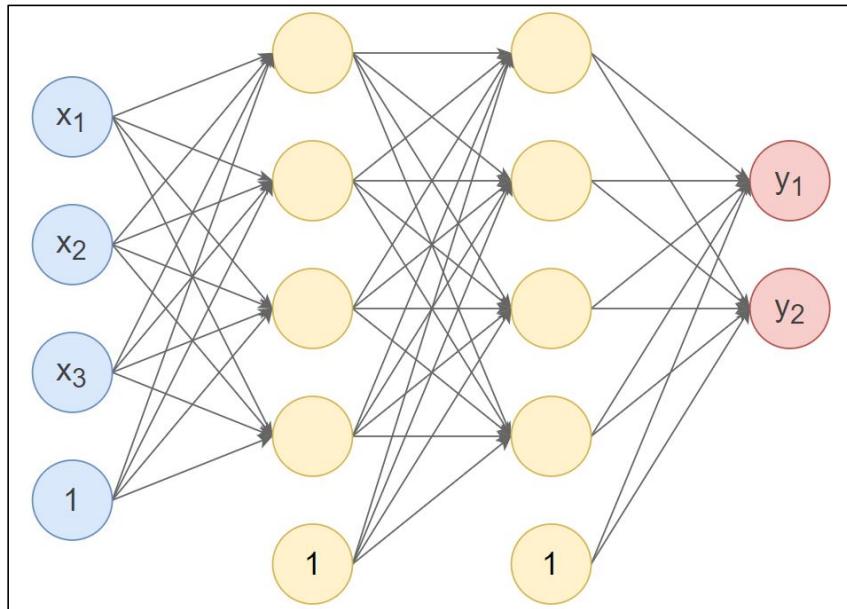
在 PyTorch 中模型就是一个 Module，各网络层、模块也是 Module。Module 是所有神经网络的基类。

在定义一个 Module 时，我们需要继承 torch.nn.Module 并主要实现两个方法：

- `__init__`: 定义网络各层的结构，并初始化参数。

- forward: 根据输入进行前向传播，并返回输出。计算其输出关于输入的梯度，可通过其反向传播函数进行访问（通常自动发生）。forward 方法是每次调用的具体实现。

接下来使用 PyTorch 实现下图的神经网络：



第 1 个隐藏层：使用 Xavier 正态分布初始化权重，激活函数使用 Tanh。

第 2 个隐藏层：使用 He 正态分布初始化权重，激活函数使用 ReLU。

输出层：按默认方式初始化，激活函数使用 Softmax。

```
import torch
import torch.nn as nn

class Model(nn.Module):
    # 初始化
    def __init__(self):
        super(Model, self).__init__() # 调用父类初始化
        self.linear1 = nn.Linear(3, 4) # 第 1 个隐藏层，3 个输入，4 个输出
        nn.init.xavier_normal_(self.linear1.weight) # 初始化权重参数
        self.linear2 = nn.Linear(4, 4) # 第 2 个隐藏层，4 个输入，4 个输出
        nn.init.kaiming_normal_(self.linear2.weight) # 初始化权重参数
        self.out = nn.Linear(4, 2) # 输出层，4 个输入，2 个输出， 默认使用 He 均匀分布初始化

    # 前向传播
    def forward(self, x):
        x = self.linear1(x) # 经过第 1 个隐藏层
        x = torch.tanh(x) # 激活函数
        x = self.linear2(x) # 经过第 2 个隐藏层
```

```
x = torch.relu(x) # 激活函数
x = self.out(x) # 经过输出层
x = torch.softmax(x, dim=1) # 激活函数
return x

model = Model()
output = model(torch.randn(10, 3))
print("输出: \n", output)
print()

# 使用 named_parameters() 查看各层参数
print("模型参数: ")
for name, param in model.named_parameters():
    print(name, param)
    print()

# 使用 state_dict() 查看各层参数
print("模型参数: \n", model.state_dict())
```

## 7.2.2 查看模型结构和参数数量

可使用 `torchsummary.summary` 来查看模型结构与参数数量。需要先安装 `torchsummary` 库: `pip install torchsummary`。

```
from torchsummary import summary

# input_size: 特征数, batch_size: 样本数
summary(model, input_size=(3,), batch_size=10, device="cpu")
```

| Layer (type)                          | Output Shape | Param # |
|---------------------------------------|--------------|---------|
| <hr/>                                 |              |         |
| Linear-1                              | [10, 4]      | 16      |
| Linear-2                              | [10, 4]      | 20      |
| Linear-3                              | [10, 2]      | 10      |
| <hr/>                                 |              |         |
| Total params: 46                      |              |         |
| Trainable params: 46                  |              |         |
| Non-trainable params: 0               |              |         |
| <hr/>                                 |              |         |
| Input size (MB): 0.00                 |              |         |
| Forward/backward pass size (MB): 0.00 |              |         |
| Params size (MB): 0.00                |              |         |
| Estimated Total Size (MB): 0.00       |              |         |

以第 1 个隐藏层为例: 每个节点有 3 个权重与 1 个偏置, 计 4 个参数, 4 个节点共计 16 个参数。

### 7.2.3 使用 Sequential 构建模型

可以通过 `torch.nn.Sequential` 来构建模型，将各层按顺序传入。

```
# 构建模型
model = nn.Sequential(
    nn.Linear(3, 4),
    nn.Tanh(),
    nn.Linear(4, 4),
    nn.ReLU(),
    nn.Linear(4, 2),
    nn.Softmax(dim=1),
)

# 初始化参数
def init_weights(m):
    # 对 Linear 层进行初始化
    if type(m) == nn.Linear:
        nn.init.xavier_uniform_(m.weight)
        m.bias.data.fill_(0.01)

model.apply(init_weights) # apply 会遍历所有子模块并依次调用函数

output = model(torch.randn(10, 3))
print("输出: \n", output)
```

`Sequential` 类使模型构造变得简单，不必自定义类就可以组合新的架构。然而并不是所有的架构都是简单的顺序架构，当需要更强的灵活性时还是需要自定义模型。

## 7.3 损失函数

### 7.3.1 分类任务损失函数

#### 1) 二分类任务损失函数

二分类任务常用二元交叉熵损失函数（Binary Cross-Entropy Loss）。

$$L = -\frac{1}{n} \sum_{i=1}^n (y_i \log \hat{y}_i + (1 - y_i) \log (1 - \hat{y}_i))$$

其中：

- $y_i$  为真实值（通常为 0 或 1）
- $\hat{y}_i$  为预测值（表示样本  $i$  为 1 的概率）

在 PyTorch 中可使用 torch.nn.BCELoss 实现：

```
import torch
import torch.nn as nn

# 真实值
target = torch.tensor([[1], [0], [0]], dtype=torch.float32)
# 预测值
input = torch.randn((3, 1))
prediction = torch.sigmoid(input)
# 实例化损失函数
loss = nn.BCELoss()
print(loss(prediction, target))
```

## 2) 多分类任务损失函数

多分类任务常用多类交叉熵损失函数（Categorical Cross-Entropy Loss）。它是对每个类别的预测概率与真实标签之间差异的加权平均。

$$L = -\frac{1}{n} \sum_{i=1}^n \sum_{c=1}^C y_{i,c} \log \hat{y}_{i,c}$$

其中：

- C 是类别数
- $y_{i,c}$  为真实值（表示  $y_i$  是否为类别 c，通常为 0 或 1）
- $\hat{y}_{i,c}$  为预测值（表示样本  $i$  为类别 c 的概率）

在 PyTorch 中可使用 torch.nn.CrossEntropyLoss 实现：

注意：调用 torch.nn.CrossEntropyLoss 相当于调用了 torch.nn.LogSoftmax 之后再调用 torch.nn.NLLLoss。即使用 CrossEntropyLoss 时上一层的输出不需要 Softmax 激活函数，因为该损失函数内会自动处理。

```
import torch
import torch.nn as nn

# 真实值为标签
target = torch.tensor([1, 0, 3, 2, 5, 4]) # 真实值
input = torch.randn((6, 8)) # 预测值
loss = nn.CrossEntropyLoss() # 实例化损失函数
print(loss(input, target))

# 真实值为概率
target = torch.randn(6, 8).softmax(dim=1) # 真实值
```

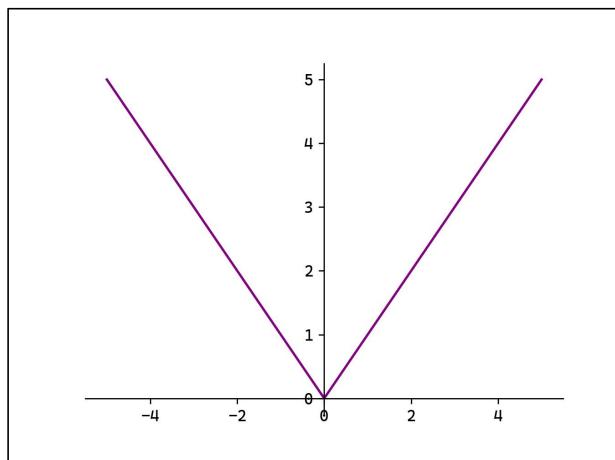
```
input = torch.randn((6, 8)) # 预测值  
loss = nn.CrossEntropyLoss() # 实例化损失函数  
print(loss(input, target))
```

### 7.3.2 回归任务损失函数

#### 1) MAE

平均绝对误差（Mean Absolute Error，MAE），也称 L1 Loss:

$$L = \frac{1}{n} \sum_{i=1}^n |y_i - \hat{y}_i|$$

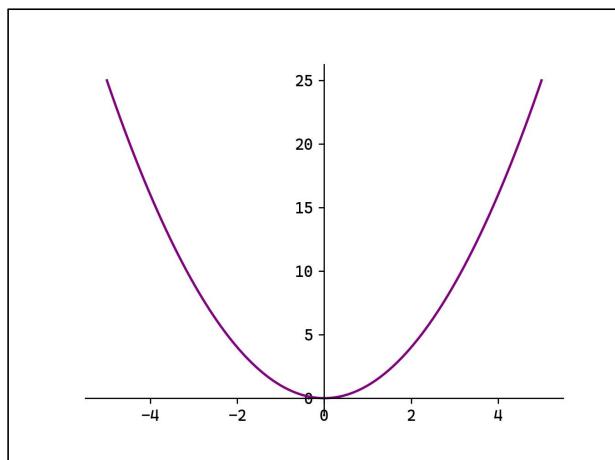


L1 Loss 对异常值鲁棒，但在 0 点处不可导。

#### 2) MSE

均方误差（Mean Squared Error，MSE），也称 L2 Loss:

$$L = \frac{1}{n} \sum_{i=1}^n (y_i - \hat{y}_i)^2$$

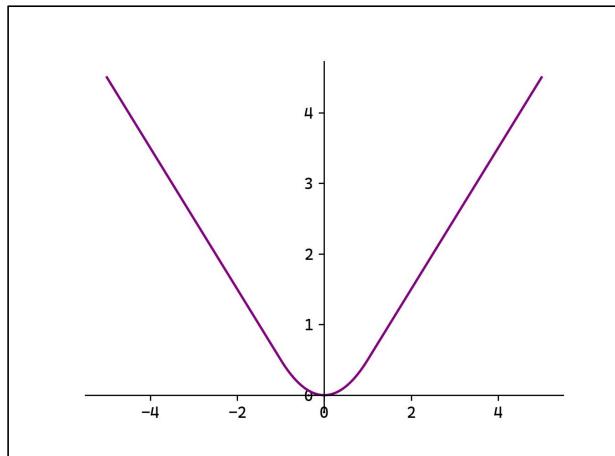


L2 Loss 对异常值敏感，遇到异常值时易发生梯度爆炸。

### 3) Smooth L1

平滑 L1:

$$\text{Smooth L1} = \begin{cases} \frac{1}{2}(y_i - \hat{y}_i)^2, & |y_i - \hat{y}_i| < 1 \\ |y_i - \hat{y}_i| - \frac{1}{2}, & |y_i - \hat{y}_i| \geq 1 \end{cases}$$



当误差较小时( $|y_i - \hat{y}_i| < 1$ )使用 L2 Loss, 使得损失函数平滑可导。当误差较大时( $|y_i - \hat{y}_i| \geq 1$ ) 使用 L1 Loss 降低异常值的影响。

示例代码:

```
import torch
from torch import nn, optim

class Model(nn.Module):
    # 初始化
    def __init__(self):
        # 调用父类初始化
        super(Model, self).__init__()
        # 全连接层
        self.linear1 = nn.Linear(5, 3)
        # 初始化权重
        self.linear1.weight.data = torch.tensor(
            [
                [0.1, 0.2, 0.3],
                [0.4, 0.5, 0.6],
                [0.7, 0.8, 0.9],
                [0.10, 1.1, 1.2],
                [1.3, 1.4, 1.5],
            ]
        )
```

```
        ]
    ).T
# 初始化偏置
self.linear1.bias.data = torch.tensor([1.0, 2.0, 3.0])

# 前向传播
def forward(self, x):
    x = self.linear1(x)
    return x

# 实例化模型
model = Model()
# 输入值
X = torch.tensor([[1, 2, 3, 4, 5], [6, 7, 8, 9, 10]], dtype=torch.float)
# 目标值
target = torch.tensor([[0, 0, 0], [0, 0, 0]], dtype=torch.float)
# 计算出输出值
output = model(X)
# 损失函数
loss = nn.MSELoss()
# 反向传播
loss(output, target).backward()
# 优化器
optimizer = optim.SGD(model.parameters(), lr=1)
# 更新参数
optimizer.step()
# 清空梯度
optimizer.zero_grad()
# 打印参数
for i in model.state_dict():
    print(i)
    print(model.state_dict()[i])
print()

# 根据计算图手动计算梯度并更新参数
E = (output - target) / 3
print(E)
weight = torch.tensor(
    [
        [0.1, 0.2, 0.3],
        [0.4, 0.5, 0.6],
        [0.7, 0.8, 0.9],
        [0.10, 1.1, 1.2],
    ]
)
```

```
[1.3, 1.4, 1.5],  
]  
)  
weight = weight - X.T @ E  
print("weight\n", weight.T)  
bias = torch.tensor([1.0, 2.0, 3.0])  
bias = bias - E.sum(dim=0)  
print("bias\n", bias)
```

## 7.4 参数更新方法

### 7.4.1 Momentum

Momentum（动量法）会保存历史梯度并给予一定的权重，使其也参与到参数更新中：

$$V \leftarrow \alpha V - \eta \nabla$$

$$W \leftarrow W + V$$

- $V$ : 历史(负)梯度的加权和
- $\alpha$ : 历史梯度的权重
- $\nabla$ : 当前梯度，即  $\frac{\partial L}{\partial W}$
- $\eta$ : 学习率

可以通过 `torch.optim.SGD()` 并设置 `momentum` 历史梯度权重参数来使用动量法。

以寻找  $f(x_1, x_2) = 0.05x_1^2 + x_2^2$  的最小值为例：

```
import torch  
import numpy as np  
import matplotlib.pyplot as plt  
  
def momentum(X, lr, momentum, n_iters):  
    """动量法手动实现"""  
    X_arr = X.detach().numpy().copy() # 拷贝，用于记录优化过程  
    V = torch.zeros_like(X) # 存放历史梯度信息  
    for epoch in range(n_iters):  
        grad = 2 * X * w.T # 当前梯度  
        V = momentum * V + grad # 加权求和  
        V = V.squeeze() # 降维  
        X.data -= lr * V # 更新参数  
        X_arr = np.vstack([X_arr, X.detach().numpy()]) # 记录优化过程  
    return X_arr
```

```
def gradient_descent(X, optimizer, n_iters):
    X_arr = X.detach().numpy().copy() # 拷贝，用于记录优化过程
    for epoch in range(n_iters):
        y = X**2 @ w
        y.backward() # 反向传播
        optimizer.step() # 更新参数
        optimizer.zero_grad() # 清空梯度
        X_arr = np.vstack([X_arr, X.detach().numpy()]) # 记录优化过程
    return X_arr

# 从(-7, 2)出发
X = torch.tensor([-7, 2], dtype=torch.float32, requires_grad=True)
w = torch.tensor([[0.05], [1.0]], requires_grad=True)
lr = 1e-2 # 学习率
n_iters = 500 # 迭代次数

# 普通梯度下降
X_clone = X.clone().detach().requires_grad_(True)
X_arr1 = gradient_descent(X_clone, torch.optim.SGD([X_clone], lr=lr),
n_iters=n_iters)
plt.plot(X_arr1[:, 0], X_arr1[:, 1], "r")

# 动量法
X_clone = X.clone().detach().requires_grad_(True)
X_arr2 = gradient_descent(X_clone, torch.optim.SGD([X_clone], lr=lr,
momentum=0.9), n_iters=n_iters)
plt.plot(X_arr2[:, 0], X_arr2[:, 1], "b")

# 动量法手动实现
X_clone = X.clone().detach().requires_grad_(True)
X_arr1 = momentum(X_clone, lr=lr, momentum=0.9, n_iters=n_iters)
plt.plot(X_arr1[:, 0], X_arr1[:, 1], c="orange", linestyle="--",
linewidth=3)

# 绘制等高线图
x1_grid, x2_grid = np.meshgrid(np.linspace(-7, 7, 100), np.linspace(-2, 2,
100))
y_grid = w.detach().numpy()[0, 0] * x1_grid**2 + w.detach().numpy()[1, 0]
* x2_grid**2
plt.contour(x1_grid, x2_grid, y_grid, levels=30, colors="gray")
plt.legend(["SGD", "Momentum", "Manual Momentum"])
plt.show()
```

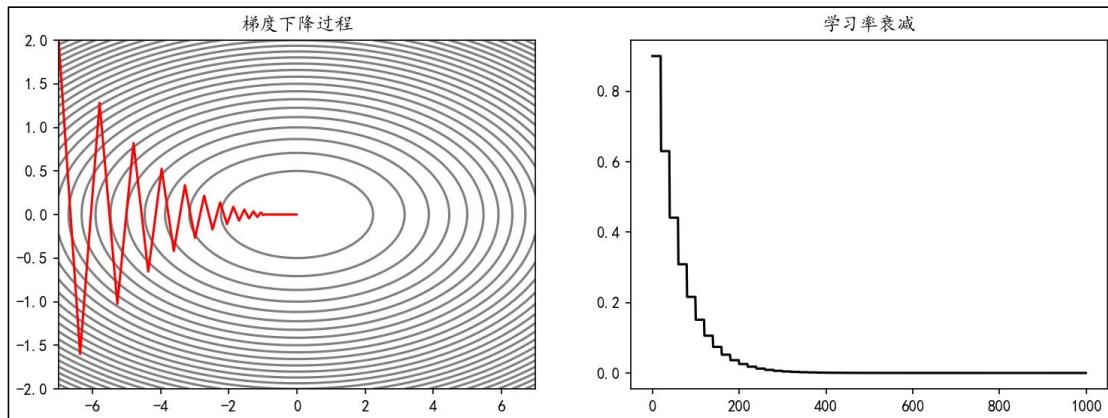
## 7.4.2 学习率衰减

### 1) 等间隔衰减

可以通过 `torch.optim.lr_scheduler.StepLR(optimizer, step_size, gamma)` 来实现学习率的等间隔衰减。

- `optimizer`: 要实现学习率衰减的优化器
- `step_size`: 间隔
- `gamma`: 衰减的比例

例如，使学习率每隔 20 epoch 衰减为之前的 0.7:



```

import torch
import numpy as np
import matplotlib.pyplot as plt

# 从(-7, 2)出发
X = torch.tensor([-7, 2], dtype=torch.float32, requires_grad=True)
w = torch.tensor([[0.05], [1.0]], requires_grad=True)
lr = 0.9 # 初始学习率
n_iters = 1000 # 迭代次数

optimizer = torch.optim.SGD([X], lr=lr)
scheduler_lr = torch.optim.lr_scheduler.StepLR(optimizer, step_size=20,
gamma=0.7) # 学习率衰减
X_arr = X.detach().numpy().copy() # 拷贝，用于记录优化过程
lr_list = [] # 记录学习率变化
for epoch in range(n_iters):
    y = X**2 @ w
    y.backward() # 反向传播
    optimizer.step() # 更新参数
    optimizer.zero_grad() # 清空梯度
    lr_list.append(scheduler_lr.get_lr())

```

```

X_arr = np.vstack([X_arr, X.detach().numpy()]) # 记录优化过程
lr_list.append(optimizer.param_groups[0]["lr"]) # 记录学习率变化
scheduler_lr.step() # 学习率衰减

plt.rcParams["font.sans-serif"] = ["KaiTi"]
plt.rcParams["axes.unicode_minus"] = False
fig, ax = plt.subplots(1, 2, figsize=(12, 4))
x1_grid, x2_grid = np.meshgrid(np.linspace(-7, 7, 100), np.linspace(-2, 2, 100))
y_grid = w.detach().numpy()[0, 0] * x1_grid**2 + w.detach().numpy()[1, 0]
* x2_grid**2
ax[0].contour(x1_grid, x2_grid, y_grid, levels=30, colors="gray")
ax[0].plot(X_arr[:, 0], X_arr[:, 1], "r")
ax[0].set_title("梯度下降过程")

ax[1].plot(lr_list, "k")
ax[1].set_title("学习率衰减")
plt.show()

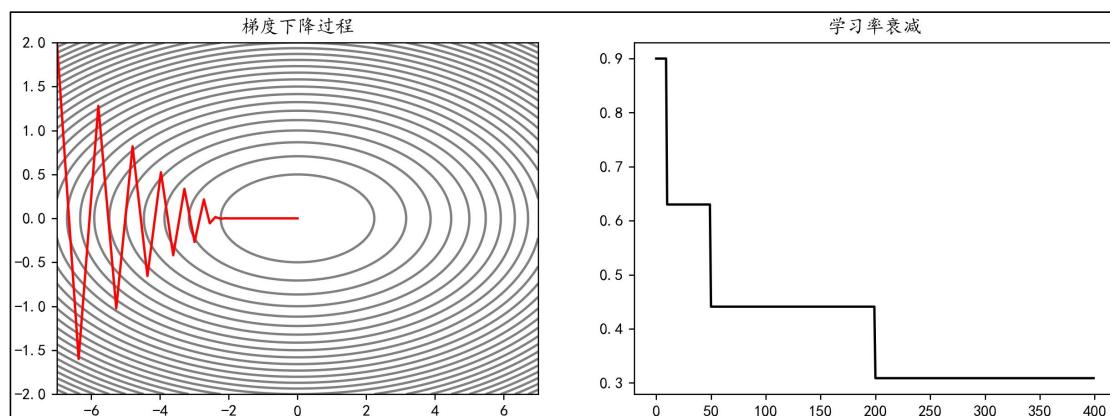
```

## 2) 指定间隔衰减

可以通过 `torch.optim.lr_scheduler.MultiStepLR(optimizer, milestones, gamma)` 来实现学习率的指定间隔衰减。

- `optimizer`: 要实现学习率衰减的优化器
- `milestones`: 指定衰减的间隔
- `gamma`: 衰减的比例

例如，使学习率在 epoch 达到 [10,50,200] 时衰减为之前的 0.7:



```

import torch
import numpy as np
import matplotlib.pyplot as plt

# 从(-7, 2)出发

```

```
X = torch.tensor([-7, 2], dtype=torch.float32, requires_grad=True)
w = torch.tensor([[0.05], [1.0]], requires_grad=True)
lr = 0.9 # 初始学习率
n_iters = 400 # 迭代次数

optimizer = torch.optim.SGD([X], lr=lr)
scheduler_lr = torch.optim.lr_scheduler.MultiStepLR(optimizer,
milestones=[10, 50, 200], gamma=0.7) # 学习率衰减
X_arr = X.detach().numpy().copy() # 拷贝，用于记录优化过程
lr_list = [] # 记录学习率变化
for epoch in range(n_iters):
    y = X**2 @ w
    y.backward() # 反向传播
    optimizer.step() # 更新参数
    optimizer.zero_grad() # 清空梯度
    X_arr = np.vstack([X_arr, X.detach().numpy()]) # 记录优化过程
    lr_list.append(optimizer.param_groups[0]["lr"]) # 记录学习率变化
    scheduler_lr.step() # 学习率衰减

plt.rcParams["font.sans-serif"] = ["KaiTi"]
plt.rcParams["axes.unicode_minus"] = False
fig, ax = plt.subplots(1, 2, figsize=(12, 4))
x1_grid, x2_grid = np.meshgrid(np.linspace(-7, 7, 100), np.linspace(-2, 2, 100))
y_grid = w.detach().numpy()[0, 0] * x1_grid**2 + w.detach().numpy()[1, 0]
* x2_grid**2
ax[0].contour(x1_grid, x2_grid, y_grid, levels=30, colors="gray")
ax[0].plot(X_arr[:, 0], X_arr[:, 1], "r")
ax[0].set_title("梯度下降过程")

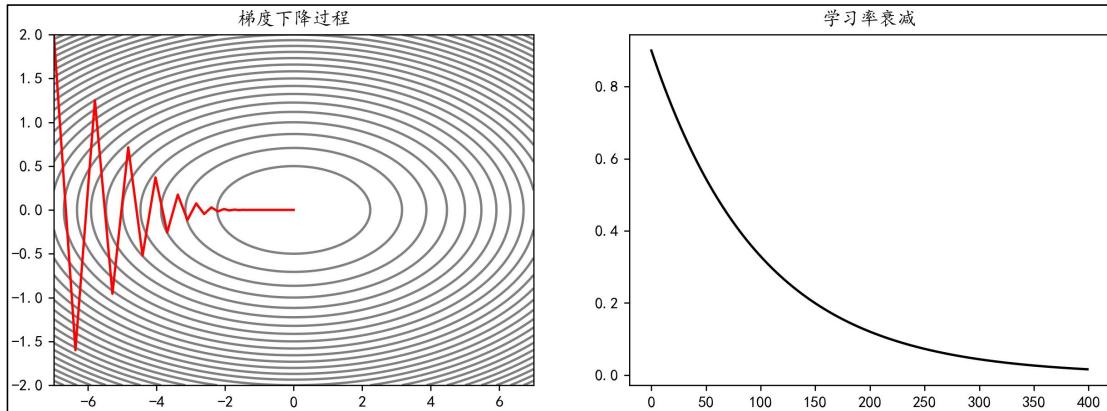
ax[1].plot(lr_list, "k")
ax[1].set_title("学习率衰减")
plt.show()
```

### 3) 指数衰减

可以通过 `torch.optim.lr_scheduler.ExponentialLR(optimizer, gamma)` 来实现学习率的指数衰减。

- `optimizer`: 要实现学习率衰减的优化器
- `gamma`: 底数, 学习率  $\leftarrow$  学习率  $\times \gamma^{epoch}$

例如, 使学习率以 0.99 为底数, `epoch` 为指数衰减:



```

import torch
import numpy as np
import matplotlib.pyplot as plt

# 从(-7, 2)出发
X = torch.tensor([-7, 2], dtype=torch.float32, requires_grad=True)
w = torch.tensor([[0.05], [1.0]], requires_grad=True)
lr = 0.9 # 初始学习率
n_iters = 400 # 迭代次数

optimizer = torch.optim.SGD([X], lr=lr)
scheduler_lr = torch.optim.lr_scheduler.ExponentialLR(optimizer,
gamma=0.99) # 学习率衰减
X_arr = X.detach().numpy().copy() # 拷贝，用于记录优化过程
lr_list = [] # 记录学习率变化
for epoch in range(n_iters):
    y = X**2 @ w
    y.backward() # 反向传播
    optimizer.step() # 更新参数
    optimizer.zero_grad() # 清空梯度
    X_arr = np.vstack([X_arr, X.detach().numpy()]) # 记录优化过程
    lr_list.append(optimizer.param_groups[0]["lr"]) # 记录学习率变化
    scheduler_lr.step() # 学习率衰减

plt.rcParams["font.sans-serif"] = ["KaiTi"]
plt.rcParams["axes.unicode_minus"] = False
fig, ax = plt.subplots(1, 2, figsize=(12, 4))
x1_grid, x2_grid = np.meshgrid(np.linspace(-7, 7, 100), np.linspace(-2, 2, 100))
y_grid = w.detach().numpy()[0, 0] * x1_grid**2 + w.detach().numpy()[1, 0] * x2_grid**2
ax[0].contour(x1_grid, x2_grid, y_grid, levels=30, colors="gray")
ax[0].plot(X_arr[:, 0], X_arr[:, 1], "r")

```

```

ax[0].set_title("梯度下降过程")

ax[1].plot(lr_list, "k")
ax[1].set_title("学习率衰减")
plt.show()

```

### 7.4.3 AdaGrad

AdaGrad (Adaptive Gradient, 自适应梯度) 会为每个参数适当地调整学习率，并且随着学习的进行，梯度会逐渐减小。

$$H \leftarrow H + \nabla^2$$

$$W \leftarrow W - \eta \frac{1}{\sqrt{H}} \nabla$$

➤  $H$ : 历史梯度的平方和

可以通过 `torch.optim.Adagrad()` 来使用 AdaGrad。

同样以寻找  $f(x_1, x_2) = 0.05x_1^2 + x_2^2$  的最小值为例：

```

import torch
import numpy as np
import matplotlib.pyplot as plt


def adagrad(X, lr, n_iters):
    """AdaGrad 手动实现"""
    X_arr = X.detach().numpy().copy() # 拷贝，用于记录优化过程
    H = torch.zeros_like(X) # 存放历史梯度信息
    for epoch in range(n_iters):
        grad = 2 * X * w.T # 当前梯度
        grad.squeeze_() # 降维
        H += grad**2 # 平方和
        X.data -= lr / (torch.sqrt(H) + 1e-8) * grad # 更新参数，这里 H 后面加 1e-8 防止分母为 0
        X_arr = np.vstack([X_arr, X.detach().numpy()]) # 记录优化过程
    return X_arr


def gradient_descent(X, optimizer, n_iters):
    X_arr = X.detach().numpy().copy() # 拷贝，用于记录优化过程
    for epoch in range(n_iters):
        y = X**2 @ w
        y.backward() # 反向传播
        optimizer.step() # 更新参数
        optimizer.zero_grad() # 清空梯度

```

```
X_arr = np.vstack([X_arr, X.detach().numpy()]) # 记录优化过程
return X_arr

# 从(-7, 2)出发
X = torch.tensor([-7, 2], dtype=torch.float32, requires_grad=True)
w = torch.tensor([[0.05], [1.0]], requires_grad=True)
lr = 0.9 # 学习率
n_iters = 500 # 迭代次数

# 普通梯度下降
X_clone = X.clone().detach().requires_grad_(True)
X_arr1 = gradient_descent(X_clone, torch.optim.SGD([X_clone], lr=lr),
n_iters=n_iters)
plt.plot(X_arr1[:, 0], X_arr1[:, 1], "r")

# AdaGrad
X_clone = X.clone().detach().requires_grad_(True)
X_arr2 = gradient_descent(X_clone, torch.optim.Adagrad([X_clone], lr=lr),
n_iters=n_iters)
plt.plot(X_arr2[:, 0], X_arr2[:, 1], "b")

# AdaGrad 手动实现
X_clone = X.clone().detach().requires_grad_(True)
X_arr1 = adagrad(X_clone, lr=lr, n_iters=n_iters)
plt.plot(X_arr1[:, 0], X_arr1[:, 1], c="orange", linestyle="--",
linewidth=3)

# 绘制等高线图
x1_grid, x2_grid = np.meshgrid(np.linspace(-7, 7, 100), np.linspace(-2, 2,
100))
y_grid = w.detach().numpy()[0, 0] * x1_grid**2 + w.detach().numpy()[1, 0]
* x2_grid**2
plt.contour(x1_grid, x2_grid, y_grid, levels=30, colors="gray")
plt.legend(["SGD", "AdaGrad", "Manual AdaGrad"])
plt.show()
```

#### 7.4.4 RMSProp

RMSProp (Root Mean Square Propagation, 均方根传播) 是在 AdaGrad 基础上的改进，它并非将过去所有梯度一视同仁的相加，而是逐渐遗忘过去的梯度，采用指数移动加权平均，呈指数地减小过去梯度的尺度。

$$H \leftarrow \alpha H + (1 - \alpha) \nabla^2$$

$$W \leftarrow W - \eta \frac{1}{\sqrt{H}} \nabla$$

- $H$ : 历史梯度平方和的指数移动加权平均
- $\alpha$ : 权重

可以通过 `torch.optim.RMSprop()` 并设置 `alpha` 权重参数来使用 RMSprop。

同样以寻找  $f(x_1, x_2) = 0.05x_1^2 + x_2^2$  的最小值为例：

```
import torch
import numpy as np
import matplotlib.pyplot as plt


def rmspropp(X, lr, alpha, n_iters):
    """rmspropp 手动实现"""
    X_arr = X.detach().numpy().copy() # 拷贝，用于记录优化过程
    H = torch.zeros_like(X) # 存放历史梯度信息
    for epoch in range(n_iters):
        grad = 2 * X * w.T # 当前梯度
        grad.squeeze_() # 降维
        H = alpha * H + (1 - alpha) * grad**2 # 历史梯度平方和的指数加权平均
        X.data -= lr / (torch.sqrt(H) + 1e-8) * grad # 更新参数，这里 H 后面加 1e-8 防止分母为 0
        X_arr = np.vstack([X_arr, X.detach().numpy()]) # 记录优化过程
    return X_arr


def gradient_descent(X, optimizer, n_iters):
    X_arr = X.detach().numpy().copy() # 拷贝，用于记录优化过程
    for epoch in range(n_iters):
        y = X**2 @ w
        y.backward() # 反向传播
        optimizer.step() # 更新参数
        optimizer.zero_grad() # 清空梯度
        X_arr = np.vstack([X_arr, X.detach().numpy()]) # 记录优化过程
    return X_arr


# 从(-7, 2)出发
X = torch.tensor([-7, 2], dtype=torch.float32, requires_grad=True)
w = torch.tensor([[0.05], [1.0]], requires_grad=True)
lr = 1e-1 # 学习率
n_iters = 1000 # 迭代次数
```

```

# 普通梯度下降
X_clone = X.clone().detach().requires_grad_(True)
X_arr1 = gradient_descent(X_clone, torch.optim.SGD([X_clone], lr=lr),
n_iters=n_iters)
plt.plot(X_arr1[:, 0], X_arr1[:, 1], "r")

# RMSProp
X_clone = X.clone().detach().requires_grad_(True)
X_arr2 = gradient_descent(X_clone, torch.optim.RMSprop([X_clone], lr=lr,
alpha=0.99), n_iters=n_iters)
plt.plot(X_arr2[:, 0], X_arr2[:, 1], "b")

# RMSProp 手动实现
X_clone = X.clone().detach().requires_grad_(True)
X_arr1 = rmspropp(X_clone, lr=lr, alpha=0.99, n_iters=n_iters)
plt.plot(X_arr1[:, 0], X_arr1[:, 1], c="orange", linestyle="--",
linewidth=3)

# 绘制等高线图
x1_grid, x2_grid = np.meshgrid(np.linspace(-7, 7, 100), np.linspace(-2, 2,
100))
y_grid = w.detach().numpy()[0, 0] * x1_grid**2 + w.detach().numpy()[1, 0]
* x2_grid**2
plt.contour(x1_grid, x2_grid, y_grid, levels=30, colors="gray")
plt.legend(["SGD", "RMSProp", "Manual RMSProp"])
plt.show()

```

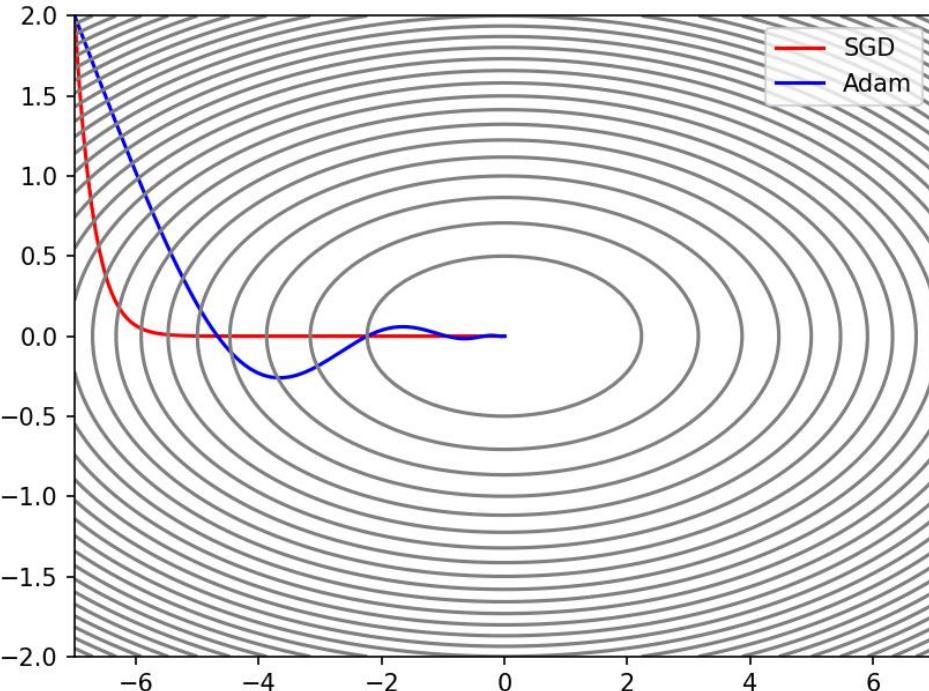
#### 7.4.5 Adam

Adam (Adaptive Moment Estimation, 自适应矩估计) 融合了 Momentum 和 AdaGrad 的方法。

$$\begin{aligned}
v &\leftarrow \alpha_1 v + (1 - \alpha_1) \nabla \\
h &\leftarrow \alpha_2 h + (1 - \alpha_2) \nabla^2 \\
\hat{v} &= \frac{v}{1 - \alpha_1^t} \\
\hat{h} &= \frac{h}{1 - \alpha_2^t} \\
W &\leftarrow W - \eta \frac{\hat{v}}{\sqrt{\hat{h}}}
\end{aligned}$$

- $\eta$ : 学习率
- $\alpha_1$ 、 $\alpha_2$ : 一次动量系数和二次动量系数

➤  $t$ : 迭代次数, 从 1 开始



可以通过 `torch.optim.Adam()` 并设置 `betas` 权重参数元组, 其中包含两个权重参数, 来使用 Adam。

```
import torch
import numpy as np
import matplotlib.pyplot as plt

def adam(X, lr, betas, n_iters):
    """Adam 手动实现"""
    X_arr = X.detach().numpy().copy() # 拷贝, 用于记录优化过程
    V = torch.zeros_like(X) # 存放历史梯度信息
    H = torch.zeros_like(X) # 存放历史梯度信息
    for epoch in range(n_iters):
        grad = 2 * X * w.T # 当前梯度
        grad.squeeze_() # 降维
        V = betas[0] * V + (1 - betas[0]) * grad # 历史梯度的指数加权平均
        H = betas[1] * H + (1 - betas[1]) * grad**2 # 历史梯度平方和的指数加权平均
        V_hat = V / (1 - betas[0]**(epoch + 1))
        H_hat = H / (1 - betas[1]**(epoch + 1))
```

```
X.data -= lr * V_hat / (torch.sqrt(H_hat) + 1e-8) # 更新参数，这里H后面加1e-8防止分母为0
X_arr = np.vstack([X_arr, X.detach().numpy()]) # 记录优化过程
return X_arr

def gradient_descent(X, optimizer, n_iters):
    X_arr = X.detach().numpy().copy() # 拷贝，用于记录优化过程
    for epoch in range(n_iters):
        y = X**2 @ w
        y.backward() # 反向传播
        optimizer.step() # 更新参数
        optimizer.zero_grad() # 清空梯度
        X_arr = np.vstack([X_arr, X.detach().numpy()]) # 记录优化过程
    return X_arr

# 从(-7, 2)出发
X = torch.tensor([-7, 2], dtype=torch.float32, requires_grad=True)
w = torch.tensor([[0.05], [1.0]], requires_grad=True)
lr = 1e-1 # 学习率
n_iters = 1000 # 迭代次数

# 普通梯度下降
X_clone = X.clone().detach().requires_grad_(True)
X_arr1 = gradient_descent(X_clone, torch.optim.SGD([X_clone], lr=lr),
n_iters=n_iters)
plt.plot(X_arr1[:, 0], X_arr1[:, 1], "r")

# Adam
X_clone = X.clone().detach().requires_grad_(True)
X_arr2 = gradient_descent(X_clone, torch.optim.Adam([X_clone], lr=lr,
betas=(0.9, 0.999)), n_iters=n_iters)
plt.plot(X_arr2[:, 0], X_arr2[:, 1], "b")

# Adam 手动实现
X_clone = X.clone().detach().requires_grad_(True)
X_arr1 = adam(X_clone, lr=lr, betas=(0.9, 0.999), n_iters=n_iters)
plt.plot(X_arr1[:, 0], X_arr1[:, 1], c="orange", linestyle="--",
linewidth=3, alpha=0.7)

# 绘制等高线图
x1_grid, x2_grid = np.meshgrid(np.linspace(-7, 7, 100), np.linspace(-2, 2,
100))
```

```
y_grid = w.detach().numpy()[0, 0] * x1_grid**2 + w.detach().numpy()[1, 0]
* x2_grid**2
plt.contour(x1_grid, x2_grid, y_grid, levels=30, colors="gray")
plt.legend(["SGD", "Adam", "Manual Adam"])
plt.show()
```

## 7.5 参数初始化和正则化

### 7.5.1 常数初始化

所有权重参数初始化为一个常数。

```
import torch.nn as nn

linear = nn.Linear(5, 2)

# 全部参数初始化为 0
nn.init.zeros_(linear.weight)
print(linear.weight)

# 全部参数初始化为 1
nn.init.ones_(linear.weight)
print(linear.weight)

# 全部参数初始化为一个常数
nn.init.constant_(linear.weight, 10)
print(linear.weight)
```

注意：将权重初始值设为 0 将无法正确进行学习。严格地说，不能将权重初始值设成一样的值。因为这意味着反向传播时权重全部都会进行相同的更新，被更新为相同的值（对称的值）。这使得神经网络拥有许多不同的权重的意义丧失了。为了防止“权重均一化”（瓦解权重的对称结构），必须随机生成初始值。

### 7.5.2 秩初始化

权重参数初始化为单位矩阵。

```
import torch.nn as nn

linear = nn.Linear(5, 2)

# 参数初始化为单位矩阵
nn.init.eye_(linear.weight)
print(linear.weight)
```

### 7.5.3 正态分布初始化

权重参数按指定均值与标准差正态分布初始化。

```
import torch.nn as nn

linear = nn.Linear(5, 2)

# 参数初始化为按指定均值与标准差正态分布
nn.init.normal_(linear.weight, mean=0.0, std=1.0)
print(linear.weight)
```

#### 7.5.4 均匀分布初始化

权重参数在指定区间内均匀分布初始化。

```
import torch.nn as nn

linear = nn.Linear(5, 2)

# 参数初始化为在区间内均匀分布
nn.init.uniform_(linear.weight, a=0, b=10)
print(linear.weight)
```

#### 7.5.5 Xavier 初始化（也叫 Glorot 初始化）

Xavier 初始化根据输入和输出的神经元数量调整权重的初始范围，确保每一层的输出方差与输入方差相近。适用于 Sigmoid 和 Tanh 等激活函数，能有效缓解梯度消失或爆炸问题。

Xavier 正态分布初始化：均值为 0，标准差为  $\sqrt{\frac{2}{n_{in}+n_{out}}}$  的正态分布。

Xavier 均匀分布初始化：区间  $[-\sqrt{\frac{6}{n_{in}+n_{out}}}, \sqrt{\frac{6}{n_{in}+n_{out}}}]$  内均匀分布。

其中  $n_{in}$  表示输入数， $n_{out}$  表示输出数。

```
import torch.nn as nn

linear = nn.Linear(5, 2)

# Xavier 正态分布初始化
nn.init.xavier_normal_(linear.weight)
print(linear.weight)

# Xavier 均匀分布初始化
nn.init.xavier_uniform_(linear.weight)
print(linear.weight)
```

#### 7.5.6 He 初始化（也叫 Kaiming 初始化）

He 初始化根据输入的神经元数量调整权重的初始范围。主要适用于 ReLU 及其变体(如 Leaky ReLU) 激活函数。

He 正态分布初始化：均值为 0，标准差为  $\sqrt{\frac{2}{n_{in}}}$  的正态分布。

He 均匀分布初始化：区间  $[-\sqrt{\frac{6}{n_{in}}}, \sqrt{\frac{6}{n_{in}}}]$  内均匀分布。

其中  $n_{in}$  表示输入数。

```
import torch.nn as nn

linear = nn.Linear(5, 2)

# Kaiming 正态分布初始化
nn.init.kaiming_normal_(linear.weight)
print(linear.weight)

# Kaiming 均匀分布初始化
nn.init.kaiming_uniform_(linear.weight)
print(linear.weight)
```

## 7.5.7 Dropout 随机失活

Dropout (随机失活，暂退法) 是一种在学习的过程中随机关闭神经元的方法。

可以通过 `torch.nn.Dropout(p)` 来使用 Dropout，并通过参数 `p` 来设置失活概率。

```
import torch

dropout = torch.nn.Dropout(p=0.5)
x = torch.randint(1, 10, (10,), dtype=torch.float32)
print("Dropout 前: ", x)
print("Dropout 后: ", dropout(x))
```

## 7.6 应用案例：房价预测

先安装 pandas 和 scikit-learn 库：`pip install pandas scikit-learn`。

使 用 House Prices 数 据 集：

<https://www.kaggle.com/c/house-prices-advanced-regression-techniques>。

|                                      |                           |
|--------------------------------------|---------------------------|
| 1 SalePrice: 目标值, 房价                 | 41 HeatingQC: 供暖质量        |
| 2 MSSubClass: 住宅类型                   | 42 CentralAir: 中央空调       |
| 3 MSZoning: 住宅分区分类                   | 43 Electrical: 电气系统类型     |
| 4 LotFrontage: 临街长度                  | 44 1stFlrSF: 一楼面积         |
| 5 LotArea: 地块面积                      | 45 2ndFlrSF: 二楼面积         |
| 6 Street: 街道路面类型                     | 46 LowQualFinSF: 低质量装修面积  |
| 7 Alley: 小巷路面类型                      | 47 GrLivArea: 地上居住面积      |
| 8 LotShape: 地块形状                     | 48 BsmtFullBath: 地下室全卫数量  |
| 9 LandContour: 地块平整度                 | 49 BsmtHalfBath: 地地下室半卫数量 |
| 10 Utilities: 公共设施情况                 | 50 FullBath: 地上全卫数量       |
| 11 LotConfig: 地块配置                   | 51 HalfBath: 地上半卫数量       |
| 12 LandSlope: 坡度                     | 52 Bedroom: 地上卧室数量        |
| 13 Neighborhood: 社区名称                | 53 Kitchen: 地上厨房数量        |
| 14 Condition1: 邻近主要道路或铁路的情况          | 54 KitchenQual: 厨房质量      |
| 15 Condition2: 邻近主要道路或铁路的情况 (如果存在多个) | 55 TotRmsAbvGrd: 地上总房间数   |
| 16 BldgType: 住宅类型                    | 56 Functional: 住宅功能评估     |
| 17 HouseStyle: 建筑风格                  | 57 Fireplaces: 壁炉数量       |
| 18 OverallQual: 整体材料/装修质量            | 58 FireplaceQu: 壁炉质量      |
| 19 OverallCond: 整体状况评分               | 59 GarageType: 车库类型       |
| 20 YearBuilt: 建筑年代                   | 60 GarageYrBlt: 车库建造年份    |
| 21 YearRemodAdd: 改造年代                | 61 GarageFinish: 车库内部装修   |
| 22 RoofStyle: 屋顶类型                   | 62 GarageCars: 车库容量       |
| 23 RoofMatl: 屋顶材料                    | 63 GarageArea: 车库面积       |
| 24 Exterior1st: 外部覆盖层材料              | 64 GarageQual: 车库质量       |
| 25 Exterior2nd: 外部覆盖层材料 (如果存在多个)     | 65 GarageCond: 车库现状       |
| 26 MasVnrType: 砌体饰面类型                | 66 PavedDrive: 车道铺装       |
| 27 MasVnrArea: 砌体饰面面积                | 67 WoodDeckSF: 木制甲板面积     |
| 28 ExterQual: 外部材料质量                 | 68 OpenPorchSF: 开放式门廊面积   |
| 29 ExterCond: 外部材料当前状况               | 69 EnclosedPorch: 封闭式门廊面积 |
| 30 Foundation: 地基类型                  | 70 3SsnPorch: 三季门廊面积      |
| 31 BsmtQual: 地下室高质量                  | 71 ScreenPorch: 纱窗门廊面积    |
| 32 BsmtCond: 地下室一般状况                 | 72 PoolArea: 游泳池面积        |
| 33 BsmtExposure: 地地下室采光等级            | 73 PoolQC: 游泳池质量          |
| 34 BsmtFinType1: 地地下室装修类型            | 74 Fence: 围栏质量            |
| 35 BsmtFinSF1: Type1装修面积             | 75 MiscFeature: 其他未分类设施   |
| 36 BsmtFinType2: 次要装修区域类型            | 76 MiscVal: 其他设施价值        |
| 37 BsmtFinSF2: Type2装修面积             | 77 MoSold: 销售月份           |
| 38 BsmtUnfSF: 未装修地下室面积               | 78 YrSold: 销售年份           |
| 39 TotalBsmtSF: 地地下室总面积              | 79 SaleType: 销售类型         |
| 40 Heating: 供暖类型                     | 80 SaleCondition: 销售条件    |

## 7.6.1 导入所需的模块

```
import torch
import pandas as pd
import torch.nn as nn
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.pipeline import Pipeline
from sklearn.impute import SimpleImputer
from sklearn.compose import ColumnTransformer
from sklearn.preprocessing import StandardScaler, OneHotEncoder
from torch.utils.data import TensorDataset, DataLoader
```

## 7.6.2 特征工程

对特征进行处理，数值型特征使用均值填充缺失值，再标准化；类别型特征使用字符串“NaN”填充缺失值，再独热编码。之后构造数据集：

```
def create_dataset():
    """构造数据集"""
    # 读取数据
    data = pd.read_csv("data/house_prices.csv")
    # 去除无关特征
    data.drop(["Id"], axis=1, inplace=True)
    # 划分特征和目标
    X = data.drop("SalePrice", axis=1)
    y = data["SalePrice"]
    # 筛选出数值型特征
    numerical_features = X.select_dtypes(exclude="object").columns
    # 筛选出类别型特征
    categorical_features = X.select_dtypes(include="object").columns
    # 划分训练集和测试集
    x_train, x_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=42)
    # 特征预处理
    # 数值型特征先用平均值填充缺失值，再进行标准化
    numerical_transformer = Pipeline(
        steps=[
            ("fillna", SimpleImputer(strategy="mean")),
            ("std", StandardScaler()),
        ]
    )
    # 类别型特征先将缺失值替换为字符串"NaN"，再进行独热编码
    categorical_transformer = Pipeline(
        steps=[
            ("fillna", SimpleImputer(strategy="constant",
fill_value="NaN")),
            ("onehot", OneHotEncoder(handle_unknown="ignore")),
        ]
    )
    # 组合特征预处理器
    preprocessor = ColumnTransformer(
        transformers=[
            ("num", numerical_transformer, numerical_features),
            ("cat", categorical_transformer, categorical_features),
        ]
    )
    # 进行特征预处理
```

```

    x_train = pd.DataFrame(preprocessor.fit_transform(x_train).toarray(),
columns=preprocessor.get_feature_names_out())
    x_test = pd.DataFrame(preprocessor.transform(x_test).toarray(),
columns=preprocessor.get_feature_names_out())
    # 构建数据集
    train_dataset = TensorDataset(torch.tensor(x_train.values).float(),
torch.tensor(y_train.values).float())
    test_dataset = TensorDataset(torch.tensor(x_test.values).float(),
torch.tensor(y_test.values).float())
    # 返回训练集, 测试集, 特征数量
    return train_dataset, test_dataset, x_train.shape[1]

train_dataset, test_dataset, feature_num = create_dataset()

```

### 7.6.3 搭建模型

```

# 搭建模型
model = nn.Sequential(
    nn.Linear(feature_num, 128),
    nn.BatchNorm1d(128),
    nn.ReLU(),
    nn.Dropout(0.2),
    nn.Linear(128, 1),
)

```

### 7.6.4 损失函数

关于房价的预测我们更加关心相对误差 $\frac{\hat{y}-y}{y}$ 而非绝对误差 $\hat{y} - y$ , 比如房价原本 20 万元而误差 10 万元, 那么误差可能难以接受; 但若房价原本 1000 万元而误差为 10 万元, 那误差可能并不算大。因此这里我们使用对数来衡量误差:

$$Loss = \sqrt{\frac{1}{n} \sum_{i=1}^n (\log \hat{y} - \log y)^2}$$

```

# 损失函数
def log_rmse(pred, target):
    mse = nn.MSELoss()
    pred.squeeze_()
    pred = torch.clamp(pred, 1, float("inf")) # 限制输出在 1 到正无穷之间
    return torch.sqrt(mse(torch.log(pred), torch.log(target)))

```

### 7.6.5 模型训练

```
# 模型训练
```

```
def train(model, train_dataset, test_dataset, lr, epoch_num, batch_size, device):
    def init_weight(layer):
        # 对线性层的权重进行初始化
        if type(layer) == nn.Linear:
            nn.init.xavier_normal_(layer.weight)

    model.apply(init_weight) # 初始化参数
    model = model.to(device) # 将模型加载到设备中
    optimizer = torch.optim.Adam(model.parameters(), lr=lr) # 优化器

    train_loss_list = [] # 记录训练损失
    test_loss_list = [] # 记录验证损失
    for epoch in range(epoch_num):

        # 训练过程
        model.train() # 将模型设置为训练模式
        train_loader = DataLoader(train_dataset, batch_size=batch_size, shuffle=True)
        train_loss_accumulate = 0
        # 训练模型
        for batch_count, (X, y) in enumerate(train_loader):
            # 前向传播
            X, y = X.to(device), y.to(device)
            output = model(X)
            # 反向传播
            loss_value = log_rmse(output, y)
            optimizer.zero_grad()
            loss_value.backward()
            optimizer.step()
            # 累加损失
            train_loss_accumulate += loss_value.item()
            # 打印进度条
            print(f"\repoch:{epoch:0>3}[{'='*int((batch_count+1) / len(train_loader)* 50 )}]:<50]", end="")
            this_train_loss = train_loss_accumulate / len(train_loader) # 计算平均损失
            train_loss_list.append(this_train_loss) # 记录训练损失

        # 验证过程
        model.eval() # 将模型设置为评估模式
        test_loader = DataLoader(dataset=test_dataset, batch_size=batch_size, shuffle=True)
        test_loss_accumulate = 0
```

```
with torch.no_grad(): # 关闭梯度计算
    for X, y in test_loader:
        # 前向传播
        X, y = X.to(device), y.to(device)
        output = model(X)
        # 累加损失
        test_loss_accumulate += loss_value.item()
    this_test_loss = test_loss_accumulate / len(test_loader) # 计算
平均损失
    test_loss_list.append(this_train_loss) # 记录验证损失

    # 打印训练损失，验证损失
    print(f" train_loss:{this_train_loss:.6f},
test_loss:{this_test_loss:.6f}")
return train_loss_list, test_loss_list

device = torch.device("cuda" if torch.cuda.is_available() else "cpu") # 如果
cuda 可用则使用 cuda，否则使用 cpu
train_loss_list, test_loss_list = train(model, train_dataset, test_dataset,
0.1, 200, 64, device)
plt.plot(train_loss_list, "r-", label="train_loss", linewidth=3) # 绘制
训练损失
plt.plot(test_loss_list, "k--", label="test_loss", linewidth=2) # 绘制验
证损失
plt.legend()
plt.show()
```

## 第 8 章 卷积神经网络

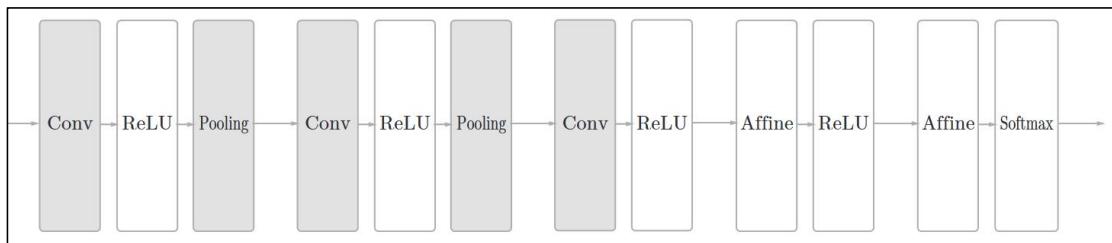
### 8.1 CNN 概述

卷积神经网络（Convolutional Neural Network, CNN）常被用于图像识别、语音识别等各种场合。它在计算机视觉领域表现尤为出色，广泛应用于图像分类、目标检测、图像分割等任务。

卷积神经网络的灵感来自于动物视觉皮层组织的神经连接方式，单个神经元只对有限区域内的刺激作出反应，不同神经元的感知区域相互重叠从而覆盖整个视野。

CNN 中新出现了卷积层（Convolution 层）和池化层（Pooling 层），下图是一个 CNN 的结构：

更多 Java -大数据 -前端 -python 人工智能资料下载，可百度访问：尚硅谷官网



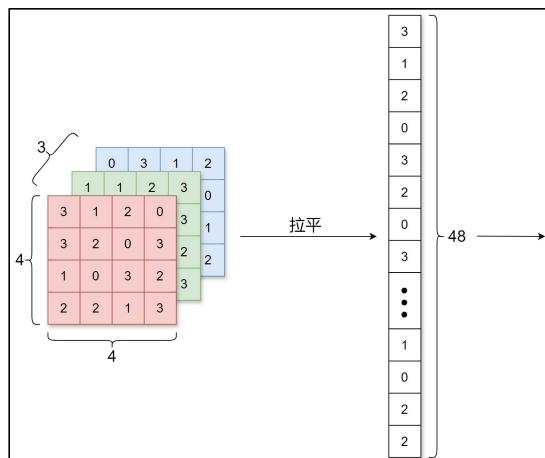
卷积层用于提取输入数据的局部特征。

池化层用于降维，增强鲁棒性并防止过拟合。

全连接层用于整合特征并输出结果。

## 8.2 卷积层

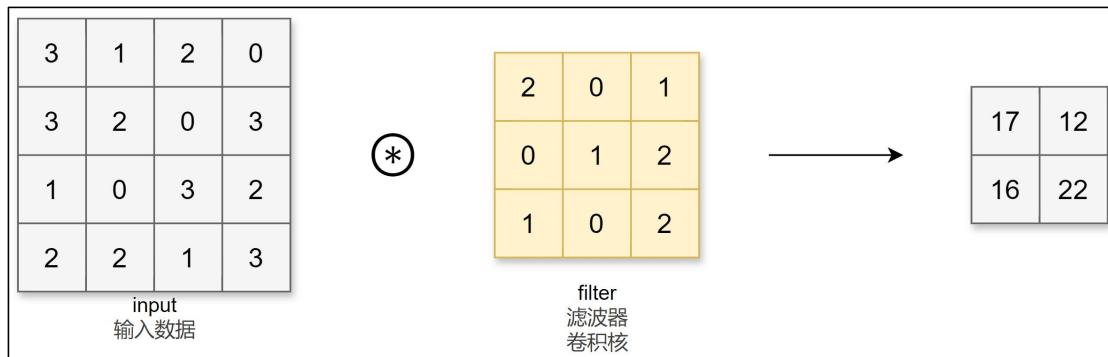
在全连接层中，相邻层的神经元全部连接在一起，输出的数量可以任意决定，但是却忽视了数据的形状。比如，输入数据为图像时，图像通常是长、宽、通道方向上的3维数据，但是向全连接层输入时却需要将其拉平为1维数据，这种情况下3维形状的数据中的空间信息可能被全连接层忽视掉。



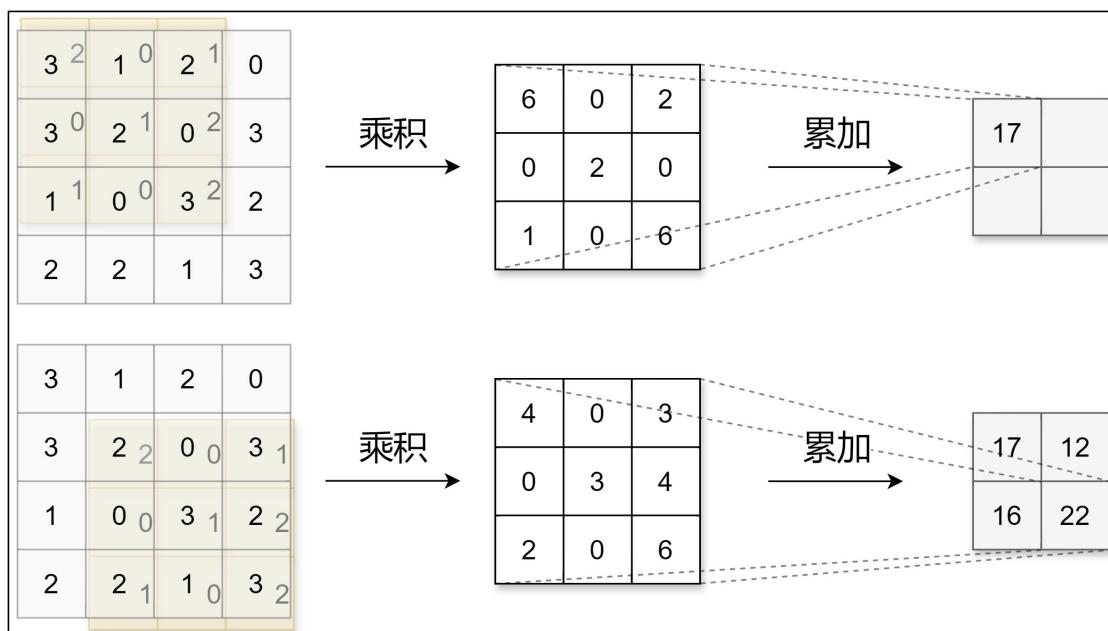
而卷积层可以保持数据形状不变，即接收3维形状的输入数据后同样以3维形状将数据输出至下一层。在CNN中，卷积层的输入输出数据也称为特征图（feature map）。

### 8.2.1 卷积运算

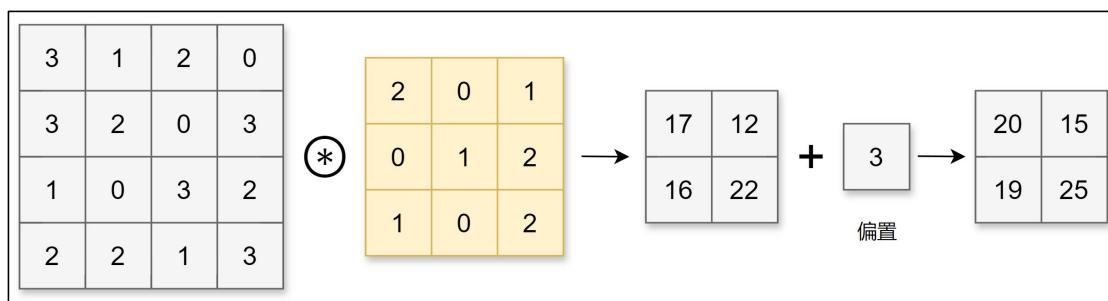
卷积层对数据进行卷积运算，卷积运算相当于图像处理中的滤波器运算。



对于输入数据，卷积运算以一定间隔滑动卷积核的窗口并应用。将各个位置上卷积核的元素和输入的对应元素相乘，然后再求和（也称为乘积累加运算、矩阵的内积）。

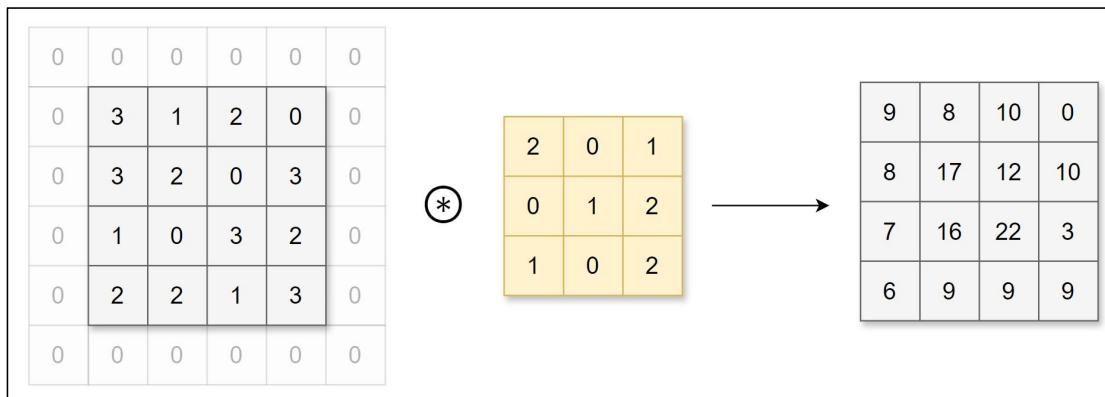


CNN 中，卷积核的参数对应之前的权重。并且 CNN 中也存在偏置。



## 8.2.2 填充

在进行卷积层的处理之前，有时要向输入数据的周围填入固定的数据（比如 0），这称为填充（padding）。例如，对形状为  $4 \times 4$  的数据进行幅度为 1 的填充，即用幅度为 1、值为 0 的数据填充周围：

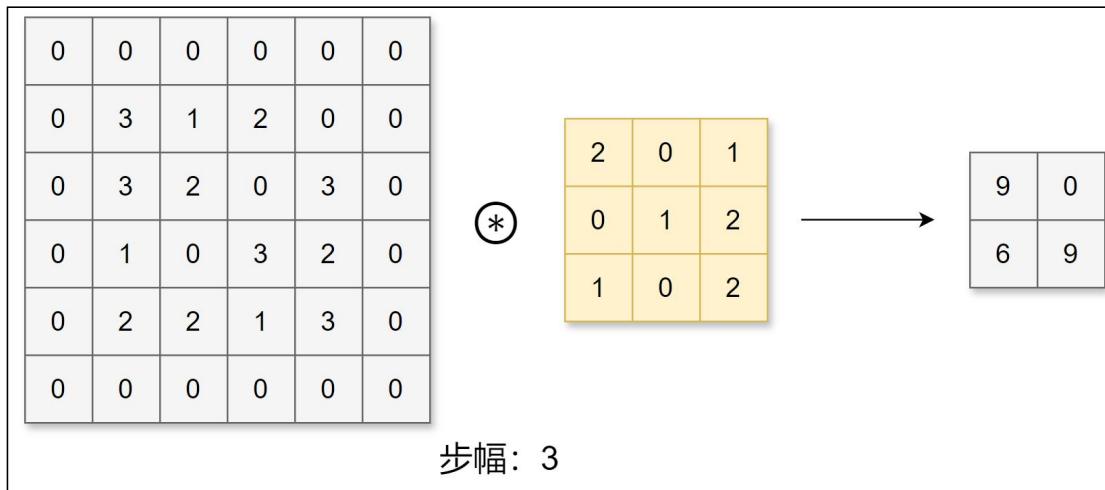


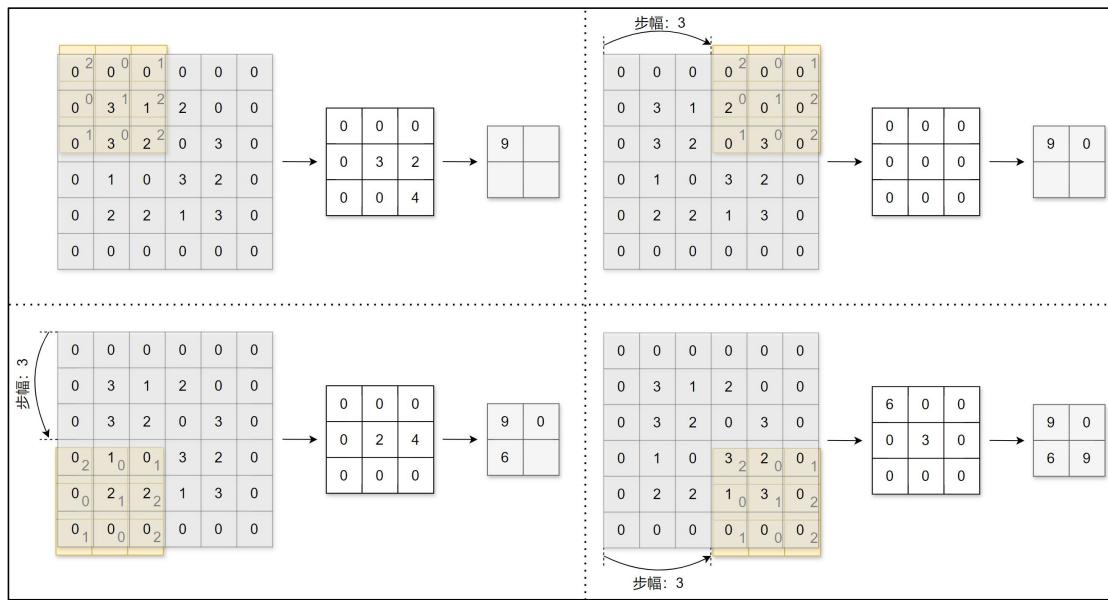
可以看到， $4 \times 4$  的数据进行幅度为 1 的填充后形状变为  $6 \times 6$ ，再经过卷积后数据的形状为  $4 \times 4$ 。使用填充主要目的是为了调整输出数据的形状大小，避免多次卷积后数据形状大小过小导致无法继续进行卷积运算。运用填充可以令数据形状在经过卷积运算后保持不变。

### 8.2.3 步幅

应用卷积核的位置间隔称之为步幅（stride）。

之前的例子中步幅都为 1，下面我们进行步幅为 3 的卷积运算：





可以看到填充和步幅都会影响输出数据的形状大小。增大填充，输出数据形状大小会变大；增大步幅，输出数据形状大小会变小。让我们分析一下给定填充和步幅如何计算输出数据的形状大小。

假设输入数据形状为 $(H, W)$ ，卷积核大小为 $(FH, FW)$ ，填充为 $P$ ，步幅为 $S$ ，输出数据形状为 $(OH, OW)$ ，可得：

$$OH = \frac{H + 2P - FH}{S} + 1$$

$$OW = \frac{W + 2P - FW}{S} + 1$$

例如，对于形状为(4,4)的输入数据应用幅度为 1 的填充，并应用步幅为 3，卷积核大小为(3,3)的卷积运算：

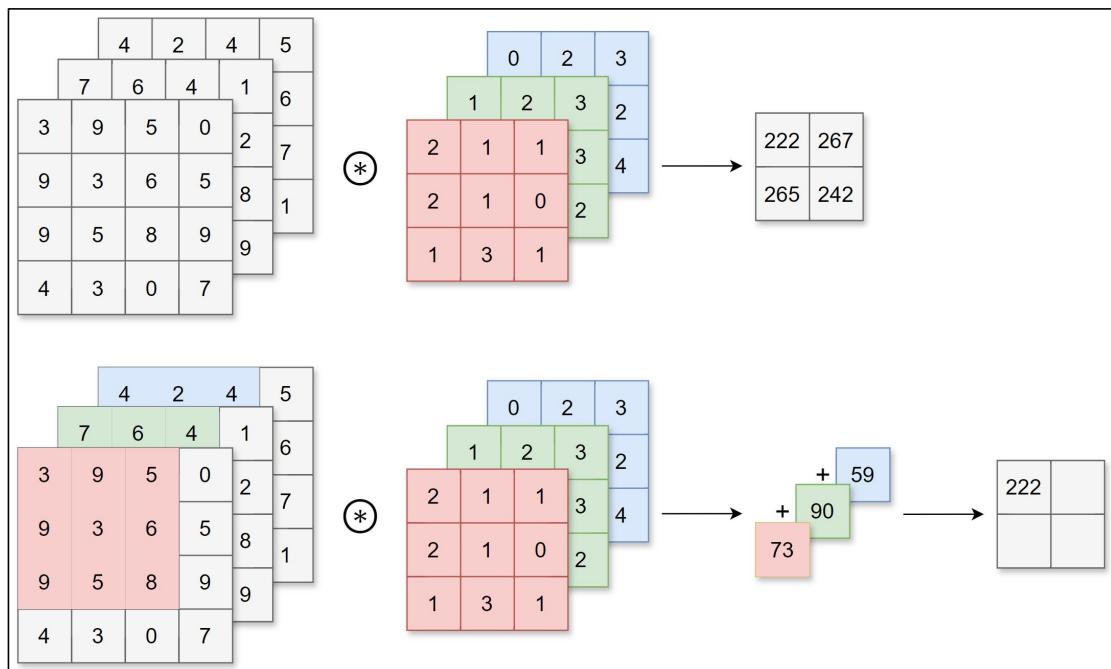
$$OH = OW = \frac{4 + 2 - 3}{3} + 1 = 2$$

得到形状为(2,2)的输出数据。

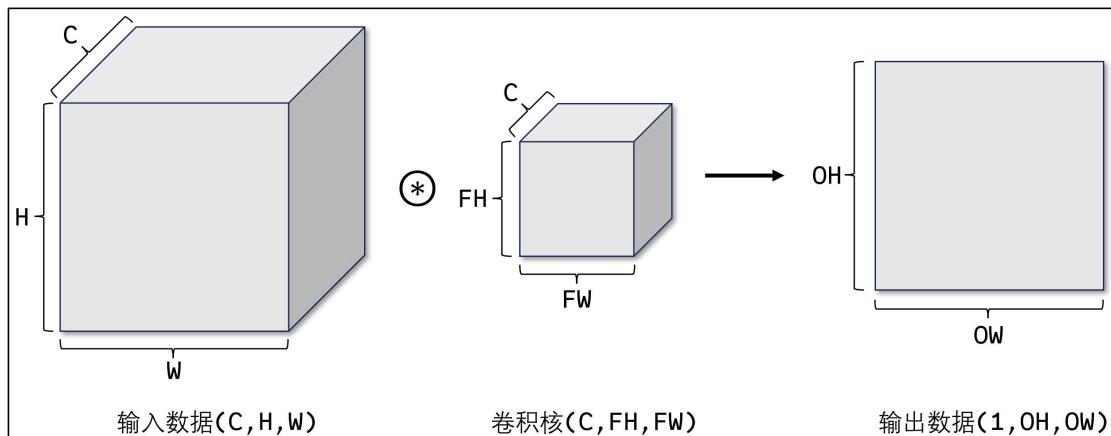
当输出大小无法除尽时，PyTorch 卷积层会自动向下取整，输出整数尺寸，舍弃无法覆盖完整卷积核的输入部分。

## 8.2.4 3 维数据的卷积运算

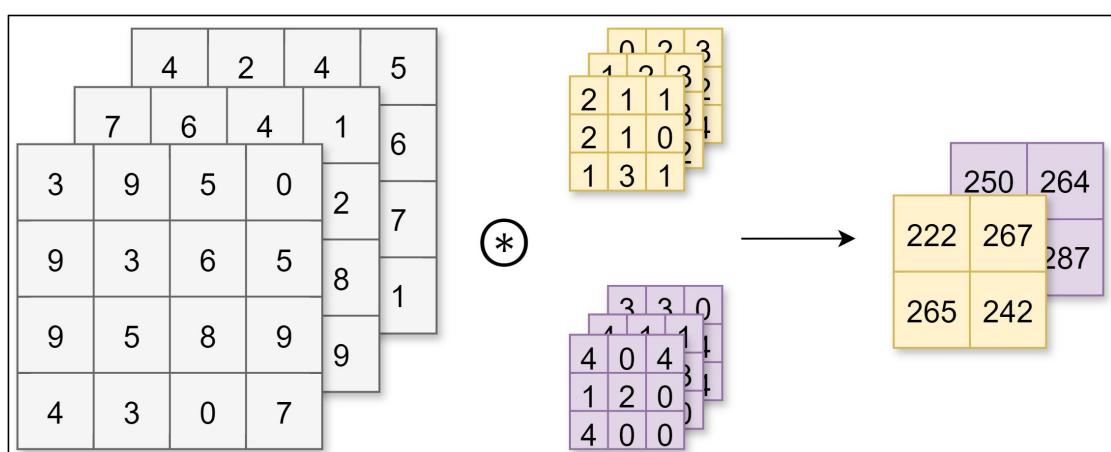
图像是 3 维数据，除了长、宽外还需要处理通道方向。在 3 维数据的卷积运算中，输入数据的通道数和卷积核的通道数须设为相同的值。当有多个通道时，会按通道进行输入数据和卷积核的卷积运算，并将结果相加得到输出数据。



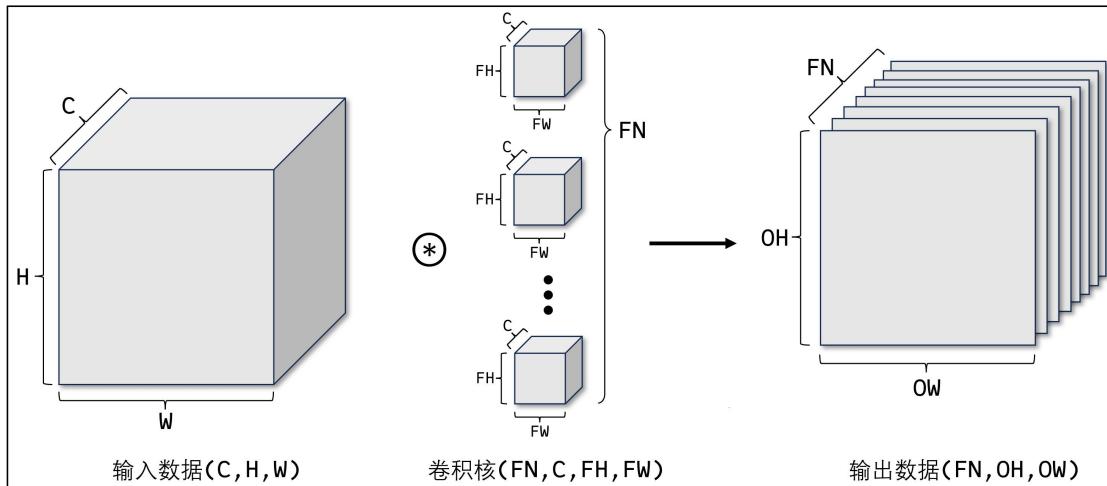
使用 1 个形状为 $(C, FH, FW)$ 的卷积核，对形状为 $(C, H, W)$ 的输入数据进行卷积运算，输出 1 张形状为 $(OH, OW)$ 特征图，即输出的通道数为 1。



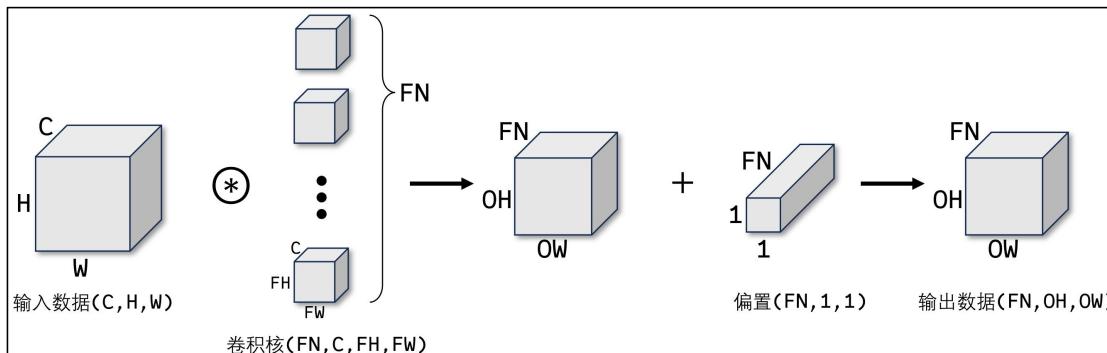
若想在通道方向获得多个卷积运算的输出，需要使用多个卷积核。



使用 $FN$ 个卷积核，输出特征图也变为 $FN$ 个：



若考虑偏置，则：



## 8.2.5 API 使用

```
torch.nn.Conv2d(in_channels, out_channels, kernel_size, stride, padding)
# inchannels:输入通道数
# outchannels:输出通道数
# kernel_size:卷积核大小
# stride:步幅
# padding:填充幅度
```

例：

```
import torch
import matplotlib.pyplot as plt

# 读取图片
img = plt.imread("data/duck.jpg")
print("图片数据形状: ", img.shape)

# 将图片数据转换为张量并改变形状
input = torch.tensor(img).permute(2, 0, 1).float()
print("输入特征图形状: ", input.shape)
```

```

# 初始化卷积核
conv = torch.nn.Conv2d(in_channels=3, out_channels=3, kernel_size=9,
stride=3, padding=0, bias=False)

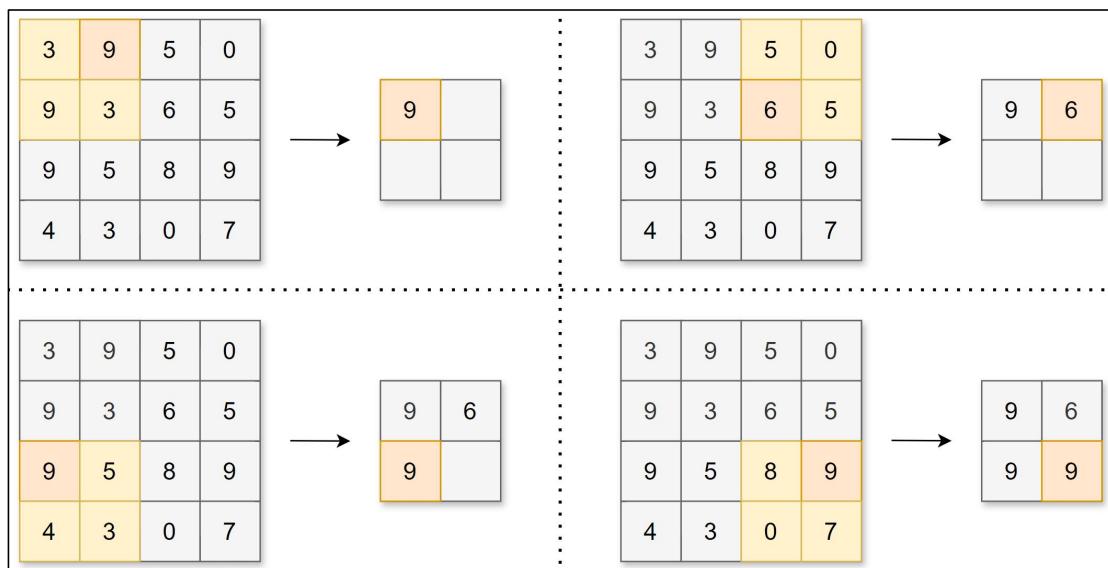
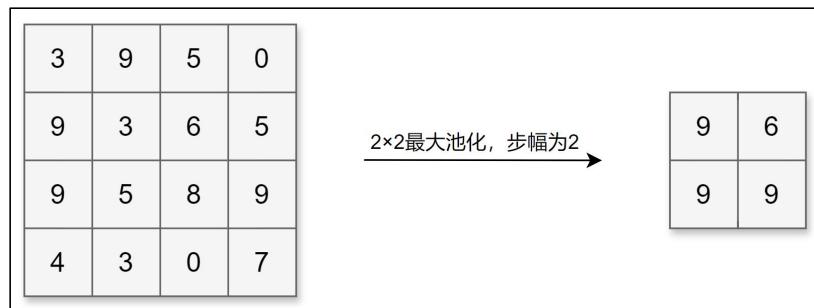
# 对输入特征图进行卷积操作
output = conv(input)
print("输出特征图形状: ", output.shape)

# 将输出特征图转换为图片
output = torch.clamp(output.int(), 0, 255) # 限制输出在 0 到 255 之间
output = output.permute(1, 2, 0).detach().numpy()
fig, ax = plt.subplots(1, 2, figsize=(10, 5))
ax[0].imshow(img)
ax[1].imshow(output)
plt.axis("off")
plt.show()

```

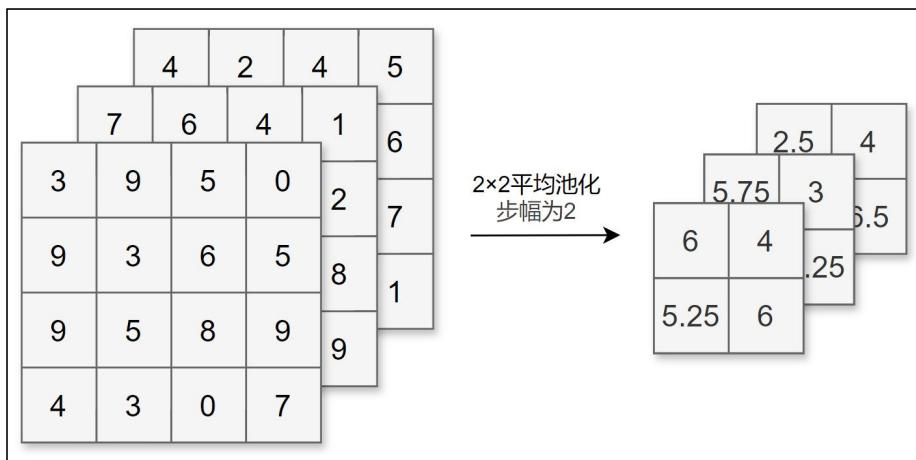
## 8.3 池化层

池化层缩小长、宽方向上的空间来进行降维，能够缩减模型的大小并提高计算速度。例如，对数据进行步幅为 2 的  $2 \times 2$  的 Max 池化：

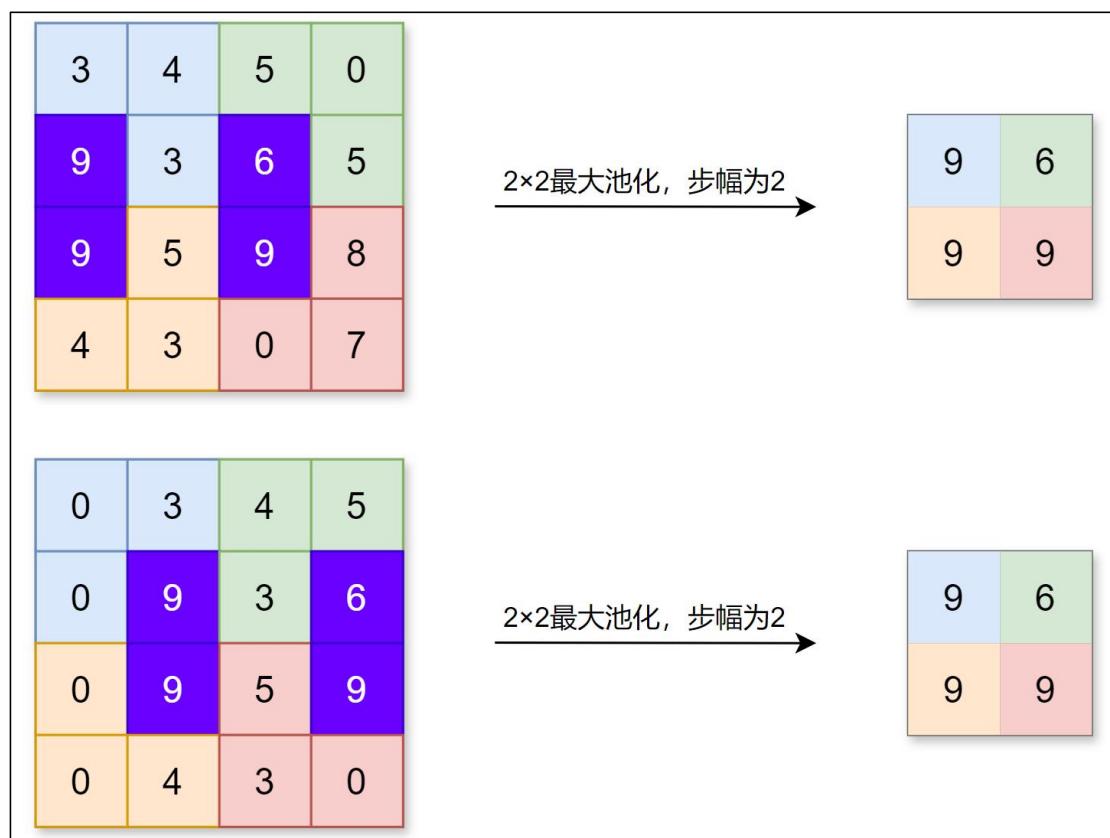


除了 Max 池化（计算窗口内的最大值），还有 Average 池化（平均池化，计算窗口内的平均值）等。一般会将池化的大小窗口和步幅设置为相同的值，比如  $2 \times 2$  的窗口大小，步幅会设置为 2。池化同样也可以设置填充。

和卷积层不同，池化层没有要学习的参数，并且池化运算按通道独立进行，经过池化运算后数据的通道数不会发生变化。



池化的另一个特点是对微小偏差具有鲁棒性，数据发生微小偏差时，池化可能会返回相同的结果。例如，数据在宽度方向上偏离 1 个元素，返回的结果不变。



API 使用:

```
# 最大池化
torch.nn.MaxPool2d(kernel_size, stride, padding)
# 平均池化
torch.nn.AvgPool2d(kernel_size, stride, padding)
```

例:

```
import torch
import matplotlib.pyplot as plt

# 读取图片
img = plt.imread("data/duck.jpg")
print("图片数据形状: ", img.shape)

# 将图片数据转换为张量并改变形状
input = torch.tensor(img).permute(2, 0, 1).float()
print("输入特征图形状: ", input.shape)

# 初始化卷积核
conv = torch.nn.Conv2d(in_channels=3, out_channels=3, kernel_size=9,
stride=3, padding=0, bias=False)

# 对输入特征图进行卷积操作
output1 = conv(input)
print("卷积后输出特征图形状: ", output1.shape)

# 初始化池化层
pool = torch.nn.MaxPool2d(kernel_size=6, stride=6, padding=1)

# 进行池化操作
output2 = pool(output1)
print("池化后输出特征图形状: ", output2.shape)

# 将输出特征图转换为图片
output1 = torch.clamp(output1.int(), 0, 255) # 限制输出在 0 到 255 之间
output1 = output1.permute(1, 2, 0).detach().numpy()
output2 = torch.clamp(output2.int(), 0, 255) # 限制输出在 0 到 255 之间
output2 = output2.permute(1, 2, 0).detach().numpy()

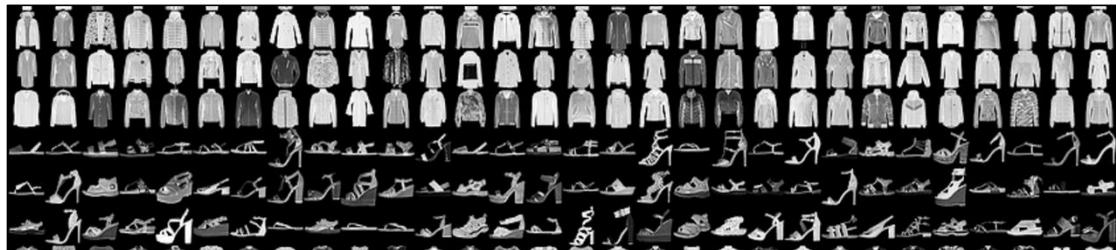
fig, ax = plt.subplots(1, 3, figsize=(15, 5))
ax[0].imshow(img)
ax[1].imshow(output1)
ax[2].imshow(output2)
plt.axis("off")
```

```
plt.show()
```

## 8.4 案例：服装分类

使 用 Fashion MNIST 数 据 集：

<https://www.kaggle.com/datasets/zalando-research/fashionmnist>。



数据集中每个样本都是  $28 \times 28$  的灰度图像，与来自 10 个类别的标签相关联。标签如下：

- 0: T 恤/上衣
- 1: 裤子
- 2: 套头衫
- 3: 连衣裙
- 4: 外套
- 5: 凉鞋
- 6: 衬衫
- 7: 运动鞋
- 8: 包
- 9: 靴子

### 8.4.1 加载数据

数据在 csv 文件中，每行数据为 1 个样本，第 1 列为标签，2 到 785 列为 784 个像素。

我们需要将其转换为  $28 \times 28$  的形状：

```
import torch
import pandas as pd
import torch.nn as nn
import matplotlib.pyplot as plt
from torch.utils.data import TensorDataset, DataLoader

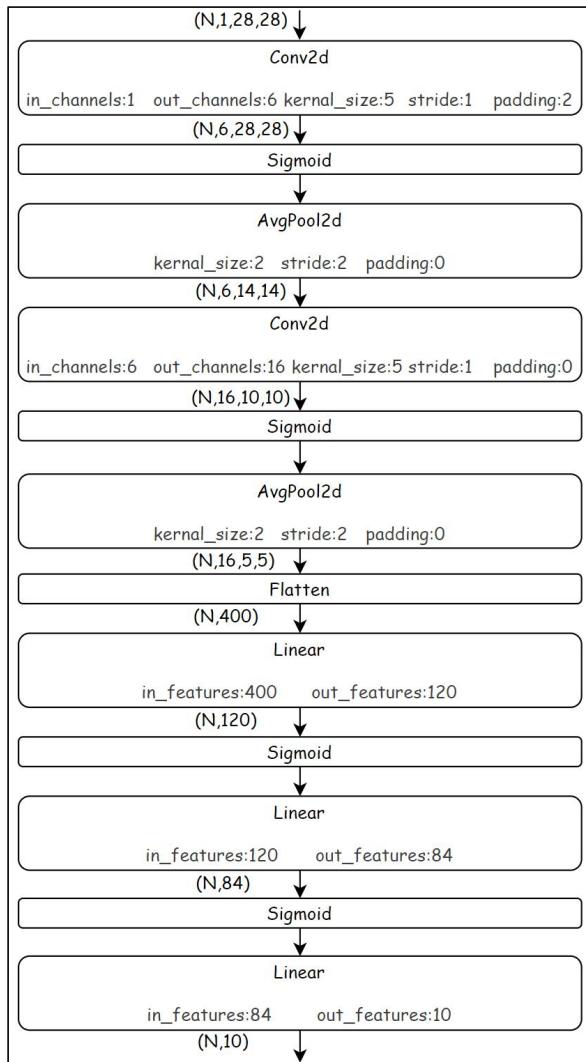
# 读取数据
fashion_mnist_train = pd.read_csv("data/fashion-mnist_train.csv")
```

更多 Java - 大数据 - 前端 - python 人工智能资料下载，可百度访问：[尚硅谷官网](#)

```
fashion_mnist_test = pd.read_csv("data/fashion-mnist_test.csv")
# 将数据转换为张量，原数据形状为 n×1×784，转换为 n×1×28×28 的张量
X_train = torch.tensor(fashion_mnist_train.iloc[:, 1:].values,
dtype=torch.float32).reshape(-1, 1, 28, 28)
y_train = torch.tensor(fashion_mnist_train.iloc[:, 0].values,
dtype=torch.int64)
X_test = torch.tensor(fashion_mnist_test.iloc[:, 1:].values,
dtype=torch.float32).reshape(-1, 1, 28, 28)
y_test = torch.tensor(fashion_mnist_test.iloc[:, 0].values,
dtype=torch.int64)
plt.imshow(X_train[12345, 0, :, :], cmap="gray")
plt.show()
# 构建数据集
train_dataset = TensorDataset(X_train, y_train)
test_dataset = TensorDataset(X_test, y_test)
```

## 8.4.2 搭建模型

搭建如下结构的模型：



# 搭建模型

```

model = nn.Sequential(
    nn.Conv2d(1, 6, kernel_size=5, padding=2),
    nn.Sigmoid(),
    nn.AvgPool2d(kernel_size=2, stride=2),
    nn.Conv2d(6, 16, kernel_size=5),
    nn.Sigmoid(),
    nn.AvgPool2d(kernel_size=2, stride=2),
    nn.Flatten(), # 拉平
    nn.Linear(400, 120),
    nn.Sigmoid(),
    nn.Linear(120, 84),
    nn.Sigmoid(),
    nn.Linear(84, 10),
)
  
```

给模型一个输入，测试各层输出值形状是否符合预期：

```
# 查看各层输出数据的形状
```

```
X = torch.rand(size=(1, 1, 28, 28), dtype=torch.float32)
for layer in model:
    X = layer(X)
    print(f'{layer.__class__.__name__:<12}output shape: {X.shape}')
```

### 8.4.3 模型训练

先初始化线性层和卷积层的权重参数。使用交叉熵损失函数和 SGD 优化方法。每个 epoch 中在训练集上训练模型，并在验证集上验证模型的准确率。

```
# 模型训练
def train(model, train_dataset, test_dataset, lr, epoch_num, batch_size,
device):
    def init_weights(layer):
        # 对线性层和卷积层使用 Xavier 均匀分布初始化参数
        if type(layer) == nn.Linear or type(layer) == nn.Conv2d:
            nn.init.xavier_uniform_(layer.weight)

    model.apply(init_weights) # 初始化参数
    model.to(device) # 将模型加载到设备
    loss = nn.CrossEntropyLoss() # 损失函数
    optimizer = torch.optim.SGD(model.parameters(), lr=lr) # 优化器
    for epoch in range(epoch_num):

        # 训练过程
        model.train() # 将模型设置为训练模式
        train_loader = DataLoader(dataset=train_dataset,
batch_size=batch_size, shuffle=True)
        loss_accumulate = 0
        train_correct_accumulate = 0
        for batch_count, (X, y) in enumerate(train_loader):
            # 前向传播
            X, y = X.to(device), y.to(device)
            output = model(X)
            # 反向传播
            loss_value = loss(output, y)
            optimizer.zero_grad()
            loss_value.backward()
            optimizer.step()
            # 累加损失
            loss_accumulate += loss_value.item()
            # 累加正确输出的数量
            _, pred = output.max(1)
            train_correct_accumulate += pred.eq(y).sum()
```

```
# 打印进度条
print(f"\repoch:{epoch:0>2}{'='*(int((batch_count+1) / len(train_loader) * 50))}<50]", end="")
    this_loss = loss_accumulate / len(train_loader) # 计算平均损失
    this_train_correct = train_correct_accumulate / len(train_dataset) # 计算训练准确率

# 验证过程
model.eval() # 将模型设置为评估模式
test_loader = DataLoader(dataset=test_dataset,
batch_size=batch_size, shuffle=True)
    test_correct_accumulate = 0
    with torch.no_grad(): # 关闭梯度计算
        for X, y in test_loader:
            # 前向传播
            X, y = X.to(device), y.to(device)
            output = model(X)
            # 累加正确输出的数量
            _, pred = output.max(1)
            test_correct_accumulate += pred.eq(y).sum()
    this_test_correct = test_correct_accumulate / len(test_dataset) # 计算验证准确率

# 打印损失，训练准确率，验证准确率
print(f" loss:{this_loss:.6f}, train_acc:{this_train_correct:.6f}, test_acc:{this_test_correct:.6f}")

device = torch.device("cuda" if torch.cuda.is_available() else "cpu") # 如果 cuda 可用则使用 cuda，否则使用 cpu
train(model, train_dataset, test_dataset, lr=0.9, epoch_num=20,
batch_size=256, device=device)
```

## 第 9 章 循环神经网络

### 9.1 自然语言处理概述

我们平日使用的语言，如汉语或英语，称为自然语言。自然语言处理（Natural Language Processing, NLP）的目标就是让计算机理解人类语言，进而完成对我们有帮助的事情。

说到计算机可以理解的语言，我们可能会想到编程语言或者标记语言等。这些语言的语法定义可以唯一性地解释代码含义，计算机也能根据确定的规则分析代码。编程语言是一种机械的、缺乏活力的语言。换句话说，它是一种“硬语言”。而汉语或英语等自然语言是“软

语言”，其含义和形式会灵活变化，并且会不断出现新的词语或新的含义。要让计算机去理解自然语言，使用常规方法是无法办到的。

### 9.1.1 基于同义词词典的方法

具有相同（同义词）或类似（近义词）含义的单词，可以归到同一个类别中；而根据单词“整体-部分”或者“上位-下位”关系，可以构建出层级的树状图。

这样，就可以构成一个庞大的“单词网络”，用它就可以教会计算机单词之间的关系，从而计算出单词的“相似度”。

主要缺点：

- 需要人工逐个定义单词之间的相关性，非常费时费力；
- 新词不断出现，语言不断变化，词典维护成本极高；
- 在表现力上也有限制。

### 9.1.2 基于计数的方法

大量的文本数据，构成了语料库（corpus）。

我们的目的，就是从语料库中，自动且高效地提取出语言的“本质”。最简单的做法，就是统计“词频”。

- 分词：对词进行统计，首先需要对文本内容进行切分，找出一个个基本单元；
- 词关联 ID：给单词标上一个 ID，构建单词和 ID 的关联字典（称为“词表”）；
- 词向量化：用一个固定长度的向量来表示单词，也称为词的“分布式表示”。

对每一个词，可以统计它周围出现了什么单词、出现了多少次（称为“上下文”）；把这些词频统计出来，就构成了一个向量；这个向量就可以表示当前的词了，称为“**词向量**”（word vector）。这样，所有词对应的向量，汇总起来就是一个矩阵，被称为 共现矩阵（co-occurrence matrix）。

主要缺点：

- 对所有词进行向量化表示的计算复杂度极高。

### 9.1.3 基于推理的方法

除了基于计数的方法，还可以使用推理的方法把词用向量表示出来。

我们希望在已知上下文的前提下，“推测”当前位置的词是什么。

利用神经网络，接收上下文信息作为输入，通过模型计算，输出各个单词可能得出现概率；从而就可以根据上下文，预测该出现的单词了。

## 9.2 词嵌入层

### 9.2.1 什么是词嵌入

自然语言是由文字构成的，而语言的含义是由单词构成的。即单词是含义的最小单位。因此为了让计算机理解自然语言，首先要让它理解单词含义。

词向量是用于表示单词意义的向量，也可以看作词的特征向量。将词映射到向量的技术称为 **词嵌入**（Word Embedding）。

|   |   |   |
|---|---|---|
| 你 | → | [0.3307, 0.4814, 1.4244, -0.3161, 0.0859]   |
| 我 | → | [0.9491, -0.1030, 2.5101, 0.1955, -0.0147]  |
| 他 | → | [0.2709, -0.2052, -0.6052, -0.5227, 0.9971] |

还有一种使用向量表示单词意义的方式是独热向量，独热向量很容易构建，但它们通常不是一个好的选择。一个主要原因是独热向量不能准确表达不同词之间的相似度。比如使用余弦相似度  $\frac{\mathbf{x}^T \mathbf{y}}{\|\mathbf{x}\| \|\mathbf{y}\|}$  来表示两个词之间的相似程度，由于任意两个不同词的独热向量之间的余弦相似度为 0，所以独热向量不能编码词之间的相似性。另一个原因是随着词汇量的增大，独热向量表示的向量大小也会增大，在词汇量较大的情况下会消耗大量的存储资源与计算资源。

将词转换为词向量时：

- 首先需要对文本进行分词，再根据需要进行清洗和标准化。
- 构建词表（Vocabulary），每个词对应一个索引。
- 使用词嵌入矩阵将词索引转换为词向量。

### 9.2.2 API 使用

可使用 `torch.nn.Embedding` 来初始化词嵌入矩阵：

```
torch.nn.Embedding(num_embeddings, embedding_dim)  
# num_embeddings:词的数量  
# embedding_dim:词向量的维度
```

例：

先安装 `jieba` 库用于分词： `pip install jieba`。

```

import torch
import torch.nn as nn
import jieba

# 设置随机种子
torch.manual_seed(42)
text = "自然语言是由文字构成的，而语言的含义是由单词构成的。即单词是含义的最小单位。因此为了让计算机理解自然语言，首先要让它理解单词含义。"
# 自定义停用词和标点符号
stopwords = {"的", "是", "而", "由", "，", "。", "、"}
# 分词，过滤停用词和标点，去重，构建词表
words = [word for word in jieba.lcut(text) if word not in stopwords]
vocab = list(set(words)) # 词表
# 构建词到索引的映射
word2idx = dict()
for idx, word in enumerate(vocab):
    word2idx[word] = idx
# 初始化嵌入层
embed = nn.Embedding(num_embeddings=len(word2idx), embedding_dim=5)
# 打印词向量
for idx, word in enumerate(vocab):
    word_vec = embed(torch.tensor(idx)) # 通过索引获取词向量
    print(f"{idx:>2}:{word:>8}\t{word_vec.detach().numpy()}")

```

|         |   |
|---------|---|
| 0:最小    | [ 1.9269153 1.4872841 0.9007172 -2.105521 0.6784184]        |
| 1:理解    | [ -1.2345449 -0.04306748 -1.604667 -0.7521353 1.648723 ]    |
| 2:单词    | [ -0.39247864 -1.4036071 -0.7278813 -0.5594302 -0.7688389 ] |
| 3:构成    | [ 0.7624454 1.6423169 -0.15959747 -0.49739754 0.43958926 ]  |
| 4:单位    | [ -0.75813115 1.0783176 0.80080056 1.6806206 1.2791244 ]    |
| 5:它     | [ 1.2964228 0.6104665 1.3347378 -0.23162432 0.04175949 ]    |
| 6:含义    | [ -0.2515753 0.8598585 -1.3846737 -0.87123615 -0.22336592 ] |
| 7:为了    | [ 1.7173615 0.31888032 -0.42451897 0.30572093 -0.7745925 ]  |
| 8:语言    | [ -1.5575725 0.9956361 -0.87978584 -0.60114205 -1.2741512 ] |
| 9:因此    | [ 2.122785 -1.2346531 -0.48791388 -0.913823 -0.65813726 ]   |
| 10:让    | [ 0.07802387 0.52580875 -0.48799172 1.191369 -0.81400764 ]  |
| 11:要    | [ -0.7359928 -1.4032478 0.03600367 -0.06347727 0.6756149 ]  |
| 12:计算机  | [ -0.09780689 1.844594 -1.1845374 1.3835493 1.4451338 ]     |
| 13:即    | [ 0.85641253 2.2180758 0.5231655 0.34664667 0.5246226 ]     |
| 14:自然语言 | [ 1.1412016 0.0516436 0.72810954 -0.7106419 -0.6020683 ]    |
| 15:文字   | [ 0.9604488 0.40481427 -0.82776886 1.334703 0.48353928 ]    |
| 16:首先   | [ -0.19756168 1.2683114 1.2242628 0.09811721 1.7422527 ]    |

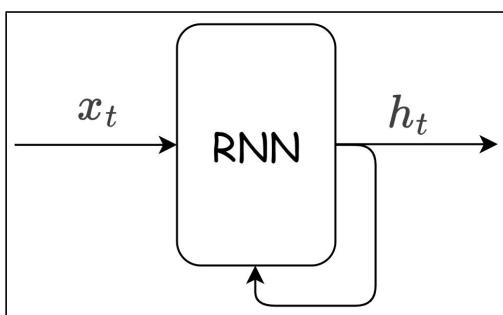
## 9.3 循环网络层

### 9.3.1 RNN 层介绍

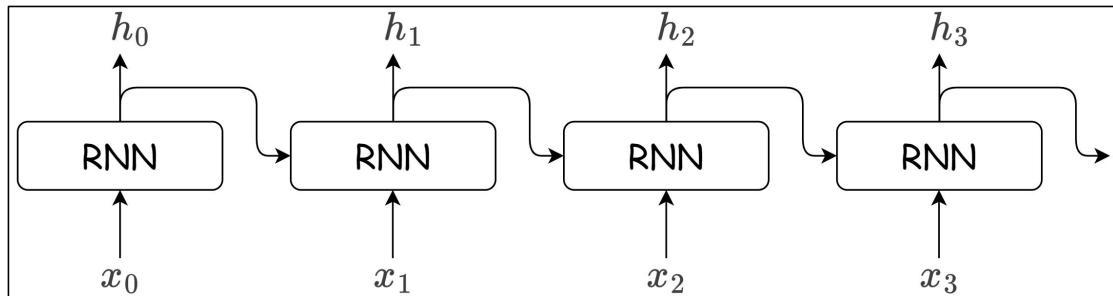
文本是连续的，具有序列特性。如果其序列被重排可能就会失去原有的意义。比如“狗咬人”这段文本具有序列关系，如果文字的序列颠倒可能就会表达不同的意思。

目前我们接触的神经网络都是前馈型神经网络。前馈（feedforward）是指网络的传播方向是单向的。具体地说，将输入信号传给下一层，下一层接收到信号后传给下下一层，然后再传给下下下一层...像这样，信号仅在一个方向上传播。虽然前馈网络结构简单、易于理解，并且可以应用于许多任务中。不过，这种网络存在一个大问题，就是不能很好地处理时间序列数据。更确切地说，单纯的前馈网络无法充分学习时序数据的性质。于是，循环神经网络（Recurrent Neural Network, RNN）应运而生。

RNN 层具有环路，通过环路数据可以在层内循环。向时序数据( $x_0, x_1 \dots x_t$ )输入层中，相应的会输出( $h_0, h_1 \dots h_t$ )。



RNN 层的循环的展开：



由图可知，各个时刻的 RNN 层接收传给该层的输入 $x_t$ 和前一个时刻 RNN 层的输出 $h_{t-1}$ ，据此计算当前时刻 RNN 层的输出 $h_t$ ：

$$h_t = \tanh(h_{t-1}W_h + x_tW_x + b)$$

RNN 层有 2 个权重，分别是与输入 $x_t$ 运算的权重 $W_x$ ，和与前一时刻 RNN 层的输出 $h_{t-1}$ （也叫隐藏状态，隐状态）运算的权重 $W_h$ 。执行完乘积和求和运算之后使用 tanh 函数转换，其结果就是时刻 $t$ 的输出 $h_t$ 。

### 9.3.2 API 使用

可使用 torch.nn.RNN 来初始化 RNN 层：

```
rnn = torch.nn.RNN(input_size, hidden_size, num_layers)
```

```
# input_size:输入数据的特征数量
# hidden_size:隐藏状态的特征数量
# num_layers:隐藏层的层数，如果设置多个层，前一个隐藏层的输出作为下一个隐藏层的输入
```

调用时需要传入 2 个参数：

```
output, hn = rnn(input, hx)
# input:输入数据[seq_len 序列长度, batch_size 批量大小, input_size]
# hx:初始隐状态[num_layers, batch_size, hidden_size]
# output:输出数据[seq_len, batch_size, hidden_size]
# ht:隐状态[num_layers, batch_size, hidden_size]
```

例：

```
import torch

rnn = torch.nn.RNN(input_size=8, hidden_size=16, num_layers=2)
input = torch.rand(1, 3, 8)
hx = torch.randn(2, 3, 16)
output, hn = rnn(input=input, hx=hx)
print(output.shape) # torch.Size([1, 3, 16])
print(hn.shape) # torch.Size([2, 3, 16])
```

## 9.4 案例：古诗生成

### 9.4.1 数据预处理

```
1 兰叶春葳蕤，桂华秋皎洁。欣欣此生意，自尔为佳节。谁知林栖者，闻风坐相悦。草木有本心，何求美人折？
2 江南有丹橘，经冬犹绿林。岂伊地气暖，自有岁寒心。可以荐佳客，奈何阻重深。运命唯所遇，循环不可寻。徒言树桃李，此木岂无阴。
3 蔡从碧山下，山月随人归。却顾所来径，苍苍横翠微。相携及田家，童稚开荆扉。绿竹入幽径，青萝拂行衣。欢言得所憩，美酒聊共挥。
4 花间一壶酒，独酌无相亲。举杯邀明月，对影成三人。月既不解饮，影徒随我身。暂伴月将影，行乐须及春。我歌月徘徊，我舞影零乱。
5 燕草如碧丝，秦桑低绿枝。当君怀归日，是妾断肠时。春风不相识，何事入罗帏？
6 岱宗夫如何，齐鲁青未了。造化钟神秀，阴阳割昏晓。荡胸生层云，决眦入归鸟。会当凌绝顶，一览众山小。
7 人生不相见，动如参与商。今夕复何夕，共此灯烛光。少壮能几时，鬓发各已苍。访旧半为鬼，惊呼热中肠。焉知二十载，重上君子堂。
8 绝代有佳人，幽居在空谷。自云良家女，零落依草木。关中昔丧乱，兄弟遭杀戮。官高何足论，不得收骨肉。世情恶衰歇，万事随转烛。
9 死别已吞声，生别常恻恻。江南瘴疠地，逐客无消息。故人入我梦，明我常相忆。君今在罗网，何以有羽翼？恐非平生魂，路远不可测。
10 浮云终日行，游子久不至。三夜频梦君，情亲见君意。告归常局促，苦道来不易。江湖多风波，舟楫恐失坠。出门搔白首，若负平生志。
11 圣代无隐士，英灵尽来归。遂令东山客，不得顾采薇。既至金门远，孰云吾道非。江淮度寒食，京洛缝春衣。置酒长安道，同心与我违。
12 下马饮君酒，问君何所之。君言不得意，归卧南山陲。但去莫复问，白云无尽时。
13 言入黄花川，每逐青溪水。随山将万转，趣途无百里。声喧乱石中，色静深松里。漾漾泛菱荇，澄澄映葭苇。我心素已闲，清川澹如此。
14 斜光照墟落，穷巷牛羊归。野老念牧童，倚杖候荆扉。雉雊麦苗秀，蚕眠桑叶稀。田夫荷锄至，相见语依依。即此羡闲逸，怅然吟式微。
15 颀色天下重，西施宁久微。朝为越溪女，暮作吴宫妃。贱日岂殊众，贵来方悟稀。邀人傅粉粉，不自著罗衣。君宠爱娇态，君怜无是非。
16 北山白云里，隐者自怡悦。相望始登高，心随雁飞灭。愁因薄暮起，兴是清秋发。时见归村人，沙行渡头歇。天边树若荠，江畔洲如月。
17 山光忽西落，池月渐东上。散发乘夕凉，开轩卧闲敞。荷风送香气，竹露滴清响。欲取鸣琴弹，恨无知音赏。感此怀故人，中宵劳梦想。
18 夕阳度西岭，群壑倏已暝。松月生夜凉，风泉满清响。樵人归尽欲，烟鸟栖初定。之子宿来，孤琴候萝径。
19 高卧南斋时，开帷月初吐。清辉淡水木，演漾在窗户。苒苒几盈虚，澄澄变今古。美人清江畔，是夜越吟苦。千里其如何，微风吹兰杜。
20 绝顶一茅茨，直上三十里。叩关无僮仆，窥室惟案几。若非巾柴车，应是钓秋水。差池不相见，黾勉空仰止。草色新雨中，松声晚窗里。
```

数据在.txt 文件中，每行为一首诗。

首先将每个字作为一个词构建词表。并将原文转换为索引序列。

```
import re
import torch
from torch import nn, optim
from torch.utils.data import Dataset, DataLoader
```

```

# 数据预处理
def process_poems(file_path):
    poems = [] # 保存处理后的诗
    char_set = set() # 保存所有不重复的字
    with open(file_path, "r", encoding="utf-8") as f:
        for line in f:
            # 逐行处理
            line = re.sub("[，。、？！：]", "", line).strip() # 去掉标点
    符号与两侧空白
            # 按字分割并去重
            char_set.update(list(line))
            # 按字保存诗
            poems.append(list(line))

    # 构建词表
    vocab = list(char_set) + ["<UNK>"]
    # 创建词到索引的映射
    word2idx = {word: idx for idx, word in enumerate(vocab)}

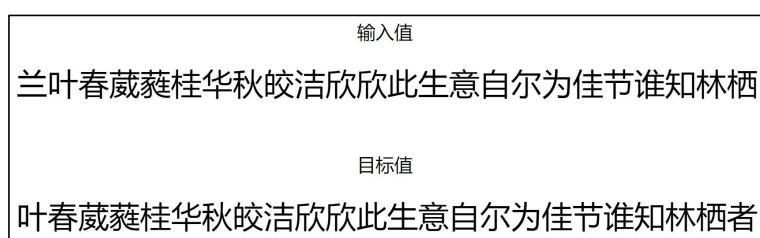
    # 将诗转换为索引序列
    sequences = []
    for poem in poems:
        seq = [word2idx.get(word) for word in poem]
        sequences.append(seq)
    return sequences, word2idx, vocab

sequences, word2idx, vocab = process_poems("data/poems.txt")

```

## 9.4.2 自定义 Dataset

构建训练模型的数据集，将一个序列作为输入，另一个序列作为目标。



```

# 自定义 Dataset
class PoetryDataset(Dataset):
    def __init__(self, sequences, seq_len):
        self.seq_len = seq_len
        self.data = []
        for seq in sequences:

```

```
        for i in range(0, len(seq) - self.seq_len):
            self.data.append((seq[i : i + self.seq_len], seq[i + 1 : i + 1 + self.seq_len]))

    def __len__(self):
        return len(self.data)

    def __getitem__(self, idx):
        x = torch.LongTensor(self.data[idx][0])
        y = torch.LongTensor(self.data[idx][1])
        return x, y

dataset = PoetryDataset(sequences, 24)
```

### 9.4.3 搭建模型

词嵌入层→RNN→全连接层。

```
# 搭建模型
class PoetryRNN(nn.Module):
    def __init__(self, vocab_size, embedding_dim=128, hidden_size=256,
num_layers=1):
        super().__init__()
        self.embed = nn.Embedding(num_embeddings=vocab_size,
embedding_dim=embedding_dim)
        self.rnn = nn.RNN(input_size=embedding_dim,
hidden_size=hidden_size, num_layers=num_layers)
        self.linear = nn.Linear(in_features=hidden_size,
out_features=vocab_size)

    def forward(self, input, hx=None):
        embed = self.embed(input)
        output, hidden = self.rnn(embed, hx)
        output = self.linear(output)
        return output, hidden

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
model = PoetryRNN(len(vocab), 256, 512, 2).to(device)
```

### 9.4.4 模型训练

使用交叉熵损失函数，Adam 优化方法。

```
# 模型训练
def train(model, dataset, lr, epoch_num, batch_size, device):
```

```

model.train() # 设置为训练模式
dataloader = DataLoader(dataset, batch_size=batch_size, shuffle=True)
loss = nn.CrossEntropyLoss() # 损失函数
optimizer = optim.Adam(model.parameters(), lr=lr) # 优化器
for epoch in range(epoch_num):
    loss_accumulate = 0 # 累加损失
    for batch_count, (x, y) in enumerate(dataloader):
        # 前向传播
        x, y = x.to(device), y.to(device)
        output, _ = model(x)
        # 反向传播
        loss_value = loss(output.transpose(1, 2), y)
        optimizer.zero_grad()
        loss_value.backward()
        optimizer.step()
        # 累加损失
        loss_accumulate += loss_value.item()
        print(f"\repoch:{epoch:0>2}{'='*int((batch_count+1) / len(dataloader) * 50)}:<50]", end="")
    print(f" loss:{loss_accumulate/len(dataloader):.6f}")

train(model=model, dataset=dataset, lr=1e-3, epoch_num=20, batch_size=32,
device=device)

```

## 9.4.5 测试

输入一个起始词让模型开始生成。

```

# 生成
def generate_poem(model, word2idx, vocab, start_token, line_num=4,
line_length=7):
    model.eval() # 设置为预测模式
    poem = [] # 记录生成结果
    current_line_length = line_length # 当前句的剩余长度
    start_token = word2idx.get(start_token, word2idx["<UNK>"]) # 起始
token
    # 如果起始 token 在词典中，添加到结果中
    if start_token != word2idx["<UNK>"]:
        poem.append(vocab[start_token])
        current_line_length -= 1
    input = torch.LongTensor([[start_token]]).to(device) # 输入
    hidden = None # 初始化隐状态
    with torch.no_grad(): # 关闭梯度计算
        for _ in range(line_num): # 生成的行数

```

```
for interpunction in [", ", ". \n"]: # 每行两句
    while current_line_length > 0: # 每句诗 line_length 个字
        output, hidden = model(input, hidden)
        prob = torch.softmax(output[0, 0], dim=-1) # 计算概率
        next_token = torch.multinomial(prob, 1) # 从概率分布中
随机采样
        poem.append(vocab[next_token.item()]) # 将采样结果添
加到结果中
        input = next_token.unsqueeze(0)
        current_line_length -= 1
        current_line_length = line_length
        poem.append(interpunction) # 每句结尾添加标点符号
    return "".join(poem) # 将列表转换为字符串

print(generate_poem(model, word2idx, vocab, start_token="—", line_num=4,
line_length=7))
```