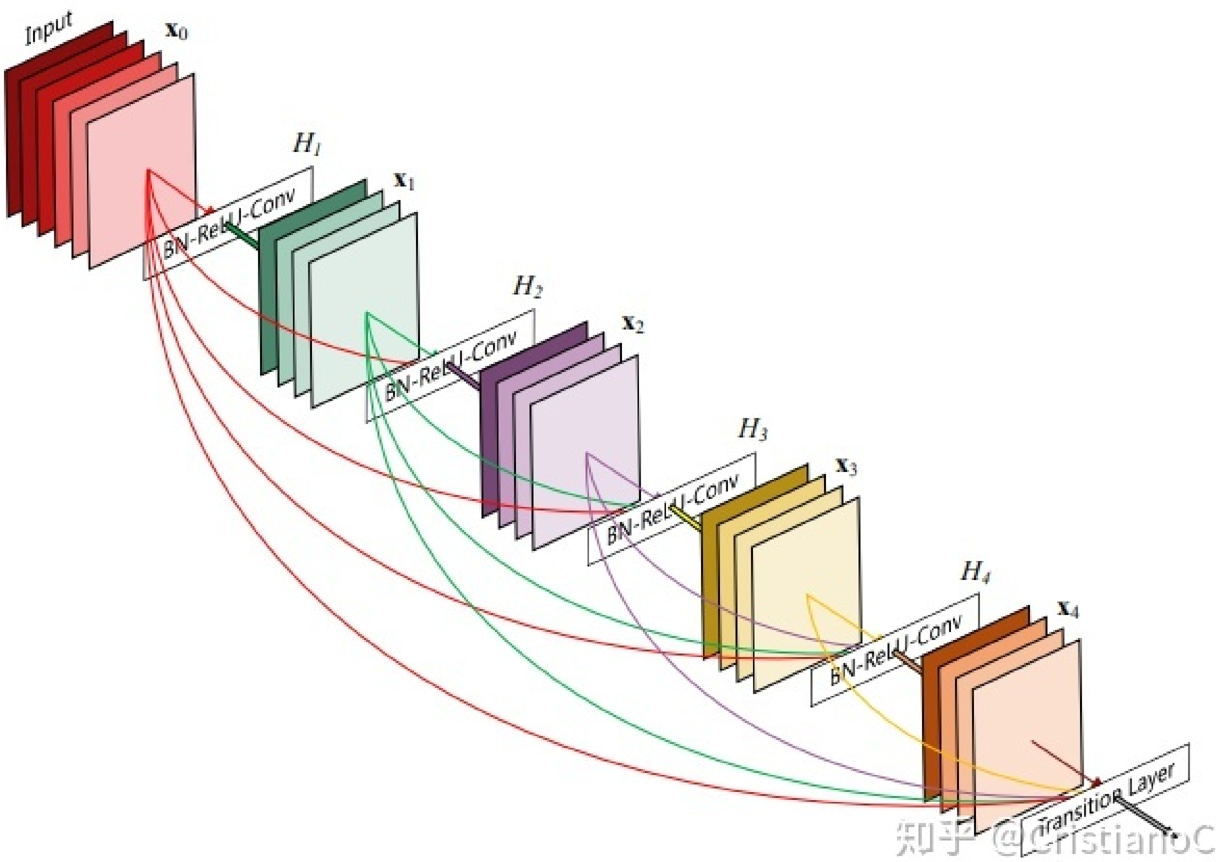
DenseNet

背景

在DenseNet出现之前，有对网络宽度下手的Inception网络，有对网络深度记性思考的Resnet网络，这里DenseNet作者选择了从feature着手，其新结构不但减缓了梯度消失，参数量也减少了。

DenseNet

作者提出了一个很简单的想法，就是让**前面所有层与后面的层进行建立密集连接**（以增加维度的方式），实现对特征feature的重用，以解决有网络深度的增加，反向传播时梯度很有可能会消失的问题。



一般的神经网络来说，可以把输入输出看作成一个公式：$X_l = H_l(X_{L-1})$，而X_l代表输入，H_l代表一个组合函数，常包括BN、ReLU、Pooling、Conv等操作。

> Resnet网络就可以用：$X_l = H_l(X_{L-1}) + X_{L-1}$来表示

而对于DenseNet来说，它是采用了跨通道concat相加的方式来操作feature map的，可用公式：$X_l = H_l(X_0, X_1, ..., X_{L-1})$来表示；

> 这里的concat方式，是指channel维度进行叠加，那就需要保证它们的feature map 大小保持一致。

> 那为何参数量相对Resnet少了呢？
>
> 通过对比网络结构，可以看出DenseNet网络中的**输入、输出的channel通道数**相对来说**少了**很多 ==> 导致BN层的参数也会少 ==> 也会导致FCN的参数也会少很多。
>
> 注意的是：参数虽然少了，速度慢了，因为DenseNet网络中的feature map要相对Resnet大很多 ==> 导致conv卷积过程的计算量增加

DenseNet网络架构

因此，上面的图可以认为是一个**Dense Block（稠密块）**，而整体网络就是由若干个不同维度的Dense Block组成的，保证了Dense Block内部的feature map size一致，而Dense Block之间维度不同，为了concat方便，使用了一个**Transition模块（过渡层）**来进行下采样过渡。

Layers	Output Size	DenseNet-121	DenseNet-169	DenseNet-201	DenseNet-264
Convolution	112×112	7×7 conv, stride 2			
Pooling	56×56	3×3 max pool, stride 2			
Dense Block (1)	56×56	$\begin{bmatrix} 1 \times 1 \text{ conv} \\ 3 \times 3 \text{ conv} \end{bmatrix} \times 6$	$\begin{bmatrix} 1 \times 1 \text{ conv} \\ 3 \times 3 \text{ conv} \end{bmatrix} \times 6$	$\begin{bmatrix} 1 \times 1 \text{ conv} \\ 3 \times 3 \text{ conv} \end{bmatrix} \times 6$	$\begin{bmatrix} 1 \times 1 \text{ conv} \\ 3 \times 3 \text{ conv} \end{bmatrix} \times 6$
Transition Layer (1)	56×56	1×1 conv			
	28×28	2×2 average pool, stride 2			
Dense Block (2)	28×28	$\begin{bmatrix} 1 \times 1 \text{ conv} \\ 3 \times 3 \text{ conv} \end{bmatrix} \times 12$	$\begin{bmatrix} 1 \times 1 \text{ conv} \\ 3 \times 3 \text{ conv} \end{bmatrix} \times 12$	$\begin{bmatrix} 1 \times 1 \text{ conv} \\ 3 \times 3 \text{ conv} \end{bmatrix} \times 12$	$\begin{bmatrix} 1 \times 1 \text{ conv} \\ 3 \times 3 \text{ conv} \end{bmatrix} \times 12$
Transition Layer (2)	28×28	1×1 conv			
	14×14	2×2 average pool, stride 2			
Dense Block (3)	14×14	$\begin{bmatrix} 1 \times 1 \text{ conv} \\ 3 \times 3 \text{ conv} \end{bmatrix} \times 24$	$\begin{bmatrix} 1 \times 1 \text{ conv} \\ 3 \times 3 \text{ conv} \end{bmatrix} \times 32$	$\begin{bmatrix} 1 \times 1 \text{ conv} \\ 3 \times 3 \text{ conv} \end{bmatrix} \times 48$	$\begin{bmatrix} 1 \times 1 \text{ conv} \\ 3 \times 3 \text{ conv} \end{bmatrix} \times 64$
Transition Layer (3)	14×14	1×1 conv			
	7×7	2×2 average pool, stride 2			
Dense Block (4)	7×7	$\begin{bmatrix} 1 \times 1 \text{ conv} \\ 3 \times 3 \text{ conv} \end{bmatrix} \times 16$	$\begin{bmatrix} 1 \times 1 \text{ conv} \\ 3 \times 3 \text{ conv} \end{bmatrix} \times 32$	$\begin{bmatrix} 1 \times 1 \text{ conv} \\ 3 \times 3 \text{ conv} \end{bmatrix} \times 32$	$\begin{bmatrix} 1 \times 1 \text{ conv} \\ 3 \times 3 \text{ conv} \end{bmatrix} \times 48$
Classification Layer	1×1	7×7 global average pool			
		1000D fully-connected, softmax			

> global average pool：关键点在于它是整张feature map进行操作，得到1x1的feature map结果。（形如AxBxN的feature map --> 1x1xN的feature map）

以上为不同配置的DenseNet网络图，可以看出：

- 不同大小的网络输入均通过了一次7x7卷积+3x3最大池化，
- Dense Block内部，均由1x1卷积+3x3卷积为一个单元组成的（具体些就是：BN + ReLU + 1x1 Conv + BN + ReLU + 3x3 Conv），

```
# 定义Dense Layer
class _DenseLayer(nn.Module):
    def __init__(
        """
        Args:
            num_init_features (int) - 输入的通道数
            growth_rate (int) - 每层需要去额外concat的filters通道数量
            bn_size (int) - 乘法因子
              (i.e. bn_size * k features in the bottleneck layer)
            drop_rate (float) - dropout rate after each dense layer
            memory_efficient (bool) - If True, uses checkpointing. 内存效率更高
        """
        self, num_input_features: int, growth_rate: int, bn_size: int, drop_rate: float,
memory_efficient: bool = False
    ) -> None:
        super().__init__()
        self.norm1 = nn.BatchNorm2d(num_input_features)
        self.relu1 = nn.ReLU(inplace=True)
        self.conv1 = nn.Conv2d(num_input_features, bn_size * growth_rate, kernel_size=1,
stride=1, bias=False)

        self.norm2 = nn.BatchNorm2d(bn_size * growth_rate)
        self.relu2 = nn.ReLU(inplace=True)
        self.conv2 = nn.Conv2d(bn_size * growth_rate, growth_rate, kernel_size=3, stride=1,
padding=1, bias=False)
```

> 重点说明下内存优化，图什么?
>
> DenseNet的内存主要消耗:
>
> 1. 一个在于拼接过程，每次拼接都会开辟新的内存空间。
> 2. 另一个在于forward和backward相互之间存在依赖关系。
>
> 解决方案:
>
> 1）针对1.中，可通过内存共享来解决，比如将之前模块相加的结果可以保留给下一模块使用;
>
> 2）针对2.中，可通过释放forward占用的内存换空间。（forward前向传播时的内存一旦释放，在backward计算的时候，由于依赖于forward信息，就需要重新计算，此时时间消耗增加15%，却可以节省70%的空间。）

```python
# 定义Dense Block模块
# 可通过输入的num_layers来得知当前Dense Block存在多少个Dense单元
class _DenseBlock(nn.ModuleDict):
    _version = 2

    def __init__(
        self,
        num_layers: int,
        num_input_features: int,
        bn_size: int,
        growth_rate: int,
        drop_rate: float,
        memory_efficient: bool = False,
    ) -> None:
        super().__init__()
        for i in range(num_layers):
            layer = _DenseLayer(
                num_input_features + i * growth_rate,
                growth_rate=growth_rate,
                bn_size=bn_size,
                drop_rate=drop_rate,
                memory_efficient=memory_efficient,
            )
            self.add_module("denselayer%d" % (i + 1), layer)

    def forward(self, init_features: Tensor) -> Tensor:
        features = [init_features]
        for name, layer in self.items():
            new_features = layer(features)
            features.append(new_features)
        return torch.cat(features, 1)
```

- Transition Layer就是由1x1卷积+2x2平均池化组成，（这里卷积实现通道数改变，池化可实现feature map size调整）（**专业点就是：起到降低通道数、压缩模型的作用**）

```
# 定义Transition Layer
class _Transition(nn.Sequential):
    def __init__(self, num_input_features: int, num_output_features: int) -> None:
        super().__init__()
        self.norm = nn.BatchNorm2d(num_input_features)
        self.relu = nn.ReLU(inplace=True)
        self.conv = nn.Conv2d(num_input_features, num_output_features, kernel_size=1,
stride=1, bias=False)
        self.pool = nn.AvgPool2d(kernel_size=2, stride=2)
```

- 输出也一致，通过一次7x7全局池化+FCN+softmax ==> 得到结果。

```
# 根据配置定制DenseNet网络结构
class DenseNet(nn.Module):
    """
    Args:
        block_config (list of 4 ints) - DenseNet的网络配置list
        num_classes (int) - 类别数
    """
    def __init__(
        self,
        growth_rate: int = 32,
        block_config: Tuple[int, int, int, int] = (6, 12, 24, 16),
        num_init_features: int = 64,
        bn_size: int = 4,
        drop_rate: float = 0,
        num_classes: int = 1000,
        memory_efficient: bool = False,
    ) -> None:

        super().__init__()
        _log_api_usage_once(self)  # 监视记录API调用的

        # First convolution——这里对应的就是输入后的操作
        self.features = nn.Sequential(
            OrderedDict(
                [
                    ("conv0", nn.Conv2d(3, num_init_features, kernel_size=7, stride=2, padding=3,
bias=False)),
                    ("norm0", nn.BatchNorm2d(num_init_features)),
                    ("relu0", nn.ReLU(inplace=True)),
```

```
                ("pool0", nn.MaxPool2d(kernel_size=3, stride=2, padding=1)),
            ]
        )
    )

    # Each denseblock
    num_features = num_init_features
    for i, num_layers in enumerate(block_config):
        block = _DenseBlock(
            num_layers=num_layers,
            num_input_features=num_features,
            bn_size=bn_size,
            growth_rate=growth_rate,
            drop_rate=drop_rate,
            memory_efficient=memory_efficient,
        )
        self.features.add_module("denseblock%d" % (i + 1), block)
        num_features = num_features + num_layers * growth_rate
        # 在对应位置加上对应的Transition Layer
        if i != len(block_config) - 1:
            trans = _Transition(num_input_features=num_features,
num_output_features=num_features // 2)
            self.features.add_module("transition%d" % (i + 1), trans)
            num_features = num_features // 2

    # Final batch norm
    self.features.add_module("norm5", nn.BatchNorm2d(num_features))

    # Linear layer
    self.classifier = nn.Linear(num_features, num_classes)

    # Official init from torch repo.
    for m in self.modules():
        if isinstance(m, nn.Conv2d):
            nn.init.kaiming_normal_(m.weight)
        elif isinstance(m, nn.BatchNorm2d):
            nn.init.constant_(m.weight, 1)
            nn.init.constant_(m.bias, 0)
        elif isinstance(m, nn.Linear):
            nn.init.constant_(m.bias, 0)
  # 前向处理
  def forward(self, x: Tensor) -> Tensor:
```

```
features = self.features(x)
out = F.relu(features, inplace=True)
out = F.adaptive_avg_pool2d(out, (1, 1))
out = torch.flatten(out, 1)
out = self.classifier(out)
return out
```

Method	Depth	Params	C10	C10+	C100	C100+	SVHN
Network in Network [22]	-	-	10.41	8.81	35.68	-	2.35
All-CNN [32]	-	-	9.08	7.25	-	33.71	-
Deeply Supervised Net [20]	-	-	9.69	7.97	-	34.57	1.92
Highway Network [34]	-	-	-	7.72	-	32.39	-
FractalNet [17]	21	38.6M	10.18	5.22	35.34	23.30	2.01
with Dropout/Drop-path	21	38.6M	7.33	4.60	28.20	23.73	1.87
ResNet [11]	110	1.7M	-	6.61	-	-	-
ResNet (reported by [13])	110	1.7M	13.63	6.41	44.74	27.22	2.01
ResNet with Stochastic Depth [13]	110	1.7M	11.66	5.23	37.80	24.58	1.75
	1202	10.2M	-	4.91	-	-	-
Wide ResNet [42]	16	11.0M	-	4.81	-	22.07	-
	28	36.5M	-	4.17	-	20.50	-
with Dropout	16	2.7M	-	-	-	-	1.64
ResNet (pre-activation) [12]	164	1.7M	11.26*	5.46	35.58*	24.33	-
	1001	10.2M	10.56*	4.62	33.47*	22.71	-
DenseNet ($k = 12$)	40	1.0M	**7.00**	5.24	**27.55**	24.42	1.79
DenseNet ($k = 12$)	100	7.0M	**5.77**	**4.10**	**23.79**	**20.20**	1.67
DenseNet ($k = 24$)	100	27.2M	**5.83**	**3.74**	**23.42**	**19.25**	**1.59**
DenseNet-BC ($k = 12$)	100	0.8M	**5.92**	4.51	**24.15**	22.27	1.76
DenseNet-BC ($k = 24$)	250	15.3M	**5.19**	**3.62**	**19.64**	**17.60**	1.74
DenseNet-BC ($k = 40$)	190	25.6M	-	**3.46**	-	**17.18**	-

为了验证这一idea，作者在CIFAR-10及其数据增强后的数据集, CIFAR-100及其数据增强后的数据集以及SVHN数据集上进行试验，上图均为error rate比较。可看出，在性能上都得到较好的结果。

Model	top-1	top-5
DenseNet-121	25.02 / 23.61	7.71 / 6.66
DenseNet-169	23.80 / 22.08	6.85 / 5.92
DenseNet-201	22.58 / 21.46	6.34 / 5.54
DenseNet-264	22.15 / 20.80	6.12 / 5.29

在Imagenet数据集上进行crop-1和crop-10上的top-1和top-5的error rate比较。