

pytorch 使用帮助

torch.nn.Module

`torch.nn.Module()`

实现一个组件，是 pytorch 中所有组件——包括以下介绍的所有层——的基类。最外层的模型也应该继承该类

成员函数

- `__init__()`

初始化模型，模型或组件中包含的子组件必须在该函数中构建

- `forward(*input)`

前向传播，实现模型或组件的功能，将输入进行相应的操作并输出。一般通过直接调用类的实例，间接调用该函数

- `parameters(recuse=True)`

返回一个包含组件或模型参数的迭代器

参数

- `recurse (bool, 可选)` 返回的迭代器是否包含子组件或模型的参数

示例

```
import torch
import torch.nn as nn

# 定义并实现一个模型
class MyModel(nn.Module):
    def __init__(self, input_size, intermediate_size):
        super().__init__()

        # 依次构建模型的两个线性层，一个Sigmoid激活函数
        self.dense1 = nn.Linear(input_size, intermediate_size)
        self.sigmoid = nn.Sigmoid()
        self.dense2 = nn.Linear(intermediate_size, 1) # 要注意维数的匹配

    def forward(self, x):
        # 一层层调用模型的组件，实现前向传播
        x = self.dense1(x)
        x = self.sigmoid(x)
        x = self.dense2(x)
        return x
```

```
# 创建一个模型实例
model = MyModel(256, 512)

input_tensor = torch.randn(1, 256) # 产生大小为(1, 256)的随机张量

# 输入模型, 得到输出
output_tensor = model(input_tensor)
print(output_tensor.detach()) # tensor([[0.0394]]) 此为随机值
# detach: 返回一个数值上相等但没有梯度的张量
```

更多信息请参考[官方文档](#)

torch.nn.Conv2d

```
torch.nn.Conv2d(in_channels, out_channels, kernel_size, stride=1, padding=0, dilation=1, groups=1,
bias=True, padding_mode='zeros', device=None, dtype=None)
```

2D 卷积层

初始化参数

- `in_channels` (`int`) 输入图像应有的通道数 (或 feature 应有的维数)
- `out_channels` (`int`) 输出图像的通道数 (或 feature 的维数)
- `kernel_size` (`int` 或 `tuple`) 卷积核的大小
- `stride` (`int` 或 `tuple`, 可选) 卷积核滑动的步长
- `padding` (`int`, `tuple` 或 `str`, 可选) 当卷积核滑动到图像的角落, 无法容纳一个完整的窗口时, 可能需要对图像的边界进行扩展和填充。该参数指定填充大小
 - `'valid'` 不填充, 丢弃图像中无法容纳完整窗口的部分
 - `'same'` 自动计算填充大小, 保证输入和输出的图像尺寸相同。该选项只有在 `stride` 为 `1` 时才能使用
- `padding_mode` (`str`, 可选) 指定填充内容
 - `'zeros'` 填充 `0`
 - `'reflect'`
 - `'replicate'`
 - `'circular'`

reflect



replicate



circular



- *bias* (**bool**, 可选) 卷积核是否有可学习的偏置项

输入大小

$\text{\texttt{\$}}(\text{\texttt{\text{batch_size}}}, \text{\texttt{\text{channels}}_}\text{\texttt{\text{in}}}, \text{\texttt{\text{height}}}, \text{\texttt{\text{width}}})\text{\texttt{\$}}$

输出大小

$\text{\texttt{\$}}(\text{\texttt{\text{batch_size}}}, \text{\texttt{\text{channels}}}\text{\texttt{\text{out}}}, \text{\texttt{\text{height}}}\text{\texttt{\text{out}}}, \text{\texttt{\text{width}}_}\text{\texttt{\text{out}}})\text{\texttt{\$}}$

示例

```
import torch
from torch import nn

# 创建一个卷积核大小为3 x 3的卷积层，输入的通道数应为3，输出的通道数为8，尺寸与原图像的一致
conv = nn.Conv2d(3, 8, 3, padding="same")

input_tensor = torch.randn(1, 3, 50, 100)

# 对输入进行卷积操作
output_tensor = conv(input_tensor)
print(output_tensor.size()) # torch.Size([1, 8, 50, 100])
# size: 返回张量的大小
```

更多信息请参考[官方文档](#)

torch.nn.MaxPool2d

```
torch.nn.MaxPool2d(kernel_size, stride=None, padding=0, dilation=1, return_indices=False,
                    ceil_mode=False)
```

2D Max Pooling 层

初始化参数

- *kernel_size* (**int** 或 **tuple**) 滑动窗口的大小

- *stride* (`int`, `tuple` 或 `None`, 可选) 窗口滑动的步长。若为 `None`, 则步长为 *kernel_size*
- *padding* (`int` 或 `tuple`, 可选) 填充大小

输入大小

$(\text{batch_size}, \text{channels}, \text{height}, \text{width})$

输出大小

$(\text{batch_size}, \text{channels}, \text{height} \setminus \text{out}, \text{width} \setminus \text{out})$

更多信息请参考[官方文档](#)

torch.nn.AvgPool2d

```
torch.nn.AvgPool2d(kernel_size, stride=None, padding=0, ceil_mode=False, count_include_pad=True, divisor_override=None)
```

2D Average Pooling 层, 与Max Pooling 层类似, 但求的不是最值, 而是均值

初始化参数

同 MaxPool2d

输入大小

同 MaxPool2d

输出大小

同 MaxPool2d

更多信息请参考[官方文档](#)

Linear

```
torch.nn.Linear(in_features, out_features, bias=True, device=None, dtype=None)
```

线性层, 对输入做线性运算 $y = x A^T + b$, 可以作为 Fully Connected Feedforward Network 的一部分。若输入张量的阶数 > 2 , 则对最后一个维度做线性变换

初始化参数

- *in_features* (`int`) 输入 feature 应有的维数
- *out_features* (`int`) 输出 feature 的维数
- *bias* (`bool`, 可选) 卷积核是否有可学习的偏置项

输入大小

$(\dots, \text{features}_{\text{in}})$

输出大小

$\$(\text{dots}, \text{\text{features}}_ \text{\text{out}})\$$

示例

```
import torch
from torch import nn

# 创建一个线性层，输入feature的维数应为256，输出feature的维数为256，添加偏置项
dense = nn.Linear(256, 512, bias=True)

input_tensor = torch.randn(1, 50, 50, 256)

# 对输入进行线性变换
output_tensor = dense(input_tensor)
print(output_tensor.size()) # torch.Size([1, 50, 50, 512])
```

更多信息请参考[官方文档](#)

激活函数

```
torch.nn.Sigmoid()

torch.nn.Tanh()

torch.nn.ReLU(inplace=False)

torch.nn.Softmax(dim=None)
```

激活函数层，对输入执行一个非线性变换，可以作为 Fully Connected Feedforward Network 的一部分

更多信息请参考官方文档：[Sigmoid](#)、[Tanh](#)、[ReLU](#)、[Softmax](#)

CrossEntropyLoss

```
torch.nn.CrossEntropyLoss(weight=None, size_average=None, ignore_index=- 100, reduce=None,
reduction='mean', label_smoothing=0.0)
```

交叉熵损失函数层，先对输入求 Softmax，再求其与标签之间的交叉熵损失 \begin{align}

$$Z_n = \sum_{c=1}^{N_{\text{class}}} \exp(x_{n,c}) \quad \hat{p}_n = \frac{1}{Z_n} \left[\exp(x_{n,1}), \dots, \exp(x_{n,N_{\text{class}}}) \right] \quad \text{loss}_n = - \sum_{c=1}^{N_{\text{class}}} 1_{\{y_n=c\}} \cdot \log \hat{p}_{n,c}$$

\end{align}

初始化参数

- *reduction* (**str**, 可选) 对 batch 内所有样本结果的化简方式
 - 'none' 保留每个样本的损失
 - 'mean' 最终输出为样本损失的均值
 - 'sum' 最终输出为样本损失的和

输入大小

- $input$ $(\text{batch_size}, N, \text{class})$, 第二个维度表示样本对于每个类别的 logits (而非概率)
- $target$ (batch_size) , 表示类别标签, 范围为 $[0, N, \text{class})$

输出大小

若 $reduction$ 为 'none', 则与 $target$ 相同; 否则为一个标量

示例

```
import torch
from torch import nn

# 创建一个交叉熵损失函数层
criterion = nn.CrossEntropyLoss()

logits = torch.as_tensor(
    [[9.2, -3.5, 2.7],
     [6.3, 2.9, -1.4]]
) # 根据输入的列表创建张量, 返回的张量与输入的列表共享同一块内存
target = torch.as_tensor([0, 1])

loss = criterion(logits, target)
print(loss) # tensor(1.7174)
```

更多信息请参考[官方文档](#)

torch.Tensor.view

Tensor.view(*shape)

返回一个数值相同, 但形状不同的张量。新张量与原始张量共享同一块内存

示例

```
import torch

x = torch.randn(4, 4)

y = x.view(16)
print(y.size()) # torch.Size([16])
print(x.size()) # torch.Size([4, 4]), 不变

z = x.view(-1, 8)
print(z.size()) # torch.Size([2, 8]), -1所在的维度自动计算

w = x.view(2, 4, 2)
print(w.size()) # torch.Size([2, 4, 2])
```

更多信息请参考[官方文档](#)

torch.Tensor.backward

```
Tensor.backward(gradient=None, retain_graph=None, create_graph=False, inputs=None)
```

自动求出计算路径上所有张量对当前张量的梯度。通常对 loss 调用该函数，一般情况下 loss 需要是一个标量

示例

```
import torch

x = torch.as_tensor(
    [[0.0, 1.0],
     [2.0, 3.0]]
)
y.requires_grad_() # 由于x是直接创建的张量，而非模型中的参数，因此pytorch默认不会计算该张量的梯度。requires_grad_告诉pytorch需要计算x的梯度。以下y类似

y = torch.as_tensor(
    [[1.0, 2.0],
     [3.0, 4.0]]
)
y.requires_grad_()

loss = x * y # 按位置相乘，并非矩阵乘法
loss = torch.sum(loss) # 对张量中所有元素求和
loss.backward()
print(x.grad) # tensor([[1., 2.], [3., 4.]])
print(y.grad) # tensor([[0., 1.], [2., 3.]])
# grad: 返回张量的梯度
```

更多信息请参考[官方文档](#)

torch.optim.SGD

```
torch.optim.SGD(params, lr, momentum=0, dampening=0, weight_decay=0, nesterov=False, **kwargs, maximize=False)
```

SGD 优化器，使用随机梯度下降的方式更新模型参数

初始化参数

- *params* ([Iterable](#)) 包含所有需要更新参数的可迭代对象
- *lr* ([float](#)) 学习率
- *weight_decay* ([float](#), 可选) 该优化器自动添加 L2 正则项，该参数指定正则项的系数

成员函数

```
step(closure=None)
```

更新参数

```
zero_grad(set_to_none=False)
```

将模型参数的梯度设为0（反向传播前的准备，否则计算得到的梯度将会与原来的梯度相加）

示例

```
import torch
import torch.nn as nn
import torch.optim as optim

class ToyModel(nn.Module):
    def __init__(self):
        super().__init__()
        self.dense = nn.Linear(2, 1)

    def forward(self, x):
        return self.dense(x)

model = ToyModel()
criterion = nn.L1Loss() # 平均绝对误差
optimizer = optim.SGD(model.parameters(), lr=1e-2)
model.train()

train_data = torch.as_tensor([[ -2.0, 1.0], [ 2.0, 1.0], [ -3.0, 3.0], [ 1.0, 0.0]])
train_target = torch.as_tensor([ -0.5, 1.5, 0.0, 0.5])

# 更新100个epoch
for e in range(100):
    optimizer.zero_grad()

    output = model(train_data)
    loss = criterion(output.squeeze(-1), train_target) # squeeze(-1): 去掉最后一个
    # 维度
    loss.backward()

    optimizer.step()

# 验证模型
test_data = torch.as_tensor([[ 1.0, 2.0], [ 3.0, 4.0], [ -1.0, -2.0], [ -3.0, -4.0]])
test_target = torch.as_tensor([ 1.5, 3.5, -1.5, -3.5])

model.eval()
with torch.no_grad(): # no_grad: 被包裹的代码不计算梯度, 节省空间和时间
    output = model(test_data)
    loss = criterion(output.squeeze(-1), test_target)
    print(output) # tensor([[ 1.5245], [ 3.5845], [ -1.5643], [ -3.6243]])
    print(loss) # tensor(0.0744)

# 输出所有参数
for p in model.named_parameters():
```



```
    print(p)
'''
('dense.weight', Parameter containing:
tensor([[0.5156, 0.5144]], requires_grad=True))
('dense.bias', Parameter containing:
tensor([-0.0199], requires_grad=True))
'''
```

更多信息请参考[官方文档](#)