Learning PyTorch

Tensor

▼ 属性

- `data`: 数值
- `grad`: 梯度值
- `grad_fn`
- `requires_grad`
- `is_leaf`
  - 叶子节点的概念主要是为了节省内存，所有节点都依赖于叶子节点（初始节点），在计算图中的一轮反向传播结束之后，非叶子节点的梯度会被释放（除非指定 retain_grad()）
  - PyTorch 使用动态图机制，运算和搭建同时进行，可以先计算前面节点的值，再根据这些值搭建后面的计算图
- `dtype`: 数据类型，如 torch.FloatTensor，torch.cuda.FloatTensor
  - 分为 3 大类：float (16-bit, 32-bit, 64-bit)、integer (unsigned-8-bit ,8-bit, 16-bit, 32-bit, 64-bit)、Boolean。模型参数和数据用的最多的类型是 float-32-bit。label 常用的类型是 integer-64-bit
- `device`

▼ 创建

▼ 根据数值创建

- `torch.tensor(data, dtype, device, requires_grad)`
  - `data`: list/numpy…
- `torch.from_numpy(np.ndarray)`
- `torch.zeros(*size, out=None, dtype=None, layout=torch.strided, device=None, requires_grad=False)`
  - `out`: 输出的张量，如果指定了 out，那么 `torch.zeros()` 返回的张量和 out 指向的是同一个地址
  - `layout`: 内存中布局形式，有 strided，sparse_coo 等。当是稀疏矩阵时，设置为 sparse_coo 可以减少内存占用
- `torch.zeros_like(input)`
- `torch.full(size, fill_value)`：创建全为自定义数值（fill_value）的张量
- `torch.arange(start=0, end, step=1)`：创建 **[start,end)** 的一维张量
- `torch.linspace(start, end, steps=100)`：创建 **[start, end]，元素个数为 steps** 的均分一维张量

▼ 根据分布创建

- `torch.logspace(start, end, steps=100, base=10)`：创建 **[base^start, base^end]，元素个数为 steps** 的对数均分一维张量
- `torch.eye(n, m=None)`：创建行为 n，列为 m（默认与 n 相同）的单位对角矩阵
- `torch.normal(mean, std)`：正态分布采样
  1. mean 为标量，std 为标量，**这时需要设置 size**，在同一正态分布中采样
  2. mean 为标量，std 为张量
  3. mean 为张量，std 为标量

     采样的分布一个相同，另一个不同
  4. mean 为张量，std 为张量

     从不同分布中采样
- `torch.randn(*size)` `torch.randn_like(input)`：生成**标准**正态分布

- `torch.rand(*size)` `torch.rand_like(input)`：生成 **[0,1)** 上的**均匀**分布
- `torch.randint(low=0, high, size)`：生成 **[low, high)** 上**整数**均匀分布
- `torch.randperm(n)`：生成 **0 到 n-1 的随机排列**，常用于**生成索引**
- `torch.bernoulli(input)`：生成以 input (tensor) 为概率 (p) 的伯努利分布（0-1 离散分布）

▼ 操作

▼ 广播机制

1. 从最后一个维度开始匹配
2. 在前面插入若干维度（进行 unsqueeze 操作）
3. 将维度的 size 从 1 扩张到与某个张量相同的维度

例如：

- Feature maps：[4, **32, 14, 14**]
- Bias：[**32, 1, 1**]（Tip：后面的两个1是手动unsqueeze插入的维度）-> [1, 32, 1, 1] -> [4, 32, 14, 14]

▼ 拼接

- `torch.cat(tensors, dim=0)`：将张量在 dim 维上拼接（除了 dim 维其他维度必须相同）
- `torch.stack(tensors, dim=0)`：将张量在**新创建的** dim 维上拼接（两个张量 shape 必须相同）

▼ 切分

- `torch.chunk(input, chunks, dim=0)`：将 input 向量在 dim 维上**切分为 chunks 份**（若不能整除，则最后一份张量小于其他张量）【按个数拆分】
- `torch.split(tensor, split_size_or_sections, dim=0)`：split_size_or_sections 为 int 时，表示**切分的每一份长度**（若不能整除，则最后一份张量小于其他张量）；为 list 时，按照 list 元素作为每一个分量的长度切分（list 中元素之和必须等于切分维度的值）【按长度拆分】

▼ 索引

- `torch.index_select(input, dim, index)`：在 dim 维度上，按照 index 索引取出 input 中的数据拼接为张量返回

```
# 创建均匀分布
t = torch.randint(0, 9, size=(3, 3))
# 注意 idx 的 dtype 不能指定为 torch.float
idx = torch.tensor([0, 2], dtype=torch.long)
# 取出第 0 行和第 2 行
t_select = torch.index_select(t, dim=0, index=idx)
print("t:\n{}\nt_select:\n{}".format(t, t_select))

'''
t:
tensor([[4, 5, 0],
        [5, 7, 1],
        [2, 5, 8]])
t_select:
tensor([[4, 5, 0],
        [2, 5, 8]])
'''
```

- `torch.mask_select(input, mask)`：按照 mask 中的 True 进行索引拼接得到**一维张量**返回（mask 为与 input 同形的布尔类张量）

```
t = torch.randint(0, 9, size=(3, 3))
mask = t.le(5)  # ge is mean greater than or equal/   gt: greater than  le  lt
# 取出大于 5 的数
t_select = torch.masked_select(t, mask)
print("t:\n{}\nmask:\n{}\nt_select:\n{} ".format(t, mask, t_select))
'''
t:
tensor([[4, 5, 0],
        [5, 7, 1],
        [2, 5, 8]])
mask:
tensor([[ True,  True,  True],
        [ True, False,  True],
        [ True,  True, False]])
t_select:
tensor([4, 5, 0, 5, 1, 2, 5])
'''
```

- `torch.ge(input, other)` `le` `gt` `lt` `nonzero`：根据比较结果返回布尔型张量，`other` 可以是数值或张量

▼ 变换

- `torch.reshape(input, shape)`

  注意：当张量在内存中是连续的时，返回的张量和原来的张量**共享数据内存**，改变一个变量另一个也会改变！

- `torch.transpose(input, dim0, dim1)`：交换张量的两个维度
- `torch.permute(input, dims)`：置换张量维度（内存不变）
- `torch.squeeze(input, dim=None)`：压缩长度为 1 的维度（dim 若为 None 则移除所有长度为 1 的维度；若指定维度，则当且仅当该维度长度为 1 时可以移除）
- `torch.unsqueeze(input, dim)`：根据 dim 扩展维度，长度为 1

▼ 扩张与缩减

- `torch.expand()`
- repeat
- narrow

▼ 近似与裁剪

- `torch.floor` `torch.ceil` `torch.trunc` `torch.frac` `torch.round`：向下取整、向上取整、取整数部分、取小数部分、四舍五入
- `torch.clamp(input, min, max)`：对张量中的元素，小于 min 的都设置为 min，大于 max 的都设置成 max

▼ 统计属性

- `torch.norm(input, p, dim=None)`：在 dim 维上对 input 张量**求 p 范数**
- `torch.prod(input, dim=None)`：累积
- `torch.argmax/argmin(input, dim=None)`：最大值最小值索引（若不指定 dim，默认会将Tensor打平后取最大值索引和最小值索引）
- `torch.topk(input, k, dim=None, largest=True, sorted=True)`：返回二元组 `values` `indices`，分别是 input 中前 k 大（ largest=False 时为前 k 小）的值和索引（**不指定 dim 时默认为最后一维**，返回的两个张量与 input 相比除了 dim 维的值变成 k 外其他维不变）

```
>>> x = torch.arange(1., 6.)
>>> x
tensor([ 1.,  2.,  3.,  4.,  5.])
>>> torch.topk(x, 3)
torch.return_types.topk(values=tensor([5., 4., 3.]), indices=tensor([4, 3, 2]))
```

- `torch.kthvalue(input, k, dim=None, keepdim=False)`：返回 dim 维上**第 k 小**的元素及其索引

```
>>> x = torch.arange(1., 6.)
>>> x
tensor([ 1.,  2.,  3.,  4.,  5.])
>>> torch.kthvalue(x, 4)
torch.return_types.kthvalue(values=tensor(4.), indices=tensor(3))

>>> x=torch.arange(1.,7.).resize_(2,3)
>>> x
tensor([[ 1.,  2.,  3.],
        [ 4.,  5.,  6.]])
>>> torch.kthvalue(x, 2, 0, True)
torch.return_types.kthvalue(values=tensor([[4., 5., 6.]]), indices=tensor([[1, 1, 1]]))
```

▼ 其他高阶操作

- where
- gather

▼ 自动求导机制

- `torch.autograd.backward(tensors, grad_tensors=None, retain_graph=False, create_graph=None)`：反向求导
  - `tensors`: 用于求导的张量，如 loss

- `grad_tensors`：当有多个 loss 混合需要计算梯度时，设置每个 loss 的权重
- `retain_graph`：保存计算图
- `create_graph`：创建导数计算图，用于高阶求导

```
x = torch.tensor([3.], requires_grad=True)
y = torch.pow(x, 2)     # y = x**2
# 如果需要求 2 阶导，需要设置 create_graph=True，让一阶导数 grad_1 也拥有计算图
grad_1 = torch.autograd.grad(y, x, create_graph=True)   # grad_1 = dy/dx = 2x = 2 * 3 = 6
print(grad_1)
# 这里求 2 阶导
grad_2 = torch.autograd.grad(grad_1[0], x)              # grad_2 = d(dy/dx)/dx = d(2x)/dx = 2
print(grad_2)

'''
(tensor([6.], grad_fn=<MulBackward0>),)
(tensor([2.]),)
'''
```

- `torch.aotograd.grad(outputs, inputs, grad_outputs=None, retain_graph=False, create_graph=None)`：**outputs 对 inputs 求取梯度**（返回一个 tuple，取出第 0 个元素为梯度计算结果）

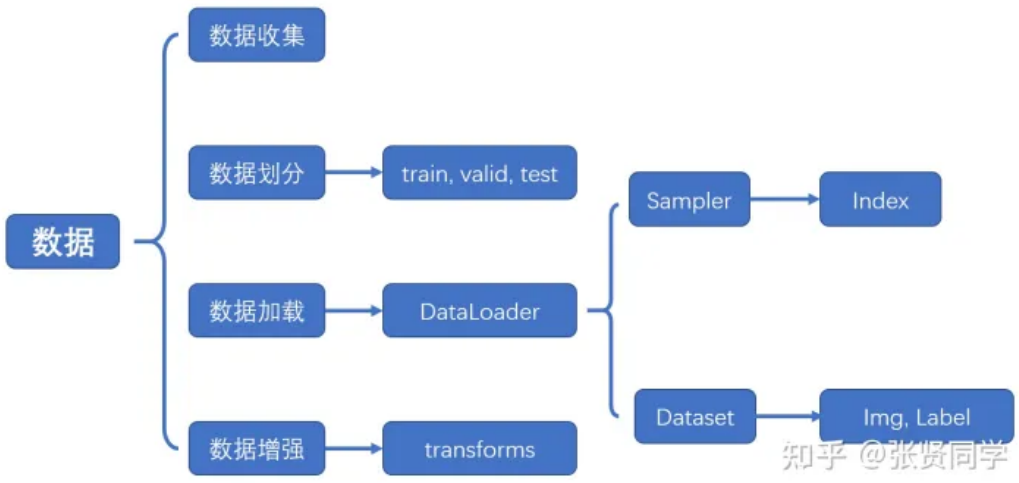
▼ 注意

1. PyTorch 采用动态图机制，默认每次反向传播之后都会释放计算图，因此默认情况下**连续**两次调用 backward 会报错
2. 每次反向求导时，计算的梯度不会自动清零，若多次迭代计算而梯度没有清零，那么梯度会在前一次的基础上叠加；因此每次调用 backward 后，记得将梯度清零：`a.grad.zero_()` 或 `optimizer.zero_grad()`
3. 依赖于叶子节点的节点，requires_grad 属性默认为 True
4. 叶子节点不可执行 inplace 操作

> 以加法来说，inplace 操作有`a += x`，`a.add_(x)`，改变后的值和原来的值内存地址是同一个。非inplace 操作有`a = a + x`，`a.add(x)`，改变后的值和原来的值内存地址不是同一个。

如果在反向传播之前 inplace 改变了叶子节点的值，再执行 backward 会报错（这是因为在进行前向传播时，计算图中依赖于叶子节点的那些节点，会记录叶子节点的地址，在反向传播时就会利用叶子节点的地址所记录的值来计算梯度）

数据处理



▼ `DataLoader` 和 `Dataset`

- `torch.utils.Dataset`：所有自定义数据集的**基类**，需要重写下列方法：
  - `__getitem(index)__`：接收一个索引，返回索引对应的**样本**和**标签**
  - `__len()__`：返回所有样本的数量

> 数据读取包括下列三方面：

> - 读取哪些数据：每个 Iteration 读取一个 Batchsize 大小的数据，每个 Iteration 应该读取哪些数据。
> - 从哪里读取数据：如何找到硬盘中的数据，应该在哪里设置文件路径参数
> - 如何读取数据：不同的文件需要使用不同的读取方法和库。

- `torch.utils.data.DataLoader(dataset, batch_size=1, shuffle=False, sampler=None, batch_sampler=None, num_workers=0, drop_last=False)`：数据加载器
  - sampler : 自定义从数据集中采样的策略，如果制定了采样策略， shuffle 则必须为 False（所有定义了 `__len__` 的 *`Iterable`* 都可以作为 sampler）
  - num_workers : 使用线程的数量，当为0时数据直接加载到主程序，默认为0
  - drop_last:布尔类型，为 True 时将会把最后不足 batch_size 的数据丢掉，为F将会把剩余的数据作为最后一小组

> Epoch: 所有训练样本都已经输入到模型中，称为一个 Epoch
>
> Iteration: 一批样本输入到模型中，称为一个 Iteration
>
> Batch size: 批大小，决定一个 iteration 有多少样本，也决定了一个 Epoch 有多少个 Iteration

  - `DataLoader` 迭代器源码：

```
def __iter__(self):
    if self.num_workers == 0:
        return _SingleProcessDataLoaderIter(self)
    else:
        return _MultiProcessingDataLoaderIter(self)
```

对单进程，在 `_SingleProcessDataLoaderIter` 里只有一个方法`_next_data()`，如下：

```
def _next_data(self):
   index = self._next_index()  # may raise StopIteration
   data = self._dataset_fetcher.fetch(index)  # may raise StopIteration
   if self._pin_memory:
      data = _utils.pin_memory.pin_memory(data)
   return data
```

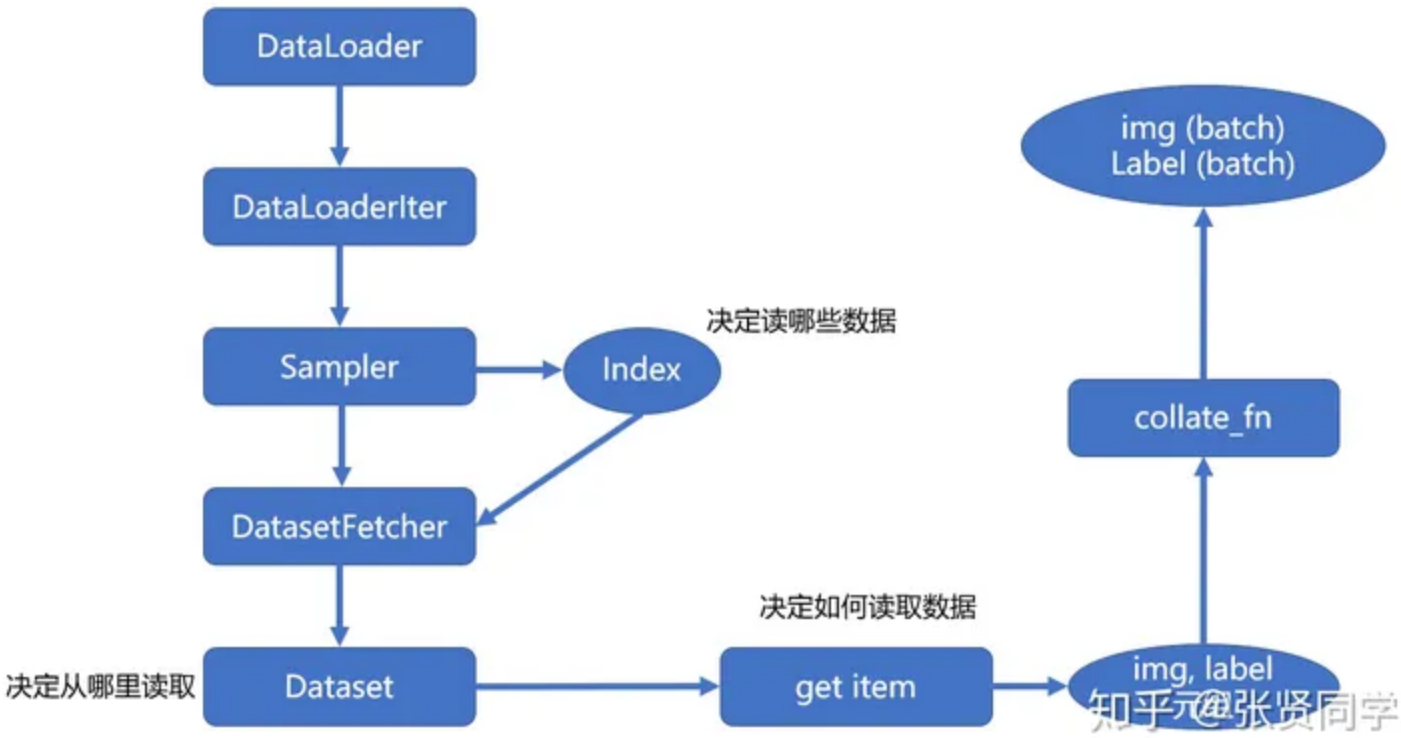
在该方法中，`self._next_index()` 是**获取一个 batchsize 大小的 index 列表**，代码如下：

```
def _next_index(self):
    return next(self._sampler_iter)  # may raise StopIteration
```

其中调用的 `sampler` 类的 `__iter__()` 方法返回 **batch_size 大小的随机 index 列表**

在第二行中调用了 `self._dataset_fetcher.fetch(index)` 获取数据。这里会调用 `_MapDatasetFetcher` 中的 `fetch()` 函数：

```
def fetch(self, possibly_batched_index):
 if self.auto_collation:
  data = [self.dataset[idx] for idx in possibly_batched_index]
 else:
  data = self.dataset[possibly_batched_index]
 return self.collate_fn(data)
```



▼ 数据处理与数据增强

▼ `torchvision` 库简介

- `torchvision.transforms`：包括常用图像预处理方法，**可用在 Dataset 类初始化**中，如：
  - 数据中心化
  - 数据标准化
  - 缩放
  - 裁剪
  - 旋转
  - 翻转
  - 填充
  - 噪声添加
  - 灰度变换
  - 线性变换
  - 仿射变换
  - 亮度、饱和度以及对比度变换。
- `torchvision.datasets`：包含常用数据集如 mnist、CIFAR-10、ImageNet 等
- `torchvision.models`：包含常用预训练模型，如 AlexNet、VGG、ResNet、GoogleNet 等

▼ `transforms` 中的图像处理方法

▼ 裁剪与填充

- `transforms.CenterCrop(size)`：根据给定尺寸裁剪出图像中心（若裁剪的 size 比原图大，会填充值为 0 的像素）
- `transforms.Pad(padding, fill=0, padding_mode='constant')`：边界填充
  - padding : 设置填充大小
    - 当为 a 时，上下左右均填充 a 个像素
    - 当为 (a, b) 时，左右填充 a 个像素，上下填充 b 个像素
    - 当为 (a, b, c, d) 时，左上右下分别填充 a，b，c，d
  - pad_if_need : 当图片小于设置的 size，是否填充
  - padding_mode:
    - constant: 像素值由 fill 设定
    - edge: 像素值由图像边缘像素设定

    - fill : 当 padding_mode 为 constant 时，设置填充的像素值（可以为三元 tuple，表示 RGB 三元组）
- `transforms.RandomCrop(size, padding=None, pad_if_need=False, fill=0, padding_mode='constant')`：随机裁剪出尺寸为 size 的图片；如果 padding 不为 None，则先 padding 再裁剪
- `transforms.RandomResizedCrop(size, scale=(0.08, 1.0), ratio=(0.75, 1.3333333333333333), interpolation=2)`
    - size: 裁剪的图片尺寸
    - scale: 随机缩放面积比例，默认随机选取 (0.08, 1) 之间的一个数（放大）
    - ratio: 随机长宽比，默认随机选取 (3/4, 4/3) 之间的一个数。因为超过这个比例会有明显的失真
    - interpolation: 当裁剪出来的图片小于 size 时，就要使用插值方法 resize
        - PIL.Image.NEAREST
        - PIL.Image.BILINEAR
        - PIL.Image.BICUBIC

▼ 翻转旋转与仿射变换

- `transforms.RandomHorizontalFlip(p)` / `transforms.RandomVerticalFlip(p)`：根据给定概率，在水平或垂直方向翻转图片
- `transforms.RandomRotation(degrees, center=None, expand=False)`：随机旋转
    - degrees : 旋转角度
        - 当为 a 时，在 (-a, a) 之间随机选择旋转角度
        - 当为 (a, b) 时，在 (a, b) 之间随机选择旋转角度
    - center : 旋转点设置，是坐标，默认中心旋转,如设置左上角为 (0, 0)
    - expand : 是否扩大矩形框，以保持原图信息。根据中心旋转点计算扩大后的图片。如果旋转点不是中心，即使设置 expand = True，还是会有部分信息丢失
- `transforms.RandomAffine(degrees, translate=None, scale=None, shear=None, resample=False, fillcolor=0)`：随机一般仿射变换，囊括五种基本操作——翻转、旋转、平移、缩放、错切
    - degree: 旋转角度设置
    - translate: 平移区间设置，如 (a, b)，a 设置宽 (width)，b 设置高 (height)，图像在宽维度平移的区间为 image_width * [-a, +a]，高同理
    - scale: 缩放比例，以面积为单位
    - fillcolor: 填充颜色设置
    - shear: 错切角度设置，有水平错切和垂直错切
        - 若为 a，则仅在 x 轴错切，在 (-a, a) 之间随机选择错切角度
        - 若为 (a, b)，x 轴在 (-a, a) 之间随机选择错切角度，y 轴在 (-b, b) 之间随机选择错切角度
        - 若为 (a, b, c, d)，x 轴在 (a, b) 之间随机选择错切角度，y 轴在 (c, d) 之间随机选择错切角度

▼ 颜色调整与随机遮挡

- `transforms.ColorJitter(brightness=0, contrast=0, saturation=0, hue=0)`：调整亮度/对比度/饱和度/色相
    - brightness、contrast、saturation 参数
        - 当为 a 时，从 [max(0, 1-a), 1+a] 中随机选择
        - 当为 (a, b) 时，从 [a, b] 中选择
    - hue : 色相参数（介于 0 到 0.5）
        - 当为 a 时，从 [-a, a] 中选择参数
        - 当为 (a, b) 时，从 [a, b] 中选择参数
- `transforms.RandomGrayscale(p=0.1, num_output_channels=1)`：根据指定概率将图片转为灰度图
    - num_output_channels : 输出的通道数。只能设置为 1 或者 3（如果在后面使用了 `transforms.Normalize`，则必须设为 3，因为 `transforms.Normalize` 只能接收 3 通道的输入）
- `transforms.RandomErasing(p=0.5, scale=(0.02, 0.33), ratio=(0.3, 3.3), value=0, inplace=False)`：根据给定概率对图像随机遮挡

    - scale: 遮挡区域的面积。如(a, b)，则会随机选择 (a, b) 中的一个遮挡比例
    - ratio: 遮挡区域长宽比。如(a, b)，则会随机选择 (a, b) 中的一个长宽比
    - value: 设置遮挡区域的像素值。(R, G, B) 或者 Gray，或者任意字符串（此时会用高斯噪声遮挡）。若之前执行了`transforms.ToTensor()`，像素值归一化到了 0~1 之间，因此这里设置的 (R, G, B) 要除以 255

▼ 特殊操作

- `transforms.Compose([list])`：当需要多个 `transforms` 操作时，需要作为一个 **list** 包装在 `transforms.Compose` 中
- `transforms.ToTensor()`：把图片转换为张量，**同时进行归一化操作**，把每个通道 0~255 的值**归一化为 0~1 [C, H, W]**
- `transforms.Normalize(mean, std, inplace=False)`：**逐 channel** 地对图像进行标准化 output = (input - mean) / std
- `transforms.Lambda([lambda function])`：自定义变换

模型构建与模型训练

▼ `nn.Module` 类

- 必须实现 `forward()` 函数
- 两种 mode：`model.train()` / `model.eval()`
    - 注意与 `requires_grad` 毫无关系，它仅仅是对在 train 和 eval 模式下表现不同的模块如 Dropout 和 BatchNorm 有用！

▼ 模块与参数管理

- 获取 `modules` (包括 `children`) `parameters` `buffers` 迭代器：不带 named_ 返回 *Tensor* 的迭代器，带 named_ 的返回 *Tuple[str, Tensor]* 的迭代器
    - `modules()` `named_modules(prefix='')`：返回由浅入深遍历所有子模块的迭代器
    - `children()` `named_children(prefix='')`：返回遍历所有一级子模块的迭代器
    - `parameters(recurse=True)` `named_parameters(prefix='', recurse=True)`：返回所有参数（默认递归迭代所有子模块）
    - `buffers(recurse=True)` `named_buffers(prefix='', recurse=True)`：返回模块的 buffers（模型状态参数，不用梯度下降更新）
- 根据名字获取 `submodule` `parameter` `buffer`：`get_***(target)`

  例如模块 A：

```
A(
    (net_b): Module(
        (net_c): Module(
            (conv): Conv2d(16, 33, kernel_size=(3, 3), stride=(2, 2))
        )
        (linear): Linear(in_features=100, out_features=200, bias=True)
    )
)
```

可以通过 `get_submodule("net_b.linear")` `get_submodule("net_b.net_c.conv")` 等获取相应子模块

- 注册 `module` `parameter` `buffer`：`register_***(name, [content])`

- 更改参数类型/设备：`model.type(dst_type)` 或 `model.to(type/device)`，包括 `model.half()` `model.cuda()` 等
- 载入模型参数：`model.load_state_dict(state_dict)`

▼ 模型容器（同样继承自 `nn.Module`）

- `nn.Sequetial`：顺序性，按照顺序包装多个网络层，常用于 block 构建
- `nn.ModuleList`：迭代性，像 python 的 list 一样包装多个网络层，包含 `append()` `extend()` `insert()` 等方法，常用于大量重复网络构建，通过 for 循环实现重复构建
- `nn.ModuleDict`：索引性，像 python 的 dict一样包装多个网络层，通过 (key, value) 的方式为每个网络层指定名称，常用于可选择的网络层

▼ 模型组件

▼ 卷积层

- 二维卷积：`nn.Cov2d(self, in_channels, out_channels, kernel_size, stride=1, padding=1, dilation=1, groups=1, bias=True, padding_mode='zeros')`
  - in_channels：输入通道数
  - out_channels：输出通道数，等价于卷积核个数
  - kernel_size：卷积核尺寸
  - stride：步长
  - padding：填充宽度，主要是为了调整输出的特征图大小，一般把 padding 设置合适的值后，保持输入和输出的图像尺寸不变。
  - dilation：空洞卷积大小，默认为 1，这时是标准卷积，常用于图像分割任务中，主要是为了提升感受野
  - groups：分组卷积设置，主要是为了模型的轻量化，如在 ShuffleNet、MobileNet、SqueezeNet 中用到
  - bias：偏置

完整版卷积尺寸计算考虑了空洞卷积，假设输入图片大小为 $I \times I$，卷积核大小为 $k \times k$，stride 为 s，padding 的像素数为 p，dilation 为 d，图片经过卷积之后的尺寸 O 如下：。

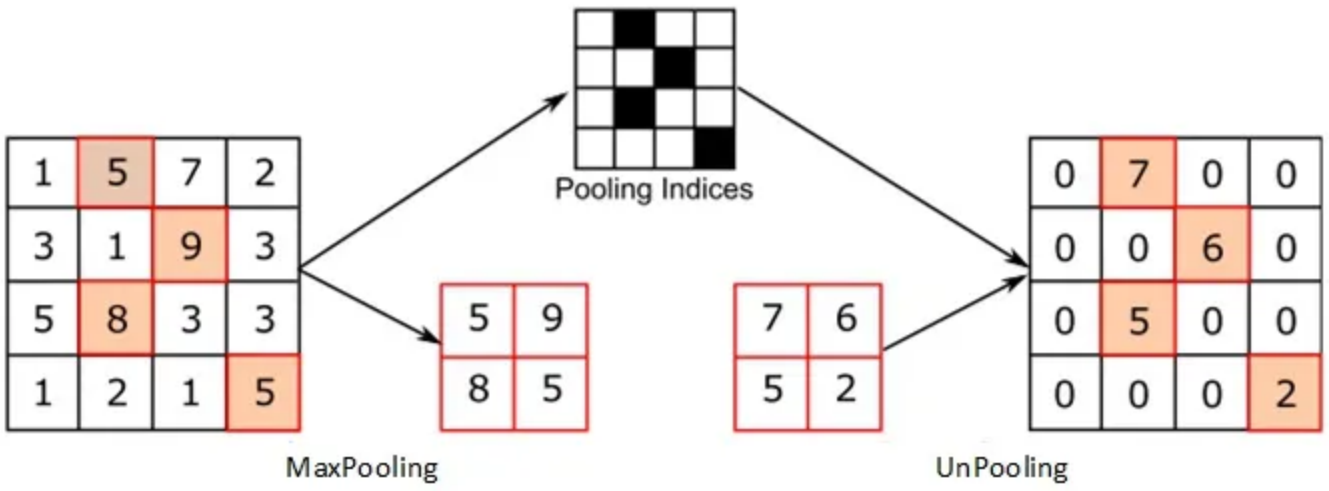
$$O = \frac{I - d \times (k-1) + 2 \times p - 1}{s} + 1$$

- 转置卷积：`nn.ConvTranspose2d(self, in_channels, out_channels, kernel_size, stride=1, padding=0, output_padding=0, groups=1, bias=True, dilation=1, padding_mode='zeros')`

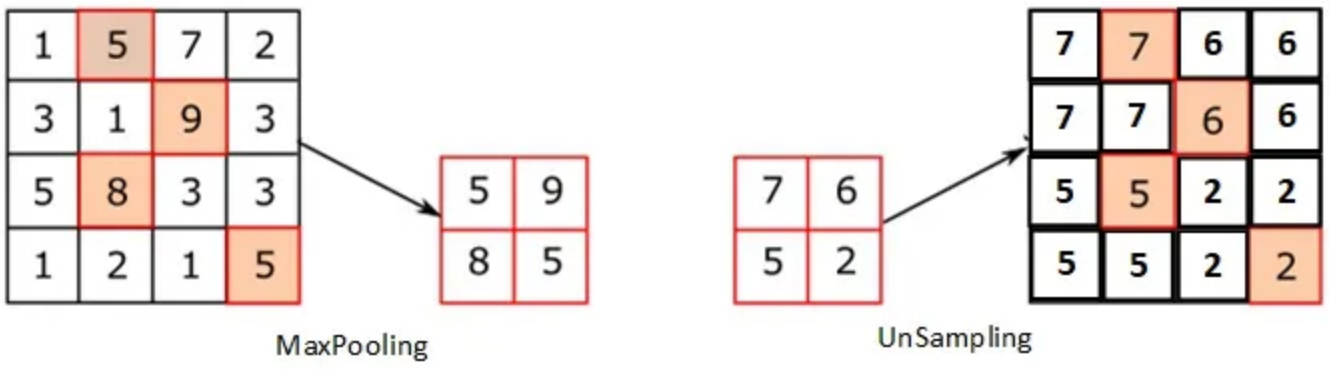
▼ 池化层

- 最大池化：`nn.MaxPool2d(kernel_size, stride=None, padding=0, dilation=1, return_indices=False, ceil_mode=False)`
  - kernel_size : 池化核尺寸（无重叠，即通常 stride=kernel_size）
  - return_indices：为 True 时，返回最大池化所使用的像素的索引，这些记录的索引通常在反最大池化时使用，把小的特征图反池化到大的特征图时，每一个像素放在哪个位置

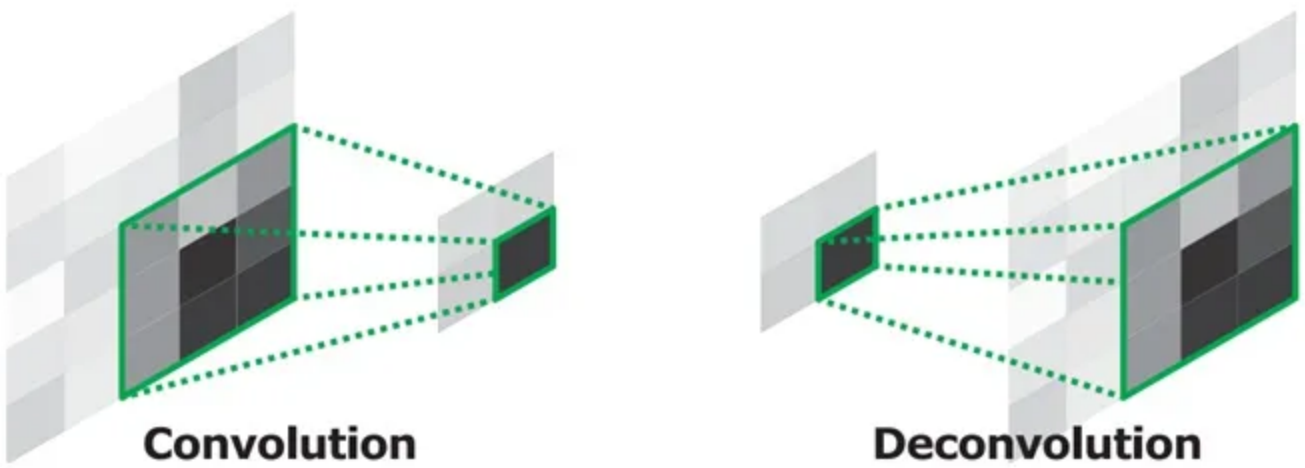
  下图 (a) 表示反池化，(b) 表示上采样，(c) 表示反卷积：
- 平均池化：`nn.AvgPool2d(...)`
- 最大值反池化：`nn.MaxUnpool2d(kernel_size, stride=None, padding=0)`



(a)



(b)



(c)

▼ `nn.init` 参数初始化

- 目的：保持每一层输出的方差不能太大也不能太小
- 纯线性层：正态初始化，标准差 $\sigma(W)=\sqrt{\frac{1}{n}}$
- 饱和激活函数（sigmoid/tanh）：Xavier 初始化

  Xavier 是 2010 年提出的，针对有非线性激活函数时的权值初始化方法，目标是保持数据的方差维持在 1 左右，主要针对饱和激活函数如 sigmoid 和 tanh 等。同时考虑前向传播和反向传播，需要满足两个等式：$\boldsymbol{n_i} * \boldsymbol{D(W)} = \mathbf{1}$ 和 $\boldsymbol{n_{i+1}} * \boldsymbol{D(W)} = \mathbf{1}$，可得：$D(W)=\frac{2}{n_i+n_{i+1}}$。为了使 Xavier 方法初始化的权值服从均匀分布，假设 W 服从均匀分布 $U[-a,a]$，那么方差 $D(W)=\frac{(-a-a)^2}{12}=\frac{(2a)^2}{12}=\frac{a^2}{3}$，令 $\frac{2}{n_i+n_{i+1}}=\frac{a^2}{3}$，解得：$\boldsymbol{a}=\frac{\sqrt{6}}{\sqrt{n_i+n_{i+1}}}$，所以 W 服从分布 $U\left[-\frac{\sqrt{6}}{\sqrt{n_i+n_{i+1}}},\frac{\sqrt{6}}{\sqrt{n_i+n_{i+1}}}\right]$
- ReLU 激活函数：Kaiming 初始化

▼ 损失函数（同样继承 `nn.Module`，因此也可视作一个网络层）

▼ 基本概念

- 损失函数（Loss Function）：**一个**样本的模型输出与真实标签的差异
- 代价函数（Cost Function）：整个样本集的模型输出与真实标签的差异，是所有样本损失函数的平均值
- 目标函数（Objective Function）：代价函数加上正则项

[PyTorch 学习笔记] 4.2 损失函数 - 知乎 (zhihu.com)

▼ `torch.optim` 优化器

▼ 基本方法

- 初始化：给定待优化的模型参数、学习率
- `optimizer.step()`：进行一步迭代
- `optimizer.zero_grad()`：将所有待优化的张量梯度置零
- `optimizer.state_dict()` `optimizer.load_state_dict(state_dict)`：获取/填入优化器的 state_dict

▼ 常用优化器

- `optim.SGD(params, lr, momentum=0, dampening=0, weight_decay=0)`
  - params：管理的参数组
  - lr：初始学习率
  - momentum：动量系数
  - weight_decay：L2 正则化系数
- `torch.optim.Adam(params, lr=0.001, betas=(0.9, 0.999), eps=1e-08, weight_decay=0)`

▼ 正则化

▼ 模型其他操作