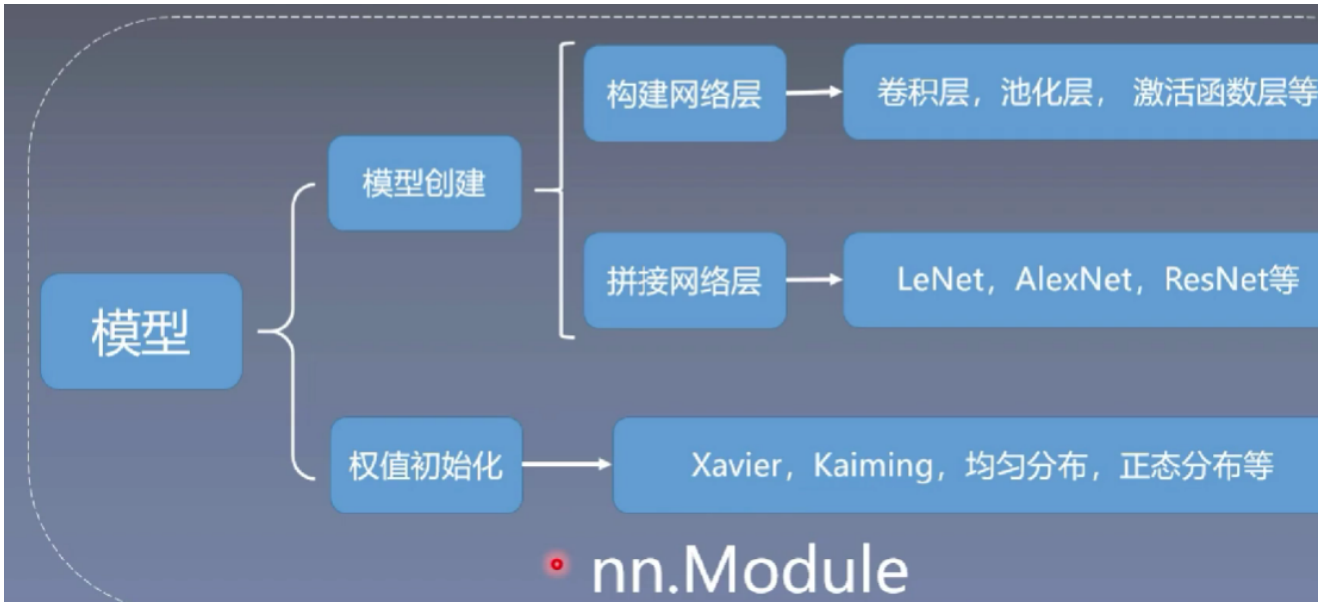
BUILD MODEL

一、内容

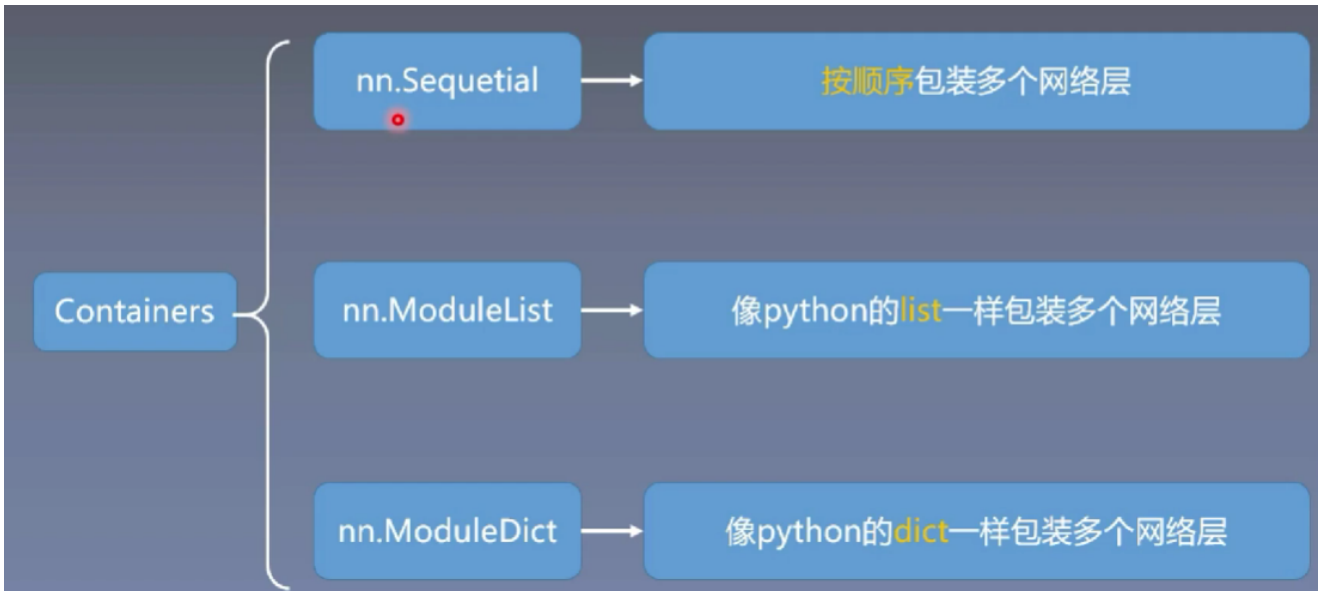
torch.nn 命名空间提供了构建自己的神经网络所需的所有构建块。PyTorch 中的每个模块都是 nn.Module 的子类。神经网络本身也是一个模块，它由其他模块（层）组成。这种嵌套结构允许轻松地构建和管理复杂的架构。



二、代码

一、Containers容器

主要包括Sequential、ModuleList、ModuleDict这三个类。



1、Sequential

该模块将按照它们在构造函数中传递的顺序添加到其中。也可以传入一个模块的有序字典。

```
from torch import nn

model = nn.Sequential(
                nn.Conv2d(1,20,5),
                nn.ReLU(),
                nn.Conv2d(20,64,5),
                nn.ReLU()
            )

print(model)
```

```
Sequential(
  (0): Conv2d(1, 20, kernel_size=(5, 5), stride=(1, 1))
  (1): ReLU()
  (2): Conv2d(20, 64, kernel_size=(5, 5), stride=(1, 1))
  (3): ReLU()
)
```

> 在每一层前都有序号，但当层数过多时，序号并不能帮助理清网络结构。可以采用下面的方式实现。

```
model = nn.Sequential(OrderedDict([
              ('conv1', nn.Conv2d(1,20,5)),
              ('relu1', nn.ReLU()),
              ('conv2', nn.Conv2d(20,64,5)),
              ('relu2', nn.ReLU())
            ]))
print(model)
```

```
Sequential(
  (conv1): Conv2d(1, 20, kernel_size=(5, 5), stride=(1, 1))
  (relu1): ReLU()
  (conv2): Conv2d(20, 64, kernel_size=(5, 5), stride=(1, 1))
  (relu2): ReLU()
)
```

2、ModuleList

ModuleList 可以像普通的 Python 列表一样进行索引

```
class MyModule(nn.Module):
      def __init__(self):
            super().__init__()
            self.linears = nn.ModuleList([nn.Linear(10, 10) for i in range(10)])

      def forward(self, x):
            # ModuleList can act as an iterable, or be indexed using ints
            for layer in self.linears:
                  x = layer(x)
            return x

model = MyModule()
print(model)
```

```
MyModule(
  (linears): ModuleList(
    (0-9): 10 x Linear(in_features=10, out_features=10, bias=True)
  )
)
```

3、ModuleDict

```
class MyModule2(nn.Module):
      def __init__(self):
            super().__init__()
            self.choices = nn.ModuleDict({
                        'conv': nn.Conv2d(10, 10, 3),
                        'pool': nn.MaxPool2d(3)
```

```
            })
            self.activations = nn.ModuleDict([
                        ['lrelu', nn.LeakyReLU()],
                        ['prelu', nn.PReLU()]
            ])

        def forward(self, x, choice, act):
            x = self.choices[choice](x)
            x = self.activations[act](x)
            return x

model = MyModule2()
X = torch.rand(10,20,20) # CHW
y = model(X,'conv','lrelu')
print(model)
print(y.shape)
```

```
MyModule2(
  (choices): ModuleDict(
    (conv): Conv2d(10, 10, kernel_size=(3, 3), stride=(1, 1))
    (pool): MaxPool2d(kernel_size=3, stride=3, padding=0, dilation=1, ceil_mode=False)
  )
  (activations): ModuleDict(
    (lrelu): LeakyReLU(negative_slope=0.01)
    (prelu): PReLU(num_parameters=1)
  )
)
torch.Size([10, 18, 18])
```

> **注意：前向传播时要输入选择的层结构**y = model(X,'conv','lrelu')

二、卷积层

参数：

- in_channels (int) - 输入图像的通道数
- out_channels (int) - 卷积产生的通道数
- kernel_size (int **或** tuple) - 卷积核的大小
- stride (int **或** tuple，**可选**) - 卷积的步长。默认值：1
- padding (int, tuple **或** str，**可选**) - 添加到输入所有四边的填充。默认值：0
- padding_mode (str, **可选**) - ‘zeros’，‘reflect’，‘replicate’ 或 ‘circular’。1默认值：'zeros'2
- dilation (int **或** tuple，**可选**) - 内核元素之间的间距。默认值：1
- groups (int, **可选**) - 从输入通道到输出通道的阻塞连接数。默认值：1

- bias (bool, **可选**) - 如果为 True，则在输出中添加一个可学习的偏置。默认值：True。

输入输出大小：

$$\text{Input: } (N, C_{in}, H_{in}, W_{in}) \text{ or } (C_{in}, H_{in}, W_{in})$$
$$\text{Output: } (N, C_{out}, H_{out}, W_{out}) \text{ or } (C_{out}, H_{out}, W_{out})$$

公式：

$$n_H = \frac{n_H - f}{s} + 1$$

$$n_W = \frac{n_W - f}{s} + 1$$

> 以上是经过卷积以后tensor大小的计算公式。n_W、n_H是feature map的大小，f是kernel大小，s是步长。

```
m = nn.Conv2d(16, 33, 3, stride=2)
input = torch.randn(20, 16, 50, 100)
output = m(input)
print(output.shape)
```

```
torch.Size([20, 33, 24, 49])
```

三、Pooling layers

参数：

- kernel_size (int **或** (int, int) **元组**) - 窗口大小，用于取最大值
- stride (int **或** (int, int) **元组**) - 窗口的步长。默认值为 kernel_size
- padding (int **或** (int, int) **元组**) - 隐式负无穷填充，添加到两侧
- dilation (int **或** (int, int) **元组**) - 控制窗口中元素的步长的参数
- return_indices (bool) - 如果为 True，则将返回最大索引和输出。对于 torch.nn.MaxUnpool2d 很有用
- ceil_mode (bool) - 如果为 True，则使用 ceil 而不是 floor 来计算输出形状。

$$n_H = \frac{n_H - f}{s} + 1$$

$$n_W = \frac{n_W - f}{s} + 1$$

> 以上是经过Pooling以后tensor大小的计算公式。n_W、n_H是feature map的大小，f是kernel大小，s是步长。

```
m = nn.MaxPool2d(3, stride=2)
input = torch.randn(20, 16, 50, 32)
output = m(input)
print(output.shape)
```

```
torch.Size([20, 16, 24, 15])
```

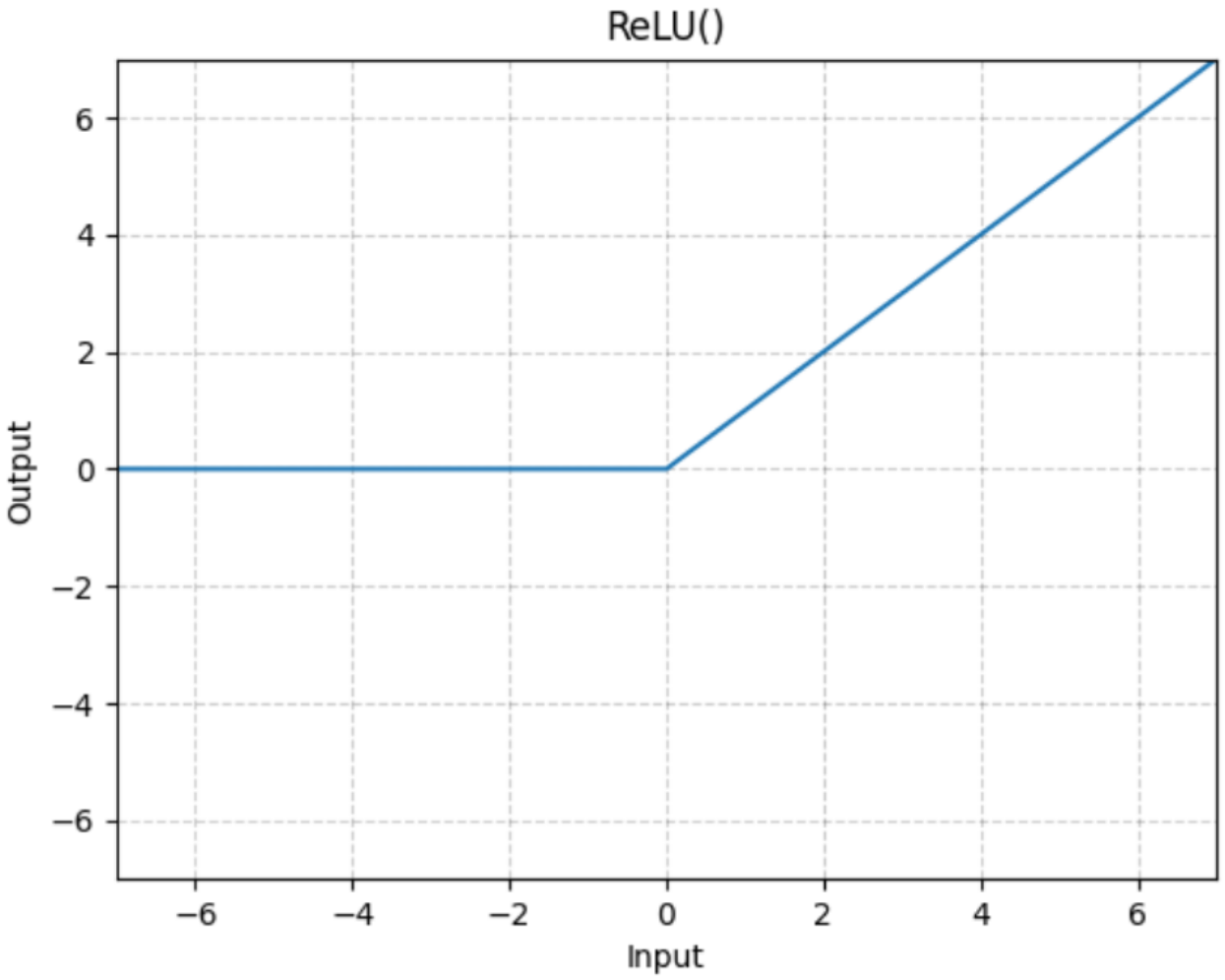
四、Non-linear Activations

1、RELU

公式：

$$ReLU(x) = max(0, x)$$

函数图像：




```
m = nn.ReLU()
input = torch.tensor(range(-5,5))
output = m(input)
print(output)
```

```
tensor([0, 0, 0, 0, 0, 0, 1, 2, 3, 4])
```

2、Softmax

公式：

$$Softmax(x_i) = \frac{x_i}{\sum x_i}$$

参数：

dim (int)：一个沿着 Softmax 将被计算的维度（所以沿着 dim 的每个切片都会加起来等于 1）

```
m = nn.Softmax(dim=1)
input = torch.randn(3, 2)
output = m(input)
print(output)
```

```
tensor([[0.3051, 0.6949],
        [0.4785, 0.5215],
        [0.3953, 0.6047]])
```

五、BatchNorm2d

公式：

$$y = \frac{x - E[x]}{\sqrt{Var[x] + \epsilon}} * \gamma + \beta$$

其中，$\frac{x-E[x]}{\sqrt{Var[x]+\epsilon}}$ 是对miniBatch数据的归一化，在这一层中将学习，类似于线性层中的参数W和b。

参数：

- num_features (int) – 通道数C （N, C, H, W）
- eps (float) – 为了数值稳定性，加到分母上的一个值。默认值：1e-5
- momentum (float) – 用于计算 running_mean 和 running_var 的值。可以设置为 None，表示累积移动平均（即简单平均）。默认值：0.1
- affine (bool) – 一个布尔值，当设置为 True 时，这个模块有可学习的仿射参数（参数γ和β）。默认值：True
- track_running_stats (bool) – 一个布尔值，当设置为 True 时，这个模块跟踪运行的均值和方差，当设置为 False 时，这个模块不跟踪这些统计量，并将统计缓冲区 running_mean 和 running_var 初始化为 None。当这些缓冲区为 None 时，这个模块在训练和评估模式下都使用批量统计数据。默认值：True。**注意：这个意思是在测试数据集上是否用测试数据上计算得到的均值和方差，不是在测试集上依然使用训练集上的均值和方差**

```python
m = nn.BatchNorm2d(100, affine=False)
input = torch.randn(20, 100, 35, 45)
output = m(input)
print(output.shape)
```

```
torch.Size([20, 100, 35, 45])
```

六、Linear

参数：

- in_features (int) – 每个输入样本的大小
- out_features (int) – 每个输出样本的大小
- bias (bool) – 如果设置为 False，该层将不会学习一个加法偏置。默认值：True

权重的形状为 (out_features , in_features) 。它表示模块对输入数据应用的线性变换的系数矩阵。out_features 是输出样本的大小，in_features 是输入样本的大小。权重矩阵的每一行对应一个输出特征，每一列对应一个输入特征。**权重**（weight）是通过从均匀分布 $U(-k, k)$ 中初始化的，其中 $k = \frac{1}{in\_features}$。这意味着，权重矩阵中的每个元素都是从一个均匀分布中随机采样得到的，其范围为 -k 到 k，其中 k 的值取决于输入特征的数量（in_features）。这种初始化方法有助于防止梯度消失或梯度爆炸，并有助于模型的收敛。bias也是同样初始化的。

七、Dropout

以概率 p 随机将输入张量的一些元素置零。每个通道在每次前向调用时都会独立地被置零。

参数：

- p (float) – 元素被置零的概率。默认值：0.5
- inplace (bool) – 如果设置为 True，将会进行原地操作。默认值：False

```
m = nn.Dropout(p=0.9)
input = torch.randn(3, 5)
output = m(input)
print(output)
```

```
tensor([[  0.0000,   0.0000,   0.0000,  -0.0000, -18.3604],
        [ -0.0000,  -0.0000,   0.0000,  -0.0000,   0.0000],
        [ -0.0000,  -0.0000,  -0.0000,   0.0000,  -0.0000]])
```

八、Loss Functions

1、MSELoss

参数：

- size_average (bool, optional) - 已弃用（参见 reduction）。默认情况下，损失在批量中的每个损失元素上取平均。注意，对于一些损失，每个样本有多个元素。如果 size_average 字段设置为 False，则损失会对每个小批量求和。当 reduce 为 False 时忽略。默认值：True
- reduce (bool, optional) - 已弃用（参见 reduction）。默认情况下，根据 size_average，损失在每个小批量中对观察值进行平均或求和。当 reduce 为 False 时，返回每个批量元素的损失，而忽略 size_average。默认值：True
- reduction (str, optional) - 指定要应用于输出的约简：'none' | 'mean' | 'sum'。'none'：不会应用任何约简，'mean'：输出的总和将除以输出中的元素数量，'sum'：输出将被求和。注意：size_average 和 reduce 正在被弃用，在此期间，指定这两个参数中的任何一个都将覆盖 reduction。默认值：'mean'

```
loss = nn.MSELoss()
input = torch.randn(3, 5, requires_grad=True)
target = torch.randn(3, 5)
output = loss(input, target)
output.backward()
```