第3章

机器学习基础知识

视频讲解

3.1 模型评估与模型参数选择

如何评估一些训练好的模型并从中选择最优的模型参数？对于给定的输入 $\boldsymbol{x}$，若某个模型的输出 $\hat{\boldsymbol{y}}=\boldsymbol{f}(\boldsymbol{x})$ 偏离真实目标值 $\boldsymbol{y}$，那么就说明模型存在**误差**；$\hat{\boldsymbol{y}}$ 偏离 $\boldsymbol{y}$ 的程度可以用关于 $\hat{\boldsymbol{y}}$ 和 $\boldsymbol{y}$ 的某个函数 $L(\boldsymbol{y},\hat{\boldsymbol{y}})$ 来表示，作为误差的度量标准：这样的函数 $L(\boldsymbol{y},\hat{\boldsymbol{y}})$ 称为损失函数。

在某种损失函数度量下，训练集上的平均误差被称为**训练误差**，测试集上的误差称为**泛化误差**。由于训练得到一个模型最终的目的是为了在未知的数据上得到尽可能准确的结果，因此泛化误差是衡量一个模型泛化能力的重要标准。

之所以不能把训练误差作为模型参数选择的标准，是因为训练集可能存在以下问题。

(1) 训练集样本太少，缺乏代表性。

(2) 训练集中本身存在错误的样本，即**噪声**。

如果片面地追求训练误差的最小化，就会导致模型参数复杂度增加，使得模型**过拟合**(Overfitting)，如图 3.1 所示。

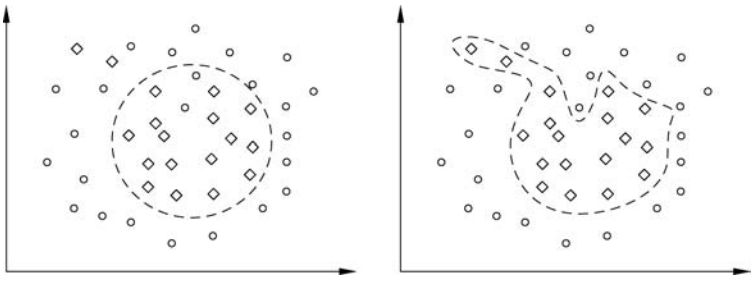

图 3.1 拟合与过拟合

为了选择效果最佳的模型，防止出现过拟合的问题，通常可以采取的方法有：使用验证集调参和对损失函数进行正则化两种方法。

3.1.1 验证

模型不能过拟合于训练集，否则将不能在测试集上得到最优结果；但是否能直接以测试集上的表现来选择模型参数呢？答案是否定的。因为这样的模型参数将会是针对某个特定测试集的，那么得出来的评价标准将会失去其公平性，失去了与其他同类或不同类模型相比较的意义。

这就好比要证明某一个学生学习某门课程的能力比别人强（模型算法的有效性），那么就要让他和其他学生听一样的课、做一样的练习（相同的训练集），然后以这些学生没做过的题目来考核他们（测试集与训练集不能交叉）；但是如果直接在测试集上调参，那就相当于让这个学生针对考试题目来复习，这样与其他学生的比较显然是不公平的。

因此参数的选择（即**调参**）必须在一个独立于训练集和测试集的数据集上进行，这样的用于模型调参的数据集被称为**开发集**或**验证集**。

然而很多时候能得到的数据量非常有限。这个时候可以不显式地使用验证集，而是重复使用训练集和测试集，这种方法称为**交叉验证**。常用的交叉验证方法有以下两种。

（1）简单交叉验证。在训练集上使用不同超参数训练，使用测试集选出最佳的一组超参数设置。

（2）K-重交叉验证（K-fold cross validation）。将数据集划分成 K 等份，每次使用其中一份作为测试集，剩余的为训练集；如此进行 K 次之后，选择最佳的模型。

3.1.2 正则化

为了避免过拟合，需要选择参数复杂度最小的模型。这是因为如果有两个效果相同的模型，而它们的参数复杂度不相同，那么冗余的复杂度一定是由于过拟合导致的。为了选择复杂度较小的模型，一种策略是在优化目标中加入**正则化项**，以惩罚冗余的复杂度：

$$\min_{\theta} L(\boldsymbol{y},\hat{\boldsymbol{y}};\boldsymbol{\theta})+\lambda \cdot J(\boldsymbol{\theta})$$

其中，$\boldsymbol{\theta}$ 为模型参数，$L(\boldsymbol{y},\hat{\boldsymbol{y}};\boldsymbol{\theta})$为原来的损失函数，$J(\boldsymbol{\theta})$是正则化项，$\lambda$ 用于调整正则化项的权重。正则化项通常为 $\boldsymbol{\theta}$ 的某阶向量范数。

3.2 监督学习与非监督学习

模型与最优化算法的选择，很大程度上取决于能得到什么样的数据。如果数据集中样本点只包含模型的输入 $\boldsymbol{x}$，那么就需要采用非监督学习的算法；如果这些样本点以$\langle \boldsymbol{x},\boldsymbol{y}\rangle$这样的输入-输出二元组的形式出现，那么就可以采用监督学习的算法。

3.2.1　监督学习

在监督学习中，根据训练集$\{\langle \boldsymbol{x}^{(i)}, \boldsymbol{y}^{(i)} \rangle\}_{i=1}^{N}$中的观测样本点来优化模型$\boldsymbol{f}(\cdot)$，使得给定测试样例$\boldsymbol{x}'$作为模型输入，其输出$\hat{\boldsymbol{y}}$尽可能接近正确输出$\boldsymbol{y}'$。

监督学习算法主要适用于两大类问题：回归和分类。这两类问题的区别在于：回归问题的输出是连续值，而分类问题的输出是离散值。

1. 回归

回归问题在生活中非常常见，其最简单的形式是一个连续函数的拟合。如果一个购物网站想要计算出其在某个时期的预期收益，研究人员会将相关因素如广告投放量、网站流量、优惠力度等纳入自变量，根据现有数据拟合函数，得到在未来某一时刻的预测值。

回归问题中通常使用均方损失函数来作为度量模型效果的指标，最简单的求解例子是最小二乘法。

2. 分类

分类问题也是生活中非常常见的一类问题，例如，需要从金融市场的交易记录中分类出正常的交易记录以及潜在的恶意交易。

度量分类问题的指标通常为**准确率**(Accuracy)：对于测试集中的D个样本，有k个被正确分类，$D-k$个被错误分类，则准确率为：

$$\text{Accuracy}=\frac{k}{D}$$

然而在一些特殊的分类问题中，属于各类的样本并不是均一分布，甚至其出现概率相差很多个数量级，这种分类问题称为**不平衡类问题**。在不平衡类问题中，准确率并没有多大意义。例如，检测一批产品是否为次品时，若次品出现的概率为1%，那么即使某个模型完全不能识别次品，只要每次都“蒙”这件产品不是次品，仍然能够达到99%的准确率。显然我们需要一些别的指标。

通常在不平衡类问题中，使用***F*-度量**来作为评价模型的指标。以二元不平衡分类问题为例，这种分类问题往往是异常检测，模型的好坏往往取决于能否很好地检出异常，同时尽可能不误报异常。定义占样本少数的类为**正类**(positive class)，占样本多数的为**负类**(negative class)，那么预测只可能出现以下4种状况。

(1) 将正类样本预测为正类(true positive, TP)。

(2) 将负类样本预测为正类(false positive, FP)。

(3) 将正类样本预测为负类(false negative, FN)。

(4) 将负类样本预测为负类(true negative, TN)。

定义**召回率**(recall)：

$$R=\frac{|\text{TP}|}{|\text{TP}|+|\text{FN}|}$$

召回率度量了在所有的正类样本中，模型正确检出的比率，因此也称为**查全率**。

定义**精确率**(precision)：

$$P=\frac{|\ \mathrm{TP}\ |}{|\ \mathrm{TP}\ |+|\ \mathrm{FP}\ |}$$

精确率度量了在所有被模型预测为正类的样本中，正确预测的比率，因此也称为**查准率**。

F-度量则是在召回率与精确率之间去调和平均数；有时候在实际问题上，若更加看重其中某一个度量，还可以给它加上一个权值 α，称为 F_α-度量：

$$F_\alpha=\frac{(1+\alpha^2)RP}{R+\alpha^2 P}$$

特殊地，当 $\alpha=1$ 时：

$$F_1=\frac{2RP}{R+P}$$

可以看到，如果模型“不够警觉”，没有检测出一些正类样本，那么召回率就会受损；而如果模型倾向于“滥杀无辜”，那么精确率就会下降。因此较高的 F-度量意味着模型倾向于“不冤枉一个好人，也不放过一个坏人”，是一个较为适合不平衡类问题的指标。

可用于分类问题的模型很多，例如，Logistic 回归分类器、决策树、支持向量机、感知器、神经网络，等等。

3.2.2 非监督学习

在非监督学习中，数据集 $\{\boldsymbol{x}^{(i)}\}_{i=1}^{N}$ 中只有模型的输入，而并不提供正确的输出 $\boldsymbol{y}^{(i)}$ 作为监督信号。

非监督学习通常用于这样的分类问题：给定一些样本的特征值，而不给出它们正确的分类，也不给出所有可能的类别；而是通过学习确定这些样本可以分为哪些类别、它们各自都属于哪一类。这一类问题称为**聚类**。

非监督学习得到的模型的效果应该使用何种指标来衡量呢？由于通常没有正确的输出 $\boldsymbol{y}$，可采取一些其他办法来度量其模型效果。

(1) 直观检测，这是一种非量化的方法。例如，对文本的主体进行聚类，可以在直观上判断属于同一个类的文本是否具有某个共同的主题，这样的分类是否有明显的语义上的共同点。由于这种评价非常主观，通常不采用。

(2) 基于任务的评价。如果聚类得到的模型被用于某个特定的任务，可以维持该任务中其他的设定不变，而使用不同的聚类模型，通过某种指标度量该任务的最终结果来间接判断聚类模型的优劣。

(3) 人工标注测试集。有时候采用非监督学习的原因是人工标注成本过高，导致标注数据缺乏，只能使用无标注数据来训练。在这种情况下，可以人工标注少量的数据作为测试集，用于建立量化的评价指标。

第4章

PyTorch深度学习基础

在介绍 PyTorch 之前，读者需要先了解 NumPy。NumPy 是用于科学计算的框架，它提供了一个 N 维矩阵对象 ndarray，初始化、计算 ndarray 的函数，以及变换 ndarray 形状，组合拆分 ndarray 的函数。

PyTorch 的 Tensor 和 NumPy 的 ndarray 十分类似，但是 Tensor 具备两个 ndarray 不具备但是对于深度学习来说非常重要的功能，其一是 Tensor 能利用 GPU 计算，GPU 根据芯片性能的不同，在进行矩阵运算时，比 CPU 速度快几十倍；其二是，Tensor 在计算时，能够作为节点自动地加入计算图当中，而计算图可以为其中的每个节点自动地计算微分，也就是说，当使用 Tensor 时，就不需要手动计算微分了。下面首先介绍 Tensor 对象及其运算。

```
1 import torch
2 import numpy as np
```

4.1 Tensor 对象及其运算

Tensor 对象是一个维度任意的矩阵，但是一个 Tensor 中所有元素的数据类型必须一致。torch 包含的数据类型和普遍编程语言的数据类型类似，包含浮点型、有符号整型和无符号整型，这些类型既可以定义在 CPU 上，也可以定义在 GPU 上。在使用 Tensor 数据类型时，可以通过 dtype 属性指定它的数据类型，device 指定它的设备（CPU 或者 GPU）。

```
 1 #torch.tensor
 2 print('torch.Tensor 默认为:{}'.format(torch.Tensor(1).dtype))
 3 print('torch.tensor 默认为:{}'.format(torch.tensor(1).dtype))
 4 #可以用 list 构建
 5 a = torch.tensor([[1,2],[3,4]], dtype = torch.float64)
 6 #也可以用 ndarray 构建
 7 b = torch.tensor(np.array([[1,2],[3,4]]), dtype = torch.uint8)
 8 print(a)
 9 print(b)
10
11 #通过 device 指定设备
12 cuda0 = torch.device('cuda:0')
13 c = torch.ones((2,2), device = cuda0)
14 print(c)
>>> torch.Tensor 默认为:torch.float32
>>> torch.tensor 默认为:torch.int64
>>> tensor([[1., 2.],
           [3., 4.]], dtype = torch.float64)
>>> tensor([[1, 2],
           [3, 4]], dtype = torch.uint8)
>>> tensor([[1., 1.],
           [1., 1.]], device = 'cuda:0')
```

通过 device 指定在 GPU 上定义变量后，可以在终端上通过 nvidia-smi 命令查看显存占用。torch 还支持在 CPU 和 GPU 之间复制变量。

```
1 c = c.to('cpu', torch.double)
2 print(c.device)
3 b = b.to(cuda0, torch.float)
4 print(b.device)
>>> cpu
>>> cpu:0
```

对 Tensor 执行算术运算符的运算时，是两个矩阵对应元素的运算。torch.mm 执行矩阵乘法的计算。

```
1 a = torch.tensor([[1,2],[3,4]])
2 b = torch.tensor([[1,2],[3,4]])
3 c = a * b
4 print("逐元素相乘:", c)
5 c = torch.mm(a, b)
6 print("矩阵乘法: ", c)
>>> 逐元素相乘: tensor([[ 1, 4],
      [ 9, 16]])
>>> 矩阵乘法: tensor([[ 7, 10],
      [15, 22]])
```

此外，还有一些具有特定功能的函数，这里列举一部分。torch.clamp 起的是分段函数的作用，可用于去掉矩阵中过小或者过大的元素；torch.round 将小数部分化整；torch.tanh 计算双曲正切函数，该函数将数值映射到(0,1)。

```
1 a = torch.tensor([[1,2],[3,4]])
2 torch.clamp(a, min = 2, max = 3)
>>> tensor([[2, 2],
           [3, 3]])
1 a = torch.tensor([ - 1.1, 0.5, 0.501, 0.99])
2 torch.round(a)
>>> tensor([[2, 2],
           [3, 3]])
1 a = torch.Tensor([ - 3, - 2, - 1, - 0.5,0,0.5,1,2,3])
2 torch.tanh(a)
>>> tensor([ - 0.9951, - 0.9640, - 0.7616, - 0.4621, 0.0000, 0.4621, 0.7616, 0.9640,
            0.9951])
```

除了直接从 ndarray 或 list 类型的数据中创建 Tensor 外，PyTorch 还提供了一些函数可直接创建数据，这类函数往往需要提供矩阵的维度。torch.arange 和 Python 内置的 range 的使用方法基本相同，其第 3 个参数是步长。torch.linspace 第 3 个参数指定返回的个数。torch.ones 返回全 0，torch.zeros 返回全 0 矩阵。

```
1 print(torch.arange(5))
2 print(torch.arange(1,5,2))
3 print(torch.linspace(0,5,10))
>>> tensor([0, 1, 2, 3, 4])
>>> tensor([1, 3])
>>> tensor([0.0000, 0.5556, 1.1111, 1.6667, 2.2222, 2.7778, 3.3333, 3.8889, 4.4444,
           5.0000])
1 print(torch.ones(3,3))
2 print(torch.zeros(3,3))
>>> tensor([[1., 1., 1.],
           [1., 1., 1.],
           [1., 1., 1.]])
>>> tensor([[0., 0., 0.],
           [0., 0., 0.],
           [0., 0., 0.]])
```

torch.rand 返回从[0,1]的均匀分布采样的元素所组成的矩阵，torch.randn 返回从正态分布采样的元素所组成的矩阵。torch.randint 返回指定区间的均匀分布采样的随机整数所组成的矩阵。

```
1 torch.rand(3,3)
>>> tensor([[0.0388, 0.6819, 0.3144],
           [0.7826, 0.0966, 0.4319],
           [0.6758, 0.2630, 0.9727]])
```

```
1 torch.randn(3,3)
>>> tensor([[ -0.6956, 0.6792, 0.8957],
           [ 0.2271, 0.9885, -0.7817],
           [ -0.2658, 1.5465, -0.2519]])
>>>
1 torch.randint(0, 9, (3,3))
>>> tensor([[5, 2, 7],
           [8, 4, 8],
           [2, 1, 4]])
```

4.2 Tensor 的索引和切片

Tensor 支持基本的索引和切片操作，不仅如此，它还支持 ndarray 中的高级索引（整数索引和布尔索引）操作。

```
1 a = torch.arange(9).view(3,3)
2 #基本索引
3 a[2,2]
>>> tensor(8)
1 #切片
2 a[1:, :-1]
>>> tensor([[3, 4],
           [6, 7]])
1 #带步长的切片(PyTorch 现在不支持负步长)
2 a[::2]
>>> tensor([[0, 1, 2],
           [6, 7, 8]])
1 #整数索引
2 rows = [0, 1]
3 cols = [2, 2]
4 a[rows, cols]
>>> tensor([2, 5])
1 #布尔索引
2 index = a>4
3 print(index)
4 print(a[index])
>>> tensor([[0, 0, 0],
           [0, 0, 1],
           [1, 1, 1]], dtype=torch.uint8)
>>> tensor([5, 6, 7, 8])
torch.nonzero 用于返回非零值的索引矩阵.
1 a = torch.arange(9).view(3, 3)
2 index = torch.nonzero(a>= 8)
3 print(index)
>>> tensor([[2, 2]])
1 a = torch.randint(0, 2, (3,3))
```

```
2 print(a)
3 index = torch.nonzero(a)
4 print(index)
>>> tensor([[0, 0, 1],
            [0, 0, 1],
            [1, 1, 0]])
>>> tensor([[0, 2],
            [1, 2],
            [2, 0],
            [2, 1]])
```

torch.where(condition, x, y)判断 condition 的条件是否满足，当某个元素满足时，则返回对应矩阵 x 相同位置的元素，否则返回矩阵 y 的元素。

```
1 x = torch.randn(3, 2)
2 y = torch.ones(3, 2)
3 print(x)
4 print(torch.where(x > 0, x, y))
>>> tensor([[ 0.0914, -0.8913],
            [-0.0046, 0.0617],
            [ 1.0744, -1.2068]])
>>> tensor([[0.0914, 1.0000],
            [1.0000, 0.0617],
            [1.0744, 1.0000]])
```

4.3 Tensor 的变换、拼接和拆分

PyTorch 提供了大量的对 Tensor 进行操作的函数或方法，这些函数内部使用指针实现对矩阵的形状变换、拼接、拆分等操作，使得人们无须关心 Tensor 在内存中的物理结构或者管理指针就可以方便且快速地执行这些操作。Tensor.nelement()，Tensor.ndimension()，ndimension.size()可分别用来查看矩阵元素的个数，轴的个数以及维度，属性 Tensor.shape 也可以用来查看 Tensor 的维度。

```
1 a = torch.rand(1,2,3,4,5)
2 print("元素个数", a.nelement())
3 print("轴的个数", a.ndimension())
4 print("矩阵维度", a.size(), a.shape)
>>> 元素个数 120
>>> 轴的个数 5
>>> 矩阵维度 torch.Size([1, 2, 3, 4, 5]) torch.Size([1, 2, 3, 4, 5])
```

在 PyTorch 中，Tensor.reshape 和 Tensor.view 都能被用来更改 Tensor 的维度。它们的区别在于，Tensor.view 要求 Tensor 的物理存储必须是连续的，否则将报错，而 Tensor.reshape 则没有这种要求。但是，Tensor.view 返回的一定是一个索引，更改返回

值，则原始值同样被更改，Tensor. reshape 返回的是引用还是复制是不确定的。它们的相同之处是都接收要输出的维度作为参数，且输出的矩阵元素个数不能改变，可以在维度中输入－1，PyTorch 会自动推断它的数值。

```
1 b = a.view(2 * 3,4 * 5)
2 print(b.shape)
3 c = a.reshape( - 1)
4 print(c.shape)
5 d = a.reshape(2 * 3, - 1)
6 print(d.shape)
>>> torch.Size([6, 20])
>>> torch.Size([120])
>>> torch.Size([6, 20])
```

torch. squeeze 和 torch. unsqueeze 用来给 Tensor 去掉和添加轴。torch. squeeze 去掉维度为 1 的轴，而 torch. unsqueeze 用于给 Tensor 的指定位置添加一个维度为 1 的轴。

```
1 b = torch.squeeze(a)
2 b.shape
>>> torch.Size([2, 3, 4, 5])
1 torch.unsqueeze(b, 0).shape
```

torch. t 和 torch. transpose 用于转置 2 维矩阵。这两个函数只接收 2 维 Tensor，torch. t 是 torch. transpose 的简化版。

```
1 a = torch.tensor([[2]])
2 b = torch.tensor([[2, 3]])
3 print(torch.transpose(a, 1, 0,))
4 print(torch.t(a))
5 print(torch.transpose(b, 1, 0,))
6 print(torch.t(b))
>>> tensor([[2]])
>>> tensor([[2]])
>>> tensor([[2],
           [3]])
>>> tensor([[2],
           [3]])
```

对于高维度 Tensor，可以使用 permute()方法来变换维度。

```
1 a = torch.rand((1, 224, 224, 3))
2 print(a.shape)
3 b = a.permute(0, 3, 1, 2)
4 print(b.shape)
>>> torch.Size([1, 224, 224, 3])
>>> torch.Size([1, 3, 224, 224])
```

PyTorch 提供了 torch. cat 和 torch. stack 用于**拼接**矩阵，不同的是，torch. cat 在已有的轴 dim 上拼接矩阵，给定轴的维度可以不同，而其他轴的维度必须相同。torch. stack 在新的轴上拼接，它要求被拼接的矩阵所有维度都相同。下面的例子可以很清楚地表明它们的使用方式和区别。

```
1 a = torch.randn(2, 3)
2 b = torch.randn(3, 3)
3
4 #默认维度为 dim = 0
5 c = torch.cat((a, b))
6 d = torch.cat((b, b, b), dim = 1)
7
8 print(c.shape)
9 print(d.shape)
>>> torch.Size([5, 3])
>>> torch.Size([3, 9])
1 c = torch.stack((b, b), dim = 1)
2 d = torch.stack((b, b), dim = 0)
3 print(c.shape)
4 print(d.shape)
>>> torch.Size([3, 2, 3])
>>> torch.Size([2, 3, 3])
```

除了拼接矩阵，PyTorch 还提供了 torch. split 和 torch. chunk 用于**拆分**矩阵。它们的不同之处在于，torch. split 传入的是拆分后每个矩阵的大小，可以传入 list，也可以传入整数，而 torch. chunk 传入的是拆分的矩阵个数。

```
1 a = torch.randn(10, 3)
2 for x in torch.split(a, [1,2,3,4], dim = 0):
3     print(x.shape)
>>> torch.Size([1, 3])
>>> torch.Size([2, 3])
>>> torch.Size([3, 3])
>>> torch.Size([4, 3])
1 for x in torch.split(a, 4, dim = 0):
2     print(x.shape)
>>> torch.Size([4, 3])
>>> torch.Size([4, 3])
>>> torch.Size([2, 3])
1 for x in torch.chunk(a, 4, dim = 0):
2     print(x.shape)
>>> torch.Size([3, 3])
>>> torch.Size([3, 3])
>>> torch.Size([3, 3])
>>> torch.Size([1, 3])
```

4.4 PyTorch 的 Reduction 操作

Reduction 运算的特点是它往往对一个 Tensor 内的元素做归约操作，比如 torch.max 找极大值，torch.cumsum 计算累加，它还提供了 dim 参数来指定沿矩阵的哪个维度执行操作。

```
1 #默认求取全局最大值
2 a = torch.tensor([[1,2],[3,4]])
3 print("全局最大值: ", torch.max(a))
4 #指定维度 dim 后,返回最大值及其索引
5 torch.max(a, dim = 0)
>>> 全局最大值: tensor(4)
>>> (tensor([3, 4]), tensor([1, 1]))
1 a = torch.tensor([[1,2],[3,4]])
2 print("沿着横轴计算每一列的累加: ")
3 print(torch.cumsum(a, dim = 0))
4 print("沿着纵轴计算每一行的累乘: ")
5 print(torch.cumprod(a, dim = 1))
>>> 沿着横轴计算每一列的累加:
>>> tensor([[1, 2],
           [4, 6]])
>>> 沿着纵轴计算每一行的累乘:
>>> tensor([[ 1, 2],
           [ 3, 12]])
1 #计算矩阵的均值,中值,协方差
2 a = torch.Tensor([[1,2],[3,4]])
3 a.mean(), a.median(), a.std()
>>> (tensor(2.5000), tensor(2.), tensor(1.2910))
1 #torch.unique 用来找出矩阵中出现了哪些元素
2 a = torch.randint(0, 3, (3, 3))
3 print(a)
4 print(torch.unique(a))
>>> tensor([[0, 0, 0],
           [2, 0, 2],
           [0, 0, 1]])
>>> tensor([1, 2, 0])
```

4.5 PyTorch 的自动微分

将 Tensor 的 requires_grad 属性设置为 True 时，PyTorch 的 torch.autograd 会自动地追踪它的计算轨迹，当需要计算微分的时候，只需要对最终计算结果的 Tensor 调用 backward 方法，中间所有计算节点的微分就会被保存在 grad 属性中了。