深度学习架构

第2章 Pytorch

本章大纲

- 1 PyTorch简介
- 2 PyTorch基本元素
  - 张量(Tensor)
  - 变量(Variable)
  - 神经网络模块(nn.Module)
- 3 常用模块
  - 线性（全连接）网络
  - 卷积神经网络
  - 池化模块
  - 激活模块
  - 循环神经网络

参考资料

- PyTorch官方文档
  https://pytorch.org/docs/stable/torch.html
- PyTorch 中文手册
  - https://pytorchbook.cn/chapter1/1.1-pytorch-introduction/
  - https://pytorch.panchuang.net

课后作业

- 根据GitHub上中文诗词爱好者收集的5万首唐诗数据集（https://github.com/chinese-poetry/chinese-poetry），用LSTM写出一首唐诗。

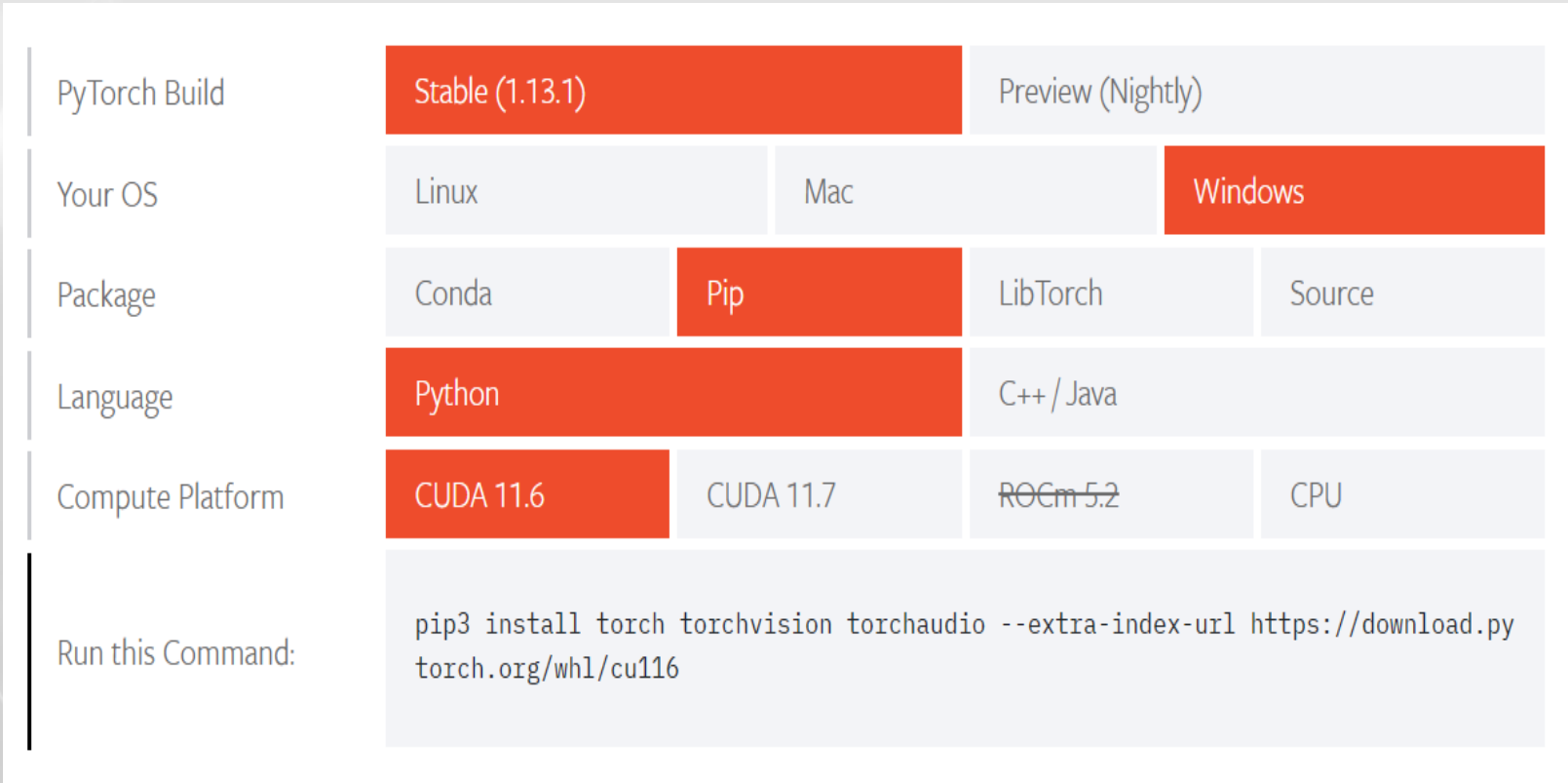
1. PyTorch简介

PyTorch 是一个基于 python 的科学计算包，主要用途为：

- 作为 NumPy 的替代品，可以利用 GPU 的性能进行高效计算
- 作为一个高灵活性，速度快的深度学习平台

安装

- https://pytorch.org/get-started/locally/



测试

```
import torch
x = torch.rand(5, 3)
print(x)

tensor([[0.3380, 0.3845, 0.3217],
        [0.8337, 0.9050, 0.2650],
        [0.2979, 0.7141, 0.9069],
        [0.1449, 0.1132, 0.1375],
        [0.4675, 0.3947, 0.1426]])
```

torch.cuda.is_available()

2. PyTorch基本元素

1. 张量(Tensor)
2. 变量(Variable)
3. 神经网络模块(nn.Module)

2.1 张量(Tensor)

- 张量是PyTorch中最基本的元素，相当于numpy.ndarray。两者的运算方式也如出一辙，在PyTorch中也可以相互转化
- 张量是一种专有的数据结构，与数组和矩阵非常相似
- 张量类似于NumPy的ndarrays，但tensor可以在GPU或其他硬件加速器上运行
- 张量和NumPy数组共享相同的底层内存，不需要复制数据
- 张量为自动微分进行了优化

Tensor初始化

- 直接初始化

```
x1 = torch.tensor([1, 2, 3])
```

- 从Numpy数组初始化

```
np_array = np.array([1, 2, 3])
x2 = torch.from_numpy(np_array)
```

- 从其他tensor初始化

```
x3 = torch.ones_like(x1)
x4 = torch.rand_like(x1, dtype=torch.float)
x5 = torch.full_like(x1, 5)
```

- 使用随机数和常数初始化

```
shape = (2,3,)
x6 = torch.rand(shape)
x7 = torch.ones(shape)
x8 = torch.zeros(shape)
x9 = torch.full(shape, 9)
```

Tensor转其它类型

- 转换为Numpy类型
  Tensor有numpy()函数可以转换为Numpy类型

```
>>>print(x1)
tensor([1,2,3])
>>>nx1 = x1.numpy()
>>>print(nx1)
[1 2 3]
```

- 转换为Python数值,只能转换单个Tensor数值为Python数值

```
>>>print(x1)
tensor([1,2,3])
>>>px1 = x1[0].item()
>>>print(px1)
1
```

Tensor性质

- Tensor具有形状、数据类型以存储设备三个属性，可分别用shape,dtype,device访问

```
>>>print(x1)
tensor([1,2,3])
>>>print(x1.shape)
torch.Size([3])
>>>print(x1.dtype)
torch.int64
>>>print(x1.device)
cpu
```

- Torch Tensor和 NumPy数组共享底层内存位置，因此当一个改变时，另外也会改变。

```
print(np_array)
[1,2,3]
print(x2)
tensor([1,2,3], dtype=torch.int32)
x2.add_(1)
print(np_array)
[2,3,4]
print(x2)
tensor([2,3,4], dtype=torch.int32)
```

在GPU运行Tensor

默认情况下，tensor是在CPU上创建的，可使用.to()方法将tensor移动到GPU上.

```
if torch.cuda.is_available():
  x1 = x1.to('cuda')
  print(x1.device)
```

- 原位操作
  将结果存储到操作数中的操作被称为原位操作,这些操作函数用后缀来“_”表示。

```
print(x3)
tensor([1,1,1])
x3.add_(10)
tensor([11,11,11])
print(x3)
tensor([11,11,11])
```

序列化

- 将Tensor保存为文件

```
torch.save(x1,"x1.file")
```

- 加载文件到Tensor

```
x1file = torch.load("x1.file")
```

自动微分

在训练神经网络时，最常使用的算法是反向传播(BP)算法。在这种算法中，参数（模型权重）是根据损失函数相对于给定参数的梯度来调整的。

为了计算这些梯度，PyTorch有一个内置的微分引擎，叫做torch.autograd。它支持对任何计算图的梯度进行自动计算。

自动微分示例

对线性操作z=w*x+b，当目标为y，损失函数为“二值交叉熵”时，计算其反向梯度。

```
import torch

x = torch.ones(5)
y = torch.zeros(3)
w = torch.randn(5, 3, requires_grad=True)
b = torch.randn(3, requires_grad=True)
z = torch.matmul(x, w)+b
loss = torch.nn.functional.binary_cross_entropy_with_logits(z, y)
```

为了优化神经网络中的参数权重，需要计算损失函数相对于参数的导数，即：

$$\frac{\partial loss}{\partial w}$$

$$\frac{\partial loss}{\partial b}$$

只要调用 loss.backward()，就可以从 w.grad 和 b.grad 中获取其梯度数值。

```
loss.backward()
print(w.grad)
print(b.grad)
```

```
>>> print(w.grad)
None
>>> print(b.grad)
None
>>> loss.backward()
>>> print(w.grad)
tensor([[0.3306, 0.0206, 0.0220],
        [0.3306, 0.0206, 0.0220],
        [0.3306, 0.0206, 0.0220],
        [0.3306, 0.0206, 0.0220],
        [0.3306, 0.0206, 0.0220]])
>>> print(b.grad)
tensor([0.3306, 0.0206, 0.0220])
```

2.2 变量(Variable)

- Tensor是PyTorch中对numpy.ndarray的替代品，但搭建神经网络时，还需要variable来构建计算图。Variable是对tensor的封装，是一个存放会变化的值的物理位置，这个值就是tensor。每个variable有3个属性：
  - variable.data: variable中tensor的值；
  - variable.grad: variable中tensor的梯度;
  - variable.gradfn: 指向Function对象，用于反向传播的梯度计算之用。

2.3 神经网络模块(nn.Module)

- nn是PyTorch中专门为神经网络设计的接口。
- nn.Module提供各种网络层的定义以及前向传播(forward)的方法。在定义自己的神经网络时，需要继承nn.Module类，并实现自己的forward方法

神经网络训练

神经网络训练过程包括以下步骤：

1.定义一个包含可训练参数的神经网络
2.迭代整个输入
3.通过神经网络处理输入
4.计算损失(loss)
5.反向传播梯度到神经网络的参数
6.更新网络的参数，典型更新方法：
weight = weight - learning_rate *gradient

定义神经网络示例

```
import torch
import torch.nn as nn
import torch.nn.functional as F


class Net(nn.Module):

    def __init__(self):
        super(Net, self).__init__()
        # 定义一个卷积操作：1个图像通道，6个输出通道，5x5卷积
        self.conv1 = nn.Conv2d(1, 6, 5)
        self.conv2 = nn.Conv2d(6, 16, 5)

        self.fc1 = nn.Linear(16 * 5 * 5, 120)
        self.fc2 = nn.Linear(120, 84)
        self.fc3 = nn.Linear(84, 10)
```

```
    def forward(self, x):
        # (2, 2)最大池化操作
        x = F.max_pool2d(F.relu(self.conv1(x)), (2, 2))
        # 当池化操作为方形时，可以用一个整数表示方形尺寸
        x = F.max_pool2d(F.relu(self.conv2(x)), 2)
        x = x.view(-1, self.num_flat_features(x))
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x
```

```
    def num_flat_features(self, x):
        size = x.size()[1:]  # 计算除通道以外的所有大小
        num_features = 1
        for s in size:
            num_features *= s
        return num_features
# 生成网络
net = Net()

print(net)
Net(
  (conv1): Conv2d(1, 6, kernel_size=(5, 5), stride=(1, 1))
  (conv2): Conv2d(6, 16, kernel_size=(5, 5), stride=(1, 1))
  (fc1): Linear(in_features=400, out_features=120, bias=True)
  (fc2): Linear(in_features=120, out_features=84, bias=True)
  (fc3): Linear(in_features=84, out_features=10, bias=True)
)
```

forward()为自定义的前馈函数，可以在前馈函数上使用任何张量操作。
而反向传播函数是被自动通过 autograd 定义的。

一个模型可训练的参数可以通过调用 net.parameters() 返回

```
params = list(net.parameters())
print(len(params))
10
print(params[0].size())  # 第一层卷积的权重系数
torch.Size([6, 1, 5, 5])
```

随机生成一个 32x32 的输入。

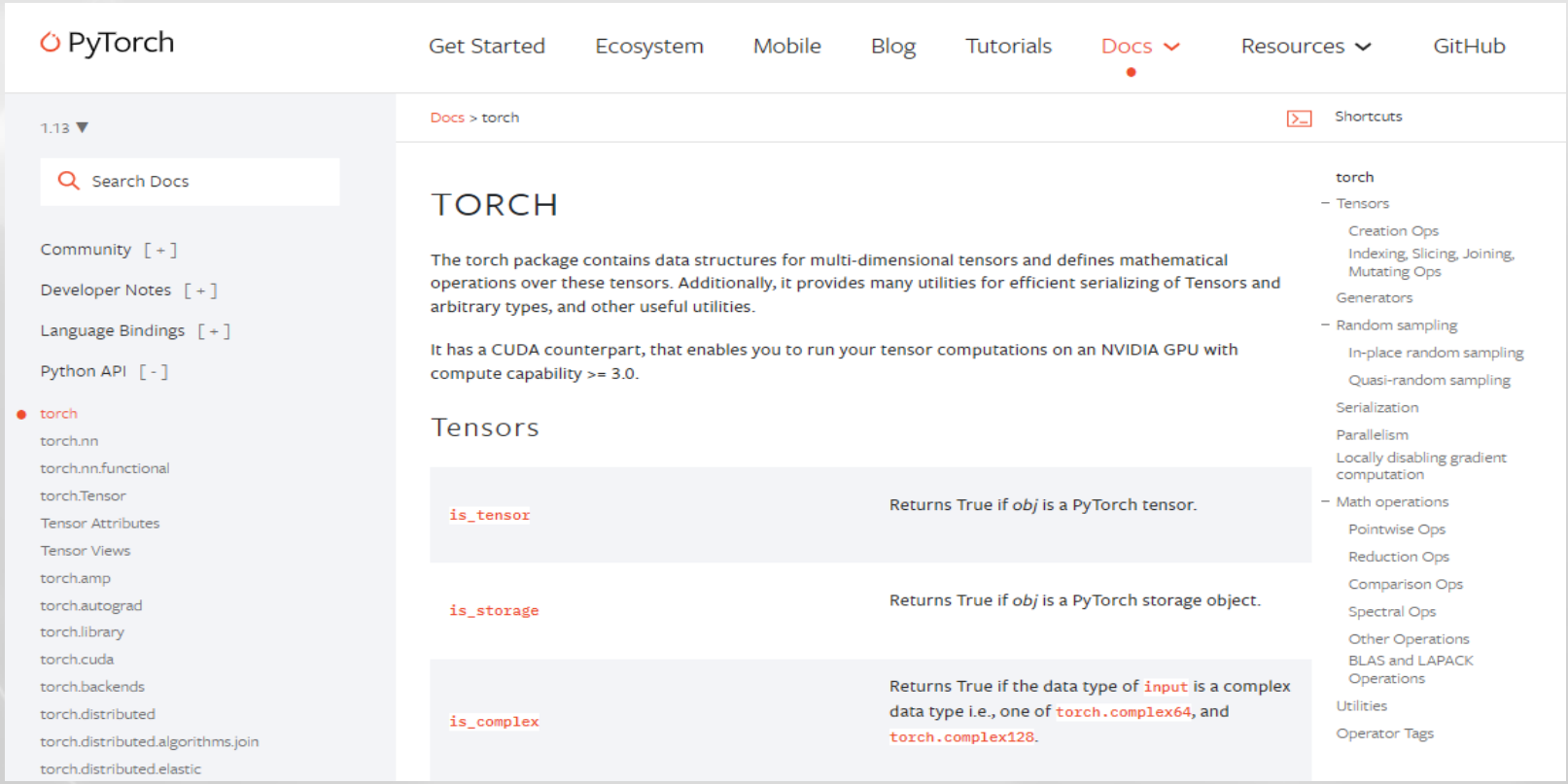
```
input = torch.randn(1, 1, 32, 32)
out = net(input)
print(out)
tensor([[-0.0233,  0.0159, -0.0249,  0.1413,  0.0663,  0.0297, -0.0940, -0.0135,
          0.1003, -0.0559]], grad_fn=<AddmmBackward>)
```

把所有参数梯度缓存器置零，用随机的梯度来反向传播

```
net.zero_grad()
out.backward(torch.randn(1, 10))
```

3. 常用模块

https://pytorch.org/docs/stable/torch.html



常用模块

- torch
  - Tensors
  - Creation Ops
  - Indexing, Slicing, Joining, Mutating Ops
- Generators
- Random sampling
  - In-place random sampling
  - Quasi-random sampling
- Serialization
- Parallelism

- Locally disabling gradient computation
- Math operations
  - Pointwise Ops
  - Reduction Ops
  - Comparison Ops
  - Spectral Ops
  - Other Operations
  - BLAS and LAPACK Operations
- Utilities
- Operator Tags

3.1 $Linear$模块(全连接)

- 用于设置网络中的全连接层，全连接层的输入与输出都是二维张量
- 输入的一般形状为[batch_size, size]。

```
torch.nn.Linear(in_features, out_features, bias=True)
```

- in_features指的是输入的二维张量的大小，即输入的二维张量形状为[batch_size, in_features]。
- out_features指的是输出的二维张量的大小，即输出的二维张量形状为[batch_size，out_features]，它也代表了该全连接层的神经元个数。
- 从输入输出的张量的shape角度来理解，相当于一个输入为[batch_size, in_features]的张量变换成了[batch_size, out_features]的输出张量。

- Linear其实就是对输入 $x_{d\times i}$执行了一个线性变换，即：

$$y_{d\times o} = x_{d\times i}w_{i\times o} + b$$

- w：模型要学习的参数，w的维度为$i \times o$
- b：o维的向量偏置
- d：输入向量的维度（例如，你将单词编码成了10维向量，那么d就是10）
- o：输出神经元的个数
- i：输入神经元的个数

*Linear*示例

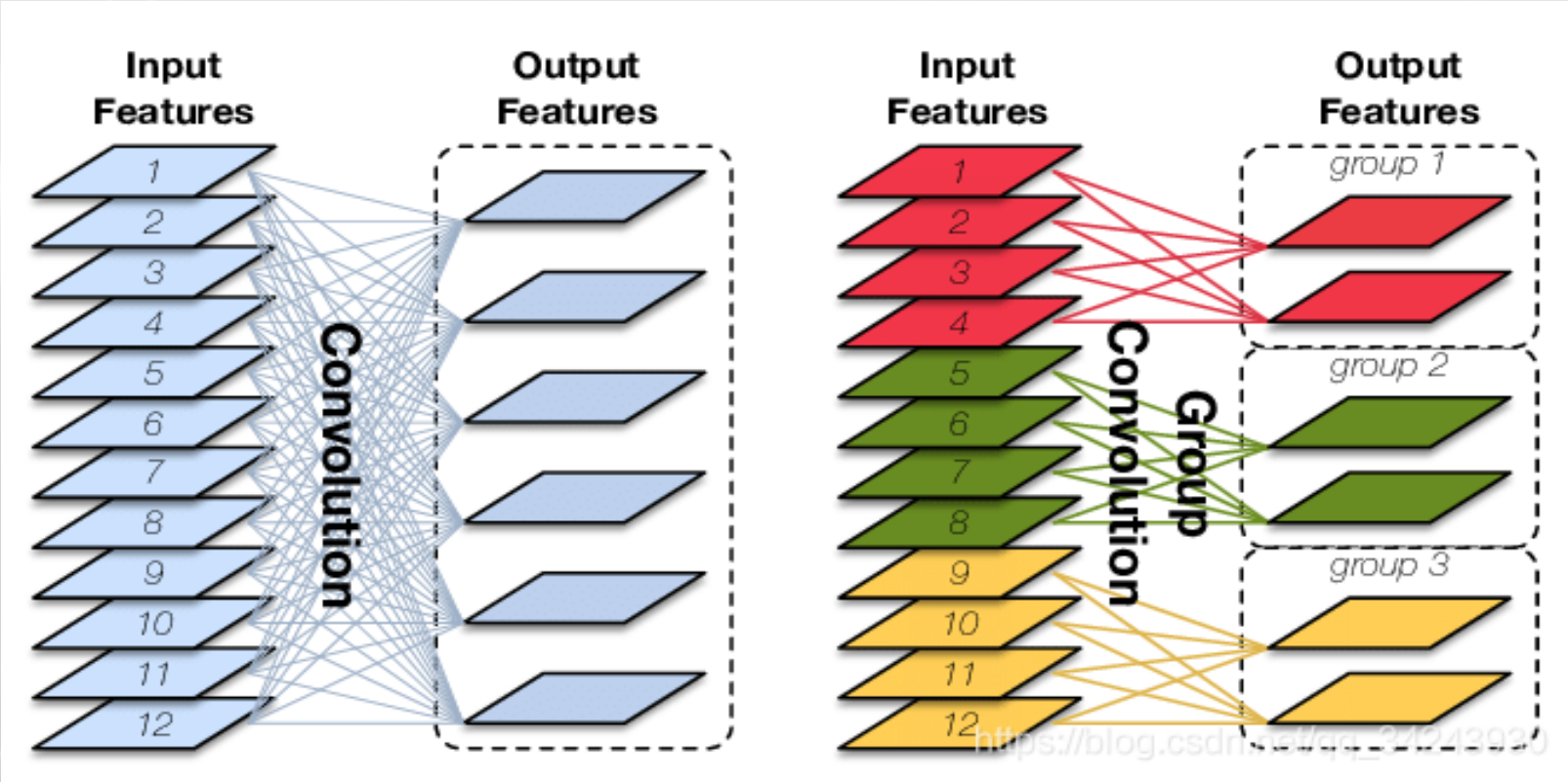
```
from torch import nn
import torch

model = nn.Linear(2,1)

input = torch.Tensor([1,2])
output = model(input)
print(output)

for param in model.parameters():
  print(param)
```

3.2 $Conv2d$模块

```
torch.nn.Conv2d(in_channels, out_channels, kernel_size,
    stride=1, padding=0, dilation=1, groups=1, bias=True)
```

- kernel_size:卷积核的大小，可以用(H,W)表示HxW的输出，也可以使用单个数字H表示H*H大小的输出。
- padding:填充操作，控制padding_mode的数目。默认为Zero-padding。
- dilation:扩张操作，控制kernel点（卷积核点）的间距，默认为1。
- group:控制分组卷积，默认不分组，为1组。
- bias:是否添加偏置，如为真，则在输出中添加一个可学习的偏差，默认为True。

分组卷积


Input Features
Output Features
1
2
3
4
5
6
7
8
9
10
11
12
Convolution
Input Features
Output Features
group 1
group 2
group 3
Group Convolution

$Conv2d$的输入和输出

- Conv2d输入

```
[ N, C_in, H_in, W_in ]
```

- Conv2d输出

```
[ N, C_out, H_out, W_out ]
```

- N为批量数

$$H_{out} = \frac{H_{in} - \text{dilation}_H \times (\text{kernel\_size}_H - 1) + 2 \times \text{padding}_H - 1}{\text{stride}_H} + 1$$

$$W_{out} = \frac{W_{in} - \text{dilation}_W \times (\text{kernel\_size}_W - 1) + 2 \times \text{padding}_W - 1}{\text{stride}_W} + 1$$

$Conv2d$示例1

```
import torch

x = torch.randn(3,1,5,4)
print(x)

conv = torch.nn.Conv2d(1,4,(2,3))
res = conv(x)

print(res.shape)    # torch.Size([3, 4, 4, 2])
```

$Conv2d$示例2

```
import torch as t
from torch import nn

# 假定输入的图像形状为[3,64,64]
x = t.randn(10, 3, 64, 64)
# 10张3个channel的，大小为64x64的图片

x = nn.Conv2d(3, 64, kernel_size=3, stride=3, padding=0)(x)
print(x.shape)
#torch.Size([10, 64, 21, 21])
```

*Conv2d*和*Linear*连接

- Conv2d的输出为四维张量，转换为二维张量之后，才能作为全连接层的输入

```
x = x.view(x.size(0), -1)
print(x.shape)
#torch.Size([10, 28224])

# in_features由输入张量的形状决定，out_features则决定了输出张量的形状
connected_layer = nn.Linear(in_features = 64*21*21, out_features = 10)

# 调用全连接层
output = connected_layer(x)
print(output.shape)
#torch.Size([10, 10])
```

3.4 $BatchNorm2d$二维批量归一化模块

```
torch.nn.BatchNorm2d(num_features, eps=1e-05, momentum=0.1, affine=True)
```

- num_features：特征数C
- eps：分母中添加的一个系数，目的是为了计算的稳定性，默认：1e-5
- momentum：用于均值和方差更新的一个估计参数，默认：0.1
- affine：指定是否需要仿射,即γ和β是否能被学习
- 输入、输出
  [N, C, H, W]

$BatchNorm2d$示例

```
import torch
from torch import nn

m = nn.BatchNorm2d(2)
print(m.weight)
print(m.bias)

input = torch.randn(1,2,3,3)
print(input)
output = m(input)
print(output)
print(output.size())
```

3.5 池化模块

1. $MaxPool2d$最大池化模块

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

- kernel_size: 池化核的大小
- 输入: (N,C,H_in,W_in)
- 输出: (N,C,H_out,W_out)

$$H_{out} = \frac{H_{in} - \text{dilation}_H \times (\text{kernel\_size}_H - 1) + 2 \times \text{padding}_H - 1}{\text{stride}_H} + 1$$

$$W_{out} = \frac{W_{in} - \text{dilation}_W \times (\text{kernel\_size}_W - 1) + 2 \times \text{padding}_W - 1}{\text{stride}_W} + 1$$

$MaxPool2d$示例

```
m = nn.MaxPool2d((3, 2), stride=(2, 1))
input = torch.randn(20, 16, 50, 32)
output = m(input)
print(output.size())
torch.Size([20, 16, 24, 31])
```

2. $AvgPool2d$平均池化模块

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

- kernel_size: 池化核的大小
- ceil_mode:如果为True，则在计算输出形状时使用ceil函数替代floor
- 输入: (N,C,H_in,W_in)
- 输出: (N,C,H_out,W_out)

计算平均池化输出形状

$$H_{out} = \frac{H_{in} - \text{kernel\_size}_H + 2 \times \text{padding}_H}{\text{stride}_H} + 1$$

$$W_{out} = \frac{W_{in} - \text{kernel\_size}_W + 2 \times \text{padding}_W}{\text{stride}_W} + 1$$

3. $AdaptiveAvgPool2d$自适应平均池化模块

```
torch.nn.AdaptiveAvgPool2d(output_size)
```

- output_size: 输出信号的尺寸
- 对输入信号，提供2维的自适应平均池化操作
- Input: (N, C, H_in, W_in)
- Output: (N, C, H_out, W_out)

- Adaptive Pooling特殊性在于：
- 输出张量的大小是给定的output_size
- 例如输入张量大小为(1, 64, 8, 9)，设定输出大小为(5,7)，通过Adaptive Pooling层，可以得到大小为(1, 64, 5, 7)的张量
- 对于任何输入大小的输入，可以将输出尺寸指定为H*W，但是输入和输出特征的数目不会变化。

```
input = torch.randn(1,64,8,9)
m = nn.AdaptiveAvgPool2d((5,7))
output = m(input)
output.shape

torch.Size([1, 64, 5, 7])
```

Pytorch函数类操作

- 位置：torch.nn.functional
- 特征：函数名全部小写，在函数外定义所有参数
- 例如
  - torch.nn.functional.adaptive_avg_pool2d(input, output_size)

```
from torch.nn import functional as F
# adaptive_avg_pool2d的第2个参数规定了池化操作后的特征图尺寸。
# 例如，设置为(1,1)以后，最终特征图大小都为(1,1)
# x = F.adaptive_avg_pool2d(x, [1,1])
# [b, 64, h, w] => [b, 64, 1, 1]
```

3. 6 激活函数

1. $ReLU$激活模块

```
torch.nn.ReLU(inplace=False)
```

- inplace:选择是否进行原位运算，即x = x+1
- 输入：$(N, *)$, *代表任意数目附加维度
- 输出：$(N, *)$，与输入拥有同样的形状

```
import torch
from torch import nn
m = nn.ReLU()
input = torch.randn(2)
output = m(input)
input, output

(tensor([-0.0468,  0.2225]), tensor([0.0000, 0.2225]))
```

2. $LeakyReLU$激活模块

```
torch.nn.LeakyReLU(negative_slope=0.01, inplace=False)
```

- 是ReLU的变形，给所有负值赋予一个非零斜率
- negative_slope：控制负斜率的角度，默认等于0.01
- inplace:选择是否进行原位运算，即x= x+1，默认为False

```
m = nn.LeakyReLU(0.1)
input = torch.randn(2)
output = m(input)
input,output

(tensor([-1.3222,  0.8163]), tensor([-0.1322,  0.8163]))
```

3. $Sigmoid$示例

```
m = nn.Sigmoid()
input = torch.randn(2)
output = m(input)
input, output

(tensor([-0.8425,  0.7383]), tensor([0.3010, 0.6766]))
```

4. $Tanh$示例

```
m = nn.Tanh()
input = torch.randn(2)
output = m(input)
input, output

(tensor([1.3372, 0.6170]), tensor([0.8710, 0.5490]))
```

3.7 RNN模块

- 执行如下运算：
$$h_t = \tanh(W_{ih}x_t + b_{ih} + W_{hh}h_{(t-1)} + b_{hh})$$

```
参数:torch.nn.RNN(input_size, hidden_size, num_layers)
```

- input_size：输入特征的维度，比如房价预测，房价都是用一维的数直接表示的，所以此时input_size为1；如果输入的是字符编码，比如一个字符用3维编码表示，那么此时input_size为3；
- hidden_size：隐含层神经元个数,也是输出的维度，因为rnn输出为各个时间步上的隐藏状态;
- num_layers：隐含层的层数；

RNN其它参数

- nonlinearity：非线形操作，'tanh' 或者 'relu'。 默认为 'tanh'
- bias：是否使用偏置系数b_{ih} 和 b_{hh}。 默认为True
- dropout：是否应用dropout, 默认不使用, 如若使用将其设置成一个0到1之间的数字即可
- birdirectional：是否使用双向的 rnn，默认是 False

参数符号

- N=batch size
- L=sequence length
- D=2, bidirectional=True; 1, bidirectional=False
- H_{in}=input_size
- H_{out}=hidden_size

- batch_first：输入数据的顺序，默认是 False，**输入数据**顺序为:
  - x：$[L, N, H_{in}]$
  - $h0$：$[\text{num\_layers}, N, H_{out}]$

  当 batch_first设置为True时，输入数据的顺序变为：
  - x：$[N, L, H_{in}]$
  - $h0$：$[N, \text{num\_layers}, H_{out}]$。

RNN输出

- RNN的输出包含两部分：输出值Y（即output）和最后一个时刻隐含层的输出h_n
- output:
  - $(L, N, D*H_{out})$, batch_first = False
  - $(N, L, D*H_{out})$, batch_first = True
- h_n: $(D*\text{num\_layers}, N, H_{out})$

RNN示例

```
from torch import nn

# 词向量维度100维，输出维度10
rnn = nn.RNN(100, 10)
print(rnn._parameters.keys())
# ['weight_ih_l0', 'weight_hh_l0', 'bias_ih_l0', 'bias_hh_l0']
# weight_ih_l0: 第0层的输入层和隐含层之间的权重
# weight_hh_l0: 第0层的隐含层之间在不同时间步之间的权重

print(rnn.weight_ih_l0.shape, rnn.weight_hh_l0.shape)
# torch.Size([10, 100]) torch.Size([10, 10])

# batch size: 10
print(rnn.bias_ih_l0.shape, rnn.bias_hh_l0.shape)
# torch.Size([10]) torch.Size([10])
```

RNN前向预测

```
out, ht = rnn.forward(x, h0)
# 或
out, ht = rnn(x, h0)
```

- x：[seq_len, batch, feature_len], 即 $[L, N, H_{in}]$。
  一次性将所有时刻特征输入，不需要每次输入当前时刻的 x_t;

- h0：[num_layers, batch, hidden_len]，即 $[\text{hum\_layers}, N, H_{out}]$。
- h0是第一个初始时刻所有层的记忆单元的Tensor（理解成每一层中每个句子的隐藏输出）；
- out：[seq_len, batch, hidden_len] ，即$[L, N, H_{out}]$。
- out是每一个时刻上 空间上最后一层的输出(相当于 y_t)

RNN示例1

```
# 5层RNN
import torch
from torch import nn

# (词向量维度)feature_len=100, (神经元数)hidden_len=20, 网络层数=5
rnn = nn.RNN(input_size=100, hidden_size=20, num_layers=5)
# 单词数量(seq_len=10),句子数量(batch=3),每个特征100维度(feature_len=100)
x = torch.randn(10, 3, 100)

# h_0的shape是[网络层数=5, batch=3, (神经元数)hidden_len=20]
# forward
out, h = rnn(x, torch.zeros(5, 3, 20))

print(out.shape)  # torch.Size([10, 3, 20])
print(h.shape)    # at: torch.Size([5, 3, 20])
```

RNN示例2

- 比如我现在想设计一个4层的RNN，用来做语音翻译，输入是一段中文，输出是一段英文。假设每个中文字符用100维数据进行编码，每个隐含层的维度是20，有4个隐含层。所以input_size = 100，hidden_size = 20，num_layers = 4。再假设模型已经训练好了，现在有个1个长度为10的句子做输入，那么seq_len = 10，batch_size = 1。代码如下：

```
import torch
import torch.nn as nn

input_size = 100    # 输入数据编码的维度
hidden_size = 20    # 隐含层维度
num_layers = 4      # 隐含层层数

rnn = nn.RNN(input_size=input_size,hidden_size=hidden_size,
    num_layers=num_layers)
print("rnn:",rnn)
```

```
seq_len = 10          # 句子长度
batch_size = 1
x = torch.randn(seq_len,batch_size,input_size)          # 输入数据x
h0 = torch.zeros(num_layers,batch_size,hidden_size)     # 输入数据h0

out, h = rnn(x, h0)  # 输出数据

print("out.shape:",out.shape)
print("h.shape:",h.shape)
```

$RNNCell$模块

- nn.RNN是一次性将 所有时刻 特征输入网络的
- nn.RNNCell将序列上的'每个时刻的数据' 分开来处理
- 例如：如果要处理3个句子，每个句子10个单词，每个单词用100维的嵌入向量表示
- nn.RNN传入的Tensor的shape是[10,3,100]
- nn.RNNCell传入的Tensor的shape是[3,100]，将此计算单元运行10次

$RNNCell$前向预测

```
ht = forward(xt, ht-1)
```

- xt ：[batch, feature_len]表示当前时刻的输入;
- ht-1：[num_layers, batch, hidden_len]前一个时刻的单元输出，ht是下一时刻的单元输入;
- out： out相当于 y_t

$RNNCell$示例

```
import torch
from torch import nn

# 单层RNN, feature_len=100, hidden_len=20
cell1 = nn.RNNCell(100, 20)
h1 = torch.zeros(3, 20)
x = torch.randn(10, 3, 100)
for xt in x:                    # xt.shape=[3, 100]
    h1 = cell1(xt, h1)
print(h1.shape)                 # torch.Size([3, 20])
```

```
# 多层RNN
cell1 = nn.RNNCell(100, 30)
cell2 = nn.RNNCell(30, 20)

h1 = torch.zeros(3, 30)
h2 = torch.zeros(3, 20)
x = torch.randn(10, 3, 100)
for xt in x:
    h1 = cell1(xt, h1)
    h2 = cell2(h1, h2)

print(h1.shape)           # torch.Size([3, 30])
print(h2.shape)           # torch.Size([3, 20])
```

3.8 $LSTM$模块

```
torch.nn.LSTM(input_size, hidden_size, num_layers)
```

- input_size：输入特征的维度，一般rnn中输入的是词向量，那么 input_size 就等于一个词向量的维度，即 feature_len;
- hidden_size：隐藏层神经元个数，或者也叫输出的维度（因为rnn输出为各个时间步上的隐藏状态）;
- num_layers：网络的层数;

$LSTM$输入及输出格式

```
out, (h_t, c_t) = lstm(x, [h_t0, c_t0])
```

- x：[seq_len, batch, feature_len]
- h/c：[num_layers, batch, hidden_len]
- out：[seq_len, batch, hidden_len]

$LSTM$示例

```
import torch
from torch import nn

# 4层的LSTM,输入的每个词用100维向量表示,隐藏单元和记忆单元的尺寸是20
lstm = nn.LSTM(input_size=100, hidden_size=20, num_layers=4)

# 3句话，每句10个单词，每个单词的词向量维度(长度)100
x = torch.rand(10, 3, 100)

# 不传入h_0和c_0则会默认初始化
out, (h, c) = lstm(x)

print(out.shape)     # torch.Size([10, 3, 20])
print(h.shape)       # torch.Size([4, 3, 20])
print(c.shape)       # torch.Size([4, 3, 20])
```

$LSTMCell$模块

- nn.LSTMCell 与 nn.LSTM 的区别 和 nn.RNN 与 nn.RNNCell 的区别一样。
- nn.LSTMCell()初始化方法和nn.LSTM一样。

```
h_t, c_t = lstmcell(x_t, [h_t-1, c_t-1])
```

- xt：[batch, feature_len]表示t时刻的输入
- h_t−1,c_t−1：[batch, hidden_len]，t−1时刻本层的隐藏单元和记忆单元

$LSTMCell$ 示例

```
import torch
from torch import nn

# 单层LSTM
# 1层的LSTM，输入的每个词用100维向量表示，隐藏单元和记忆单元的尺寸是20
cell = nn.LSTMCell(input_size=100, hidden_size=20)

# seq_len=10个时刻的输入,每个时刻shape都是[batch,feature_len]
# x = [torch.randn(3, 100) for _ in range(10)]
x = torch.randn(10, 3, 100)
```

```
# 初始化隐藏单元h和记忆单元c,取batch=3
h = torch.zeros(3, 20)
c = torch.zeros(3, 20)

# 对每个时刻,传入输入xt和上个时刻的h和c
for xt in x:
    b, c = cell(xt, (h, c))

print(b.shape)    # torch.Size([3, 20])
print(c.shape)    # torch.Size([3, 20])
```

```
# 两层LSTM
# 输入的feature_len=100,隐藏单元和记忆单元hidden_len=30
cell_L0 = nn.LSTMCell(input_size=100, hidden_size=30)
# hidden_len从L0层的30变到这一层的20
cell_L1 = nn.LSTMCell(input_size=30, hidden_size=20)

# 分别初始化L0层和L1层的隐藏单元h 和 记忆单元C,取batch=3
h_L0 = torch.zeros(3, 30)
C_L0 = torch.zeros(3, 30)

h_L1 = torch.zeros(3, 20)
C_L1 = torch.zeros(3, 20)
```

```
x = torch.randn(10, 3, 100)

for xt in x:
    # L0层接受xt输入
    h_L0, C_L0 = cell_L0(xt, (h_L0, C_L0))
    # L1层接受L0层的输出h作为输入
    h_L1, C_L1 = cell_L1(h_L0, (h_L1, C_L1))

print(h_L0.shape, C_L0.shape)
# torch.Size([3, 30]) torch.Size([3, 30])
print(h_L1.shape, C_L1.shape)
# torch.Size([3, 20]) torch.Size([3, 20])
```

3.9 *Embedding*词嵌入模块

```
torch.nn.Embedding(num_embeddings, embedding_dim, padding_idx=None,
                   max_norm=None, norm_type=2.0, scale_grad_by_freq=False,
                   sparse=False, _weight=None)
```

- 嵌入层是一个存储固定大小词典的向量查找表。
- 即，给一个编号，嵌入层就能返回这个编号对应的嵌入向量，嵌入向量反映了各个编号代表的符号之间的语义关系。
- 输入：一个编号列表，输出：对应的符号嵌入向量列表。

- num_embeddings(int)：词典的大小尺寸，比如总共出现5000个词，那就输入5000。此时index为（0-4999）;
- embedding_dim(int)： 嵌入向量的维度，即用多少维来表示一个符号;
- padding_idx(int,可选)：比如，输入长度为100，但是每次的句子长度并不一样，后面就需要用统一的数字填充，而这里就是指定这个数字;

- max_norm(float,可选)：最大范数，如果嵌入向量的范数超过了这个界限，就要进行再归一化;
- norm_type (float, 可选)：指定利用什么范数计算，并用于对比max_norm，默认为2范数;
- scale_grad_by_freq (boolean, 可选)：根据单词在mini-batch中出现的频率，对梯度进行放缩，默认为False;
- sparse (bool, 可选)：若为True，则 与权重矩阵相关的梯度转变为稀疏张量;

词嵌入示例

```
import torch
from torch import nn

# 给单词编索引号
word_to_idx = {'hello':0, 'world':1}
# 得到目标单词索引
lookup_tensor = torch.tensor([word_to_idx['hello']], dtype=torch.long)

embeds = nn.Embedding(num_embeddings=2, embedding_dim=5)
# 传入单词的index，返回对应的嵌入向量
hello_embed = embeds(lookup_tensor)
print(hello_embed)
```

THE END