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

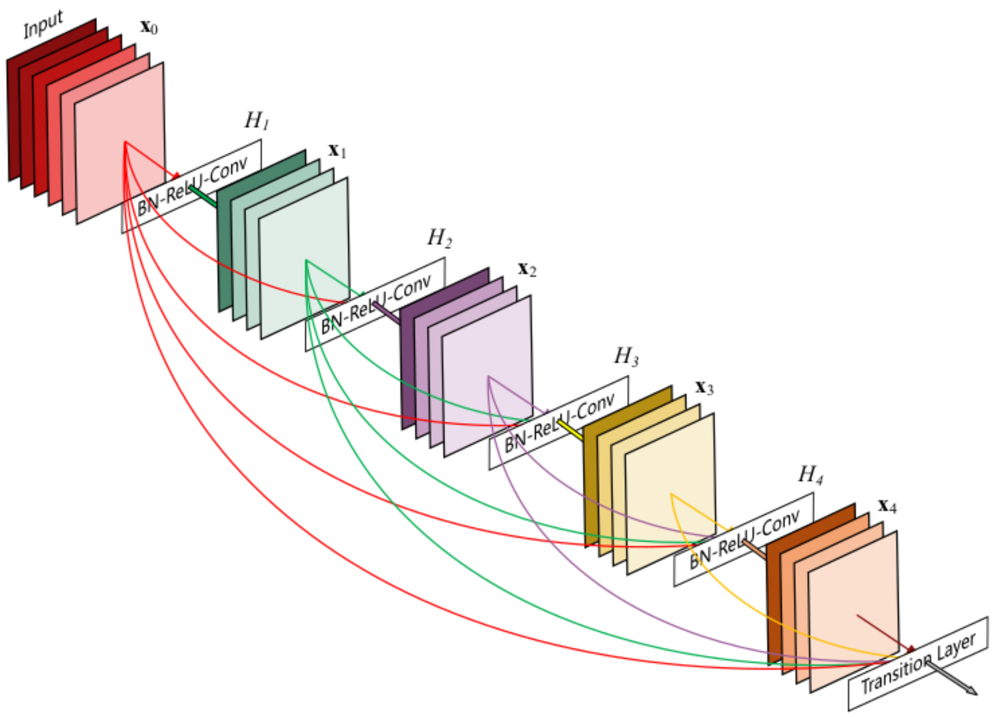

Figure 1: A 5-layer dense block with a growth rate of $k = 4$. Each layer takes all preceding feature-maps as input.

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

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

`ResNet` 的一个优点是梯度可以直接通过标识函数（identify function）从后面的层流向前面的层。但是，标识函数（identify 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 $X_0, \ldots, X_{l-1}$ 作为输入：

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

其中 $[X_0, X_1, \ldots, X_{l-1}]$ 表示的是 $0, \ldots, l-1$ 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。

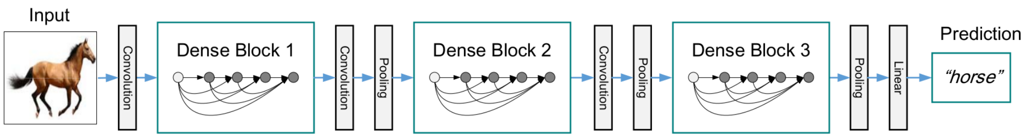

Figure 2: A deep DenseNet with three dense blocks. The layers between two adjacent blocks are referred to as transition layers and change feature-map sizes via convolution and pooling.

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 x 1 卷积层，紧跟一个 2 x 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 x 3卷积前使用 1 x 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 $\lfloor \theta_m \rfloor$ 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 x 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 x 1 的卷积接着一个 2 x 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 x 32、 16 x 16和8 x 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 x 224，`dense block` 有4个。初始的卷积层包含 2k 个步长为2的7 x 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	112×112	7×7 conv, stride 2			
Pooling	56×56	3×3 max pool, stride 2			
Dense Block (1)	56×56	$\begin{bmatrix}1\times1\text{ conv}\\3\times3\text{ conv}\end{bmatrix}\times6$	$\begin{bmatrix}1\times1\text{ conv}\\3\times3\text{ conv}\end{bmatrix}\times6$	$\begin{bmatrix}1\times1\text{ conv}\\3\times3\text{ conv}\end{bmatrix}\times6$	$\begin{bmatrix}1\times1\text{ conv}\\3\times3\text{ conv}\end{bmatrix}\times6$
Transition Layer (1)	56×56	1×1 conv			
	28×28	2×2 average pool, stride 2			
Dense Block (2)	28×28	$\begin{bmatrix}1\times1\text{ conv}\\3\times3\text{ conv}\end{bmatrix}\times12$	$\begin{bmatrix}1\times1\text{ conv}\\3\times3\text{ conv}\end{bmatrix}\times12$	$\begin{bmatrix}1\times1\text{ conv}\\3\times3\text{ conv}\end{bmatrix}\times12$	$\begin{bmatrix}1\times1\text{ conv}\\3\times3\text{ conv}\end{bmatrix}\times12$
Transition Layer (2)	28×28	1×1 conv			
	14×14	2×2 average pool, stride 2			
Dense Block (3)	14×14	$\begin{bmatrix}1\times1\text{ conv}\\3\times3\text{ conv}\end{bmatrix}\times24$	$\begin{bmatrix}1\times1\text{ conv}\\3\times3\text{ conv}\end{bmatrix}\times32$	$\begin{bmatrix}1\times1\text{ conv}\\3\times3\text{ conv}\end{bmatrix}\times48$	$\begin{bmatrix}1\times1\text{ conv}\\3\times3\text{ conv}\end{bmatrix}\times64$
Transition Layer (3)	14×14	1×1 conv			
	7×7	2×2 average pool, stride 2			
Dense Block (4)	7×7	$\begin{bmatrix}1\times1\text{ conv}\\3\times3\text{ conv}\end{bmatrix}\times16$	$\begin{bmatrix}1\times1\text{ conv}\\3\times3\text{ conv}\end{bmatrix}\times32$	$\begin{bmatrix}1\times1\text{ conv}\\3\times3\text{ conv}\end{bmatrix}\times32$	$\begin{bmatrix}1\times1\text{ conv}\\3\times3\text{ conv}\end{bmatrix}\times48$
Classification Layer	1×1	7×7 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	**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**	-

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.

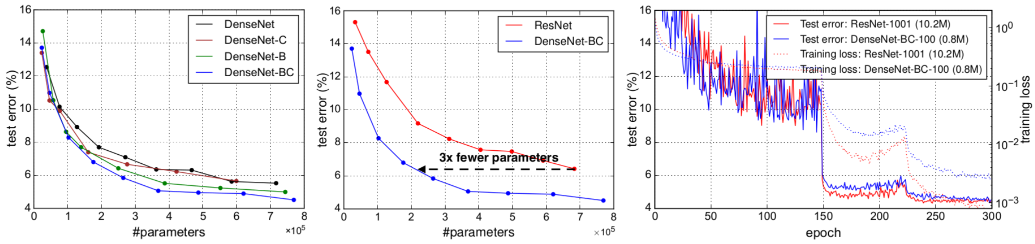

Figure 4: *Left:* Comparison of the parameter efficiency on C10+ between DenseNet variations. *Middle:* Comparison of the parameter efficiency between DenseNet-BC and (pre-activation) ResNets. DenseNet-BC requires about 1/3 of the parameters as ResNet to achieve comparable accuracy. *Right:* Training 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[4]，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

```
# import basic modules
import os
import time
import math
from torchvision import datasets, transforms
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.utils.checkpoint as cp
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

```
# 设置参数
data = 'data'
depth = 40
growth_rate = 12
valid_size = 5000
n_epochs = 300
batch_size = 64
efficient = True
save = './save'
```

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

数据转换

```python
# Data transforms
mean=[0.49139968, 0.48215841, 0.44653091]
stdv= [0.24703223, 0.24348513, 0.26158784]
train_transforms = transforms.Compose([
    transforms.RandomCrop(32, padding=4),
    transforms.RandomHorizontalFlip(),
    transforms.ToTensor(),
    transforms.Normalize(mean=mean, std=stdv),
])
test_transforms = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize(mean=mean, std=stdv),
])
```

数据集下载

```python
# Datasets
train_set = datasets.CIFAR10(data, train=True,
transform=train_transforms, download=True)
test_set = datasets.CIFAR10(data, train=False,
transform=test_transforms, download=False)

if valid_size:
    valid_set = datasets.CIFAR10(data, train=True,
transform=test_transforms)
    indices = torch.randperm(len(train_set))
    train_indices = indices[:len(indices) - valid_size]
    valid_indices = indices[len(indices) - valid_size:]
    train_set = torch.utils.data.Subset(train_set,
train_indices)
    valid_set = torch.utils.data.Subset(valid_set,
valid_indices)
else:
    valid_set = None
```

DenseNet模型

模型定义

```
def _bn_function_factory(norm, relu, conv):
    def bn_function(*inputs):
        concated_features = torch.cat(inputs, 1)
        bottleneck_output = conv(relu(norm(concated_features)))
        return bottleneck_output

    return bn_function


class _DenseLayer(nn.Module):
    def __init__(self, num_input_features, growth_rate, bn_size, drop_rate, efficient=False):
        super(_DenseLayer, self).__init__()
        self.add_module('norm1', nn.BatchNorm2d(num_input_features)),
        self.add_module('relu1', nn.ReLU(inplace=True)),
        self.add_module('conv1', nn.Conv2d(num_input_features, bn_size * growth_rate,
                        kernel_size=1, stride=1, bias=False)),
        self.add_module('norm2', nn.BatchNorm2d(bn_size * growth_rate)),
        self.add_module('relu2', nn.ReLU(inplace=True)),
        self.add_module('conv2', nn.Conv2d(bn_size * growth_rate, growth_rate,
                        kernel_size=3, stride=1, padding=1, bias=False)),
        self.drop_rate = drop_rate
        self.efficient = efficient

    def forward(self, *prev_features):
        bn_function = _bn_function_factory(self.norm1, self.relu1, self.conv1)
        if self.efficient and any(prev_feature.requires_grad for prev_feature in prev_features):
            bottleneck_output = cp.checkpoint(bn_function, *prev_features)
```

```
        else:
            bottleneck_output = bn_function(*prev_features)
        new_features = self.conv2(self.relu2(self.norm2(bottleneck_output)))
        if self.drop_rate > 0:
            new_features = F.dropout(new_features, p=self.drop_rate, training=self.training)
        return new_features


class _Transition(nn.Sequential):
    def __init__(self, num_input_features, num_output_features):
        super(_Transition, self).__init__()
        self.add_module('norm', nn.BatchNorm2d(num_input_features))
        self.add_module('relu', nn.ReLU(inplace=True))
        self.add_module('conv', nn.Conv2d(num_input_features, num_output_features,
                                          kernel_size=1, stride=1, bias=False))
        self.add_module('pool', nn.AvgPool2d(kernel_size=2, stride=2))


class _DenseBlock(nn.Module):
    def __init__(self, num_layers, num_input_features, bn_size, growth_rate, drop_rate, efficient=False):
        super(_DenseBlock, self).__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,
                efficient=efficient,
            )
            self.add_module('denselayer%d' % (i + 1), layer)

    def forward(self, init_features):
```

```python
        features = [init_features]
        for name, layer in self.named_children():
            new_features = layer(*features)
            features.append(new_features)
        return torch.cat(features, 1)


class DenseNet(nn.Module):
    r"""Densenet-BC model class, based on
    `"Densely Connected Convolutional Networks" <https://arxiv.org/pdf/1608.06993.pdf>`
    Args:
        growth_rate (int) - how many filters to add each layer (`k` in paper)
        block_config (list of 3 or 4 ints) - how many layers in each pooling block
        num_init_features (int) - the number of filters to learn in the first convolution layer
        bn_size (int) - multiplicative factor for number of bottle neck layers
            (i.e. bn_size * k features in the bottleneck layer)
        drop_rate (float) - dropout rate after each dense layer
        num_classes (int) - number of classification classes
        small_inputs (bool) - set to True if images are 32x32. Otherwise assumes images are larger.
        efficient (bool) - set to True to use checkpointing. Much more memory efficient, but slower.
    """
    def __init__(self, growth_rate=12, block_config=(16, 16, 16), compression=0.5,
                 num_init_features=24, bn_size=4, drop_rate=0,
                 num_classes=10, small_inputs=True, efficient=False):

        super(DenseNet, self).__init__()
        assert 0 < compression <= 1, 'compression of densenet should be between 0 and 1'
```

```python

        # First convolution
        if small_inputs:
            self.features = nn.Sequential(OrderedDict([
                ('conv0', nn.Conv2d(3, num_init_features, kernel_size=3, stride=1, padding=1, bias=False)),
            ]))
        else:
            self.features = nn.Sequential(OrderedDict([
                ('conv0', nn.Conv2d(3, num_init_features, kernel_size=7, stride=2, padding=3, bias=False)),
            ]))
            self.features.add_module('norm0', nn.BatchNorm2d(num_init_features))
            self.features.add_module('relu0', nn.ReLU(inplace=True))
            self.features.add_module('pool0', nn.MaxPool2d(kernel_size=3, stride=2, padding=1,
                                                           ceil_mode=False))

        # 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,
                efficient=efficient,
            )
            self.features.add_module('denseblock%d' % (i + 1), block)
            num_features = num_features + num_layers * growth_rate
            if i != len(block_config) - 1:
                trans = _Transition(num_input_features=num_features,
                                    num_output_features=int(num_features * compression))
```

```
                self.features.add_module('transition%d' % (i + 1), trans)
                num_features = int(num_features * compression)

        # Final batch norm
        self.features.add_module('norm_final', nn.BatchNorm2d(num_features))

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

        # Initialization
        for name, param in self.named_parameters():
            if 'conv' in name and 'weight' in name:
                n = param.size(0) * param.size(2) * param.size(3)
                param.data.normal_().mul_(math.sqrt(2. / n))
            elif 'norm' in name and 'weight' in name:
                param.data.fill_(1)
            elif 'norm' in name and 'bias' in name:
                param.data.fill_(0)
            elif 'classifier' in name and 'bias' in name:
                param.data.fill_(0)

    def forward(self, x):
        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
```

模型调用

```python
# Models
model = DenseNet(
    growth_rate=growth_rate,
    block_config=block_config,
    num_init_features=growth_rate*2,
    num_classes=10,
    small_inputs=True,
    efficient=efficient,
)
print(model)
```

```
DenseNet(
  (features): Sequential(
    (conv0): Conv2d(3, 24, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    (denseblock1): _DenseBlock(
      (denselayer1): _DenseLayer(
        (norm1): BatchNorm2d(24, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (relu1): ReLU(inplace=True)
        (conv1): Conv2d(24, 48, kernel_size=(1, 1), stride=(1, 1), bias=False)
        (norm2): BatchNorm2d(48, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (relu2): ReLU(inplace=True)
        (conv2): Conv2d(48, 12, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      )
      (denselayer2): _DenseLayer(
        (norm1): BatchNorm2d(36, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (relu1): ReLU(inplace=True)
        (conv1): Conv2d(36, 48, kernel_size=(1, 1), stride=(1, 1), bias=False)
        (norm2): BatchNorm2d(48, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (relu2): ReLU(inplace=True)
        (conv2): Conv2d(48, 12, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      )
```

```
      (denselayer3): _DenseLayer(
        (norm1): BatchNorm2d(48, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (relu1): ReLU(inplace=True)
        (conv1): Conv2d(48, 48, kernel_size=(1, 1), stride=(1, 1), bias=False)
        (norm2): BatchNorm2d(48, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (relu2): ReLU(inplace=True)
        (conv2): Conv2d(48, 12, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      )
      (denselayer4): _DenseLayer(
        (norm1): BatchNorm2d(60, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (relu1): ReLU(inplace=True)
        (conv1): Conv2d(60, 48, kernel_size=(1, 1), stride=(1, 1), bias=False)
        (norm2): BatchNorm2d(48, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (relu2): ReLU(inplace=True)
        (conv2): Conv2d(48, 12, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      )
      (denselayer5): _DenseLayer(
        (norm1): BatchNorm2d(72, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (relu1): ReLU(inplace=True)
        (conv1): Conv2d(72, 48, kernel_size=(1, 1), stride=(1, 1), bias=False)
        (norm2): BatchNorm2d(48, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (relu2): ReLU(inplace=True)
        (conv2): Conv2d(48, 12, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      )
      (denselayer6): _DenseLayer(
        (norm1): BatchNorm2d(84, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (relu1): ReLU(inplace=True)
```

```
        (conv1): Conv2d(84, 48, kernel_size=(1, 1), stride=(1, 1), bias=False)
        (norm2): BatchNorm2d(48, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (relu2): ReLU(inplace=True)
        (conv2): Conv2d(48, 12, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      )
    )
    (transition1): _Transition(
      (norm): BatchNorm2d(96, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv): Conv2d(96, 48, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (pool): AvgPool2d(kernel_size=2, stride=2, padding=0)
    )
    (denseblock2): _DenseBlock(
      (denselayer1): _DenseLayer(
        (norm1): BatchNorm2d(48, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (relu1): ReLU(inplace=True)
        (conv1): Conv2d(48, 48, kernel_size=(1, 1), stride=(1, 1), bias=False)
        (norm2): BatchNorm2d(48, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (relu2): ReLU(inplace=True)
        (conv2): Conv2d(48, 12, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      )
      (denselayer2): _DenseLayer(
        (norm1): BatchNorm2d(60, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (relu1): ReLU(inplace=True)
        (conv1): Conv2d(60, 48, kernel_size=(1, 1), stride=(1, 1), bias=False)
        (norm2): BatchNorm2d(48, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (relu2): ReLU(inplace=True)
        (conv2): Conv2d(48, 12, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
```

```
      )
      (denselayer3): _DenseLayer(
        (norm1): BatchNorm2d(72, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (relu1): ReLU(inplace=True)
        (conv1): Conv2d(72, 48, kernel_size=(1, 1), stride=(1, 1), bias=False)
        (norm2): BatchNorm2d(48, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (relu2): ReLU(inplace=True)
        (conv2): Conv2d(48, 12, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      )
      (denselayer4): _DenseLayer(
        (norm1): BatchNorm2d(84, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (relu1): ReLU(inplace=True)
        (conv1): Conv2d(84, 48, kernel_size=(1, 1), stride=(1, 1), bias=False)
        (norm2): BatchNorm2d(48, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (relu2): ReLU(inplace=True)
        (conv2): Conv2d(48, 12, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      )
      (denselayer5): _DenseLayer(
        (norm1): BatchNorm2d(96, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (relu1): ReLU(inplace=True)
        (conv1): Conv2d(96, 48, kernel_size=(1, 1), stride=(1, 1), bias=False)
        (norm2): BatchNorm2d(48, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (relu2): ReLU(inplace=True)
        (conv2): Conv2d(48, 12, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      )
      (denselayer6): _DenseLayer(
        (norm1): BatchNorm2d(108, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (relu1): ReLU(inplace=True)
```

```
        (conv1): Conv2d(108, 48, kernel_size=(1, 1), stride=(1, 1), bias=False)
        (norm2): BatchNorm2d(48, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (relu2): ReLU(inplace=True)
        (conv2): Conv2d(48, 12, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      )
    )
    (transition2): _Transition(
      (norm): BatchNorm2d(120, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv): Conv2d(120, 60, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (pool): AvgPool2d(kernel_size=2, stride=2, padding=0)
    )
    (denseblock3): _DenseBlock(
      (denselayer1): _DenseLayer(
        (norm1): BatchNorm2d(60, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (relu1): ReLU(inplace=True)
        (conv1): Conv2d(60, 48, kernel_size=(1, 1), stride=(1, 1), bias=False)
        (norm2): BatchNorm2d(48, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (relu2): ReLU(inplace=True)
        (conv2): Conv2d(48, 12, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      )
      (denselayer2): _DenseLayer(
        (norm1): BatchNorm2d(72, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (relu1): ReLU(inplace=True)
        (conv1): Conv2d(72, 48, kernel_size=(1, 1), stride=(1, 1), bias=False)
        (norm2): BatchNorm2d(48, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (relu2): ReLU(inplace=True)
        (conv2): Conv2d(48, 12, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
```

```
      )
      (denselayer3): _DenseLayer(
        (norm1): BatchNorm2d(84, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (relu1): ReLU(inplace=True)
        (conv1): Conv2d(84, 48, kernel_size=(1, 1), stride=(1, 1), bias=False)
        (norm2): BatchNorm2d(48, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (relu2): ReLU(inplace=True)
        (conv2): Conv2d(48, 12, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      )
      (denselayer4): _DenseLayer(
        (norm1): BatchNorm2d(96, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (relu1): ReLU(inplace=True)
        (conv1): Conv2d(96, 48, kernel_size=(1, 1), stride=(1, 1), bias=False)
        (norm2): BatchNorm2d(48, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (relu2): ReLU(inplace=True)
        (conv2): Conv2d(48, 12, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      )
      (denselayer5): _DenseLayer(
        (norm1): BatchNorm2d(108, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (relu1): ReLU(inplace=True)
        (conv1): Conv2d(108, 48, kernel_size=(1, 1), stride=(1, 1), bias=False)
        (norm2): BatchNorm2d(48, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (relu2): ReLU(inplace=True)
        (conv2): Conv2d(48, 12, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      )
      (denselayer6): _DenseLayer(
        (norm1): BatchNorm2d(120, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (relu1): ReLU(inplace=True)
```

```
            (conv1): Conv2d(120, 48, kernel_size=(1, 1), stride=(1, 1), bias=False)
            (norm2): BatchNorm2d(48, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
            (relu2): ReLU(inplace=True)
            (conv2): Conv2d(48, 12, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
          )
        )
        (norm_final): BatchNorm2d(132, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      )
      (classifier): Linear(in_features=132, out_features=10, bias=True)
    )

```

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

```
# Make save directory
if not os.path.exists(save):
    os.makedirs(save)
if not os.path.isdir(save):
    raise Exception('%s is not a dir' % save)
```

训练

定义训练函数

```
class AverageMeter(object):
    """
    Computes and stores the average and current value
    Copied from: https://github.com/pytorch/examples/blob/master/imagenet/main.py
    """
    def __init__(self):
        self.reset()

    def reset(self):
```

```
        self.val = 0
        self.avg = 0
        self.sum = 0
        self.count = 0

    def update(self, val, n=1):
        self.val = val
        self.sum += val * n
        self.count += n
        self.avg = self.sum / self.count

def train_epoch(model, loader, optimizer, epoch, n_epochs, print_freq=1):
    batch_time = AverageMeter()
    losses = AverageMeter()
    error = AverageMeter()

    # Model on train mode
    model.train()

    end = time.time()
    for batch_idx, (input, target) in enumerate(loader):
        # Create vaiables
        if torch.cuda.is_available():
            input = input.cuda()
            target = target.cuda()

        # compute output
        output = model(input)
        loss = torch.nn.functional.cross_entropy(output, target)

        # measure accuracy and record loss
        batch_size = target.size(0)
        _, pred = output.data.cpu().topk(1, dim=1)
        error.update(torch.ne(pred.squeeze(), target.cpu()).float().sum().item() / batch_size, batch_size)
        losses.update(loss.item(), batch_size)

        # compute gradient and do SGD step
```

```
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        # measure elapsed time
        batch_time.update(time.time() - end)
        end = time.time()

        # print stats
        if batch_idx % print_freq == 0:
            res = '\t'.join([
                'Epoch: [%d/%d]' % (epoch + 1, n_epochs),
                'Iter: [%d/%d]' % (batch_idx + 1, len(loader)),
                'Time %.3f (%.3f)' % (batch_time.val, batch_time.avg),
                'Loss %.4f (%.4f)' % (losses.val, losses.avg),
                'Error %.4f (%.4f)' % (error.val, error.avg),
            ])
            print(res)

    # Return summary statistics
    return batch_time.avg, losses.avg, error.avg


def test_epoch(model, loader, print_freq=1, is_test=True):
    batch_time = AverageMeter()
    losses = AverageMeter()
    error = AverageMeter()

    # Model on eval mode
    model.eval()

    end = time.time()
    with torch.no_grad():
        for batch_idx, (input, target) in enumerate(loader):
            # Create vaiables
            if torch.cuda.is_available():
```

```python
                input = input.cuda()
                target = target.cuda()

            # compute output
            output = model(input)
            loss = torch.nn.functional.cross_entropy(output, target)

            # measure accuracy and record loss
            batch_size = target.size(0)
            _, pred = output.data.cpu().topk(1, dim=1)
            error.update(torch.ne(pred.squeeze(), target.cpu()).float().sum().item() / batch_size, batch_size)
            losses.update(loss.item(), batch_size)

            # measure elapsed time
            batch_time.update(time.time() - end)
            end = time.time()

            # print stats
            if batch_idx % print_freq == 0:
                res = '\t'.join([
                    'Test' if is_test else 'Valid',
                    'Iter: [%d/%d]' % (batch_idx + 1, len(loader)),
                    'Time %.3f (%.3f)' % (batch_time.val, batch_time.avg),
                    'Loss %.4f (%.4f)' % (losses.val, losses.avg),
                    'Error %.4f (%.4f)' % (error.val, error.avg),
                ])
                print(res)

    # Return summary statistics
    return batch_time.avg, losses.avg, error.avg

def train(model, train_set, valid_set, test_set, save, n_epochs=300,
```

```
          batch_size=64, lr=0.1, wd=0.0001, momentum=0.9, seed=None):
    if seed is not None:
        torch.manual_seed(seed)

    # Data loaders
    train_loader = torch.utils.data.DataLoader(train_set, batch_size=batch_size, shuffle=True,
                                               pin_memory=(torch.cuda.is_available()), num_workers=0)
    test_loader = torch.utils.data.DataLoader(test_set, batch_size=batch_size, shuffle=False,
                                              pin_memory=(torch.cuda.is_available()), num_workers=0)
    if valid_set is None:
        valid_loader = None
    else:
        valid_loader = torch.utils.data.DataLoader(valid_set, batch_size=batch_size, shuffle=False,
                                                   pin_memory=(torch.cuda.is_available()), num_workers=0)
    # Model on cuda
    if torch.cuda.is_available():
        model = model.cuda()

    # Wrap model for multi-GPUs, if necessary
    model_wrapper = model
    if torch.cuda.is_available() and torch.cuda.device_count() > 1:
        model_wrapper = torch.nn.DataParallel(model).cuda()

    # Optimizer
    optimizer = torch.optim.SGD(model_wrapper.parameters(), lr=lr, momentum=momentum, nesterov=True, weight_decay=wd)
    scheduler = torch.optim.lr_scheduler.MultiStepLR(optimizer, milestones=[0.5 * n_epochs, 0.75 * n_epochs],
                                                     gamma=0.1)

```

```python
    # Start log
    with open(os.path.join(save, 'results.csv'), 'w') as f:
        f.write('epoch,train_loss,train_error,valid_loss,valid_error,test_error\n')

    # Train model
    best_error = 1
    for epoch in range(n_epochs):
        _, train_loss, train_error = train_epoch(
            model=model_wrapper,
            loader=train_loader,
            optimizer=optimizer,
            epoch=epoch,
            n_epochs=n_epochs,
        )
        scheduler.step()
        _, valid_loss, valid_error = test_epoch(
            model=model_wrapper,
            loader=valid_loader if valid_loader else test_loader,
            is_test=(not valid_loader)
        )

        # Determine if model is the best
        if valid_loader:
            if valid_error < best_error:
                best_error = valid_error
                print('New best error: %.4f' % best_error)
                torch.save(model.state_dict(), os.path.join(save, 'model.dat'))
        else:
            torch.save(model.state_dict(), os.path.join(save, 'model.dat'))

        # Log results
        with open(os.path.join(save, 'results.csv'), 'a') as f:
            f.write('%03d,%0.6f,%0.6f,%0.5f,%0.5f,\n' % (
                (epoch + 1),
                train_loss,
```

```python
                train_error,
                valid_loss,
                valid_error,
            ))

    # Final test of model on test set
    model.load_state_dict(torch.load(os.path.join(save, 'model.dat')))
    if torch.cuda.is_available() and torch.cuda.device_count() > 1:
        model = torch.nn.DataParallel(model).cuda()
    test_results = test_epoch(
        model=model,
        loader=test_loader,
        is_test=True
    )
    _, _, test_error = test_results
    with open(os.path.join(save, 'results.csv'), 'a') as f:
        f.write(',,,,,%0.5f\n' % (test_error))
    print('Final test error: %.4f' % test_error)
```

开始训练

```python
# Train the model
train(model=model, train_set=train_set, valid_set=valid_set, test_set=test_set, save=save,
        n_epochs=n_epochs, batch_size=batch_size)
print('Done!')
```

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

> 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