Pytorch

pytorch□tensorflow的区别:

pytorch支持动态计算图

- 计算图在代码运行时动态生成
- 可根据运行逻辑实时更改计算图

tensorflow支持静态计算图

- 先定义后执行
- 计算图在编译后无法再修改

datasets

```
from torchvision import datasets
```

□载数据集

- `.□据集名称`
  - `root`：数据集保存路径
  - `train`：是否加载数据集
  - `download`：自动下载
  - `transform`：可传入预处理变换

Dataset

在`torch.utils.data`下，是一个抽象类，用于表示数据集

自定义数据集需要继承`torch.utils.data.Dataset`并实现以下两个方法:

- `__len__()`：返回数据集的总样本数。
- `__getitem__(idx)`：根据索引`idx`返回单个样本（数据和标签）。

DataLoader

在`torch.utils.data`下，是对`Dataset`的封装，有以下功能:

- 批量□载（`batch_size`）：将多个样本组合成一个批次（batch）。
- 随机打乱（`shuffle=True`）：每个 epoch 重新打乱数据顺序。
- 多进程□速（`num_workers`）：使用多线程/进程并行加载数据

参数	说明
dataset	要加载的Dataset对象
batch_size	每个批次的样本数
shuffle	是否在每个epoch打乱顺序
num_workers	加载数据的进程数（可为CPU核心数）
drop_last	是否丢弃最后一个不完整批次
pin_memory	是否将数据固定到GPU显存

Transfrom

在 `torchvision` 下，可用作图像预处理

方法	说明
`ToTensor()`	将 PIL 图像或数组转为张量
`Normalize(mean, std)`	标准化（减均值，除标准差）
`RandomCrop(size)`	随机裁剪
`RandomHorizontalFlip()`	随机水平翻转

激活函数

在 `torch.nn.Function` 下 $$ ReLU(x) = max(0,x) $$

- Relu
  - 输出范围：$[0, +\infty)$
  - 零中心分布，梯度比Sigmoid更强。
  - 梯度消失问题仍存在。

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

- Tanh
  - 输出范围：(-1, 1)
  - 计算高效，缓解梯度消失（正区间梯度为1）。
  - 稀疏激活（负输入直接置0）。

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

- Sigmoid
  - 输出范围：（0,1）
  - 梯度消失（输入绝对值较大时梯度接近0）。
  - 输出非零中心（影响梯度更新效率）。

$$ GELU(x) = x \cdot \Phi(x) $$

- GELU（高斯误差线性单元）
  - 比 ReLU 更平滑，梯度更稳定，适合深层网络
  - $\Phi(x)$是标注正态分布的累积分布函数
  - 适合 Transformer 结构

$$ Swish(x) = x \cdot \sigma(\beta x) $$

- Swish
  - σ为Sigmoid函数，β可以是可学习参数或者是固定超参数
  - 平滑，非单调，梯度更稳定（优于**ReLU**）
  - 适用于Transformer、大规模预训练模型，但计算量略高

```
class Swish(nn.Module):
    def __init__(self,beta = 1.0):
        super().__init__()
        self.beta = beta

    def forward(self,x):
        return x * torch.sigmoid(self.beta * x)
```

$$ PReLU(x) = \begin{cases} x & \text{if } x \geq 0 \ ax & \text{if } x < 0 \end{cases} $$

- PReLU
    - 自动学习负区间的斜率，a为可学习参数
    - 应用于深层CNN，可缓解梯度消失！！！

损失函数

在 `torch.nn` 下

1. 均方误差损失（**MSE**）
    - 公式：

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

    - 用于回归任务
    - `nn.MSELoss()`
2. 交叉熵损失（**Cross-Entropy**）
    - 公式：

    $$ CE = -\sum_{i=1}^n y_i\log(\hat{y_i}) $$

    - 用于分类任务
    - `nn.CrossEntropyLoss()`

优化器

在 `torch.optim` 下

优化器的基本使用步骤如下：

1. 定义模型（如 `nn.Linear`、`nn.Conv2d`）。
2. 选择优化器（如 `optim.SGD`、`optim.Adam`）。
3. 在训练循环中更新参数：
    - `optimizer.zero_grad()`：清空梯度。
    - `loss.backward()`：计算梯度。
    - `optimizer.step()`：更新参数。

SGD（随机梯度下降）

- 参数：
    - model.paramaters()（模型参数）
    - lr（学习率）
    - momentum（动量）
    - dampening（动量阻尼，默认 0）

    - weight_decay（L2 正则化，默认 0

$$ v_t = \beta v_{t-1} + (1-\beta) \frac{\partial J}{\partial w} \\ w = w - \alpha v_t $$

Adam（自适应矩估计）

- 结合动量（一阶矩）和自适应学习率（二阶矩），适应不同参数的梯度变化
- 参数:
    - model.paramaters()（模型参数）
    - lr（学习率）
    - betas（元组）
        - β_1: 一阶矩衰减率
        - β_2: 二阶矩衰减率
    - eps（数值稳定系数，避免除0）
    - weight_decay（权重衰减，L2正则化）

1. 计算一阶矩（动量）和二阶矩（梯度平方）

g_t为梯度 $$ m_t = \beta_1 \cdot m_{t-1} + (1 - \beta_1)\cdot g_t \\ v_t = \beta_2\cdot v_{t-1}+(1-\beta_2)\cdot g_t^2 $$

2. 偏差修正

$$ \hat m_t = \frac{m_t}{1-\beta_1^t}\\ \hat v_t = \frac{v_t}{1-\beta_2^t} $$

3. 参数更新

$$ \theta_{t+1} = \theta_t - \frac{\alpha \cdot \hat m_t}{\sqrt{\hat v_t + \epsilon}} $$

RMSprop

- 对梯度平方进行指数加权平均
- 参数
    - model.paramaters()（模型参数）
    - lr（学习率）
    - alpha（梯度平方的衰减率）
    - eps（数值稳定系数，避免除0）
    - weight_decay（权重衰减，L2正则化）

$$ v_t = \beta v_{t-1} + (1-\beta) (\frac{\partial J}{\partial w})^2 \\ w = w - \alpha \frac{\frac{\partial J}{\partial w}}{\sqrt{v_t + \epsilon}} $$

学习率调度器

动态调整学习率的利器

`from torch.optim import lr_scheduler`

StepLR（固定步长衰减）

参数	说明
`optimizer`	优化器对象
`step_size`	每隔多少epoch衰减一次学习率
`gamma`	衰减因子（学习率乘以0.1）

`last_epoch`	最后一个epoch索引

`scheduler.step()`：更新学习率（**scheduler**为调度器实例对象）

ReduceLROnPlateau (根据指标动态调整)

参数	说明
`optimizer`	优化器对象
`mode`	`'min'`（监控指标下降）或`'max'`（监控指标上升）
`factor`	学习率衰减因子
`patience`	等待多少个 epoch 指标无改善才降低学习率
`threshold`	变化阈值，只有变化超过该值才认为有改善
`threshold_mode`	`'rel'`（相对变化）或 `'abs'`（绝对变化）
`cooldown`	降低学习率后等待多少个 epoch 才再次监测
`min_lr`	学习率下限
`verbose`	是否打印更新信息

`scheduler.step(val_loss)`：传入监控指标

CosineAnnealingLR (余弦退火)

参数	说明
`optimizer`	优化器对象
`T_max`	余弦周期长度（epoch数）
`eta_min`	最小学习率
`last_epoch`	最后一个epoch索引

`scheduler.step()`：更新学习率

Conv

在`torch.nn`下，用于提取局部特征，通过滑动窗口计算输入数据的加权和

每个卷积核的维度是$(in_channels,H,W)$

- 处理图片时，每个卷积核的通道分别和输入的通道进行相乘
- 再把通道的卷积结果相加，得到一个单通道特征图

参数：

- `in_channels`：输入图像的通道数（RGB图像为3）
- `out_channels`：输出通道数（卷积核数量）
- `kernel_size`：卷积核大小，为元组形式
- `stride`：滑动步长
- `padding`：边缘填充

- `bias`：偏置项
- `dilation`：空洞卷积（扩大感受野，采样点变稀疏）
- `groups`：将通道分为几组，组内计算卷积，组间互不交互

输出尺寸的计算 $$ H_{out} = \frac{H_{in} + 2 \times padding - dilation\times(kernel_size - 1) - 1 }{stride} $$

TransPosed Conv

在 `torch.nn` 下，为转置卷积，通常用在语义分割的上采样的过程中，`nn.ConvTranspose2d`

转置卷积本质上就是卷积

输入和卷积核的每个元素分别相乘并叠在输出图上和下面的运算步骤是完全等价的

运算步骤

1. 在输入特征图元素间填充 `s-1` 行、列0
2. 在输入特征图四周填充 `k-p-1` 行、列0
3. 将卷积核参数上下、左右翻转
4. 做正常卷积运算

参数：

- `in_channels`：输入图像的通道数（RGB图像为3）
- `out_channels`：输出通道数（卷积核数量）
- `kernel_size`：卷积核大小，为元组形式
- `stride`：元素间填充的0的间隔
- `padding`：边缘填充
- `bias`：偏置项
- `out_padding`：解决尺寸对齐问题，在输出特征图进行填充

 image-20250805130358084

 image-20250805130421265

输出尺寸的计算 $$ H_{out} = (H_{in} - 1) \times stride - 2 \times padding + kernel_size+output_padding $$

Pool

在 `torch.nn` 下，用于将提取到的特征进行降维、增强特征不变性

常用池化层

- `nn.MaxPool2d`：二维最大池化（保留显著特征）
- `nn.AvgPool2d`：二维平均池化（平滑特征）
- `nn.AdaptiveMaxPool2d`：自适应最大池化（固定输出尺寸）

参数：

- `kernel_size`：池化窗口大小
- `stride`：步长（默认等于 `kernel_size`）
- `padding`：边缘填充
- `dilation`：空洞池化

输出尺寸的计算 $$ H_{out} = \frac{H_{in} + 2 \times padding - dilation\times(kernel_size - 1) - 1 }{stride} $$

Linear

在 `torch.nn` 下，将输入数据的最后一个维度进行线性映射 $$ y = x \cdot W^T + b $$ 其中：

- x的最后一个维度必须等于$in_features$

参数：

- in_feature:输入数据大小
- out_feature:输出数据大小
- bias:偏置

输入维度要和前一层输出维度的最后一个维度相等，线性层只对前一层的最后一个维度做变换

Flatten

在 `torch.nn` 下，将多维输入张量展平为一维向量

- 输入：多维张量（如卷积层的输出 (batch_size, channels, height, width))
- 输出：展平后的 2D 张量，形状为 (batch_size, channels * height * width)

Normalization

都在 `torch.nn` 下

Batch Normalization

对多个输入的同一特征进行归一化

用于全连接层：`nn.BatchNorm1d`

- 参数为特征维度

用于卷积层：`nn.BatchNorm2d`（每个通道独立进行跨批量（Batch）的归一化）

- 输入张量形状：`[Batch_size, Channels, Height, Width]`
- 对每个通道的所有Batch样本的所有像素点独立计算均值和方差，跨**Batch**归一化
- 参数为通道数（Channels）

作用：

- 作用在每一层的输入上
- 对一个batch内的数据按特征维度作归一化
- 缓解梯度消失，梯度爆炸，加快收敛

Layer Normalization

对单个样本的不同特征维度进行归一化

`nn.LayerNorm`

作用：

- 作用在样本的每一层上
- 适合**RNN/Transformer**

Group Normalization

`nn.GroupNorm`

[illegible]

[illegible]

- `num_groups` [illegible]
- `num_channels` [illegible]
- `affine` [illegible]

Upsample

`nn.Upsample`

[illegible]

[illegible]

[illegible]	[illegible]	[illegible]
`size`	int[illegible]tuple	[illegible]`(256, 256)`[illegible]None[illegible]`scale_factor`[illegible]
`scale_factor`	float[illegible]tuple	[illegible]`2`[illegible]2[illegible]`(2, 3)`[illegible]2[illegible]3[illegible]
`mode`	str	[illegible]`'nearest'`[illegible]`'bilinear'`[illegible]`'bicubic'`[illegible]`'nearest'`[illegible]
`align_corners`	bool	[illegible]`bilinear/bicubic`[illegible]False[illegible]

Dropout

[illegible] `torch.nn` [illegible]0[illegible]

[illegible] **`.train()`** [illegible] **`.eval()`** [illegible]

Interpolate

[illegible] `torch.nn.functional.interpolate` [illegible]

[illegible]/[illegible]

[illegible]

[illegible]	[illegible]	[illegible]
`size`	`(target_h, target_w)`	[illegible]
`scale_factor`	`float` [illegible] `(float, float)`	[illegible] `2.0` [illegible]2[illegible]
`mode`	`'nearest'`, `'bilinear'`, `'bicubic'`, `'area'`	[illegible]
`align_corners`	`bool`	[illegible]`True` [illegible]

[illegible]

[illegible]

- `torch.utils.checkpoint`
  - [illegible]

    1. 将需要检查点的代码块包装在 `checkpoint` 函数中
    2. 前向传播时，用 `checkpoint` 替代直接调用模块
  - 参数
    - `fn`：要检查点的函数或模块
    - `**args`：输入参数（通常是张量）

示例

```python
import torch
import torch.nn as nn
from torch.utils.checkpoint import checkpoint

class BigModel(nn.Module):
    def __init__(self):
        super().__init__()
        self.layer1 = nn.Linear(1000, 1000)
        self.layer2 = nn.Linear(1000, 1000)
        self.layer3 = nn.Linear(1000, 1000)

    def forward(self, x):
        # 对计算量大的部分启用检查点
        x = checkpoint(self.layer1, x)  # 代替直接调用 self.layer1(x)
        x = checkpoint(self.layer2, x)
        x = self.layer3(x)  # 最后一层不检查点（因为需要输出）
        return x

## 使用
model = BigModel()
inputs = torch.randn(32, 1000)
outputs = model(inputs)
loss = outputs.sum()
loss.backward()  # 反向传播时会重新计算layer1和layer2的激活值
```

- `torch.utils.checkpoint_sequential`：**Sequential**模块专用
  - 参数
    - `functions`：Sequential模块或模块列表
    - `input`：输入参数（通常是张量）
    - `segments`：分段数量，将Sequential模块分成多少段检查点
      - 将 `nn.Sequential` 均匀分成 `segments` 段，每段的前向传播中间激活值会被缓存，段内的中间激活值会被丢弃
      - 显存紧张时可以增加 `segments`，模型分成更多段，显存占用更少

示例

```python
from torch.utils.checkpoint import checkpoint_sequential

model = nn.Sequential(
    nn.Linear(10, 10),
    nn.ReLU(),
    nn.Linear(10, 10),
    nn.ReLU(),
```

```
    nn.Linear(10, 10)
)

input_tensor = torch.randn(2, 10, requires_grad=True)
## 将模型按 chunk 分成2段，每段 forward 时会 checkpoint
output = checkpoint_sequential(model, chunks=2, input=input_tensor)
```

梯度累积

代码实现

```
## 梯度累积步
accumulation_steps = 4
## 梯度清零
optimizer.zero_grad()

for i ,(inputs,labels) in enumerate(datalaoder):
    outputs = model(inputs)
    # 计算损失
    loss = criterion(outputs,labels)
    # 损失除以梯度累积步，为了严格等价于大批量训练
    loss = loss / accumulation_steps
    # 反向传播
    loss.backward() # 这里注意不进行参更新和梯度清零

    if (i + 1) % accumulation_steps == 0:
        # 参更新
        optimizer.step()
        # 梯度清零
        optimizer.zero_grad()
```

推导:

- 真实大批量梯度（$batch_size$=36）

$$ \nabla L_{true} = \frac{1}{32}\sum_{i = 1}^{32}\nabla loss_i $$

- 梯度累积（分4次，$mini_batch$ = 8）

$$ \nabla L_{accu} = \sum_{j=1}^4(\frac{1}{4}\cdot \frac{1}{8}\sum_{k=1}^8\nabla loss_{jk}) = \frac{1}{32}\sum_{i=1}^{32}\nabla loss_i $$

梯度裁剪

在 `torch.nn.utils` 下，通过限制梯度的最大范数，防止梯度爆炸

用在反向传播之后，参数更新之前

按值裁剪（ `clip_grad_value_` ）

- 直接限制梯度值的范围，把所有梯度裁剪到自己设定的范围，但可能丢失梯度方向等信息
- `torch.nn.utils.clip_grad_value_`
  - `model.parameters()` ：模型参数

  - `clip_value`：梯度的最大值

按范数裁剪（`clip_grad_norm_`）

- 计算所有梯度的**L2**范数，如果超过阈值，则按比例缩放梯度，能够保留梯度方向
- `torch.nn.utils.clip_grad_norm_`
  - `model.parameters()`：模型参数
  - `max_norm`：梯度的最大L2范数

早停机制

pytorch本身不支持，可以自定义实现

```python
class EarlyStopping:
    def __init__(self,patience=3):
        self.patience = patience
        self.best_loss = float('inf')
        self.counter = 0
    # 调用类的实例的时候会调用这个方法
    def __call__(self,val_loss):
        if val_loss < self.best_loss:
            self.best_loss = val_loss
            self.counter = 0
            return False
        else:
            self.counter += 1
            return self.counter >= self.patience
```

量化与剪枝

量化

模型融合是必须的

动态量化：在推理时将权重转为`int8`，激活仍为`float32`

- 在`torch.quantization`下，`from torch.quantization import quantize_dynamic`
  - `model`：原始的模型
  - `{nn.Linear,nn.LSTM}`：需要量化的层的集合
  - `dtype`：需要量化的精度类型

静态量化：权重和激活都量化为`int8`

- 步骤：
  1. 定义或加载模型
  2. 模型融合（`fuse_modules`）
     - `torch.quantization.fuse_modules`
  3. 指定量化配置（`qconfig`）
     - `torch.quantization.get_default_qconfig`
       - `fbgemm`用于x86，`qnnpack`用于ARM

```
model.qconfig = torch.quantization.get_default_qconfig('fbgemm')
```

4. 准备量化（`prepare`）
    - `torch.quantization.prepare()`
        - `inplace` 表示是否原地修改
5. 校准（用一小部分数据跑 forward）
6. 转换为量化模型（`convert`）
    - `torch.quantization.convert`

示例代码

```
model = MyModel()
model.eval()

## 1. 融合模块
model_fused = torch.quantization.fuse_modules(model, [['conv', 'relu']])

## 2. 设置qconfig
model_fused.qconfig = torch.quantization.get_default_qconfig('fbgemm')

## 3. 准备量化
torch.quantization.prepare(model_fused, inplace=True)

## 4. 校准
with torch.no_grad():
    for data, _ in calibration_loader:
        model_fused(data)

## 5. 转换
torch.quantization.convert(model_fused, inplace=True)
```

量化感知训练**(QAT)**：在训练阶段就**“**模拟量化”的影响，通过 `FakeQuantize` 操作让网络感知量化的误差

- 步骤
    1. 融合模块
        - `torch.quantization.fuse_modules`
    2. 设置 `qconfig`（使用 QAT 配置）
        - `torch.quantization.get_default_qat_qconfig`
    3. 使用 `prepare_qat`
        - `torch.quantization.prepare_qat()`
    4. 训练若干 epoch
    5. `convert` 得到最终模型
        - `torch.quantization.convert()`

示例代码

```
model.train()
model_fused = torch.quantization.fuse_modules(model, [['conv', 'relu']])

## 1. 设置为 QAT 配置
model_fused.qconfig = torch.quantization.get_default_qat_qconfig('fbgemm')

## 2. 准备 QAT
torch.quantization.prepare_qat(model_fused, inplace=True)

## 3. 正常训练
for epoch in range(num_epochs):
    for data, target in train_loader:
        output = model_fused(data)
        loss = loss_fn(output, target)
        loss.backward()
        optimizer.step()

## 4. 转换为真正的量化模型
model_fused.eval()
torch.quantization.convert(model_fused, inplace=True)
```

剪枝

在 `torch.nn.utils.prune` 下

常见方法：

- `prune.l1_unstructured`：按**L1**范数裁剪（非结构化剪枝）
    - 计算一个权重张量中每个元素的**L1**范数
    - 按照设定比例把绝对值最小的一部分权重置为**0**
- `prune.random_unstructured`：随机裁剪（非结构化剪枝）
    - `module`：要剪枝的模块
    - `name`：字符串，指定剪枝的权重参数名称
    - `amount`：
        - 如果是 `float`，表示要剪枝的比例
        - 如果是 `int`，表示要剪掉的权重个数
- `prune.ln_structured`：保留整列/整行
    - 结构化剪枝
    - `amount`：
        - 如果是 `float`，表示要剪枝的比例
        - 如果是 `int`，表示要剪掉的权重个数
    - `n`：L-n范数
    - `dim`：在哪个维度剪
        - `0`：按行剪
        - `1`：按列剪
- `prune.remove`：移除掩码，导出稀疏模型
    - `module`：要剪枝的模块
    - `name`：字符串，指定剪枝的权重参数名称

补充：剪枝后的模型参数仍然保留在完整权重中，只是部分权重被mask为0，可以导出稀疏模型进一步压缩

可视化工具

一般用**tensorboard**

在 `torch.utils` 下，一般 `from torch.ultis.tensorboard import SummaryWriter`

- 初始化函数可以输入一个文件夹名称，使得这个文件被tensorboard解析,使用完后要把对象关掉

```
from torch.utils.tensorboard import SummaryWriter
## 创建 SummaryWriter 并指定日志目录

writer = SummaryWriter('C:/study/pytorch/logs')

## 关闭 SummaryWriter

writer.close()
```

1. 记录标量
    - `writer.add_scalar()`
        - `tag`：图表标题
            - 使用 `train/loss`，`eval/loss` 就可以实现自动分组
        - `scalar_value`：要保存的数值，对应的是y轴
        - `global_step`：训练了多少步，对应的是x轴
2. 记录模型结构，显示每一层的名称
    - `writer.add_graph()`
        - 模型
        - 输入张量
3. 记录超参数组合及其效果
    - `writer.add_hparams()`
        - 超参数组合，为字典格式
        - 性能指标，为字典格式

tqdm

展现训练或测试进度条

`from tqdm import tqdm`

`tqdm()`

- 训练集或者测试集的**Dataloader**实例
- `desc`：进度条左侧描述文字
- `total`：总批次数
- `leave`：进度条完成后是否保留

- `unit`：进度单位（`it/s`或者`batch/s`）
- `ncols`：进度条宽度

`实例.set_postfix()`：进度条右侧添加性能指标信息

- 单指标：`train_bar.set_postfix(loss=f"{loss.item():.4f}")`
- 多指标：

```
train_bar.set_postfix({
        'loss': f"{loss.item():.4f}",
        'acc': f"{acc.item():.2%}"  # 百分比格式
    })
```

模型保存与□载

- 保存模型权重

`torch.save(model.state_dict(), './model_weights.pth')`

- 从权重文件□载模型

`model.load_state_dict(torch.load('./model_weights.pth'))`

- 保存整个模型（含结构）

`torch.save(model,path)`

- □载模型

`model = torch.load(path)`

单机多卡

导入四个包

- `from torch.nn.parallel import DistributedDataParallel as DDP`
- `from torch.utils.data.distributed import DistributedSampler`
- `import torch.distributed as dist`
- `import torch.multiprocessing as mp`

具体步骤：

1. 初始化进程组

设置通信后端，每个进程绑定一块GPU

2. 清理进程组

训练结束后销毁进程组，释放资源

3. 用`DDP`包装模型，并用`DistributedSampler`分片数据
4. 使用`torch.multiprocessing.spawn`启动多进程

易错点

- `loss.backward()`是通过激活值计算出来梯度并在参数上的`.grad`属性进行累加，计算出来梯度后，缓存在显存中的激活值会被释放

  - 如果不调用 `optimizer.step()`，参数不会更新
  - 如果不调用 `optimizer.zero_grad()`，梯度会一直累加
  - 进行前向传播的时候会把中间过程的激活值算出来并存储在显存中
- `torch.no_grad()`：禁用梯度计算，用于推理
- `.requires_grad_`：控制是否对张量启用梯度计算，不启用则不会存储前向传播的中间激活值
- `loss.item()`：将单元素张量转换为 **Python**标量
- `optim.zero_grad()`：梯度清零
- `optimizer.step()`：参数更新
- `loss.backward()`：反向传播
- `self.training` 是 `nn.Module` 类的一个内置属性，用于指示模型当前是否处于训练模式（`True`）或评估模式（`False`）
- 归一化一般都放在激活函数之前
- 多维张量相乘是最后两维相乘，其他维度广播对齐
- `.permute()` 是对张量进行旋转（不改变数据相对顺序），`.reshape()` 是对数据进行重排列（改变数据相对顺序）