

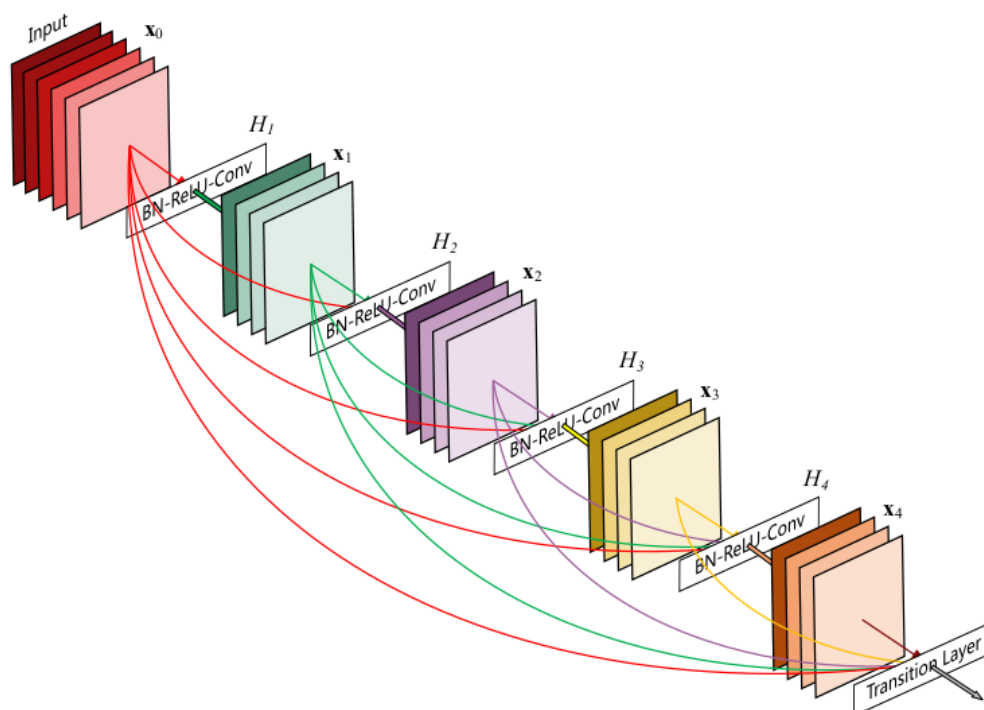
# DenseNet学习笔记及实现

## 前言

之前的一些研究表明了在输入层和输出层之间添加一些跳接可以让网络架构更深，且训练更有效率。例如 [ResNet](#) [1]，解决了深层网络梯度消失的问题，而 [GoogleNet](#) [2] 则是让网络加宽。借鉴这两种思想，让网络中各层之间的信息传递，将**所有的层连接起来**，这就是 [DenseNet](#) [3] 的基本思想。

在传统的卷积神经网络中，第  $L$  层就有  $L$  个连接，每一层和其他的层相互连接，所以总共的跳接就有  $\frac{L(L+1)}{2}$ ，如 [Figure 1](#) 所示。对于每一层来说，所有此前的网络层的特征图作为输入，而其自身的特征图作为之后所有层的输入。[DenseNet](#) 有以下几个优点：

- 减轻了梯度消失问题 (vanishing-gradient)
- 加强了特征传播 (feature propagation)
- 更有效地利用特征
- 大大减少了参数数量



**Figure 1:** A 5-layer dense block with a growth rate of  $k = 4$ . Each layer takes all preceding feature-maps as input.

## DenseNet 架构

假设  $X_0$  是输入卷积网络的单张图片，网络包括  $L$  层，每一层都实现了非线性变换  $H_l(\cdot)$ ，其中  $l$  表示的是第  $l$  层。 $H_l(\cdot)$  是包含了批量归一化 (Batch Normalization, BN)、ReLU、池化和卷积的组合操作，将  $l^{th}$  层的输出命名为  $X_l$ 。

## ResNets

传统的卷积前馈网络将  $l^{th}$  的输出作为  $(l + 1)^{th}$  层的输入，得到这个转换公式： $X_l = H_l(X_{l-1})$ 。而 **ResNet** 通过标识函数 (identity function) 添加了一个绕过非线性变换  $H_l(\cdot)$  的跳接

$$X_l = H_l(x_{l-1}) + x_{l-1} \quad (1)$$

**ResNet** 的一个优点是梯度可以直接通过标识函数 (identity function) 从后面的层流向前面的层。但是，标识函数 (identity function) 和  $H_l$  层的输出通过求和进行组合，这可能会阻碍网络中信息的流动。

## Dense 连接

为了进一步地层与层之间的信息流，**DenseNet** 提出了一个不同的连接模型：对于每一层，都添加一个跳接到其他所有之后的层。**Figure 1**表示了 **DenseNet** 连接的方式。因此， $l^{th}$  层网络接受了所有之前层的特征图  $X_0, \dots, X_{l-1}$  作为输入：

$$X_l = H_l([X_0, X_1, \dots, X_{l-1}]) \quad (2)$$

其中  $[X_0, X_1, \dots, X_{l-1}]$  表示的是  $0, \dots, l - 1$  层得到的特征图拼接的结果。

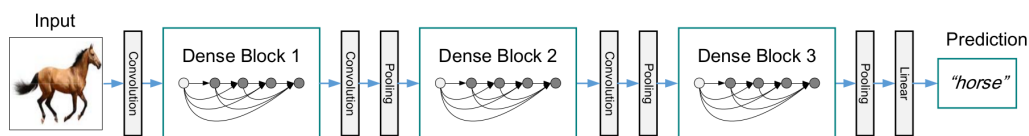
## Composite function

$H_l(\cdot)$  表示的是三个连续的操作：

- batch normalization (BN)
- rectified linear unit (ReLU)
- 3 x 3 Conv

## 池化层

当特征图尺寸变化时，**式2**中的拼接操作不可行。但是，卷积网络一个重要的部分就是降采样层，用于改变特征图的尺寸。为了在 **DenseNet** 架构中实现降采样，将网络分为多个紧密连接的 **dense blocks**，如**Figure 2**所示。



**Figure 2:** A deep DenseNet with three dense blocks. The layers between two adjacent blocks are referred to as transition layers and change feature-map sizes via convolution and pooling.

将 **dense block** 之间的层叫做过渡层，在这里做卷积和池化操作。过渡层包含批量归一层和  $1 \times 1$  卷积层，紧跟一个  $2 \times 2$  平均池化层

## Growth rate

如果每个函数  $H_l$  产生  $k$  个特征图，之后的  $l^{th}$  层有  $k_0 + k \times (l - 1)$  个输入特征图，其中  $k_0$  表示输入层的通道数。**DenseNet** 和现有的网络架构最重要的区别是 **DenseNet** 层数很窄，仅有  $k = 12$ 。将  $k$  定义为网络的增长率。

## Bottleneck layers

尽管每一层都只产生  $k$  个输出特征图，仍然有许多输入。**ResNet** 中在  $3 \times 3$  卷积前使用  $1 \times 1$  卷积作为 **bottleneck** 层减少输入特征图的数量，可以提高计算效率。使用了 **Bottleneck** 的网络命名为 **DenseNet-B**。

## Compression

为了进一步使模型更加紧凑，在过渡层减少特征图的数量。如果 **dense block** 包括  $m$  个特征图，让之后的过渡层产生  $[\theta_m]$  输出特征图，其中  $0 < \theta \leq 1$  表示压缩因子。如果  $\theta = 1$ ，表示特征图数量经过过渡层保持不变。在试验中设置  $\theta = 0.5$ 。将使用了 **bottleneck** 和过渡层设置  $\theta < 1$  的网络命名为 **DenseNet-BC**

## 实现细节

在所有除了 **ImageNet** 的数据集中，实验使用的 **DenseNet** 有三个 **dense block**，每个块的层数相等。在第一个 **dense block** 之前，对输入图像进行一个带有16（或者是 **DenseNet-BC** 增长率两倍）个输出通道的卷积操作。对于卷积核大小为  $3 \times 3$  的卷积层，输入的每一侧都用一个像素进行零填充以修正特征图尺寸。在两个连续的 **dense block** 之间使用一个  $1 \times 1$  的卷积接着一个  $2 \times 2$  的池化层组成的过渡层。在最后一个 **dense block**，使用一个全局平均池化层和一个 **softmax** 函数。在这三个 **dense block** 中的特征图分别为  $32 \times 32$ 、 $16 \times 16$  和  $8 \times 8$ 。

基本的 **DenseNet** 架构使用了以下的参数配置：

- $L = 40, k=12$
- $L = 100, k=12$
- $L = 100, k=24$

对于 DenseNet-BC，使用了以下的参数：

- $L = 100, k=12$
- $L = 250, k=24$
- $L = 190, k=40$

在 ImageNet 数据集的实验中，使用了 DenseNet-BC 结构，输入图像尺寸为  $224 \times 224$ ，dense block 有4个。初始的卷积层包含  $2k$  个步长为2的  $7 \times 7$  卷积；其他层的特征图数量遵循设置  $k$ 。ImageNet 配置如Table 1 所示

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

**Table 1:** DenseNet architectures for ImageNet. The growth rate for all the networks is  $k = 32$ . Note that each “conv” layer shown in the table corresponds the sequence BN-ReLU-Conv.

## 实验

### 数据集

#### CIFAR

训练集-50,000张图片，测试集10,000张图片，从训练集中选 5,000 张图片作为验证集。

- 使用了标准的数据增强，镜像，平移等
- 预处理使用了标准化

#### SVHN

训练集 73,257张图片，测试集26,032图片，还有531,131张图片作为额外的训练，从训练集中挑选6,000张图片作为验证集

- 没有使用任何数据增强

## ImageNet

训练集使用了1.2m张图片，50,000张图片作为验证

- 使用了标准的数据增强
- 在测试的使用应用了 **single-crop** 和 **10-crop**

## 训练

- 使用的SGD方法训练
- **CIFAR**
  - batch size 64
  - epoch 300
- **SVHN**
  - batch size 64
  - epoch 40
- 初始学习率设置为0.1，在50%和75%训练进度除以10
- **ImageNet**
  - epoch 90
  - batch size 256
  - lr 0.1, 在30和60 epoch除以10

## 结果

**CIFAR**和**SVHN**主要的结果如table 2所示

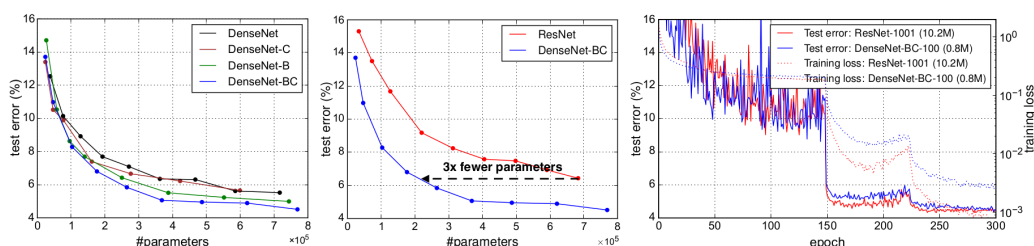
Method	Depth	Params	C10	C10+	C100	C100+	SVHN
Network in Network [22]	-	-	10.41	8.81	35.68	-	2.35
All-CNN [32]	-	-	9.08	7.25	-	33.71	-
Deeply Supervised Net [20]	-	-	9.69	7.97	-	34.57	1.92
Highway Network [34]	-	-	-	7.72	-	32.39	-
FractalNet [17]	21	38.6M	10.18	5.22	35.34	23.30	2.01
with Dropout/Drop-path	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	<b>7.00</b>	5.24	<b>27.55</b>	24.42	1.79
DenseNet ( $k = 12$ )	100	7.0M	<b>5.77</b>	<b>4.10</b>	<b>23.79</b>	<b>20.20</b>	1.67
DenseNet ( $k = 24$ )	100	27.2M	<b>5.83</b>	<b>3.74</b>	<b>23.42</b>	<b>19.25</b>	<b>1.59</b>
DenseNet-BC ( $k = 12$ )	100	0.8M	<b>5.92</b>	4.51	<b>24.15</b>	22.27	1.76
DenseNet-BC ( $k = 24$ )	250	15.3M	<b>5.19</b>	<b>3.62</b>	<b>19.64</b>	<b>17.60</b>	1.74
DenseNet-BC ( $k = 40$ )	190	25.6M	-	<b>3.46</b>	-	<b>17.18</b>	-

**Table 2:** Error rates (%) on CIFAR and SVHN datasets.  $k$  denotes network's growth rate. Results that surpass all competing methods are **bold** and the overall best results are **blue**. "+" indicates standard data augmentation (translation and/or mirroring). \* indicates results run by ourselves. All the results of DenseNets without data augmentation (C10, C100, SVHN) are obtained using Dropout. DenseNets achieve lower error rates while using fewer parameters than ResNet. Without data augmentation, DenseNet performs better by a large margin.

在**ImageNet**分类的结果和 **ResNet** 的对比如table 3和Figure 4所示。

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

**Table 3:** The top-1 and top-5 error rates on the ImageNet validation set, with single-crop / 10-crop testing.



**Figure 4:** Left: Comparison of the parameter efficiency on C10+ between DenseNet variations. Middle: Comparison of the parameter efficiency between DenseNet-BC and (pre-activation) ResNets. DenseNet-BC requires about 1/3 of the parameters as ResNet to achieve comparable accuracy. Right: Training and testing curves of the 1001-layer pre-activation ResNet [12] with more than 10M parameters and a 100-layer DenseNet with only 0.8M parameters.

## 实现

这里的仿真代码基本上参照的是这个仓库：

[gpleiss/efficient\\_densenet\\_pytorch<sup>\[4\]</sup>](#)，笔者还是喜欢使用 `jupyter` 调试，这里将其改为 `jupyter` 格式的代码，可以参考这个仓库 [madao33/computer-vision-learning](#)

首先导入基本模块

```

1 # import basic modules
2 import os
3 import time
4 import math
5 from torchvision import datasets, transforms
6 import torch
7 import torch.nn as nn
8 import torch.nn.functional as F
9 import torch.utils.checkpoint as cp
10 from collections import OrderedDict

```

## 数据集准备

这里使用的是 CIFAR10 数据集，通过 torchvision 下载实在是过于缓慢，所以直接下载数据集，然后在当前目录下创建一个 data 文件夹，将下载好的文件不解压直接放在这个 data 文件夹中

## 参数设置

设置的参数是参照论文中的 table 2，但是本人电脑配置较差，仅一块 GTX 1066，要完整的运行 300 epoch 大约需要耗费 4 个小时，暂时没有完整地运行，有想法的可以尝试一下

```
1 # 设置参数
2 data = 'data'
3 depth = 40
4 growth_rate = 12
5 valid_size = 5000
6 n_epochs = 300
7 batch_size = 64
8 efficient = True
9 save = './save'
```

```
1 # Get densenet configuration
2 if (depth - 4) % 3:
3     raise Exception('Invalid depth')
4 block_config = [(depth - 4) // 6 for _ in range(3)]
```

## 数据转换

```

1  # Data transforms
2  mean=[0.49139968, 0.48215841, 0.44653091]
3  stdv= [0.24703223, 0.24348513, 0.26158784]
4  train_transforms = transforms.Compose([
5      transforms.RandomCrop(32, padding=4),
6      transforms.RandomHorizontalFlip(),
7      transforms.ToTensor(),
8      transforms.Normalize(mean=mean, std=stdv),
9  ])
10 test_transforms = transforms.Compose([
11     transforms.ToTensor(),
12     transforms.Normalize(mean=mean, std=stdv),
13 ])

```

## 数据集下载

```

1  # Datasets
2  train_set = datasets.CIFAR10(data, train=True,
3                               transform=train_transforms, download=True)
4
5  test_set = datasets.CIFAR10(data, train=False,
6                              transform=test_transforms, download=False)
7
8  if valid_size:
9      valid_set = datasets.CIFAR10(data, train=True,
10                                  transform=test_transforms)
11     indices = torch.randperm(len(train_set))
12     train_indices = indices[:len(indices) - valid_size]
13     valid_indices = indices[len(indices) - valid_size:]
14     train_set = torch.utils.data.Subset(train_set,
15                                         train_indices)
16     valid_set = torch.utils.data.Subset(valid_set,
17                                         valid_indices)
18 else:
19     valid_set = None

```

## DenseNet模型



## 模型定义

```

1  def _bn_function_factory(norm, relu, conv):
2      def bn_function(*inputs):
3          concated_features = torch.cat(inputs, 1)
4          bottleneck_output =
5      conv(relu(norm(concated_features)))
6          return bottleneck_output
7
8      return bn_function
9
10 class _DenseLayer(nn.Module):
11     def __init__(self, num_input_features, growth_rate,
12 bn_size, drop_rate, efficient=False):
13         super(_DenseLayer, self).__init__()
14         self.add_module('norm1',
15 nn.BatchNorm2d(num_input_features)),
16         self.add_module('relu1', nn.ReLU(inplace=True)),
17         self.add_module('conv1',
18 nn.Conv2d(num_input_features, bn_size * growth_rate,
19             kernel_size=1, stride=1,
20             bias=False)),
21         self.add_module('norm2', nn.BatchNorm2d(bn_size *
22 growth_rate)),
23         self.add_module('relu2', nn.ReLU(inplace=True)),
24         self.add_module('conv2', nn.Conv2d(bn_size *
25 growth_rate, growth_rate,
26             kernel_size=3, stride=1, padding=1,
27             bias=False)),
28         self.drop_rate = drop_rate
29         self.efficient = efficient
30
31     def forward(self, *prev_features):
32         bn_function = _bn_function_factory(self.norm1,
33 self.relu1, self.conv1)
34         if self.efficient and
35 any(prev_feature.requires_grad for prev_feature in
36 prev_features):
37             bottleneck_output = cp.checkpoint(bn_function,
38 *prev_features)

```

```

28         else:
29             bottleneck_output = bn_function(*prev_features)
30             new_features =
31             self.conv2(self.relu2(self.norm2(bottleneck_output)))
32             if self.drop_rate > 0:
33                 new_features = F.dropout(new_features,
34                                         p=self.drop_rate, training=self.training)
35             return new_features
36
37 class _Transition(nn.Sequential):
38     def __init__(self, num_input_features,
39                 num_output_features):
40         super(_Transition, self).__init__()
41         self.add_module('norm',
42                         nn.BatchNorm2d(num_input_features))
43         self.add_module('relu', nn.ReLU(inplace=True))
44         self.add_module('conv',
45                         nn.Conv2d(num_input_features, num_output_features,
46                                 kernel_size=1,
47                                 stride=1, bias=False))
48         self.add_module('pool', nn.AvgPool2d(kernel_size=2,
49                                                stride=2))
50
51 class _DenseBlock(nn.Module):
52     def __init__(self, num_layers, num_input_features,
53                 bn_size, growth_rate, drop_rate, efficient=False):
54         super(_DenseBlock, self).__init__()
55         for i in range(num_layers):
56             layer = _DenseLayer(
57                 num_input_features + i * growth_rate,
58                 growth_rate=growth_rate,
59                 bn_size=bn_size,
60                 drop_rate=drop_rate,
61                 efficient=efficient,
62             )
63             self.add_module('denselayer%d' % (i + 1),
64                             layer)
65
66     def forward(self, init_features):

```

```

60         features = [init_features]
61         for name, layer in self.named_children():
62             new_features = layer(*features)
63             features.append(new_features)
64         return torch.cat(features, 1)
65
66
67 class DenseNet(nn.Module):
68     r"""Densenet-BC model class, based on
69     `Densely Connected Convolutional Networks`
70     <https://arxiv.org/pdf/1608.06993.pdf>`
71     Args:
72         growth_rate (int) - how many filters to add each
73         layer (`k` in paper)
74         block_config (list of 3 or 4 ints) - how many
75         layers in each pooling block
76         num_init_features (int) - the number of filters to
77         learn in the first convolution layer
78         bn_size (int) - multiplicative factor for number of
79         bottle neck layers
80         (i.e. bn_size * k features in the bottleneck
81         layer)
82         drop_rate (float) - dropout rate after each dense
83         layer
84         num_classes (int) - number of classification
85         classes
86         small_inputs (bool) - set to True if images are
87         32x32. Otherwise assumes images are larger.
88         efficient (bool) - set to True to use
89         checkpointing. Much more memory efficient, but slower.
90     """
91     def __init__(self, growth_rate=12, block_config=(16,
92     16, 16), compression=0.5,
93         num_init_features=24, bn_size=4,
94         drop_rate=0,
95         num_classes=10, small_inputs=True,
96         efficient=False):
97
98         super(DenseNet, self).__init__()
99         assert 0 < compression <= 1, 'compression of
100         densenet should be between 0 and 1'

```

```

87
88     # First convolution
89     if small_inputs:
90         self.features = nn.Sequential(OrderedDict([
91             ('conv0', nn.Conv2d(3, num_init_features,
92 kernel_size=3, stride=1, padding=1, bias=False)),
93         ]))
94     else:
95         self.features = nn.Sequential(OrderedDict([
96             ('conv0', nn.Conv2d(3, num_init_features,
97 kernel_size=7, stride=2, padding=3, bias=False)),
98         ]))
99         self.features.add_module('norm0',
nn.BatchNorm2d(num_init_features))
100         self.features.add_module('relu0',
nn.ReLU(inplace=True))
101         self.features.add_module('pool0',
nn.MaxPool2d(kernel_size=3, stride=2, padding=1,
102 ceil_mode=False))
103
104     # Each denseblock
105     num_features = num_init_features
106     for i, num_layers in enumerate(block_config):
107         block = _DenseBlock(
108             num_layers=num_layers,
109             num_input_features=num_features,
110             bn_size=bn_size,
111             growth_rate=growth_rate,
112             drop_rate=drop_rate,
113             efficient=efficient,
114         )
115         self.features.add_module('denseblock%d' % (i +
116 1), block)
117         num_features = num_features + num_layers *
growth_rate
118     if i != len(block_config) - 1:
119         trans =
120         _Transition(num_input_features=num_features,
121 num_output_features=int(num_features * compression))

```

```

118         self.features.add_module('transition%d' %
    (i + 1), trans)
119         num_features = int(num_features *
    compression)
120
121     # Final batch norm
122     self.features.add_module('norm_final',
    nn.BatchNorm2d(num_features))
123
124     # Linear layer
125     self.classifier = nn.Linear(num_features,
    num_classes)
126
127     # Initialization
128     for name, param in self.named_parameters():
129         if 'conv' in name and 'weight' in name:
130             n = param.size(0) * param.size(2) *
    param.size(3)
131             param.data.normal_().mul_(math.sqrt(2. /
    n))
132
133         elif 'norm' in name and 'weight' in name:
134             param.data.fill_(1)
135         elif 'norm' in name and 'bias' in name:
136             param.data.fill_(0)
137         elif 'classifier' in name and 'bias' in name:
138             param.data.fill_(0)
139
140     def forward(self, x):
141         features = self.features(x)
142         out = F.relu(features, inplace=True)
143         out = F.adaptive_avg_pool2d(out, (1, 1))
144         out = torch.flatten(out, 1)
145         out = self.classifier(out)
146         return out

```

## 模型调用

```

1 # Models
2 model = DenseNet(
3     growth_rate=growth_rate,
4     block_config=block_config,
5     num_init_features=growth_rate*2,
6     num_classes=10,
7     small_inputs=True,
8     efficient=efficient,
9 )
10 print(model)

```

```

1 DenseNet(
2   (features): Sequential(
3     (conv0): Conv2d(3, 24, kernel_size=(3, 3), stride=(1,
4       1), padding=(1, 1), bias=False)
5     (denseblock1): _DenseBlock(
6       (denselayer1): _DenseLayer(
7         (norm1): BatchNorm2d(24, eps=1e-05, momentum=0.1,
8         affine=True, track_running_stats=True)
9         (relu1): ReLU(inplace=True)
10        (conv1): Conv2d(24, 48, kernel_size=(1, 1), stride=
11        (1, 1), bias=False)
12        (norm2): BatchNorm2d(48, eps=1e-05, momentum=0.1,
13        affine=True, track_running_stats=True)
14        (relu2): ReLU(inplace=True)
15        (conv2): Conv2d(48, 12, kernel_size=(3, 3), stride=
16        (1, 1), padding=(1, 1), bias=False)
17      )
18      (denselayer2): _DenseLayer(
19        (norm1): BatchNorm2d(36, eps=1e-05, momentum=0.1,
20        affine=True, track_running_stats=True)
21        (relu1): ReLU(inplace=True)
22        (conv1): Conv2d(36, 48, kernel_size=(1, 1), stride=
23        (1, 1), bias=False)
24        (norm2): BatchNorm2d(48, eps=1e-05, momentum=0.1,
25        affine=True, track_running_stats=True)
26        (relu2): ReLU(inplace=True)
27        (conv2): Conv2d(48, 12, kernel_size=(3, 3), stride=
28        (1, 1), padding=(1, 1), bias=False)
29      )
30    )
31  )

```

```
21         (denselayer3): _DenseLayer(
22             (norm1): BatchNorm2d(48, eps=1e-05, momentum=0.1,
affine=True, track_running_stats=True)
23             (relu1): ReLU(inplace=True)
24             (conv1): Conv2d(48, 48, kernel_size=(1, 1), stride=
(1, 1), bias=False)
25             (norm2): BatchNorm2d(48, eps=1e-05, momentum=0.1,
affine=True, track_running_stats=True)
26             (relu2): ReLU(inplace=True)
27             (conv2): Conv2d(48, 12, kernel_size=(3, 3), stride=
(1, 1), padding=(1, 1), bias=False)
28         )
29         (denselayer4): _DenseLayer(
30             (norm1): BatchNorm2d(60, eps=1e-05, momentum=0.1,
affine=True, track_running_stats=True)
31             (relu1): ReLU(inplace=True)
32             (conv1): Conv2d(60, 48, kernel_size=(1, 1), stride=
(1, 1), bias=False)
33             (norm2): BatchNorm2d(48, eps=1e-05, momentum=0.1,
affine=True, track_running_stats=True)
34             (relu2): ReLU(inplace=True)
35             (conv2): Conv2d(48, 12, kernel_size=(3, 3), stride=
(1, 1), padding=(1, 1), bias=False)
36         )
37         (denselayer5): _DenseLayer(
38             (norm1): BatchNorm2d(72, eps=1e-05, momentum=0.1,
affine=True, track_running_stats=True)
39             (relu1): ReLU(inplace=True)
40             (conv1): Conv2d(72, 48, kernel_size=(1, 1), stride=
(1, 1), bias=False)
41             (norm2): BatchNorm2d(48, eps=1e-05, momentum=0.1,
affine=True, track_running_stats=True)
42             (relu2): ReLU(inplace=True)
43             (conv2): Conv2d(48, 12, kernel_size=(3, 3), stride=
(1, 1), padding=(1, 1), bias=False)
44         )
45         (denselayer6): _DenseLayer(
46             (norm1): BatchNorm2d(84, eps=1e-05, momentum=0.1,
affine=True, track_running_stats=True)
47             (relu1): ReLU(inplace=True)
```

```
48         (conv1): Conv2d(84, 48, kernel_size=(1, 1), stride=
(1, 1), bias=False)
49         (norm2): BatchNorm2d(48, eps=1e-05, momentum=0.1,
affine=True, track_running_stats=True)
50         (relu2): ReLU(inplace=True)
51         (conv2): Conv2d(48, 12, kernel_size=(3, 3), stride=
(1, 1), padding=(1, 1), bias=False)
52     )
53 )
54     (transition1): _Transition(
55         (norm): BatchNorm2d(96, eps=1e-05, momentum=0.1,
affine=True, track_running_stats=True)
56         (relu): ReLU(inplace=True)
57         (conv): Conv2d(96, 48, kernel_size=(1, 1), stride=(1,
1), bias=False)
58         (pool): AvgPool2d(kernel_size=2, stride=2, padding=0)
59     )
60     (denseblock2): _DenseBlock(
61         (denselayer1): _DenseLayer(
62             (norm1): BatchNorm2d(48, eps=1e-05, momentum=0.1,
affine=True, track_running_stats=True)
63             (relu1): ReLU(inplace=True)
64             (conv1): Conv2d(48, 48, kernel_size=(1, 1), stride=
(1, 1), bias=False)
65             (norm2): BatchNorm2d(48, eps=1e-05, momentum=0.1,
affine=True, track_running_stats=True)
66             (relu2): ReLU(inplace=True)
67             (conv2): Conv2d(48, 12, kernel_size=(3, 3), stride=
(1, 1), padding=(1, 1), bias=False)
68         )
69         (denselayer2): _DenseLayer(
70             (norm1): BatchNorm2d(60, eps=1e-05, momentum=0.1,
affine=True, track_running_stats=True)
71             (relu1): ReLU(inplace=True)
72             (conv1): Conv2d(60, 48, kernel_size=(1, 1), stride=
(1, 1), bias=False)
73             (norm2): BatchNorm2d(48, eps=1e-05, momentum=0.1,
affine=True, track_running_stats=True)
74             (relu2): ReLU(inplace=True)
75             (conv2): Conv2d(48, 12, kernel_size=(3, 3), stride=
(1, 1), padding=(1, 1), bias=False)
```



```
76         )
77         (denselayer3): _DenseLayer(
78             (norm1): BatchNorm2d(72, eps=1e-05, momentum=0.1,
affine=True, track_running_stats=True)
79             (relu1): ReLU(inplace=True)
80             (conv1): Conv2d(72, 48, kernel_size=(1, 1), stride=
(1, 1), bias=False)
81             (norm2): BatchNorm2d(48, eps=1e-05, momentum=0.1,
affine=True, track_running_stats=True)
82             (relu2): ReLU(inplace=True)
83             (conv2): Conv2d(48, 12, kernel_size=(3, 3), stride=
(1, 1), padding=(1, 1), bias=False)
84         )
85         (denselayer4): _DenseLayer(
86             (norm1): BatchNorm2d(84, eps=1e-05, momentum=0.1,
affine=True, track_running_stats=True)
87             (relu1): ReLU(inplace=True)
88             (conv1): Conv2d(84, 48, kernel_size=(1, 1), stride=
(1, 1), bias=False)
89             (norm2): BatchNorm2d(48, eps=1e-05, momentum=0.1,
affine=True, track_running_stats=True)
90             (relu2): ReLU(inplace=True)
91             (conv2): Conv2d(48, 12, kernel_size=(3, 3), stride=
(1, 1), padding=(1, 1), bias=False)
92         )
93         (denselayer5): _DenseLayer(
94             (norm1): BatchNorm2d(96, eps=1e-05, momentum=0.1,
affine=True, track_running_stats=True)
95             (relu1): ReLU(inplace=True)
96             (conv1): Conv2d(96, 48, kernel_size=(1, 1), stride=
(1, 1), bias=False)
97             (norm2): BatchNorm2d(48, eps=1e-05, momentum=0.1,
affine=True, track_running_stats=True)
98             (relu2): ReLU(inplace=True)
99             (conv2): Conv2d(48, 12, kernel_size=(3, 3), stride=
(1, 1), padding=(1, 1), bias=False)
100         )
101         (denselayer6): _DenseLayer(
102             (norm1): BatchNorm2d(108, eps=1e-05, momentum=0.1,
affine=True, track_running_stats=True)
103             (relu1): ReLU(inplace=True)
```

```
104         (conv1): Conv2d(108, 48, kernel_size=(1, 1),
stride=(1, 1), bias=False)
105         (norm2): BatchNorm2d(48, eps=1e-05, momentum=0.1,
affine=True, track_running_stats=True)
106         (relu2): ReLU(inplace=True)
107         (conv2): Conv2d(48, 12, kernel_size=(3, 3), stride=
(1, 1), padding=(1, 1), bias=False)
108     )
109 )
110     (transition2): _Transition(
111         (norm): BatchNorm2d(120, eps=1e-05, momentum=0.1,
affine=True, track_running_stats=True)
112         (relu): ReLU(inplace=True)
113         (conv): Conv2d(120, 60, kernel_size=(1, 1), stride=
(1, 1), bias=False)
114         (pool): AvgPool2d(kernel_size=2, stride=2, padding=0)
115     )
116     (denseblock3): _DenseBlock(
117         (denselayer1): _DenseLayer(
118             (norm1): BatchNorm2d(60, eps=1e-05, momentum=0.1,
affine=True, track_running_stats=True)
119             (relu1): ReLU(inplace=True)
120             (conv1): Conv2d(60, 48, kernel_size=(1, 1), stride=
(1, 1), bias=False)
121             (norm2): BatchNorm2d(48, eps=1e-05, momentum=0.1,
affine=True, track_running_stats=True)
122             (relu2): ReLU(inplace=True)
123             (conv2): Conv2d(48, 12, kernel_size=(3, 3), stride=
(1, 1), padding=(1, 1), bias=False)
124         )
125         (denselayer2): _DenseLayer(
126             (norm1): BatchNorm2d(72, eps=1e-05, momentum=0.1,
affine=True, track_running_stats=True)
127             (relu1): ReLU(inplace=True)
128             (conv1): Conv2d(72, 48, kernel_size=(1, 1), stride=
(1, 1), bias=False)
129             (norm2): BatchNorm2d(48, eps=1e-05, momentum=0.1,
affine=True, track_running_stats=True)
130             (relu2): ReLU(inplace=True)
131             (conv2): Conv2d(48, 12, kernel_size=(3, 3), stride=
(1, 1), padding=(1, 1), bias=False)
```

```
132         )
133         (denselayer3): _DenseLayer(
134             (norm1): BatchNorm2d(84, eps=1e-05, momentum=0.1,
affine=True, track_running_stats=True)
135             (relu1): ReLU(inplace=True)
136             (conv1): Conv2d(84, 48, kernel_size=(1, 1), stride=
(1, 1), bias=False)
137             (norm2): BatchNorm2d(48, eps=1e-05, momentum=0.1,
affine=True, track_running_stats=True)
138             (relu2): ReLU(inplace=True)
139             (conv2): Conv2d(48, 12, kernel_size=(3, 3), stride=
(1, 1), padding=(1, 1), bias=False)
140         )
141         (denselayer4): _DenseLayer(
142             (norm1): BatchNorm2d(96, eps=1e-05, momentum=0.1,
affine=True, track_running_stats=True)
143             (relu1): ReLU(inplace=True)
144             (conv1): Conv2d(96, 48, kernel_size=(1, 1), stride=
(1, 1), bias=False)
145             (norm2): BatchNorm2d(48, eps=1e-05, momentum=0.1,
affine=True, track_running_stats=True)
146             (relu2): ReLU(inplace=True)
147             (conv2): Conv2d(48, 12, kernel_size=(3, 3), stride=
(1, 1), padding=(1, 1), bias=False)
148         )
149         (denselayer5): _DenseLayer(
150             (norm1): BatchNorm2d(108, eps=1e-05, momentum=0.1,
affine=True, track_running_stats=True)
151             (relu1): ReLU(inplace=True)
152             (conv1): Conv2d(108, 48, kernel_size=(1, 1),
stride=(1, 1), bias=False)
153             (norm2): BatchNorm2d(48, eps=1e-05, momentum=0.1,
affine=True, track_running_stats=True)
154             (relu2): ReLU(inplace=True)
155             (conv2): Conv2d(48, 12, kernel_size=(3, 3), stride=
(1, 1), padding=(1, 1), bias=False)
156         )
157         (denselayer6): _DenseLayer(
158             (norm1): BatchNorm2d(120, eps=1e-05, momentum=0.1,
affine=True, track_running_stats=True)
159             (relu1): ReLU(inplace=True)
```

```

160         (conv1): Conv2d(120, 48, kernel_size=(1, 1),
stride=(1, 1), bias=False)
161         (norm2): BatchNorm2d(48, eps=1e-05, momentum=0.1,
affine=True, track_running_stats=True)
162         (relu2): ReLU(inplace=True)
163         (conv2): Conv2d(48, 12, kernel_size=(3, 3), stride=
(1, 1), padding=(1, 1), bias=False)
164     )
165 )
166     (norm_final): BatchNorm2d(132, eps=1e-05, momentum=0.1,
affine=True, track_running_stats=True)
167 )
168     (classifier): Linear(in_features=132, out_features=10,
bias=True)
169 )
170

```

```

1 # Print number of parameters
2 num_params = sum(p.numel() for p in model.parameters())
3 print("Total parameters: ", num_params)

```

```

1 # Make save directory
2 if not os.path.exists(save):
3     os.makedirs(save)
4 if not os.path.isdir(save):
5     raise Exception('%s is not a dir' % save)

```

## 训练

### 定义训练函数

```

1 class AverageMeter(object):
2     """
3     Computes and stores the average and current value
4     Copied from:
5     https://github.com/pytorch/examples/blob/master/imagenet/main.py
6     """
7     def __init__(self):
8         self.reset()
9     def reset(self):

```

```

10         self.val = 0
11         self.avg = 0
12         self.sum = 0
13         self.count = 0
14
15     def update(self, val, n=1):
16         self.val = val
17         self.sum += val * n
18         self.count += n
19         self.avg = self.sum / self.count
20
21 def train_epoch(model, loader, optimizer, epoch, n_epochs,
22                print_freq=1):
23     batch_time = AverageMeter()
24     losses = AverageMeter()
25     error = AverageMeter()
26
27     # Model on train mode
28     model.train()
29
30     end = time.time()
31     for batch_idx, (input, target) in enumerate(loader):
32         # Create variables
33         if torch.cuda.is_available():
34             input = input.cuda()
35             target = target.cuda()
36
37         # compute output
38         output = model(input)
39         loss = torch.nn.functional.cross_entropy(output,
40                                                  target)
41
42         # measure accuracy and record loss
43         batch_size = target.size(0)
44         _, pred = output.data.cpu().topk(1, dim=1)
45         error.update(torch.ne(pred.squeeze(),
46                               target.cpu()).float().sum().item() / batch_size,
47                     batch_size)
48         losses.update(loss.item(), batch_size)
49
50     # compute gradient and do SGD step

```

```

47         optimizer.zero_grad()
48         loss.backward()
49         optimizer.step()
50
51         # measure elapsed time
52         batch_time.update(time.time() - end)
53         end = time.time()
54
55         # print stats
56         if batch_idx % print_freq == 0:
57             res = '\t'.join([
58                 'Epoch: [%d/%d]' % (epoch + 1, n_epochs),
59                 'Iter: [%d/%d]' % (batch_idx + 1,
len(loader)),
60                 'Time %.3f (%.3f)' % (batch_time.val,
batch_time.avg),
61                 'Loss %.4f (%.4f)' % (losses.val,
losses.avg),
62                 'Error %.4f (%.4f)' % (error.val,
error.avg),
63             ])
64             print(res)
65
66         # Return summary statistics
67         return batch_time.avg, losses.avg, error.avg
68
69
70 def test_epoch(model, loader, print_freq=1, is_test=True):
71     batch_time = AverageMeter()
72     losses = AverageMeter()
73     error = AverageMeter()
74
75     # Model on eval mode
76     model.eval()
77
78     end = time.time()
79     with torch.no_grad():
80         for batch_idx, (input, target) in
enumerate(loader):
81             # Create variables
82             if torch.cuda.is_available():

```

```

83         input = input.cuda()
84         target = target.cuda()
85
86         # compute output
87         output = model(input)
88         loss =
torch.nn.functional.cross_entropy(output, target)
89
90         # measure accuracy and record loss
91         batch_size = target.size(0)
92         _, pred = output.data.cpu().topk(1, dim=1)
93         error.update(torch.ne(pred.squeeze(),
target.cpu()).float().sum().item() / batch_size,
batch_size)
94         losses.update(loss.item(), batch_size)
95
96         # measure elapsed time
97         batch_time.update(time.time() - end)
98         end = time.time()
99
100        # print stats
101        if batch_idx % print_freq == 0:
102            res = '\t'.join([
103                'Test' if is_test else 'Valid',
104                'Iter: [%d/%d]' % (batch_idx + 1,
len(loader)),
105                'Time %.3f (%.3f)' % (batch_time.val,
batch_time.avg),
106                'Loss %.4f (%.4f)' % (losses.val,
losses.avg),
107                'Error %.4f (%.4f)' % (error.val,
error.avg),
108            ])
109            print(res)
110
111        # Return summary statistics
112        return batch_time.avg, losses.avg, error.avg
113
114    def train(model, train_set, valid_set, test_set, save,
n_epochs=300,

```

```

115         batch_size=64, lr=0.1, wd=0.0001, momentum=0.9,
seed=None):
116     if seed is not None:
117         torch.manual_seed(seed)
118
119     # Data loaders
120     train_loader = torch.utils.data.DataLoader(train_set,
batch_size=batch_size, shuffle=True,
121
pin_memory=
(torch.cuda.is_available()), num_workers=0)
122     test_loader = torch.utils.data.DataLoader(test_set,
batch_size=batch_size, shuffle=False,
123
pin_memory=
(torch.cuda.is_available()), num_workers=0)
124     if valid_set is None:
125         valid_loader = None
126     else:
127         valid_loader =
torch.utils.data.DataLoader(valid_set,
batch_size=batch_size, shuffle=False,
128
pin_memory=(torch.cuda.is_available()), num_workers=0)
129     # Model on cuda
130     if torch.cuda.is_available():
131         model = model.cuda()
132
133     # Wrap model for multi-GPUs, if necessary
134     model_wrapper = model
135     if torch.cuda.is_available() and
torch.cuda.device_count() > 1:
136         model_wrapper = torch.nn.DataParallel(model).cuda()
137
138     # Optimizer
139     optimizer = torch.optim.SGD(model_wrapper.parameters(),
lr=lr, momentum=momentum, nesterov=True, weight_decay=wd)
140     scheduler =
torch.optim.lr_scheduler.MultiStepLR(optimizer, milestones=
[0.5 * n_epochs, 0.75 * n_epochs],
141
gamma=0.1)
142

```



```

143     # Start log
144     with open(os.path.join(save, 'results.csv'), 'w') as f:
145
146         f.write('epoch,train_loss,train_error,valid_loss,valid_err
147         or,test_error\n')
148
149     # Train model
150     best_error = 1
151     for epoch in range(n_epochs):
152         _, train_loss, train_error = train_epoch(
153             model=model_wrapper,
154             loader=train_loader,
155             optimizer=optimizer,
156             epoch=epoch,
157             n_epochs=n_epochs,
158         )
159         scheduler.step()
160         _, valid_loss, valid_error = test_epoch(
161             model=model_wrapper,
162             loader=valid_loader if valid_loader else
163             test_loader,
164             is_test=(not valid_loader)
165         )
166
167     # Determine if model is the best
168     if valid_loader:
169         if valid_error < best_error:
170             best_error = valid_error
171             print('New best error: %.4f' % best_error)
172             torch.save(model.state_dict(),
173                 os.path.join(save, 'model.dat'))
174         else:
175             torch.save(model.state_dict(),
176                 os.path.join(save, 'model.dat'))
177
178     # Log results
179     with open(os.path.join(save, 'results.csv'), 'a')
180     as f:
181         f.write('%03d,%.6f,%.6f,%.5f,%.5f,\n' % (
182             (epoch + 1),
183             train_loss,

```

```

178         train_error,
179         valid_loss,
180         valid_error,
181     ))
182
183     # Final test of model on test set
184     model.load_state_dict(torch.load(os.path.join(save,
185         'model.dat'))))
186     if torch.cuda.is_available() and
187     torch.cuda.device_count() > 1:
188         model = torch.nn.DataParallel(model).cuda()
189     test_results = test_epoch(
190         model=model,
191         loader=test_loader,
192         is_test=True
193     )
194     _, _, test_error = test_results
195     with open(os.path.join(save, 'results.csv'), 'a') as f:
196         f.write(',,,,,%0.5f\n' % (test_error))
197     print('Final test error: %.4f' % test_error)

```

## 开始训练

```

1 # Train the model
2 train(model=model, train_set=train_set, valid_set=valid_set,
3     test_set=test_set, save=save,
4     n_epochs=n_epochs, batch_size=batch_size)
5 print('Done!')

```

```

1 Epoch: [1/300]  Iter: [1/704]  Time 0.071 (0.071)  Loss
2 2.3249 (2.3249)  Error 0.9062 (0.9062)
3 Epoch: [1/300]  Iter: [2/704]  Time 0.100 (0.085)  Loss
4 2.3424 (2.3336)  Error 0.9375 (0.9219)
5 Epoch: [1/300]  Iter: [3/704]  Time 0.096 (0.089)  Loss
6 2.2885 (2.3186)  Error 0.8438 (0.8958)
7 Epoch: [1/300]  Iter: [4/704]  Time 0.097 (0.091)  Loss
8 2.3133 (2.3173)  Error 0.8906 (0.8945)
9 Epoch: [1/300]  Iter: [5/704]  Time 0.093 (0.091)  Loss
10 2.3092 (2.3157)  Error 0.8750 (0.8906)
11 ...

```

可以看到这次训练一个 `batch size` 需要0.1秒左右，总共有704个 `batch`，并且包含了300个 `epoch`，总共需要的时间就是  $0.1 \times 704 \times 300 = 21120s = 352m = 5.87h$ ，时间实在是太长了，尝试了下 `colab`，速度差不多，啥时候有空了再完整的运行一次。

## 参考

---

1. He K , Zhang X , Ren S , et al. Deep Residual Learning for Image Recognition[J]. 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 2016.
2. Szegedy C , Liu W , Jia Y , et al. Going Deeper with Convolutions[J]. IEEE Computer Society, 2014.
3. Huang G , Liu Z , Laurens V , et al. Densely Connected Convolutional Networks[J]. IEEE Computer Society, 2016.
4. [gpleiss/efficient\\_densenet\\_pytorch](#)