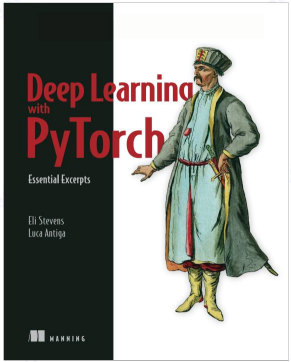
淡蓝小点技术系列：pytorch编程快速学习

淡蓝小点Bluedotdot

微信：bluedotdot_cn

2024 年 8 月 18 日

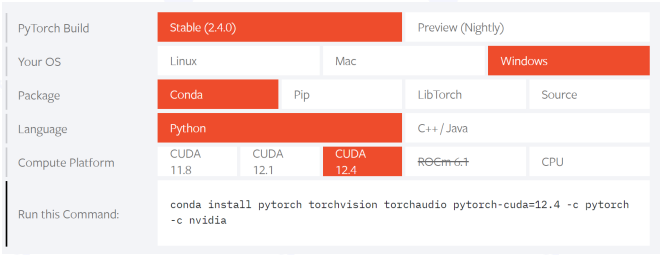
参考文献



PyTorch

官方网站网页资料

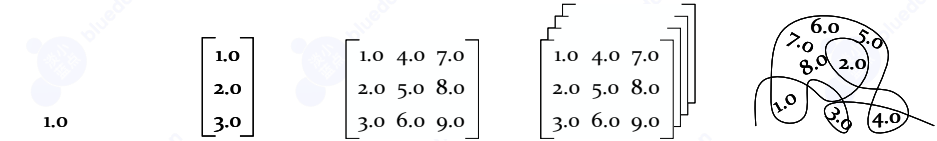
关于pytorch安装请参考pytorch网站（https://pytorch.org），根据自己环境配置选择相关安装命令



1 关于tensor

Tensor翻译成中文叫“张量”，它是pytorch中最重要、最基础的概念之一，它可以表示几乎任意类型的数据，pytorch中的计算都是基于tensor进行的

深度学习中：tensor表示“多维数组”，它是二维数组、三维数组的推广和抽象，将数据存储在一片连续的存储空间上，可利用下标随机访问[1]



标量 scalar　向量 vector　矩阵 matrix　张量 tensor　张量 tensor

[1]在物理学、工程学、控制论等诸多学科中，tensor是有特殊定义的数据表示

介绍三类tensor的构造方式：1.构造特殊类型tensor；2.利用列表或数组构造；3.指定填充方式构造

1.构造特殊类型tensor：全1张量、全0张量、对角张量[2]

```
import torch
a = torch.ones(3)
b = torch.ones(2,3)
```

```
import torch
a = torch.zeros(2)
b = torch.zeros(2,3,4)
```

```
import torch
a = torch.eye(3) # 3x3 diagonal matrix
b = torch.eye(3,4) # 3x4 matrix, only diagonal elements are 1s
```

[2]ones()函数和zeros()函数中采用了可变参数，可传递任意多维度数；eye()函数这里没有's'，它只能定义二维对角张量

2.利用列表或数组（元组）构造

```
import torch
t1_list = torch.tensor([1,2,3,4])
t2_list = torch.tensor([[1,1,1,1],[2,2,2,2]])
```

```
import torch
import numpy as np
np1_array = np.array([1,2,3,4])
t1_tensor = torch.tensor(np1_array)
np2_array = np.array([[1,2,3,4],[2,3,4,5]])
t2_tensor = torch.tensor(np2_array)
```

```
import torch
tup = (1,2,3)
tup_tensor = torch.tensor(tup)
```

3.指定填充方式构造：填充指定值、填充随机值、填充区间值

```
import torch
fill_tensor = torch.full((2,3),3.14)
```

rand()从$[0,1)$范围一致分布采样填充；*randn*()从标准正态分布采样填充；*normal*()从一般正态分布采样填充

```
import torch
u_tensor = torch.rand(2,2)
n_tensor = torch.randn(3,3,4)
nn = torch.normal(mean=2,std=3,size=(2,3))
```

arange()指定间隔但不指定样本数量；*linspace*()指定样本数量但不指定间隔[3]

```
import torch
a_tensor = torch.arange(0,10,2)
l_tensor = torch.linspace(0,10,5)
```

[3]'arange'表示array range，'linspace'表示linear space

1关于Tensor：命名张量

tensor维度很高时往往难以记清各个维度的意义，例如有些时候图像是按(*batch*, *channel*, *height*, *width*)存储的，但有些时候可能是按(*channel*, *batch*, *height*, *width*)存储的，命名张量（named tensor）可以帮助我们

```
import torch
imgs = torch.randn(2,3,30,30, names=('batch','channel','height','width'))
print(imgs.names)
```

可以通过名字重排列更简单准确的调整tensor的形状（通用办法是*view*()函数，下文介绍）

```
sw_imgs = imgs.align_to('channel','batch','height','width')
print(sw_imgs.shape, sw_imgs.names)
```

还可以通过名字对数据维度进行操作[4]

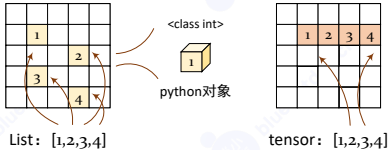
```
max_v, max_i = imgs.max('width')
mean_imgs = imgs.mean(('height','width'))
```

[4] *max_v*保留最大值，*max_i*保留最大值索引，它们的形状都是(*batch*, *channel*, *height*)；目前pytorch对于命名张量的未来持犹豫态度，未来有可能保留这一特性也有可能去掉相关接口，重要开发项目请慎重使用命名张量接口

1关于Tensor：tensor的优势

torch.tensor和numpy.array、python内建的list有什么区别？[5]

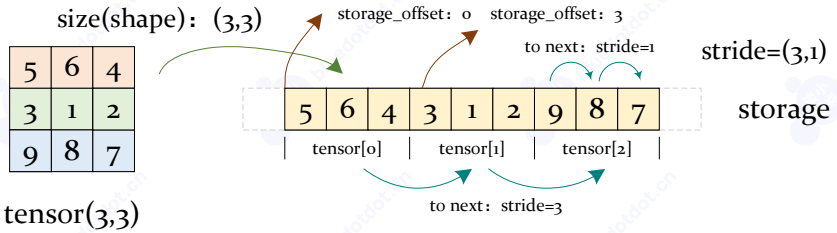
- list元素随机存储在各个位置，tensor和array占用连续存储空间
- list中存放的是类对象，tensor和array中存放的是原生数值
- tensor可作GPU加速，list、array不能
- tensor可基于链式法则自动求梯度，list、array不行
- tensor内建了大量计算操作如转置、内积等，list、array没有
- tensor可自动广播，list、array不支持需要手动调整



[5]list是python的内建数据类型，numpy.array则是几乎所有数据处理软件都能处理的数据类型，如SciPy、Scikit-learn、Pandas等，tensor是深度学习中最核心的数据结构，但tensor和array之间几乎可以无缝转换

torch.tensor有三个非常重要的元数据（meta data）：*tensor.size*()[6]、*tensor.storage_offset*()、*tensor.stride*()

- *tensor.size*()：返回一个元组（tuple）的子类，表示tensor有多少维、每个维度有多少个数据
- *tensor.storage_offset*()：返回tensor中某个对象距离storage起始处的偏移量
- *tensor.stride*()：返回当前storage下访问第$[i,j]$个对象时每行、每列应前进的距离



[6]等同于tensor.shape

1关于Tensor：tensor的存储和访问

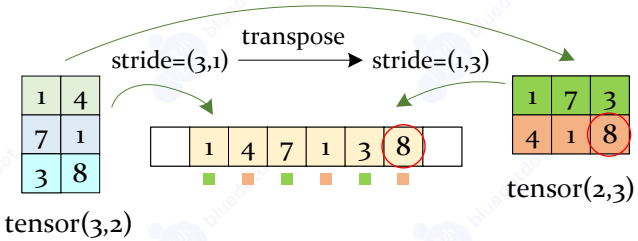
```python
import torch
a_tensor = torch.rand(3,3)
print(a_tensor.size())
print(a_tensor[0].size())

print(a_tensor.storage_offset())
print(a_tensor[1].storage_offset())
print(a_tensor[1][1].storage_offset())

print(a_tensor.stride())
print(a_tensor[0].stride())
print(a_tensor[0][0].stride())
```

```
torch.Size([3, 3])
torch.Size([3])

0
3
4

(3, 1)
(1,)
()
```

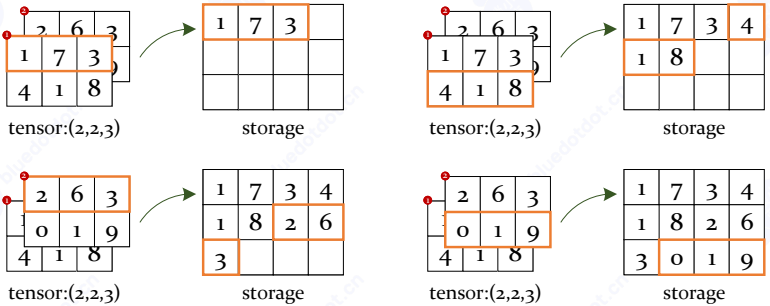
基于上述属性值可以对tensor做转置或变形而不需开辟新的存储空间，只需要修改stride即可

如下图所示，原先$(3,2)$的矩阵$stride = (2,1)$；转置后行列互换，相当于$stride$也被转置所以有$stride^T = (1,2)$。假设访问转置后的第$[i,j] = [1,2]$个元素，那它相对于$storage$的偏移量为

$$i * stride^T[0] + j * stride^T[1] = 1 * 1 + 2 * 2 = 5$$



对于多维张量，storage按从右至左的维度顺序连续排列



可利用*is_contiguous*()判定张量的storage是否是连续的，如果不是连续的，可利用*contiguous*()将其转变成连续的。此时pytorch会开辟一块新的、连续内存，并按顺序将元素值拷贝进去[7]

```
1  import torch
2
3  t = torch.rand(2,3)
4  print(id(t.untyped_storage()))
5  print(t.is_contiguous())
6
7  t_t = t.t()
8  print(id(t_t.untyped_storage()))
9  print(t_t.is_contiguous())
10
11 t_t_c = t_t.contiguous()
12 print(id(t_t_c.untyped_storage))
13 print(t_t_c.is_contiguous())
```

```
2190764611104
True

2190764611104
False

2190961127680
True
```

[7]tensor占据的是连续存储空间，为什么还会有不连续的张量？因为针对张量的操作有些返回的只是一个视图，这种情况下返回值可能是不连续的。例如对某个张量做转置操作，或对张量做切片（t = tensor.rand(2,3),t'=t[:,1:2]）

tensor在storage中可看作一维向量，所以它也可以按列表的方式以索引或切片的形式访问[8]

```
1  import torch
2
3  t = torch.rand(3,4)
4  print(t)
5  print(t[:])
6  print(t[:,1])
7  print(t[:,1:2])
8  print(t[1:3,::2])
```

```
tensor([[0.4479, 0.0409, 0.3620, 0.1004],
        [0.4771, 0.2309, 0.4251, 0.5966],
        [0.5136, 0.6100, 0.8365, 0.1819]])

tensor([[0.4479, 0.0409, 0.3620, 0.1004],
        [0.4771, 0.2309, 0.4251, 0.5966],
        [0.5136, 0.6100, 0.8365, 0.1819]])

tensor([0.0409, 0.2309, 0.6100])

tensor([[0.0409],
        [0.2309],
        [0.6100]])

tensor([[0.4771, 0.4251],
        [0.5136, 0.8365]])
```

[8]注意$t[:,1]$和$t[:,1:2]$是不一样的，前者只选择第1列，后者选择第1-2列（不含2）。虽然二者选中的数据看似相同，但后者理论上是多列选择结果，所以它会保持维度不塌缩

tensor可以以文件形式存储和读取[9]

```
import torch
t = torch.rand(2,3)
print(t)
with open('data.bdd', 'wb') as f:
    torch.save(t, f)
#torch.save(t,'data.bdd')
```

```
tensor([[0.2069, 0.7935, 0.9429],
        [0.6891, 0.2898, 0.3404]])
```

```
import torch
with open('data.bdd','rb') as f:
    t = torch.load(f)
# t = torch.load('data.bdd')
# t = torch.load('data.bdd', weights_only=True)
```

```
tensor([[0.2069, 0.7935, 0.9429],
        [0.6891, 0.2898, 0.3404]])
```

[9]采用open()语句打开可以方便的执行更多操作，这里只是演示仅需save()和load()操作就能实现tensor 的序列化和反序列化；在现有pytorch中load()函数的weights_only默认为False，意味着会对文件中的所有数据做反序列化，恶意用户可能利用这一特性载入木马。安全起见可在读取时设定weights_only=True，这样只加载参数部分

可存储为通用格式，在不同训练框架中交换数据。通用数据格式很多，例如numpy数组、HDF5、ONNX、Json、CSV、Protobuf等。这里以HDF5为例[10]。

```
1  import torch
2  import h5py
3
4  t = torch.rand(2,3)
5  print(t)
6  f = h5py.File('hdf5_data.hdf5', 'w')
7  dset = f.create_dataset('coords', data=t.numpy())
8  f.close()
9
10 f2 = h5py.File('hdf5_data.hdf5','r')
11 dset2= f2['coords']
12 t2 = torch.from_numpy(dset2[:])
13 print(t2)
```

```
tensor([[0.0612, 0.8362, 0.2694],
        [0.2829, 0.3603, 0.0363]])

tensor([[0.0612, 0.8362, 0.2694],
        [0.2829, 0.3603, 0.0363]])
```

[10]需要在环境中首先安装h5py；hdf5是一种非常通用的数据存储交换格式，tensorflow、keras等都支持hdf5文件读写

几个常见的关于tensor的操作

1. $torch.tensor(data)$是一个函数，根据传入的$data$构造并返回一个Tensor对象；$torch.Tensor(size)$是直接构成一个类对象，它的值是未定义的（内存中的随机值）

2. $torch.view(size)$可以改变一个tensor的shape，但要求改变前后数据元素数量是一样的，如原本tensor的$size = (3, 4)$，可以改变成$t_new = t.view(2, 3, 2)$，此时t_new的shape为$size = (2, 3, 2)$

3. pytorch中，凡是函数名以下划线“_”结尾的函数都是对原有tensor进行操作（in place），否则是创建一个新的tensor后对新tensor操作。如$t.add_(1)$会使得t的取值全部加1，但是$t.add(1)$不会改变原有t的值，而是返回一个新tensor

几个常见的关于tensor的操作（-续）

4. *tensor.to()*函数在CPU和GPU之间移动tensor，只有在同一个设备上的对象才能做计算[11]

```
1  import torch
2
3  t1 = torch.rand(2,3).to(device='cuda')
4  print(t1)
5  t2 = torch.rand(2,3).to(device='cpu')
6  print(t2)
7  t3 = t1 + t2
```

RuntimeError: Expected all tensors to be on the same device, but found at least two devices, cuda:0 and cpu!

[11]或者用更简洁的方法：$t1.cpu = t1.cup()$；当系统中有多个GPU（多张卡）时还能进一步指定在哪个GPU上，如GPU0；不在同一个GPU上的数据也不能做计算；当有多个GPU时需要手动协调数据再做计算，或者借助并行库完成，如DataParallel或DistributedDataParallel，更多信息请参考CUDA并行/分布式编程

几个常见的关于tensor的操作（-续）

5. 可借助*item*()函数将tensor中的元素转变为python类型，注意，它只能对(1, 1)型的tensor有用

```
1  import torch
2
3  t1 = torch.zeros(2,3)
4  s = t1[0][0].item()
5  print(type(t1[0][0]))
6  print(type(s))
```

```
<class 'torch.Tensor'>
<class 'float'>
```

6. 其它常用操作：tensor.t()（转置）、tensor.abs()、tensor.cos()（逐点类操作，求绝对值、余弦值）、tensor.mean()、tensor.std()、tensor.norm()（归约类操作，求均值、标准差、范数）等

2 用tensor表示数据

2.用tensor表示数据：表格型数据

假设数据存储在CSV文件中[12] （一个CSV文件可看作是一个数据库），这类数据通常是独立同分布的，即表中的每一行代表一个独立的数据。这里以一个公开的葡萄酒质量数据为例进行说明[13]

fixed acidity;"volatile acidity";"citric acid";"residual sugar";"chlorides";"free sulfur dioxide";"total sulfur dioxide";"density";"pH";"sulphates";"alcohol";"quality"
7;0.27;0.36;20.7;0.045;45;170;1.001;3;0.45;8.8;6
6.3;0.3;0.34;1.6;0.049;14;132;0.994;3.3;0.49;9.5;6
8.1;0.28;0.4;6.9;0.05;30;97;0.9951;3.26;0.44;10.1;6
7.2;0.23;0.32;8.5;0.058;47;186;0.9956;3.19;0.4;9.9;6
6.1;0.27;0.43;7.5;0.049;65;243;0.9957;3.12;0.47;9;5
6.9;0.24;0.33;1.7;0.035;47;136;0.99;3.26;0.4;12.6;7
...

[12] comma-separated values，由逗号隔开的数据值，例如{"*x_coord*","*y_coord*","*z_coord*"}有三个数据项，分别由逗号隔开；但这并不意味着数据项只能由逗号隔开，可自由指定分隔符如分号";"

[13] 该数据记录了来自于葡萄牙北部的葡萄酒的各种成份含量及综合评分，访问此处可获得；；此表格的数据项分别表示：固定酸度、挥发性酸度、柠檬酸、残糖、...、质量

2.用tensor表示数据：表格型数据

np.loadtxt()将CSV文件看作一个文本文件读入，跳过一行是为了跳过开始的表头；csv.reader()适合处理数据量有限且结构比较简单CSV文件，它返回一个迭代器，可以逐行读取

```
1  import csv
2  import numpy as np
3
4  file_path = "winequality-white.csv"
5  data_numpy = np.loadtxt(file_path, dtype=np.float32, delimiter=";",
       skiprows=1)
6  print(data_numpy)
7
8  col_list = next(csv.reader(open(file_path), delimiter=";"))
9  print(col_list)
```

```
[[ 7.    0.27  ...  0.45  8.8   6.  ]
 [ 6.3   0.3   ...  0.49  9.5   6.  ]
 [ 8.1   0.28  ...  0.44 10.1   6.  ]
 ...
 [ 6.5   0.24  ...  0.46  9.4   6.  ]
 [ 5.5   0.29  ...  0.38 12.8   7.  ]
 [ 6.    0.21  ...  0.32 11.8   6.  ]]

['fixed acidity',
'volatile acidity',
'citric acid',
'residual sugar',
'chlorides',
'free sulfur dioxide',
'total sulfur dioxide',
'density',
'pH',
'sulphates',
'alcohol',
'quality']
```

2.用tensor表示数据：表格型数据

数据最后一列是综合得分，所以前面的列相当于特征，最后一列相当于标签，在编码时通常将这两部分分开，分别用tensor存储以待备用

```
1  ......
2  data_tensor = torch.from_numpy(data_numpy) # convert initial data to tensor
3  features_tensor = data_tensor[:,:-1]
4  print(features_tensor, features_tensor.shape)
5  target_tensor = data_tensor[:,-1].long()
6  print(target_tensor, target_tensor.shape)
```

```
tensor([[ 7.0000,  0.2700,  0.3600,  ...,  3.0000,  0.4500,  8.8000],
        [ 6.3000,  0.3000,  0.3400,  ...,  3.3000,  0.4900,  9.5000],
        [ 8.1000,  0.2800,  0.4000,  ...,  3.2600,  0.4400, 10.1000],
        ...,
        [ 6.5000,  0.2400,  0.1900,  ...,  2.9900,  0.4600,  9.4000],
        [ 5.5000,  0.2900,  0.3000,  ...,  3.3400,  0.3800, 12.8000],
        [ 6.0000,  0.2100,  0.3800,  ...,  3.2600,  0.3200, 11.8000]]) torch.Size([4898, 11])

tensor([6., 6., 6.,  ..., 6., 7., 6.]) torch.Size([4898])
```

2.用tensor表示数据：表格型数据

数据值可分为三种类型：连续值、顺序值、类别值

- 连续值：连续实值，例如用介于0 ～ 10之间的实数表示评分
- 顺序值：取值具有顺序性，值含义与顺序有关，例如“一颗星、两颗星、三颗星、四颗星、五颗星”
- 类别值：代表不同类别，如“男、女”，“红色、蓝色、绿色”等。

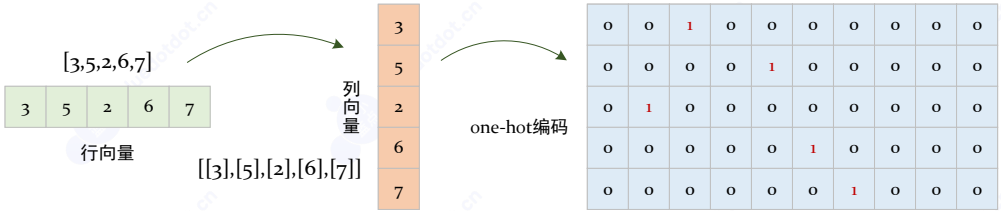
顺序值和类别值的主要区别

- 顺序值有特定的顺序，顺序值之间可作大小比较
- 不同顺序值的顺序差可作为差别大小的度量，例如“不满意”和“超级满意”间的差距明显大于“不满意”和“一般满意”

受教育程度、满意度得分等情况适合用顺序值，颜色、性别、种类等情况适合用类别值。在做统计分析时，顺序值常用中位数等作分析；类别值则常用频数、众数等作分析。

2.用tensor表示数据：表格型数据

顺序值可就用离散值表示，如$\{1, 2, 3, \cdots\}$；类别值虽然也可用离散值表示，但很多时候也用one-hot编码。下面介绍如何将顺序值（离散值）转换为one-hot编码。



2.用tensor表示数据：表格型数据

假设one-hot向量是m维，此处需要三步：1.构造一个$n \times m$型全0张量备用；2.将n型标签向量扩充（或称广播）成$n \times 1$型；3. 根据标签向量设定全0张量中的1位

```
1  ......
2  target_onehot = torch.zeros(target_tensor.shape[0], 10)
3  target_onehot.scatter_(1, target_tensor.unsqueeze(1), 1.0)
4  print(target_onehot)
```

```
tensor([[0., 0., 0., ..., 0., 0., 0.],
        [0., 0., 0., ..., 0., 0., 0.],
        [0., 0., 0., ..., 0., 0., 0.],
        ...,
        [0., 0., 0., ..., 0., 0., 0.],
        [0., 0., 0., ..., 1., 0., 0.],
        [0., 0., 0., ..., 0., 0., 0.]])
```

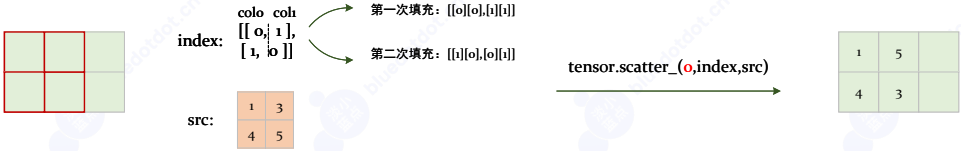
tensor.*unsqueeze*()函数在指定维度上扩展一个额外维度：扩展后tensor的*size*会在指定维度处增加一个1，新增维度的*stride*为0其它维度*stride*不变，任意元素的*storage_offset*不变
若tensor现有$size = (2, 3)$，*tensor*.*unsqueeze*(0)将在第0个维度上扩展一个维度变成$size = (1, 2, 3)$ [14]

tensor.*scatter_*(*dim*, *index*, *src*)表示按指定维度（*dim*）指定位置（*index*）处的值进行改写（*src*） [15]

[14] 若是*tensor*.*unqueeze*(1) 则变成$size = (2, 1, 3)$

[15] $dim = 0$表示*index*指定的是各列的行坐标，$dim = 1$表示*index*指定的是各行的列坐标；注意*index*中的元素必须是*int*64类型，所以*target_tensor*应提前转为*long*型；要求tensor和src具有相同的数据类型，例如同为float或同为int

$dim = 0$时$tensor.unsqueeze()$按指定的行做填充（$index$中的每个元素都代表填充所涉及到的行坐标），元素所在的列则代表了填充所涉及到的列



本例中直接采用顺序值是合适的，可以不转成one-hot编码

读取数据后、开展计算前一般会对数据做预处理，这里用最简单的标准化方法，它将数据项变换成均值为0、方差为1 [16]

$$z = \frac{x - \mu}{\sigma}$$

```
1  ......
2  features_mean = torch.mean(features_tensor, 0)
3  print(features_mean)
4  features_var = torch.var(features_tensor, 0)
5  print(features_var)
6  features_normalized = (features_tensor - features_mean)/torch.sqrt(
       features_var)
7  print(features_normalized)
```

[16]此方法也称为z-score标准化；对数据做预处理几乎已成必备项，它是一种简单但对模型精度又很有帮助的数据处理方法；这里可以直接使用*np.sqrt*(*feature_var*)替代*torch.sqrt*()，但并非所有需要numpy.array的地方都能直接用tensor替换；虽然*torch.sqrt*()和*np.sqrt*()在功能上是类似的，但是*torch.sqrt*()可以在GPU上运行

可以对现有数据按条件进行筛选，例如选择所有得分不大于3分的数据项，或者得分大于3分但小于7分的数据项[17]

```
1  ......
2  bad_data_index = torch.le(target_tensor, 3)
3  bad_data = features_tensor[bad_data_index]
4  print(bad_data.shape)
5  mid_data_index = torch.gt(target_tensor, 3) & torch.lt(target_tensor, 7)
6  mid_data = features_tensor[mid_data_index]
7  print(mid_data.shape)
```

[17] *tensor.le()*、*tensor.gt()*等返回与tensor同型的布尔张量，因此可以做与、或操作，也可以从tensor中提取被选中的元素并返回一个新张量

2.用tensor表示数据：表格型数据

本例中，将数据分成三类：得分较差、得分中等、得分较高。分别求这三类数据各项特征的均值。

```
1  ......
2  bad_data = features_tensor[torch.le(target_tensor,3)]
3  mid_data = features_tensor[target_tensor.gt(3) & target_tensor.lt(7)]
4  good_data = features_tensor[torch.ge(target_tensor, 7)]
5
6  bad_mean = bad_data.mean(0)
7  mid_mean = mid_data.mean(0)
8  good_mean = good_data.mean(0)
9
10 for i, args in enumerate(zip(col_list, bad_mean, mid_mean, good_mean)):
11     print("{:2} {:20} {:6.2f} {:6.2f} {:6.2f}".format(i, *args))
```

从结果中可以看到，劣等、中等、上等酒之间最明显的差别是第6项——总二氧化硫的含量。

0 fixed acidity	7.60	6.89	6.73
1 volatile acidity	0.33	0.28	0.27
2 citric acid	0.34	0.34	0.33
3 residual sugar	6.39	6.71	5.26
4 chlorides	0.05	0.05	0.04
5 free sulfur dioxide	53.33	35.42	34.55
6 total sulfur dioxide	170.60	141.83	125.25
7 density	0.99	0.99	0.99
8 pH	3.19	3.18	3.22
9 sulphates	0.47	0.49	0.50
10 alcohol	10.34	10.26	11.42

2.用tensor表示数据：表格型数据

可以简单的以第6项特征为判定标准，筛选出第6项得分在均值以上的酒及综合得分在5分以上的酒

```
1  ......
2  total_sulfur_threshold = mid_mean[6]
3  total_sulfur_data = features_tensor[:,6]
4  predicted_indexes = total_sulfur_data.lt(total_sulfur_threshold)
5  print(predicted_indexes.shape, predicted_indexes.sum())
6  actual_indexes = target_tensor.gt(5)
7  print(actual_indexes.shape, actual_indexes.sum())
```

```
torch.Size([4898]) tensor(2727)
torch.Size([4898]) tensor(3258)
```

通过精准率和召回率分析此判定方法的可靠程度

```
1  ......
2  correct_matches = torch.sum(predicted_indexes & actual_indexes).item()
3  total_predicted = predicted_indexes.sum().item()
4  total_actual = actual_indexes.sum().item()
5  precision = correct_matches / total_actual
6  recall = correct_matches / total_predicted
7  print(correct_matches, precision, recall)
```

```
2018  0.6193984039287906  0.74000733406674
```

2.用tensor表示数据：序列数据

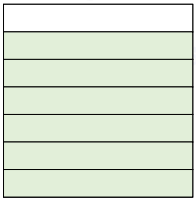
序列数据是指数据项之间存在某些关联性，不再是独立同分步的。这里以2011年-2012年华盛顿自行车租赁数据为例进行说明，数据按小时排列，它的数据关联性主要体现在上一小时和下一小时之间[18]

instant,dteday,season,yr,mnth,hr,holiday,weekday,workingday,weathersit,temp,atemp,hum,windspeed,casual,registered,cnt
1,2011-01-01,1,0,1,0,0,6,0,1,0.24,0.2879,0.81,0,3,13,16
2,2011-01-01,1,0,1,1,0,6,0,1,0.22,0.2727,0.8,0,8,32,40
3,2011-01-01,1,0,1,2,0,6,0,1,0.22,0.2727,0.8,0,5,27,32
...

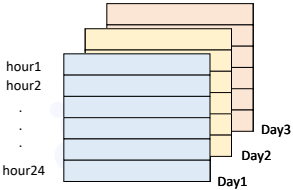
[18]例如，如果上一小时在下雨那下一小时被租赁的可能小就比较小

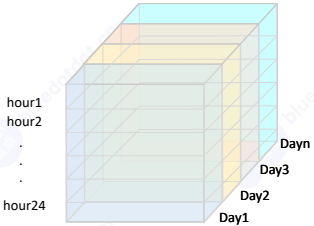
2.用tensor表示数据：序列数据

将全部数据读入，按Day分组，每一天的数据再按hour排列，由此二维表格变成三维张量



全部数据





2.用tensor表示数据：序列数据

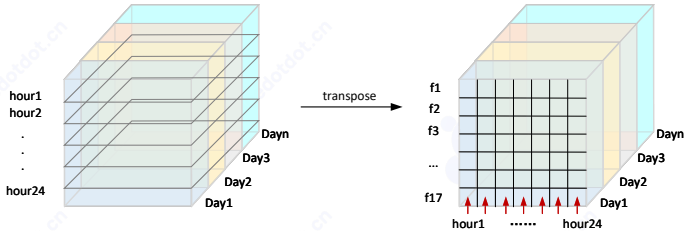
```
1  import torch
2  import csv
3  import numpy as np
4
5  file_path = "bike_sharing_dataset/hour-fixed.csv"
6  bikes_numpy = np.loadtxt(file_path, dtype=np.float32, delimiter=",",
       skiprows=1, converters={1: lambda x: float(x[8:10])})
7  bikes_tensor = torch.from_numpy(bikes_numpy)
8  print(bikes_tensor.shape, bikes_tensor.stride())
9  daily_bikes_tensor = bikes_tensor.view(-1, 24, bikes_tensor.shape[1])
10 print(daily_bikes_tensor.shape, daily_bikes_tensor.stride())
```

```
torch.Size([17520, 17]) (17, 1)
torch.Size([730, 24, 17]) (408, 17, 1)
```

19

[19] converters参数定义一个字典，该字典的键是列的索引，值是一个函数，用于对特定列的数据进行操作

假设一共有N天，每天有L个小时，每个小时都收集C个特征；因为我们想了解每个特征在不同小时内的取值，因此将数据转置成$N \times C \times L$ [20]



[20]实际一共有730天（两年），每天24小时，每个小时统计17个数据

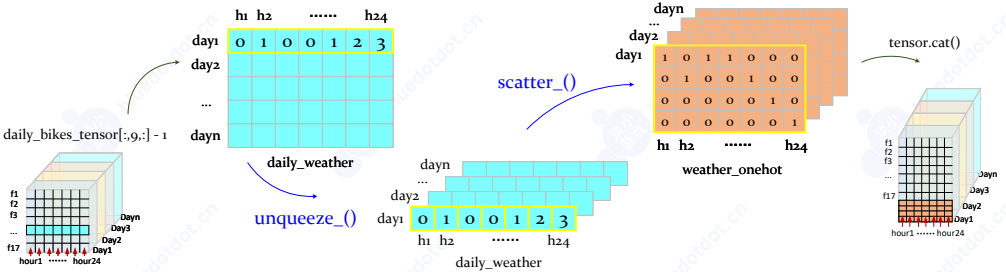
原本天气是类别值，共有四种天气：$\{1,2,3,4\}$分别表示{很好，较好，一般，不太好}，如果要将原本的类别值替换为one-hot编码[21]，可以用如下方式实现[22]

```
1  ......
2  daily_bikes_tensor.transpose_(1,2)
3  weather_onehot = torch.zeros(daily_bikes_tensor.shape[0],4,24)
4  daily_weather = daily_bikes_tensor[:,9,:] - 1
5  daily_weather.unsqueeze_(2)
6  print(daily_weather.shape)
7  weather_onehot.scatter_(1, daily_weather.long(), 1.0)
8  print(weather_onehot.shape)
9  daily_bikes_tensor = torch.cat((daily_bikes_tensor, weather_onehot), 1)
10 print(daily_bikes_tensor.shape)
```

[21]这里当然也可不替换，就用$\{1,2,3,4\}$表示

[22]注意，这里是要将天气特征按列方向扩展成四维one-hot向量，即转变成$(1,0,0,0)^T$而非$(1,0,0,0)$，具体过程请见下一页

转置之后第9行表示某一天24小时内的天气，所以daily_weather是一张表它的每一行表示一天24小时的天气情况。现在要做的是将原本的数值转换为一个四维向量，所以转换后原先的第9行变成了第9~12行，这四行的每一列代表一个小时的天气状况，用one-hot编码[23]



[23]注意现在是要按行填充，而weather_onehot是三维张量，所以这里行相当于它的*shape*[1]，所以*scatter_*()中*dim* = 1

类似的，对部分特征对应做正则化以帮助模型提升预测精度。以第10项特征温度为例，既可以将其转换为[0.0, 1.0]之间，也可以将其转换为均值为0、方差为1。

```
1  ......
2  temp = daily_bikes_tensor[:,10,:]
3  temp_max = torch.max(temp)
4  temp_min = torch.min(temp)
5  daily_bikes_tensor[:,10,:] = (temp - temp_min) / (temp_max - temp_min)
```

```
1  ......
2  temp = daily_bikes_tensor[:,10,:]
3  daily_bikes_tensor[:,10,:] = (temp - torch.mean(temp))/torch.std(temp)
```

这里讨论用最简单的形式将文本数据转换成张量：字符级one-hot及单词级one-hot。这里以纯英文文本[24]为例。

"The Project Gutenberg EBook of Pride and Prejudice, by Jane Austen

This eBook is for the use of anyone anywhere at no cost and with almost no restrictions whatsoever. You may copy it, give it away or re-use it under the terms of the Project Gutenberg License included with this eBook or online at www.gutenberg.org

Title: Pride and Prejudice

Author: Jane Austen ....."

[24] https://github.com/deep-learning-with-pytorch/dlwpt-code/blob/master/data/p1ch4/jane-austen/1342-0.txt

2.用tensor表示数据：文本数据

首先从文本文件中将text读取进来，最开始读取的结果为一个字符串

```
1  import torch
2
3  with open("1342-0.txt", encoding='utf8') as f:
4      text = f.read()
5  print(type(text), len(text))
```

```
<class 'str'> 704190
```

2.用tensor表示数据：文本数据

先讨论字符级one-hot编码。对于纯英文文本可以用ASCII编码，ASCII只能编码128个字符，因而任意一个字符可以用一个长度为128位的向量表示。由于文本太长，这里以文本中的任意一行为例，将该行中的字符转换成128位one-hot向量。

Code	Char	Code	Char	Code	Char	Code	Char	Code	Char	Code	Char
32	[space]	48	0	64	@	80	P	96	`	112	p
33	!	49	1	65	A	81	Q	97	a	113	q
34	"	50	2	66	B	82	R	98	b	114	r
35	#	51	3	67	C	83	S	99	c	115	s
36	$	52	4	68	D	84	T	100	d	116	t
37	%	53	5	69	E	85	U	101	e	117	u
38	&	54	6	70	F	86	V	102	f	118	v
39	'	55	7	71	G	87	W	103	g	119	w
40	(	56	8	72	H	88	X	104	h	120	x
41	)	57	9	73	I	89	Y	105	i	121	y
42	*	58	:	74	J	90	Z	106	j	122	z
43	+	59	;	75	K	91	[	107	k	123	{
44	,	60	<	76	L	92	\	108	l	124	|
45	-	61	=	77	M	93	]	109	m	125	}
46	.	62	>	78	N	94	^	110	n	126	~
47	/	63	?	79	O	95	_	111	o	127	[backspace]

```
1  ......
2  lines= text.split('\n')
3  random_pick = 200
4  line = lines[random_pick]
5  letter_tensor = torch.zeros(len(line), 128)
6  print(line,"\n", letter_tensor.shape)
7
8  for i, letter in enumerate(line.lower().strip()):
9      letter_index = ord(letter) if ord(letter) < 128 else 0
10     letter_tensor[i][letter_index] = 1
```

“Impossible, Mr. Bennet, impossible, when I am not acquainted with him
torch.Size([70, 128])

2.用tensor表示数据：文本数据

再讨论单词级的one-hot编码。因为是按单词做编码，所以首先将text中的空格、换行、标点符号等都去掉；然后将整个text转换成一个个单词。先以某一行为例展示处理效果。

```
1  ......
2  def str_to_clean_word_list(input_str):
3      punctuation = '.,;:"!?_-""'
4      word_list = input_str.lower().replace('\n',' ').split()
5      word_list = [word.strip(punctuation) for word in word_list]
6      return word_list
7
8  words_in_line = str_to_clean_word_list(line)
9  print(line, "\n", words_in_line)
```

“Impossible, Mr. Bennet, impossible, when I am not acquainted with him
['impossible', 'mr', 'bennet', 'impossible', 'when', 'i', 'am', 'not', 'acquainted', 'with', 'him']

2.用tensor表示数据：文本数据

将text的全部内容提取成单词列表，然后去掉重复的单词并排序，这样就得到了一个由单词组成的code-book。利用每个单词在code-book中的编号，就可以将单词转换成one-hot向量，向量维度为code-book中的单词总数。

```
1  ......
2  text_words = str_to_clean_word_list(text)
3  words_list = sorted(set(text_words))
4  word2index_dict = {word:i for (i, word) in enumerate(words_list)}
5  word_tensor = torch.zeros(len(text_words), len(word2index_dict))
6  for i, word in enumerate(text_words):
7      word_index = word2index_dict[word]
8      word_tensor[i][word_index]=1
9      print('{:2} {:4} {}'.format(i, word_index, word))
```

2.用tensor表示数据：文本数据

由于总文本太长，还是以其中的某一行为例展示编码后的效果。

```
1   ......
2   print(len(word2index_dict))
3   line_words = str_to_clean_word_list(line)
4   line_tensor = torch.zeros(len(line_words),
5                             len(word2index_dict))
6   for i, word in enumerate(line_words):
7       word_index = word2index_dict[word]
8       line_tensor[i][word_index] = 1
9       print('{:2} {:4} {}'.format(1, word_index, word))
10  print(line_tensor.shape)
```

```
7261
“Impossible, Mr. Bennet, impossible, when I
am not acquainted with him
 0 3394 impossible
 1 4305 mr
 2  813 bennet
 3 3394 impossible
 4 7078 when
 5 3315 i
 6  415 am
 7 4436 not
 8  239 acquainted
 9 7148 with
10 3215 him
torch.Size([11, 7261])
```

2.用tensor表示数据：图像

图像本身就是高度结构化的数据，跟张量十分类似，只需要将其读进来再做简单的处理即可（pytorch中有一个转门的torchvision.transform.v2类，里面提供了大量对图像的操作）。是否做transpose根据网络结构定，有的训练框架是按$N \times C \times H \times W$定义网络的，有的是按$N \times H \times W \times C$定义的。

```
1  import torch
2  import imageio.v3 as iio
3
4  img_arr = iio.imread("bobby.jpg")
5  print(type(img_arr), img_arr.shape)
6
7  img_tensor = torch.from_numpy(img_arr)
8  img_tensor.transpose_(0, 2).transpose_(1,2)
9  print(img_tensor.shape)
```

```
<class 'numpy.ndarray'> (720, 1280, 3)
torch.Size([3, 720, 1280])
```

也可以批量读取数据，只需要在循环中不断加载图像即可。做正则化和前面的类似，只不过这里是按channel做batch normalization。

```
1  ......
2  filenames = [name for name in os.listdir("image_cats") if os.path.splitext(
       name)[1] == '.png']
3  batch_tensor = torch.zeros(len(filenames), 3, 256, 256, dtype=torch.uint8)
4  for i, filename in enumerate(filenames):
5      img_arr = iio.imread(os.path.join("image_cats", filename))
6      batch_tensor[i] = torch.from_numpy(img_arr).transpose_(0,2)
7  print(batch_tensor.shape)
8
9  batch_tensor = batch_tensor.float()
10 batch_tensor /= 255.0
11 for c in range(batch_tensor.shape[1]): #channels
12     mean = torch.mean(batch_tensor[:,c])
13     std = torch.std(batch_tensor[:,c])
14     batch_tensor[:,c] = (batch_tensor[:,c] - mean)/std
```

3 模型的训练

3.模型的训练：手动计算梯度

假设我们从外星球得到一种新型的、未知的温度计，它能显示当前温度的数值，但这个数值对应多少摄氏度并不明确。我们还有一个标准的摄氏度温度计。为了弄清楚新型温度计和标准温度之间的对应关系，我们记录了一系列温度数据并建立了一个最基本的线性模型，来模拟二者之间的映射关系。现在利用已有数据寻找最佳的$\{w, b\}$取值。这里采用二次损失函数。

新型温度计

t_u = [35.7, 55.9, 58.2, 81.9, 56.3, 48.9, ...]

[0.5, 14.0, 15.0, 28.0, 11.0, 8.0, 3.0, ...] = t_c

$t\_p = w * t\_u + b$

$Loss = (t\_p - t\_c)^2$

标准温度计

3.模型的训练：手动计算梯度

```
1  import torch
2
3  def model(t_u, w, b):
4      return w*t_u + b
5  def loss_fn(t_p, t_c):
6      squared_diffs = (t_p - t_c)**2
7      return squared_diffs.mean()
8
9  w = torch.ones(1)
10 b = torch.zeros(1)
11 t_c = torch.tensor([0.5, 14.0, 15.0, 28.0, 11.0, 8.0, 3.0, -4.0, 6.0, 13.0,
       21.0])
12 t_u = torch.tensor([35.7, 55.9, 58.2, 81.9, 56.3, 48.9, 33.9, 21.8, 48.4,
      60.4, 68.4])
13 t_p = model(t_u, w, b)
14 loss = loss_fn(t_p, t_c)
15 print(t_p)
16 print(loss)
```

```
tensor([35.7000, 55.9000, 58.2000, 81.9000, 56.3000,
48.9000, 33.9000, 21.8000, 48.4000, 60.4000, 68.4000])
tensor(1763.8848)
```

简单说明为什么参数应按其导数的相反方向增长

- 若$loss_rate_of_change_w > 0$说明$w$取值增加后损失反而变大，此时应减小$w$
- 若$loss_rate_of_change_w < 0$说明$w$取值增加后损失也减小，此时应继续增大$w$

可见，w的增长方向应与损失关于w变化率的方向相反，也就是沿w的梯度的反方向，所以有

$$w = w - learning\_rate * loss\_rate\_of\_change\_w$$

```
1  delta = 0.1
2
3  loss_rate_of_change_w = (loss_fn(model(t_u, w + delta, b), t_c)
4                  - loss_fn(model(t_u, w, b), t_c)) / delta
```

后面我们用$dloss_dw$和$dloss_db$分别表示损失函数关于w, b的梯度

因为已知有

$$Loss = (t\_p - t\_c)^2, t\_p = model(t\_u, w, b) \rightarrow Loss = (model(t\_u, w, b) - t\_c)^2$$

所以根据链式求导法则（chain-rule）有

$$\frac{\partial Loss}{\partial w} = \frac{\partial Loss}{\partial model}\frac{\partial model}{\partial w}, \quad \frac{\partial Loss}{\partial b} = \frac{\partial Loss}{\partial model}\frac{\partial model}{\partial b},$$

并且

$$\frac{\partial Loss}{\partial model} = 2(model(t\_u, w, b) - t\_c), \quad \frac{\partial model}{\partial w} = t_u, \quad \frac{\partial model}{\partial b} = 1$$

3.模型的训练：手动计算梯度

```
def dloss_fn(t_p, t_c):
    dsq_diffs = (t_p - t_c)*2
    return dsq_diffs

def dmodel_dw(t_u, w, b):
    return t_u

def dmodel_db(t_u, w, b):
    return 1.0

def grad_fn(t_u, t_c, t_p, w, b):
    dloss_dw = dloss_fn(t_p, t_c)*dmodel_dw(t_u, w, b)
    dloss_db = dloss_fn(t_p, t_c)*dmodel_db(t_u, w, b)
    return torch.stack([dloss_dw.mean(), dloss_db.mean()])

grad = grad_fn(t_u, t_c, t_p, w, b)
```

3.模型的训练：手动计算梯度

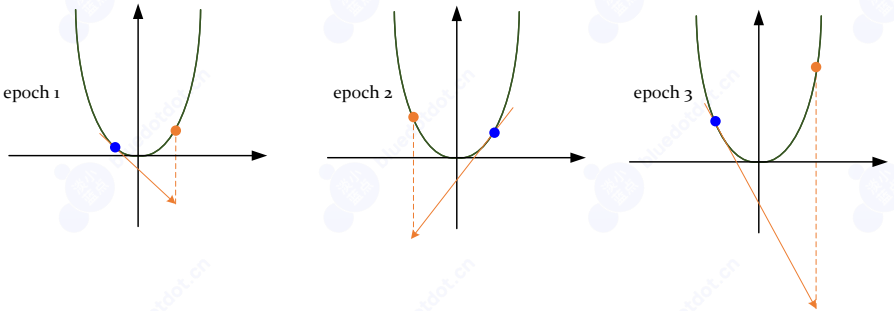
把用完全部数据作更新称作一个epoch，通常数据仅用一次是不够的，所说我们需要多个epoch，这意味着我们要把更新过程放到一个循环中。在循环中不断更新参数、不断计算损失和梯度，直到训练完毕。

```
1  def training_loop(n_epochs, learning_rate, params, t_u, t_c):
2      for epoch in range(1, n_epochs + 1):
3          w, b = params
4          t_p = model(t_u, w, b)
5          loss = loss_fn(t_p, t_c)
6          grad = grad_fn(t_u, t_c, t_p, w, b)
7          params = params - learning_rate * grad
8          print("Epoch %d, Loss %f" % (epoch, float(loss)))
9          print("Params:{}".format(params))
10         print("Grad:{}\n".format(grad))
11     return params
12
13 training_loop(100, 1e-2, torch.tensor([1.0, 0.0]), t_u, t_c)
```

虽然理论上一切都很完美，但在实际上我们会发现，每迭代一次损失不仅没有减少反而快速增加，第11次时就变得非常非常大，以至于计算机都无法正常显示。

```
.....
Epoch 9, Loss 271768651958811160221295847669760.000000
Params:tensor([-5.5580e+15, -9.7903e+13])
Grad:tensor([5.6541e+17, 9.9596e+15])

Epoch 10, Loss 909010754784581309611713619778600960.000000
Params:tensor([3.2144e+17, 5.6621e+15])
Grad:tensor([-3.2700e+19, -5.7600e+17])

Epoch 11, Loss inf
Params:tensor([-1.8590e+19, -3.2746e+17])
Grad:tensor([1.8912e+21, 3.3313e+19])
```

这是由于我们设定的学习率过大导致的，只需要调整学习率就能改善这种情况



3.模型的训练：手动计算梯度

将学习率从$1e-2$调整为$1e-4$[25]

```
1  ......
2  training_loop(100, 1e-4, torch.tensor([1.0, 0.0]), t_u, t_c)
```

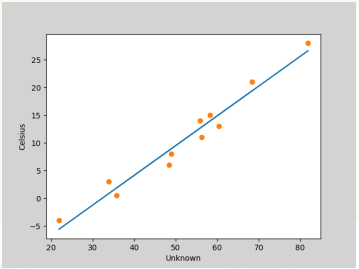
```
Epoch 1, Loss 1763.884766
Params:tensor([ 0.5483, -0.0083])
Grad:tensor([4517.2964,  82.6000])
......

Epoch 99, Loss 29.023582
Params:tensor([ 0.2327, -0.0435])
Grad:tensor([-0.0533,  3.0226])

Epoch 100, Loss 29.022667
Params:tensor([ 0.2327, -0.0438])
Grad:tensor([-0.0532,  3.0226])
```

[25] 当然，还可以将数据正则化之后再计算

3.模型的训练：手动计算梯度

通常最后可以将训练结果用图形化的方式展现出来。这里展现的是用正则化后的数据训练得到的模型。

```
1   ......
2   t_un = (t_u - t_u.mean())/t_u.std()
3   param = training_loop(1000, 1e-2, torch.tensor([1.0, 0.0]), t_un, t_c)
4   t_p = model(t_un, *param)
5
6   fig = plt.figure(dpi=300)
7   plt.xlabel("Unknown")
8   plt.ylabel("Celsius")
9   plt.plot(t_u.numpy(), t_p.detach().numpy())
10  plt.plot(t_u.numpy(), t_c.numpy(), 'o')
11  plt.show()
```



3.模型的训练：自动计算梯度

前面的过程中，我们是全完手动计算梯度并更新参数，这样做只是为了展现整个过程的原理。想像一下，对于一个庞大的、复杂的神经网络，我们是不可能自己去编程计算每一个参数的梯度的，训练框架都会提供自动求导功能（Autograd），这也是训练框架存在的主要意义之一。

一种最简单的方法就是为参数的张量指定$requires_grad = True$，然后调用$loss.backward()$。前者告诉pytorch要记录此张量的梯度，后者则是自动计算所有$requires_grad = True$的张量的梯度[26]。

```
1  ......
2  params = torch.tensor([1.0, 0.0], requires_grad=True)
3  loss = loss_fn(model(t_u, *params), t_c)
4  loss.backward()
5  print(params.grad)
```

```
tensor([4517.2969,  82.6000])
```

[26] pytorch中tensor默认设置$requires_grad = False$

重构*training_loop*()函数如下，有几个注意点

- pytorch中张量的梯度是累积式而非存储式，为了避免重复计算在每个epoch时都要首先清空参数的梯度
- *loss*.*backward*()只计算各个参数的梯度，而不会自动利用梯度更新参数的值
- 只要loss是*requires_grad* = *True*的张量，就能调用*backward*()函数发起自动求梯度操作[27]

[27]能否发起*backward*()的关键在于*loss*对象本身，pytorch中如果一个张量设置*requires_grad* = *True*，那么所以由它计算得到的下游张量其*requires_grad* = *True*

3.模型的训练：自动计算梯度

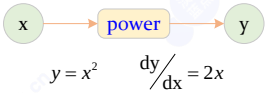
```
1  ......
2  def training_loop(n_epochs, learning_rate, params, t_u, t_c):
3      for epoch in range(1, n_epochs + 1):
4          if params.grad is not None:
5              params.grad.zero_()
6
7          t_p = model(t_u, *params)
8          loss = loss_fn(t_p, t_c)
9          loss.backward()
10         params = (params-learning_rate*params.grad).detach().requires_grad_()
11         if epoch % 500 == 0:
12             print('Epoch %d, Loss %f' % (epoch, float(loss)))
13     return params
14
15 t_un = (t_u - t_u.mean())/t_u.std()
16 training_loop(5000,1e-2,torch.tensor([1.0,0.0],requires_grad=True),t_un,t_c)
```

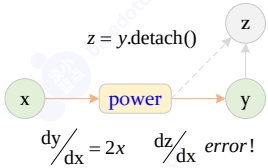
```
......
Epoch 4000, Loss 2.927645
Epoch 4500, Loss 2.927645
Epoch 5000, Loss 2.927645
```

3.模型的训练：张量分离

在模型的设计与训练中，有可能需要对模型的中间结果进行额外操作，但又不希望这些操作影响原本参数的梯度计算，因此可利用*detach*()函数从这些张量中分离出新的张量再做计算

- 分离张量与原张量共享同一片内存
- 分离张量本身不能计算梯度，但不影响原张量及其它张量计算梯度





3.模型的训练：张量分离

```python
import torch

x = torch.tensor([1,2,3])
y = x.detach()
print(id(x.untyped_storage()))
print(id(y.untyped_storage()))

y[0] = 4
print(x)
```

```
1996156645488
1996156645488
tensor([4, 2, 3])
```

因为$y=(x_1^2,x_2^2,x_3^2)^T$且$y.sum()=x_1^2+x_2^2+x_3^2$，所以利用$backward()$求梯度时得到的是一个向量[28]

$$\frac{dy}{d\mathbf{x}}=\begin{pmatrix}\frac{\partial y}{\partial x_1}\\ \frac{\partial y}{\partial x_2}\\ \frac{\partial y}{\partial x_3}\end{pmatrix}=\begin{pmatrix}2x_1\\ 2x_2\\ 2x_3\end{pmatrix}$$

```python
import torch

x = torch.tensor([1,2,3.5], requires_grad=True)
y = x ** 2
z = y.detach()
y.sum().backward() # y = x1^2 + x2^2 + x3^3
print(x.grad)
#z.sum().backward() # error!
```

```
tensor([2., 4., 7.])
```

[28]向量、矩阵求导请参见《面向机器学习（深度学习、人工智能）的数学基础》

$\mathbf{t} = \mathbf{z} + \mathbf{x}^3$，因为$\mathbf{z}$是分离变量它无法求梯度，但不影响$t$求梯度，因此$\frac{\mathbf{dt}}{\mathbf{dx}} = 3\mathbf{x}^2$

```python
import torch

x = torch.tensor([1,2,3.5], requires_grad=True)
y = x ** 2
z = y.detach()

t = z + x**3 # t = (x1^2,x2^2,x3^2) + (x1^3,x2^3,x3^3)
t.sum().backward() # t.sum() = x1^2 + x2^2 + x3^2 + x1^3 + x2^3 + x3^3
print(x.grad)
```

```
tensor([ 3.0000, 12.0000, 36.7500])
```

3.模型的训练：参数自动更新

optimizer能实现基于梯度的参数自动更新。pytorch中预置了众多不同的学习算法，如：Adam、SGD、RMSprop等。

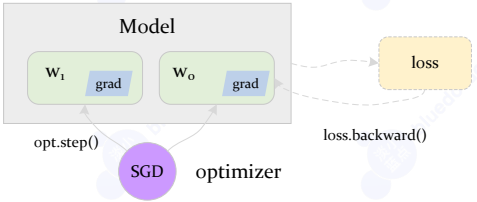
定义optimizer对象时传递两组参数

- *parmas*：需要用优化器更新的参数，通常是tensor的可迭代对象或指定tensor的字典
- $Dict[str, Any]$：指定优化器的超参数，如学习率、动量等

```
1  import torch.optim as optim
2  ...
3  optimizer = optim.SGD([params], lr=0.01)
```

optimizer实现对参数及其梯度的管理[29]，有最重要的两个操作

- *optimizer.zero_grad()*：清空参数的梯度
- *optimizer.step()*：利用梯度更新所有参数



[29]梯度本身存储在参数内而非优化器内

3.模型的训练：参数自动更新

调用参数自动更新时有以几点注意

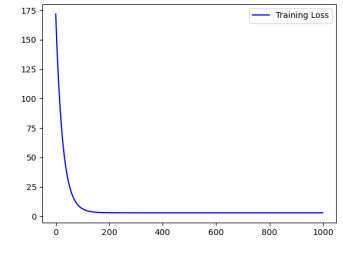
- pytorch中只有float型tensor才能计算张量
- 每个epoch内应首先调用$opt.zero_grad()$清空梯度，否则梯度会累加，给计算带来误差
- 应提前设置参数属性$params.requires_grad = True$
- 只有tensor对象才能发起$backward()$，应注意loss对象的类型[30]

[30]若loss是float或其它非tensor类型，调用$backward()$会报错

3.模型的训练：参数自动更新

直接调用下列函数完成训练，包括梯度自动计算、参数自动更新

```python
def training_loop(n_epochs, learning_rate, params, t_u, t_c):
    w, b = params
    w.requires_grad = True
    b.requires_grad = True
    opt = optim.SGD([w,b], lr=learning_rate)
    losses = []
    for epoch in range(1, n_epochs + 1):
        opt.zero_grad()
        t_p = model(t_u, w, b)
        loss = loss_fn(t_p, t_c)
        losses.append(loss.item())
        loss.backward()
        opt.step()
        #print training info here...
    return params, losses
```



3.模型的训练：*backward*()补充说明

通常情况下最终的*loss*值是一个标量，因此pytorch设计只能由标量发起*backward*()调用。如果当前*loss*不是标量，调用*backward*()会报错

```
1 import torch
2 x = torch.tensor([1.,2.], requires_grad=True)
3 y = x ** 2
4 y.backward()  # y is tensor, y.size()=([2])
5 print(x.grad)
```

```
raise RuntimeError(
RuntimeError: grad can be implicitly created only for scalar outputs
```

```
1 import torch
2 x = torch.tensor([1.,2.], requires_grad=True)
3 y = x ** 2
4 y.sum().backward()  # y.sum() reduces to a scalar
5 print(x.grad)
```

```
tensor([2., 4.])
```

3.模型的训练：*backward*()补充说明

如果**y**是多维的，pytorch设计中认为它是中间变量（例如隐层参数），它的梯度由下游反向传递而来，因此多维量发起*backward*()函数时应传入下游反向传递而来的梯度，其维度等于**y**的维度

```
1  import torch
2  x = torch.tensor([1.,2.], requires_grad=True)
3  y = x ** 2
4  y.backward(torch.tensor([1.,2.]))
5  print(x.grad)
```

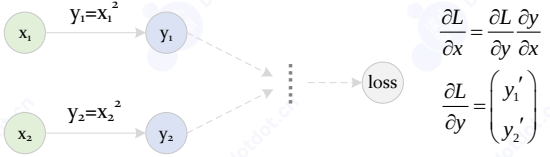
```
tensor([2., 8.])
```

前面代码中向$\mathbf{y}$.*backward*()传入的*torch.tensor*([1., 2.])相当于这里的$\frac{\partial L}{\partial \mathbf{y}} = \begin{pmatrix} y_1' \\ y_2' \end{pmatrix}$，根据BP算法有[31]

$$\frac{\partial L}{\partial x_1} = \frac{\partial L}{\partial y_1}\frac{\partial y_1}{\partial x_1} + \frac{\partial L}{\partial y_2}\frac{\partial y_2}{\partial x_1}$$

因为根据定义$\mathbf{y} = \begin{pmatrix} x_1^2 \\ x_2^2 \end{pmatrix}$，也就是$y_1$和$x_2$无关，$y_2$和$x_1$无关，所以最终有

$$\frac{\partial L}{\partial x_1} = \frac{\partial L}{\partial y_1}\frac{\partial y_1}{\partial x_1} = 1 * 2x_1 = 2$$
$$\frac{\partial L}{\partial x_2} = \frac{\partial L}{\partial y_2}\frac{\partial y_2}{\partial x_2} = 2 * 2x_2 = 8$$



[31]有关BP算法介绍请参考PRML第五章或PRML Page-by-page note的5-027

4 神经网络模型及训练

pytorch中从加载数据到构建神经网络模型再到训练最重要的是四个类：torch.nn、torch.optim、Dataset、DataLoader。

torch.nn本身是一个命名空间（并非一个具体的类），几乎所有与构建神经网络模型相关的类和函数都在这个命名空间下。torch.Module是torch.nn下一个最基本、最重要的基类

- 所有的网络模型或网络层都是其子类，如nn.RNN、nn.Linear、nn.Conv2d等
- 模型容器是其子类，如nn.Sequential、nn.ModuleList、nn.ModuleDict等
- 激活函数是其子类，如ReLu、Sigmoid、Tanh等

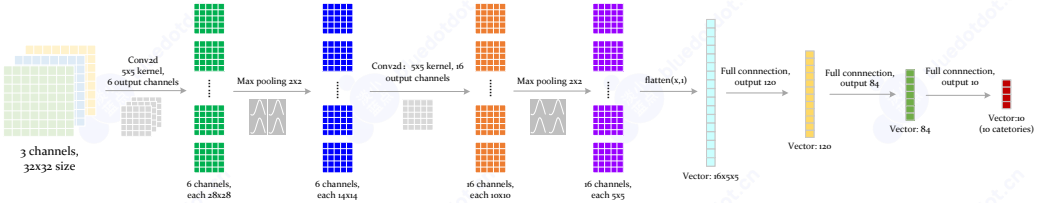
pytorch中NN模型是继承自nn.Module的子类，它至少有两个函数要实现：*__init__*()和*forward*()

- *__init__*()：构造函数，一般首先调用父类的构造函数（*super*().*__init__*()），再声名构成网络所需各模块
- *forward*()：定义模型前向计算过程，将模型实例作为可运行对象调用时会自动调用该函数以完成前向计算[32]

在pytorch中利用已有类构造模型时，模型参数默认设置为$requires_grad = True$

[32] nn.Module父类中定义了*__call__*()函数，因此nn.Module的实例可以像函数一样被调用，如$m(x)$（m是nn.Module的实例）。调用时首先运行*__call__*()函数，*__call__*()中则会调用*forward*()函数

以CIFAR10数据为输入，构造并训练一个网络实现对图像的分类。CIFAR10数据集中每个图像是3 × 32 × 32型数据（channel、Height、Width），总共被标记为十个类别：airplane、automobile、bird、cat、deer、dog、frog、horse、ship、truck。按如下结构构造一个神经网络（卷积层后接有激活函数，图中省略）。



4.神经网络模型及训练：网络结构

```python
import torch.nn as nn
import torch.nn.functional as F
class Net(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(3,6,5)
        self.pool = nn.MaxPool2d(2,2)
        self.conv2 = nn.Conv2d(6,16,5)
        self.fc1 = nn.Linear(16*5*5, 120)
        self.fc2 = nn.Linear(120,84)
        self.fc3 = nn.Linear(84,10)
    def forward(self, x):
        x = self.pool(F.relu(self.conv1(x)))
        x = self.pool(F.relu(self.conv2(x)))
        x = torch.flatten(x, 1)
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x
```

假设输入数据形状为$(B, C_{in}, H_{in}, W_{in})$（分别代表Batch_size, Channels, Height, Width），当前指定的卷积核大小为$kernel_size = [k1, k2]$，衬垫填充大小为$padding = [p1, p2]$，前进步伐为$stride = [s1, s2]$，卷积后数据维度为$(N, C_{out}, H_{out}, W_{out})$

- C_{out}由用户在定义卷积层时指定
- $H_{out} = \frac{H_{in} - kernel\_size[0] + 2 * padding[0]}{stride[0]} + 1$
- $W_{out} = \frac{W_{in} - kernel\_size[1] + 2 * padding[1]}{stride[1]} + 1$

假设输入数据为32×32，若$kernel = 5$（$padding = 0, stride = 1$取默认值），经卷积后有

$$H_{out} = \frac{32 - 5 + 0}{1} + 1 = 27 + 1 = 28, \quad W_{out} = \frac{32 - 5 + 0}{1} + 1 = 27 + 1 = 28$$

若输入数据为14×14且$kernel = 5$（$padding = 0, stride = 1$取默认值），经卷积后有

$$H_{out} = \frac{14 - 5 + 0}{1} + 1 = 10, \quad W_{out} = \frac{14 - 5 + 0}{1} + 1 = 10$$

有一个专门用于视觉的库torchvision[33]，里面提供了大量用于图像的操作及数据集，其中transforms就是专门用于对图像做变形、处理的。我们利用torchvision获取CIFAR10数据并利用torchvision.transform对数据做预处理。处理包括两步：首先将读取的图像文件转换成tensor，然后将tensor中的数据取值范围规范化到$[-1,1]$之间且标准差为0.5（CIFAR10中图像数据默认取值范围为）[34]

$$x' = \frac{x-\mu}{\sigma} = \frac{x-0.5}{0.5}$$

将像素值映射到$[-1,1]$的范围内是为了后面网络的计算，这是一种在深度学习中非常常见的预处理步骤

- 很多激活函数的有效值范围在$[-1,1]$内，例如Sigmoid函数、tanh函数等
- 与特定的参数初始化方法配合如Xavier初始化、He初始化等，能让（预训练）模型训练更容易
- 增强计算稳定性、避免上下溢出等

[33]也是由Meta公司开发的，利用pytorch官方命令安装pytorch时会一并安装torchvision、torchaudio

[34]默认情况下CIFAR10图像的像素值取值在$[0,1]$范围内，$(x-0.5)\in[-0.5,0.5]$，$(x-0.5)/0.5\in[0,1]$（除以0.5相当于乘以2，乘以2是线性映射）；一般做规范化时，应首先从数据集中计算得到均值和方差（标准差），再做规范化，这里为什么直接令均值为0.5、标准差为0.5？这是一种简单的、经验性做法

4.神经网络模型及训练：数据加载

```
import torchvision
import torchvision.transforms as transforms

transform = transforms.Compose([transforms.ToTensor(),
     transforms.Normalize((0.5,0.5,0.5),(0.5,0.5,0.5))])
batch_size = 4

trainset = torchvision.datasets.CIFAR10(root='./data', train=True,
    download=True, transform=transform)
trainloader = torch.utils.data.DataLoader(trainset, batch_size=
    batch_size, shuffle=True, num_workers=0)

testset = torchvision.datasets.CIFAR10(root='./data', train=False,
    download=True, transform=transform)
testloader = torch.utils.data.DataLoader(testset, batch_size=batch_size,
     shuffle=False, num_workers=0)

classes = ('plane', 'car', 'bird', 'cat', 'deer', 'dog', 'frog', 'horse'
    , 'ship', 'truck')
```

Dataset和DataLoader是pytorch中常用的两个帮助加载数据的工具，Dataset用于加载数据，DataLoader则将Dataset包装成可迭代对象[35]，所以想要利用*next*()访问数据需要将dataloader进一步转换成迭代器

```
import matplotlib.pyplot as plt
import numpy as np
def imshow(img):
    img = img/2 + 0.5
    npimg = img.numpy()
    plt.imshow(np.transpose(npimg, (1,2,0)))
    plt.show()
dataiter = iter(trainloader)
images, labels = next(dataiter)
imshow(torchvision.utils.make_grid(images, nrow=2))
print(' '.join(f'{classes[labels[j]]:5s}' for j in range(batch_size)))
```

[35]分别从torch.utils.data.dataset和torch.utils.data.dataloader引用；可迭代对象和迭代器是两个紧密相关但不完全一样的概念，可迭代对象和迭代器都实现了*__iter__*()函数，但迭代器进一步实现了*__next__*()函数；简单说可迭代对象只能在循环语句中访问包含的对象，而迭代器不仅可在循环语句中访问包含对象，还能通过*next*()函数访问下一个对象；在《淡蓝小点直播系列：DDPM原理推导及代码实现》中我们曾自己实现过一个Dataset，相关代码可点击此处访问

结合前面定义的网络模型，将所有的数据放到同一个设备上[36]，再按之前介绍的方法定义损失函数和优化器

```
net = Net()
device = torch.device('cuda:0' if torch.cuda.is_available() else 'cpu')
net = net.to(device)

import torch.optim as optim

criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(net.parameters(), lr=0.001, momentum=0.9)
```

[36]都放到CPU上或都放到GPU上，否则无法计算；对于多GPU、分布式GPU此处不做介绍

类似于前文所述，构造一个训练过程。在训练过程中记得每次循环前清除上一次计算得到的梯度，调用*loss.backward*()函数自动计算梯度，调用*optim.step*()函数自动更新参数

```
for epoch in range(2):
    running_loss = 0.0
    for i, data in enumerate(trainloader, 0):
        inputs, labels = data
        inputs = inputs.to(device)
        labels = labels.to(device)
        optimizer.zero_grad()
        outputs = net(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()
        running_loss += loss.item()
        if i % 2000 == 1999:
            print(f'[{epoch+1}, {i+1:5d} loss:{running_loss / 2000: .3f}]')
            running_loss = 0.0
```

将训练得到的模型序列化保存，然后还能再用对应函数反序列化加载到内存中[37]

- *model.state_dict()*：返回模型中所有可学习参数的字典
- *load_state_dict()*：将以字典形式存储的参数读入到模型中，设置*weights_only = True*更加安全

```
print('Finished Training')
PATH = './cifar_net.pth'
torch.save(net.state_dict(), PATH)

net = Net()
net.load_state_dict(torch.load(PATH, weights_only=True))
```

[37]每个参数都有名称，名称可以由用户指定，未指定时默认情况下就是用户定义的nn.Module的实例名；如本例中就包括conv1.weight、conv1.bias等；*weights_only*标置位的作用在第一部分曾介绍过

从测试数据中读取第一个Batch的4张图，输出图像和模型预测结果

```
dataiter = iter(testloader)
images, labels = next(dataiter)

imshow(torchvision.utils.make_grid(images))
print("GroundTruth:", "".join(f'{classes[labels[j]]:5s}' for j in range(4)))

outputs = net(images)
_, predicted = torch.max(outputs, 1)
print("Predicted:", "".join(f'{classes[predicted[j]]:5s}' for j in range(4)))
```

我们将测试数据全部导入模型中，用模型做预测。由于模型比较简单，在测试数据上的表现不算特别优秀，但比随机猜测仍要强不少[38]

```
correct = 0
total = 0

with torch.no_grad():
    for data in testloader:
        images, labels = data
        outputs = net(images)
        _, predicted = torch.max(outputs.data, 1)
        total += labels.size(0)
        correct += (predicted == labels).sum().item()
print(f"Accuracy of the network on 10000 test imags: {100* correct//total}%")
```

```
Accuracy of the network on
10000 test imags: 55%
```

[38]因为有10个类别，所以随机猜测的准确性应在10%左右

进一步更细致的查看在不同类别数据上，模型的表现结果如何[39]

```
correct_pred = {classname: 0 for classname in classes}
total_pred = {classname:0 for classname in classes}
with torch.no_grad():
    for data in testloader:
        images, labels = data
        outputs = net(images)
        _, predictions = torch.max(outputs, 1)
        for label, predictions in zip(labels, predictions):
            if label == predictions:
                correct_pred[classes[label]] += 1
            total_pred[classes[label]] += 1
for classname, correct_count in correct_pred.items():
    accuracy = 100 * float(correct_count) / total_pred[classname]
    print(f"Accuracy for class: {classname:5s} is {accuracy:.1f}%")
```

```
Accuracy for class: plane is 56.7%
Accuracy for class: car   is 68.5%
Accuracy for class: bird  is 29.1%
Accuracy for class: cat   is 22.7%
Accuracy for class: deer  is 41.9%
Accuracy for class: dog   is 62.1%
Accuracy for class: frog  is 77.3%
Accuracy for class: horse is 56.5%
Accuracy for class: ship  is 69.2%
Accuracy for class: truck is 73.6%
```

完结

Congrats | 淡小蓝点 bluedotdot.cn

微信号：bluedotdot_cn

More：《PRML Page-by-page》、《学习理论精炼介绍》、《面向机器学习（深度学习、人工智能）的数学基础》、《pytorch编程快速学习》、《DDPM原理推导及代码实现》、《OpenAI编程基础》等