一、模型分析

如果卷积神经网络包含接近输入层和接近输出层之间的较短连接, 则卷积神经网络可以更精确有效地进行训练。随着网络层数的加深，网络在训练过程中的前传信号和梯度信号在经过很多层之后可能会逐渐消失。DenseNet模型设计了一种全新的连接模式。为了最大化网络中所有层之间的信息流，将网络中的所有层两两都进行了连接，使得网络中每一层都接受它前面所有层的特征作为输入。由于网络中存在着大量密集的连接，这种网络结构称为 DenseNet。

本文基于密集卷积网络（Densely Connected Convolutional Networks, DenseNet）, 以前馈方式将各层连接。在传统L层的卷积网络中具有L个连接，这个连接在每个层与其后的连接层之间, 而在DenseNet结构中有个直连接。对于每一层，所有前面的层的特征映射被用作输入, 并且它自己的特征映射被用作所有后续层的输入。DenseNet结构减轻消失梯度问题、加强特征传播、鼓励特征重用、大幅减少参数数量。

考虑一个通过卷积网络传送的图像, 该网络包括层, 其中每个实现非线性变换,表示层。可以是复合函数, 例如BN, ReLU, Pooling, Conv。我们将输出层表示为。

1. **ResNets.**

传统卷积神经网络将输出层作为输入连接到层, 这引起了下面的层转换：。ResNet添加一个跳过连接, 使用公式(3)函数绕过非线性转换：

 (3)

ResNets的一个优点是梯度可以直接通过身份函数从后面的层流向前面的层。然而, 的身份函数和输出通过求和相结合, 这可能会阻碍网络中的信息流。

1. **Dense connectivity.**

密集连接。为了进一步改善层之间的信息流, 我们提出了不同的连接模式：我们引入了从任何层到所有后续层的直接连接。图1展示了由此产生的DenseNet的布局。因此, 层接收所有前面的层的特征映射作为输入：

 (4)

其中指0到层中产生的特征映射的拼接。由于密集连通性, 我们将这种网络结构称为密集卷积网络（Dense Convolutional Network, DenseNet）。为了便于实现, 我们在等式中连接的多个输入变成一个张量。

图片包含 文字, 地图

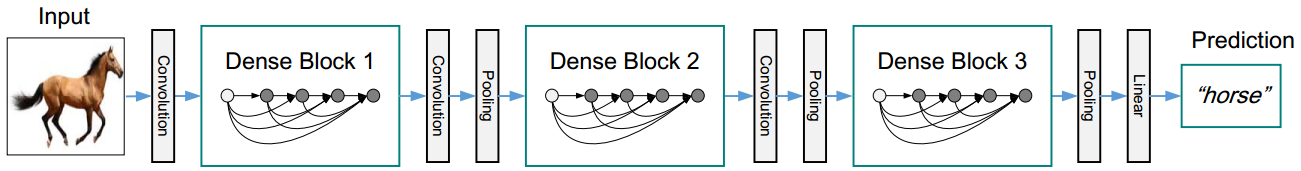
已生成极高可信度的说明

**图1 DenseNet结构图**

1. **Composite function.**

复合功能。我们将定义为三个连续操作的复合函数：批量归一化(BN), 然后是一个经整流的线性单元（ReLU）和3×3卷积（Conv）。

1. **Pooling layers.**

合并图层。方程中使用的连接操作在特征映射的大小改变时是不可行的。然而, 卷积网络的一个重要部分是下采样层, 它改变了要素图的大小。为了便于在我们的体系结构中进行下采样, 我们将网络划分为多个密集连接的密集块; 如图2所示。我们将块之间的层称为过渡层, 它们进行卷积和合并。在我们的实验中使用的过渡层包括批量正规化层和1×1卷积层, 然后是2×2平均池层。

**图2 密集连接块**

1. **Growth rate.**

增长率。如果每个函数产生k个特征映射, 则层具有个输入特征映射, 其中是输入层中的通道数量。DenseNet和现有网络架构之间的一个重要区别是DenseNet可以有非常窄的层次, 例如k = 12。我们将参数k称为网络的增长率。每一层都可以访问其块中的所有前面的特征映射, 因此可以访问网络的“集体知识”。可以将特征图视为网络的全局状态。每个图层都将自己的k个特征映射添加到这个状态。增长率规定了每个层次对全球国家贡献的新信息量。全局状态一旦写入, 就可以从网络中的任何地方访问, 与传统的网络体系结构不同, 不需要一层一层地复制

1. **Bottleneck layers.**

瓶颈层。尽管每层只能生成k个输出特征映射, 但它通常有更多的输入。在每次卷积之前可以将卷积引入瓶颈层, 以减少输入特征图的数量, 从而提高计算效率。

1. **Compression.**

压缩。为了进一步提高模型的紧凑性, 我们可以减少过渡层的特征映射数量。如果密集块包含m个特征映射, 我们让下面的过渡层生成输出特征映射, 其中被称为压缩因子。当时, 过渡层上的特征映射的数量保持不变。我们称的DenseNet为DenseNet-C。

每层以之前层的输出为输入, 对于有L层的传统网络, 一共有L个连接, 对于DenseNet, 则有。它主要拥有以下两个特性：1）一定程度上减轻在训练过程中梯度消散的问题。因为从上左图我们可以看出, 在反传时每一层都会接受其后所有层的梯度信号, 所以不会随着网络深度的增加, 靠近输入层的梯度会变得越来越小。2）由于大量的特征被复用, 使得使用少量的卷积核就可以生成大量的特征, 最终模型的尺寸也比较小。主要特征如图3所示：

1. 为了进行特征复用, 在跨层连接时使用的是在特征维度上的Concatenate 操作, 而不是 Element-wise Addition 操作。
2. 由于不需要进行Elewise-wise操作, 所以在每个单元模块的最后不需要一个的卷积来将特征层数升维到和输入的特征维度一致。
3. 采用Pre-activation的策略来设计单元, 将 BN 操作从主支上移到分支之前。（BN->ReLU->1x1Conv->BN->ReLU->3x3Conv）

**图3 特征结构图 图4 采样阶段**

1. 由于网络中每层都接受前面所有层的特征作为输入, 为了避免随着网络层数的增加, 后面层的特征维度增长过快, 在每个阶段之后进行下采样的时候, 首先通过一个卷积层将特征维度压缩至当前输入的一半, 然后再进行Pooling的操作。如图4所示。
2. 增长率的设置。增长率指的是每个单元模块最后那个的卷积核的数量, 记为k。由于每个单元模块最后是以Concatenate的方式来进行连接的, 所以每经过一个单元模块, 下一层的特征维度就会增长k。它的值越大意味着在网络中流通的信息也越大, 相应地网络的能力也越强, 但是整个模型的尺寸和计算量也会变大。

二、代码分析

**densenet网络模型构建分为两个程序：1. DenseConnectLayer.lua和2. densenet.lua**

* **DenseConnectLayer.lua #密集层连接程序**

**#数据载入**

**local function ShareGradInput(module, key)**

**assert(key)**

**module.\_\_shareGradInputKey = key**

**return module**

**End**

**#定制密集连接层（高效存储）**

**#网络初始化，设置第一层网络**

**function DenseConnectLayerCustom:\_\_init(nChannels, opt)**

**parent.\_\_init(self)**

**self.train = true**

**self.opt = opt**

**self.net1 = nn.Sequential()**

**self.net1:add(ShareGradInput(cudnn.SpatialBatchNormalization(nChannels), 'first'))**

**self.net1:add(cudnn.ReLU(true))**

**#如果存在瓶颈层，设置第二层网络，**在每次卷积之前可以将卷积引 入瓶颈层, 以减少输入特征图的数量, 从而提高计算效率。

**self.net2 = nn.Sequential()**

**if opt.bottleneck then**

**self.net2:add(cudnn.SpatialConvolution(nChannels, 4\*opt.growthRate, 1, 1, 1, 1, 0, 0))**

**nChannels = 4 \* opt.growthRate**

**self.net2:add(cudnn.SpatialBatchNormalization(nChannels))**

**self.net2:add(cudnn.ReLU(true))**

**end**

**self.net2:add(cudnn.SpatialConvolution(nChannels, opt.growthRate, 3, 3, 1, 1, 1, 1))**

**-- 先前层连续输出**

**self.input\_c = torch.Tensor():type(opt.tensorType)**

**-- save a copy of BatchNorm statistics before forwarding it for the second time when optMemory=4**

**self.saved\_bn\_running\_mean = torch.Tensor():type(opt.tensorType)**

**self.saved\_bn\_running\_var = torch.Tensor():type(opt.tensorType)**

**self.gradInput = {}**

**self.output = {}**

**self.modules = {self.net1, self.net2}**

**end**

**#网络更新**

**function DenseConnectLayerCustom:updateOutput(input)**

**if type(input) ~= 'table' then**

**self.output[1] = input**

**self.output[2] = self.net2:forward(self.net1:forward(input))**

**else**

**for i = 1, #input do**

**self.output[i] = input[i]**

**end**

**torch.cat(self.input\_c, input, 2)**

**self.net1:forward(self.input\_c)**

**self.output[#input+1] = self.net2:forward(self.net1.output)**

**end**

* **densenet.lua densenet网络构建程序**

**#导入DenseConnectLayer**

**require 'models/DenseConnectLayer'**

**#生成训练模型**

**local function createModel(opt)**

**#设置增长率**

**--growth rate**

**local growthRate = opt.growthRate**

**#设置dropout rate，并初始化为0，防止过拟合**

**--dropout rate, set it to 0 to disable dropout, non-zero number to enable dropout and set drop rate**

**local dropRate = opt.dropRate**

**--# 通道进入第一密集区之前，增长率扩大两倍给通道**

**local nChannels = 2 \* growthRate**

**#ResNets.**

**Dense connectivity. （密集连接）**

**Composite function. （复合功能）**

**Pooling layers. （合并图层）**

**Growth rate. (增长率）**

**Bottleneck layers. （瓶颈层）**

**Compression. （压缩）**

**--压缩率在过渡层**

**local reduction = opt.reduction**

**--是否使用瓶颈结构**

**local bottleneck = opt.bottleneck**

**--N: # 每个密集块中密集的连通层**

**local N = (opt.depth - 4)/3**

**if bottleneck then N = N/2 end**

**#叠加密集层**

**local function addDenseBlock(model, nChannels, opt, N)**

**for i = 1, N do**

**addLayer(model, nChannels, opt)**

**nChannels = nChannels + opt.growthRate**

**end**

**return nChannels**

**end**

**-- 建立 DenseNet 模型**

**#功能模块使用1x1的卷积核来实现降低参数量，在pooling之后添加1x1x32的卷积核，通过改变通道大小来进行参数降低。**

**local model = nn.Sequential()**

**#选择数据集训练**

**if opt.dataset == 'cifar10' or opt.dataset == 'cifar100' then**

**-初始卷积层**

**model:add(cudnn.SpatialConvolution(3, nChannels, 3,3, 1,1, 1,1))**

**--密集块为1时及过渡阶段（对应图2结构中的Dense block1)**

**nChannels = addDenseBlock(model, nChannels, opt, N)**

**addTransition(model, nChannels, math.floor(nChannels\*reduction), opt)**

**nChannels = math.floor(nChannels\*reduction)**

**-密集块为2时及过渡阶段（对应图2结构中的Dense block2)**

**nChannels = addDenseBlock(model, nChannels, opt, N)**

**addTransition(model, nChannels, math.floor(nChannels\*reduction), opt)**

**nChannels = math.floor(nChannels\*reduction)**

**--密集块为3时及过渡阶段（对应图2结构中的Dense block3)**

**nChannels = addDenseBlock(model, nChannels, opt, N)**

**addTransition(model, nChannels, nChannels, opt, true, 8)**

**#如果数据集是imagenet**

**elseif opt.dataset == 'imagenet' then**

**--#如果是121层神经网络每层的参数设置为：**

**if opt.depth == 121 then**

**stages = {6, 12, 24, 16}**

**#如果是169层神经网络每层的参数设置为：**

**elseif opt.depth == 169 then**

**stages = {6, 12, 32, 32}**

**#如果是201层神经网络每层的参数设置为：**

**elseif opt.depth == 201 then**

**stages = {6, 12, 48, 32}**

**#如果是161层神经网络每层的参数设置为：**

**elseif opt.depth == 161 then**

**stages = {6, 12, 36, 24}**

**else**

**stages = {opt.d1, opt.d2, opt.d3, opt.d4}**

**end**

**--初始转换遵循ResNet（卷积为224x224）**

**model:add(cudnn.SpatialConvolution(3, nChannels, 7,7, 2,2, 3,3))**

**model:add(cudnn.SpatialBatchNormalization(nChannels))**

**model:add(cudnn.ReLU(true))**

**model:add(nn.SpatialMaxPooling(3, 3, 2, 2, 1, 1))**

**--密集块1和转换（卷积为56x56）**

**nChannels = addDenseBlock(model, nChannels, opt, stages[1])**

**addTransition(model, nChannels, math.floor(nChannels\*reduction), opt)**

**nChannels = math.floor(nChannels\*reduction)**

**--密集块2和转换(卷积为28x28)**

**nChannels = addDenseBlock(model, nChannels, opt, stages[2])**

**addTransition(model, nChannels, math.floor(nChannels\*reduction), opt)**

**nChannels = math.floor(nChannels\*reduction)**

**--密集块3和转换(卷积为14x14)**

**nChannels = addDenseBlock(model, nChannels, opt, stages[3])**

**addTransition(model, nChannels, math.floor(nChannels\*reduction), opt)**

**nChannels = math.floor(nChannels\*reduction)**

**--密集块4和转换(卷积为7x7)**

**nChannels = addDenseBlock(model, nChannels, opt, stages[4])**

**addTransition(model, nChannels, nChannels, opt, true, 7)**

**End**

三、实验分析

图像分类在计算机视觉研究领域中是个热点问题，一直以来备受学者关注，尤其是近些年来，图像的规模呈现爆炸式增长态势，图像分类已成为许多领域的一项关键任务，因此其研究的价值和意义变得越来越重要，各种新的分类技术纷纷被提出。我们在这里利用Densenet模型对Caltech256数据集进行图像分类实验。

**(一）环境搭建**

本次实验基于AWS EC2 p2.xlarge服务器。p2.xlarge为12G显存的Tesla K80显卡, 4核CPU, 60G 内存;

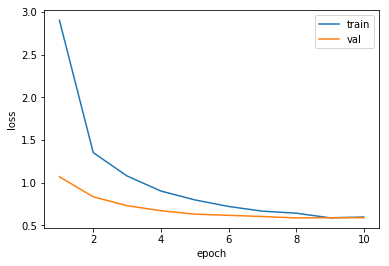
实验平台基于ubuntu 16.04 64位。已搭好MXNet, TensorFlow, Caffe, Caffe2, PyTorch, Theano, CNTK, Keras等主流的深度学习框架。本次实验基于PyTorch p\_36。

**（二）实验结果**

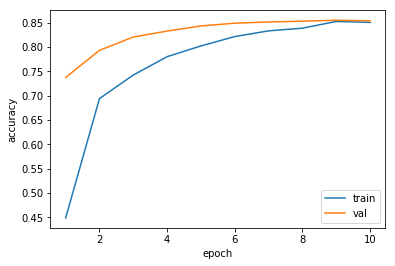
构建卷积神经网络，参数如表1所示loss随迭代趋于稳定，如图8所示，而准确率最终稳定在0.85，如图9所示。在复杂背景图像中，预测分类结果较低，如图9所示；而单一背景下预测分类结果较高。

表1 网络参数

|  |  |  |
| --- | --- | --- |
| **Layers** | **Output Size** | **DenseNet121** |
| **Convolution** | 112\*112 | 7×7 conv, stride 2 |
| **Pooling** | 56\*56 | 3×3 max pool, stride 2 |
| **Dense Block** | 56\*56 |  |
| **Transition Layer** | 56\*56 | 1×1 conv |
| 28\*28 | 2×2 average pool, stride 2 |
| **Dense Block** | 28\*28 |  |
| **Transition Layer** | 28\*28 | 1×1 conv |
| 14\*14 | 2×2 average pool, stride 2 |
| **Dense Block** | 14\*14 |  |
| **Transition Layer** | 14\*14 | 1×1 conv |
| 7\*7 | 2×2 average pool, stride 2 |
| **Dense Block** | 7\*7 |  |
| **Classification Layer** | 1\*1 | 7×7 global average pool |
|  | 1000D fully-connected, softmax |



**图8 loss趋于平缓**



**图9 准确率逐渐稳定**