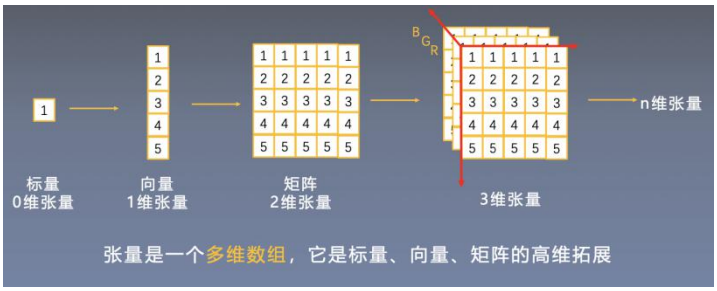
第一讲：张量

一、认识张量

（一）张量的介绍

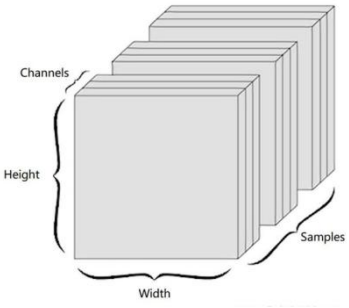
张量（也可以叫做 Tensors）是 pytorch 中数据存储和表示的一个基本数据结构和形式，它是一个多维数组，是标量、向量、矩阵的高维拓展。它相当于 Numpy 的多维数组(ndarrays)，但是 tensor 可以应用到 GPU 上加快计算速度，并且能够存储数据的梯度信息。



维度大于 2 的一般称为高维张量。以计算机中的图像数据存储为例。了解下**高维张量的应用**

3 维张量，可以表示图像的：通道数 x 高 x 宽

4 维张量，通常表示：样本数 x 通道数 x 高 x 宽



（二）基于 torch.tensor()创建张量

1、torch.tensor() 基于数据创建

```
torch.tensor( data,
              dtype=None,
              device=None,
              requires_grad=False,
              pin_memory=False)
```

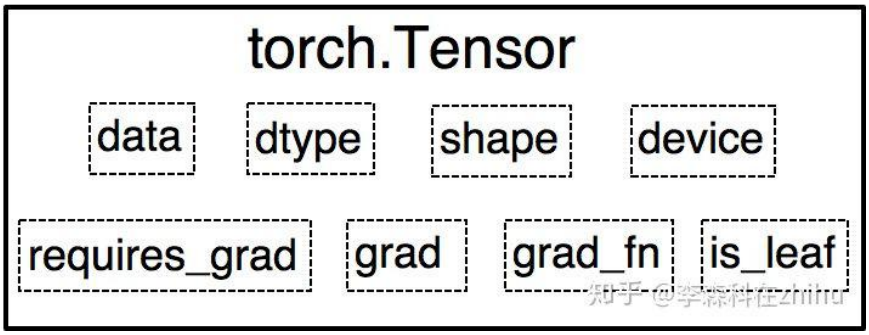
表 1 常用参数说明

参数名称	说明
data	数据，可以是 list，ndarray
dtype	数据类型，默认与 data 的一致
device	所在设备，cuda/cpu。Cuda 表示是 gpu
requires_grad	是否需要梯度

分别传入一个 list 和 numpy 创建 tensor

```
import torch
import numpy as np

t1 = torch.tensor([5.5, 3])
print(t1)
# tensor([5.5000, 3.0000])


arr = np.arange(1, 9).reshape(2, 4)
print("ndarray的数据类型：", arr.dtype)
t2 = torch.tensor(arr)
# t2 = torch.tensor(arr, device='cuda')
print(t2)

# ndarray的数据类型： int64
# tensor([[1, 2, 3, 4],
#         [5, 6, 7, 8]])
```

（三）张量的属性



张量一共 8 个属性

data：被包装的张量

dtype：张量的数据类型

shape：张量的形状，二维张量为:行*列

device：张量所在设备，GPU/CPU，是加速处理的关键

require_grad：指示是否需要梯度，默认为 False

grad：data 的梯度值

Grad_fn：记录创建该张量时所用的方法（函数）,fn 是 function 函数的意思

is_leaf：指示是否是叶子结点（张量），用于计算图

```
print("tensor2的data\n", t2.data)
print("tensor2的shape\t", t2.shape)
print("tensor2的dtype\t", t2.dtype)

# tensor2的data
#  tensor([[1, 2, 3, 4],
#          [5, 6, 7, 8]], dtype=torch.int32)
# tensor2的shape  torch.Size([2, 4])
# tensor2的dtype  torch.int32

print("tensor1的shape\t", t1.shape)
# tensor1的shape  torch.Size([2])
```

```
print("tensor2的device\t", t2.device)
print("tensor2的requires_grad\t", t2.requires_grad)
print("tensor2的grad\t", t2.grad)
print("tensor2的grad_fn\t", t2.grad_fn)
print("tensor2的is_leaf\t", t2.is_leaf)
# tensor2的device     cpu
# tensor2的requires_grad  False
# tensor2的grad    None
# tensor2的grad_fn    None
# tensor2的is_leaf    True
```

关于 require_grad、grad、grad_fn、is_leaf，会面会详细介绍

（四）张量的数据类型

Pytorch 是一种包含单一数据类型元素的多维矩数组，即同一个张量中元素的数据是一致的。pytorch 中的数据类型是区分 CPU 和 GPU 的。

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	`torch.float16` or `torch.half`	`torch.HalfTensor`	`torch.cuda.HalfTensor`
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`

二、张量的特殊创建形式

（一）基于 from_numpy()创建 tensor

1、 torch.from_numpy()

从 torch.from_numpy 创建的 tensor 与原 ndarray 共享内存（更严谨：共享数组的值），当修改其中一个的数据，另一个也会被改动

```
arr = np.array([[1, 2, 3], [4, 5, 6]])
t3 = torch.from_numpy(arr)
print("numpy array: \n", arr)
print("tensor : \n", t3)

# numpy array:
#  [[1 2 3]
#  [4 5 6]]
# tensor :
#  tensor([[1, 2, 3],
#          [4, 5, 6]])


print("\n修改arr")
arr[0, 0] = 1000
print("numpy array: \n", arr)
print("tensor : \n", t3)

# 修改arr
# numpy array:
#  [[1000    2    3]
#  [   4    5    6]]
# tensor :
#  tensor([[1000,    2,    3],
#          [   4,    5,    6]])
```

```
print("\n修改tensor")
t3[1, 2] = 1000
print("numpy array:\n", arr)
print("tensor :\n", t3)
# 修改tensor
# numpy array:
#  [[   1    2    3]
#  [   4    5 1000]]
# tensor :
#  tensor([[   1,    2,    3],
#          [   4,    5, 1000]])
```

补充：这里如果查看 t3 和 arr 的内存地址，会发现不相同，这是因为 tensor 与 numpy 数组是共享部分内存，而非所有内存，准确说是共享值，所以直接用 id()得到的内存地址肯定是不同的

2、 tensor 转换为 numpy

也可以将 Tensor 转换为 Numpy 数组，调用 `张量名.numpy()` 可以实现这个转换操作。如果修改其中一个值，另一个也跟着变化。

```
# t3是一个tensor
new_arr = t3.numpy()
print(new_arr)
```

三、张量的梯度

（一）梯度的认识

在创建 tensor 时，将.requires_grad 设置为 true，**则会开始跟踪**针对 tensor 的所有操作。完成计算后，调用 .backward() 来自动计算结果对于该张量的梯度。该张量的梯度将累积到 .grad 属性中

训练深度学习模型本质上就是不断更新权值，而权值的更新需要求解梯度，因此求解梯度非常关键。然而求解梯度十分繁琐，pytorch 提供自动求导系统，利用这个自动求导系统，我们不需要手动计算梯度，就可以很简便的得到所有张量的梯度。

1、 创建包含梯度的 tensor

```
# 创建一个 tensor， 并让requires_grad=True 来追踪该变量相关的计算操作
x = torch.ones(2, 2, requires_grad=True)
print(x)
# tensor([[1., 1.],
#         [1., 1.]], requires_grad=True)
```

可以在定义一个变量时指定 require_grad 属性是 True。也可以定义变量后，调用 .requires_grad_设置为 True ， 这里带有后缀 _ 是会改变变量本身的属性

```
# 通过.requires_grad_()来用in-place的方式改变requires_grad属性
a = torch.randn(2,2)  # 默认requires_grad= False
a = ((a * 3) / (a - 1))
print(a.requires_grad)  # False
a.requires_grad_(True)
print(a.requires_grad)  # True
```

2、 对要求梯度的张量 进行计算操作

```
# 执行任意计算操作
y = x + 2
print(y)
print("y.grad_fn\t", y.grad_fn)  # 由什么函数生成的y
print("y.is_leaf\t", y.is_leaf)  # 非叶子
print("y.requires_grad\t", y.requires_grad) # 因为x的requires_grad=True,所以y的也为True
# tensor([[3., 3.],
#         [3., 3.]], grad_fn=<AddBackward0>)
# y.grad_fn  <AddBackward0 object at 0x0000021545315E48>
# y.is_leaf  False
# y.requires_grad    True

print("x.grad_fn\t", x.grad_fn)
print("x.is_leaf\t", x.is_leaf)
# x.grad_fn  None
# x.is_leaf  True
```

注意

每个 tensor 都有一个.grad_fn()属性，该属性创建该 tensor 的 function 类，就是说该 tensor 是不是通过某些运算得到的，若是，则 grad_fn 返回一个与这些运算相关的对象，否则为 none

x 是直接创建的，所以它没有 grad_fn 属性。y 是通过一个加法操作所创建的，所以它的 grad_fn 为 AddBackward0。

像 x 这种直接创建的称为叶子节点，叶子节点对应的 grad_fn 是 None

```
# 继续操作
z = y * y * 3
out = z.mean()

print('z=', z)
print('out=', out)
# z= tensor([[27., 27.],
#         [27., 27.]], grad_fn=<MulBackward0>)
# out= tensor(27., grad_fn=<MeanBackward0>)
```

至此，完成了 out = ((x+2)*(x+2)*3).mean()的计算式

3、 调用 .backward() 获得梯度

完成计算后，可以调用.backward()来完成所有梯度计算。计算结果值对于叶子节点张量的梯度，关于叶子节点的梯度将累积到该张量的.grad 属性中。

```
torch.autograd.backward(tensors, grad_tensors=None, retain_graph=None,
                        create_graph=False)
```

参数名称	说明
tensor	用于求导的张量，例如损失函数
retain_graph	保存计算图，由于 pytorch 采用动态图机制，在每一次反向传播结束之后，计算图都会释放掉。如果想继续使用计算图，就需要设置参数 retain_graph 为 True
create_graph	创建导数计算图，用于高阶求导
grad_tensors	多梯度权重；当有多个损失函数需要去计算梯度的时候，就要设计各个损失函数之间的权重比例

标量的反向传播

```
print("before梯度")
print("x.grad is {0}\n y.grad is {1}".format(x.grad, y.grad))
out.backward()
print("After梯度")
print("x.grad is {0}\n y.grad is {1}".format(x.grad, y.grad))
# before梯度
# x.grad is None
#  y.grad is None
# After梯度
# x.grad is tensor([[4.5000, 4.5000],
#         [4.5000, 4.5000]])
#  y.grad is None
# warnings.warn("The .grad attribute of a Tensor that
# is not a leaf Tensor is being accessed
```

结果分析：out.backward 之后，可以自动得出 x 的梯度，因为 y 是非叶子节点，所以仍然输出 None,并且输出警告。

注意：grad 在反向传播的过程中是累加的，这意味着每一次运行反向传播计算梯度都会累加之前的梯度，所以一般在反向传播之前需要把梯度清零

```
# 梯度不自动清零
out.backward()
print("第一次backward\n", x.grad)
z1 = (x*x).mean()
z1.backward()
print("第二次backward\n", x.grad)  # [4.5 +0.5]
```

```
new_x = torch.ones(2, 2, requires_grad=True)
new_z = (new_x*new_x).mean()
new_z.backward()
print("backward\n", new_x.grad)

# backward
#  tensor([[0.5000, 0.5000],
#          [0.5000, 0.5000]])
```

如果 z 不是一个标量，所以在调用 backward 时需要传入一个和 z 同形的权重向量进行加权求和得到一个标量

```
# 如果利用z的结果进行反向传播
print("before梯度")
print("x.grad is {0}".format(x.grad))
wt = torch.tensor([0.25, 0.25, 0.25, 0.25]).reshape(2, 2)
z.backward(wt)
print("After梯度")
print("x.grad is {0}\n".format(x.grad))
# before梯度
# x.grad is None
# After梯度
# x.grad is tensor([[4.5000, 4.5000],
#          [4.5000, 4.5000]])
```

此外还可以使用 with torch.no_grad()将不想被追踪的操作代码块包裹起来，这种方法在评估模型的时候很常用。

```
# 梯度中断
x = torch.tensor(1.0,requires_grad=True)
y1 = x ** 2
with torch.no_grad():
    y2 = x ** 3
y3 = y1 + y2

print(x.requires_grad)  # True
print(y1,y1.requires_grad)  # tensor(1., grad_fn=<PowBackward0>) True
print(y2,y2.requires_grad)  # tensor(1.) False
print(y3,y3.requires_grad)  # tensor(2., grad_fn=<AddBackward0>) True
y3.backward()
print(x.grad)  # tensor(2.),这里等于2是因为y2不能求导
```

第二讲：数据准备

一、数据模块的认识

数据模块可以细分为四个子模块：

数据收集：在进行实验之前，需要收集数据，数据包括原始样本和标签

数据拆分：有了原始数据之后，需要对数据集进行拆分，把数据集分为训练集、验证集和测试集；训练集用于训练模型，验证集用于验证模型是否过拟合，也可以理解为用验证集挑选模型的超参数，测试集用于测试模型的性能，测试模型的泛化能力；

数据读取和分批：

Dataset 是根据索引去读取数据以及对应的标签

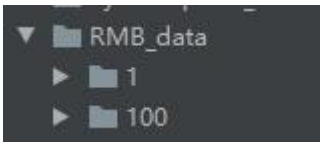
Dataloder 将从 Dataset 中拿到的数据，整理成 batch 的形式

数据预处理：把数据读取进来往往还需要对数据进行一系列的预处理，比如说数据的中心化，标准化，旋转或者翻转等等， pytorch 中数据预处理是通过 transforms 模块进行实现的

二、数据收集和拆分

（一）收集数据集

现在收集了一批人民币数据，包含 1 元和 100 元，两种类型各 100 张图片。将不同面额的数据放置在不同的目录下，如下图所示。这样根据文件夹的名字就可以得出图片对应的标签。现在数据和标签都已经准备好啦。



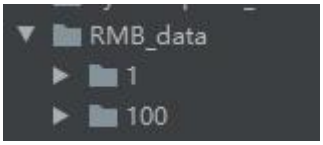
（二）拆分数据集

收集收集完成后，需要将数据集拆分了训练集、验证集和测试集，通常是按照 8:1:1

```
import os
import random
import shutil

def makedir(new_dir):
    if not os.path.exists(new_dir):
        os.makedirs(new_dir)

if __name__ == '__main__':

    random.seed(1)

    dataset_dir = "../../dataset/RMB_data"
    split_dir = "../../dataset/rmb_split"
    train_dir = os.path.join(split_dir, "train")
    valid_dir = os.path.join(split_dir, "valid")
    test_dir = os.path.join(split_dir, "test")

    train_pct = 0.8
    valid_pct = 0.1
    test_pct = 0.1
```

```
    for root, dirs, files in os.walk(dataset_dir):
        for sub_dir in dirs:

            imgs = os.listdir(os.path.join(root, sub_dir))
            imgs = list(filter(lambda x: x.endswith('.jpg'), imgs))
            random.shuffle(imgs)
            img_count = len(imgs)

            train_point = int(img_count * train_pct)
            valid_point = int(img_count * (train_pct + valid_pct))

            for i in range(img_count):
                if i < train_point:
                    out_dir = os.path.join(train_dir, sub_dir)
                elif i < valid_point:
                    out_dir = os.path.join(valid_dir, sub_dir)
                else:
                    out_dir = os.path.join(test_dir, sub_dir)

                makedir(out_dir)

                target_path = os.path.join(out_dir, imgs[i])
                src_path = os.path.join(dataset_dir, sub_dir, imgs[i])

                shutil.copy(src_path, target_path)

            print('Class:{}, train:{}, valid:{}, test:{}'.format(sub_dir, train_point, valid_point-
train_point,
                                                                 img_count-valid_point))
```

拆分前原始目录结构



拆分后的目录结构

三、数据读取和分批

（一）数据分批 Dataloader

pytorch 实际进行训练时，不是将所有样本一次性放入网络中进行训练。因为样本数量通常是以万、十万、百万位单位的，一次性读取需要大量的时间和内存消耗。因此，在深度学习训练时使用的方法是将数据拆分为多组，一般称为分批（batch）。在每次迭代训练时，读取一个 batch 的数据，然后不断重复以下过程：前向传播预测结果、通过损失函数计算误差、误差反向传播计算梯度、优化器更新误差梯度，直到所有 batch 都遍历完成

在 pytorch 中的 torch.utils.data 中提供了 Dataloader 类，该类可以将数据集组织为 batch 的形式，并形成可迭代的数据装载器

```
DataLoader(dataset,
           batch_size=1,
           shuffle=False,
           sampler=None,
           batch_sampler=None,
           num_workers=0,
           collate_fn=None,
           pin_memory=False,
           drop_last=False,
           timeout=0,
           worker_init_fn=None,
           multiprocessing_context=None)
```

功能：构建可迭代的数据装载器

参数：从上面的代码中可以看到， Dataloader 的参数非常多，共有 11 个参数，但常用的就是下面五个:

dataset ： Dataset 类，决定数据从哪里读取及如何读取【下面会详细讲】

batchsize ：批大小，每批有多少个样本

num_works ：是否多进程读取数据，进程数目为 2，表示 2 个进程。默认是 1

shuffle ：每个 epoch 中数据是否乱序

drop_last ：当样本总数不能被 batchsize 整除时，是否舍弃最后一批数据

解释：重点解释一下 **epoch**，**iteration**，**batchsize**

参数名称	说明
epoch	所有训练样本都已经输入到模型中，称为一个 epoch，1 个 epoch 表示训练集中的所有样本被遍历了一次
batchsize	批大小，表示每组/每批的样本数量
iteration	处理一个 batch 样本的过程称为一个 iteration, 遍历完一个 epoch 需要的迭代次数:epoch/batchsize（如果不能整，该值取决于 drop_last）

drop_last 作用：

样本总数	Batchsize	drop_last	Epoch
87	8	true	= 10 iteration
87	8	false	= 11 iteration

解释说明：当 drop_last 为 True,丢弃最后 7 个样本。如果为 False,则会从前面 80 个在随机抽取一个

（二）数据读取 Dataset

在 DataLoader 中有一个重要的参数是 dataset，主要负责从特定位置访问数据集，并对每个样本进行读取、转换处理。在 torch.utils.data 中提供了一个抽象 Dataset 类，所有自定义的 Dataset 需要继承它，并且复写 __getitem__()

1、 抽象类 dataset 说明

```
class Dataset(object):
    def __getitem__(self, index):
        raise NotImplementedError
    def __add__(self, other)
        return ConcatDataset([self,other])
```

功能：用来定义数据从哪里读取，以及如何读取的问题。

参数：getitem ：接收一个索引，返回一个样本

比如根据的目录去读取文件、以哪个图像处理工具读取

2、 案例：人民币数据

首先，定义一个 RMBDataset 类

传入参数为：图片路径和图像预处理操作（data_dir，transform）

```
import os
import random
from PIL import Image
from torch.utils.data import Dataset

random.seed(1)
rmb_label = {"1": 0, "100": 1}

class RMBDataset(Dataset):
    def __init__(self, data_dir, transform=None):
        """
        rmb面额分类任务的Dataset
        :param data_dir: str, 数据集所在路径
        :param transform: torch.transform, 数据预处理
        """
        self.label_name = {"1": 0, "100": 1}
        self.data_info = self.get_img_info(data_dir)  # data_info存储所有图片路径和标签，在DataLoader中通过index读取样本
        self.transform = transform

    def __getitem__(self, index):
        path_img, label = self.data_info[index]
        img = Image.open(path_img).convert('RGB')     # 0~255

        if self.transform is not None:
            img = self.transform(img)   # 在这里做transform，转为tensor等等

        return img, label

    def __len__(self):
        return len(self.data_info)
```

```
    @staticmethod
    def get_img_info(data_dir):
        data_info = list()
        for root, dirs, _ in os.walk(data_dir):
            # 遍历类别
            for sub_dir in dirs:
                img_names = os.listdir(os.path.join(root, sub_dir))
                img_names = list(filter(lambda x: x.endswith('.jpg'), img_names))

                # 遍历图片
                for i in range(len(img_names)):
                    img_name = img_names[i]
                    path_img = os.path.join(root, sub_dir, img_name)
                    label = rmb_label[sub_dir]
                    data_info.append((path_img, int(label)))

        return data_info
```

3、 实例化一个 RMBDataset 的对象

构建 trian_data 对象（传入训练集路径）。将 trian_data 放入 DataLoader，可生成 pytorch 训练过程中要求的数据格式

```
split_dir = "../../dataset/rmb_split"
train_dir = os.path.join(split_dir, "train")

norm_mean = [0.485, 0.456, 0.406]
norm_std = [0.229, 0.224, 0.225]

train_transform = transforms.Compose([
    transforms.Resize((32, 32)),
    transforms.ToTensor(),
    transforms.Normalize(norm_mean, norm_std),
])

# 构建MyDataset实例
train_data = RMBDataset(data_dir=train_dir, transform=train_transform)
train_loader = DataLoader(dataset=train_data, batch_size=BATCH_SIZE,
shuffle=True)
```

train_transform 对每张图像进行缩放到同一尺寸、转换为 tensor()、数据归一化处理，这是数据处理的必要步骤。