Pytorch神经网络基础

广播机制

除矩阵乘法（严格的形状匹配才行）以外，其他的张量加减乘除等都需要广播机制。引入：`[[1, 2, 3]]`的`torch.Size([1, 3])`，是一行三列的数据；`[1, 2, 3]`的`torch.Size([3])`，是维度为三的数据。也就是有几个方括号就是几维的数据。

假设有两个张量`A`和`B`，它们的形状分别是`(a, b, c, d)`和`(x, y, z, w)`，简单来说广播机制就是**维度从右向左，只要两个张量中有一个数据维度大小是1**，就可以进行计算（以上的关键是必须要有一个维度是1）。具体运算规则如下：

- 两个张量的维度数量不同（`(a, b, c, d)`和`(x, y, z)`）：
  - 维度较少的张量会在**最左边**增加大小为`1`的维度，直到它们的维度数量相同。
  - 例子：`(x, y, z)`）变成`(1, x, y, z)`），接下来按下面的操作广播就行。
- 两个张良的维度数量相同（`(a, b, c, d)`和`(x, y, z, w)`）：
  - **从右向左**（最后一个维度开始）比较两个张量的形状：
    - 如果维度相同，则可以直接进行计算。
    - 如果一个维度是`1`，则会扩展（复制）这个维度 使其匹配另一个张量的大小。
    - 如果维度不同且不等于`1`，则会报错。

比如说`(1, 2, 1, 3)`和`(3, 1, 2, 1)`就可以直接进行计算。

改变张量形状或维度的函数

1. `view()`

- `view()` 用于 **改变张量的形状**，它返回一个新的张量，**但数据共享**。如果你想改变张量的维度而不改变数据，可以使用 `view()`。
- 必须保证在调用 `view()` 时，新形状的元素总数与原张量的元素总数一致。
- 使用这个函数之前，要求**数据是连续存储**的，否则需要先`contiguous()`（在数据进行**转置/改变维度顺序**等操作后，数据存储变得不连续，这个函数的作用就是将其变得连续）后再`view()`。而`reshape()`函数也是用于改变张量的形状，但与`view()`不同，不需要数据连续存储，直接就可以改变张量形状。

```
import torch
x = torch.randn(4, 4)
x_reshaped = x.view(16)  # 转换为一维张量
print(x_reshaped.shape)  # 输出: torch.Size([16])
```

2. `reshape()`

- `reshape()` 与 `view()` 类似，**功能相同**，但 `reshape()` 会在需要时**返回新的内存**。它比 `view()` 更加灵活，因为它会自动处理一些复杂情况。
- 当你调用 `reshape()` 时，如果无法返回原始张量的共享内存（例如在某些情况下原始张量存储的内存不可访问），它会返回一个新的张量。

```
x_reshaped = x.reshape(8, 2)
print(x_reshaped.shape)  # 输出: torch.Size([8, 2])
```

3. `unsqueeze()`

- `unsqueeze()` 用于 **在指定的位置插入一个大小为 1 的维度**，增加一个新的轴。
- 例如，如果你有一个形状为 `(3, 4)` 的张量，使用 `unsqueeze(0)` 会将它转换为 `(1, 3, 4)`，即在第一个维度插入一个大小为 1 的维度。

```
x = torch.randn(3, 4)
x_unsqueezed = x.unsqueeze(0)  # 在第一个维度添加一个维度
print(x_unsqueezed.shape)  # 输出: torch.Size([1, 3, 4])
```

4. `squeeze()`

- `squeeze()` 用于 **删除张量中大小为 1 的维度。**
- 例如，如果你有一个形状为 `(1, 3, 1, 4)` 的张量，调用 `squeeze()` 后会去除所有大小为 1 的维度，结果形状会变成 `(3, 4)`。

```
x = torch.randn(1, 3, 1, 4)
x_squeezed = x.squeeze()  # 去除大小为1的维度
print(x_squeezed.shape)  # 输出: torch.Size([3, 4])
```

5. `transpose()` / `T`

- `transpose()` 用于 **交换两个维度**。如果你有一个形状为 `(m, n)` 的张量，使用 `transpose(0, 1)` 或 `x.T` 可以交换其维度，得到形状为 `(n, m)` 的张量。

```
x = torch.randn(3, 2)
x_transposed = x.transpose(0, 1)  # 交换第0维和第1维
print(x_transposed.shape)  # 输出: torch.Size([2, 3])
```

或者：

```
x_transposed = x.T  # 使用 T 属性交换维度
print(x_transposed.shape)  # 输出: torch.Size([2, 3])
```

6. `permute()`

- `permute()` 用于 **重新排列张量的维度顺序**。它返回一个新的张量，通过指定每个维度的顺序来改变张量的形状。
- 例如，`permute(1, 0)` 可以交换 `0` 和 `1` 维度，适用于更多维度的情况。

```
x = torch.randn(2, 3, 4)
x_permuted = x.permute(2, 0, 1)  # 改变维度顺序
print(x_permuted.shape)  # 输出: torch.Size([4, 2, 3])
```

7. `flatten()`

- `flatten()` 用于 **将多维张量展平为一维**，可以指定开始和结束的维度，选择哪些维度进行展平。
- 例如，`flatten(1)` 会将第1维（索引为1）及其之后的维度展平，将后面的维度数值相乘就得到这个值。

```
x = torch.randn(2, 3, 4)
x_flattened = x.flatten(1)  # 从第1维开始展平
print(x_flattened.shape)  # 输出: torch.Size([2, 12])
```

8. `expand()`

- `expand()` 用于 **扩展张量的尺寸**（仅扩张尺寸，不扩张维度），沿着指定的维度进行扩展，扩展后的维度大小为 `1` 的维度将会广播。
- `expand()` 并不会实际复制数据，而是通过 **共享内存**的方式扩展张量。

```
x = torch.randn(1, 3, 1)
x_expanded = x.expand(4, 3, 5)  # 扩展为(4, 3, 5)
print(x_expanded.shape)  # 输出: torch.Size([4, 3, 5])
```

9. `expand_as()`

- `expand_as()` 是 `expand()` 的一种变体，它会将张量扩展为和另一个张量相同的形状。

```
x = torch.randn(1, 3, 1)
y = torch.randn(4, 3, 5)
x_expanded = x.expand_as(y)  # 扩展为y的形状
print(x_expanded.shape)  # 输出: torch.Size([4, 3, 5])
```

10. `cat()`

- `cat()` 用于 **连接多个张量**，可以在指定的维度上将它们拼接在一起。

```
x = torch.randn(2, 3)
y = torch.randn(2, 3)
z = torch.cat((x, y), dim=0)  # 在第0维拼接
print(z.shape)  # 输出: torch.Size([4, 3])
```

11. `stack()`

- `stack()` 用于 **沿着新的维度将多个张量堆叠**，与 `cat()` 不同，`stack()` 会在新的维度上添加一个维度。

```
x = torch.randn(2, 3)
y = torch.randn(2, 3)
z = torch.stack((x, y), dim=0)  # 在第0维上堆叠
print(z.shape)  # 输出: torch.Size([2, 2, 3])
```

一些常用函数：

1. `enumerate`

- `enumerate` 函数是python中的内置函数，用于在迭代过程中同时获取元素的值和索引：

```
# 示例列表
fruits = ['apple', 'banana', 'cherry']

# 使用enumerate
for index, fruit in enumerate(fruits):
    print(index, fruit)

# 输出:
# 0 apple
# 1 banana
# 2 cherry
```

2. `sum`

- `sum` 函数用于tensor（张量）中的所有元素进行求和：

```
import torch

tensor = torch.tensor([[1, 2], [3, 4]])
result = torch.sum(tensor)  # 求和所有元素
result_dim0 = torch.sum(tensor, dim=0)  # 沿着第0维求和（列）
result_dim1 = torch.sum(tensor, dim=1)  # 沿着第1维求和（行）
```

- *有两个比较重要的作用，一是解包列表或元组；二是收集位置参数。

```
#解包列表或元组
def add(a, b):
    return a + b

numbers = (2, 3)
result = add(*numbers)  # 解包，结果为 5

#收集位置参数
def sum_all(*args):
    return sum(args)

result = sum_all(1, 2, 3)  # 结果为 6
```

3. `torch.normal`

- `torch.normal`函数会生成服从**正态分布**（Normal Distribution）的随机数：

```
import torch

# 生成一个 3x4 的矩阵，均值为 0，标准差为 1
mean = 0.0
std = 1.0
size = (3, 4)
weights = torch.normal(mean, std, size)
```

4. `torch.randn`和`torch.rand`

- `torch.randn` 是 PyTorch 中用于生成随机数的函数，它生成服从标准正态分布（均值为 0，标准差为 1）的随机数。（**正态分布** （也称高斯分布）可以具有任意的均值和标准差，而 **标准正态分布** 始终具有均值为 0 和标准差为 1。）

```
import torch

# 生成一个 3x4 的张量，包含随机数
random_tensor = torch.randn(3, 4)
print(random_tensor)
```

- `torch.rand` 是 PyTorch 中用于生成服从**均匀分布**的随机数的函数。生成的随机数在[0,1)区间内均匀分布，常用于初始化模型权重或生成随机输入数据。

```
import torch

# 生成一个 3x4 的张量，包含均匀分布的随机数
random_tensor = torch.rand(3, 4)
print(random_tensor)
```

5. `torch.zeros`

- `torch.zeros` 用于生成一个指定形状的张量，所有元素均初始化为零。

```
import torch

# 创建一个 2x3 的全零张量
tensor1 = torch.zeros(2, 3)
print(tensor1)

# 创建一个 3D 张量，形状为 2x3x4
tensor2 = torch.zeros(2, 3, 4)
print(tensor2)

# 创建一个整型全零张量
tensor3 = torch.zeros(2, 3, dtype=torch.int32)
print(tensor3)

# 在 GPU 上创建全零张量（如果可用）
if torch.cuda.is_available():
    tensor4 = torch.zeros(2, 3, device='cuda')
 print(tensor4)
```

6. `torch.matmul`

- `torch.matmul` 用于计算矩阵的乘法，如果输入是一维张量，则计算内积；`torch.dot` 函数专门用于计算两个一维张量的内积。

```
import torch

# 示例张量
a = torch.tensor([1, 2, 3])
b = torch.tensor([4, 5, 6])

# 计算内积
result1 = torch.matmul(a, b)
result2 = torch.dot(a, b)
print(result1)              # 输出: tensor(32) (1*4 + 2*5 + 3*6)
print(result2)              # 输出: tensor(32)
```

7. `torch.mul`

- `torch.mul` 是 PyTorch 中的 **逐元素乘法函数（等价于*号）**，它的作用是对两个张量的每个对应位置进行相乘操作。一般mask操作都用这个函数。

```
a = torch.tensor([1, 2, 3])
b = torch.tensor([4, 5, 6])
result = torch.mul(a, b)
print(result)  # tensor([ 4, 10, 18])
```