# 课程介绍

- 拟合、线性可分、张量
- pytorch安装
- 张量运算
- 自动微分

# 1. 线性代数

在深度学习中，数据通常表示为向量或矩阵，线性代数提供了操作这些数据表示的工具。

线性代数是关于向量空间和线性映射的一个数学分支，包括对向量、线、面、空间的研究，是纯数学和应用数学的核心。

$$f(x+y) = f(x) + f(y)$$

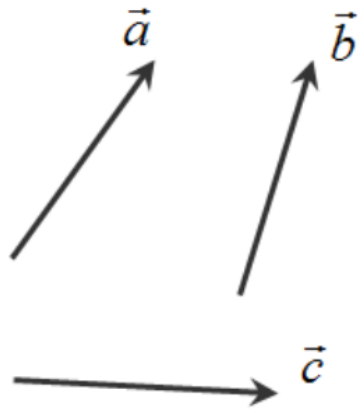
## 1.1 向量

在数学中，向量（也称为欧几里得向量、几何向量），指具有大小（magnitude）和方向的量。它可以形象化地表示为带箭头的线段。箭头所指：代表向量的方向；线段长度：代表向量的大小。与向量对应的量叫做数量（物理学中称标量），数量（或标量）只有大小，没有方向。

n个有序的数x1, x2...xn所组成的数组称为n维向量，这n个数称为该向量的分量，第i个数xi称为第i个分量。

$$\vec{u} = (x_1, x_2, \ldots, x_n)$$

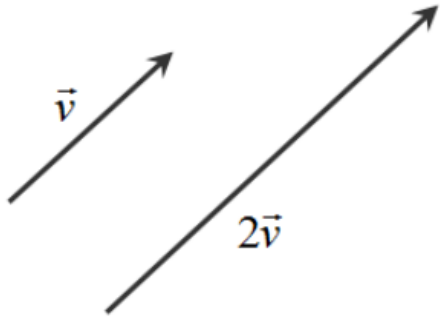
$$\|\vec{u}\| = \sqrt{\sum_{i=1}^{n} x_i^2}$$




n维向量可以写成一行，称为行向量；n维向量写成一列，称为列向量

$$\vec{v} = \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{pmatrix}$$

一个标量$k$和一个向量$\vec{v}$之间可以做乘法，得出的结果是另一个与$\vec{v}$方向相同或相反，大小是$\vec{v}$的大小的$|k|$倍的向量，可以记成$k\vec{v}$。-1乘以任意向量会得到它的反向量，0乘以任何向量都会得到零矢量$\vec{0}$。如图5所示，标量2与向量$\vec{v}$的数乘，得到$2\vec{v}$，方向与$\vec{v}$相同，大小是它的两倍。




向量的数量积(点乘)

数量积也叫点积，它是向量与向量的乘积，其结果为一个标量（非向量）。几何上，数量积可以定义如下：

设$\vec{a}$、$\vec{b}$为两个任意向量，它们的夹角为$\theta$，则他们的数量积为：

$$\vec{a} \cdot \vec{b} = |\vec{a}| \left|\vec{b}\right| \cos\theta$$

## 1.2 矩阵性质

(1)加法

- A+B=B+A
- A+O=A (其中O是元素全为0的同型矩阵)
- A+(-A)=O

(2)数乘

- k(mA)=(km)A=m(kA)
- (k+m)A=kA+mA
- k(A+B)=kA+kB
- 1A=A
- 0A=0

(3)乘法

- (AB)C=A(BC)
- A(B+C)=AB+AC
- (B+C)A=BA+CA（**注意顺序不可以颠倒**）

[矩阵相乘]
[https://baike.baidu.com/item/%E7%9F%A9%E9%98%B5%E4%B9%98%E6%B3%95/5446029]

设**A**为$m \times p$ 的矩阵，**B**为$p \times n$ 的矩阵，那么称$m \times n$ 的矩阵**C**为矩阵**A**与**B**的乘积，记作$C = AB$ ，其中矩阵C中的第$i$ 行第$j$ 列元素可以表示为： [1]

$$(AB)_{ij} = \sum_{k=1}^{p} a_{ik} b_{kj} = a_{i1} b_{1j} + a_{i2} b_{2j} + \cdots + a_{ip} b_{pj}$$

如下所示：

$$A = \begin{bmatrix} a_{1,1} & a_{1,2} & a_{1,3} \\ a_{2,1} & a_{2,2} & a_{2,3} \end{bmatrix}$$

$$B = \begin{bmatrix} b_{1,1} & b_{1,2} \\ b_{2,1} & b_{2,2} \\ b_{3,1} & b_{3,2} \end{bmatrix}$$

$$C = AB = \begin{bmatrix} a_{1,1}b_{1,1} + a_{1,2}b_{2,1} + a_{1,3}b_{3,1}, & a_{1,1}b_{1,2} + a_{1,2}b_{2,2} + a_{1,3}b_{3,2} \\ \\ a_{2,1}b_{1,1} + a_{2,2}b_{2,1} + a_{2,3}b_{3,1}, & a_{2,1}b_{1,2} + a_{2,2}b_{2,2} + a_{2,3}b_{3,2} \end{bmatrix}$$

(4)转置

- (A+B)T=AT+BT
- (kA)T=kAT
- (AB)T=BTAT
- (AT)T=A

# 2. 欠拟合、拟合、过拟合

拟合是指用数学模型或算法来近似数据中的关系，以便进行数据分析或预测。它的目标是找到一个模型，使其与数据最好地匹配，以便对数据进行有意义的分析和应用。

"拟合" 这个词语在数学和统计学中常用，它的来源主要是从拉丁文 "aptare"（适应）而来。在数学和统计学的上下文中，"拟合" 指的是通过某种数学模型、曲线或函数来逼近或适应实际观测到的数据点。

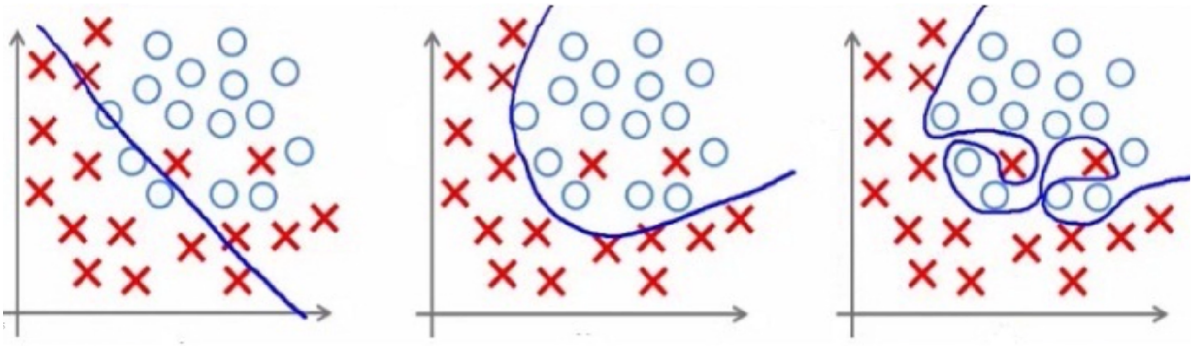
拟合的目标是找到一个模型，使得模型的输出与实际观测数据尽可能接近。这可以通过调整模型的参数来实现，通常涉及到损失函数或误差函数，以确保模型的预测与真实数据的差距最小。

在机器学习和深度学习中，拟合是模型训练的一个重要步骤。使其能够对新的、未见过的数据进行准确预测。

$$2x+y=4$$
$$4x+2y=8$$

$$2x+y=4$$
$$4x+3y=9$$

$$2x+y=4$$
$$4x+3y=9$$
$$5x+2y=7$$

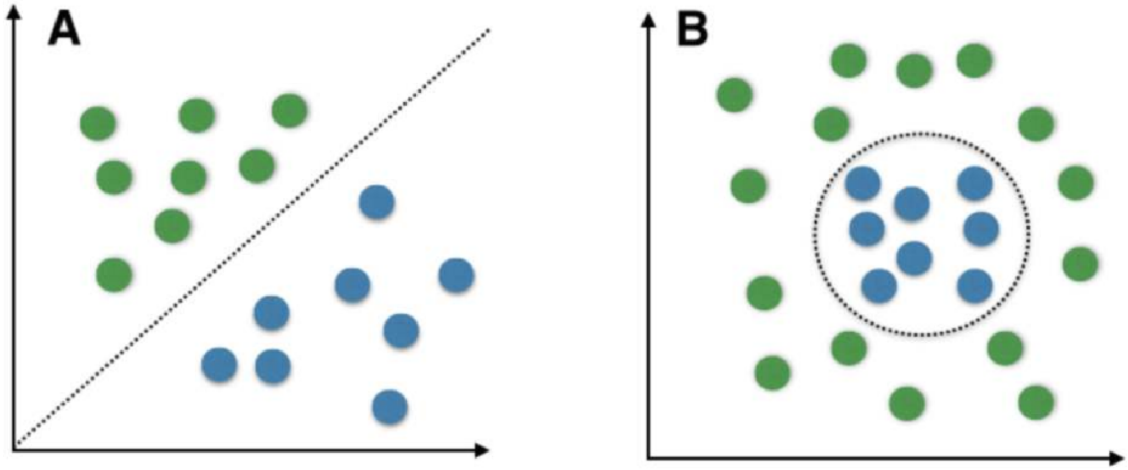


**欠拟合** **好的拟合** **过拟合**

1. **欠拟合（Underfitting）**：
   - **概念**：欠拟合指的是模型对数据的拟合程度不够好，它未能捕捉数据中的特征。
   - **表现**：在训练数据和测试数据上都表现得很差，预测性能低下。
   - **原因**：通常是因为模型太简单，不能捕捉数据的复杂的特征。
2. **拟合（Good Fit）**：
   - **概念**：拟合表示模型对数据的拟合程度适中，能够很好地捕捉数据中的特征。
   - **表现**：在训练数据和测试数据上表现良好，预测性能较好。
   - **原因**：模型的复杂性适中，可以很好地平衡捕捉数据的特征和避免过度拟合。
3. **过拟合（Overfitting）**：
   - **概念**：过拟合指的是模型对训练数据过于拟合，捕捉了数据中的噪声。
   - **表现**：在训练数据上表现得很好，但在测试数据上表现不佳，预测性能较差。
   - **原因**：通常是因为模型过于复杂，有太多的参数，以至于过分适应了训练数据的细节。

> 通俗地说，欠拟合就像模型太天真，无法理解数据；拟合就像模型恰到好处，能够理解数据中的主要特征；而过拟合就像模型太聪明了，过于执着于捕捉数据的细微差异。选择合适的模型复杂性，以及进行适当的训练和验证，是解决欠拟合和过拟合问题的关键。目标是获得一个“拟合得刚刚好”的模型，以实现最佳的预测性能。

# 3. 线性可分与不可分




**线性可分：**

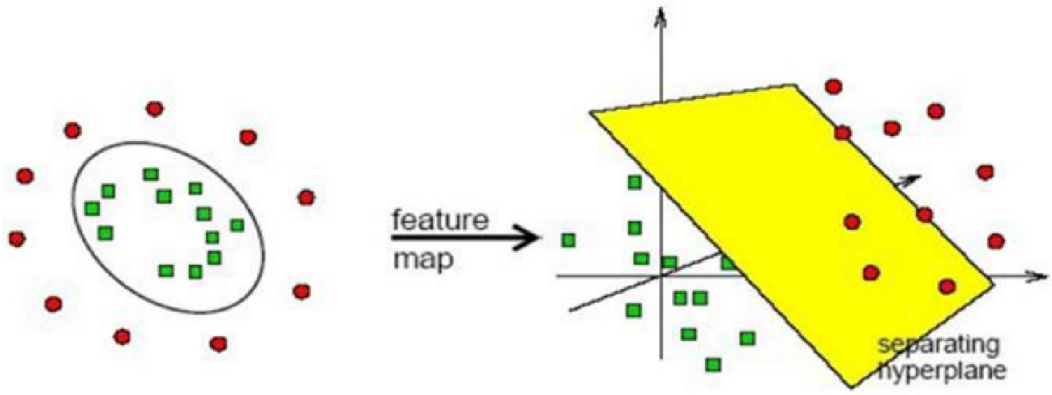
指的是在特征空间中的数据可以通过一个线性超平面（例如，一条直线或一个平面）将不同的类别分开。这表示存在一种线性关系，可以完全分隔不同类别的数据点。线性可分问题的例子包括一些简单的分类任务，如线性二分类问题。

**线性不可分：**

线性不可分"是指样本不能通过一条直线（或者超平面）进行完美地分割。

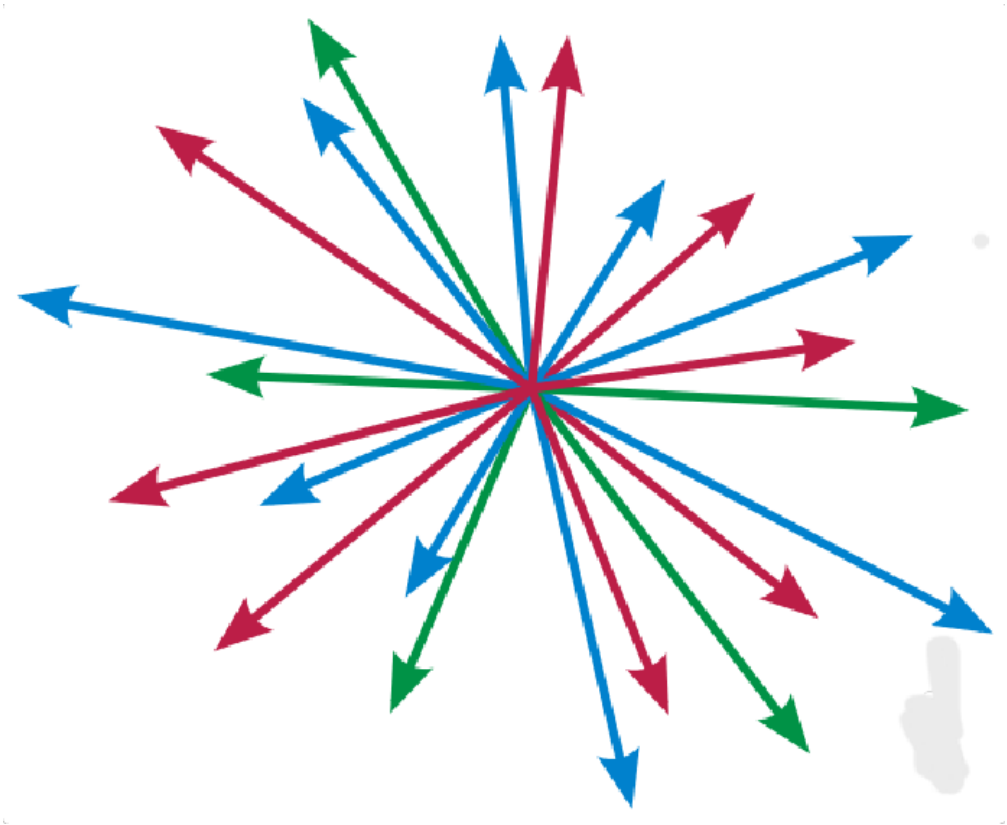
> 定义：在 n 维空间中，超平面是一个 n-1 维的线性子空间。这意味着如果我们处于三维空间（n=3），那么超平面将是一个二维的平面。在更高维度的空间中，超平面是一个 n-1 维的平面或子空间。

# 4. 核方法（函数了解）




**核方法**：使用核方法，将数据映射到更高维的特征空间，以使数据在高维度空间中变得线性可分。常见的核函数包括径向基函数（RBF核）。

# 5. 标量&向量



- 标量：一个单独的数 i = 1
- 向量：具有大小和方向的量（一维），带箭头的线段，箭头代表向量的方向，线段长度代表向量的大小。可以看成是只有一行或一列的矩阵

# 6. 矩阵&张量

**在深度学习中，我们通常将数据以张量的形式进行表示，比如我们用三维张量表示一个RGB图像，四维张量表示视频。**

矩阵：矩阵是向量的升维，例如 （3×3） 个数字的网格，二维数组

```
[
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 9]
]
```

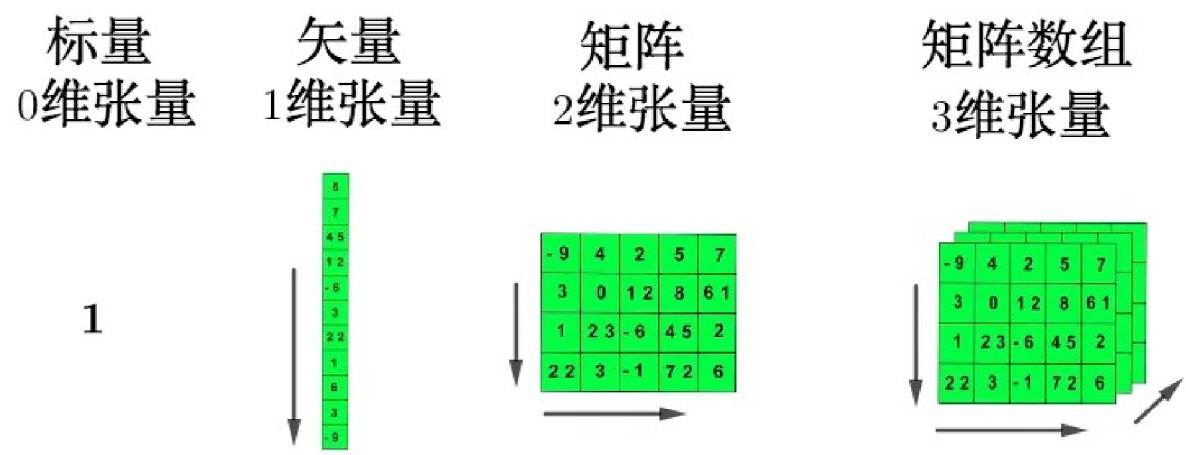
几何代数中定义的张量是基于向量和矩阵的推广，比如我们可以将标量视为零阶张量，矢量可以视为一阶张量，矩阵就是二阶张量。

```
[
    [[1,2],[3,4]],
    [[5,6],[7,8]]
]
```

几何代数中定义的张量是基于向量和矩阵的推广，比如我们可以将标量视为零阶张量，矢量可以视为一阶张量，矩阵就是二阶张量。

| 张量维度 | 代表含义 |
| --- | --- |
| 0维张量 | 代表的是标量（数字） |
| 1维张量 | 代表的是向量 |
| 2维张量 | 代表的是矩阵 |
| 3维张量 | 时间序列数据 股价 文本数据 单张彩色图片(**RGB**) |

**张量**




**标量&向量&矩阵&张量之间的关系**

- 标量是0维空间中的一个点
- 向量是一维空间中的一条线
- 矩阵是二维空间的一个面
- 三维张量是三维空间中的一个体
- 向量是由标量组成的，矩阵是向量组成的，三维张量是矩阵组成

例子:

在机器学习工作中，我们经常要处理不止一张图片，我们要处理一个集合。我们可能有10,000张小黄人的图片，这意味着，我们将用到4D张量：

如果一张图像表示为:

```
(height, width, channel) = 3D
```

```
(batch_size, height, width, channel) = 4D
```

# 7. 张量运算框架Pytorch

PyTorch是一个开源的Python机器学习库，是一个以Python优先的深度学习框架，不仅能够实现强大的GPU加速，同时还支持动态神经网络，它提供两个高级功能：

- 具有强大的GPU加速的张量计算。
- 包含自动求导系统的深度神经网络。

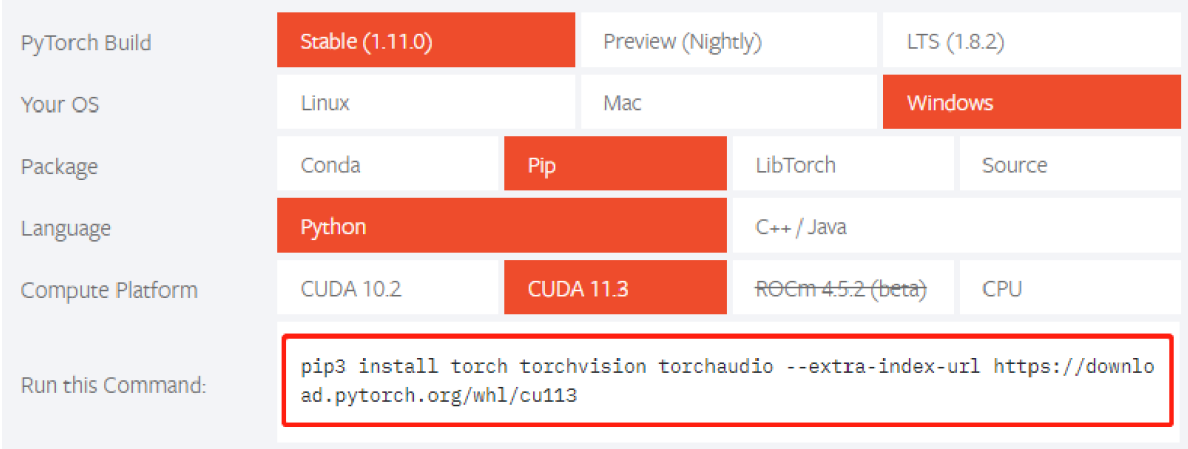
在PyTorch中，`torch.Tensor`是存储和变换数据的主要工具。之前用过`NumPy`，会发现`Tensor`和NumPy的多维数组非常类似。然而，`Tensor`提供GPU计算和自动求梯度等更多功能，这些使`Tensor`这一数据类型更加适合深度学习。

NumPy和PyTorch都是Python中用于数值计算和科学计算的库，但它们有一些重要的区别：

1. **Tensor（张量）支持**:
   - NumPy是一个用于数学计算的库，它提供了多维数组（ndarray），但不具备深度学习框架的内置支持。
   - PyTorch是一个深度学习框架，它引入了张量（tensor）这一概念，提供了高度优化的张量操作，使其更适合神经网络的构建和训练。
2. **深度学习支持**:
   - PyTorch是一个深度学习框架，专门设计用于神经网络的创建、训练和部署。它提供了自动微分（Autograd）功能，使梯度计算和反向传播更容易实现。
   - NumPy主要用于传统数学和科学计算任务，没有内置的深度学习支持。
3. **GPU加速**:
   - PyTorch支持GPU加速，允许在GPU上进行张量操作，从而大大提高深度学习模型的训练速度。
   - NumPy通常需要额外的库（如CuPy）来实现GPU加速，不如PyTorch直接支持GPU操作方便。

# 7.1 安装

**官网安装，pip命令；CUDA(11.3)安装；cadnn (11.3)安装**




**如果是集成显卡选择CPU**

# 7.2 张量的声明&类型转换

## 7.2.1 类型

Torch定义了10种带有CPU和GPU变体的张量类型，如下所示:

| Data type | dtype | CPU tensor | GPU tensor |
| --- | --- | --- | --- |
| 32-bit floating point | torch.float32 or torch.float | torch.FloatTensor | torch.cuda.FloatTensor |
| 64-bit floating point | torch.float64 or torch.double | torch.DoubleTensor | torch.cuda.DoubleTensor |
| 16-bit floating point 1 | torch.float16 or torch.half | torch.HalfTensor | torch.cuda.HalfTensor |
| 16-bit floating point 2 | torch.bfloat16 | torch.BFloat16Tensor | torch.cuda.BFloat16Tensor |
| 32-bit complex | torch.complex32 | | |
| 64-bit complex | torch.complex64 | | |
| 128-bit complex | torch.complex128 or torch.cdouble | | |
| 8-bit integer (unsigned) | torch.uint8 | torch.ByteTensor | torch.cuda.ByteTensor |
| 8-bit integer (signed) | torch.int8 | torch.CharTensor | torch.cuda.CharTensor |
| 16-bit integer (signed) | torch.int16 or torch.short | torch.ShortTensor | torch.cuda.ShortTensor |
| 32-bit integer (signed) | torch.int32 or torch.int | torch.IntTensor | torch.cuda.IntTensor |
| 64-bit integer (signed) | torch.int64 or torch.long | torch.LongTensor | torch.cuda.LongTensor |
| Boolean | torch.bool | torch.BoolTensor | torch.cuda.BoolTensor |
| quantized 8-bit integer (unsigned) | torch.quint8 | torch.ByteTensor | / |
| quantized 8-bit integer (signed) | torch.qint8 | torch.CharTensor | / |
| quantized 32-bit integer (signed) | torch.qfint32 | torch.IntTensor | / |
| quantized 4-bit integer (unsigned) 3 | torch.quint4x2 | torch.ByteTensor | / |

效率对比:

```python
import numpy as np
import torch
import time

# 创建一个大数组/张量
size = 10000
numpy_array = np.random.rand(size, size)
# pytorch_tensor = torch.rand(size, size)
pytorch_tensor = torch.from_numpy(numpy_array)
pytorch_tensor = pytorch_tensor.cuda()
# 使用NumPy进行矩阵乘法
start_time = time.time()
# 使用 np.dot 计算一维数组的点积（内积）
# # 在多维数组的情况下，np.dot 会执行矩阵乘法操作
result_numpy = np.dot(numpy_array, numpy_array)
numpy_time = time.time() - start_time

# 使用PyTorch进行矩阵乘法
start_time = time.time()
result_pytorch = torch.mm(pytorch_tensor, pytorch_tensor)
pytorch_time = time.time() - start_time

print("NumPy运行时间:", numpy_time)
print("PyTorch运行时间:", pytorch_time)
print(result_numpy[:5])
print('*' * 50)
```

```
print(result_pytorch[:5])
```

备注:
[CPU和GPU效果对比][https://www.bilibili.com/video/BV1ry4y1y7KZ/?spm_id_from=333.337.search-card.all.click&vd_source=ef93d04beba44cdc55055d214bf1f69c]

### 7.2.2 创建tensor

```
t1 = torch.Tensor([1, 2, 3])
print(t1)
print(t1.dtype)
t2 = torch.tensor([1.0, 2, 3])
t2 = t2.cuda()
print(t2)
print(t2.dtype)
t3 = torch.tensor([1, 2, 3], dtype=torch.float32)
#t3 = torch.FloatTensor([1, 2, 3])
t3 = t3.cuda()
# 选择第一个 CUDA 设备
torch.cuda.set_device(0)
# 创建一个在 GPU 上的浮点型张量
cuda_tensor = torch.cuda.FloatTensor([1.0, 2.0, 3.0])
# 执行一些操作
result = cuda_tensor * 2
print(result)
```

## 7.3 tensor 形状变换&形状获取

```
a = torch.Tensor([[[1, 2, 3], [7, 8, 9]], [[4, 5, 6], [1, 1, 1]]])
a = a.reshape(2, 6)
a = a.view(2, 6)
a = a.reshape(-1)
print(a)
a = a.reshape(-1, 3)
print(a)
a = a.reshape(-1, 3, 2)
print(a)
```

## 7.4 tensor 和 numpy.ndarray() 转换

```
data = [[1, 2], [3, 4]]
n = np.array(data)
# numpy转换成tensor
t = torch.tensor(n)
# tensor转换成numpy
n1 = t.numpy()
# numpy转换成tensor
n2 = torch.from_numpy(n1)
print(type(n)), print(type(t)), print(type(n1)), print(type(n2))
```

## 7.5 张量的声明

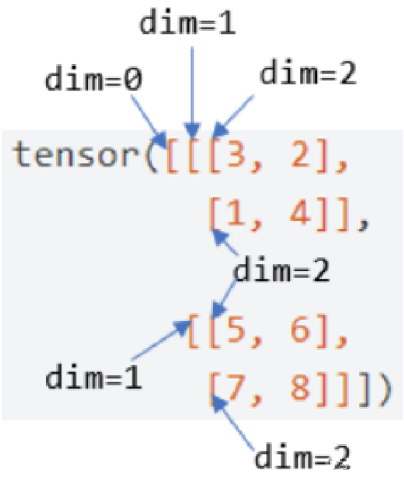
```
a = torch.empty(5, 3)
b = torch.rand(5, 3)
c = torch.zeros(5, 3, dtype=torch.long)
d = torch.ones(5, 3, dtype=torch.long)
# 打印张量
print(a), print(b), print(c), print(d)
# 获取张量的形状
print(a.size()), print(b.size()), print(c.size()), print(d.size())
```

## 7.6 CPU张量和GPU张量之间的转换

```
tensor = torch.rand(3, 4)
print(f"tensor的形状: {tensor.shape}")
print(f"tensor的数据类型: {tensor.dtype}")
print(f"tensor的运算形式: {tensor.device}")
tensor = tensor.cuda()  # GPU运算
print(f"tensor的运算形式: {tensor.device}")
tensor = tensor.cpu()  # CPU运算
print(f"tensor的运算形式: {tensor.device}")
```

## 7.7 张量的维度运算：dim属性

```
a = torch.Tensor([[[1, 2, 3], [7, 8, 9]], [[4, 5, 6], [1, 1, 1]]])
print(a.sum(dim=0))
print(a.sum(dim=1))
print(a.sum(dim=2))
```




## 7.8 张量的广播机制

当对两个形状不同的 Tensor 按元素运算时，可能会触发广播(broadcasting)机制：先适当复制元素使这两个 Tensor 形状相同后再按元素运算。

```
x = torch.arange(1, 3).view(1, 2)
print(x)
y = torch.arange(1, 4).view(3, 1)
print(y)
print(x + y)
tensor([[1, 2]])
tensor([[1],
        [2],
        [3]])
tensor([[2, 3],
        [3, 4],
        [4, 5]])
```

由于x和y分别是1行2列和3行1列的矩阵，如果要计算x+y，那么x中第一行的2个元素被广播 (复制)到了第二行和第三行，而y中第一列的3个元素被广播(复制)到了第二列。如此，就可以对2个3行2列的矩阵按元素相加。

## 7.9 torch的常用函数

```
torch.numel(a)       # 返回张量中元素的个数
torch.eye(5)         # 创建 5 * 5张量，对角线为1
a.item()             # 张量是一个数的时候才能使用
a.int()              # 张量转换为int（float、short、long）类型
a.view(16)  #返回一个有相同数据但形状不同的tensor(同reshape)
torch.transpose(a, 0, 1)     # 交换一个tensor的两个维度
```

## 7.10 矩阵和张量的基本运算

- 加/减/乘/除
- 点乘
- 数乘
- 叉乘

- 转置

### 7.10.1 对应位置相乘

$$a=[a_1,a_2,\ldots a_n] \quad b=[b_1,b_2,\ldots b_n]$$

a和b的点积公式为：

$$a \bullet b = a_1b_1 + a_2b_2 + \ldots + a_nb_n$$

```
t1 = torch.arange(8).reshape(2, 4)
t2 = torch.arange(9, 17).reshape(2, 4)
print(t1 * t2)
```

### 7.10.2 multiply()、mul()（多维）

```
t1 = torch.arange(8).reshape(2, 4)
t2 = torch.arange(9, 17).reshape(2, 4)
print(t1), print(t2)
print(torch.multiply(t1, t2))  # mul()平替
"""
t1:          [[0, 1, 2, 3], [4, 5, 6, 7]]
t2:          [[9, 10, 11, 12], [13, 14, 15, 16]]
multiply:    [[0, 10, 22, 36], [ 52, 70, 90, 112]]
"""
```

### 7.10.3 torch.mm()二维、matmul()高维

```
import torch

# 创建两个矩阵
matrix1 = torch.tensor([[1, 2], [3, 4]])
matrix2 = torch.tensor([[5, 6], [7, 8]])

# 使用 torch.matmul 执行矩阵乘法
result = torch.matmul(matrix1, matrix2)
print("torch.matmul 结果:")
print(result)

# 创建两个具有相同维度的三维张量
tensor1 = torch.tensor([
    [[1, 2],
     [3, 4]],
    [[5, 6],
     [7, 8]]
])
tensor2 = torch.tensor([
```

```
    [[9, 10],
     [11, 12]],
    [[13, 14],
     [15, 16]]
])

# 使用 torch.matmul 执行多维张量的矩阵乘法
# 2 2 2
result = torch.matmul(tensor1, tensor2)
print("torch.matmul 结果:")
"""
tensor([[[ 31,  34],
         [ 71,  78]],

        [[155, 166],
         [211, 226]]])
"""
print(result)

# 创建两个具有相同形状的张量
tensor1 = torch.tensor([1, 2, 3])
tensor2 = torch.tensor([4, 5, 6])

# 使用 torch.mul 执行逐元素的乘法
result = torch.mul(tensor1, tensor2)
print("torch.mul 结果:")
print(result.shape)
tensor([ 4, 10, 18])
print(result)

# 创建两个二维矩阵
matrix1 = torch.tensor([[1, 2],
                        [3, 4]])
matrix2 = torch.tensor([[5, 6],
                        [7, 8]])

# 使用 torch.mm 执行二维矩阵乘法
result = torch.mm(matrix1, matrix2)
#tensor([[19, 22],
#        [43, 50]])
print("torch.mm 结果:")
print(result)
```

```
# torch.mul(a, b)是矩阵a和b对应位相乘，比如a的维度是(1, 2)，b的维度是(1, 2)，返回的仍是(1, 2)的矩阵
# torch.mm(a, b)是矩阵a和b矩阵相乘，比如a的维度是(1, 2)，b的维度是(2, 3)，返回的就是(1, 3)的矩阵
a = torch.rand(1, 2)
b = torch.rand(1, 2)
c = torch.rand(2, 3)

print(torch.mul(a, b))  # 返回 1*2 的tensor
# # RuntimeError: self must be a matrix
print(torch.mm(a, c))   # 返回 1*3 的tensor
```

matmul:

```
tensor1:
[
  [
    [1, 2],
    [3, 4]
  ],
  [
    [5, 6],
    [7, 8]
  ]
]

tensor2:
[
  [
    [9, 10],
    [11, 12]
  ],
  [
    [13, 14],
    [15, 16]
  ]
]

将执行 torch.matmul(tensor1, tensor2) 的操作，逐步计算结果的每个元素。
计算结果的形状为 (2, 2, 2)，因为两个输入张量的形状相同。
对于结果中的每个元素 (i, j, k)，我们计算如下：
(0, 0, 0): 第一个矩阵的第一个行向量 [1, 2] 与第二个矩阵的第一个列向量 [9, 11] 的内积。
结果：1*9 + 2*11 = 31
(0, 0, 1): 第一个矩阵的第一个行向量 [1, 2] 与第二个矩阵的第二个列向量 [10, 12] 的内积。
结果：1*10 + 2*12 = 34
(0, 1, 0): 第一个矩阵的第二个行向量 [3, 4] 与第二个矩阵的第一个列向量 [9, 11] 的内积。
结果：3*9 + 4*11 = 71
(0, 1, 1): 第一个矩阵的第二个行向量 [3, 4] 与第二个矩阵的第二个列向量 [10, 12] 的内积。
结果：3*10 + 4*12 = 78

(1, 0, 0): 第二个矩阵的第一个行向量 [5, 6] 与第二个矩阵对应列向量 [13, 15] 的内积。
结果：5*13 + 6*15 = 155
(1, 0, 1): 第二个矩阵的第二个行向量 [5, 6] 与第二个矩阵对应列向量 [14, 16] 的内积。
结果：5*14 + 6*16 = 166
(1, 1, 0): 第二个矩阵的第二个行向量 [7, 8] 与第二个矩阵对应列向量 [13, 15] 的内积。
结果：7*13 + 8*15 = 211
(1, 1, 1): 第二个矩阵的第二个行向量 [7, 8] 与第二个矩阵对应列向量 [14, 16] 的内积。
结果：7*14 + 8*16 = 226
最终的结果张量如下：
result:
tensor([[[ 31,  34],
         [ 71,  78]],

        [[155, 166],
         [211, 226]]])
```

### 7.10.4 dot(点乘、内积，只能用于一维，且元素个数一样)

点积是两个向量对应元素相乘后的和

```
t1 = torch.tensor([1, 2, 3])
t2 = torch.tensor([4, 5, 6])
print(torch.dot(t1, t2))
# tensor(32)
```

```
t1 = torch.tensor([[1, 2, 3], [1, 2, 3]])
t2 = torch.tensor([[1, 2, 3], [1, 2, 3]])
# RuntimeError: 1D tensors expected, but got 2D and 2D tensors
print(torch.dot(t1, t2))
```

### 7.10.5 转置

```
a = torch.arange(1, 5).reshape(2, 2)
print(a)
print(a.T)
"""
tensor([[1, 2], [3, 4]])
tensor([[1, 3], [2, 4]])
"""
```

### 7.10.6 连接

```
import torch
a = torch.tensor([[1], [2]])
b = torch.tensor([[3], [4]])
c = torch.cat([a, b], dim=0)    # 按行连接
d = torch.cat([a, b], dim=1)    # 按列连接
print(c), print(d)
```

### 7.10.7 压缩torch.squeeze()

```
x = torch.zeros(2, 1, 2, 1, 2)
print(x.size())
print(torch.squeeze(x).size())
# 运行结果：
torch.Size([2, 1, 2, 1, 2])
torch.Size([2, 2, 2])
```

### 7.11.9 argmax()、eq()、mean()

```
import torch
x = torch.tensor([1, 2, 3, 4, 5], dtype=float)
y = torch.tensor([1, 2, 3, 4, 5])
print(torch.argmax(x))  # 求集合中最大元素值的索引
print(torch.eq(x, y))  # 比较是否相等
print(torch.mean(x))  # 求平均
```

# 8 动态计算图

## 8.1.可微分性相关属性

requires_grad属性：可微分性

```
# 构建可微分张量
x = torch.tensor(1.,requires_grad = True)
# 显示结果
# tensor(1., requires_grad=True)
```

```
# 构建函数关系
y = x ** 2
```

grad_fn属性：存储Tensor微分函数
此时张量y具有了一个`grad_fn`属性，并且取值为`<PowBackward0>`，我们可以查看该属性

```
print(y.grad_fn)
print(y)
# tensor(1., grad_fn=<PowBackward0>)
```

grad_fn`存储了可微分张量在进行计算的过程中函数关系，此处x到y其实就是进行了幂运算。

需要注意的是，变量 `y` 不仅仅是与变量 `x` 存在幂运算关系（表示为 `y = x**2`），更重要的是，`y` 本身是通过对 `x` 进行张量计算而得到的结果。

现在接着引入了一个新的变量 `z`，定义为 `z = y + 1`

```
z = y + 1
print(z)
# tensor(2., grad_fn=<AddBackward0>)
print(z.grad_fn)
# <AddBackward0 at 0x200a2037648>
```

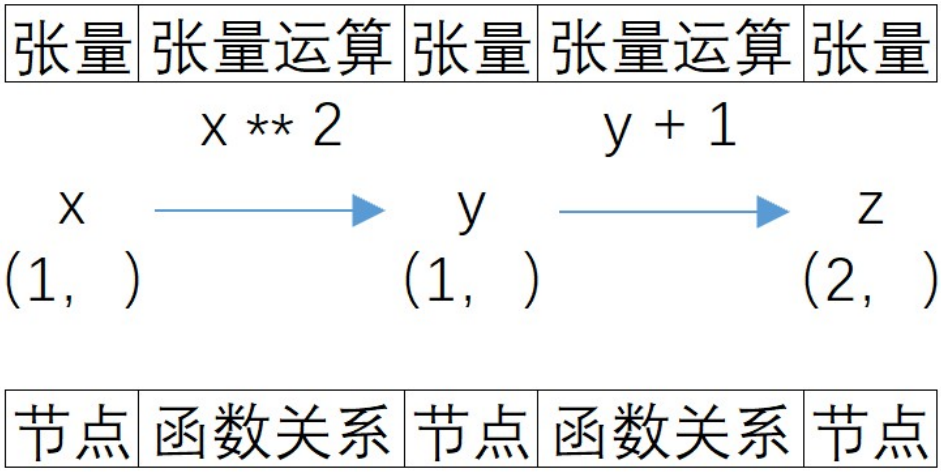
小结:

1. **可微性质：** 在PyTorch的张量计算过程中，如果我们从一个可微的初始张量开始，通过一系列计算得到的新张量 `z` 也是可微的。这意味着我们可以计算关于 `z` 的梯度，了解它对于输入的变化是如何响应的。
2. **数值存储：** 张量 `z` 同时存储了计算过程中的数值结果。这表示我们可以直接访问 `z` 的值，了解最终计算的数值。
3. **函数关系保存：** PyTorch巧妙地保存了每一步计算的函数关系，包括和变量 `y` 的计算关系，例如加法操作。这个特性被称为回溯机制。

4. **回溯机制的作用：** 回溯机制使得我们能够清晰地追溯每一步计算，了解张量是如何由初始张量一步步计算而来的。这对于深度学习中的自动微分和模型优化非常有用。
5. **绘制计算图：** 基于回溯机制，我们能够非常清楚地掌握整个张量计算的过程，并可以根据这些信息绘制出张量计算图，展示每个张量是如何相互关联的。

## 8.2 张量计算图

借助回溯机制，将张量的复杂计算过程抽象为一张图（Graph），例如此前我们定义的x、y、z三个张量，三者的计算关系就可以由下图进行表示。




**计算图**

上图就是用于记录可微分张量计算关系的张量计算图，图由节点和有向边构成，其中节点表示张量，边表示函数计算关系，方向则表示实际运算方向。

**节点类型**

在张量计算图中，每个节点代表可微分张量，但它们之间有一些差异。以前例为例，变量 `y` 和 `z` 保存了函数计算关系，而变量 `x` 则没有。在实际计算中，我们可以观察到 `z` 是所有计算的终点。因此，尽管 `x`、`y`、`z` 都是节点，

此处我们可以将节点分为三类，分别是：

1. **叶节点**： 这些节点表示初始输入的可微分张量，通常是我们在计算图中的起始点。在前例中，变量 `x` 就是一个叶节点，它是计算的起点。
2. **输出节点：** 这些节点表示最后计算得出的张量，它们是整个计算图的终点。在前例中，变量 `z` 是一个输出节点，代表了计算的结果。
3. **中间节点：** 除了叶节点和输出节点外，其他节点都被归类为中间节点。这些节点在计算图中扮演着连接和转换信息的角色。在前例中，变量 `y` 是一个中间节点，它参与了计算，并保存了与变量 `x` 和 `z` 的函数计算关系。

## 8.3 计算图的动态性

PyTorch的计算图是动态计算图，会根据可微分张量的计算过程自动生成，并且伴随着新张量或运算的加入不断更新，这使得PyTorch的计算图更加灵活高效，并且更加易于构建。

## 8.4 阻止计算图追踪

在默认情况下，只要初始张量是可微分张量，系统就会自动追踪其相关运算，并保存在计算图关系中，我们也可通过grad_fn来查看记录的函数关系，但在特殊的情况下，我们并不希望可微张量从创建到运算结果输出都被记录，此时就可以使用一些方法来阻止部分运算被记录。

## with torch.no_grad()

> with torch.no_grad()：阻止计算图记录

例如，我们希望x、y的函数关系被记录，而y的后续其他运算不被记录，可以使用with torch.no_grad()来阻止部分运算不被记录。

```
x = torch.tensor(1.,requires_grad = True)
y = x ** 2
with torch.no_grad():
    z = y ** 2
```

`requires_grad=True`：这是张量的一个属性参数。当设置 `requires_grad` 为 `True` 时，PyTorch会开始跟踪在张量上的所有操作，构建计算图以用于梯度计算。这允许使用自动微分进行梯度反向传播。

如果一个张量具有 `requires_grad=True`，那么在执行与该张量相关的操作时，PyTorch将跟踪这些操作，并在后续调用 `.backward()` 方法时计算相对于该张量的梯度。

默认情况下，`requires_grad` 的值是 `False`，这意味着张量不会跟踪操作，并且不会计算梯度。

with相当于是一个上下文管理器，with torch.no_grad()内部代码都“屏蔽”了计算图的追踪记录

```
z
# tensor(1.)
z.requires_grad
# False
y
# tensor(1., grad_fn=<PowBackward0>)
```

异常

```
import torch

with torch.no_grad():
    # 在这个代码块中，任何张量的梯度都不会被计算
    x = torch.randn(2, 2, requires_grad=True)
    y = x * 2
    z = y.mean()
    # 在这里，z.backward() 将会报错，因为梯度计算被禁用了
    z.backward()
```

主要作用包括：

1. **节省内存：** 在推理阶段，不需要计算梯度，因为模型的参数不再更新。使用 `torch.no_grad()` 可以阻止 PyTorch 保存梯度相关的信息，从而减少内存占用。
2. **避免不必要的计算：** 在一些情况下，只需要模型的预测结果而不需要梯度信息。禁用梯度计算可以避免进行不必要的计算。

## .detach()方法

> .detach()：创建一个不可导的相同张量

在某些情况下，我们也可以创建一个不可导的相同张量参与后续运算，从而阻断计算图的追踪

```
x = torch.tensor(1.,requires_grad = True)
y = x ** 2
y1 = y.detach()
z = y1 ** 2
y
# tensor(1., grad_fn=<PowBackward0>)
y1
# tensor(1.)
z
# tensor(1.)
```

```
x = torch.randn(2, 2, requires_grad=True)
y = x.detach()
# y 与 x 具有相同的值，但是 y 不再与计算图有关
print(y.grad_fn)
# element 0 of tensors does not require grad and does not have a grad_fn
# y.backward()
```

`detach()` 方法的作用是创建一个新的张量，该张量与原始张量共享相同的数据，但是与计算图的历史关系被断开。这意味着通过 `detach()` 创建的张量不再参与梯度计算，不会影响到反向传播过程。

具体来说，`detach()` 方法有以下作用：

1. **阻断梯度传播：** 通过 `detach()` 创建的张量不再与计算图有关，因此它们不会参与反向传播，不会对梯度的计算产生影响。这对于需要保留某个张量值但不希望其梯度传播到其他张量的情况非常有用。
2. **避免内存占用：** 使用 `detach()` 创建的张量与原始张量共享数据，而不是复制数据，因此不会占用额外的内存。这使得在不影响梯度计算的情况下，可以有效地共享张量的值。
3. **保存值状态：** 有时候在训练过程中，我们可能需要保留某个张量在某个时刻的值，但不希望这个值参与后续的梯度计算。`detach()` 允许我们在需要时截断计算图，保留值状态。

# 9 自动微分模块

PyTorch 的自动微分模块是其深度学习框架中的一个关键组件，它使得在神经网络中进行梯度计算变得非常方便。这个模块的核心是 `torch.autograd` 包，它提供了自动微分的功能。

## 9.1 单标量梯度的计算

```python
# 单标量梯度的计算
def test01():
    # 定义求导的张量，张量的类型必须是浮点类型 如果需要自动求导 requires_grad设置为True，
    # dtype是torch.float32的原因是求导之后有小数
    x = torch.tensor(20, requires_grad=True, dtype=torch.float32)
    # 变量经过中间变量运算
    y = x ** 2 + 10
    # 自动微分
    y.backward()
    # 打印变量x的梯度
    # tensor(40.)
    print(x.grad)
```

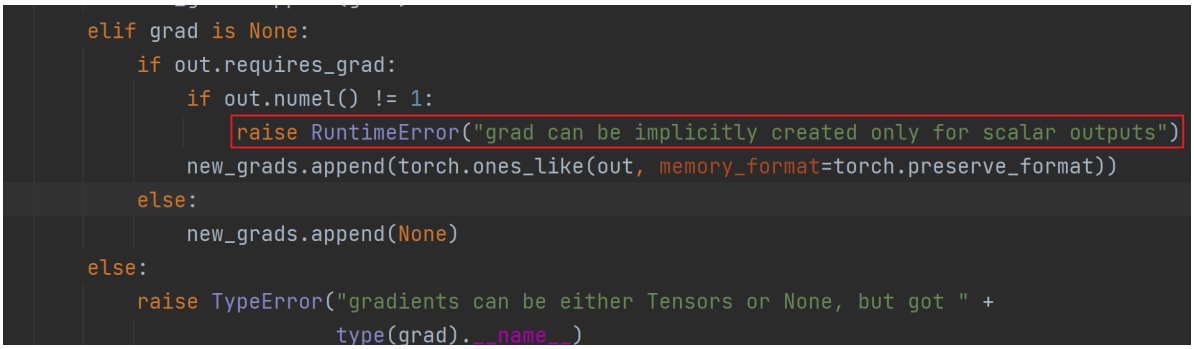
## 9.2 单向量的梯度计算

```python
# 单向量的梯度计算
def test02():
    # 定义需要求导的张量
    x = torch.tensor([10, 20, 30, 40], requires_grad=True, dtype=torch.float32)
    # 计算
    y = x**2 + 10
    # 自动微分 如果是向量不能够直接backward，必须是一个标量
    # 类似之前线性回归，在进行梯度计算的时候有一个损失值，再反向进行梯度计算
    # 取均值或者求和均可
    # 1/4 * 2x
    # 求均值操作有助于稳定梯度计算，减少梯度爆炸或梯度消失的可能性。
    # 在神经网络的训练中，通常使用均值而不是求和，以获得更稳定的梯度更新。
    y2 = y.mean()
    # y = y.sum()
    # RuntimeError: grad can be implicitly created only for scalar outputs
    y2.backward()
    print(x.grad)
```

张量 x 的定义：
x 是一个包含四个元素的张量，分别为 [10, 20, 30, 40]。
requires_grad=True 表示你希望计算关于 x 的梯度。
计算 y：
y = x**2 + 10 表示对 x 中的每个元素进行平方操作，然后再加上 10。
计算 y2：
y2 = y.mean() 表示对 y 中的所有元素取均值。
反向传播：
y2.backward() 表示从 y2 开始进行反向传播，计算关于 x 的梯度。
结果分析：
y2 对 x 的梯度计算是基于链式法则的。
对于 y2 = y.mean()，它等价于 y2 = (1/N) * (y[0] + y[1] + y[2] + y[3])，其中 N 是元素个数。
对于 y = x**2 + 10，梯度计算为 [2*x[0], 2*x[1], 2*x[2], 2*x[3]]。
最终的梯度结果是 [2*10/4, 2*20/4, 2*30/4, 2*40/4]，即 [5, 10, 15, 20]。
所以，最终的输出结果是 tensor([5., 10., 15., 20.])，代表了 y2 对 x 的梯度。这是对每个元素的平均梯度。

```
# 单向量的梯度计算
import torch

# 定义需要求导的张量
x = torch.tensor([10, 20, 30, 40], requires_grad=True, dtype=torch.float32)
# 计算
y = x**2 + 10
# RuntimeError: grad can be implicitly created only for scalar outputs
y.backward()
print(x.grad)
```




输出张量 `out` 需要梯度（`requires_grad=True`），则会检查输出张量是否为标量（`out.numel() != 1`）。如果输出张量不是标量，就会触发 `RuntimeError`。

在PyTorch的`backward()`方法默认只能应用于标量输出。这是为了避免梯度计算的混乱，因为对于非标量的情况，存在多个可能的梯度值。

```
import torch

# 定义需要求导的张量
x = torch.tensor([1.0, 2.0, 3.0], requires_grad=True)
y = x**2

# 对 y 进行反向传播
# RuntimeError: grad can be implicitly created only for scalar outputs
# y.backward()
# 输出 x 的梯度
# print(x.grad)
# 计算关于第一个元素的梯度
grad_x1 = torch.autograd.grad(y[2], x)
print(grad_x1)
```

## 9.3 多标量梯度计算

```
# 多标量梯度计算
# y = x1 ** 2 + x2** 2 + x1*2
def test03():
    x1 = torch.tensor(10, requires_grad=True, dtype=torch.float32)
    x2 = torch.tensor(20, requires_grad=True, dtype=torch.float32)
    # 计算过程
```

```
    y = x1 ** 2 + x2 ** 2 + x1*x2
    print(y)
    # 自动微分
    y.backward()
    # 查看梯度
    print(x1.grad)
    print(x2.grad)
    pass
```

## 9.4 梯度清零

### 9.4.1 梯度累加

```
# 梯度累加
def test01():
    x = torch.tensor([10, 20], requires_grad=True, dtype=torch.float32)
    for _ in range(3):
        # 对输入x进行计算
        y = x**2 + 20
        # 因为这里是一个向量需要转换为一个标量
        # 1/2 * 2*x
        y2 = y.mean()
        # 自动微分
        y2.backward()
        # 重复对x进行梯度计算会产生累加效果即累加到x的grad属性,希望不累加历史梯度
        # tensor([10., 20.])
        # tensor([20., 40.])
        # tensor([30., 60.])
        print(x.grad)
```

### 9.4.2 梯度清零

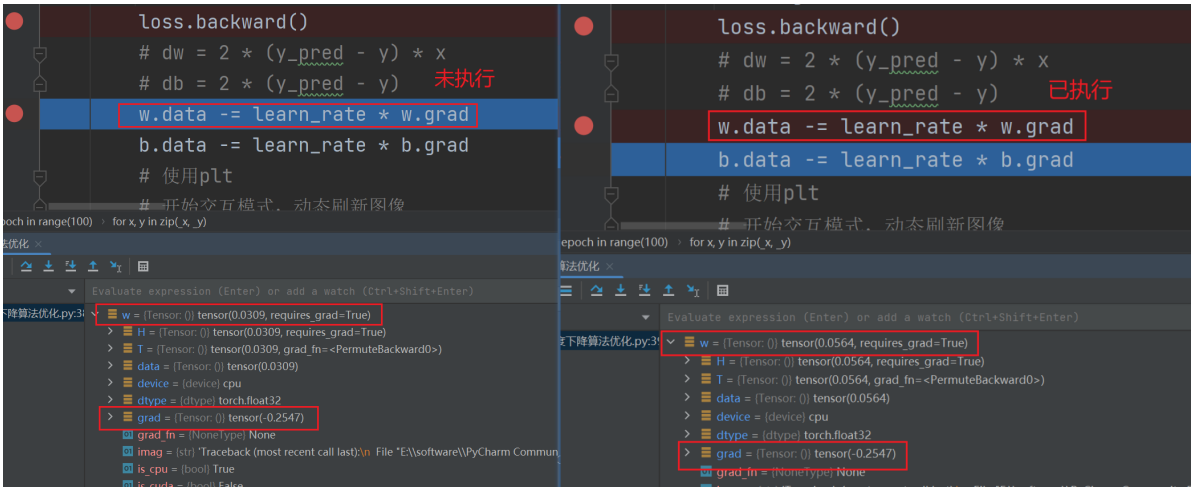
```
def test02():
    x = torch.tensor([10, 20], requires_grad=True, dtype=torch.float32)
    for _ in range(3):
        # 对输入x进行计算
        y = x**2 + 20
        # 因为这里是一个向量需要转换为一个标量
        # 1/2 * 2*x
        y2 = y.mean()
        # 第一次是没有反向传播 grad没有值，所以需要加上判断
        if x.grad is not None:
            x.grad.data.zero_()
        # 自动微分
        y2.backward()
        print(x.grad)
```

### 9.4.3 优化梯度下降算法

```
# 优化梯度下降算法
def test03():
    # loss = w ** 2
    # w值是多少的时候loss最小
    w = torch.tensor(10, requires_grad=True, dtype=torch.float32)
    for _ in range(1000):
        # 正向计算
        loss = w ** 2
        # 梯度清零
        if w.grad is not None:
            w.grad.data.zero_()
        # 反向传播
        loss.backward()
        # 更新参数
        w.data -= 0.01 * w.grad
        print(f"{w.data:.9f}")
```

```
就地修改报错:
w -= learn_rate * w.grad
requires_grad=True 的 叶子张量在求梯度过程中是不能被改变
```

参数分析




# 作业

## (1)、根据以下image、kernel矩阵得出第三个矩阵

```
image:
[[1, 2, 3, 4],
 [5, 6, 7, 8],
 [9, 10, 11, 12],
 [13, 14, 15, 16]]

kernel:
    [[1, 0],
     [0, -1]]
```

```
result:
[[-5. -5. -5.]
 [-5. -5. -5.]
 [-5. -5. -5.]]
```

**扩展:**

1、将之前的线性回归的代码修改为张量

2、按要求完成

```
data = torch.tensor(
    [
        [[0.5078, 0.2781, 0.3335, 0.4427, 0.7964],
         [0.7584, 0.2672, 0.3906, 0.4314, 0.4188]],

        [[0.7904, 0.2729, 0.2646, 0.4162, 0.4124],
         [0.5050, 0.1037, 0.2326, 0.4085, 0.4596]],

        [[0.7126, 0.4523, 0.5729, 0.4901, 0.6933],
         [0.1695, 0.1700, 0.1220, 0.4271, 0.4141]],

        [[0.7892, 0.2588, 0.3648, 0.5982, 0.4003],
         [0.1344, 0.1349, 0.2387, 0.4515, 0.4261]
         ]
    ])
```

**2.1 找出大于0.7所在的行**

```
tensor([[0.7584, 0.2672, 0.3906, 0.4314, 0.4188],
        [0.7904, 0.2729, 0.2646, 0.4162, 0.4124],
        [0.7126, 0.4523, 0.5729, 0.4901, 0.6933],
        [0.7892, 0.2588, 0.3648, 0.5982, 0.4003]])
```

### 2.2 使用切片取出以下数据

```
tensor([[0.2672, 0.3906, 0.4314, 0.4188],
        [0.2729, 0.2646, 0.4162, 0.4124],
        [0.4523, 0.5729, 0.4901, 0.6933],
        [0.2588, 0.3648, 0.5982, 0.4003]])
```

### 2.3 求出

target_box 与 boxes和**iou(查阅资料完成)**

```
tensor([[0.2672, 0.3906, 0.4314, 0.4188],
        [0.2729, 0.2646, 0.4162, 0.4124],
        [0.4523, 0.5729, 0.4901, 0.6933],
        [0.2588, 0.3648, 0.5982, 0.4003]])
 乘以100得出以下矩形框
boxes = tensor([[26, 39, 43, 41],
        [27, 26, 41, 41],
        [45, 57, 49, 69],
        [25, 36, 59, 40]], dtype=torch.int32)

目标框target_box = torch.tensor([46, 57, 49, 68])
```