

目錄

前言	1.1
第一章：经典网络	1.2
ImageNet Classification with Deep Convolutional Neural Network	1.2.1
Very Deep Convolutional Networks for Large-Scale Image Recognition	1.2.2
Going Deeper with Convolutions	1.2.3
Deep Residual Learning for Image Recognition	1.2.4
PolyNet: A Pursuit of Structural Diversity in Very Deep Networks	1.2.5
Squeeze-and-Excitation Networks	1.2.6
Densely Connected Convolutional Networks	1.2.7
SQUEEZENET: ALEXNET-LEVEL ACCURACY WITH 50X FEWER PARAMETERS AND <0.5MB MODEL SIZE	1.2.8
MobileNet v1 and MobileNet v2	1.2.9
Xception: Deep Learning with Depthwise Separable Convolutions	1.2.10
Aggregated Residual Transformations for Deep Neural Networks	1.2.11
ShuffleNet v1 and ShuffleNet v2	1.2.12
CondenseNet: An Efficient DenseNet using Learned Group Convolution	1.2.13
Neural Architecture Search with Reinforcement Learning	1.2.14
Learning Transferable Architectures for Scalable Image Recognition	1.2.15
Progressive Neural Architecture Search	1.2.16
Regularized Evolution for Image Classifier Architecture Search	1.2.17
实例解析：12306验证码破解	1.2.18
第二章：自然语言处理	1.3
Recurrent Neural Network based Language Model	1.3.1
Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation	1.3.2
Neural Machine Translation by Jointly Learning to Align and Translate	1.3.3
Hierarchical Attention Networks for Document Classification	1.3.4
Connectionist Temporal Classification : Labelling Unsegmented Sequence Data with Recurrent Neural Network	1.3.5
About Long Short Term Memory	1.3.6
Attention Is All you Need	1.3.7

BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding	1.3.8
第三章：语音识别	1.4
Speech Recognition with Deep Recurrent Neural Network	1.4.1
第四章：物体检测	1.5
Rich feature hierarchies for accurate object detection and semantic segmentation	
Spatial Pyramid Pooling in Deep Convolutional Networks for Visual Recognition	1.5.1
Fast R-CNN	1.5.2
Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks	1.5.3
R-FCN: Object Detection via Region-based Fully Convolutional Networks	1.5.4
Mask R-CNN	1.5.5
You Only Look Once: Unified, Real-Time Object Detection	1.5.6
SSD: Single Shot MultiBox Detector	1.5.7
YOLO9000: Better, Faster, Stronger	1.5.8
Focal Loss for Dense Object Detection	1.5.9
YOLOv3: An Incremental Improvement	1.5.10
Learning to Segment Every Thing	1.5.11
SNIPER: Efficient Multi-Scale Training	1.5.12
第五章：光学字符识别	1.6
场景文字检测	1.6.1
DeepText: A Unified Framework for Text Proposal Generation and Text Detection in Natural Images	1.6.1.1
Detecting Text in Natural Image with Connectionist Text Proposal Network	
Scene Text Detection via Holistic, Multi-Channel Prediction	1.6.1.2
Arbitrary-Oriented Scene Text Detection via Rotation Proposals	1.6.1.3
文字识别	1.6.2
Spatial Transform Networks	1.6.2.1
Robust Scene Text Recognition with Automatic Rectification	1.6.2.2
端到端文字检测与识别	1.6.3
Reading Text in the Wild with Convolutional Neural Networks	1.6.3.1
Deep TextSpotter: An End-to-End Trainable Scene Text Localization and Recognition Framework	1.6.3.2

实例解析：字符验证码破解	1.6.4
二维信息识别	1.6.5
Show and Tell: A Neural Image Caption Generator	1.6.5.1
第六章：语义分割	1.7
U-Net: Convolutional Networks for Biomedical Image Segmentation	1.7.1
第七章：人脸识别	1.8
人脸检测	1.8.1
DenseBox: Unifying Landmark Localization with End to End Object Detection	
UnitBox: An Advanced Object Detection Network	1.8.1.2 1.8.1.1
第八章：网络优化	1.9
Batch Normalization	1.9.1
Layer Normalization	1.9.2
Weight Normalization	1.9.3
Instance Normalization	1.9.4
Group Normalization	1.9.5
Switchable Normalization	1.9.6
第九章：生成对抗网络	1.10
Generative Adversarial Nets	1.10.1
其它应用	1.11
Holistically-Nested Edge Detection	1.11.1
Image Style Transfer Using Convolutional Neral Networks	1.11.2
Tags	1.12
References	1.13

前言

本书是作者在学习和使用深度学习过程中总结的博客性质的文章。

目前深度学习可以说是计算机界最火的一个应用领域了，在OCR，ASR，NLP上均取得了突破性的进展。密切关注深度学习领域的进展可以说是深度学习工作者必不可少的工作内容之一了。这本书写作的目的在于对深度学习领域中最前沿的技术，源码以及论文的分析总结。本书很少涉及基础知识的介绍，主要内容在于跟踪深度学习中比较流行的，效果比较好的，创新性高的，对应用有质的提升的论文及源码，并结合实际经验进行分析总结。

当然，由于能力和经验都有限，难免会有分析错误的地方，欢迎在评论区给与指出。

本书将会长期更新下去，各种版本的电子书均可以在[gitbook](#)主页下载。如果有想看的论文，可以在评论区留言。如果觉得对您有帮助，欢迎在主页加星，你的星将是我更新下去最大的动力。

更新日志：

- 2018-01-12 : Rich feature hierarchies for accurate object detection and semantic segmentation
- 2018-01-25 : spatial pyramid pooling in deep convolutional networks for visual recognition
- 2018-01-31 : Fast R-CNN
- 2018-02-06 : faster r-cnn towards real-time object detection with region proposal networks
- 2018-02-12 : detecting text in natural image with connectionist text proposal network
- 2018-03-16 : Connectionist Temporal Classification : Labelling Unsegmented Sequence Data with Recurrent Neural Networks
- 2018-03-22 : Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation
- 2018-03-22 : Neural Machine Translation by Jointly Learning to Align and Translate
- 2018-05-07 : About Long Short Term Memory
- 2018-05-09 : Speech Recognition with Deep Recurrent Neural Network
- 2018-05-30 : ImageNet Classification with Deep Convolutional Neural Network
- 2018-05-31 : Very Deep Convolutional Networks for Large-Scale Image Recognition
- 2018-06-05 : Going Deeper with Convolutions
- 2018-06-05 : Deep Residual Learning for Image Recognition
- 2018-06-08 : Densely Connected Convolutional Networks
- 2018-06-15 : R-FCN: Object Detection via Region-based Fully Convolutional Networks
- 2018-07-01 : Mask R-CNN

- 2018-07-04 : 字符验证码破解
- 2018-07-10 : You Only Look Once: Unified, Real-Time Object Detection
- 2018-07-19 : SSD: Single Shot MultiBox Detector
- 2018-07-25 : YOLO9000 : Better, Faster, Stronger
- 2018-07-27 : YOLOv3 : An Incremental Improvement
- 2018-08-01 : Learning to Segment Every Thing
- 2018-08-06 : Reading Text in the Wild with Convolutional Neural Networks
- 2018-08-09 : Spatial Transform Networks
- 2018-08-10 : Deep TextSpotter: An End-to-End Trainable Scene Text Localization and Recognition Framework
- 2018-08-14 : Robust Scene Text Recognition with Automatic Rectification
- 2018-08-20 : DeepText: A Unified Framework for Text Proposal Generation and Text Detection in Natural Images
- 2018-08-28 : Holistically-Nested Edge Detection
- 2018-08-29 : Scene Text Detection via Holistic, Multi-Channel Prediction
- 2018-09-04 : U-Net: Convolutional Networks for Biomedical Image Segmentation
- 2018-09-06 : DenseBox: Unifying Landmark Localization with End to End Object Detection
- 2018-09-06 : UnitBox: An Advanced Object Detection Network
- 2018-09-27 : Arbitrary-Oriented Scene Text Detection via Rotation Proposals
- 2018-10-15 : SNIPER: Efficient Multi-Scale Training
- 2018-10-23 : Squeeze-and-Excitation Networks
- 2018-11-04 : Attention Is All You Need
- 2018-11-05 : BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding
- 2018-11-08 : Focal Loss for Dense Object Detection
- 2018-11-12 : SQUEEZENET: ALEXNET-LEVEL ACCURACY WITH 50X FEWER PARAMETERS AND < 0.5MB MODEL SIZE
- 2018-11-16 : MobileNet v1 and MobileNet v2
- 2018-11-26 : Xception: Deep Learning with Depthwise Separable Convolutions
- 2018-11-28 : Aggregated Residual Transformations for Deep Neural Networks
- 2018-12-04 : ShuffNet v1 and ShuffleNet v2
- 2018-12-10 : CondenseNet: An Efficient DenseNet using Learned Group Convolutions
- 2018-12-11 : PolyNet: A Pursuit of Structural Diversity in Very Deep Networks
- 2018-12-15 : Neural Architecture Search with Reinforcement Learning
- 2018-12-17 : Learning Transferable Architectures for Scalable Image Recognition
- 2018-12-19 : Progressive Neural Architecture Search
- 2018-12-26 : 实例解析 : 12306验证码破解
- 2019-01-06 : Batch Normalization
- 2019-01-10 : Layer Normalization

- 2019-01-17 : Weight Normalization
- 2019-01-28 : Recurrent Neural Network based Language Model
- 2019-01-29 : Image Style Transfer Using Convolutional Neral Networks
- 2019-02-12 : Instance Normalization
- 2019-02-13 : Group Normalization
- 2019-02-19 : Hierarchical Attention Networks for Document Classification
- 2019-02-23 : Regularized Evolution for Image Classifier Architecture Search
- 2019-02-27 : Switchable Normalization

第一章：经典网络

物体分类是深度学习中最经典也是目前研究的最为透彻的一个领域，该领域的开创者也是深度学习的名人堂级别的人物，例如Geoffrey Hinton, Yoshua Bengio等。物体分类常见的数据集由数字数据集MNIST，物体数据集CIFAR-10和类别更多的CIFAR-100，以及任何state-of-the-art的网络实验都规避不了的超大数据集ImageNet。ImageNet是李飞飞教授主办的ILSVRC比赛中使用的数据集，ILSVRC的每年比赛中产生的网络也指引了卷积网络的发展方向。

2012年是ILSVRC的第三届比赛，这次比赛的冠军作品是Hinton团队的AlexNet[123]（链接处图3），他们将2011年的top-5错误率从25.8%降低到16.4%。他们的最大贡献在于验证了卷积操作在大数据集上的有效性，从此物体分类进入了深度学习时代。

2013年的ILSVRC已由深度学习算法霸榜，其冠军网络是ZFNet[122]。ZFNet使用了更深的深度，并且在论文中给出了CNN的有效性的初步解释。

2014年是深度学习领域经典算法最为井喷的一年，在物体检测方向也是如此。这一届比赛的冠军是谷歌团队提出的GoogLeNet[121] (top5 : 7.3%)，亚军则是牛津大学的VGG[120] (top5 : 6.7%)，但是在分类任务中VGG则击败GoogLeNet成为冠军。

VGG（链接处图1）提出了搭建卷积网络的几个思想在现在依旧非常具有指导性，例如按照降采样的分布对网络进行分块，使用小卷积核，每次降采样之后Feature Map的数量加倍等等。另外VGG使用了当初贾扬清提出的Caffe[5]作为深度学习框架并开源了其模型，再凭借其比GoogLeNet更高效的特性，使VGG很快占有了大量的市场，尤其是在物体检测领域。VGG也将卷积网络凭借增加深度来提升精度推上了最高峰。

GoogLeNet（链接处图9）则从特征多样性的角度研究了卷积网络，GoogLeNet的特征多样性是基于一种并行的使用了多个不同尺寸的卷积核的单元来完成的。GoogLeNet的最大贡献在于指出卷积网络精度的增加不仅仅可以依靠深度，增加网络的复杂性也是一种有效的策略。

2015年的冠军网络是何恺明等人提出的残差网络[118]（链接处图7，top5 : 3.57%）。他们指出卷积网络的精度并不会随着深度的增加而增加，导致问题的原因是网络的退化问题。残差网络的核心思想是企图通过向网络中添加直接映射（跳跃连接）的方式解决退化问题。由于残差网络的简单易用的特征使其成为了目前使用的最为广泛的网络结构之一。

2016年ILSVRC的前几名都是模型集成，卷积网络的开创性结构陷入了短暂的停滞。当年的冠军是商汤可以和港中文联合推出的CUIImage，它是6个模型的模型集成，并无创新性，此处不再赘述。

2017年是ILSVRC比赛的最后一届，这一届的冠军由Momenta团队获得，他们提出了基于注意力机制的SENet[116]（链接处图1，top5 : 2.21%），该方法通过自注意力（self-attention）机制为每个Feature Map学习一个权重。

另外一个非常重要的网络是黄高团队于CVPR2017中提出的[DenseNet\[117\]](#)，本质上是各个单元都有连接的密集连接结构（链接处图1）。

除了ILSVRC的比赛中个冠军作品们之外，在提升网络精度中还有一些值得学习的算法。例如[Inception](#)的几个变种[114][113][112]。基于多项式提出的[PolyNet\[111\]](#)，PolyNet采用了更加多样性的特征。

卷积网络的另外一个方向是轻量级的网络，即在不大幅度损失模型精度的前提下，尽可能的压缩模型的大小，提升预测的速度。

轻量级网络的第一个尝试是[SqueezeNet\[110\]](#)，SqueezeNet的策略是使用一部分 1×1 卷积代替 3×3 卷积，它对标的模型是[AlexNet](#)。

轻量级网络最经典的策略是深度可分离卷积的提出，经典算法包括[MobileNetv1\[109\]](#)和[Xception\[119\]](#)。深度可分离卷积由深度卷积和单位卷积组成，深度卷积一般是以通道为单位的 3×3 卷积，在这个过程中不同通道之间没有消息交换。而信息交换则由单位卷积完成，单位卷积就是标准的 1×1 卷积。深度可分离卷积的一个比较新的方法是[MobileNetv2\[115\]](#)，它将深度可分离卷积和残差结构进行了结合，并通过一些列理论分析和实验得出了一种更优的结合方式。

轻量级网络的另外一种策略是在传统卷积和深度可分离卷积中的一个折中方案，是由[ResNeXt\[108\]](#)中提出的，所谓分组卷积是指在深度卷积中以几个通道为一组的普通卷积。[ShuffleNetv1\[107\]](#)提出了通道洗牌策略以加强不同通道之间的信息流通，[ShuffleNetv2\[106\]](#)则是通过分析整个测试时间，提出了对内存访问更高效的[ShuffleNetv2\[106\]](#)。[ShuffleNetv2\[106\]](#)得出的结构是一种和DenseNet非常近似的密集连接结构。黄高团队的[CondenseNet\[102\]](#)则是通过为每个分组学习一个索引层的形式来完成通道直接的信息流通的。

目前在ImageNet上表现最好的是谷歌DeepMind团队提出的NAS系列文章，他们的核心观点是使用强化学习来生成一个完整的网络或是一个网络节点。[NAS\[105\]](#)是该系列的第一篇文章，它使用了强化学习在CIFAR-10上学习到了一个类似于DenseNet的完整的密集连接的网络，如链接处图4。

[NASNet\[106\]](#)解决了NAS不能应用在ImageNet上的问题，它学习的不再是一个完整的网络而是一个网络单元，见链接处图2。这种单元的结构往往比NAS网络要简答得多，因此学习起来效率更高；而且通过堆叠更多NASNet单元的形式可以非常方便的将其迁移到其它任何数据集，包括权威的ImageNet。

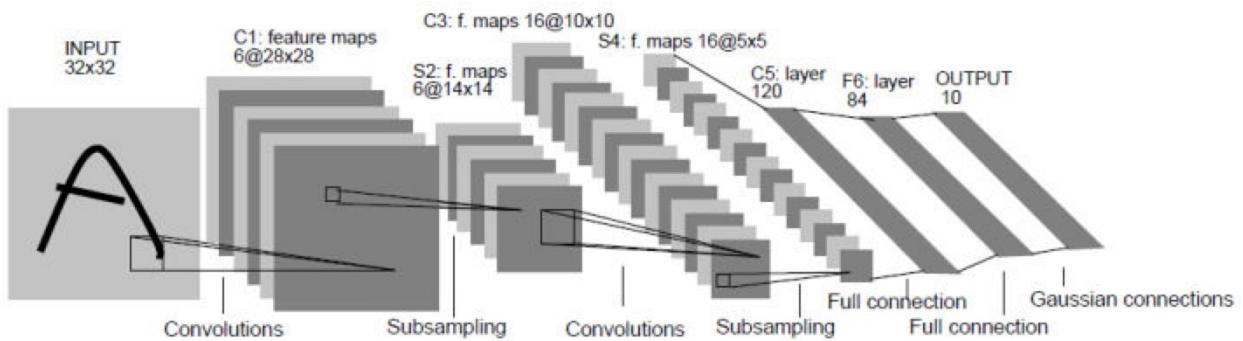
[PNASNet\[103\]](#)则是一个性能更高的强化学习方法，其比NASNet具有更小的搜索空间，而且使用了启发式搜索，策略函数等强化学习领域的办法又花了网络超参的学习过程，其得到的网络也是目前ImageNet数据集上效果最好的网络。网络结构见链接处图2。

ImageNet Classification with Deep Convolutional Neural Network

1. 从LeNet-5开始

使用卷积网络解决图像分类的问题可以往前追溯到1998年LeCun发表的LeNet[101]，解决手写数字识别一文。LeNet又名LeNet-5，是因为在LeNet中，使用的均是 5×5 的卷积核。LeNet的结构如图1。

图1：LeNet-5结构



AlexNet中使用的结构直接影响了其之后沿用至今，卷积+池化+全连接至今仍然是最主流的结构。卷积操作使网络可以响应和卷积核形状类似的特征，而池化则使网络拥有了一定程度的不变性。下面我们简单分析一下LeNet的结构。

INPUT(32×32)： 32×32 的手写数字（共10类）的黑白图片

C1：C1层使用了6个卷积核，每个卷积核的大小均是 5×5 ， $\text{pad}=0$, $\text{stride}=1$ ，激活函数使用的是反正切 \tanh ，所以一次卷积之后，Feature Map的大小是 $(32 - 5 + 1)/1 = 28$ ，该层共有 $28 \times 28 \times 6 = 4704$ 个神经元。加上偏置项，该层共有 $(5 \times 5 + 1) \times 6 = 146$ 个参数

$$f(x) = \tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

S2：S2层是卷积网络常使用的降采样层，在LeNet中，使用的是Max Pooling，降采样的步长是2，降采样核的大小也是2。经过S2层，Feature Map的大小减小一半，变成 14×14 。该层共有 $14 \times 14 \times 6 = 1176$ 个神经元；

C3：C3层是16个大小为 5×5 ，深度为6的卷积核， $\text{pad} = 0$ ， $\text{stride} = 1$ ，激活函数 = \tanh ，一次卷积后，Feature Map的大小是 $(14 - 5 + 1)/1 = 10$ ，神经元个数为 $10 \times 10 \times 16 = 1600$ ，参数个数为 $(5 \times 5 \times 6 + 1) \times 16 = 2416$ 个参数；

S4：步长是2，大小是2的Max Pooling降采样，该层使Feature Map变成 5×5 ，共有 $5 \times 5 \times 16 = 400$ 个神经元。注意池化并不是一种具体的运算，而是代表着一种对统计信息的提取，所以不含有参数。例如在衡量学生考试成绩时，我们不考虑学生试卷的每一道题，而是用总分（平均分）作为评判标准。另外一种常见的池化方式是平均池化（Average Pooling）。

C5：节点数为120的全连接，激活函数是tanh，参数个数是 $(400 + 1) \times 120 = 48120$ ；

F6：节点数为84的全连接，激活函数是tanh，参数个数是 $(120 + 1) \times 84 = 10164$ ；

OUTPUT：10分类的输出层，所以使用的是softmax激活函数，参数个数是 $(84 + 1) \times 10 = 850$ 。softmax用于分类有如下优点：

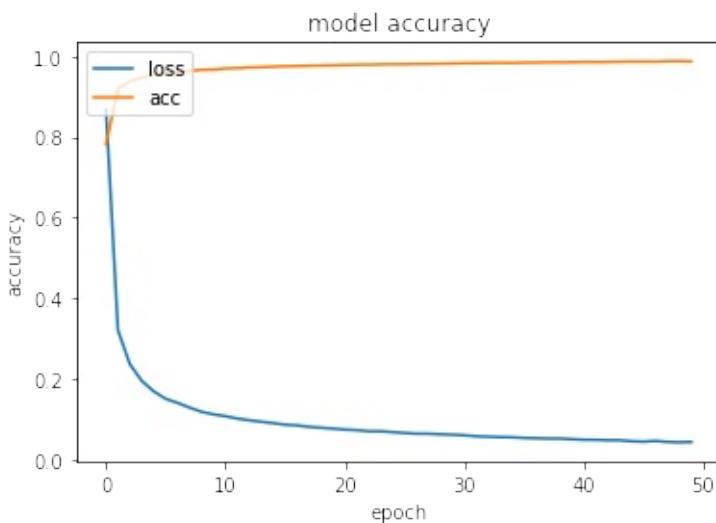
1. e^x 使所有样本的值均大于0，且指数的性质使样本的区分度尽量高；
2. softmax所有可能的和为1，反映出分类为该类别的概率，输出选择概率最高的即可。

使用keras搭建LeNet5的代码如下

```
# 构建LeNet-5网络
model = Sequential()
model.add(Conv2D(input_shape = (28, 28, 1), filters=6, kernel_size=(5,5), padding='valid',
, activation='tanh'))
model.add(MaxPool2D(pool_size=(2,2), strides=2))
model.add(Conv2D(input_shape=(14, 14, 6), filters=16, kernel_size=(5,5), padding='valid',
, activation='tanh'))
model.add(MaxPool2D(pool_size=(2,2), strides=2))
model.add(Flatten())
model.add(Dense(120, activation='tanh'))
model.add(Dense(84, activation='tanh'))
model.add(Dense(10, activation='softmax'))
```

如图2所示，经过10个epoch后，LeNet5就基本已收敛。

图2：LeNet5在MNIST数据集上的收敛曲线图



完整的LeNet5在MNIST上完整训练过程见链接(<https://github.com/senliuy/CNN-Structures/blob/master/LeNet5.ipynb>)。

2. AlexNet

LeNet之后，卷积网络沉寂了14年。直到2012年，AlexNet[123]在ILSVRC2010一举夺魁，直接把ImageNet的精度提升了10个百分点，它将卷积网络的深度和宽度都提升到了新的高度。从此开始，深度学习开始在计算机视觉的各个领域开始披荆斩棘，至今深度学习仍是最热的话题。AlexNet作为教科书式的网络，值得每个学习深度学习的人深入研究。

AlexNet名字取自该论文的第一作者Alex Krizhevsky。在120万张图片的1000类分类任务上的top-1精度是37.5%，top-5则是15.3%，直接比第二的26.2%高出了近10个百分点。AlexNet取得如此成功的原因是其使网络的宽度和深度达到了前所未有高度，而该模型也使网络的可学参数达到了58,322,314个。为了学习该网络，AlexNet并行使用了两块GTX 580，大幅提升了训练速度。这些共同促成了AlexNet的形成。

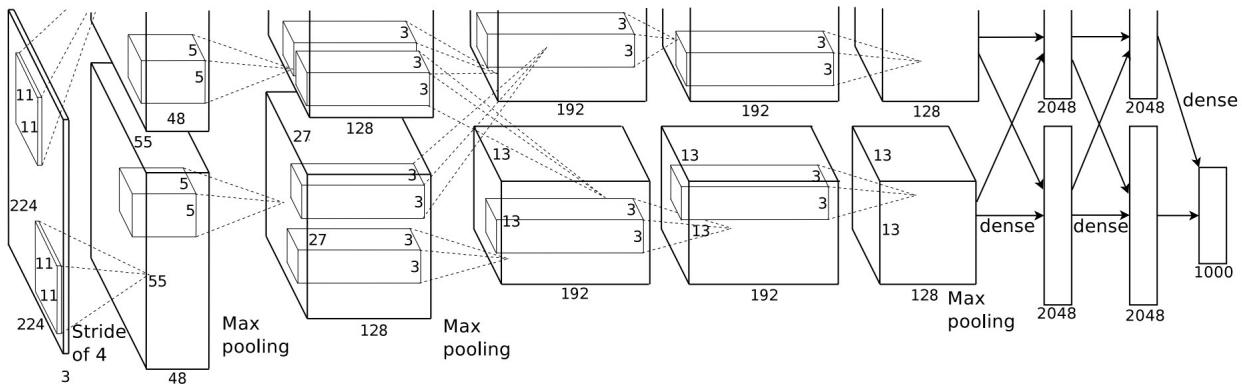
我们知道，当我们想要使用机器学习解决非常复杂的问题时，我们必须使用容量足够大的模型。在深度学习中，增加网络的宽度和网络的深度是提升网络的容量，但是提升容量的同时也会带来两个问题

1. 计算资源的消耗
2. 模型容易过拟合

计算资源是当时限制深度学习发展的最关键的瓶颈，2011年Ciresan[99]等人提出了使用GPU部署CNN的技术框架，由此深度学习有了解决其计算瓶颈的硬件支持。

下面，我们来详细分析一下AlexNet，AlexNet的结构如下图

图3：AlexNet的网络结构



keras实现的AlexNet代码如下

```
# 构建AlexNet-5网络
model = Sequential()
model.add(Conv2D(input_shape = (227,227,3), strides = 4, filters=96, kernel_size=(11,11),
), padding='valid', activation='relu'))
model.add(BatchNormalization())
model.add(MaxPool2D(pool_size=(3,3), strides=2))
model.add(Conv2D(filters=256, kernel_size=(5,5), padding='same', activation='relu'))
model.add(BatchNormalization())
model.add(MaxPool2D(pool_size=(3,3), strides=2))
model.add(Conv2D(filters=384, kernel_size=(3,3), padding='same', activation='relu'))
model.add(BatchNormalization())
model.add(Conv2D(filters=384, kernel_size=(3,3), padding='same', activation='relu'))
model.add(BatchNormalization())
model.add(Conv2D(filters=256, kernel_size=(3,3), padding='same', activation='relu'))
model.add(BatchNormalization())
model.add(MaxPool2D(pool_size=(2,2), strides=2))
model.add(Flatten())
model.add(Dense(4096, activation='tanh'))
model.add(Dropout(0.5))
model.add(Dense(4096, activation='tanh'))
model.add(Dropout(0.5))
model.add(Dense(10, activation='softmax'))
model.summary()
```

根据keras提供的summary()工具，可以得到图4的AlexNet统计图，计算方法参照LeNet5，不再赘述。

图4：根据keras的summary函数得到AlexNet网络参数统计图

Layer (type)	Output Shape	Param #
conv2d_70 (Conv2D)	(None, 55, 55, 96)	34944
batch_normalization_1 (Batch Normalization)	(None, 55, 55, 96)	384
max_pooling2d_42 (MaxPooling)	(None, 27, 27, 96)	0
conv2d_71 (Conv2D)	(None, 27, 27, 256)	614656
batch_normalization_2 (Batch Normalization)	(None, 27, 27, 256)	1024
max_pooling2d_43 (MaxPooling)	(None, 13, 13, 256)	0
conv2d_72 (Conv2D)	(None, 13, 13, 384)	885120
batch_normalization_3 (Batch Normalization)	(None, 13, 13, 384)	1536
conv2d_73 (Conv2D)	(None, 13, 13, 384)	1327488
batch_normalization_4 (Batch Normalization)	(None, 13, 13, 384)	1536
conv2d_74 (Conv2D)	(None, 13, 13, 256)	884992
batch_normalization_5 (Batch Normalization)	(None, 13, 13, 256)	1024
max_pooling2d_44 (MaxPooling)	(None, 6, 6, 256)	0
flatten_17 (Flatten)	(None, 9216)	0
dense_47 (Dense)	(None, 4096)	37752832
dropout_3 (Dropout)	(None, 4096)	0
dense_48 (Dense)	(None, 4096)	16781312
dropout_4 (Dropout)	(None, 4096)	0
dense_49 (Dense)	(None, 10)	40970
<hr/>		
Total params: 58,327,818		
Trainable params: 58,325,066		
Non-trainable params: 2,752		

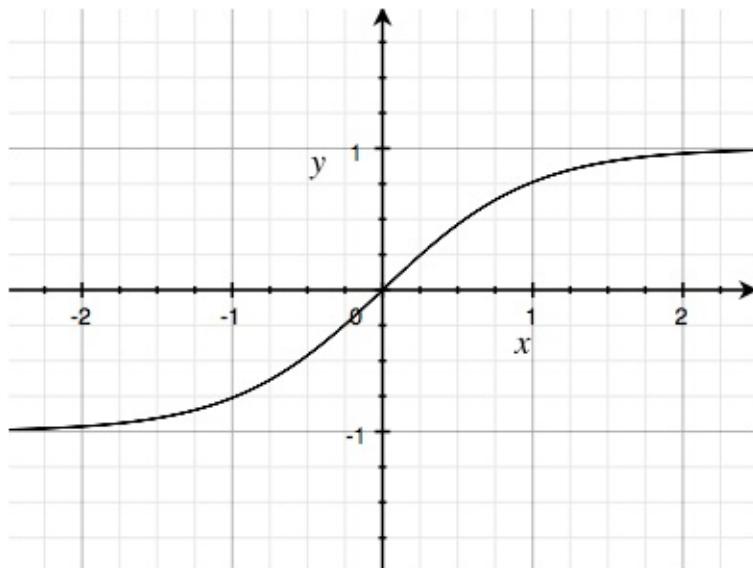
2.1 多GPU训练

首先对比图1和图3，我们发现AlexNet将网络分成了两个部分，这幅图这么画的原因是为了提升训练速度，作者使用了两块GPU(叫做GPU1和GPU2)并行训练模型，例如第二个卷积每个GPU只使用自身显存中的feature map，而第三个卷积是需要使用另外一个GPU显存中的feature map。不过得益于TensorFlow等开源框架对多机多卡的支持和显卡显存的提升，我们已经不太关心网络的底层实现了，所以这一部分就不再赘述。

2.2 整流线性单元ReLU

在LeNet5中，论文使用了tanh作为激活函数，反正切的函数曲线如图5

图5：反正切（tanh）函数曲线



在BP的反向过程中，局部梯度会与整个损失函数关于该单元输出的梯度相乘。因为，当 $\tanh(x)$ 中的 x 的绝对值比较大的时候，此时该单元的梯度便非常接近于0，在深度学习中，激活函数这个现象叫做“饱和”。同样 $sigmoid$ 激活函数也存在饱和的现象。

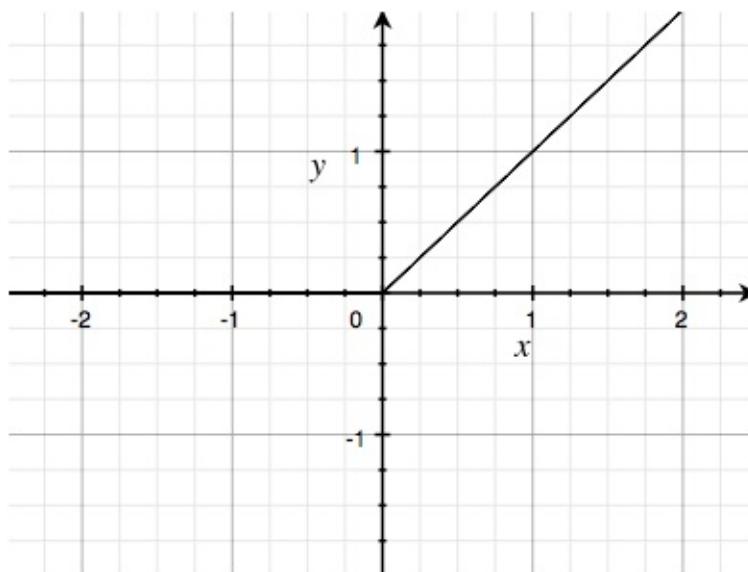
“饱和”带来的一个深度学习中一个非常致命的问题，那便是梯度消失。梯度消失是由于是由BP中链式法则的乘法特性导致的，反应在深度学习的训练过程中便是越接近损失函数的参数，梯度越大，成为了主要学习的函数，而远离损失函数的参数的梯度则非常接近0，导致没有梯度传到这一部分参数，从而使得这一部分的参数很难学习到。

为了解决这个问题，AlexNet引入了ReLU激活函数

$$f(x) = \max(0, x)$$

ReLU的函数曲线如下图

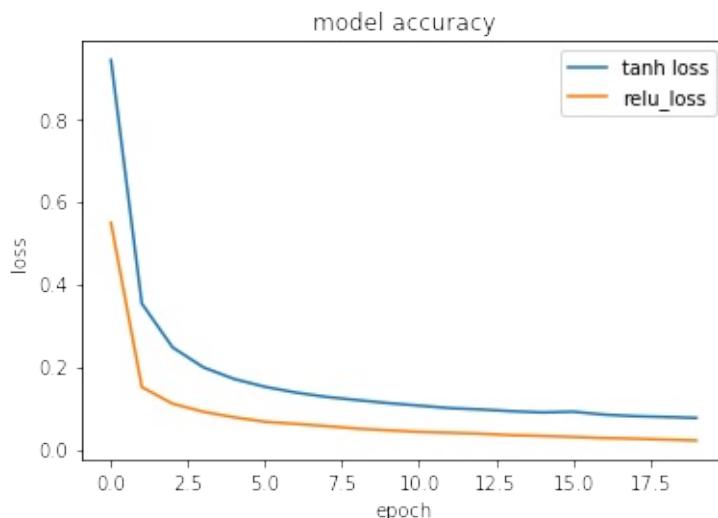
图6：整流线性单元（relu）函数曲线



在ReLU中，无论 x 的取值是多大， $f(x)$ 导数都是1，也就不存在导数小于1导致的梯度消失的现象发生了。图7是我在MNIST数据集，根据LeNet5使用不同的激活函数中得到的不同模型的收敛速度。

此外，由于ReLU将小于0的部分全部置0，所以ReLU的另外一个特点就是稀疏性，不仅可以优化网络的性能，还一定程度上缓解了过拟合的问题。

图7：LeNet5使用不同激活函数的收敛速度对比



虽然使用ReLU的节点不会有饱和问题，但是会“死掉”——大部分甚至所有的值为负值，从而导致该层的梯度都为0。死神经元是由进入网络的负值引起的（例如在大规模的梯度更新之后可能发生），减少学习率能缓解该现象。

2.3 LRN

局部响应归一化，模拟的是动物神经中的横向抑制效应，是一个已近被淘汰的算法，有VGG [120]的论文中已经指出，LRN并没有什么效果¹。在现在的网络中，LRN已经被其它归一化方法所替代，例如我在上面代码中给出的Batch Normalization。LRN是使用同一位置临近的

feature map来归一化当前feature map的值的一种方法，其表达式为

$$b_{x,y}^i = \frac{a_{x,y}^i}{(k + \alpha \sum_{j=\max(0, i-n/2)}^{\min(N-1, i+n/2)} (a_{x,y}^j)^2)^\beta}$$

其中 N 表示 feature map 的数量， $n = 5, k = 2, \alpha = 0.5, \beta = 0.75$ ，这些值均由验证集得出。

另外，AlexNet 把 LRN 放在了池化层之前，这在计算上是非常不经济的，一种更好的尝试是把 LRN 放在池化之后。

2.4 Overlap pooling

当进行 pooling 的时候，如果步长 stride 小于 pooling 核的尺寸，相邻之间的 pooling 核会有相互覆盖的地方，这种方式便叫做 overlap pooling。论文中指出这种方式可以减轻过拟合，至今未想通原因。

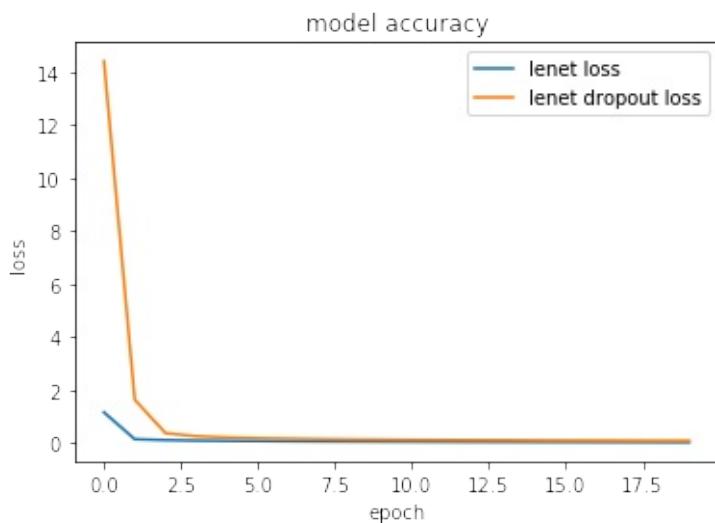
2.5 Dropout

在 AlexNet 的前两层，作者使用了 Dropout[100] 来减轻容量高的模型容易发生过拟合的现象。Dropout 的使用方法是在训练过程中随机将一定比例的隐层节点置 0，Dropout 能够缓解过拟合的原因是每次训练都会采样出一个不同的网络架构，但是这些结构是共享权值的。这种技术减轻了节点之间的耦合性，因为一个节点不能依赖网络的其它节点。因此，节点能够学习到更健壮的特征，因为只有这样，节点才能适应每次采样得到的不同的网络结构。在测试时，我们是不对节点进行 Drop 的。

显然 Dropout 会减慢收敛速度，但其对减轻过拟合的优异表现仍旧使其在当前的网络中得到广泛的使用。

下图是 LeNet-5 中加入 Dropout 之后模型的训练 loss 曲线图，从图中我们可以看出，加入 Dropout 之后，训练速度放缓了一些，20 个 epoch 之后，训练集的损失函数也高于没有 Dropout 的，但是加入 Dropout 之后，虽然 loss=0.0735 远高于没有 Dropout 的 0.0155，但是测试集的准确率从 0.9826 上升到 0.9841，可见 Dropout 对于缓解过拟合还是非常有帮助的。实验代码见：https://github.com/senliuy/CNN-Structures/blob/master/LeNet_Dropout.ipynb

图 8 : Dropout vs None Dropout



¹. Such normalisation does not improve the performance on the ILSVRC dataset, but leads to increased memory consumption and computation time. Where applicable, the parameters for the LRN layer are those of (Krizhevsky et al., 2012). ↵

Very Deep Convolutional NetWorks for Large-Scale Image Recognition

1. 前言

时间来到2014年，随着AlexNet在ImageNet数据集上的大放异彩，探寻针对ImageNet的数据集的最优网络成为了提升该数据集精度的一个最先想到的思路。牛津大学计算机视觉组（Visual Geometry Group）和Google的Deep Mind的这篇论文便是对卷积网络的深度和其性能的探索，由此该网络也被命名为VGG[120]。

VGG的结构非常清晰：

- 按照 2×2 的Pooling层，网络可以分成若干段；
- 每段之内由若干个same卷积操作构成，段之内的Feature Map数量固定不变；
- Feature Map按段以2倍的速度逐渐递增，第四段和第五段都是512（64-128-256-512-512）。

VGG的结构非常容易扩展到其它数据集。在VGG中，段数每增加1，Feature Map的尺寸减少一半，所以通过减少段的数目将网络应用到例如MNIST，CIFAR等图像尺寸更小的数据集。段内的卷积的数量是可变的，因为卷积的个数并不会影响图片的尺寸，我们可以根据任务的复杂度自行调整段内的卷积数量。

VGG的表现效果也非常好，在ILSVRC2014分类中排名第二（第一是GoogLeNet[121]，没有办法），定位比赛排名第一。

VGG将其模型开源在其官方网站上，为其它任务提供了非常好的迁移学习的材料，使得VGG马上占有了大量商业市场。关于不同框架的VGG模型，自行在网上搜索。

2. VGG介绍

VGG关于网络结构的探索可以总结为图1，图1包含了大量信息，我们一一分析之

图1：VGG家族

ConvNet Configuration					
A	A-LRN	B	C	D	E
11 weight layers	11 weight layers	13 weight layers	16 weight layers	16 weight layers	19 weight layers
input (224 × 224 RGB image)					
conv3-64	conv3-64 LRN	conv3-64 conv3-64	conv3-64 conv3-64	conv3-64 conv3-64	conv3-64 conv3-64
maxpool					
conv3-128	conv3-128	conv3-128 conv3-128	conv3-128 conv3-128	conv3-128 conv3-128	conv3-128 conv3-128
maxpool					
conv3-256 conv3-256	conv3-256 conv3-256	conv3-256 conv3-256	conv3-256 conv3-256 conv1-256	conv3-256 conv3-256 conv3-256	conv3-256 conv3-256 conv3-256 conv3-256
maxpool					
conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512 conv1-512	conv3-512 conv3-512 conv3-512	conv3-512 conv3-512 conv3-512 conv3-512
maxpool					
conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512 conv1-512	conv3-512 conv3-512 conv3-512	conv3-512 conv3-512 conv3-512 conv3-512
maxpool					
FC-4096					
FC-4096					
FC-1000					
soft-max					

图2反应了VGG家族的各个模型的性能

图2：VGG家族的性能表现

ConvNet config. (Table 1)	smallest image side		top-1 val. error (%)	top-5 val. error (%)
	train (S)	test (Q)		
A	256	256	29.6	10.4
A-LRN	256	256	29.7	10.5
B	256	256	28.7	9.9
C	256	256	28.1	9.4
	384	384	28.1	9.3
	[256;512]	384	27.3	8.8
D	256	256	27.0	8.8
	384	384	26.8	8.7
	[256;512]	384	25.6	8.1
E	256	256	27.3	9.0
	384	384	26.9	8.7
	[256;512]	384	25.5	8.0

关于VGG家族的keras实现和参数统计见附件A。

2.1 家族的特征

我们来看看VGG家族的共同特征

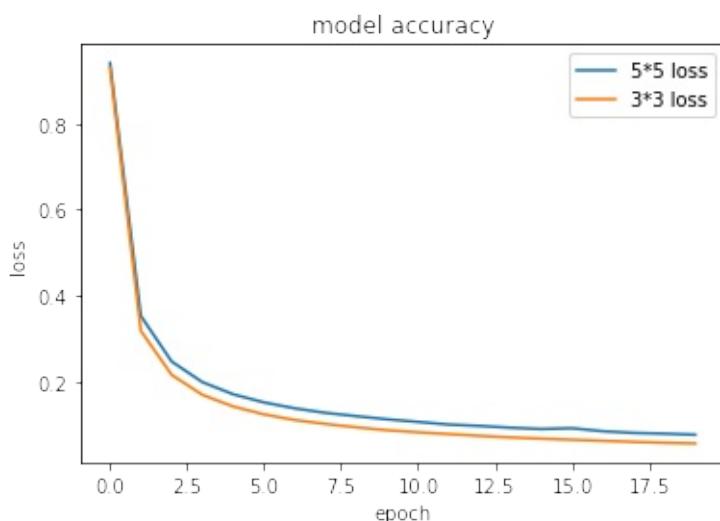
- 输入图像的尺寸均是 224×224 ；
- 均为5层Max Pooling，表示最终均会产生大小为 7×7 的Feature Map，这是一个大小比较合适的尺寸；
- 卷积部分之后（特征层）跟的是两个隐层节点数目为4096的全连接，最后接一个1000类softmax分类器。
- 所有VGG模型均可以表示为： $m \times (n \times (conv_{33}) + max_pooling)$

VGG在卷积核方向的最大改进是将卷积核全部换成更小的 3×3 或者 1×1 的卷积核，而性能最好的VGG-16和VGG-19由且仅由 3×3 卷积构成。原因有如下两点：

1. 根据感受野的计算公式 $rfsize = (out - 1) \times stride + ksize$ ，我们知道一个 7×7 的卷积核和3层 3×3 的卷积核具有相同感受野，但是由于3层感受野具有更深的深度，由此可以构建更具判别性的决策函数；
2. 假设Feature Map的数量都是C，3层 3×3 卷积核的参数个数是 $3 \times (3 \times 3 + 1) \times C^2 = 30C^2$ ，1层 7×7 卷积核的参数个数是 $1 \times (7 \times 7 + 1) \times C^2 = 50C^2$ ，3层 3×3 卷积核具有更少的参数。
3. 但由于神经元数量和层数的增多，训练速度会变得更慢

下图是把LeNet-5的 5×5 卷积换成了两层 3×3 卷积在MNIST上的收敛表现，实验表明两层 3×3 的网络确实比单层 5×5 的网络表现好，但是训练速度也慢了一倍。

图3： 3×3 LeNet vs 5×5 LeNet



另外，作者在前两层的全连接出使用 $\text{drop rate} = 0.5$ 的Dropout，然而并没有在图1中反应出来。

2.2 VGG-A vs VGG-A-LRN

VGG A-LRN 比 VGG A多了一个AlexNet介绍的LRN层，但是实验数据表明加入加入LRN的VGG-A错误率反而更高了。而且LRN的加入会更加占用内存消耗以及增加训练时间。

2.3 VGG-A vs VGG-B vs VGG-D vs VGG-E

对比VGG-A(11层), VGG-B(13层), VGG-D(16层), VGG-E(19层)的错误率，我们发现随着网络深度的增加，分类的错误率逐渐降低，当然越深的深度则表示需要的训练时间越长。但是当模型的深度到达一定深度时（VGG-D和VGG-E），网络的错误率趋近饱和，甚至偶尔会发生深层网络的错误率高于浅层网络的情况，同时考虑网络的训练时间，我们就要折中考虑选择合适的网络深度了。我相信作者一定探索了比VGG-E更深的网络，但是由于表现的不理想并没有列在论文中。后面介绍的残差网络则通过shortcut的机制将网络的深度理论上扩展到了无限大。

2.4 VGG-B vs VGG-C

VGG-C在VGG-B的基础上添加了3个 1×1 的卷积层， 1×1 的卷积是在NIN[98]中率先使用的，由于 1×1 卷积在不影响感受野的前提下提升了决策函数的非线性性，由此带来了错误率的下降。

2.5 VGG-C vs VGG-D

VGG-D将VGG-C中的 1×1 卷积换成了 3×3 卷积，该组对比表明 3×3 卷积的提升效果要大于 1×1 卷积

2.6 VGG-D vs VGG-E

当网络层数增加到16层时，网络的精度趋近于饱和。当网络提升到19层时，虽然精度有了些许的提升，但需要的训练时间也大幅增加。

3. VGG的训练和测试

3.1 训练

VGG的训练分为单尺度训练（single-scale training）和多尺度训练（multi-scale training）。在单尺度训练中，原图的短边被固定为一个固定值S（实验中S被固定为了256和384），然后等比例缩放图片。再从缩放的图片中裁剪 224×224 的子图用于训练模型。在多尺度训练中，每张图的短边随机为256到512之间的一个随机值，然后再从缩放的图片中裁剪 224×224 的子图。

3.2 测试

测试时可以使用和训练相同的图片裁剪方法，然后通过若干不同裁剪的图片的投票的方式选择最后的分类。

但测试的时候图片是单张输入的，使用裁剪的方式可能会漏掉图片的重要信息，在OverFeat [97]的论文中，提出了将整幅图做为输入的方式，过程如下：

1. 将测试图片的短边固定为Q，Q可以不等于S；
2. 将Q输入VGG，在conv5层，得到 $W \times H \times 512$ 的特征向量，W和H一般不等于7；
3. 将第一层全连接层看成 $7 \times 7 \times 512 \times 4096$ 的卷积层（原本需要先进行flatten操作，再进行FC操作），对比附件中的vgg-e和使用全卷积的vgg-e-test，可以发现两者具有相同的参数数量。
4. 将第二、三全连接层看成 $1 \times 1 \times 4096 \times 4096$ 与 $1 \times 1 \times 4096 \times numClasses$ 的卷积层
5. 如果输入图片大小为 224×224 ，则输出为 $1 \times 1 \times numClasses$ ，因为图片大小可以不一致，可以看作某张图片多个切片¹的预测结果。最终经过sum-pool，每个通道求和，得到 $1 \times 1 \times numClasses$ 的结果。作为最终输出，即取所有平均数作为最终输出。

附件A

VGG模型的keras代码和参数统计：<https://github.com/senliuy/CNN-Structures/blob/master/VGG.ipynb>

Going Deeper With Convolutions

前言

2012年之后，卷积网络的研究分成了两大流派，并且两个流派都在2014年有重要的研究成果发表。第一个流派是增加卷积网络的深度和宽度，经典的网络有ImageNet 2013年冠军ZF-net[122]以及我们在上篇文章中介绍的VGG系列[120]。另外一个流派是增加卷积核的拟合能力，或者说是增加网络的复杂度，典型的网络有可以拟合任意凸函数的Maxout Networks[96]，可以拟合任意函数的Network in Network (NIN)[98]，以及本文要解析的基于Inception的GoogLeNet[121]。为了能更透彻的了解GoogLeNet的思想，我们首先需要了解Maxout和NIN两种结构。

1. 背景知识

1.1 Maxout Networks

在之前介绍的AlexNet中，引入了Dropout [100]来减轻模型的过拟合的问题。Dropout可以看做是一种集成模型的思想，在每个step中，会将网络的隐层节点以概率 p 置0。Dropout和传统的bagging方法主要有以下两个方面不同：

1. Dropout的每个子模型的权值是共享的；
2. 在训练的每个step中，Dropout每次会使用不同的样本子集训练不同的子网络。

这样在每个step中都会有不同的节点参与训练，减轻了节点之间的耦合性。在测试时，使用的是整个网络的所有节点，只是节点的输出值要乘以Dropout的概率 p 。

作者认为，与其像Dropout这种毫无选择的平均，我们不如有条件的选择节点来生成网络。在传统的神经网络中，第*i*层的隐层的计算方式如下（暂时不考虑激活函数）：

$$h_i = Wx_i + b_i$$

假设第*i-1*层和第*i*层的节点数分别是*d*和*m*，那么*W*则是一个*d* \times *m*的二维矩阵。而在Maxout网络中，*W*是一个三维矩阵，矩阵的维度是*d* \times *m* \times *k*，其中*k*表示Maxout网络的通道数，是Maxout网络唯一的参数。Maxout网络的数学表达式为：

$$h_i = \max_{j \in [1, k]} z_{i,j}$$

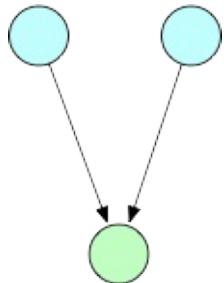
其中 $z_{i,j} = x^T W_{\dots i, j} + b_{i,j}$ 。

下面我们通过一个简单的例子来说明Maxout网络的工作方式。对于一个传统的网络，假设第 i 层有两个节点，第 $i+1$ 层有1个节点，那么MLP的计算公式就是：

$$out = g(W * X + b)$$

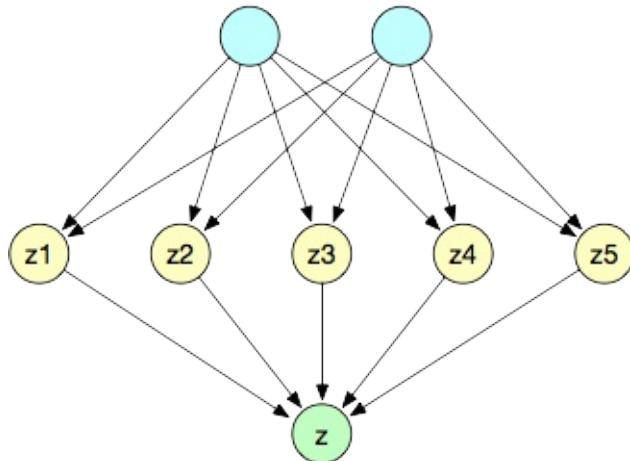
其中 $g(\cdot)$ 是激活函数，如 $tanh$ ， $relu$ 等，如图1。

图1：传统神经网络



如果我们将Maxout的参数 k 设置为5，Maxout网络可以展开成图2的形式：

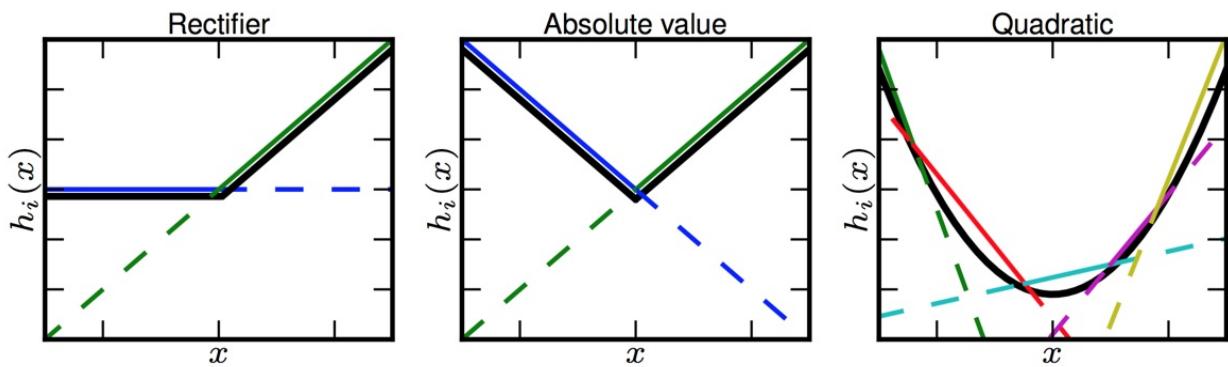
图2：Maxout网络



其中 $z = \max(z_1, z_2, z_3, z_4, z_5)$ 。

其中 z_1-z_5 为线性函数，所以 z 可以看做是分段线性的激活函数。Maxout Network的论文中给出了证明，当 k 足够大时，Maxout单元可以以任意小的精度逼近任何凸函数，如图3。

图3：Maxout的凸函数无线逼近性



在keras2.0之前的版本中，我们可以找到Maxout网络的实现，其核心代码只有一行。

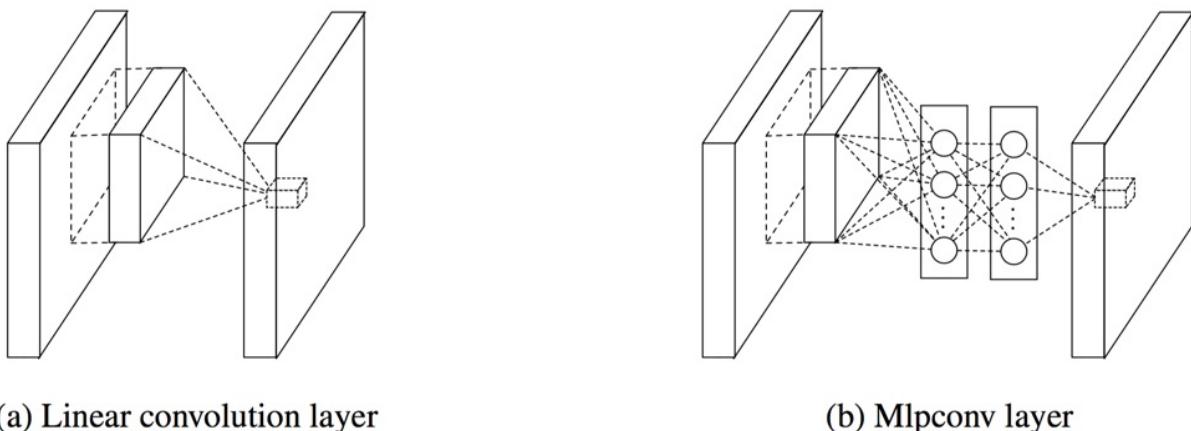
```
output = K.max(K.dot(X, self.W) + self.b, axis=1)
```

Maxout网络存在的最大的一个问题就是，网络的参数是传统网络的K倍，K倍的参数数量并没有带来其等价的精度提升，现基本已被工业界淘汰。

1.2 Network in Network

Maxout节点可以逼近任何凸函数，而NIN的节点理论上可以逼近任何函数。在NIN中，作者也是采用整图滑窗的形式，只是将卷积网络的卷积核替换成了一个小型的MLP网络，如图4所示：

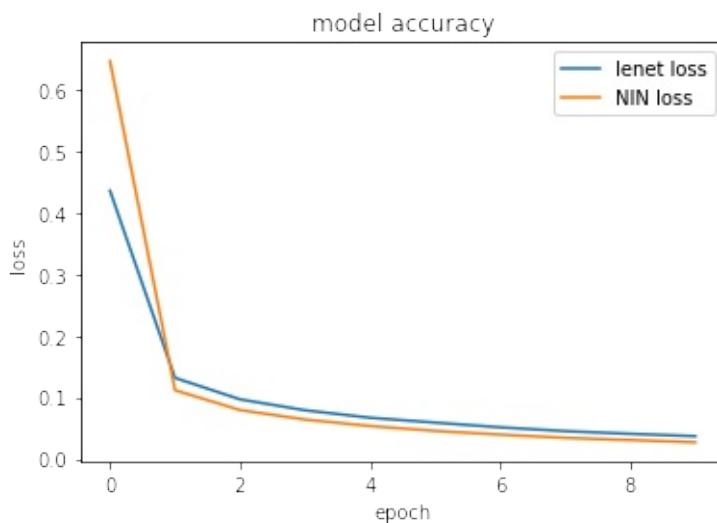
图4：Network in Network网络结构



在卷积操作中，一次卷积操作仅相当于卷积核和滑窗的一次矩阵乘法，其拟合能力有限。而MLP替代卷积操作则增加了每次滑窗的拟合能力，下图是将LeNet5改造成NIN在MNIST上的训练过程收敛曲线，通过实验结果我们可以得到三个重要信息：

1. NIN的参数数量远大于同类型的卷积网络；
2. NIN的收敛速度快于经典网络；
3. NIN的训练速度慢于经典网络。

图5：NIN vs LeNet5



```

NIN = Sequential()
NIN.add(Conv2D(input_shape=(28, 28, 1), filters= 8, kernel_size = (5,5),padding = 'same'
,activation = 'relu'))
NIN.add(Conv2D(input_shape=(28, 28, 1), filters= 8, kernel_size = (1,1),padding = 'same'
,activation = 'relu'))
NIN.add(Flatten())
NIN.add(Dense(196,activation = 'relu'))
NIN.add(Reshape((14,14,1),input_shape = (196,1)))
NIN.add(Conv2D(16,(5,5),padding = 'same',activation = 'relu'))
NIN.add(Conv2D(16,(1,1),padding = 'same',activation = 'relu'))
NIN.add(Flatten())
NIN.add(Dense(120,activation = 'relu'))
NIN.add(Dense(84,activation = 'relu'))
NIN.add(Dense(10))
NIN.add(Activation('softmax'))
NIN.summary()

```

实验内容见链接：<https://github.com/senliuy/CNN-Structures/blob/master/NIN.ipynb>

处对比全连接，NIN中的 1×1 卷积操作保存了网络隐层节点和输入图像的位置关系，NIN的思想反而在物体定位和语义分割上得到了更广泛的应用。除了保存Feature Map的图像位置关系， $x = y$ 卷积还有两个用途：

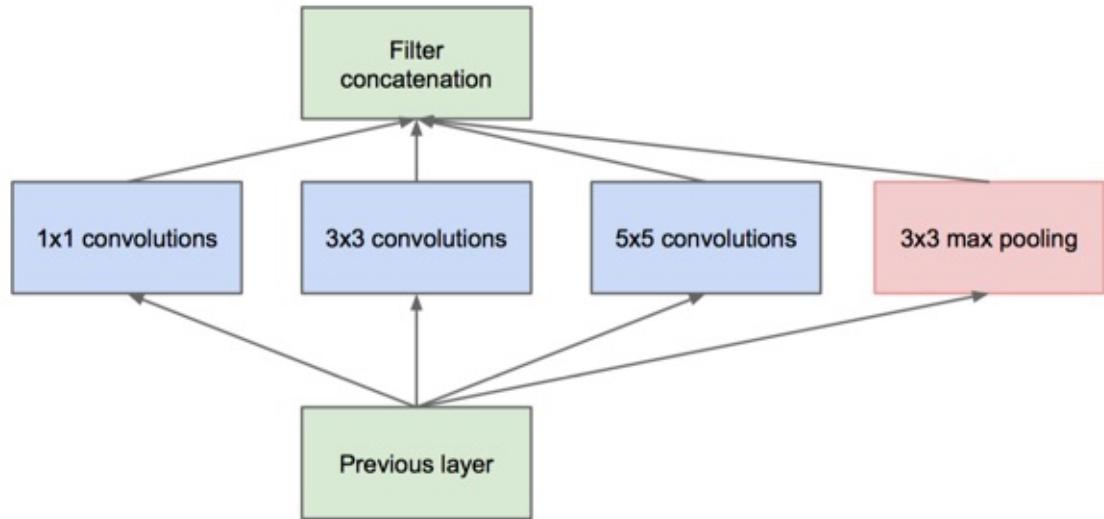
1. 实现Feature Map的升维和降维；
2. 实现跨Feature Map的交互。

另外，NIN提出了使用Global Average Pooling来减轻全连接层的过拟合问题，即在卷积的最后一层，直接将每个Feature Map求均值，然后再接softmax。

1.3 Inception v1

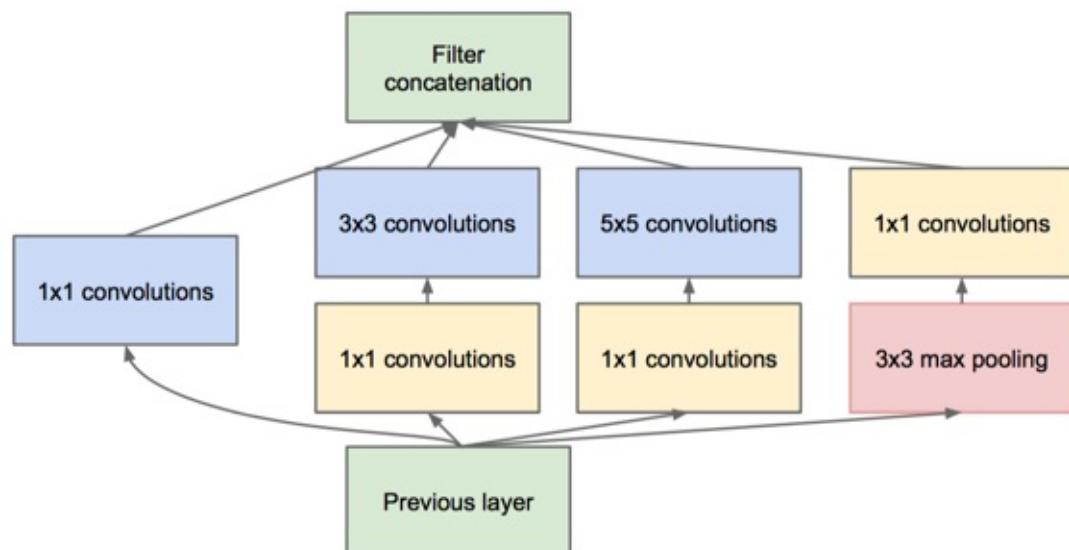
GoogLeNet的核心部件叫做Inception。根据感受野的递推公式，不同大小的卷积核对应着不同大小的感受野。例如在VGG的最后一层， 1×1 , 3×3 和 5×5 卷积的感受野分别是196, 228, 260。我们根据感受野的计算公式也可以知道，网络的层数越深，不同大小的卷积对应在原图的感受野的大小差距越大，这也就是为什么Inception通常在越深的层次中效果越明显。在每个Inception模块中，作者并行使用了 1×1 , 3×3 和 5×5 三个不同大小的卷积核。同时，考虑到池化一直在卷积网络中扮演着积极的作用，所以作者建议Inception中也要加入一个并行的max pooling。至此，一个naive版本的Inception便诞生了，见图6

图6：Naive Inception



但是这个naive版本的Inception会使网络的Feature Map的数量乘以4，随着Inception数量的增多，Feature Map的数量会呈指数形式的增长，这对应着大量计算资源的消耗。为了提升运算速度，Inception使用了NIN中介绍的 1×1 卷积在卷积操作之前进行降采样，由此便诞生了Inception v1，见图7。

图7：Inception结构

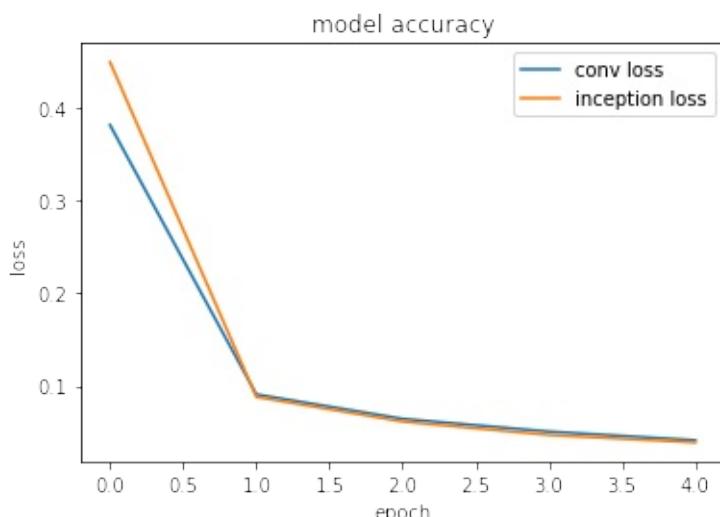


Inception的代码也比较容易实现，建立4个并行的分支并在最后merge到一起即可。为了运行MNIST数据集，我使用了更窄的网络（Feature Map数量均为4），论文中feature map的数量已注释在代码中。

```
def inception(x):
    inception_1x1 = Conv2D(4,(1,1), padding='same', activation='relu')(x) #64
    inception_3x3_reduce = Conv2D(4,(1,1), padding='same', activation='relu')(x) #96
    inception_3x3 = Conv2D(4,(3,3), padding='same', activation='relu')(inception_3x3_reduce) #128
    inception_5x5_reduce = Conv2D(4,(1,1), padding='same', activation='relu')(x) #16
    inception_5x5 = Conv2D(4,(5,5), padding='same', activation='relu')(inception_5x5_reduce) #32
    inception_pool = MaxPool2D(pool_size=(3,3), strides=(1,1), padding='same')(x) #192
    inception_pool_proj = Conv2D(4,(1,1), padding='same', activation='relu')(inception_pool) #32
    inception_output = merge([inception_1x1, inception_3x3, inception_5x5, inception_pool_proj],
                           mode='concat', concat_axis=3)
    return inception_output
```

图8是使用相同Feature Map总量的Inception替代卷积网络的一层卷积的在MNIST数据集上的收敛速度对比，从实验结果可以看出，对于比较小的数据集，Inception的提升非常有限。对比两个网络的容量，我们发现Inception和相同Feature Map的3*3卷积拥有相同数量的参数个数，实验内容见链接：<https://github.com/senliuy/CNN-Structures/blob/master/Inception.ipynb>。

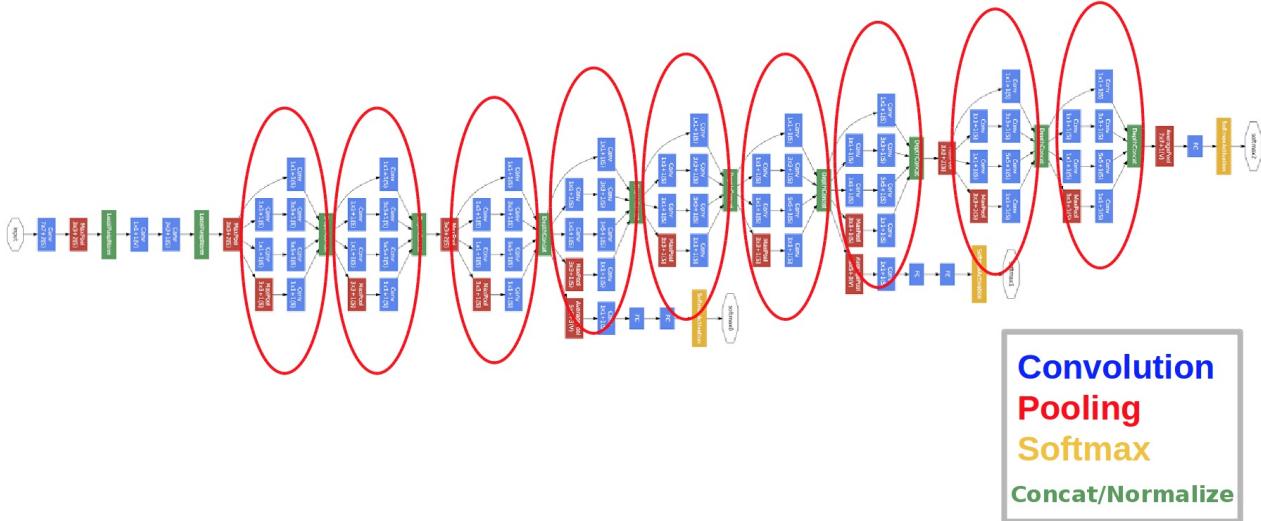
图8：Inception vs 卷积网络



1.4 GoogLeNet

GoogLeNet取名是为了致敬第一代深度卷积网络LeNet5，作者通过堆叠Inception的形式构造了一个由9个Inception模块共22层的网络，并一举拿下了ILSVRC2014图像分类任务的冠军。GoogLeNet的网络结构如图9 (高清图片参考论文)。

图9：GoogLeNet



对比其他网络，GoogLeNet的一个最大的不同是在中间多了两个softmax分支作为辅助损失函数（auxiliary loss）。在训练时，这两个分类器的损失会以0.3的比例添加到损失函数上。根据论文的解释，该分支有两个作用：

1. 保证较低层提取的特征也有分类物体的能力；
2. 具有提供正则化并克服梯度消失问题的能力；

需要注意的是，在测试的时候，这两个softmax分支会被移除。

辅助损失函数的提出，是遵循信息论中的数据处理不等式（Data Processing Inequality, DPI），所谓数据处理不等式，是指数据处理的步骤越多，则丢失的信息也会越多，表达如下

$$X \rightarrow Y \rightarrow Z$$

$$I(X;Z) \leq I(X;Y)$$

上式也就是说，在数据传输的过程中，信息有可能消失，但绝对不会凭空增加。反应到BP中，也就是在计算梯度的时候，梯度包含的损失信息会逐层减少，所以GoogLeNet网络的中间层添加了两组损失函数以防止损失的过度丢失。

1.5 Inception V2

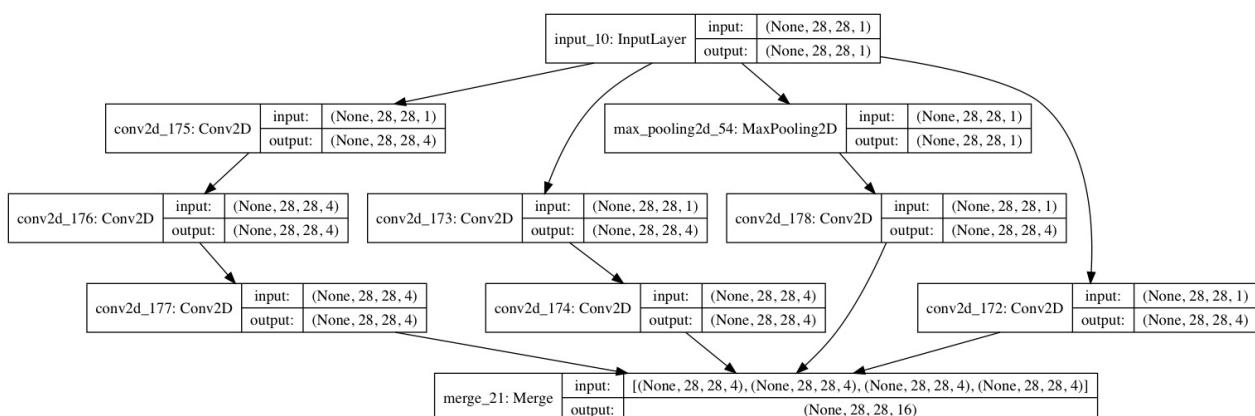
在VGG中，我们讲解过，一个5*5的卷积核与两个3*3的卷积核拥有相同大小的感受野，但是两个3*3的卷积核拥有更强的拟合能力，所以在Inception V2 [114]的版本中，作者将5*5的卷积替换为两个3*3的卷积。其实本文的最大贡献是Batch normalization的提出，关于BN，我们会另开一个版块单独讲解。

```

def inception_v2(x):
    inception_1x1 = Conv2D(4,(1,1), padding='same', activation='relu')(x)
    inception_3x3_reduce = Conv2D(4,(1,1), padding='same', activation='relu')(x)
    inception_3x3 = Conv2D(4,(3,3), padding='same', activation='relu')(inception_3x3_reduce)
    inception_5x5_reduce = Conv2D(4,(1,1), padding='same', activation='relu')(x)
    inception_5x5_1 = Conv2D(4,(3,3), padding='same', activation='relu')(inception_5x5_reduce)
    inception_5x5_2 = Conv2D(4,(3,3), padding='same', activation='relu')(inception_5x5_1)
    inception_pool = MaxPool2D(pool_size=(3,3), strides=(1,1), padding='same')(x)
    inception_pool_proj = Conv2D(4,(1,1), padding='same', activation='relu')(inception_pool)
    inception_output = merge([inception_1x1, inception_3x3, inception_5x5_2, inception_pool_proj],
                            mode='concat', concat_axis=3)
    return inception_output

```

图10 : Inception v2



1.6 Inception V3

Inception V3[113]是将Inception V1和V2中的 $n \times n$ 卷积换成一个 $n \times 1$ 和一个 $1 \times n$ 的卷积，这样做带来的好处有以下几点：

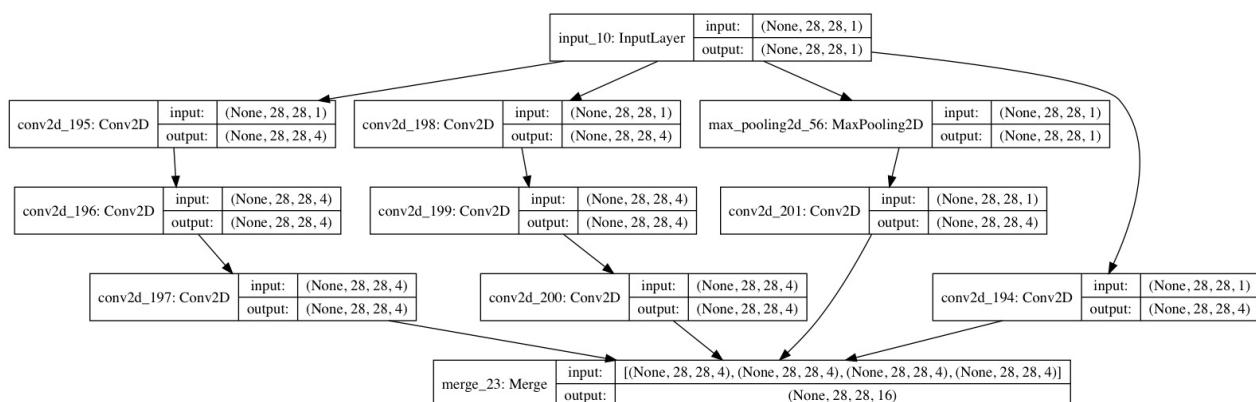
1. 节约了大量参数，提升了训练速度，减轻了过拟合的问题；
2. 多层卷积增加了模型的拟合能力；
3. 非对称卷积核的使用增加了特征的多样性。

```

def inception_v3(x):
    inception_1x1 = Conv2D(4,(1,1), padding='same', activation='relu')(x)
    inception_3x3_reduce = Conv2D(4,(1,1), padding='same', activation='relu')(x)
    inception_3x1 = Conv2D(4,(3,1), padding='same', activation='relu')(inception_3x3_reduce)
    inception_1x3 = Conv2D(4,(1,3), padding='same', activation='relu')(inception_3x1)
    inception_5x5_reduce = Conv2D(4,(1,1), padding='same', activation='relu')(x)
    inception_5x1 = Conv2D(4,(5,1), padding='same', activation='relu')(inception_5x5_reduce)
    inception_1x5 = Conv2D(4,(1,5), padding='same', activation='relu')(inception_5x1)
    inception_pool = MaxPool2D(pool_size=(3,3), strides=(1,1), padding='same')(x)
    inception_pool_proj = Conv2D(4,(1,1), padding='same', activation='relu')(inception_pool)
    inception_output = merge([inception_1x1, inception_1x3, inception_1x5, inception_pool_proj],
                             mode='concat', concat_axis=3)
    return inception_output

```

图11 : Inception v3



1.7 Inception V4

Inception V4 [112]将残差网络[118]融合到了Inception模型中，即相当于在Inception中加入了
一条输入到输出的short cut。

Deep Residual Learning for Image Recognition

前言

在VGG中，卷积网络达到了19层，在GoogLeNet中，网络史无前例的达到了22层。那么，网络的精度会随着网络的层数增多而增多吗？在深度学习中，网络层数增多一般会伴着下面几个问题

1. 计算资源的消耗
2. 模型容易过拟合
3. 梯度消失/梯度爆炸问题的产生

问题1可以通过GPU集群来解决，对于一个企业资源并不是很大的问题；问题2的过拟合通过采集海量数据，并配合Dropout正则化等方法也可以有效避免；问题3通过Batch Normalization也可以避免。貌似我们只要无脑的增加网络的层数，我们就能从此获益，但实验数据给了我们当头一棒。

作者发现，随着网络层数的增加，网络发生了退化（degradation）的现象：随着网络层数的增多，训练集loss逐渐下降，然后趋于饱和，当你再增加网络深度的话，训练集loss反而会增大。注意这并不是过拟合，因为在过拟合中训练loss是一直减小的。

当网络退化时，浅层网络能够达到比深层网络更好的训练效果，这时如果我们把低层的特征传到高层，那么效果应该至少不比浅层的网络效果差，或者说如果一个VGG-100网络在第98层使用的是和VGG-16第14层一模一样的特征，那么VGG-100的效果应该会和VGG-16的效果相同。所以，我们可以在VGG-100的98层和14层之间添加一条直接映射（Identity Mapping）来达到此效果。

从信息论的角度讲，由于DPI（数据处理不等式）的存在，在前向传输的过程中，随着层数的加深，Feature Map包含的图像信息会逐层减少，而ResNet的直接映射的加入，保证了 $l+1$ 层的网络一定比 l 层包含更多的图像信息。

基于这种使用直接映射来连接网络不同层直接的思想，残差网络[118],[95]应运而生。

1. 残差网络

1.1 残差块

残差网络是由一系列残差块组成的（图1）。一个残差块可以用表示为：

$$x_{l+1} = x_l + \mathcal{F}(x_l, W_l)$$

残差块分成两部分直接映射部分和残差部分。 $h(x_l)$ 是直接映射，反应在图1中是左边的曲线； $\mathcal{F}(x_l, W_l)$ 是残差部分，一般由两个或者三个卷积操作构成，即图1中右侧包含卷积的部分。

图1：残差块

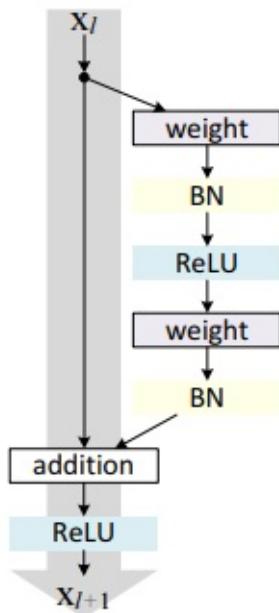


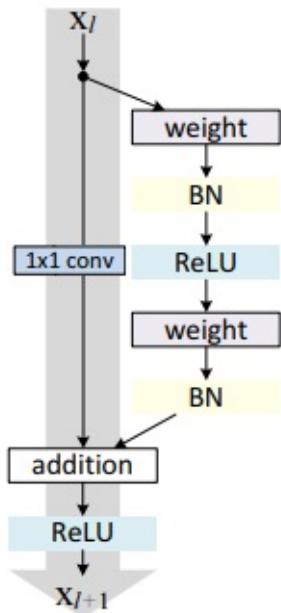
图1中的'Weight'在卷积网络中是指卷积操作，'addition'是指单位加操作。

在卷积网络中， x_l 可能和 x_{l+1} 的Feature Map的数量不一样，这时候就需要使用 1×1 卷积进行升维或者降维（图2）。这时，残差块表示为：

$$x_{l+1} = h(x_l) + \mathcal{F}(x_l, W_l)$$

其中 $h(x_l) = W'_l x$ 。其中 W'_l 1×1 卷核，是实验结果 1×1 卷积对模型性能提升有限，所以一般是在升维或者降维时才会使用。

图2：1*1残差块



一般，这种版本的残差块叫做resnet_v1，keras代码实现如下：

```

def res_block_v1(x, input_filter, output_filter):
    res_x = Conv2D(kernel_size=(3,3), filters=output_filter, strides=1, padding='same')(x)
    res_x = BatchNormalization()(res_x)
    res_x = Activation('relu')(res_x)
    res_x = Conv2D(kernel_size=(3,3), filters=output_filter, strides=1, padding='same')(res_x)
    res_x = BatchNormalization()(res_x)
    if input_filter == output_filter:
        identity = x
    else: #需要升维或者降维
        identity = Conv2D(kernel_size=(1,1), filters=output_filter, strides=1, padding='same')(x)
    x = keras.layers.add([identity, res_x])
    output = Activation('relu')(x)
    return output
  
```

1.2 残差网络

残差网络的搭建分为两步：

1. 使用VGG公式搭建Plain VGG网络
2. 在Plain VGG的卷积网络之间插入Identity Mapping，注意需要升维或者降维的时候加入1*1卷积。

在实现过程中，一般是直接stack残差块的方式。

```

def resnet_v1(x):
    x = Conv2D(kernel_size=(3,3), filters=16, strides=1, padding='same', activation='relu')(x)
    x = res_block_v1(x, 16, 16)
    x = res_block_v1(x, 16, 32)
    x = Flatten()(x)
    outputs = Dense(10, activation='softmax', kernel_initializer='he_normal')(x)
    return outputs

```

1.3 为什么叫残差网络

在统计学中，残差和误差是非常容易混淆的两个概念。误差是衡量观测值和真实值之间的差距，残差是指预测值和观测值之间的差距。对于残差网络的命名原因，作者给出的解释是，网络的一层通常可以看做 $y = H(x)$ ，而残差网络的一个残差块可以表示为 $H(x) = F(x) + x$ ，也就是 $F(x) = H(x) - x$ ，在单位映射中， $y = x$ 便是观测值，而 $H(x)$ 是预测值，所以 $F(x)$ 便对应着残差，因此叫做残差网络。

2. 残差网络的背后原理

残差块一个更通用的表示方式是

$$y_l = h(x_l) + \mathcal{F}(x_l, W_l)$$

$$x_{l+1} = f(y_l)$$

现在我们先不考虑升维或者降维的情况，那么在[1]中， $h(\cdot)$ 是直接映射， $f(\cdot)$ 是激活函数，一般使用ReLU。我们首先给出两个假设：

- 假设1： $h(\cdot)$ 是直接映射；
- 假设2： $f(\cdot)$ 是直接映射。

那么这时候残差块可以表示为：

$$x_{l+1} = x_l + \mathcal{F}(x_l, W_l)$$

对于一个更深的层 L ，其与 l 层的关系可以表示为

$$x_L = x_l + \sum_{i=1}^{L-1} \mathcal{F}(x_i, W_i)$$

这个公式反应了残差网络的两个属性：

1. L 层可以表示为任意一个比它浅的 l 层和他们之间的残差部分之和；

2. $x_L = x_0 + \sum_{i=0}^{L-1} \mathcal{F}(x_i, W_i)$ ， L 是各个残差块特征的单位累和，而MLP是特征矩阵的累积。

根据BP中使用的导数的链式法则，损失函数 ε 关于 x_l 的梯度可以表示为

$$\frac{\partial \varepsilon}{\partial x_l} = \frac{\partial \varepsilon}{\partial x_L} \frac{\partial x_L}{\partial x_l} = \frac{\partial \varepsilon}{\partial x_L} \left(1 + \frac{\partial}{\partial x_l} \sum_{i=1}^{L-1} \mathcal{F}(x_i, W_i)\right) = \frac{\partial \varepsilon}{\partial x_L} + \frac{\partial \varepsilon}{\partial x_L} \frac{\partial}{\partial x_l} \sum_{i=1}^{L-1} \mathcal{F}(x_i, W_i)$$

上面公式反映了残差网络的两个属性：

- 在整个训练过程中， $\frac{\partial}{\partial x_l} \sum_{i=1}^{L-1} \mathcal{F}(x_i, W_i)$ 不可能一直为-1，也就是说在残差网络中不会出现梯度消失的问题。
- $\frac{\partial \varepsilon}{\partial x_L}$ 表示 L 层的梯度可以直接传递到任何一个比它浅的 l 层。

通过分析残差网络的正向和反向两个过程，我们发现，当残差块满足上面两个假设时，信息可以非常畅通的在高层和低层之间相互传导，说明这两个假设是让残差网络可以训练深度模型的充分条件。那么这两个假设是必要条件吗？

2.1 直接映射是最好的选择

对于假设1，我们采用反证法，假设 $h(x_l) = \lambda_l x_l$ ，那么这时候，残差块（图3.b）表示为

$$x_{l+1} = \lambda_l x_l + \mathcal{F}(x_l, W_l)$$

对于更深的 L 层

$$x_L = \left(\prod_{i=l}^{L-1} \lambda_i \right) x_l + \sum_{i=l}^{L-1} \left(\prod_{i=l}^{i-1} \lambda_i \right) \mathcal{F}(x_i, W_i)$$

为了简化问题，我们只考虑公式的左半部分 $x'_L = \left(\prod_{i=l}^{L-1} \lambda_i \right) x_l$ ，损失函数 ε 对 x_l 求偏微分得

$$\frac{\partial \varepsilon}{\partial x_l} = \frac{\partial \varepsilon}{\partial x'_L} \left(\prod_{i=l}^{L-1} \lambda_i \right)$$

上面公式反映了两个属性：

- 当 $\lambda > 1$ 时，很有可能发生梯度爆炸；
- 当 $\lambda < 1$ 时，梯度变成0，会阻碍残差网络信息的反向传递，从而影响残差网络的训练。

所以 λ 必须等1。同理，其他常见的激活函数都会产生和上面的例子类似的阻碍信息反向传播的问题。

对于其它不影响梯度的 $h(\cdot)$ ，例如LSTM中的门机制（图3.c，图3.d）或者Dropout（图3.f）以及[1]中用于降维的 1×1 卷积（图3.e）也许会有效果，作者采用了实验的方法进行验证，实验结果见图4

图3：直接映射的变异模型

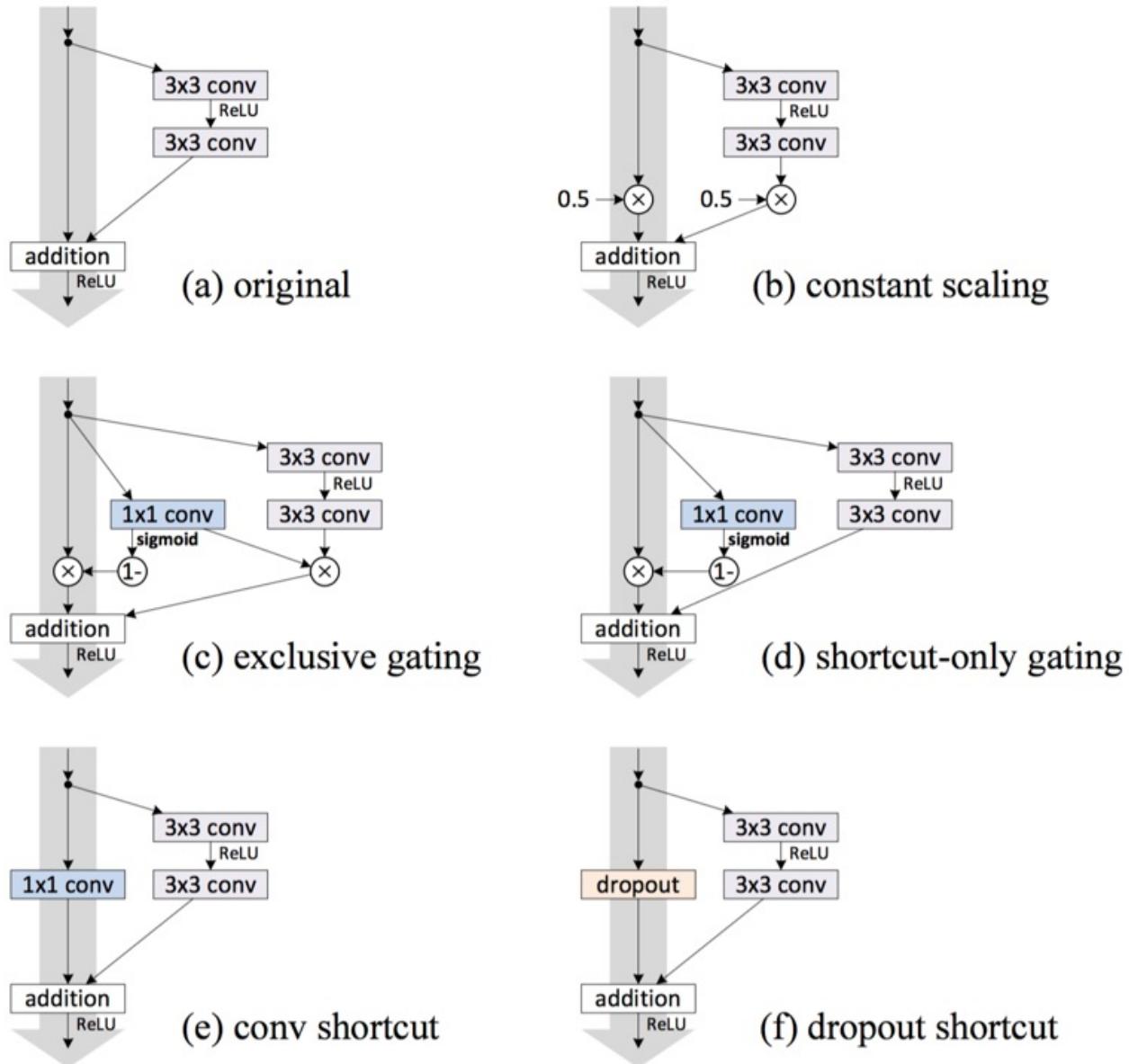


图4：变异模型（均为110层）在Cifar10数据集上的表现

case	Fig.	on shortcut	on \mathcal{F}	error (%)	remark
original [1]	Fig. 2(a)	1	1	6.61	
constant scaling	Fig. 2(b)	0	1	fail	This is a plain net
		0.5	1	fail	
		0.5	0.5	12.35	frozen gating
exclusive gating	Fig. 2(c)	$1 - g(\mathbf{x})$	$g(\mathbf{x})$	fail	init $b_g=0$ to -5
		$1 - g(\mathbf{x})$	$g(\mathbf{x})$	8.70	init $b_g=-6$
		$1 - g(\mathbf{x})$	$g(\mathbf{x})$	9.81	init $b_g=-7$
shortcut-only gating	Fig. 2(d)	$1 - g(\mathbf{x})$	1	12.86	init $b_g=0$
		$1 - g(\mathbf{x})$	1	6.91	init $b_g=-6$
1×1 conv shortcut	Fig. 2(e)	1×1 conv	1	12.22	
dropout shortcut	Fig. 2(f)	dropout 0.5	1	fail	

从图4的实验结果中我们可以看出，在所有的变异模型中，依旧是直接映射的效果最好。下面我们将对图3中的各种变异模型的分析

1. **Exclusive Gating**：在LSTM的门机制中，绝大多数门的值为0或者1，几乎很难落到0.5附近。当 $g(x) \rightarrow 0$ 时，残差块变成只有直接映射组成，阻碍卷积部分特征的传播；当 $g(x) \rightarrow 1$ 时，直接映射失效，退化为普通的卷积网络；
2. **Short-cut only gating**： $g(x) \rightarrow 0$ 时，此时网络便是[1]提出的直接映射的残差网络； $g(x) \rightarrow 1$ 时，退化为普通卷积网络；
3. **Dropout**：类似于将直接映射乘以 $1 - p$ ，所以会影响梯度的反向传播；
4. **1×1 conv**： 1×1 卷积比直接映射拥有更强的表示能力，但是实验效果却不如直接映射，说明该问题更可能是优化问题而非模型容量问题。

所以我们可以得出结论：假设1成立，即

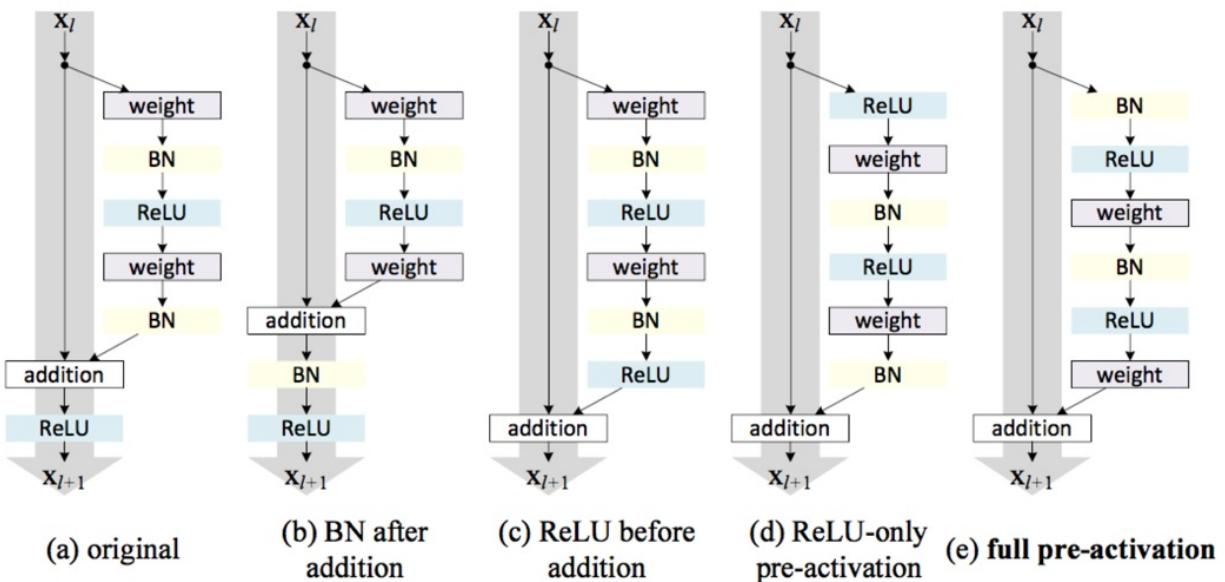
$$y_l = x_l + \mathcal{F}(x_l, w_l)$$

$$y_{l+1} = x_{l+1} + \mathcal{F}(x_{l+1}, w_{l+1}) = f(y_l) + \mathcal{F}(f(y_l), w_{l+1})$$

2.2 激活函数的位置

[1] 提出的残差块可以详细展开如图5.a，即在卷积之后使用了BN做归一化，然后在和直接映射单位加之后使用了ReLU作为激活函数。

图5：激活函数在残差网络中的使用



在2.1节中，我们得出假设“直接映射是最好的选择”，所以我们希望构造一种结构能够满足直接映射，即定义一个新的残差结构 $\hat{f}(\cdot)$ ：

$$y_{l+1} = y_l + \mathcal{F}(\hat{f}(y_l), w_{l+1})$$

上面公式反应到网络里即将激活函数移到残差部分使用，即图5.c，这种在卷积之后使用激活函数的方法叫做post-activation。然后，作者通过调整ReLU和BN的使用位置得到了几个变种，即5.d中的ReLU-only pre-activation和5.d中的full pre-activation。作者通过对照试验对比了这几种变异模型，结果见图6。

图6：基于激活函数位置的变异模型在Cifar10上的实验结果

case	Fig.	ResNet-110	ResNet-164
original Residual Unit [1]	Fig. 4(a)	6.61	5.93
BN after addition	Fig. 4(b)	8.17	6.50
ReLU before addition	Fig. 4(c)	7.84	6.14
ReLU-only pre-activation	Fig. 4(d)	6.71	5.91
full pre-activation	Fig. 4(e)	6.37	5.46

而实验结果也表明将激活函数移动到残差部分可以提高模型的精度。

该网络一般就在resnet_v2，keras实现如下：

```

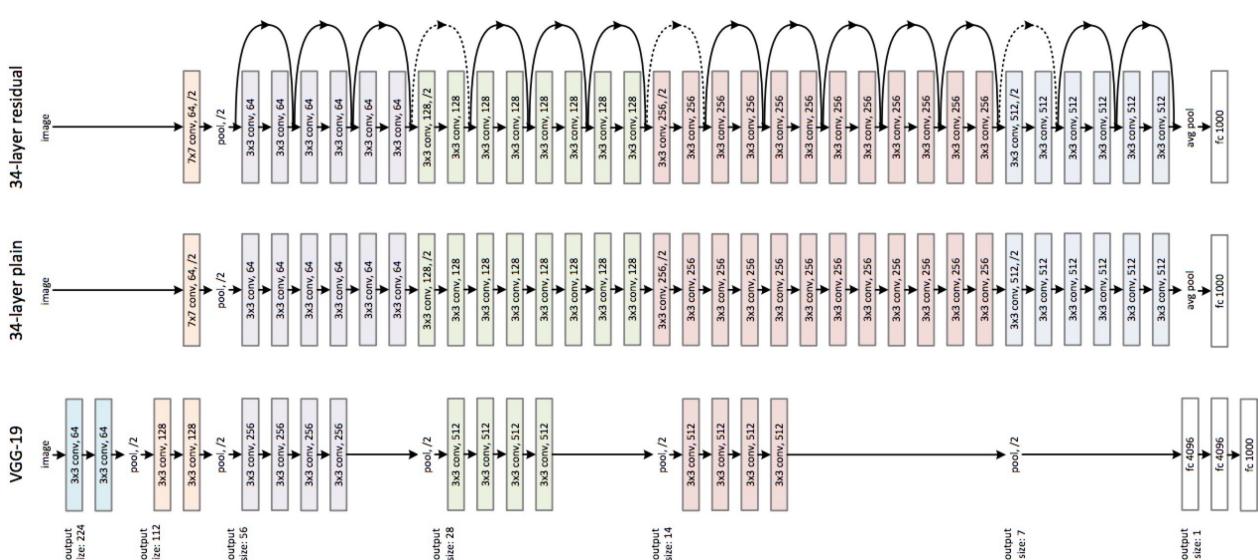
def res_block_v2(x, input_filter, output_filter):
    res_x = BatchNormalization()(x)
    res_x = Activation('relu')(res_x)
    res_x = Conv2D(kernel_size=(3,3), filters=output_filter, strides=1, padding='same')(res_x)
    res_x = BatchNormalization()(res_x)
    res_x = Activation('relu')(res_x)
    res_x = Conv2D(kernel_size=(3,3), filters=output_filter, strides=1, padding='same')(res_x)
    if input_filter == output_filter:
        identity = x
    else: #需要升维或者降维
        identity = Conv2D(kernel_size=(1,1), filters=output_filter, strides=1, padding='same')(x)
    output= keras.layers.add([identity, res_x])
    return output

def resnet_v2(x):
    x = Conv2D(kernel_size=(3,3), filters=16, strides=1, padding='same', activation='relu')(x)
    x = res_block_v2(x, 16, 16)
    x = res_block_v2(x, 16, 32)
    x = BatchNormalization()(x)
    y = Flatten()(x)
    outputs = Dense(10, activation='softmax', kernel_initializer='he_normal')(y)
    return outputs

```

一个残差网络的搭建也是采用堆叠残差块的形式，在是否降维的时候选择不同的残差块。残差网络v1给出的示意图如图7所示。

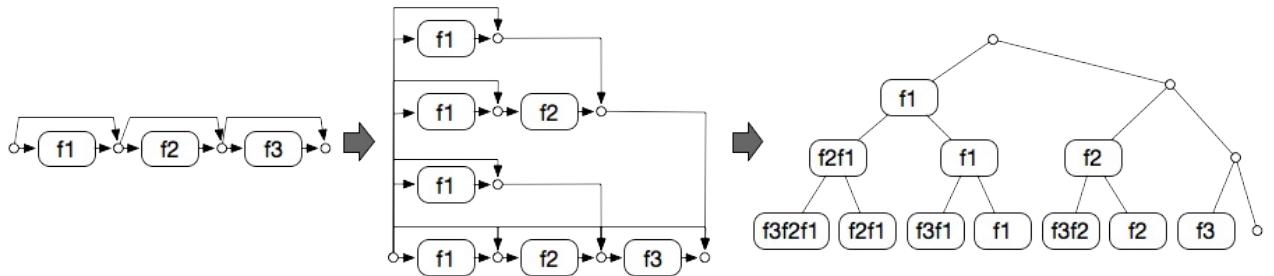
图7：残差网络展开成二叉树



3. 残差网络与模型集成

Andreas Veit等人的论文[94]指出残差网络可以从模型集成的角度理解。如图7所示，对于一个3层的残差网络可以展开成一棵含有8个节点的二叉树，而最终的输出便是这8个节点的集成。而他们的实验也验证了这一点，随机删除残差网络的一些节点网络的性能变化较为平滑，而对于VGG等stack到一起的网络来说，随机删除一些节点后，网络的输出将完全随机。

图8：残差网络展开成二叉树



PolyNet: A Pursuit of Structural Diversity in Very Deep Networks

tags: PolyNet, Inception, ResNet

前言

在 [Inception v4\[112\]](#) 中，[Inception\[121\]](#) 和 [残差网络\[118\]](#) 首次得以共同使用，后面简称 IR。这篇文章提出的 PolyNet 可以看做是 IR 的进一步扩展，它从多项式的角度推出了更加复杂且效果更好的混合模型，并通过实验得出了这些复杂模型的最优混合形式，命名为 *Very Deep PolyNet*[\[111\]](#)。

本文试图从结构多样性上说明 PolyNet 的提出动机，但还是没有摆脱通过堆积模型结构 (Inception, ResNet) 来得到更好效果的牢笼，模型创新性上有所欠缺。其主要贡献是虽然增加网络的深度和宽度能提升性能，但是其收益会很快变少，这时候如果从结构多样性的角度出发优化模型，带来的效益也许会由于增加深度带来的效益，为我们优化网络结构提供了一个新的方向。

1. PolyNet 详解

1.1 结构多样性

当前增大网络表达能力的一个最常见的策略是通过增加网络深度，但是如图1所示，随着网络的深度增加，网络的收益变得越来越小。另一个模型优化的策略是增加网络的宽度，例如增加 Feature Map 的数量。但是增加网络的宽度是非常不经济的，因为每增加 k 个参数，其计算复杂度和显存占用都要增加 k^2 。

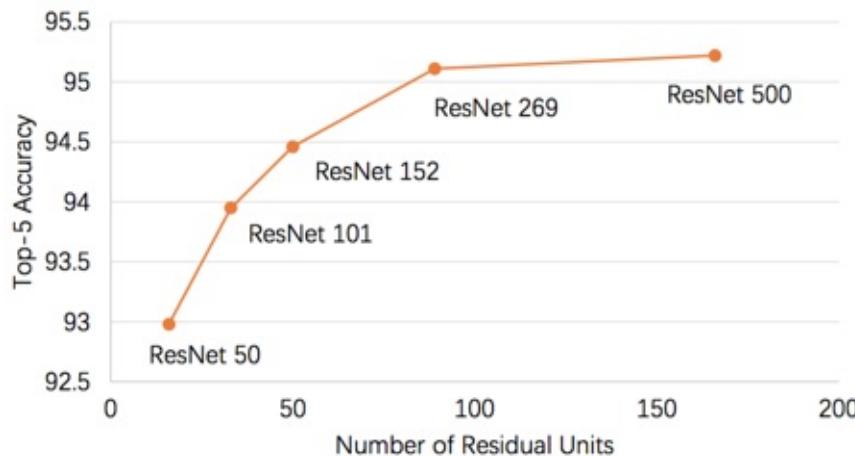


图1：网络深度和精度的关系

因此作者效仿IR的思想，希望通过更复杂的block结构来获得比增加深度更大的效益，这种策略在真实场景中还是非常有用的，即如何在有限的硬件资源条件下最大化模型的精度。

1.2 多项式模型

本文是从多项式的角度推导block结构的。首先一个经典的残差block可以表示为：

$$(I + F) \cdot \mathbf{x} = \mathbf{x} + F \cdot \mathbf{x} := \mathbf{x} + F(\mathbf{x})$$

其中 \mathbf{x} 是输入， I 是单位映射，‘+’单位加操作，表示在残差网络中 F 是两个连续的卷积操作。如果 F 是Inception的话，上式便是IR的表达式，如图2所示。

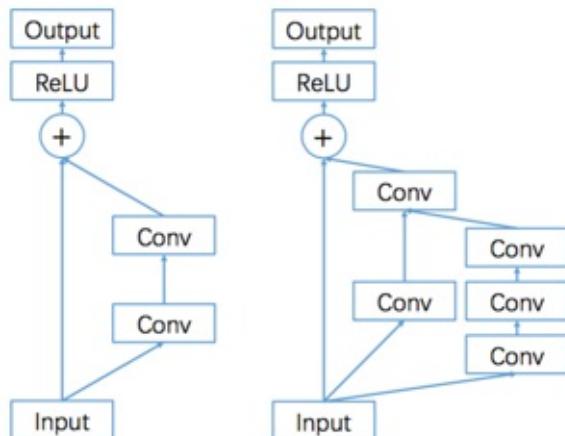


图2：（左）经典残差网络，（右）：Inception v4

下面我们将 F 一直看做Inception，然后通过将上式表示为更复杂的多项式的形式来推导出几个更复杂的结构。

- $poly-2 : I + F + F^2$ 。在这个形式中，网络有三个分支，左侧路径是一个直接映射，中间路径是一个Inception结构，右侧路径是两个连续的Inception，如图3(a)所示。在这个网络中，所有Inception的参数是共享的，所以不会引入额外的参数。由于参数共享，我们可以推出它的等价形式，如图3(b)。因为 $I + F + F^2 = I + (I + F)F$ ，而且这种形式的网络计算量少了 $1/3$ 。
- $mpoly-2 : I + F + GF$ 。这个block的结构和图3(b)相同，不同之处是两个Inception的参数不共享。其也可以表示为 $I + (I + G)F$ ，如图3(c)所示。它具有更强的表达能力，但是参数数量也加倍了。
- $2-way : I + F + G$ 。即向网络中添加一个额外且参数不共享的残差块，思想和Multi-Residual Networks[93]相同，如图3(d)。

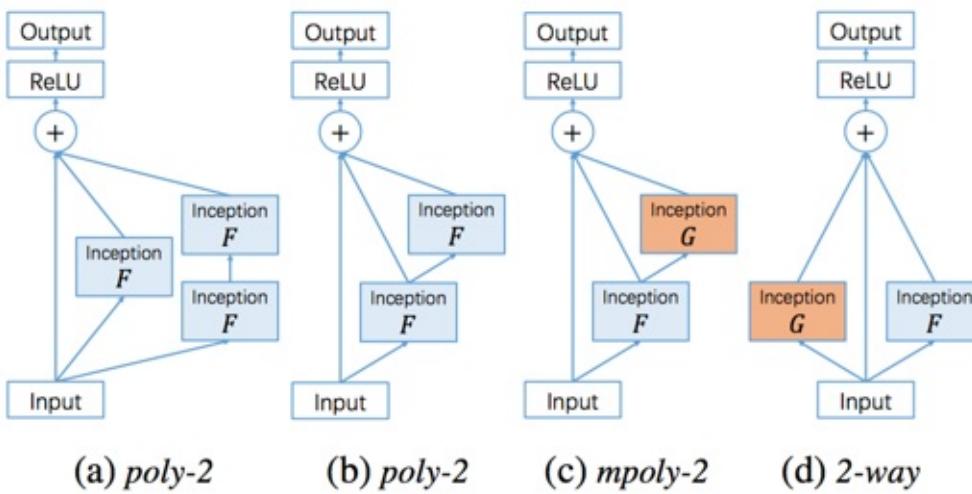


图3：PolyNet的几种block

结合上文提出的多项式的思想，几乎可以衍生出无限的网络模型，出于对计算性能的考虑，我们仅考虑下面三个结构：

- $poly-3 : I + F + F^2 + F^3$ 。
- $mpoly-3 : I + F + GF + HGF$ 。
- $3-way : I + F + G + H$ 。

如果你看过DenseNet[117]的话，你会发现DenseNet本质上也是一个多项式模型，对于一个含有 n 个卷积的block块，可以用数学公式表示为：

$$I \oplus C \oplus C^2 \oplus \dots \oplus C^n$$

其中 C 表示一个普通的卷积操作， \oplus 表示特征拼接。

1.3 对照试验

如图4所示，我们把IR分成A，B，C共3个阶段，它们处理的Feature Map尺寸分别是 35×35 , 17×17 , 8×8 。如果将A，B，C分别替换为1.2中提出的6个模型，我们可以得到共18个不同的网络结构，给与它们相同的超参数，我们得到的实验结果如图5。

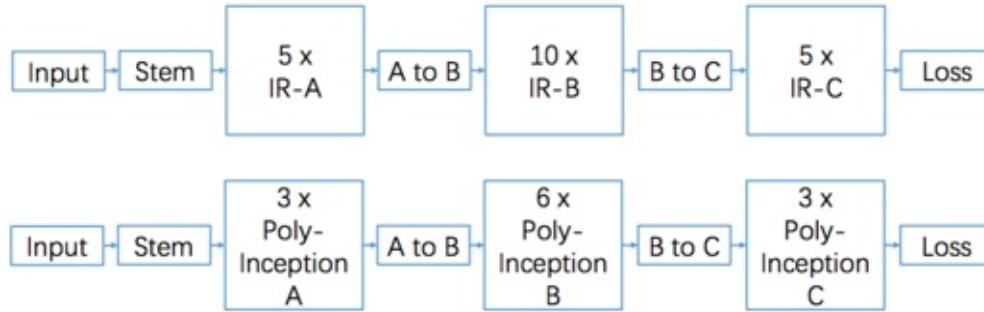


图4：*Inception*的三个阶段和*PolyNet*提供的替换方式

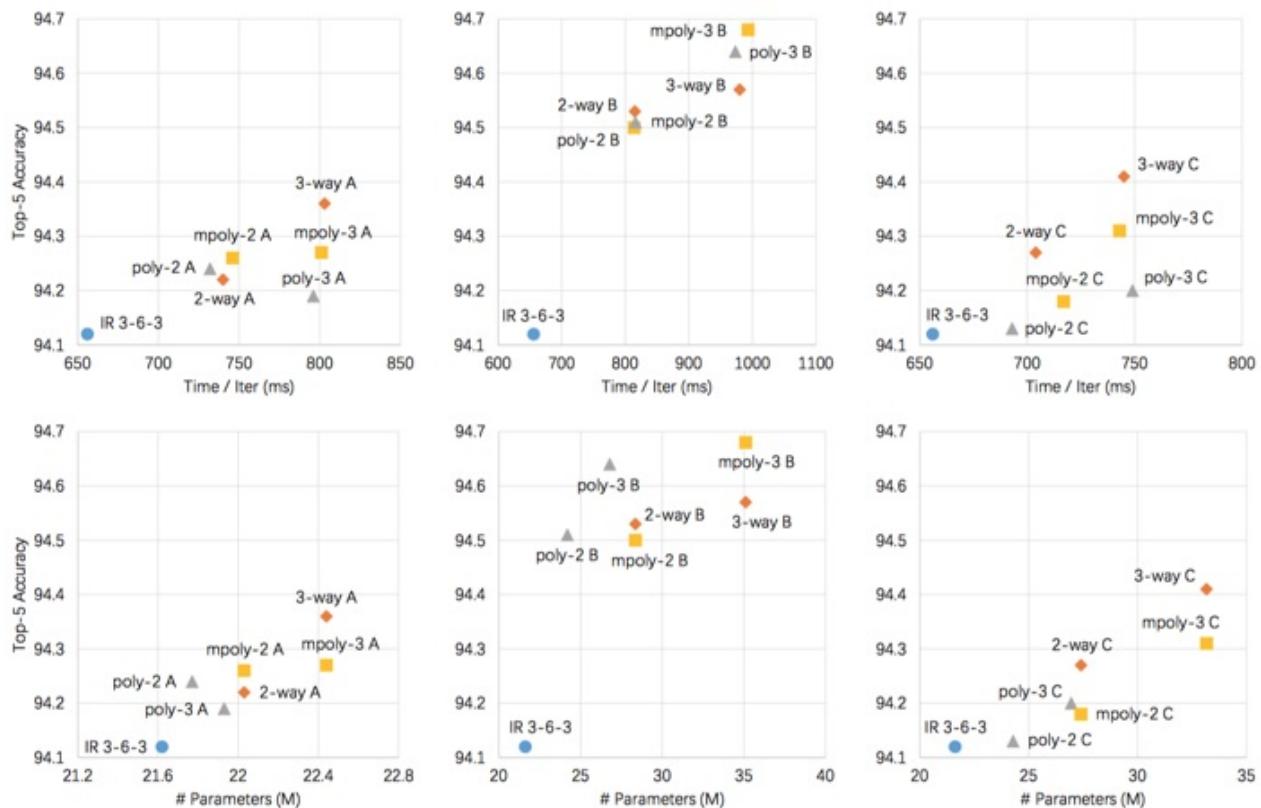


图5：*PolyNet*精度图，上面比较的训练时间和精度的关系，下面比较的是参数数量和精度的关系

通过图5我们可以抽取到下面几条的重要信息：

1. Stage-B的替换最有效；

2. Stage-B中使用 $mpoly$ -3最有效， $poly$ -3次之，但是 $poly$ -3的参数数量要少于 $mpoly$ -3；
 3. Stage-A和Stage-C均是使用3-way替换最有效，但是引入的参数也最多；
 4. 3路Inception的结构一般要优于2路Inception。

另外一种策略是混合的使用3路Inception的混合模型，在论文中使用的是4组3-way \rightarrow *mpoly-3* \rightarrow *_poly-3*替换IR中的阶段B，实验结果表明这种替换的效果要优于任何形式的非混合模型。

注意上面的几组实验

1.4 Very Deep PolyNet

基于上面提出的几个模型，作者提出了state-of-the-art的Very Deep PolyNet，结构如下：

- stageA：包含10个2-way的基础模块
 - stageB：包含10个poly-3，2-way混合的基础模块（即20个基础模型）
 - stageC：包含5个poly-3，2-way混合的基础模块（即10个基础模块）

初始化：作者发现如果先搭好网络再使用随机初始化的策略非常容易导致模型不稳定，论文使用了两个策略：

1. *initialization by insertion*(插入初始化)：策略是先使用迁移学习训练一个IR模型，再通过向其中插入一个Inception块构成poly-2；
 2. *interleaved* (交叉插入)：当加倍网络的深度时，将新加入的随机初始化的模型交叉的插入到迁移学习的模型中效果更好。

初始化如图6所示。

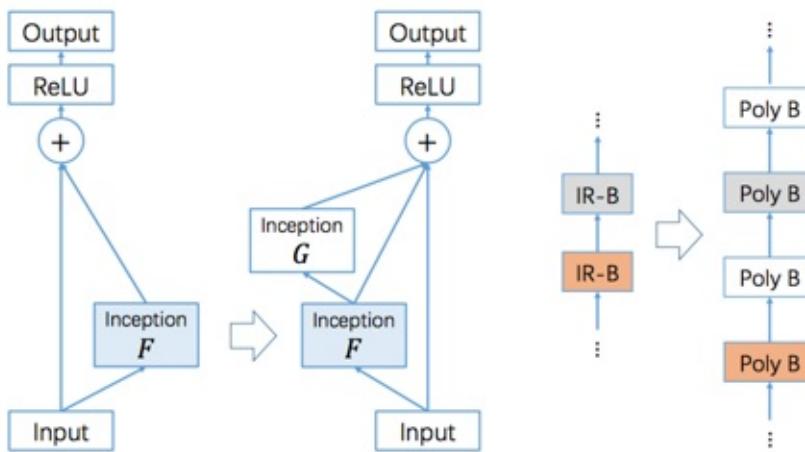


图6：（左）*initialization by insertion*，（右）*interleaved*

随机路径：收到Dropout的启发，PolyNet在训练的时候会随机丢掉多项式block中的一项或几项，如图7所示。

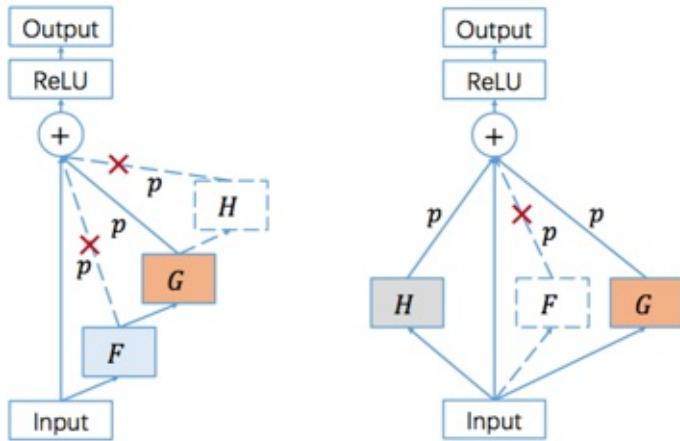


图7：随机路径。（左）： $I+F+GF+HGF$ to $I+GF$ ；（右） $I+F+G+H$ to $I+G+H$

加权路径：简单版本的多项式结构容易导致模型不稳定，Very Deep PolyNet提出的策略是为Inception部分乘以权值 β ，例如2-way的表达式将由 $I + F + G$ 变成 $I + \beta F + \beta G$ ，论文给出的 β 的参考值是0.3。

2. 总结

PolyNet从多项式的角度提出了更多由Inception和残差块组合而成的网络结构，模型并没有创新性。最大的优点在于从多项式的角度出发，并且我们从这个角度发现了PolyNet和DenseNet有异曲同工之妙。

论文中最优结构的选择是通过实验得出的，如果能结合数学推导得出前因后果本文将上升到另一个水平。结合上面的Block，作者提出了混合模型Very Deep PolyNet并在ImageNet取得了目前的效果。

最后训练的时候使用的初始化策略和基于集成思想的随机路径反而非常有实用价值。

Squeeze-and-Excitation Networks

前言

SENet[116]的提出动机非常简单，传统的方法是将网络的Feature Map等权重的传到下一层，SENet的核心思想在于建模通道之间的相互依赖关系，通过网络的全局损失函数自适应的重新矫正通道之间的特征相应强度。

SENet由一些列SE block组成，一个SE block的过程分为Squeeze（压缩）和Excitation（激发）两个步骤。其中Squeeze通过在Feature Map层上执行Global Average Pooling得到当前Feature Map的全局压缩特征向量，Excitation通过两层全连接得到Feature Map中每个通道的权值，并将加权后的Feature Map作为下一层网络的输入。从上面的分析中我们可以看出SE block只依赖与当前的一组Feature Map，因此可以非常容易的嵌入到几乎现在所有的卷积网络中。论文中给出了在当时state-of-the-art的Inception[121]和残差网络[118]插入SE block后的实验结果，效果提升显著。

SENet虽然引入了更多的操作，但是其带来的性能下降尚在可以接受的范围之内，从GFLOPs，参数数量以及运行时间的实验结果上来看，SENet的损失并不是非常显著。

1. SENet详解

1.1. SE Block

一个SE Block的结构如图1所示

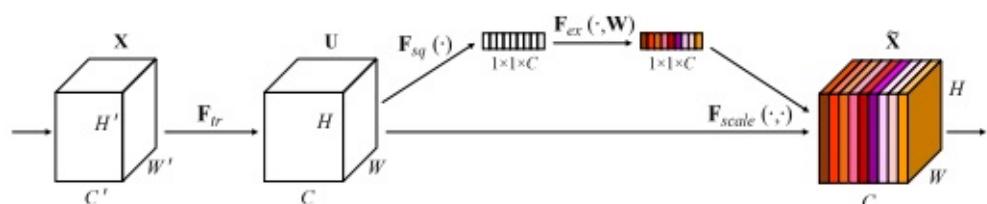


图1：SENet网络结构

网络的左半部分是一个传统的卷积变换，忽略掉这一部分并不会影响我们的SENet的理解。我们直接看一下后半部分，其中 U 是一个 $W \times H \times C$ 的Feature Map， (W, H) 是图像的尺寸， C 是图像的通道数。

经过 $F_{sq}(\cdot)$ (Squeeze操作) 后，图像变成了一个 $1 \times 1 \times C$ 的特征向量，特征向量的值由 U 确定。经过 $F_{ex}(\cdot, \mathbf{W})$ 后，特征向量的维度没有变，但是向量值变成了新的值。这些值会通过和 U 的 $F_{scale}(\cdot, \cdot)$ 得到加权后的 \tilde{X} 。 \tilde{X} 和 U 的维度是相同的。

1.2. Squeeze

Squeeze部分的作用是获得Feature Map \mathbf{U} 的每个通道的全局信息嵌入（特征向量）。在SE block中，这一步通过VGG中引入的Global Average Pooling (GAP) 实现的。也就是通过求每个通道 $c, c \in \{1, C\}$ 的Feature Map的平均值：

$$z_c = \mathbf{F}_{sq}(\mathbf{u}_c) = \frac{1}{W \times H} \sum_{i=1}^W \sum_{j=1}^H u_c(i, j)$$

通过GAP得到的特征值是全局的（虽然比较粗糙）。另外， z_c 也可以通过其它方法得到，要求只有一个，得到的特征向量具有全局性。

1.3. Excitation

Excitation部分的作用是通过 z_c 学习 C 中每个通道的特征权值，要求有两点：

1. 要足够灵活，这样能保证学习到的权值比较具有价值；
2. 要足够简单，这样不至于添加SE blocks之后网络的训练速度大幅降低；
3. 通道之间的关系是non-exclusive的，也就是说学习到的特征能过激励重要的特征，抑制不重要的特征。

根据上面的要求，SE blocks使用了两层全连接构成的门机制 (gate mechanism)。门控单元 \mathbf{s} （即图1中 $1 \times 1 \times C$ 的特征向量）的计算法方式表示为：

$$\mathbf{s} = \mathbf{F}_{ex}(\mathbf{z}, \mathbf{W}) = \sigma(g(\mathbf{z}, \mathbf{W})) = \sigma(g(\mathbf{W}_2 \delta(\mathbf{W}_1 \mathbf{z})))$$

其中 δ 表示ReLU激活函数， σ 表示sigmoid激活函数。 $\mathbf{W}_1 \in \mathbb{R}^{\frac{C}{r} \times C}$, $\mathbf{W}_2 \in \mathbb{R}^{C \times \frac{C}{r}}$ 分别是两个全连接层的权值矩阵。 r 则是中间层的隐层节点数，论文中指出这个值是16。

得到门控单元 \mathbf{s} 后，最后的输出 \tilde{X} 表示为 \mathbf{s} 和 \mathbf{U} 的向量积，即图1中的 $\mathbf{F}_{scale}(\cdot, \cdot)$ 操作：

$$\tilde{x}_c = \mathbf{F}_{scale}(\mathbf{u}_c, s_c) = s_c \cdot \mathbf{u}_c$$

其中 \tilde{x}_c 是 \tilde{X} 的一个特征通道的一个Feature Map， s_c 是门控单元 \mathbf{s} （是个向量）中的一个标量值。

以上就是SE blocks算法的全部内容，SE blocks可以从两个角度理解：

1. SE blocks学习了每个Feature Map的动态先验；
2. SE blocks可以看做在Feature Map方向的Attention，因为注意力机制的本质也是学习一组权值。

1.4. SE-Inception 和 SE-ResNet

SE blocks的特性使其能够非常容易的和目前主流的卷积结构结合，例如论文中给出的Inception结构和残差网络结构，如图2。结合方式也非常简单，只需要在Inception blocks或者Residual blocks之后直接接上SE blocks即可。

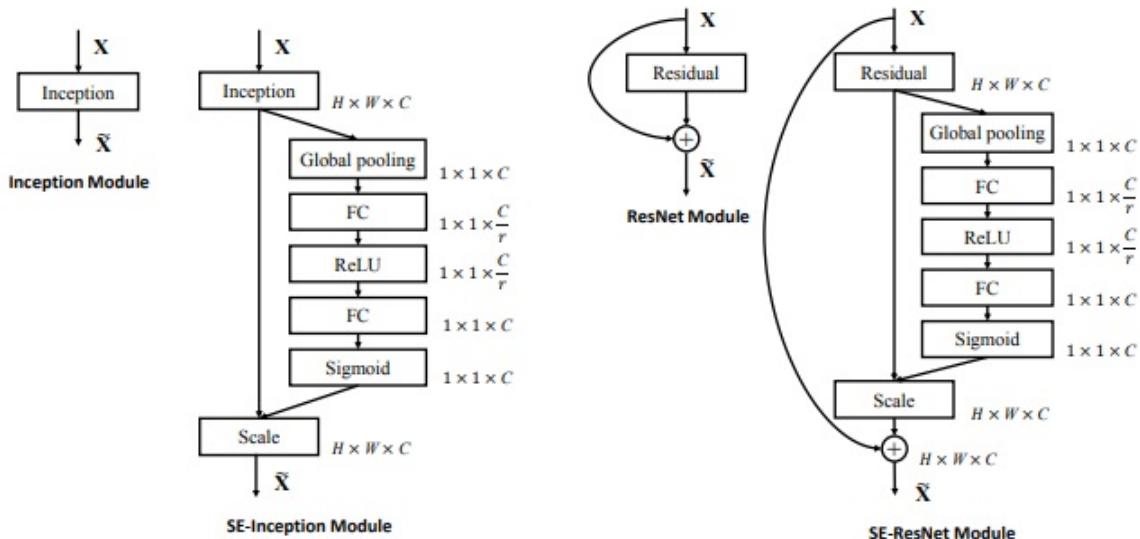


图2：SE-Inception和SE-Resnet

2. SENet的复杂性分析

SENet的本质是根据Feature Map的值学习每个Feature Map的权值。 \mathbf{U} 往往是一个由几万个节点值组成的三维矩阵，但是我们得到的 \mathbf{s} 却只有 C 个值，这种 $H \times W$ 程度的压缩是具有非常大的可操作性的。例如将 \mathbf{U} 展开成 $(W \times H \times C) \times 1$ 的特征向量，然后再通过全连接得到 \mathbf{s} ，这也是目前主流的Feature Map到FC的连接方式（`Flatten()`操作）。而且这种方式得到的 \mathbf{s} 往往也是效果优于SE blocks的策略的。但是SENet没这么做，他的原因是SENet是可以添加到网络中的任意一层之后的，而全连接操作往往是整个网络结构的性能瓶颈，尤其是当网络的节点数非常大时。

论文中主要对比了ResNet-50以及在其中的每一层之后添加了SE blocks之后的在运行性能的各方面的指标：

从计算性能的方向分析：ResNet-50需要约3.86GFLOPS，而SE-ResNet-50仅仅多了0.01个GFLOPS。

从预测速度上来看，运行一个ResNet-50的时间是190ms，SE-ResNet-50的运行时间约209ms，多了10%。

从参数数量上来看，SE-ResNet-50比ResNet-50的2500万个参数多了约250万个，约占10%。而且作者发现ResNet-50的最后几层的SE blocks可以省掉，但是性能影响并不大，这样的网络参数仅多了4%。

3. 总结

SENet的思想非常简单，即通过Feature Map为自身学习一个特征权值，通过单位乘的方式得到一组加权后的新的特征权值。使用的网络结构则是先GAP再接两层全连接的方式得到的权值向量。方法虽然简单，但是非常实用，并在ImageNet-2017上取得了非常优异的比赛成绩。

第2节对复杂性的分析引发了我们对SE blocks的进一步联想：如何在计算量和性能之间进行权衡？

下面是我的几点思考：

1. 先通过RoI Pooling得到更小的Feature Map（例如 3×3 ），在展开作为全连接的输入；
2. 在网络的深度和隐层节点的数目进行权衡，究竟是更深的网络效果更好还是更宽的网络效果更好；
3. 每一层的SE blocks是否要一定相同，比如作者发现浅层更需要SE blocks，那么我们能否给浅层使用一个计算量更大但是性能更好的SE block，而深层的SE blocks更为简单高效，例如单层全连接等。

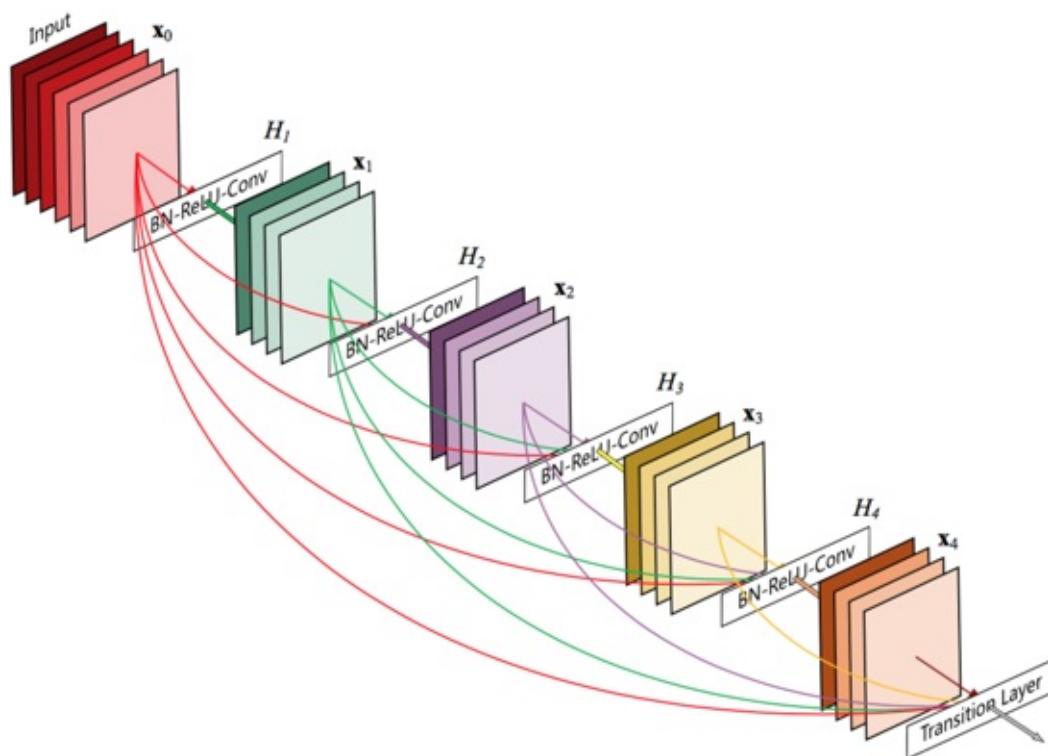
Densely Connected Convolutional Networks

前言

在残差网络的文章中，我们知道残差网格[118],[95]能够应用在特别深的网络中的一个重要原因是，无论正向计算精度还是反向计算梯度，信息都能毫无损失的从一层传到另一层。如果我们的目的是保证信息毫无阻碍的传播，那么残差网络的stacking残差块的设计便不是信息流通最合适的结构。

基于信息流通的原理，一个最简单的思想便是在网络中的每个卷积操作中，将其低层的所有特征作为该网络的输入，也就是在一个层数为 L 的网络中加入 $\frac{L(L+1)}{2}$ 个short-cut，如图1。为了更好的保存低层网络的特征，DenseNet [117]使用的是将不同层的输出拼接在一起，而在残差网络中使用的是单位加操作。以上便是DenseNet算法的动机。

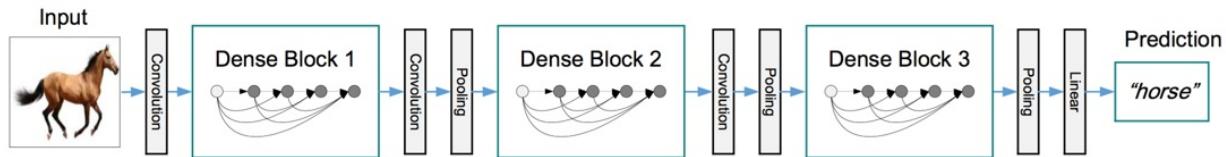
图1：DenseNet中一个Dense Block的设计



1. DenseNet算法解析及源码实现

在DenseNet中，如果全部采用图1的结构的话，第L层的输入是之前所有的Feature Map拼接在一起。考虑到现今内存/显存空间的问题，该方法显然是无法应用到网络比较深的模型中的，故而DenseNet采用了图2所示的堆积Dense Block的形式，下面我们针对图2详细解析DenseNet算法。

图2：DenseNet网络结构



1.1 Dense Block

图1便是一个Dense Block，在Dense Block中，第 l 层的输入 x_l 是这个块中前面所有层的输出：

$$x_l = [y_0, y_1, \dots, y_{l-1}]$$

$$y_l = H_l(x_l)$$

其中，中括号 $[y_0, y_1, \dots, y_{l-1}]$ 表示拼接操作，即按照Feature Map将 $l-1$ 个输入拼接成一个Tensor。 $H_l(\cdot)$ 表示合成函数（Composite function）。在实现时，我使用了`stored_features`存储每个合成函数的输出。

```
def dense_block(x, depth=5, growth_rate = 3):
    nb_input_feature_map = x.shape[3].value
    stored_features = x
    for i in range(depth):
        feature = composite_function(stored_features, growth_rate = growth_rate)
        stored_features = concatenate([stored_features, feature], axis=3)
    return stored_features
```

1.2 合成函数（Composite function）

合成函数位于Dense Block的每一个节点中，其输入是拼接在一起的Feature Map，输出则是这些特征经过BN->ReLU->3*3卷积的三步得到的结果，其中卷积的Feature Map的数量是成长率（Growth Rate）。在DenseNet中，成长率 k 一般是个比较小的整数，在论文中， $k = 12$ 。但是拼接在一起的Feature Map的数量一般比较大，为了提高网络的计算性能，DenseNet先使用了 1×1 卷积将输入数据降维到 $4k$ ，再使用 3×3 卷积提取特征，作者将这一过程标准化为BN->ReLU-> 1×1 卷积->BN->ReLU-> 3×3 卷积，这种结构定义为DenseNetB。

```

def composite_function(x, growth_rate):
    if DenseNetB: #Add 1*1 convolution when using DenseNet B
        x = BatchNormalization()(x)
        x = Activation('relu')(x)
        x = Conv2D(kernel_size=(1,1), strides=1, filters = 4 * growth_rate, padding='same')(x)
        x = BatchNormalization()(x)
        x = Activation('relu')(x)
        output = Conv2D(kernel_size=(3,3), strides=1, filters = growth_rate, padding='same')(x)
    )(x)
    return output

```

1.3 成长率 (Growth Rate)

成长率 k 是DenseNet的一个超参数，反应的是Dense Block中每个节点的输入数据的增长速度。在Dense Block中，每个节点的输出均是一个 k 维的特征向量。假设整个Dense Block的输入数据是 k_0 维的，那么第 l 个节点的输入便是 $k_0 + k \times (l - 1)$ 。作者通过实验验证， k 一般取一个比较小的值，作者通过实验将 k 设置为12。

1.4 Compression

至此，DenseNet的Dense Block已经介绍完毕，在图2中，Dense Block之间的结构叫做压缩层（Compression Layer）。压缩层有降维和降采样两个作用。假设Dense Block的输出是 m 维的特征向量，那么下一个Dense Block的输入是 $\lfloor \theta m \rfloor$ ，其中 θ 是压缩因子（Compression Factor），用户自行设置的超参数。当 θ 等于1时，Dense Block的输入和输出的维度相同，当 $\theta < 1$ 时，网络叫做DenseNet-C，在论文中， $\theta = 0.5$ 。包含瓶颈层和压缩层的DenseNet叫做DenseNet-BC。Pooling层使用的是 2×2 的Average Pooling层。

下面Demo是在MNIST数据集上的DenseNet代码，完整代码

见：<https://github.com/senliuy/CNN-Structures/blob/master/DenseNet.ipynb>

```
def dense_net(input_image, nb_blocks = 2):
    x = Conv2D(kernel_size=(3,3), filters=8, strides=1, padding='same', activation='relu')(input_image)
    for block in range(nb_blocks):
        x = dense_block(x, depth=NB_DEPTH, growth_rate = GROWTH_RATE)
        if not block == nb_blocks-1:
            if DenseNetC:
                theta = COMPRESSION_FACTOR
                nb_transition_filter = int(x.shape[3].value * theta)
                x = Conv2D(kernel_size=(1,1), filters=nb_transition_filter, strides=1, padding='same', activation='relu')(x)
                x = AveragePooling2D(pool_size=(2,2), strides=2)(x)
            x = Flatten()(x)
            x = Dense(100, activation='relu')(x)
    outputs = Dense(10, activation='softmax', kernel_initializer='he_normal')(x)
    return outputs
```

2. 分析：

DenseNet具有如下优点：

1. 信息流通更为顺畅；
2. 支持特征重用；
3. 网络更窄

由于DenseNet需要在内存中保存Dense Block的每个节点的输出，此时需要极大的显存才能支持较大规模的DenseNet，这也导致了现在工业界主流的算法依旧是残差网络。

SQUEEZENET: ALEXNET-LEVEL ACCURACY WITH 50X FEWER PARAMETERS AND < 0.5MB MODEL SIZE

tags: AlexNet, SqueezeNet

前言

从LeNet5到DenseNet，反应卷积网络的一个发展方向：提高精度。这里我们开始另外一个方向的介绍：在大幅降低模型精度的前提下，最大程度的提高运算速度。

提高运算所读有两个可以调整的方向：

1. 减少可学习参数的数量；
2. 减少整个网络的计算量。

这个方向带来的效果是非常明显的：

1. 减少模型训练和测试时候的计算量，单个step的速度更快；
2. 减小模型文件的大小，更利于模型的保存和传输；
3. 可学习参数更少，网络占用的显存更小。

SqueezeNet[110]正是诞生在这个环境下的一个精度的网络，它能够在ImageNet数据集上达到AlexNet[123]近似的效果，但是参数比AlexNet少50倍，结合他们的模型压缩技术 Deep Compression[92]，模型文件可比AlexNet小510倍。

1. SqueezeNet 详解

1.1 SqueezeNet的压缩策略

SqueezeNet的模型压缩使用了3个策略：

1. 将 3×3 卷积替换成 1×1 卷积：通过这一步，一个卷积操作的参数数量减少了9倍；
2. 减少 3×3 卷积的通道数：一个 3×3 卷积的计算量是 $3 \times 3 \times M \times N$ （其中 M, N 分别是输入Feature Map和输出Feature Map的通道数），作者任务这样一个计算量过于庞大，因此希望将 M, N 减小以减少参数数量；
3. 将降采样后置：作者认为较大的Feature Map含有更多的信息，因此将降采样往分类层移动。注意这样的操作虽然会提升网络的精度，但是它有一个非常严重的缺点：即会增加网络的计算量。

1.2 Fire模块

SqueezeNet是由若干个Fire模块结合卷积网络中卷积层，降采样层，全连接等层组成的。一个Fire模块由Squeeze部分和Expand部分组成（注意区分和Momenta的SENet[116]的区别）。Squeeze部分是一组连续的 1×1 卷积组成，Expand部分则是一组连续的 1×1 卷积和一组连续的 3×3 卷积concatenate组成，因此 3×3 卷积需要使用same卷积，Fire模块的结构见图1。在Fire模块中，Squeeze部分 1×1 卷积的通道数记做 s_{1x1} ，Expand部分 1×1 卷积和 3×3 卷积的通道数分别记做 e_{1x1} 和 e_{3x3} （论文图画的不好，不要错误的理解成卷积的层数）。在Fire模块中，作者建议 $s_{1x1} < e_{1x1} + e_{3x3}$ ，这么做相当于在两个 3×3 卷积的中间加入了瓶颈层，作者的实验中的一个策略是 $s_{1x1} = \frac{e_{1x1}}{4} = \frac{e_{3x3}}{4}$ 。图1中 $s_{1x1} = 3$ ， $e_{1x1} = e_{3x3} = 4$ 。

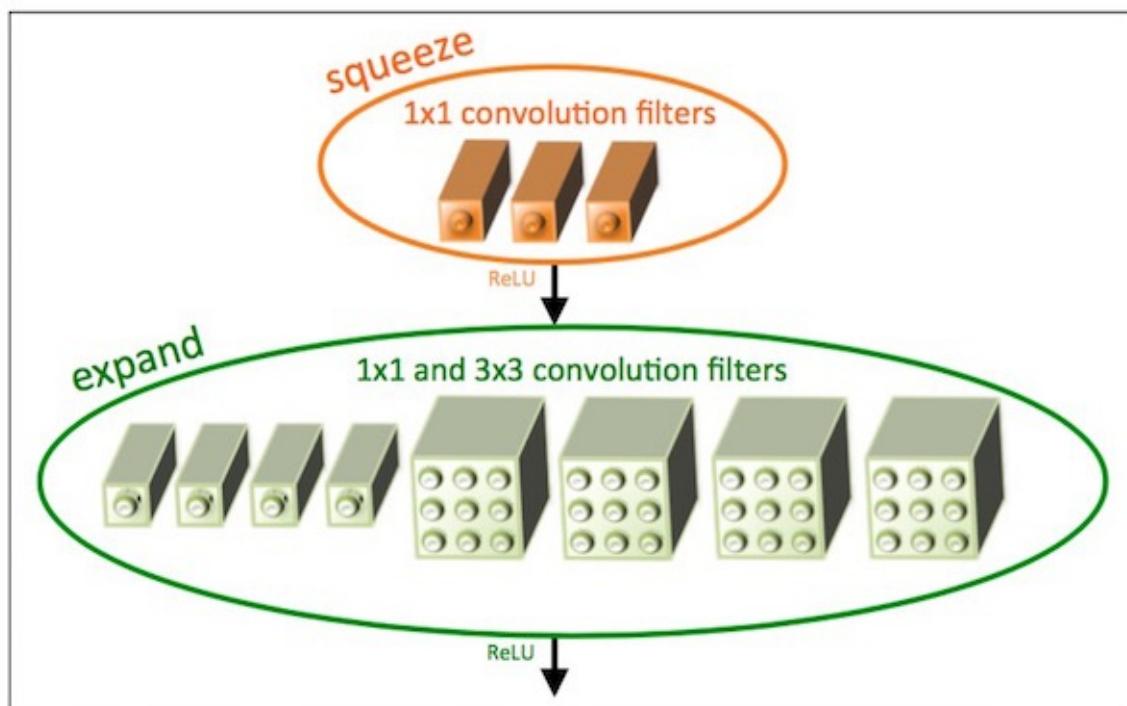


图1：SqueezeNet的Fire模块

下面代码片段是Keras实现的Fire模块，注意拼接Feature Map的时候使用的是Concatenate操作，这样不必要求 $e_{1x1} = e_{3x3}$ 。

```
def fire_model(x, s_1x1, e_1x1, e_3x3, fire_name):
    # squeeze part
    squeeze_x = Conv2D(kernel_size=(1,1), filters=s_1x1, padding='same', activation='relu',
, name=fire_name+'_s1')(x)
    # expand part
    expand_x_1 = Conv2D(kernel_size=(1,1), filters=e_1x1, padding='same', activation='rel
u', name=fire_name+'_e1')(squeeze_x)
    expand_x_3 = Conv2D(kernel_size=(3,3), filters=e_3x3, padding='same', activation='rel
u', name=fire_name+'_e3')(squeeze_x)
    expand = merge([expand_x_1, expand_x_3], mode='concat', concat_axis=3)
    return expand
```

图2是使用Keras自带的 `plot_model` 功能得到的Fire模块的可视图，其中

$$s_{1x1} = \frac{e_{1x1}}{4} = \frac{e_{3x3}}{4} = 16^\circ$$

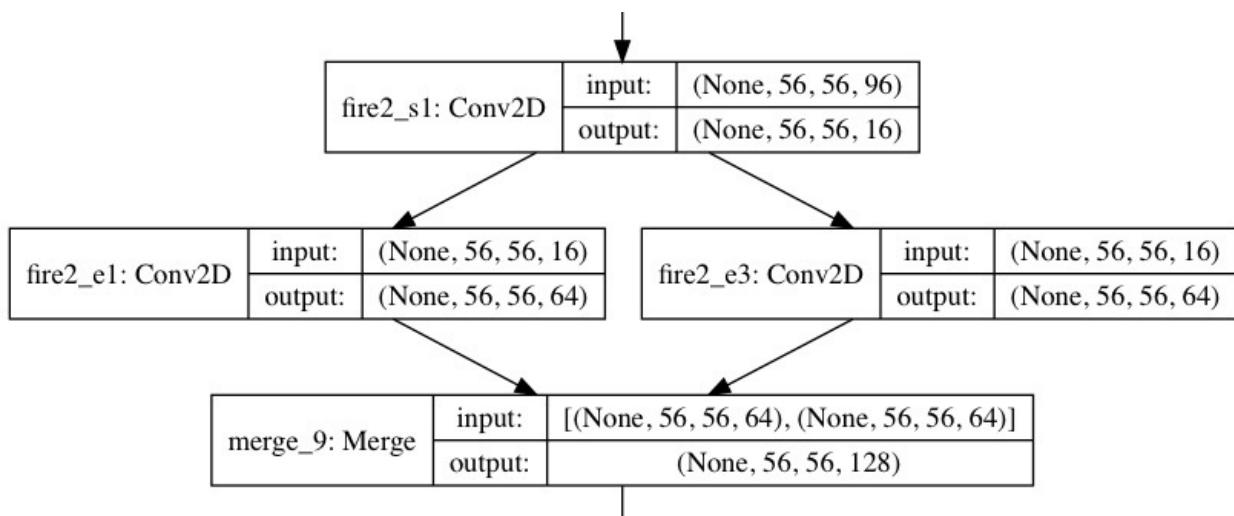


图2：Keras可视化的SqueezeNet的Fire模块

1.3 SqueezeNet的网络架构

图3是SqueezeNet的几个实现，左侧是不加short-cut的SqueezeNet，中间是加了short-cut的，右侧是short-cut跨有不同Feature Map个数的卷积的。还有一些细节图3中并没有体现出来：

1. 激活函数默认都使用ReLU；
 2. fire9之后接了一个rate为0.5的dropout；
 3. 使用same卷积。

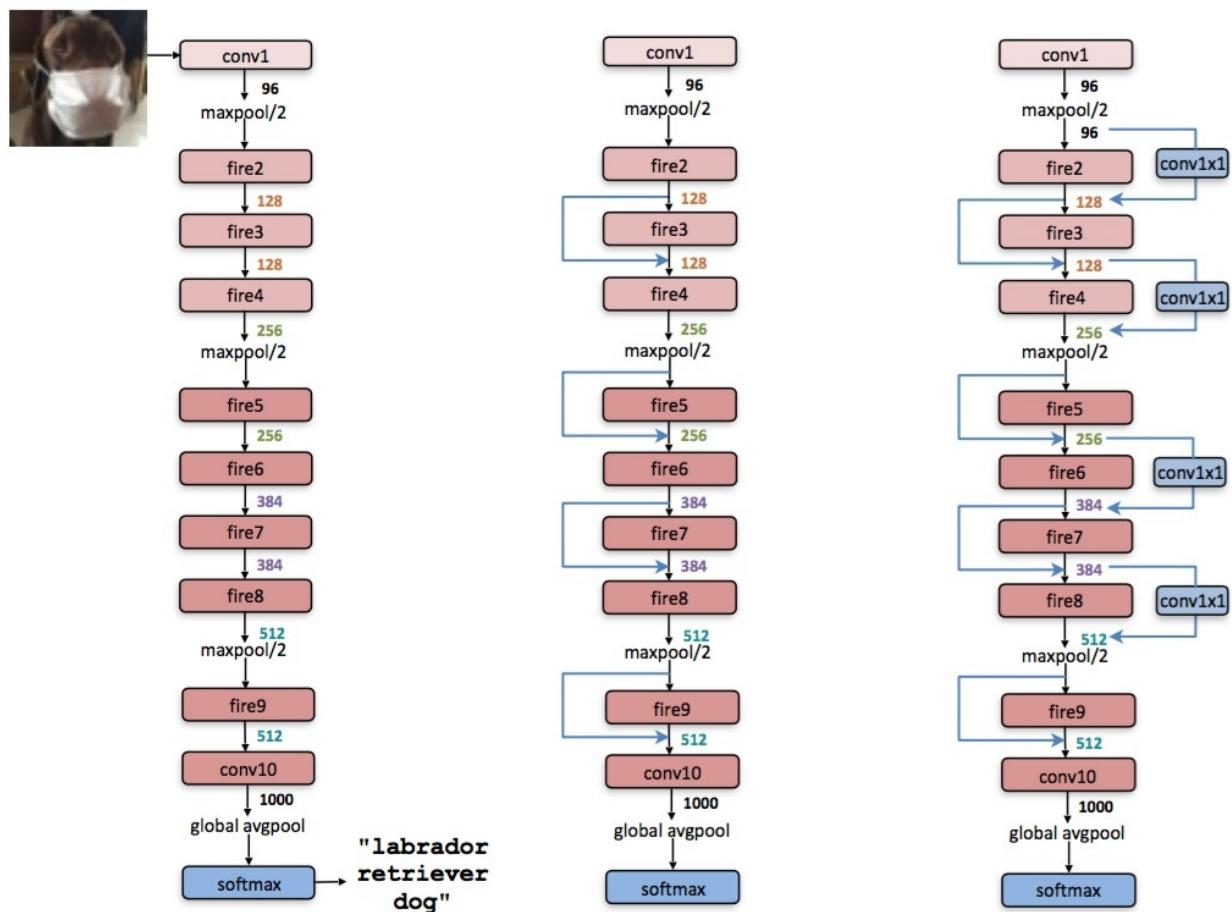


图3：SqueezeNet网络结构

表1给出了SqueezeNet的详细参数：

layer name/type	output size	filter size / stride (if not a fire layer)	depth	s_{1x1} (#1x1 squeeze)	e_{1x1} (#1x1 expand)	e_{3x3} (#3x3 expand)	s_{1x1} sparsity	e_{1x1} sparsity	e_{3x3} sparsity	# bits	#parameter before pruning	#parameter after pruning
input image	224x224x3										-	-
conv1	111x111x96	7x7/2 (x96)	1						100% (7x7)	6bit	14,208	14,208
maxpool1	55x55x96	3x3/2	0									
fire2	55x55x128		2	16	64	64	100%	100%	33%	6bit	11,920	5,746
fire3	55x55x128		2	16	64	64	100%	100%	33%	6bit	12,432	6,258
fire4	55x55x256		2	32	128	128	100%	100%	33%	6bit	45,344	20,646
maxpool4	27x27x256	3x3/2	0									
fire5	27x27x256		2	32	128	128	100%	100%	33%	6bit	49,440	24,742
fire6	27x27x384		2	48	192	192	100%	50%	33%	6bit	104,880	44,700
fire7	27x27x384		2	48	192	192	50%	100%	33%	6bit	111,024	46,236
fire8	27x27x512		2	64	256	256	100%	50%	33%	6bit	188,992	77,581
maxpool8	13x12x512	3x3/2	0									
fire9	13x13x512		2	64	256	256	50%	100%	30%	6bit	197,184	77,581
conv10	13x13x1000	1x1/1 (x1000)	1					20% (3x3)		6bit	513,000	103,400
avgpool10	1x1x1000	13x13/1	0									
activations parameters compression info											1,248,424 (total)	421,098 (total)

表1 : SqueezeNet网络参数

根据表1，我们的Keras实现如下面代码片段：

```
def squeezeNet(x):
    conv1 = Conv2D(input_shape = (224, 224, 3), strides = 2, filters=96, kernel_size=(7, 7),
    ), padding='same', activation='relu')(x)
    pool1 = MaxPool2D((2, 2))(conv1)
    fire2 = fire_model(pool1, 16, 64, 64, 'fire2')
    fire3 = fire_model(fire2, 16, 64, 64, 'fire3')
    fire4 = fire_model(fire3, 32, 128, 128, 'fire4')
    pool2 = MaxPool2D((2, 2))(fire4)
    fire5 = fire_model(pool2, 32, 128, 128, 'fire5')
    fire6 = fire_model(fire5, 48, 192, 192, 'fire6')
    fire7 = fire_model(fire6, 48, 192, 192, 'fire7')
    fire8 = fire_model(fire7, 64, 256, 256, 'fire8')
    pool3 = MaxPool2D((2, 2))(fire8)
    fire9 = fire_model(pool3, 64, 256, 256, 'fire9')
    dropout1 = Dropout(0.5)(fire9)
    conv10 = Conv2D(kernel_size=(1,1), filters=1000, padding='same', activation='relu')(dropout1)
    gap = GlobalAveragePooling2D()(conv10)
    return gap
```

上面的代码，模型的summary，以及SqueezeNet的keras可视化

见：<https://github.com/senliuy/CNN-Structures/blob/master/SqueezeNet.ipynb>。

1.4 SqueezeNet的性能以及Deep Compression

图3左侧的SqueezeNet的性能（top1：57.5%，top5：80.3%）是可以类似AlexNet的（top1：57.2%，top5：80.3%）。从表1中我们可以看出SqueezeNet总共有1,248,424个参数，同性能的AlexNet则有58,304,586个参数（主要集中于全连接层，去掉之后有3,729,472个）。使用他们提出的Deep Compression[3]算法压缩后，模型的参数数量可以降到421,098个。

2. 总结

SqueezeNet的压缩策略是依靠将 3×3 卷积替换成 1×1 卷积来达到的，其参数数量是等性能的AlexNet的2.14%。从参数数量上来看，SqueezeNet的目的达到了。SqueezeNet的最大贡献在于其开拓了模型压缩这一方向，之后的一系列文章也就此打开。

这里我们着重说一下SqueezeNet的缺点：

1. SqueezeNet的侧重的应用方向是嵌入式环境，目前嵌入式环境主要问题是实时性。SqueezeNet的通过更深的深度置换更少的参数数量虽然能减少网络的参数，但是其丧失了网络的并行能力，测试时间反而会更长，这与目前的主要挑战是背道而驰的；
2. 论文的题目非常标题党，虽然纸面上是减少了50倍的参数，但是问题的主要症结在于AlexNet本身全连接节点过于庞大，50倍参数的减少和SqueezeNet的设计并没有关系，考虑去掉全连接之后3倍参数的减少更为合适。
3. SqueezeNet得到的模型是5MB左右，0.5MB的模型还要得益于Deep Compression。虽然Deep Compression也是这个团队的文章，但是将0.5这个数列在文章的题目中显然不是很合适。

MobileNet v1 and MobileNet v2

tags: MobileNet

前言

MobileNet[109]（这里叫做MobileNet v1，简称v1）中使用的Depthwise Separable Convolution是模型压缩的一个最为经典的策略，它是通过将跨通道的 3×3 卷积换成单通道的 3×3 卷积+跨通道的 1×1 卷积来达到此目的。

MobileNet v2 [115]是在v1的Depthwise Separable的基础上引入了残差结构[118]。并发现了ReLU在通道数较少的Feature Map上有非常严重信息损失问题，由此引入了Linear Bottlenecks和Inverted Residual。

首先在这篇文章中我们会详细介绍两个版本的MobileNet，然后我们会介绍如何使用Keras实现这两个算法。

1. MobileNet v1

1.1 回顾：传统卷积的参数量和计算量

传统的卷积网络是跨通道的，对于一个通道数为 M 的输入Feature Map，我们要得到通道数为 N 的输出Feature Map。普通卷积会使用 N 个不同的 $D_K \times D_K \times M$ 以滑窗的形式遍历输入Feature Map，因此对于一个尺寸为 $D_K \times D_K$ 的卷积的参数个数为 $D_K \times D_K \times M \times N$ 。一个普通的卷积可以表示为：

$$G_{k,l,n} = \sum_{i,j,m} \mathbf{K}_{i,j,m,n} \cdot \mathbf{K}_{k+i-1,l+j-1,m}$$

它的一层网络的计算代价约为：

$$D_K \times D_K \times M \times N \times D_W \times D_H$$

其中 (D_W, D_H) 为Feature Map的尺寸。普通卷积如图1所示。

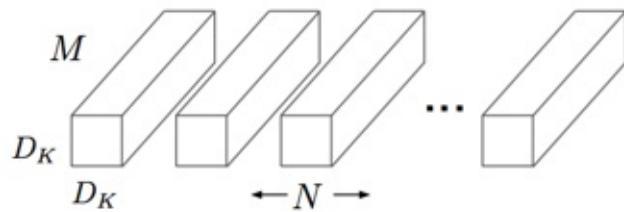


图1：普通卷积的Feature Map之间的卷积核情况

v1中介绍的Depthwise Separable Convolution就是解决了传统卷积的参数数量和计算代价过于高昂的问题。Depthwise Separable Convolution分成Depthwise Convolution和Pointwise Convolution。

1.2 Depthwise卷积

其中Depthwise卷积是指不跨通道的卷积，也就是说Feature Map的每个通道有一个独立的卷积核，并且这个卷积核作用且仅作用在这个通道之上，如图2所示。

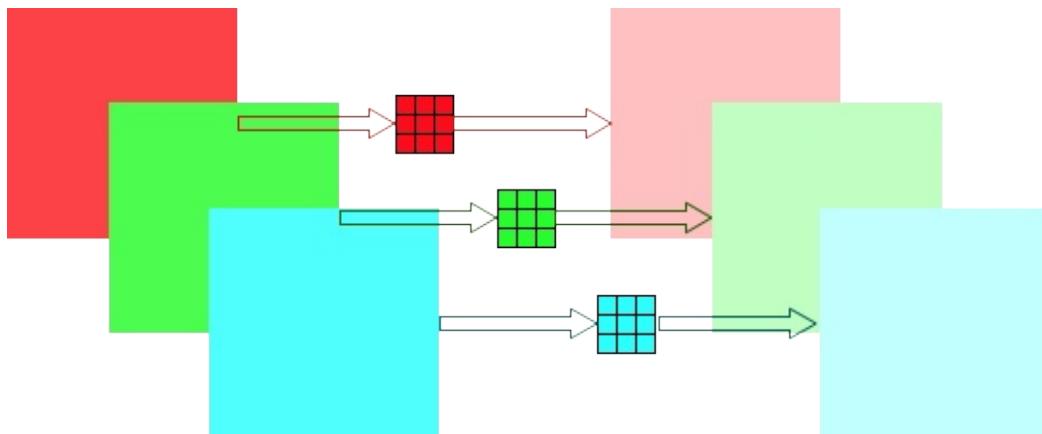


图2：Depthwise卷积示意图（3个通道）

从图2和图1的对比中我们可以看出，因为放弃了卷积时的跨通道。Depthwise卷积的参数数量为 $D_K \times D_K \times M$ 。Depthwise Convolution的数学表达式为：

$$\hat{G}_{k,l,m} = \sum_{i,j} \hat{K}_{i,j,n} \cdot F_{k+i-1,l+j-1,m}$$

它的计算代价也是传统卷积的 $\frac{1}{N}$ 为：

$$D_K \times D_K \times M \times D_W \times D_H$$

在Keras中，我们可以使用 `DepthwiseConv2D` 实现Depthwise卷积操作，它有几个重要的参数：

- `kernel_size`：卷积核的尺寸，一般设为3。
- `strides`：卷积的步长
- `padding`：是否加边
- `activation`：激活函数

由于Depthwise卷积的每个通道Feature Map产生且仅产生一个与之对应的Feature Map，也就是说输出层的Feature Map的channel数量等于输入层的Feature map的数量。因此 `DepthwiseConv2D` 不需要控制输出层的Feature Map的数量，因此并没有 `filters` 这个参数。

1.3 Pointwise 卷积

Depthwise卷积的操作虽然非常高效，但是它仅相当于对当前的Feature Map的一个通道施加了一个过滤器，并不会合并若干个特征从而生成新的特征，而且由于在Depthwise卷积中输出Feature Map的通道数等于输入Feature Map的通道数，因此它并没有升维或者降维的功能。

为了解决这些问题，v1中引入了Pointwise卷积用于特征合并以及升维或者降维。很自然的我们可以想到使用 1×1 卷积来完成这个功能。Pointwise的参数数量为 $M \times N$ ，计算量为：

$$M \times N \times D_W \times D_H$$

Pointwise的可视化如图3：

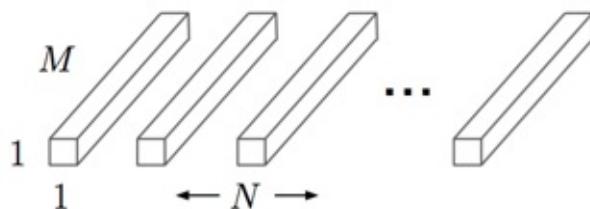


图3：Pointwise卷积示意图

1.4 Depthwise Separable 卷积

合并1.2中的Depthwise卷积和1.3中的Pointwise卷积便是v1中介绍的Depthwise Separable卷积。它的一组操作（一次Depthwise卷积加一次Pointwise卷积）的参数数量为：

$D_K \times D_K \times M + M \times N$ 是普通卷积的

$$\frac{D_K \times D_K \times M + M \times N}{D_K \times D_K \times M \times N} = \frac{1}{N} + \frac{1}{D_K^2}$$

计算量为：

$$D_K \times D_K \times M \times D_W \times D_H + M \times N \times D_W \times D_H$$

和普通卷积的比值为：

$$\frac{D_K \times D_K \times M \times D_W \times D_H + M \times N \times D_W \times D_H}{D_K \times D_K \times M \times N \times D_W \times D_H} = \frac{1}{N} + \frac{1}{D_K^2}$$

对于一个 3×3 的卷积而言，v1的参数量和计算代价均为普通卷积的 $\frac{1}{8}$ 左右。

1.5 Mobile v1的Keras实现及实验结果分析

通过上面的分析，我们知道一个普通卷积的一组卷积操作可以拆分成了个Depthwise卷积核一个Pointwise卷积，由此而形成MobileNet v1的结构。在这个实验中我们首先会搭建一个普通卷积，然后再将其改造成v1，并在MNIST上给出实验结果，代码和实验结果见链接[CPU](#)，[GPU](#)。

首先我们搭建的传统卷积的结构如下面代码片段：

```
def Simple_NaiveConvNet(input_shape, k):
    inputs = Input(shape=input_shape)
    x = Conv2D(filters=32, kernel_size=(3,3), strides=(2,2), padding='same', activation='relu', name='n_conv_1')(inputs)
    x = Conv2D(filters=64, kernel_size=(3,3), padding='same', activation='relu', name='n_conv_2')(x)
    x = Conv2D(filters=128, kernel_size=(3,3), padding='same', activation='relu', name='n_conv_3')(x)
    x = Conv2D(filters=128, kernel_size=(3,3), strides=(2,2), padding='same', activation='relu', name='n_conv_4')(x)
    x = GlobalAveragePooling2D(name='n_gap')(x)
    x = BatchNormalization(name='n_bn_1')(x)
    x = Dense(128, activation='relu', name='n_fc1')(x)
    x = BatchNormalization(name='n_bc_2')(x)
    x = Dense(k, activation='softmax', name='n_output')(x)
    model = Model(inputs, x)
    return model
```

通过将 3×3 的 Conv2D() 换成 3×3 的 DepthwiseConv2D 加上 1×1 的 Conv2D()（第一层保留传统卷积），我们将其改造成了MobileNet v1。

```

def Simple_MobileNetV1(input_shape, k):
    inputs = Input(shape=input_shape)
    x = Conv2D(filters=32, kernel_size=(3,3), strides=(2,2), padding='same', activation='relu', name='m_conv_1')(inputs)
    x = DepthwiseConv2D(kernel_size=(3,3), padding='same', activation='relu', name = 'm_dc_2')(x)
    x = Conv2D(filters=64, kernel_size=(1,1),padding='same', activation='relu', name = 'm_pc_2')(x)
    x = DepthwiseConv2D(kernel_size=(3,3), padding='same', activation='relu', name = 'm_dc_3')(x)
    x = Conv2D(filters=128, kernel_size=(1,1),padding='same', activation='relu', name = 'm_pc_3')(x)
    x = DepthwiseConv2D(kernel_size=(3,3), strides=(2,2),padding='same', activation='relu', name = 'm_dc_4')(x)
    x = Conv2D(filters=128, kernel_size=(1,1),padding='same', activation='relu', name = 'm_pc_4')(x)
    x = GlobalAveragePooling2D(name='m_gap')(x)
    x = BatchNormalization(name='m_bn_1')(x)
    x = Dense(128, activation='relu', name='m_fc1')(x)
    x = BatchNormalization(name='m_bc_2')(x)
    x = Dense(k, activation='softmax', name='m_output')(x)
    model = Model(inputs, x)
    return model

```

通过 `Summary()` 函数我们可以得到每个网络的每层的参数数量，见图4，左侧是普通卷积，右侧是MobileNet v1。

Layer (type)	Output Shape	Param #
input_44 (InputLayer)	(None, 28, 28, 1)	0
n_conv_1 (Conv2D)	(None, 14, 14, 32)	320
n_conv_2 (Conv2D)	(None, 14, 14, 64)	18496
n_conv_3 (Conv2D)	(None, 14, 14, 128)	73856
n_conv_4 (Conv2D)	(None, 7, 7, 128)	147584
n_gap (GlobalAveragePooling2	(None, 128)	0
n_bn_1 (BatchNormalization)	(None, 128)	512
n_fc1 (Dense)	(None, 128)	16512
n_bc_2 (BatchNormalization)	(None, 128)	512
n_output (Dense)	(None, 10)	1290
Total params: 259,082		
Trainable params: 258,570		
Non-trainable params: 512		

Layer (type)	Output Shape	Param #
input_45 (InputLayer)	(None, 28, 28, 1)	0
m_conv_1 (Conv2D)	(None, 14, 14, 32)	320
m_dc_2 (DepthwiseConv2D)	(None, 14, 14, 32)	320
m_pc_2 (Conv2D)	(None, 14, 14, 64)	2112
m_dc_3 (DepthwiseConv2D)	(None, 14, 14, 64)	640
m_pc_3 (Conv2D)	(None, 14, 14, 128)	8320
m_dc_4 (DepthwiseConv2D)	(None, 7, 7, 128)	1280
m_pc_4 (Conv2D)	(None, 7, 7, 128)	16512
m_gap (GlobalAveragePooling2	(None, 128)	0
m_bn_1 (BatchNormalization)	(None, 128)	512
m_fc1 (Dense)	(None, 128)	16512
m_bc_2 (BatchNormalization)	(None, 128)	512
m_output (Dense)	(None, 10)	1290
Total params: 48,330		
Trainable params: 47,818		
Non-trainable params: 512		

图4: 普通卷积和MobileNet v1网络结构参数汇总

普通卷积的参数总量为259,082，去除未改造的部分剩余的参数数量为239,936。v1的参数总量为48,330去掉未改造的部分剩余参数29,184个。两个的比值为 $\frac{239,936}{29,184} \approx 8.22$ ，符合我们之前的推算。

接着我们在MNIST上跑一下实验，我们在CPU（Intel i7）和GPU（Nvidia 1080Ti）两个环境下运行一下，得到的收敛曲线如图5。在都训练10个epoch的情况下，我们发现MobileNet v1的结果要略差于传统卷积，这点完全可以理解，毕竟MobileNet v1的参数更少。

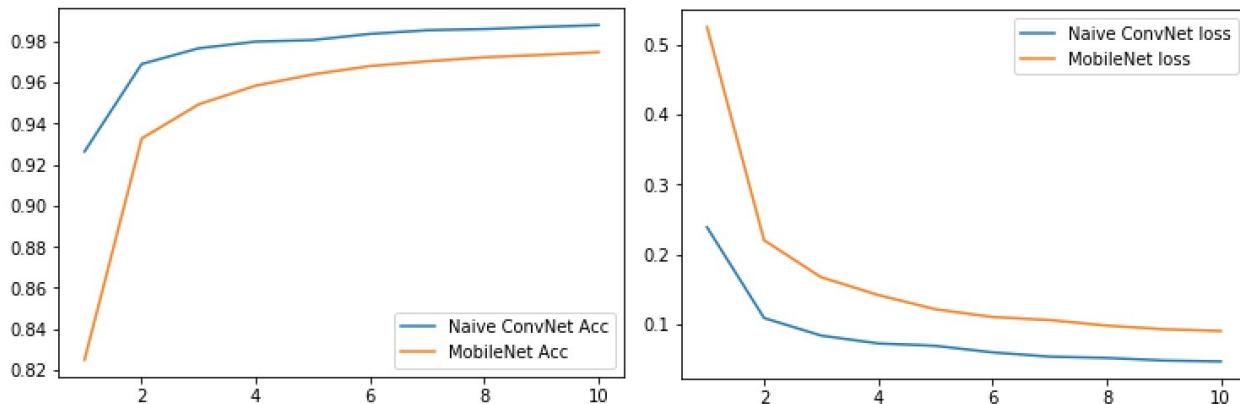


图5: 普通卷积和MobileNet v1在MNIST上的收敛曲线图

对比单个Epoch的训练时间，我们发现了一个奇怪的现象，在CPU上，v1的训练时间约70秒，传统卷积训练时间为140秒，这和我们的直觉是相同的。但是在GPU环境下，传统卷积和v1的训练时间分别为40秒和50秒，v1在GPU上反而更慢了，这是什么原因呢？

问题在于cudnn对传统卷积的并行支持比较完善，而在cudnn7之前的版本并不支持depthwise卷积，现在虽然支持了，其并行性并没有做优化，依旧采用循环的形式遍历每个通道，因此在GPU环境下MobileNet v1反而要慢于传统卷积。所以说，是开源工具慢，并不是**MobileNet v1**的算法慢。

最后，论文中给出了两个超参数 α 和 ρ 分别用于控制每层的Feature Map的数量以及输入图像的尺寸，由于并没有涉及很多特有知识，这里不过多介绍。

2. MobileNet v2 详解

在MobileNet v2中，作者将v1中加入了残差网络，同时分析了v1的几个缺点并针对性的做了改进。v2的改进策略非常简单，但是在编写论文时，缺点分析的时候涉及了流行学习等内容，将优化过程弄得非常难懂。我们在这里简单总结一下v2中给出的问题分析，希望能对论文的阅读有所帮助，对v2的motivation感兴趣的同学推荐阅读论文。

当我们单独去看Feature Map的每个通道的像素的值的时候，其实这些值代表的特征可以映射到一个低维子空间的一个流形区域上。在进行完卷积操作之后往往会接一层激活函数来增加特征的非线性性，一个最常见的激活函数便是ReLU。根据我们在[残差网络](#)中介绍的数据处理不等式(DPI)，ReLU一定会带来信息损耗，而且这种损耗是没有办法恢复的，ReLU的信息损耗是当通道数非常少的时候更为明显。为什么这么说呢？我们看图6中这个例子，其输入是一个表示流形数据的矩阵，和卷机操作类似，他会经过 n 个ReLU的操作得到 n 个通道的Feature Map，然后我们试图通过这 n 个Feature Map还原输入数据，还原的越像说明信息损耗的越少。从图6中我们可以看出，当 n 的值比较小时，ReLU的信息损耗非常严重，当时当 n 的值比较大的时候，输入流形就能还原的很好了。

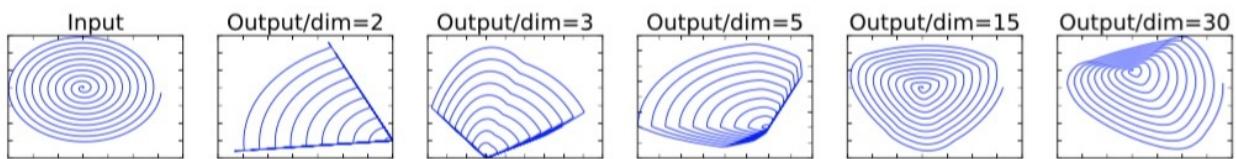


图 6: 使用ReLU激活函数的通道数和信息损耗之间的关系

根据对上面提到的信息损耗问题分析，我们可以有两种解决方案：

1. 既然是ReLU导致的信息损耗，那么我们就将ReLU替换成线性激活函数；
2. 如果比较多的通道数能减少信息损耗，那么我们就使用更多的通道。

2.1 Linear Bottlenecks

我们当然不能把ReLU全部换成线性激活函数，不然网络将会退化为单层神经网络，一个折中方案是在输出Feature Map的通道数较少的时候也就是bottleneck部分使用线性激活函数，其它时候使用ReLU。代码片段如下：

```
def _bottleneck(inputs, nb_filters, t):
    x = Conv2D(filters=nb_filters * t, kernel_size=(1,1), padding='same')(inputs)
    x = Activation(relu6)(x)
    x = DepthwiseConv2D(kernel_size=(3,3), padding='same')(x)
    x = Activation(relu6)(x)
    x = Conv2D(filters=nb_filters, kernel_size=(1,1), padding='same')(x)
    # do not use activation function
    if not K.get_variable_shape(inputs)[3] == nb_filters:
        inputs = Conv2D(filters=nb_filters, kernel_size=(1,1), padding='same')(inputs)
    outputs = add([x, inputs])
    return outputs
```

这里使用了MobileNet中介绍的ReLU6激活函数，它是对ReLU在6上的截断，数学形式为：

$$ReLU(6) = \min(\max(0, x), 6)$$

图7便是结合了残差网络和线性激活函数的MobileNet v2的一个block，最右侧是v1。

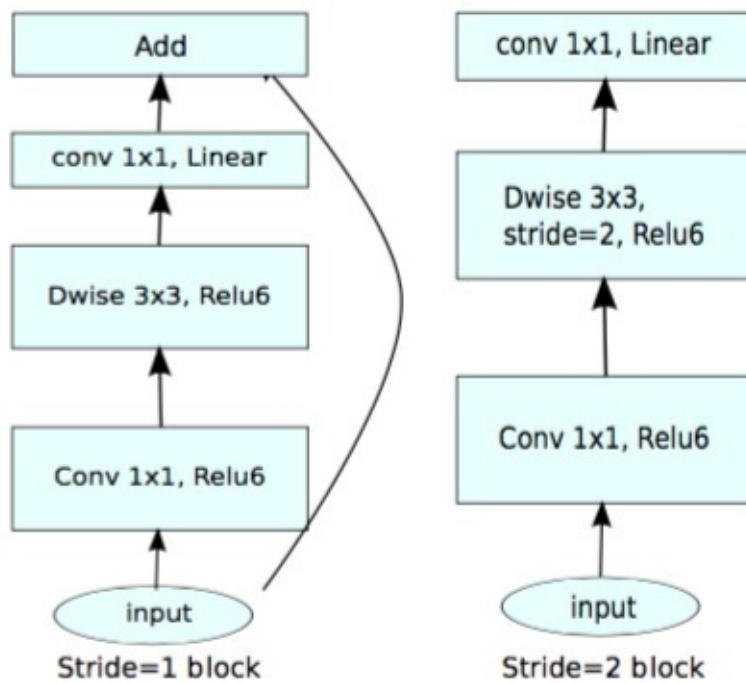


图7: v2的Linear Bottleneck和v1的Depthwise Separable卷积对比

2.2 Inverted Residual

当激活函数使用ReLU时，我们可以通过增加通道数来减少信息的损耗，使用参数 t 来控制，该层的通道数是输入Feature Map的 t 倍。传统的残差块的 t 一般取小于1的小数，常见的取值为0.1，而在v2中这个值一般是介于5 – 10之间的数，在作者的实验中， $t = 6$ 。考虑到残差网络和v2的 t 的不同取值范围，他们分别形成了锥子形（两头小中间大）和沙漏形（两头大中间小）的结构，如图8所示，其中斜线Feature Map表示使用的是线性激活函数。这也就是为什么这种形式的卷积block被叫做Inverted Residual block，因为他把short-cut转移到到了bottleneck层。

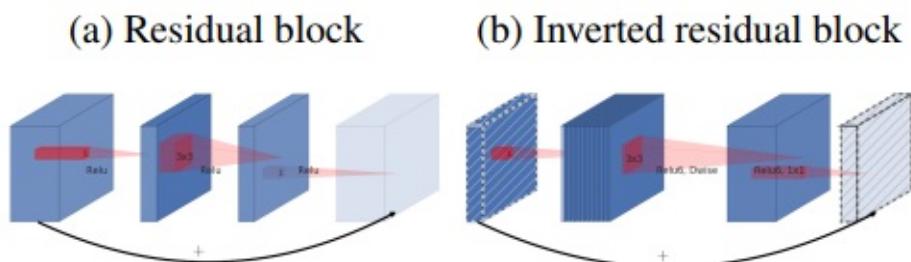


图8: 残差网络的的Residual block和v2的Inverted Residual block卷积对比

2.3 MobileNet v2

综上我们可以得到MobileNet v2的一个block的详细参数，如图9所示，其中 s 代表步长：

Input	Operator	Output
$h \times w \times k$	1x1 conv2d, ReLU6	$h \times w \times (tk)$
$h \times w \times tk$	3x3 dwise $s=s$, ReLU6	$\frac{h}{s} \times \frac{w}{s} \times (tk)$
$\frac{h}{s} \times \frac{w}{s} \times tk$	linear 1x1 conv2d	$\frac{h}{s} \times \frac{w}{s} \times k'$

图9: MobileNetv2 block的超参数

MobileNet v2的实现可以通过堆叠bottleneck的形式实现，如下面代码片段

```
def MobileNetV2_relu(input_shape, k):
    inputs = Input(shape = input_shape)
    x = Conv2D(filters=32, kernel_size=(3,3), padding='same')(inputs)
    x = _bottleneck_relu(x, 8, 6)
    x = MaxPooling2D((2,2))(x)
    x = _bottleneck_relu(x, 16, 6)
    x = _bottleneck_relu(x, 16, 6)
    x = MaxPooling2D((2,2))(x)
    x = _bottleneck_relu(x, 32, 6)
    x = GlobalAveragePooling2D()(x)
    x = Dense(128, activation='relu')(x)
    outputs = Dense(k, activation='softmax')(x)
    model = Model(inputs, outputs)
    return model
```

3. 总结

在这篇文章中，我们介绍了两个版本的MobileNet，它们和传统卷积的对比如图10。

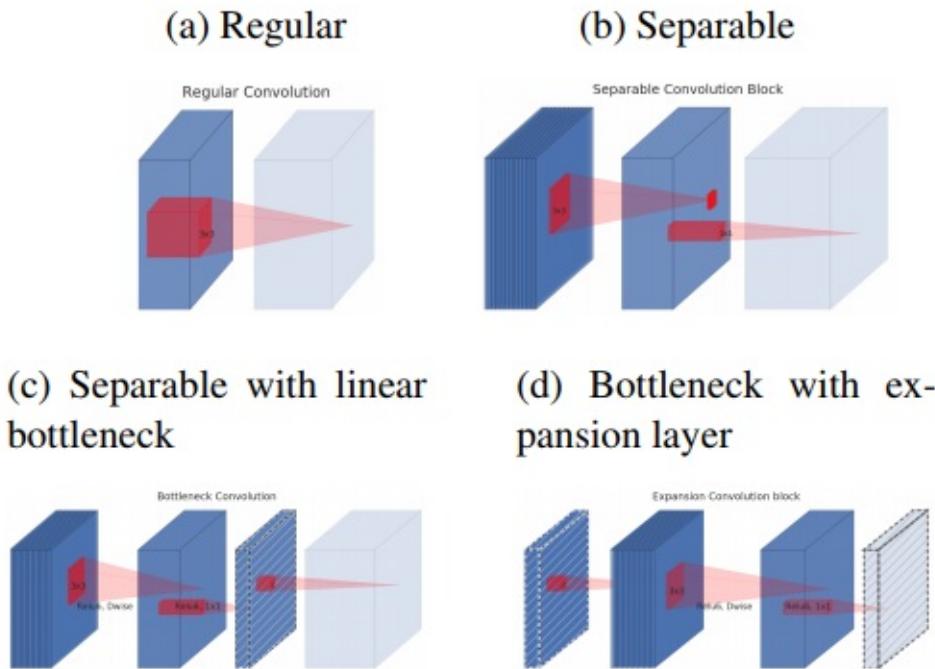


图 10: 普通卷积(a) vs MobileNet v1(b) vs MobileNet v2(c, d)

如图(b)所示，MobileNet v1最主要的贡献是使用了Depthwise Separable Convolution，它又可以拆分成Depthwise卷积和Pointwise卷积。MobileNet v2主要是将残差网络和Depthwise Separable卷积进行了结合。通过分析单通道的流形特征对残差块进行了改进，包括对中间层的扩展(d)以及bottleneck层的线性激活(c)。Depthwise Separable Convolution的分离式设计直接将模型压缩了8倍左右，但是精度并没有损失非常严重，这一点还是非常震撼的。

Depthwise Separable卷积的设计非常精彩但遗憾的是目前cudnn对其的支持并不好，导致在使用GPU训练网络过程中我们无法从算法中获益，但是使用串行CPU并没有这个问题，这也给了MobileNet很大的市场空间，尤其是在嵌入式平台。

最后，不得不承认v2的论文的一系列证明非常精彩，虽然没有这些证明我们也能明白v2的工作原理，但是这些证明过程还是非常值得仔细品鉴的，尤其是对于从事科研方向的工作人员。

Xception: Deep Learning with Depthwise Separable Convolutions

tags: Inception, Xception

前言

深度可分离卷积（Depthwise Separable Convolution）率先是由 Laurent Sifre 载气博士论文《Rigid-Motion Scattering For Image Classification》[91] 中提出。经典的 MobileNet [109] 系列算法便是采用深度可分离卷积作为其核心结构。

这篇文章主要从 Inception [121] 的角度出发，探讨了 Inception 和深度可分离卷积的关系，从一个全新的角度解释了深度可分离卷积。再结合 sto 的 残差网络 [118]，一个新的架构 Xception [119] 应运而生。Xception 取义自 Extreme Inception，即 Xception 是一种极端的 Inception，下面我们来看看它是怎样的一种极端法。

1. Inception 回顾

Inception 的核心思想是将 channel 分成若干个不同感受野大小的通道，除了能获得不同的感受野，Inception 还能大幅的降低参数数量。我们看图 1 中一个简单版本的 Inception 模型

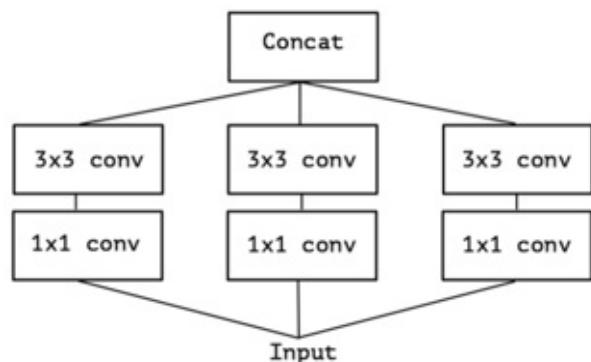


图 1：简单的 Inception

对于一个输入的 Feature Map，首先通过三组 1×1 卷积得到三组 Feature Map，它和先使用一组 1×1 卷积得到 Feature Map，再将这组 Feature Map 分成三组是完全等价的（图 2）。假设图 1 中 1×1 卷积核的个数都是 k_1 ， 3×3 的卷积核的个数都是 k_2 ，输入 Feature Map 的通道数为 m ，那么这个简单版本的参数个数为

$$m \times k_1 + 3 \times 3 \times 3 \times \frac{k_1}{3} \times \frac{k_2}{3} = m \times k_1 + 3 \times k_1 \times k_2$$

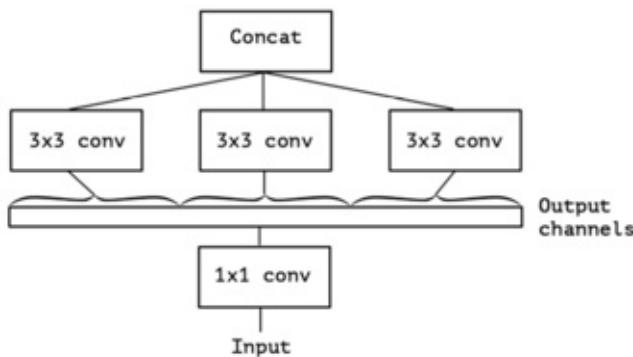


图2：简单Inception的等价形式

对比相同通道数，但是没有分组的普通卷积，普通卷积的参数数量为：

$$m \times k_1 + 3 \times 3 \times k_1 \times k_2$$

参数数量约为Inception的三倍。

2. Xception

如果Inception是将 3×3 卷积分成3组，那么考虑一种极度的情况，我们如果将Inception的 1×1 得到的 k_1 个通道的Feature Map完全分开呢？也就是使用 k_1 个不同的卷积分别在每个通道上进行卷积，它的参数数量是：

$$m \times k_1 + k_1 \times 3 \times 3$$

更多时候我们希望两组卷积的输出Feature Map相同，这里我们将Inception的 1×1 卷积的通道数设为 k_2 ，即参数数量为

$$m \times k_2 + k_2 \times 3 \times 3$$

它的参数数量是普通卷积的 $\frac{1}{k_1}$ ，我们把这种形式的Inception叫做Extreme Inception，如图3所示。

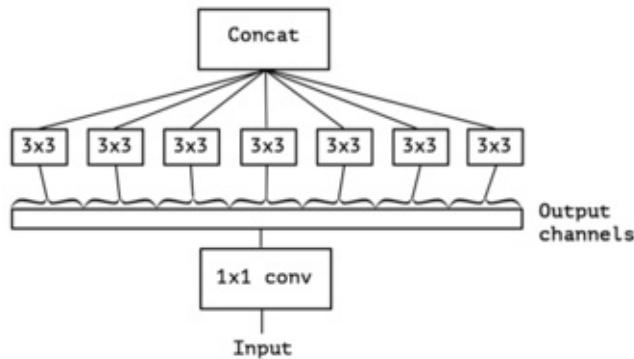


图3 : Extreme Inception

在搭建GoogLeNet网络时，我们一般采用堆叠Inception的形式，同理在搭建由Extreme Inception构成的网络的时候也是采用堆叠的方式，论文中将这种形式的网络结构叫做Xception。

如果你看过深度可分离卷积的话你就会发现它和Xception几乎是等价的，区别之一就是先计算Pointwise卷积和先计算Depthwise的卷积的区别。

在MobileNet v2[115]中，我们指出bottleneck的最后一层 1×1 卷积核为线性激活时能够更有助于减少信息损耗，这也就是Xception和深度可分离卷积（准确说是MobileNet v2）的第二个不同点。

结合残差结构，一个完整的模型见图4，其实现Keras官方已经开源。

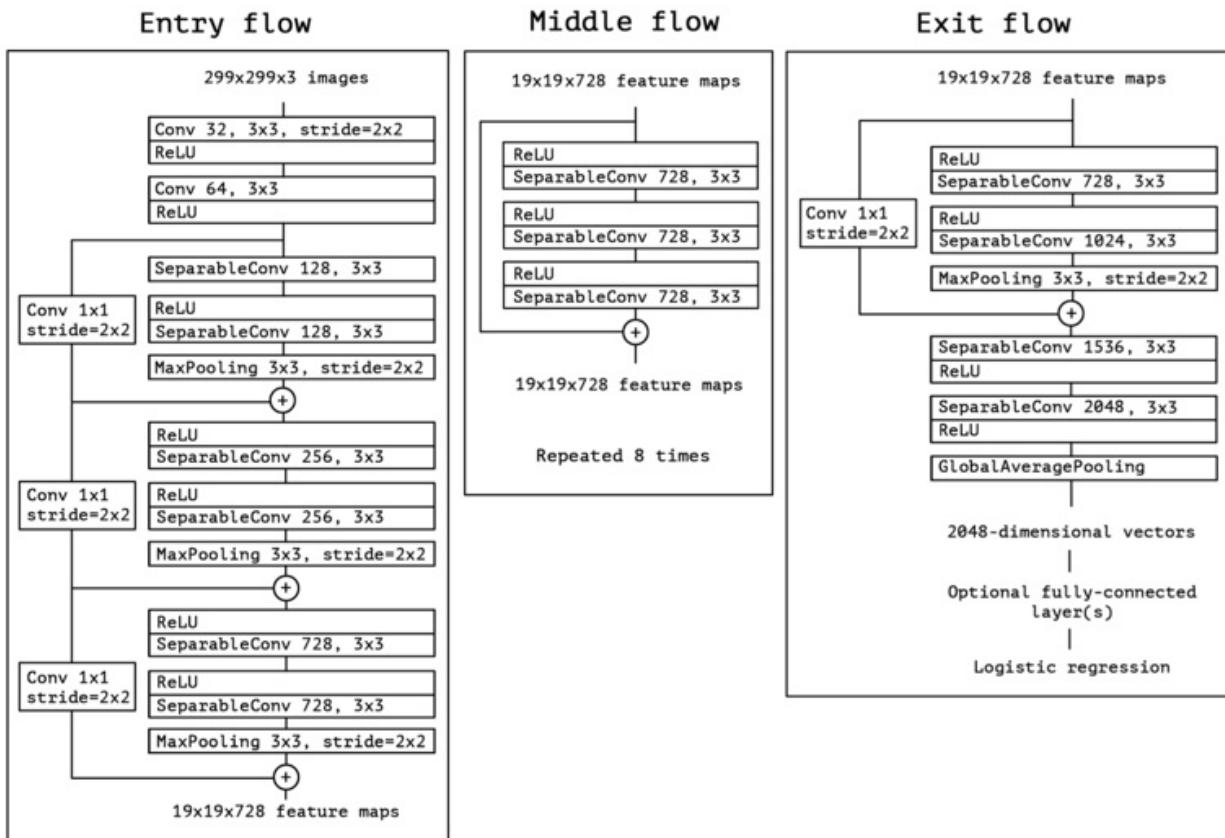


图 4 : Xception

上图中要注意的几点：

1. Keras 的 `SeparableConv` 函数是由 3×3 的 depthwise 卷积和 1×1 的 pointwise 卷积组成，因此可用于升维和降维；
2. 图 5 中的 \oplus 是 add 操作，即两个 Feature Map 进行单位加。

3. 总结

Xception 的结构和 MobileNet 非常像，两个算法的提出时间近似，不存在谁抄袭谁的问题。他们从不同的角度揭示了深度可分离卷积的强大作用，MobileNet 的思路是通过将普通 3×3 卷积拆分的形式来减少参数数量，而 Xception 是通过对 Inception 的充分解耦来完成的。

Aggregated Residual Transformations for Deep Neural Networks

tags: ResNeXt, ResNet, Inception

前言

在这篇文章中，作者介绍了ResNeXt[108]。ResNeXt是ResNet[118]和Inception[121]的结合体，不同于Inception v4[112]的是，ResNeXt不需要人工设计复杂的Inception结构细节，而是每一个分支都采用相同的拓扑结构。ResNeXt的本质是分组卷积（Group Convolution）[109]，通过变量基数（Cardinality）来控制组的数量。组卷积是普通卷积和深度可分离卷积的一个折中方案，即每个分支产生的Feature Map的通道数为 n ($n > 1$)。

1. 详解

1.1 从全连接讲起

给定一个 D 维的输入数据 $\mathbf{x} = [x_1, x_2, \dots, x_d]$ ，其输入权值为 $\mathbf{w} = [w_1, w_2, \dots, w_n]$ ，一个没有偏置的线性激活神经元为：

$$\sum_{i=1}^D w_i x_i$$

它的结构如图1所示。

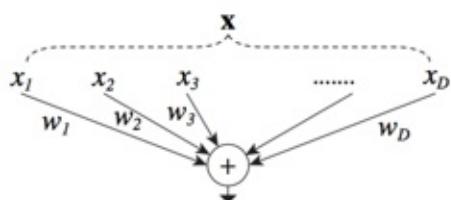


图1：全连接的split-transform-merge结构

这是一个最简单的“split-transform-merge”结构，具体的讲图1可以拆分成3步：

1. Split：将数据 \mathbf{x} split成 D 个特征；
2. Transform：每个特征经过一个线性变换；

3. Merge：通过单位加合成最后的输出。

1.2 简化Inception

Inception是一个非常明显的“split-transform-merge”结构，作者认为Inception不同分支的不同拓扑结构的特征有非常刻意的人工雕琢的痕迹，而往往调整Inception的内部结构对应着大量的超参数，这些超参数调整起来是非常困难的。

所以作者的思想是每个结构使用相同的拓扑结构，那么这时候的Inception（这里简称简化Inception）表示为

$$\mathcal{F} = \sum_{i=1}^C \mathcal{T}_i(\mathbf{x})$$

其中 C 是简Inception的基数(Cardinality)， \mathcal{T}_i 是任意的变换，例如一系列的卷积操作等。图2便是一个简化Inception，其 \mathcal{T} 是由连续的卷积组成（ $1 \times 1 \rightarrow 3 \times 3 \rightarrow 1 \times 1$ ）。

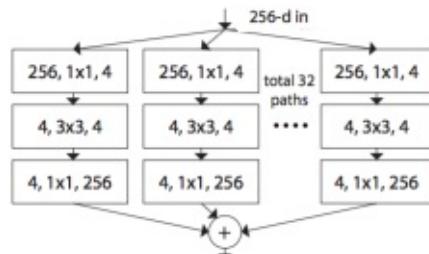


图2：简单Inception的split-transform-merge结构

1.3 ResNeXt

结合强大的残差网络，我们得到的便是完整的ResNeXt，也就是在简化Inception中添加一条short-cut，表示为：

$$\mathbf{y} = \mathbf{x} + \sum_{i=1}^C \mathcal{T}_i(\mathbf{x})$$

如图3所示：

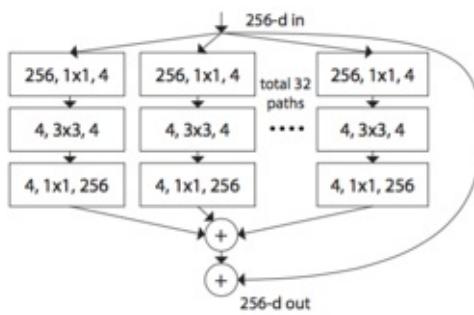


图3：ResNeXt结构

到这里我们发现ResNeXt和Inception v4是非常像的。不同之处有两点：

1. ResNeXt的分支的拓扑结构是相同的，Inception V4需要人工设计；
2. ResNeXt是先进行 1×1 卷积然后执行单位加，Inception V4是先拼接再执行 1×1 卷积，如图4所示。

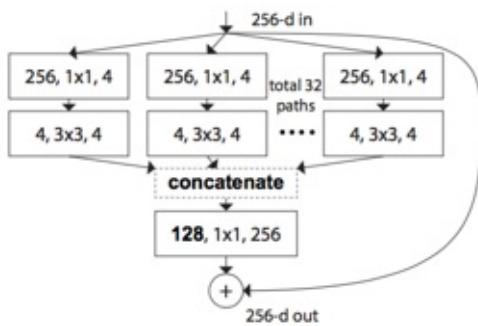


图4：Inception V4拼接在前的结构

1.4 分组卷积

分组卷积的雏形更早要追溯到2012年深度学习鼻祖文章AlexNet [123]。受限于当时硬件的限制，作者不得不将卷积操作拆分到两台GPU上运行，这两台GPU的参数是不共享的。

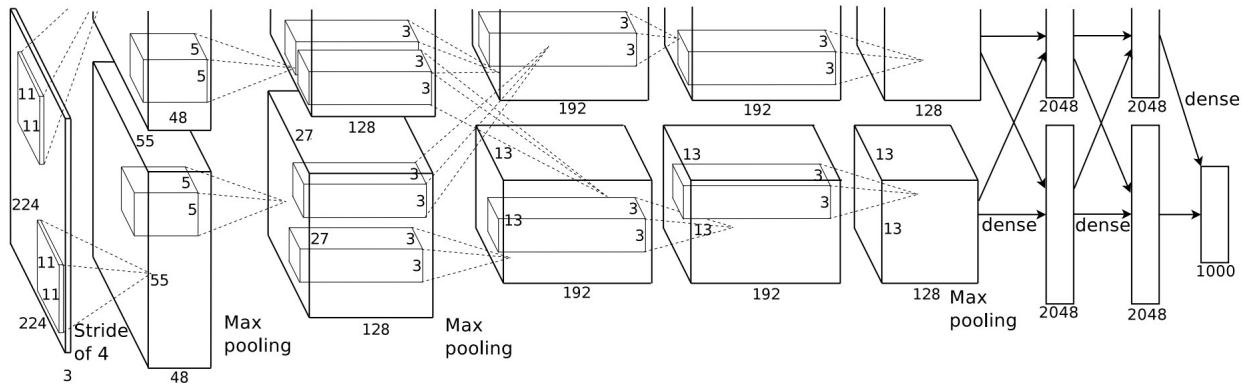


图5：AlexNet

分组卷积是介于普通卷积核深度可分离卷积的一种折中方案，不是彻底的将每个channel都要单独赋予一个独立的卷积核也不是整个Feature Map使用同一个卷积核。

除了Inception v4，分组卷积的第三种变形是将开始的 1×1 卷积也合并到一起，如图6。

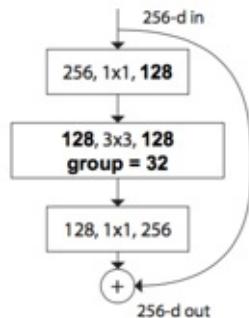


图6：分组卷积的第三种形式

总结

ResNeXt提出了一种介于普通卷积核深度可分离卷积的这种策略：分组卷积，他通过控制分组的数量（基数）来达到两种策略的平衡。分组卷积的思想是源自Inception，不同于Inception的需要人工设计每个分支，ResNeXt的每个分支的拓扑结构是相同的。最后再结合残差网络，得到的便是最终的ResNeXt。

从上面的分析中我们可以看书ResNeXt的结构非常简单，但是其在ImageNet上取得了由于相同框架的残差网络，也算是Inception直接助攻了一把吧。

ResNeXt确实比Inception V4的超参数更少，但是他直接废除了Inception的囊括不同感受野的特性仿佛不是很合理，在更多的环境中我们发现Inception V4的效果是优于ResNeXt的。类似结构的ResNeXt的运行速度应该是优于Inception V4的，因为ResNeXt的相同拓扑结构的分支

的设计是更符合**GPU**的硬件设计原则。

ShuffNet v1 and ShuffleNet v2

tags: ShuffNet v1, ShuffleNet v2

前言

在[ResNeXt\[108\]](#)的文章中，分组卷积作为传统卷积核深度可分离卷积的一种折中方案被采用。这时大量的对于整个Feature Map的Pointwise卷积成为了ResNeXt的性能瓶颈。一种更高效的策略是在组内进行Pointwise卷积，但是这种组内Pointwise卷积的形式不利于通道之间的信息流通，为了解决这个问题，ShuffleNet v1[\[107\]](#)中提出了通道洗牌（channel shuffle）操作。

在ShuffleNet v2的文章中作者指出现在普遍采用的FLOPs评估模型性能是非常不合理的，因为一批样本的训练时间除了看FLOPs，还有很多过程需要消耗时间，例如文件IO，内存读取，GPU执行效率等等。作者从内存消耗成本，GPU并行性两个方向分析了模型可能带来的非FLOPs的行动损耗，进而设计了更加高效的ShuffleNet v2[\[106\]](#)。ShuffleNet v2的架构和[DenseNet\[117\]](#)有异曲同工之妙，而且其速度和精度都要优于DenseNet。

1. ShuffleNet v1

1.1 Channel Shuffle

通道洗牌是介于整个通道的Pointwise卷积和组内Pointwise卷积的一种折中方案，传统策略是在整个Feature Map上执行 1×1 卷积。假设一个传统的深度可分离卷积由一个 3×3 的Depthwise卷积和一个 1×1 的Pointwise卷积组成。其中输入Feature Map的尺寸为

$h \times w \times c_1$ ，输出Feature Map的尺寸为 $h \times w \times c_2$ ， 1×1 处的FLOPs为

$$F = 9 \cdot h \cdot w + h \cdot w \cdot c_1 \cdot c_2$$

一般情况下 $c_1 \cdot c_2$ 是远大于9的，也就是说深度可分离卷积的性能瓶颈主要在Pointwise卷积上。

为了解决这个问题，ShuffleNet v1中提出了仅在分组内进行Pointwise卷积，对于一个分成了 g 个组的分组卷积，其FLOPs为：

$$F = 9 \cdot h \cdot w + \frac{h \cdot w \cdot c_1 \cdot c_2}{g}$$

从上面式子中我们可以看出组内 Pointwise 卷积可以非常有效的缓解性能瓶颈问题。然而这个策略的一个非常严重的问题是卷积直接的信息沟通不畅，网络趋近于一个由多个结构类似的网络构成的模型集成，精度大打折扣，如图1.(a)所示。

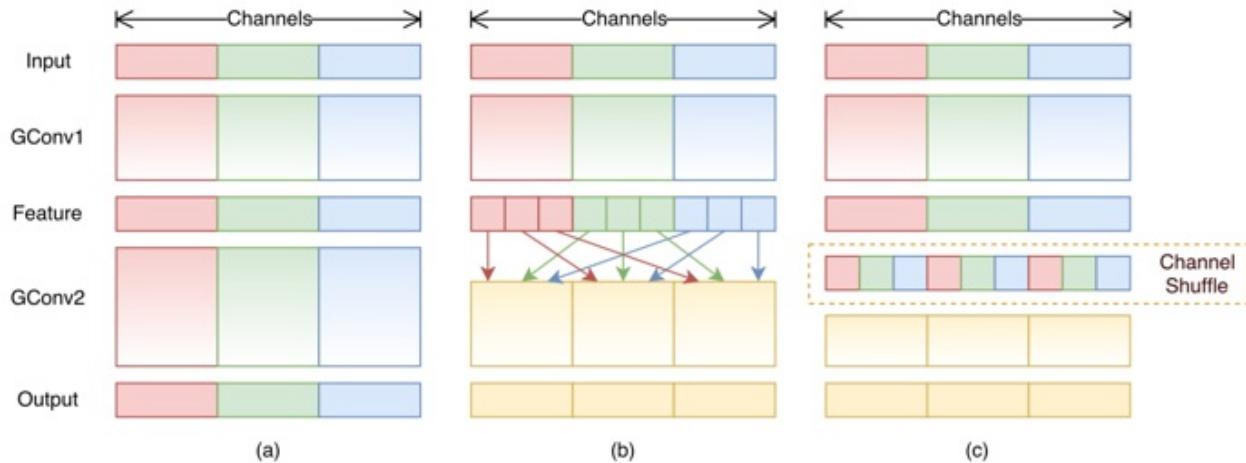


图1：分组 Pointwise 卷积 vs 通道洗牌

为了解决通道之间的沟通问题，ShuffleNet v1提出了其最核心的操作：通道洗牌（Channel Shuffle）。假设分组Feature Map的尺寸为 $w \times h \times c_1$ ，把 $c_1 = g \times n$ ，其中 g 表示分组的组数。Channel Shuffle的操作细节如下：

1. 将Feature Map展开成 $g \times n \times w \times h$ 的四维矩阵（为了简单理解，我们把 $w \times h$ 降到一维，表示为 s ）；
2. 沿着尺寸为 $g \times n \times s$ 的矩阵的 g 轴和 n 轴进行转置；
3. 将 g 轴和 n 轴进行平铺后得到洗牌之后的Feature Map；
4. 进行组内 1×1 卷积。

shuffle的结果如图1.(c)所示，具体操作细节示意图见图2，Keras实现见代码片段1。

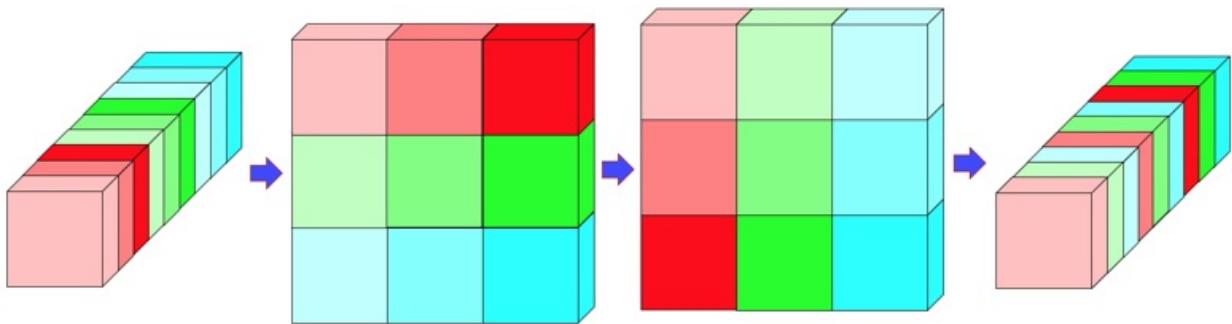


图2：通道洗牌过程详解

```

def channel_shuffle(x, groups):
    """
    Parameters
        x: Input tensor of with `channels_last` data format
        groups: int number of groups per channel
    Returns
        channel shuffled output tensor
    Examples
        Example for a 1D Array with 3 groups
        >>> d = np.array([0,1,2,3,4,5,6,7,8])
        >>> x = np.reshape(d, (3,3))
        >>> x = np.transpose(x, [1,0])
        >>> x = np.reshape(x, (9,))
        '[0 1 2 3 4 5 6 7 8] --> [0 3 6 1 4 7 2 5 8]'

    height, width, in_channels = x.shape.as_list()[1:]
    channels_per_group = in_channels // groups
    x = K.reshape(x, [-1, height, width, groups, channels_per_group])
    x = K.permute_dimensions(x, (0, 1, 2, 4, 3))  # transpose
    x = K.reshape(x, [-1, height, width, in_channels])
    return x

```

从代码中我们也可以看出，channel shuffle的操作是步步可微分的，因此可以嵌入到卷积网络中。

1.2 ShuffleNet v1 单元

图3.(a)是一个普通的带有残差结构的深度可分离卷积，例如，MobileNet[109], Xception[119]。ShuffleNet v1的结构如图3.(b)，3.(c)。其中3.(b)不需要降采样，3.(c)是需要降采样的情况。

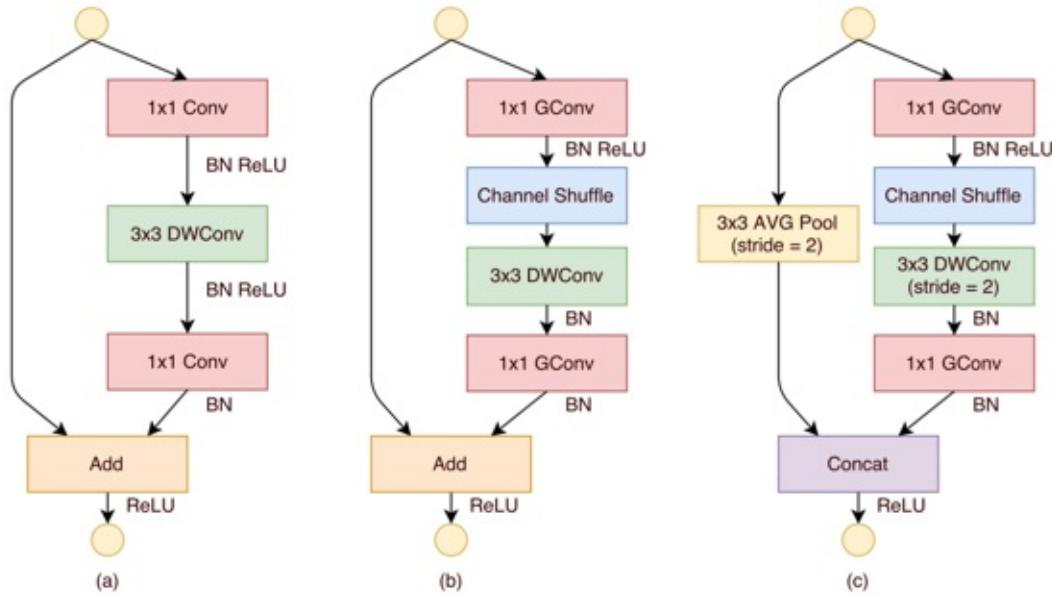


图3：(a) MobileNet, (b) ShuffleNet v1, (c) ShuffleNet v1降采样情况

3.(b)和3.(c)已经介绍了ShuffleNet v1全部的实现细节，我们仔细分析之：

1. 上下两个红色部分的 1×1 卷积替换为 1×1 的分组卷积，分组 g 一般不会很大，论文中的几个值分别是1, 2, 3, 4, 8。当 $g = 1$ 时，ShuffleNet v1退化为Xception。 g 的值确保能够被通道数整除，保证reshape操作的有效执行。
2. 在第一个 1×1 卷积之后添加一个1.1节介绍的Channel Shuffle操作。
3. 如图3.(c)中需要降采样的情况，左侧shortcut部分使用的是步长为2的 3×3 平均池化，右侧使用的是步长为2的 3×3 的Depthwise卷积。
4. 去掉了 3×3 卷积之后的ReLU激活，目的是为了减少ReLU激活造成的信息损耗，具体原因见MobileNet v2[115]。
5. 如果进行了降采样，为了保证参数数量不骤减，往往需要加倍通道数量。所以在3.(c)中使用的是拼接(Concat)操作用于加倍通道数，而3.(b)中则是一个单位加。

最后基于ShuffleNet v1单元，我们计算一下ResNet, ResNeXt, ShuffleNet v1的FLOPs，即执行一个单元需要的计算量。Channel Shuffle处的操作数非常少，这里可以忽略不计。假设输入Feature Map的尺寸为 $w \times h \times c$ ，bottleneck的通道数为 m 。

1. ResNet：

$$F_{\text{ResNet}} = hwcm + 3 \cdot 3 \cdot hwmm + hwcm = hw(2cm + 9m^2)$$

2. ResNeXt：

$9m$

$$F_{\text{ResNeXt}} = hwcm + 3 \cdot 3 \cdot hw \frac{m}{g} \frac{m}{g} \cdot g + hwcm = hw(2cm + \frac{9m^2}{g})$$

3. ShuffleNet v1 :

$$F_{\text{ShuffleNet v1}} = hw \frac{c}{g} \frac{m}{g} \cdot g + 3 \cdot 3hwm + hw \frac{c}{g} \frac{m}{g} \cdot g = hw(\frac{2cm}{g} + 9m)$$

我们可以非常容易得到它们的FLOPs的关系：

$$F_{\text{ResNet}} < F_{\text{ResNeXt}} < F_{\text{ShuffleNet v1}}$$

1.3 ShuffleNet v1 网络

ShuffleNet v1完整网络的搭建可以通过堆叠ShuffleNet v1 单元的形式构成，这里不再赘述。具体细节请查看已经开源的ShuffleNet v1的源码。

2. ShuffleNet v2

2.1 模型性能的评估指标

在上面的文章中我们统一使用FLOPs作为评估一个模型的性能指标，但是在ShuffleNet v2的论文中作者指出这个指标是间接的，因为一个模型实际的运行时间除了要把计算操作算进去之外，还有例如内存读写，GPU并行性，文件IO等也应该考虑进去。最直接的方案还应该回归到最原始的策略，即直接在同一个硬件上观察每个模型的运行时间。如图4所示，在整个模型的计算周期中，FLOPs耗时仅占50%左右，如果我们能优化另外50%，我们就能够在不损失计算量的前提下进一步提高模型的效率。

在ShuffleNet v2中，作者从内存访问代价（Memory Access Cost，MAC）和GPU并行性的方向分析了网络应该怎么设计才能进一步减少运行时间，直接的提高模型的效率。

2.2 高效模型的设计准则

G1) : 当输入通道数和输出通道数相同时，MAC最小

假设一个卷积操作的输入Feature Map的尺寸是 $w \times h \times c_1$ ，输出Feature Map的尺寸为

$w \times h \times c_2$ 。卷积操作的FLOPs为 $F = hwc_1c_2$ 。在计算这个卷积的过程中，输入Feature Map占用的内存大小是 hwc_1 ，输出Feature Map占用的内存是 hwc_2 ，卷积核占用的内存是 c_1c_2 。总计：

$$\begin{aligned}
MAC &= hw(c_1 + c_2) + c_1 c_2 \\
&= \sqrt{(hw(c_1 + c_2))^2} + \frac{B}{hw} \\
&= \sqrt{(hw)^2 \cdot (c_1 + c_2)^2} + \frac{B}{hw} \\
&\geq \sqrt{(hw)^2 \cdot 4c_1 c_2} + \frac{B}{hw} \\
&= s\sqrt{hw \cdot (hwc_1 c_2)} + \frac{B}{hw} \\
&= 2\sqrt{hwB} + \frac{B}{hw}
\end{aligned}$$

当 $c_1 = c_2$ 时上式取等号。也就是说当FLOPs确定的时候， $c_1 = c_2$ 时模型的运行效率最高，因为此时的MAC最小。

G2) : MAC与分组数量 g 成正比

在分组卷积中，FLOPs为 $F = hw \frac{c_1 c_2}{g} g = \frac{hwc_1 c_2}{g}$ ，其MAC的计算方式为：

$$\begin{aligned}
MAC &= hw(c_1 + c_2) + \frac{c_1}{g} \frac{c_2}{g} g \\
&= hw(c_1 + c_2) + \frac{c_1 c_2}{g} \\
&= Bg\left(\frac{1}{c_1} + \frac{1}{c_2}\right) + \frac{B}{hw}
\end{aligned}$$

根据G2，我们在设计网络时 g 的值不应过大。

G3) : 网络的分支数量降低并行能力

分支数量比较多的典型网络是Inception，NasNet等。作者证明这个一组准则是设计了一组对照试验：如图4所示，通过控制卷积的通道数来使5组对照试验的FLOPs相同，通过实验我们发现它们按效率从高到低排列依次是 (a) > (b) > (d) > (c) > (e)。

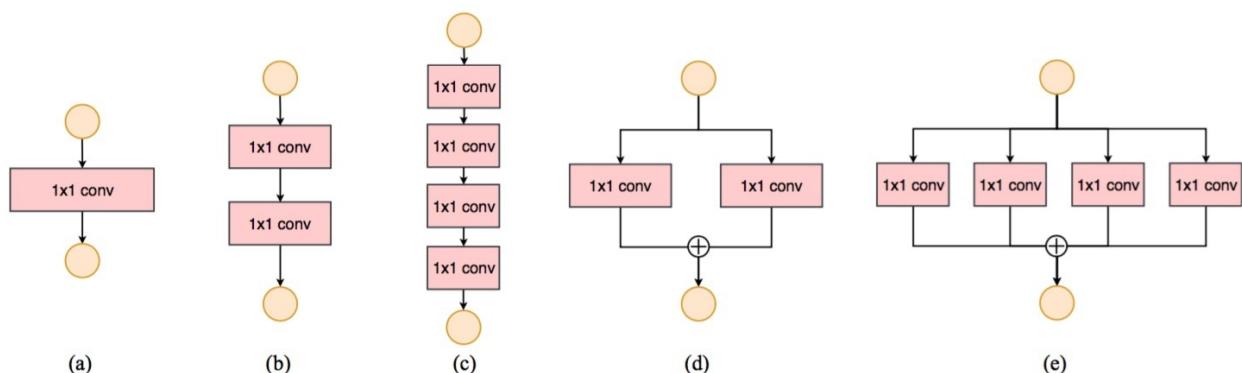


图4：网络分支对比试验样本示意图

造成这种现象的原因是更多的分支需要更多的卷积核加载和同步操作。

G4) : Element-wise操作是非常耗时的

我们在计算FLOPs时往往只考虑卷积中的乘法操作，但是一些Element-wise操作（例如ReLU激活，偏置，单位加等）往往被忽略掉。作者指出这些Element-wise操作看似数量很少，但它对模型的速度影响非常大。尤其是深度可分离卷积这种MAC/FLOPs比值较高的算法。图5中统计了ShuffleNet v1和MobileNet v2中各个操作在GPU和ARM上的消耗时间占比。

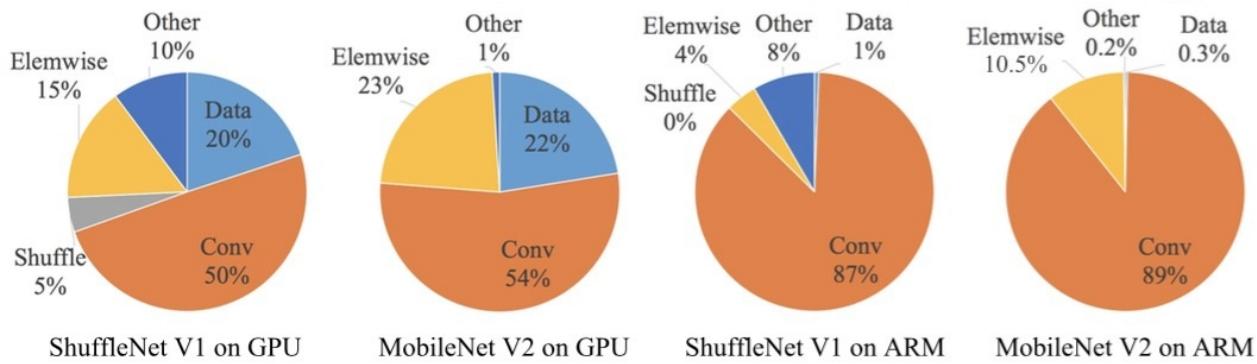


图5：模型训练时间拆分示意图

总结一下，在设计高性能网络时，我们要尽可能做到：

1. G1). 使用输入通道和输出通道相同的卷积操作；
2. G2). 谨慎使用分组卷积；
3. G3). 减少网络分支数；
4. G4). 减少element-wise操作。

例如在ShuffleNet v1中使用的分组卷积是违背G2的，而每个ShuffleNet v1单元使用了bottleneck结构是违背G1的。MobileNet v2中的大量分支是违背G3的，在Depthwise处使用ReLU6激活是违背G4的。

从它的对比实验中我们可以看到虽然ShuffleNet v2要比和它FLOPs数量近似的模型的速度要快。

2.3 ShuffleNet v2结构

图6中，(a)，(b)是刚刚介绍的ShuffleNet v1，(c)，(d)是这里要介绍的ShuffleNet v2。



图6：(a) ShuffleNet v1，(b)ShuffleNet v1 降采样，(c)ShuffleNet v2，(d)ShuffleNet v2 降采样

仔细观察(c), (d)对网络的改进我们发现了以下几点：

1. 在(c)中ShuffleNet v2使用了一个通道分割（Channel Split）操作。这个操作非常简单，即将 c 个输入Feature分成 $c - c'$ 和 c' 两组，一般情况下 $c' = \frac{c}{2}$ 。这种设计是为了尽量控制分支数，为了满足G3。
2. 在分割之后的两个分支，左侧是一个直接映射，右侧是一个输入通道数和输出通道数均相同的深度可分离卷积，为了满足G1。
3. 在右侧的卷积中， 1×1 卷积并没有使用分组卷积，为了满足G2。
4. 最后在合并的时候均是使用拼接操作，为了满足G4。
5. 在堆叠ShuffleNet v2的时候，通道拼接，通道洗牌和通道分割可以合并成1个element-wise操作，也是为了满足G4。

最后当需要降采样的时候我们通过不进行通道分割的方式达到通道数量的加倍，如图6.(d)，非常简单。

2.4 ShuffleNet v2和DenseNet

ShuffleNet v2能够得到非常高的精度是因为它和DenseNet有着思想上非常一致的结构：强壮的特征重用（Feature Reuse）。在DenseNet中，作者大量使用的拼接操作直接将上一层的Feature Map原汁原味的传到下一个乃至下几个模块。从6.(c)中我们也可以看处，左侧的直接映射和DenseNet的特征重用是非常相似的。

不同于DenseNet的整个Feature Map的直接映射，ShuffleNet v2只映射了一半。恰恰是这一点不同，是ShuffleNet v2有了和DenseNet的升级版CondenseNet[102]相同的思想。在CondenseNet中，作者通过可视化DenseNet的特征重用和Feature Map的距离关系发现距离越近的**Feature Map**之间的特征重用越重要。ShuffleNet v2中第*i*个和第*i+j*个Feature Map的重用特征的数量是 $(\frac{1}{2})^j c$ 。也就是距离越远，重用的特征越少。

总结

截止本文截止，ShuffleNet算是将轻量级网络推上了新的巅峰，两个版本都有其独到的地方。

ShuffleNet v1中提出的通道洗牌（Channel Shuffle）操作非常具有创新点，其对于解决分组卷积中通道通信困难上非常简单高效。

ShuffleNet v2分析了模型性能更直接的指标：运行时间。根据对运行时间的拆分，通过数学证明或是实验证明或是理论分析等方法提出了设计高效模型的四条准则，并根据这四条准则设计了ShuffleNet v2。ShuffleNet v2中的通道分割也是创新点满满。通过仔细分析通道分割，我们发现了它和DenseNet有异曲同工之妙，在这里轻量模型和高精度模型交汇在了一起。

ShuffleNet v2的证明和实验以及最后网络结构非常精彩，整篇论文读完给人一种畅快淋漓的感觉，建议读者们读完本文后拿出论文通读一遍，你一定会收获很多。

CondenseNet: An Efficient DenseNet using Learned Group Convolutions

tags: DenseNet, CondenseNet

前言

CondenseNet[102]是黄高团队对其DenseNet[117]的升级。DenseNet的密集连接其实是存在冗余的，其最大的影响便是影响网络的效率。首先，为了降低DenseNet的冗余问题，CondenseNet提出了在训练的过程中对不重要的权值进行剪枝，即学习一个稀疏的网络。但是测试的整个过程就是一个简单的卷积，因为网络已经在训练的时候优化完毕。其次，为了进一步提升效率，CondenseNet在 1×1 卷积的时候使用了分组卷积，分组卷积在AlexNet中首先应用于双GPU架构，并在ResNeXt[108]中作为性能提升的策略首次被提出。最后，CondenseNet中指出临近的特征重用更重要，因此采用了指数增长的增长率（Growth Rate），并在DenseNet的block之间也添加了short-cut。

DenseNet，CondenseNet的训练和测试阶段的示意图如图1。其中的细节我们会在后面的部分详细解析。

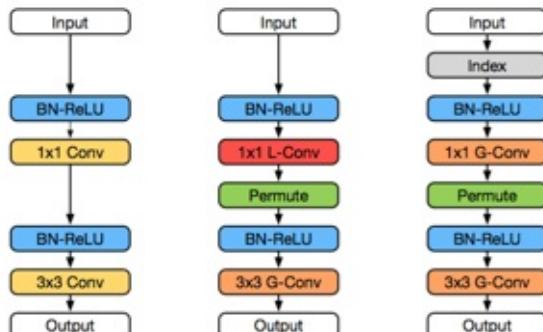


图1：CondenseNet概览，(左)：DenseNet；(中)：CondenseNet训练；(右)：CondenseNet测试

1. CondenseNet详解

1.1 分组卷积的问题

在ShuffleNet[107]中我们指出分组卷积存在通道之间的信息沟通不畅以及特征多样性不足的问题。CondenseNet提出的解决策略是在训练的过程中让模型选择更好的分组方式，理论上每个通道的Feature Map是可以和所有Feature Map沟通到的。传统的沟通不畅的分组卷积自然不可能被学习到。图2是普通卷积核分组卷积的示意图。

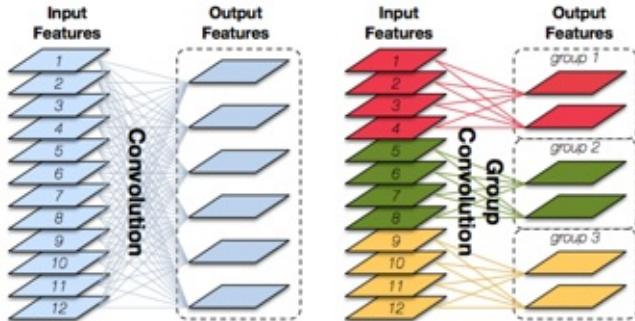


图2：(左)：普通卷积；(右)：分组卷积

我们换一个角度来看分组卷积，它也可以别看做普通卷积的稀疏表示，只不过指着稀疏方式是由认为生硬的指定的。这种稀疏连接虽然高效，但是人为的毫无根据的指定那些连接重要，哪些连接需要被删除无疑非常不合理。CondenseNet指出的解决方案是使用训练数据学习卷积网络的稀疏表示，让识别精度决定哪些权值该被保留，这个过程叫做*learning group convolution*，即图1中间红色的'L-Conv'。

1.2 自学习分组卷积

如图3所示，自学习分组卷积（Learned Group Convolution）可以分成两个阶段：浓缩（Condensing）阶段和优化（Optimizing）阶段。其中浓缩阶段用于剪枝没用的特征，优化阶段用于优化剪枝之后的网络。

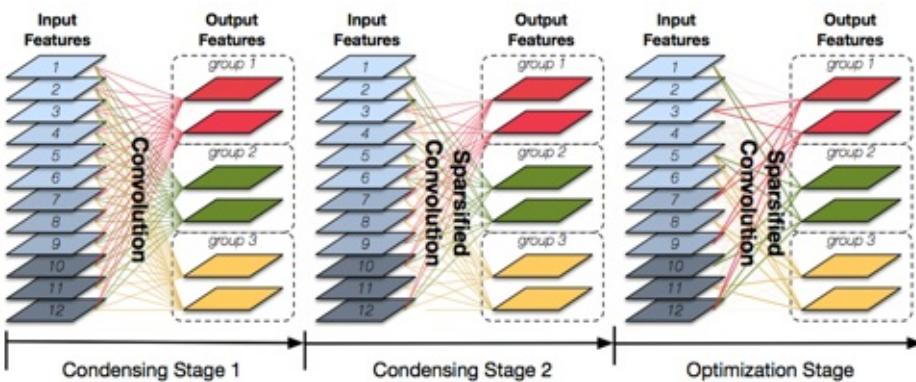


图3： $C=3$ 时的CondenseNet的训练情况

在图3中分组数 $G = 3$ 。浓缩率 $C = 3$ ，即只保留原来 $1/3$ 的特征。

浓缩率为 C 的CondenseNet会有 $C - 1$ 个浓缩阶段，它的第一个浓缩阶段（图3的最左侧）的初始化是普通的卷积网络，在训练该网络时使用了分组lasso正则项，这样学到的特征会呈现结构化稀疏分布，好处是在后面剪枝部分不会过分的影响精度。在每次浓缩阶段训练完成之后会有 $\frac{1}{C}$ 的特征被剪枝掉。也就是经过浓缩阶段后，仅有 $\frac{1}{C}$ 的特征被保留下来。

浓缩阶段之后是优化阶段，它会针对剪枝之后的网络单独做权值优化。CondenseNet用于优化阶段的总Epoch数和浓缩阶段是相同的。也就是说假设网络的总训练Epoch数是 M ，压缩率是 C 。它会有 $C - 1$ 个浓缩阶段，每个阶段的Epoch数均是 $\frac{M}{2(C-1)}$ 。以及一个优化阶段，Epoch数为 $\frac{2}{M}$ ，损失值，学习率以及Epoch分布情况见图4。

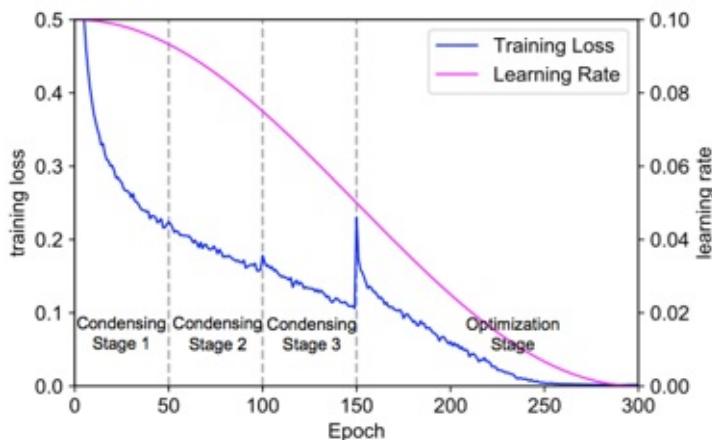


图4： $C=4$ 时的CondenseNet的训练Epoch分布情况以及cosine学习率

图4中是基于CIFAR-10数据集，CondenseNet的压缩率 $C = 4$ ，所以有3个浓缩阶段。学习率是采用的是cosine shape learning rate[90]。每次浓缩之后loss会有个明显的震动，最后一次loss震动的比较剧烈是因为一半的特征被剪枝掉。

1.3 剪枝准则

在1.2节中我们指出每个浓缩阶段中会有 $\frac{1}{C}$ 的特征被剪枝掉，这里我们来分析如何确定哪些卷积核应该被剪掉。

CondenseNet的浓缩阶段一般发生在 1×1 卷积部分，假设输入Feature Map的通道数是 R ，输出Feature Map的通道数是 O 。Feature Maps分成了 G 个组，图3中 $G = 3$ 。为了计算效率，我们希望剪枝之后每组的连接具有相同的稀疏模式，它们记做 $\{\mathbf{F}_1, \mathbf{F}_2, \dots, \mathbf{F}_G\}$ 。初始阶段， \mathbf{F}_g 的连接数是 $\frac{O}{G} \times F$ ，浓缩之后的连接数是 $\frac{O}{G} \times \frac{F}{C}$ 。 $j \in (1, R)$ 表示第输入Feature Map的索引， $i \in (1, \frac{O}{G})$ 表示组内输出Feature Map的索引。第 j 个滤波器关于这个组的重要性可以用权值的l1范数的和来表示：

$$\sum_{i=1}^{\frac{O}{G}} |\mathbf{F}_{i,j}^g|$$

我们一般可以认为 $l1$ 范数越大，该特征越重要，因为经过分组lasso正则化之后得到的是稀疏的特征，不重要的特征的值往往更接近0，因此可以被剪枝掉。在CondenseNet中我们剪枝掉的是 $\frac{O}{G} \times \frac{F}{C}$ 个最小的特征，如图3中浓缩阶段2训练的便是浓缩阶段1剪枝 $\frac{1}{C}$ 之后剩下的网络。优化阶段优化的是浓缩到只有原来的尺寸的 $\frac{1}{C}$ 的网络。

CondenseNet的剪枝并不是直接将这个特征删除，而是通过掩码的形式将被剪枝的特征置0，因此在训练的过程中CondenseNet的时间并没有减少，反而会需要更多的显存用来保存掩码。

1.4 索引层

经过训练过程的剪枝之后我们得到了一个系数结构，如图3右侧所示。目前这种形式是不能用传统的分组卷积的形式计算的，如果使用训练过程中的掩码的形式则剪枝的意义就不复存在。

为了解决这个问题，在测试的时候CondenseNet引入了索引层（Index Layer），索引层的作用是将输入Feature Map重新整理以方便分组卷积的高效运行。举例：图3中，组1使用的输入Feature Maps是(3,7,9,12)，组2使用的Feature Maps是(1,5,10,12)，组3使用的Feature Maps是(5,6,8,11)，索引层的作用就是将输入Feature Map排列成(3,7,9,12,1,5,10,12,5,6,8,11)的形式，之后便可以直接连接标准的分组卷积，如图5所示。

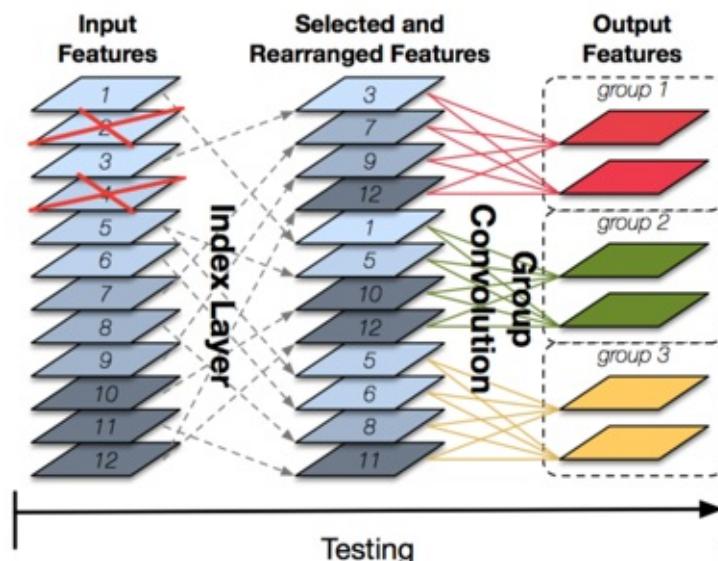


图5：CondenseNet的测试和Index层示意图

1.5 架构设计

在CondenseNet中作者对DenseNet做了两点改进：

Growth rate的指数级增长：增长率（Growth Rate） k 是在DenseNet中提出的一个超参数，反应的是每个Dense Block中Feature Map通道数的增长速度，例如一个block中第1层的Feature Map的通道数是 k_0 ，那么第 i 层的通道数即为 $k_0 + k \cdot (i - 1)$ 。

通过可视化DenseNet中特征重用的热力图，作者发现临近的Feature Map之间的特征重用更为有效，因此作者想通过增强临近节点之间的连接来增强模型的表现能力。为了实现这个动机，CondenseNet使用了指数级增长的增长率。按照上段给出的定义，第 i 层的通道数是 $k = 2^{i-1}k_0$ 。也就是说越接近block输出层的地方保留的Feature Map越多，

全密集连接：在DenseNet中，block之间是没有shortcut的，CondenseNet在block之间也增加了shortcut，结合平均池化用于实现不同尺寸的Feature Map之间的拼接用以实现更强的特征重用，如图6所示。

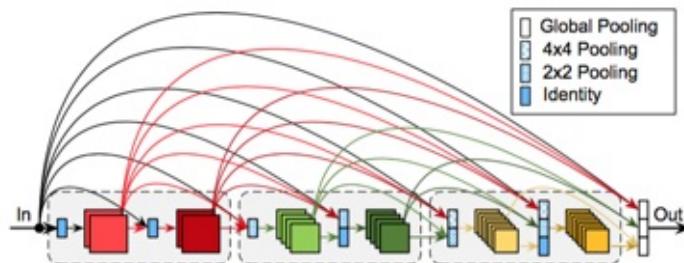


图6：CondenseNet block之间的全密集连接

2. 总结

文章最大的创新点是通过模型训练得到目前轻量级网络中常见的分组卷积的分组形式，比ShuffleNet解决分组卷积通信问题的方法更加有效，并结合Index层实现了测试时候的高效运行。最后，结合对DenseNet的dense block的改进完成了CondenseNet的完整结构。

Neural Architecture Search with Reinforcement Learning

tags: Reinforcement Learning, CNN, RNN, AutoML

前言

CNN和RNN是目前主流的CNN框架，这些网络均是由人为手动设计，然而这些设计是非常困难以及依靠经验的。作者在这篇文章中提出了使用强化学习（Reinforcement Learning）学习一个CNN（后面简称NAS-CNN）或者一个RNN cell（后面简称NAS-RNN）[105]，并通过最大化网络在验证集上的精度期望来优化网络，在CIFAR-10数据集上，NAS-CNN的错误率已经逼近当时最好的DenseNet[117]，在TreeBank数据集上，NAS-RNN要优于LSTM。

1. 背景介绍

文章提出了Neural Architecture Search（NAS），算法的主要目的是使用强化学习寻找最优网络，包括一个图像分类网络的卷积部分（表示层）和RNN的一个类似于LSTM的cell。由于现在的神经网络一般采用堆叠block的方式搭建而成，这种堆叠的超参数可以通过一个序列来表示。而这种序列的表示方式正是RNN所擅长的工作。

所以，NAS会使用一个RNN构成的控制器（controller）以概率 p 随机采样一个网络结构 A ，接着在CIFAR-10上训练这个网络并得到其在验证集上的精度 R ，然后在使用 R 更新控制器的参数，如此循环执行直到模型收敛，如图1所示。

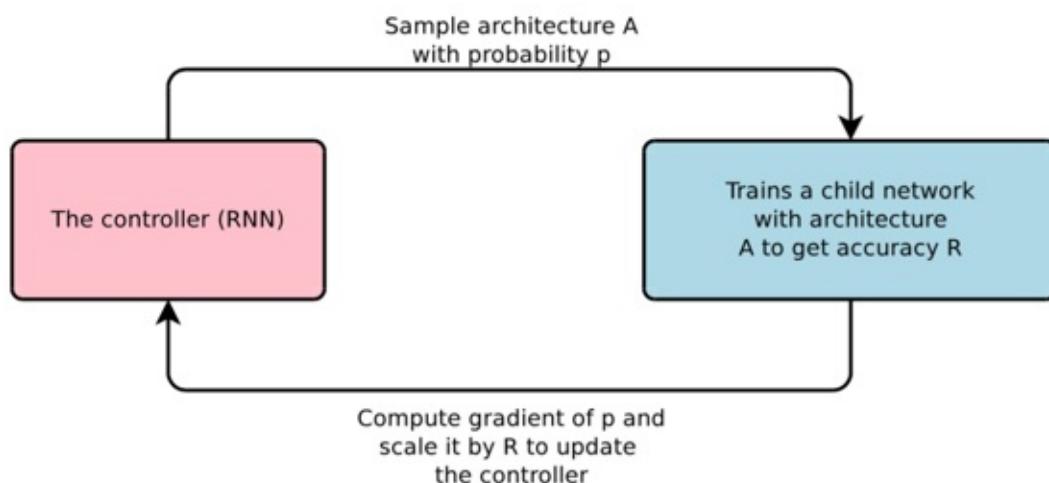


图1：NAS的算法流程图

2. NAS详细介绍

2.1 NAS-CNN

首先我们考虑最简单的CNN，即只有卷积层构成。那么这种类型的网络是很容易用控制器来表示的。即将控制器分成 N 段，每一段由若干个输出，每个输出表示CNN的一个超参数，例如Filter的高，Filter的宽，横向步长，纵向步长以及Filter的数量，如图2所示。

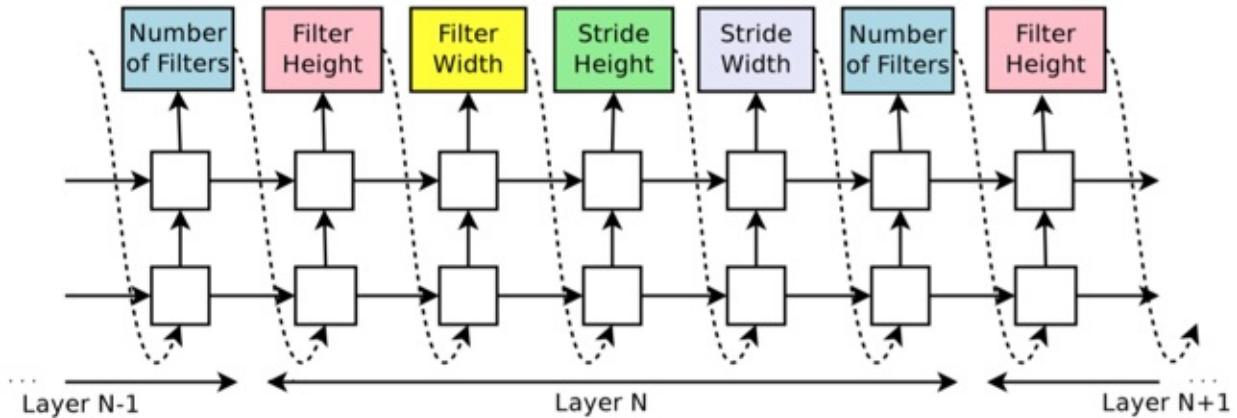


图2：NAS-CNN的控制器结构图

了解了控制器的结构以及控制器如何生成一个卷积网络，唯一剩下的也是最终要的便是如何更新控制器的参数 θ_c 。

控制器每生成一个网络可以看做一个action，记做 $a_{1:T}$ ，其中 T 是要预测的超参数的数量。当模型收敛时其在验证集上的精度是 R 。我们使用 R 来作为强化学习的奖励信号，也就是说通过调整参数 θ_c 来最大化 R 的期望，表示为：

$$J(\theta_c) = E_{P(a_{1:T}; \theta_c)}[R]$$

由于 R 是不可导的，所以我们需要一种可以更新 θ_c 的策略，NAS中采用的是Williams等人提出的REINFORCE rule [89]：

$$\nabla_{\theta_c} J(\theta_c) = \sum_{t=1}^T E_{P(a_{1:T}; \theta_c)}[\nabla_{\theta_c} \log P(a_t | a_{(t-1):1}; \theta_c) R]$$

上式近似等价于：

$$\frac{1}{m} \sum_{k=1}^m \sum_{t=1}^T \nabla_{\theta_c} \log P(a_t | a_{(t-1):1}; \theta_c) R_k$$

其中 m 是每个batch中网络的数量。

上式是梯度的无偏估计，但是往往方差比较大，为了减小方差算法中使用的是下面的更新值：

$$\frac{1}{m} \sum_{k=1}^m \sum_{t=1}^T \nabla_{\theta_c} \log P(a_t | a_{(t-1):1}; \theta_c) (R_k - b)$$

基线**b**是以前架构精度的指数移动平均值。

上面得到的控制器的搜索空间是不包含跳跃连接（skip connection）的，所以不能产生类似于ResNet或者Inception之类的网络。NAS-CNN是通过在上面的控制器中添加注意力机制[88]来添加跳跃连接的，如图3。

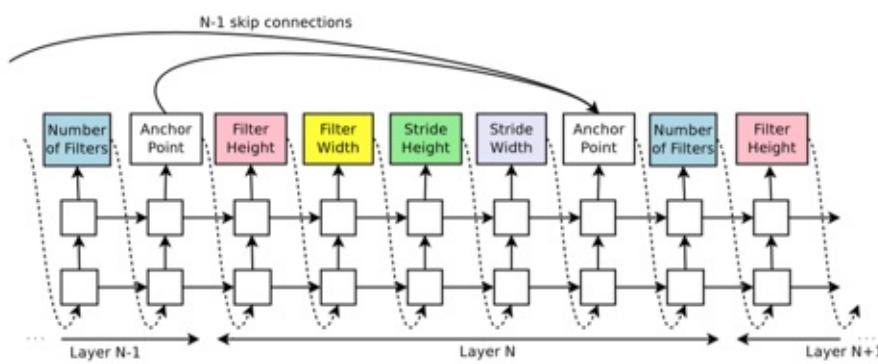


图3：NAS-CNN中加入跳跃连接的控制器结构图

在第 N 层，我们添加 $N - 1$ 个anchor来确定是否需要在该层和之前的某一层添加跳跃连接，这个anchor是通过两层的隐节点状态和sigmoid激活函数来完成判断的，具体的讲：

$$P(\text{Layer } j \text{ is an input to layer } i) = \text{sigmoid}(v^T \tanh(W_{prev} * h_j + W_{curr} * h_i))$$

其中 h_j 是第 j 层隐层节点的状态， $j \in [0, N - 1]$ 。 W_{prev} ， W_{curr} 和 v^T 是可学习的参数，跳跃连接的添加并不会影响更新策略。

由于添加了跳跃连接，而由训练得到的参数可能会产生许多问题，例如某个层和其它所有层都没有产生连接等等，所以有几个问题我们需要注意：

1. 如果一个层和其之前的所有层都没有跳跃连接，那么这层将作为输入层；
2. 如果一个层和其之后的所有层都没有跳跃连接，那么这层将作为输出层，并和所有输出层拼接之后作为分类器的输入；
3. 如果输入层拼接了多个尺寸的输入，则通过将小尺寸输入加值为0的padding的方式进行尺寸统一。

除了卷积和跳跃连接，例如池化，BN，Dropout等策略也可以通过相同的方式添加到控制器中，只不过这时候需要引入更多的策略相关参数了。

经过训练之后，在CIFAR-10上得到的卷积网络如图4所示。

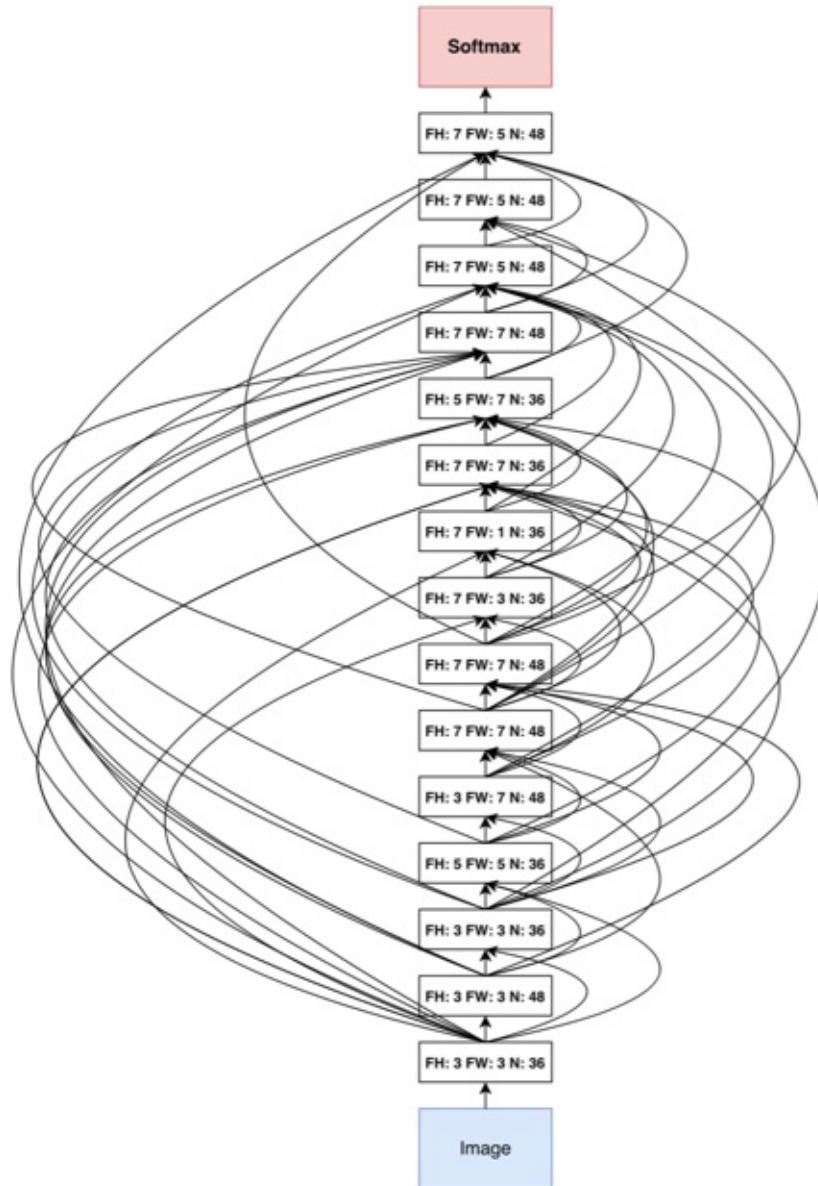


图4：NAS-CNN生成的密集连接的网络结构

从图4我们可以发现NAS-CNN和DenseNet有很多相通的地方：

1. 都是密集连接；
2. Feature Map的个数都比较少；
3. Feature Map之间都是采用拼接的方式进行连接。

在生成NAS-CNN的实验中，使用的是CIFAR-10数据集。网络中加入了BN和跳跃连接。卷积核的高的范围是 $[1, 3, 5, 7]$ ，宽的范围也是 $[1, 3, 5, 7]$ ，个数的范围是 $[24, 36, 48, 64]$ 。步长分为固定为1和在 $[1, 2, 3]$ 中两种情况。控制器使用的是含有35个隐层节点的LSTM。

2.2 NAS-RNN

在这篇文章中，作者采用强化学习的方法同样生成了RNN中类似于LSTM或者GRU的一个Cell。控制器的参数更新方法和1.2节类似，这里我们主要介绍如何使用一个RNN控制器来描述一个RNN cell。

传统RNN的输入是 x_t 和 h_{t-1} ，输出是 h_t ，计算方式是 $h_t = \tanh(W_1 x_t + W_2 h_{t-1})$ 。LSTM的输入是 x_t ， h_{t-1} 以及单元状态 c_{t-1} ，输出是 h_t 和 c_t ，LSTM的处理可以看做一个将 x_t ， h_{t-1} 和 c_{t-1} 作为叶子节点的树结构，如图5所示。

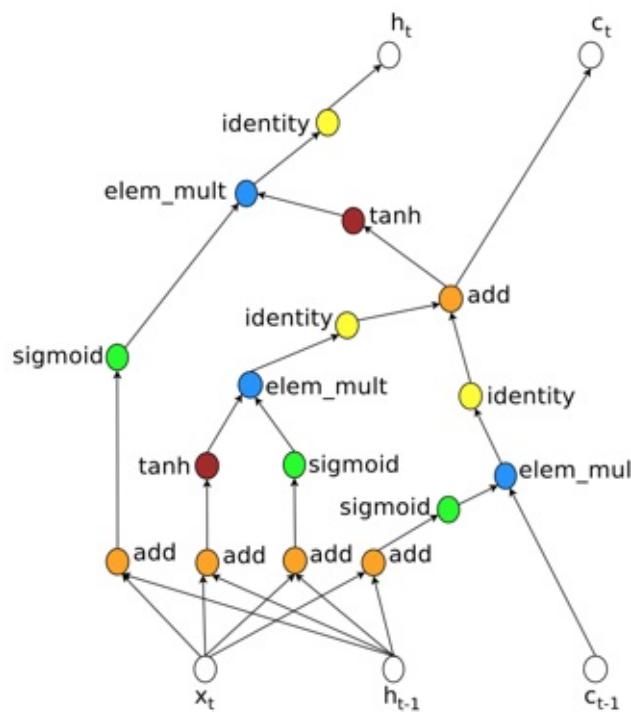


图5:LSTM的计算图

和LSTM一样，NAS-RNN也需要输入一个 c_{t-1} 并输出一个 c_t ，并在控制器的最后两个单元中控制如何使用 c_{t-1} 以及如何计算 c_t 。

如图6所示，在这个树结构中有两个叶子节点和一个中间节点，这种两个叶子节点的情况简称为base2，而图4的LSTM则是base4。叶子节点的索引是0，1，中间节点的索引是2，如图6左侧部分。也就是说控制器需要预测3个block，每个block包含一个操作（加，点乘等）和一个

激活函数（ReLU，sigmoid，tanh等）。在3个block之后接的是一个Cell inject，用于控制 c_{t-1} 的使用，最后是一个Cell indices，确定哪些树用于计算 c_t 。

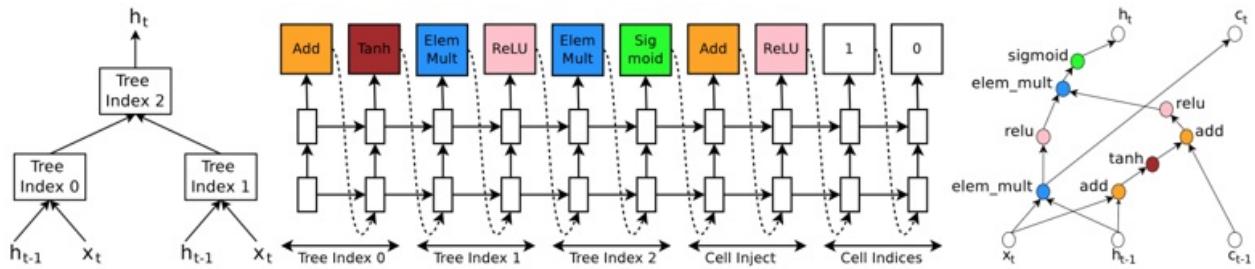


图6：NAS-RNN的控制器生成RNN节点示例图

详细分析一下图6：

1. 控制器为索引为0的树预测的操作和激活函数分别是Add和tanh，意味着

$$a_0 = \tanh(W_1 * x_t + W_2 * h_{t-1}) ;$$

2. 控制器为索引为1的树预测的操作和激活函数分别是ElemMult和ReLU，意味着

$$a_1 = \text{ReLU}((W_3 * x_t) \odot (W_4 * h_{t-1})) ;$$

3. 控制器为Cell Indices的第二个元素的预测值为0，Cell Inject的预测值是add和ReLU，意味着 a_0 值需要更新为 $a_0^{new} = \text{ReLU}(a_0 + c_{t-1})$ ，注意这里不需要额外的参数。

4. 控制器为索引为2的树预测的操作和激活函数分别是ElemMult和Sigmoid，意味着

$$a_2 = \text{sigmoid}(a_0^{new} \odot a_1) \text{，因为 } a_2 \text{ 是最大的树的索引，所以 } h_t = a_2 ;$$

5. 控制器为Cell Indices的第一个元素的预测值是1，意思是 c_t 要使用索引为1的树在使用激活函数的值，即 $c_t = (W_3 * x_t) \odot (W_4 * h_{t-1})$ 。

上面例子是使用“base 2”的超参作为例子进行讲解的，在实际中使用的是base 8，得到图7两个RNN单元。左侧是不包含max和sin的搜索空间，右侧是包含max和sin的搜索空间（控制器并没有选择sin）。

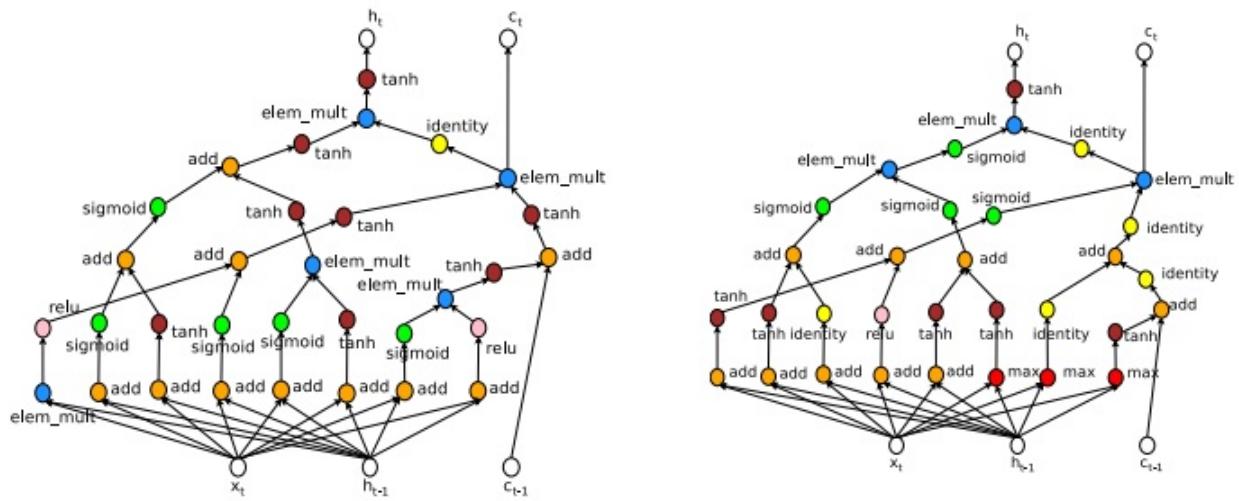


图7：NAS-CNN生成的网络节点的计算图

在生成NAS-RNN的实验中，使用的是Penn TreeBank数据集。操作的范围是[*add*, *elem_mult*]，激活函数的范围是[*identity*,*tanh*,*sigmoid*,*relu*]。

2. 总结

在如何使用强化学习方面，谷歌一直是领头羊，除了他们具有很多机构难以匹敌的硬件资源之外，更重要的是他们拥有扎实的技术积累。本文开创性的使用了强化学习进行模型结构的探索，提出了NAS-CNN和NAS-RNN两个架构，两个算法的共同点都是使用一个RNN作为控制器来描述生成的网络架构，并使用生成架构在验证集上的表现并结合强化学习算法来训练控制器的参数。

本文的创新性可以打满分，不止是其算法足够新颖，更重要的是他们开辟的使用强化学习来学习网络架构可能在未来几年引网络模型自动生成的方向，尤其是在硬件资源不再那么昂贵的时候。文章的探讨还比较基础，留下了大量的待开发空间为科研工作者所探索，期待未来几年出现更高效，更精确的模型的提出。

Learning Transferable Architectures for Scalable Image Recognition

tags: NAS, NASNet, AutoML

前言

在[NAS\[105\]](#)一文中我们介绍了如何使用强化学习学习一个完整的CNN网络或是一个独立的RNN单元，这种dataset interest的网络的效果也是目前最优的。但是NAS提出的网络的计算代价是相当昂贵的，仅仅在CIFAR-10上学习一个网络就需要500台GPU运行28天才能找到最优结构。这使得NAS很难迁移到大数据集上，更不要提ImageNet这样几百G的数据规模了。而在目前的行内规则上，如果不能在ImageNet上取得令人信服的结果，你的网络结构很难令人信服的。

为了将NAS迁移到大数据集乃至ImageNet上，这篇文章提出了在小数据（CIFAR-10）上学习一个网络单元（Cell），然后通过堆叠更多的这些网络单元的形式将网络迁移到更复杂，尺寸更大的数据集上面。因此这篇文章的最大贡献便是介绍了如何使用强化学习学习这些网络单元。作者将用于ImageNet的NAS简称为[NASNet\[104\]](#)，文本依旧采用NASNet的简称来称呼这个算法。实验数据也证明了NASNet的有效性，其在ImageNet的top-1精度和top-5精度均取得了当时最优的效果。

阅读本文前，强烈建议移步到我的[《Neural Architecture Search with Reinforcement Learning》](#)介绍文章中，因为本文并不会涉及强化学习部分，只会介绍控制器是如何学习一个NASNet网络块的。

1. NASNet详解

1.1 NASNet 控制器

在NASNet中，完整的网络的结构还是需要手动设计的，NASNet学习的是完整网络中被堆叠、被重复使用的网络单元。为了便于将网络迁移到不同的数据集上，我们需要学习两种类型的网络块：（1）*Normal Cell*：输出Feature Map和输入Feature Map的尺寸相同；（2）*Reduction Cell*：输出Feature Map对输入Feature Map进行了一次降采样，在Reduction Cell中，对使用Input Feature作为输入的操作（卷积或者池化）会默认步长为2。

NASNet的控制器的结构如图1所示，每个网络单元由 B 的网络块（block）组成，在实验中 $B = 5$ 。每个块的具体形式如图1右侧部分，每个块有并行的两个卷积组成，它们会由控制器决定选择哪些Feature Map作为输入（灰色部分）以及使用哪些运算（黄色部分）来计算输入

的Feature Map。最后它们会由控制器决定如何合并这两个Feature Map。

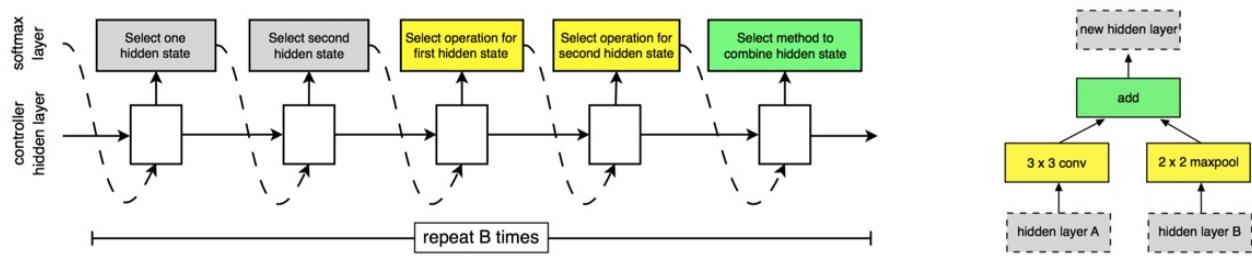


图1：NASNet控制器结构示意图

更精确的讲，NASNet网络单元的计算分为5步：

1. 从第 h_{i-1} 个Feature Map或者第 h_i 个Feature Map或者之前已经生成的网络块中选择一个Feature Map作为hidden layer A的输入，图2是学习到的网络单元，从中可以看到三种不同输入Feature Map的情况；
2. 采用和1类似的方法为Hidden Layer B选择一个输入；
3. 为1的Feature Map选择一个运算；
4. 为2的Feature Map选择一个元素；
5. 选择一个合并3, 4得到的Feature Map的运算。

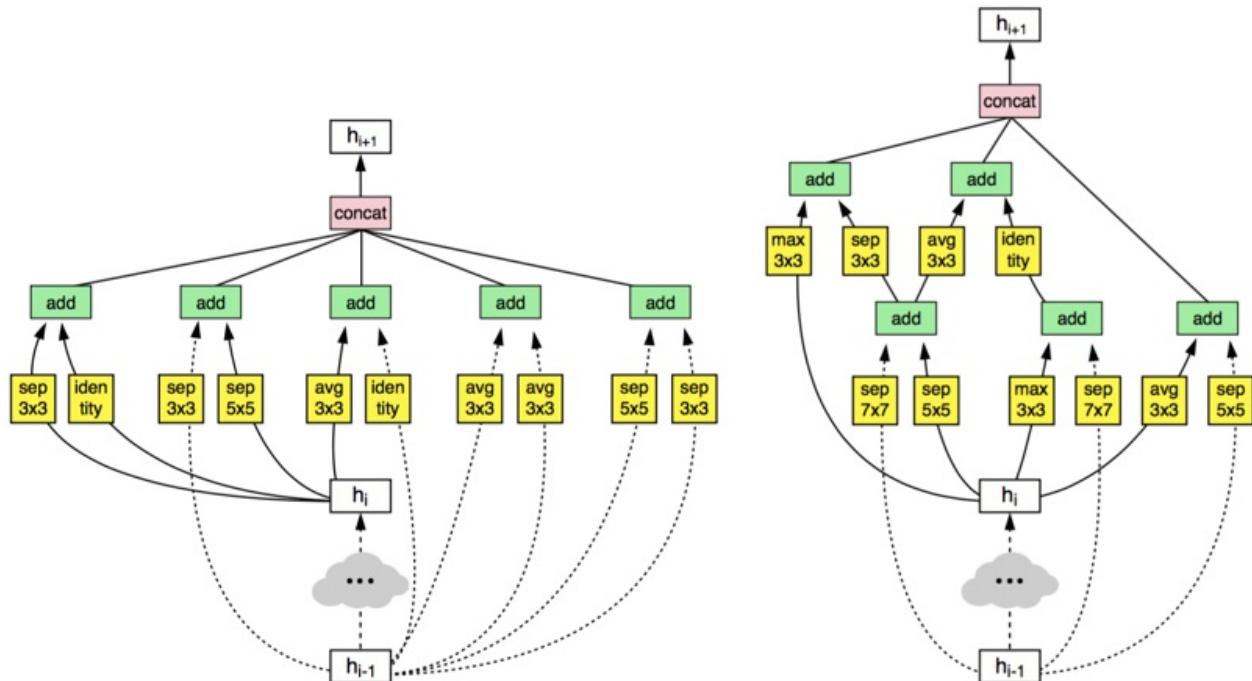


图2：NASNet生成的CNN单元。(左)：Normal Cell，(右) Reduction Cell

在3, 4中我们可以选择的操作有：

- 直接映射
- 1×1 卷积；
- 3×3 卷积；
- 3×3 深度可分离卷积；
- 3×3 空洞卷积；
- 3×3 平均池化；
- 3×3 最大池化；
- 1×3 卷积 + 3×1 卷积；
- 5×5 深度可分离卷积；
- 5×5 最大池化；
- 7×7 深度可分离卷积；
- 7×7 最大池化；
- 1×7 卷积 + 7×1 卷积；

在5中可以选择的合并操作有（1）单位加；（2）拼接。

最后所有生成的Feature Map通过拼接操作合成一个完整的Feature Map。

为了能让控制器同时预测Normal Cell和Reduction Cell，RNN会有 $2 \times 5 \times B$ 个输出，其中前 $5 \times B$ 个输出预测Normal Cell的 B 个块（如图1每个块有5个输出），后 $5 \times B$ 个输出预测Reduction Cell的 B 个块。RNN使用的是单层100个隐层节点的LSTM。

1.2 NASNet的强化学习

NASNet的强化学习思路和NAS相同，有几个技术细节这里说明一下：

1. NASNet进行迁移学习时使用的优化策略是Proximal Policy Optimization（PPO）[87]；
2. 作者尝试了均匀分布的搜索策略，效果略差于策略搜索。

1.3 Scheduled Drop Path

在优化类似于Inception的多分支结构时，以一定概率随机丢弃掉部分分支是避免过拟合的一种非常有效的策略，例如DropPath[86]。但是DropPath对NASNet不是非常有效。在NASNet的Scheduled Drop Path中，丢弃的概率会随着训练时间的增加线性增加。这么做的动机很好理解：训练的次数越多，模型越容易过拟合，DropPath的避免过拟合的作用才能发挥的越有效。

1.4 其它超参

在NASNet中，强化学习的搜索空间大大减小，很多超参数已经由算法写死或者人为调整。这里介绍一下NASNet需要人为设定的超参数。

1. 激活函数统一使用ReLU，实验结果表明ELU nonlinearityDropPath[85]效果略优于

ReLU :

2. 全部使用Valid卷积，padding值由卷积核大小决定；
3. Reduction Cell的Feature Map的数量需要乘以2，Normal Cell数量不变。初始数量人为设定，一般来说数量越多，计算越慢，效果越好；
4. Normal Cell的重复次数（图3中的N）人为设定；
5. 深度可分离卷积在深度卷积和单位卷积中间不使用BN或ReLU；
6. 使用深度可分离卷积时，该算法执行两次；
7. 所有卷积遵循ReLU->卷积->BN的计算顺序；
8. 为了保持Feature Map的数量的一致性，必要的时候添加 1×1 卷积。

堆叠Cell得到的CIFAR_10和ImageNet的实验结果如图3所示。

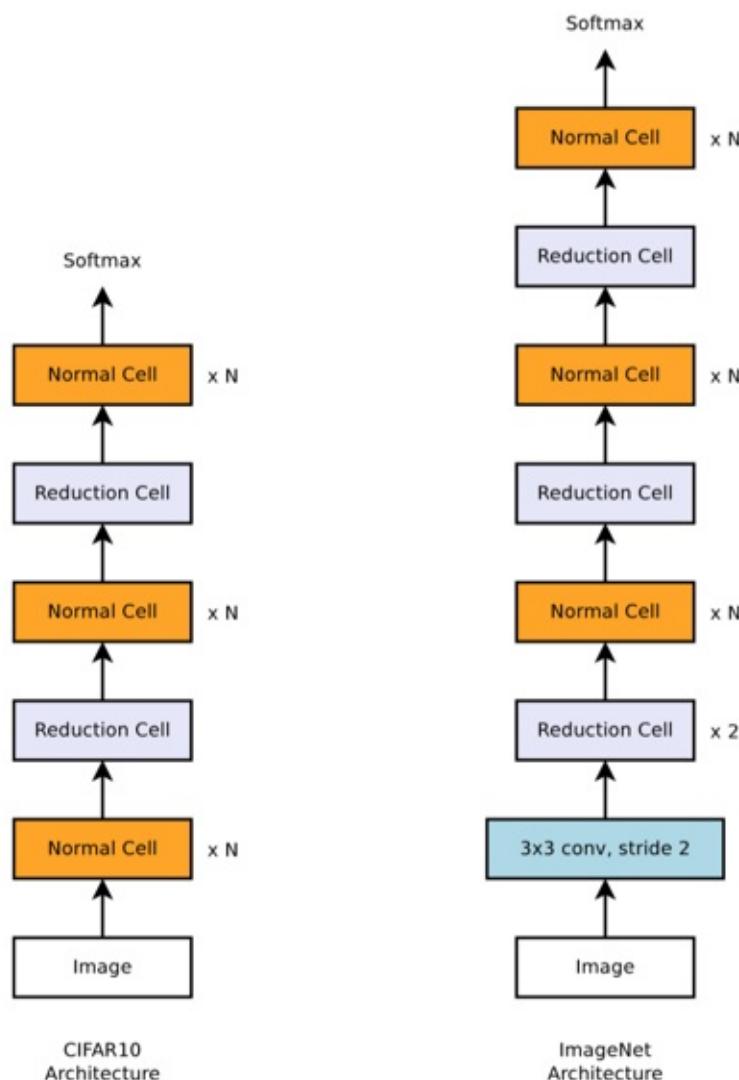


图3：NASNet的CIFAR10和ImageNet的网络结构

2. 总结

NASNet最大的贡献是解决了NAS无法应用到大数据集上的问题，它使用的策略是先在小数据集上学一个网络单元，然后在大数据集上堆叠更多的单元的形式来完成模型迁移的。

NASNet已经不再是一个dataset interest的网络了，因为其中大量的参数都是人为设定的，网络的搜索空间更倾向于密集连接的方式。这种人为设定参数的一个正面影响就是减小了强化学习的搜索空间，从而提高运算速度，在相同的硬件环境下，NASNet的速度要比NAS快7倍。

NASNet的网络单元本质上是一个更复杂的Inception，可以通过堆叠 网络单元的形式将其迁移到任意分类任务，乃至任意类型的任务中。论文中使用NASNet进行的物体检测也要优于其它网络。

本文使用CIFAR-10得到的网络单元其实并不是非常具有代表性，理想的数据集应该是ImageNet。但是现在由于硬件的计算能力受限，无法在ImageNet上完成网络单元的学习，随着硬件性能提升，基于ImageNet的NASNet一定会出现。或者我们也可以期待某个土豪团队多费电电费帮我们训练出这样一个架构来。

Progressive Neural Architecture Search

tags: NAS, NASNet, PNASNet, AutoML

前言

在[NAS\[105\]](#)和[NASNet\[104\]](#)中我们介绍了如何使用强化学习训练卷积网络的超参。NAS是该系列的第一篇，提出了使用强化学习训练一个控制器（RNN），该控制器的输出是卷积网络的超参，可以生成一个完整的卷积网络。NASNet提出学习网络的一个单元比直接整个网络效率更高且更容易迁移到其它数据集，并在ImageNet上取得了当时最优的效果。

本文是约翰霍普金斯在读博士刘晨曦在Google实习的一篇文章，基于NASNet提出了[PNASNet\[103\]](#)，其训练时间降为NASNet的1/8并且取得了比ImageNet上更优的效果。其主要的优化策略为：

1. 更小的搜索空间；
2. Sequential model-based optimization(SMBO)：一种启发式搜索的策略，训练的模型从简单到复杂，从剪枝的空间中进行搜索；
3. 代理函数：使用代理函数预测模型的精度，省去了耗时的训练过程。

在阅读本文之前，确保你已经读懂了[NAS](#)和[NASNet](#)两篇文章。

1. PNASNet详解

1.1 更小的搜索空间

回顾NASNet的控制器策略，它是一个有 $2 \times B \times 5$ 个输出的LSTM，其中2表示分别学习Normal Cell和Reduction Cell。B表示每个网络单元有B个网络块。5表示网络块有5个需要学习的超参，记做 (I_1, I_2, O_1, O_2, C) 。 $I_1, I_2 \in \mathcal{I}_b$ 用于预测网络块两个隐层状态的输入（Input），它会从之前一个，之前两个，或者已经计算的网络块中选择一个。 $O_1, O_2 \in \mathcal{O}$ 用于预测对两个隐层状态的输入的操作（Operation，共有13个，具体见NASNet。 $C \in \mathcal{C}$ 表示 O_1, O_2 的合并方式，有单位加和合并两种操作。因此它的搜索空间的大小为：

$$(2^2 \times 13^2 \times 3^2 \times 13^2 \times 4^2 \times 13^2 \times 5^2 \times 13^2 \times 6^2 \times 13^2 \times 2)^2 \approx 2.0 \times 10^{34}$$

PNASNet的控制器的运作方式和NASNet类似，但也有几点不同。

只有**Normal Cell**：PNASNet只学习了Normal Cell，是否进行降采样用户自己设置。当使用降采样时，它使用和Normal Cell完全相同的架构，只是要把Feature Map的数量乘2。这种操作使控制器的输出节点数变为 $B \times 5$ 。

更小的 \mathcal{O} ：在观察NASNet的实验结果是，我们发现有5个操作是从未被使用过的，因此我们将它们从搜索空间中删去，保留的操作剩下了8个：

- 直接映射
- 3×3 深度可分离卷积；
- 3×3 空洞卷积；
- 3×3 平均池化；
- 3×3 最大池化；
- 5×5 深度可分离卷积；
- 7×7 深度可分离卷积；
- 1×7 卷积 + 7×1 卷积；

合并 \mathcal{C} ：通过观察NASNet的实验结果，作者发现拼接操作也从未被使用，因此我们也可以将这种情况从搜索空间中删掉。因此PNASNet的超参数是四个值的集合 (I_1, I_2, O_1, O_2) 。

因此PNASNet的搜索空间的大小是：

$$2^2 \times 8^2 \times 3^2 \times 8^2 \times 4^2 \times 8^2 \times 5^2 \times 8^2 \times 6^2 \times 8^2 \approx 5.6 \times 10^{14}$$

我们可以写一些规则来排除掉两个隐层状态的对称的情况，但即使排除掉对称的情况后，NASNet的搜索空间的大小仍然为 10^{28} ，PNASNet的搜索空间仍然为 10^{12} 。这两个值的具体计算比较复杂，且和本文主要要讲解的内容关系不大，感兴趣的读者自行推算。

1.2 SMBO

尽管已经将优化搜索空间优化到了 10^{12} 的数量级，但是这个规模依然十分庞大，在其中进行搜索依旧非常耗时。这篇文章的核心便是提出了Sequential model-based optimization(SMBO)，它在模型的搜索空间中进行优化时会剪枝掉一些分支从而缩小模型的搜索空间。具体的讲SMBO的搜索是一种递进(Progressive)的形式，它的网络块的数目会从1个开始逐渐增加到 B 个。

当网络块数 $b = 1$ 时，它的搜索空间为 $2^2 \times 8^2 = 256$ （不考虑对称情况），也就是可以生成256个不同的网络块 (\mathcal{B}_1) ，构成网络的超参数为 \mathcal{S}_1 。这个搜索空间并不大，我们可以枚举出所有情况并训练由它们组成的网络 (\mathcal{M}_1) 。接着我们训练所有的 \mathcal{M}_1 个网络，接着得到训练后的模型 (\mathcal{C}_1) 。通过使用验证集我们可以得到每个模型的精度 (\mathcal{A}_1) 。有了网络超参数 \mathcal{S}_1

和它们对应的精度 \mathcal{A}_1 ，我们希望有一个代理函数 π 能够计算参数（特征）和精度（标签）额关系，这样我们就可以省去非常耗时的模型训练的过程了。代理函数的细节我们会在 1.3 节详细分析，在这你只需要把它看做从网络超参 \mathcal{S}_1 到它对应的精度 \mathcal{A}_1 的映射即可。

当网络块数 $b = 2$ 时，它的搜索空间为 $2^2 \times 8^2 \times 3^2 \times 8^2 = 147,456$ ，它的实际意义是在 $b=1$ 的基础上再扩展一个网络块，表示为 \mathcal{S}'_2 。使用 $b=1$ 时得到的代理函数 π 可以非常快速的为每个扩展模型非常快速的预测一个精度，表示为 \mathcal{A}'_2 ，这里可以称作代理精度。代理精度并不非常准确，我们需要得到真正的精度，它的作用是为我们剪枝搜索空间。具体的讲，我们会根据代理精度选取 top-K 个扩展模型 (\mathcal{S}_2)，一般 K 的值远小于搜索空间。仿照上段的过程，我们会依次使用 \mathcal{S}_2 搭建卷积网络 \mathcal{C}_2 ，使用 \mathcal{C}_2 得到模型在验证集上的精度 \mathcal{A}_2 ，最后我们使用得到的 $(\mathcal{S}_2, \mathcal{A}_2)$ 更新代理函数 π 。

仿照上一段的过程，我们可以使用 $b \geq 2$ 更新的代理函数 π 得到 $b+1$ 的 top-K 的扩展结构并更新得到新的代理函数 π 。以此类推直到 $b = B$ ，如 Algorithm 1 和图 1。

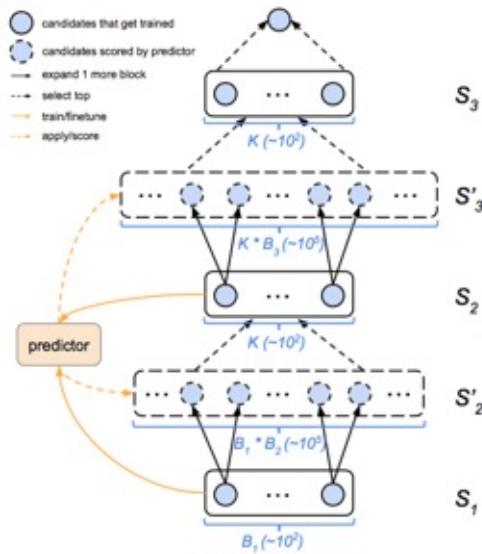
Algorithm 1 Progressive Neural Architecture Search (PNAS).

```

Inputs:  $B$  (max num blocks),  $E$  (max num epochs),  $F$  (num filters in first layer),
 $K$  (beam size),  $N$  (num times to unroll cell), trainSet, valSet.

 $\mathcal{S}_1 = \mathcal{B}_1$  // Set of candidate structures with one block
 $\mathcal{M}_1 = \text{cell-to-CNN}(\mathcal{S}_1, N, F)$  // Construct CNNs from cell specifications
 $\mathcal{C}_1 = \text{train-CNN}(\mathcal{M}_1, E, \text{trainSet})$  // Train proxy CNNs
 $\mathcal{A}_1 = \text{eval-CNN}(\mathcal{C}_1, \text{valSet})$  // Validation accuracies
 $\pi = \text{fit}(\mathcal{S}_1, \mathcal{A}_1)$  // Train the reward predictor from scratch
for  $b = 2 : B$  do
     $\mathcal{S}'_b = \text{expand-cell}(\mathcal{S}_{b-1})$  // Expand current candidate cells by one more block
     $\hat{\mathcal{A}}'_b = \text{predict}(\mathcal{S}'_b, \pi)$  // Predict accuracies using reward predictor
     $\mathcal{S}_b = \text{top-K}(\mathcal{S}'_b, \hat{\mathcal{A}}'_b, K)$  // Most promising cells according to prediction
     $\mathcal{M}_b = \text{cell-to-CNN}(\mathcal{S}_b, N, F)$ 
     $\mathcal{C}_b = \text{train-CNN}(\mathcal{M}_b, E, \text{trainSet})$ 
     $\mathcal{A}_b = \text{eval-CNN}(\mathcal{C}_b, \text{valSet})$ 
     $\pi = \text{update-predictor}(\mathcal{S}_b, \mathcal{A}_b, \pi)$  // Finetune reward predictor with new data
end for
Return top-K( $\mathcal{S}_B, \mathcal{A}_B, 1$ )

```

图1：SMBO流程图 ($B=3$)

SMBO像极了我们在[CTC](#)中介绍的宽度为K的Beam Search。

1.3 代理函数

1.2节中介绍SMBO时，代理函数 π 在其中发挥了至关重要的作用，从上面的过程中我们知道代理函数必须有下面3条特征：

1. 处理变长数据：在SMBO中我们会使用 b 的数据更新模型并在 $b+1$ 的扩展模型上预测精度；
2. 正相关：因为代理精度 \mathcal{A}'_b 的作用是用来选取top-K个扩展模型，所以其预测的精度不一定准确，但选取的top-K个扩展模型要尽可能的准确。所以保证代理函数预测的精度至少和实际精度是正相关的；
3. 样本有效：在SMBO中我们的用于训练模型的样本数量是K，为了效率K的值一般会很小，所以我们希望代理函数在小数据集上也能表现出好的结果。

处理变长数据的一个非常经典的模型便是RNN，因为它可以将输入数据按照网络块切分成时间片。具体的讲，LSTM的输入是尺寸为 $4 \times b$ 超参数 \mathcal{S}_b ，其中4指的是超参数的四个元素 (I_1, I_2, O_1, O_2) 。输入LSTM之前， (I_1, I_2) 经过one-hot编码后会通过一个共享的嵌入层进行编码， (O_1, O_2) 也会先one-hot编码再通过另外一个共享的嵌入层进行编码。最后的隐层节点经过一个激活函数为sigmoid的全连接得到最后的预测精度。损失函数使用L1损失。

作者也采用了一组MLP作为对照试验，编码方式是将每个超参数转换成一个D维的特征向量，四个超参数拼接之后会得到一个4-D的特征向量。如果网络块数 $b>1$ ，我们则取这 b 个特征向量的均值作为输入，这样不管几个网络块，MLP的输入的数据维度都是4-D。损失函数同样使用

L1损失。

由于样本数非常少，作者使用的是五个模型组成的模型集成。

为了验证代理函数在边长数据上的表示能力，作者在LSTM和MLP上做了一组排序相关性的对照试验。分析出的结论是在相同网络块下，LSTM优于MLP，但是在预测网络块多一个的模型上MLP要优于LSTM。原因可能是LSTM过拟合了。

2. PNASNet的实验结果

2.1 增进式的结构

根据1.2节介绍的SMBO的搜索过程，PNASNet可以非常容易得得出网络块数小于等于 B 的所有模型，其结果如图2所示。



图2：PNASNet得出的 $B=1, 2, 3, 4, 5$ 的几个网络单元，推荐使用 $B=5$

作者也尝试了 $B > 5$ 的情况，发现这时候模型的精度会下降，推测原因是因为搜索空间过去庞大了。

2.2 迁移到ImageNet

NAS中提倡学习dataset interest的网络结构，但是NASNet和PNASNet在CIFAR-10上学习到的结构迁移到ImageNet上也可以取得非常好的效果。作者通过一组不同网络单元在CIFAR-10和ImageNet上的实验验证了CIFAR-10和ImageNet在网络结构上的强相关性，实验结果见图3。

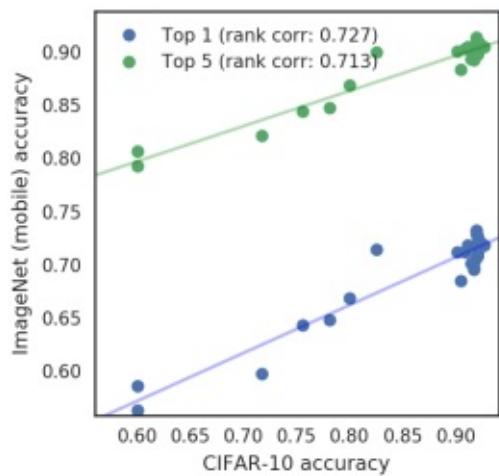


图3：CIFAR10和ImageNet对网络单元的强相关性

3. 总结

这篇PNASNet是之前NAS和NASNet的第三个系列，其着重点放在了优化强化学习的搜索空间的优化，几个优化的策略也是以此为目的。更少的参数是为了减小搜索空间的大小，SMBO是为了使用剪枝策略来优化强化学习探索的区域大小，而代理函数则提供了比随机采样更有效的采样策略。

本文使用的剪枝搜索和策略函数是强化学习最常见的技巧，例如AlphaGo。作为一个强化学习届的小白，对此无法给下一个特别准确地总结，只能期待大牛们努力推出更高效，精度更高，以及能够以更小代价得出模型的方法。

Regularized Evolution for Image Classifier Architecture Search

tags: AutoML, Aging Evolution, CNN

前言

在讲解AmoebaNet之前，先给大家讲一个故事：在一个物质资源非常匮乏的外星球上居住着一种只能进行无性繁殖的绝顶聪明的外星人，这个物质匮乏的星球上的资源匮乏到只够养活 P 个外星人。然而外星人为了种族进化还是要产生新的后代的，那么谁有资格产生后代呢，最厉害的那个外星人A提出基因最好的外星人才有资格产生后代。其它外星人不高兴了，因为他们担心整个星球都是A家族的人进而破坏了基因多样性。于是他们提出了一个折中方案，每次随机抽 S 个候选者参与竞争，里面最厉害的才有资格产生后代。如果A被抽中了，那A是里面最厉害的，就让A产生后代，如果A没有被抽中，也给其它不是很优秀的外星人一个机会。这样即保证了优秀基因容易产生更多后代，也保证了星球上的基因多样性。接着，由于产生了一个新的外星人，但是星球上的资源有限，所以必须杀死一个外星人给新的外星人留位置。A又提议了，我们杀死最笨的那个吧，其它外星人又不高兴了，生孩子的时候你A的概率最高，杀人的时候轮不到你了，久而久之这个星球上不全是你的后代了吗。经过商议，它们提出了一个最简单的方法：杀死岁数最大的那个。

故事讲完，我们开始正文。在我们之前介绍的NAS系列算法中，模型结构的搜索均是通过强化学习实现的。这篇要介绍的AmoebaNet是通过遗传算法的进化策略（Evolution）实现的模型结构的学习过程。该算法的主要特点是在进化过程中引入了年龄的概念，使进化时更倾向于选择更为年轻的性能好的结构，这样确保了进化过程中的多样性和优胜劣汰的特点，这个过程叫做年龄进化（Aging Evolution，AE）。作者为他的网络取名AmoebaNet，Amoeba中文名为变形体，是对形态不固定的生物体的统称，作者也是借这个词来表达AE拥有探索更广的搜索空间的能力。AmoebaNet取得了当时在ImageNet数据集上top-1和top-5的最高精度。

1. AmoebaNet算法详解

1.1 搜索空间

AmoebaNet使用的是和NASNet[104]相同的搜索空间。仿照NASNet的思想，AmoebaNet也是学习两个Cell：(1) Normal Cell，(2) Reduction Cell，在这里两个Cell是完全独立的。然后通过重复堆叠Normal Cell和Reduction Cell的形式我们可以得到一个完整的网络，如图1左所示。其中Normal Cell中步长始终为1，因此不会改变Feature Map的尺寸，Reduction Cell的步长为2，因此会将Feature Map的尺寸降低为原来的1/2。因此我们可以连续堆叠更多的Normal

Cell以获得更大的模型容量（不能堆叠Reduction Cell），如图1左侧图中Normal Cell右侧的 $\times N$ 的符号所示。在堆叠Normal Cell时，AmoebaNet使用了shortcut的机制，即一个Normal Cell的输入来自上一层，另外一个输入来自上一层的上一层，如图1中间部分。

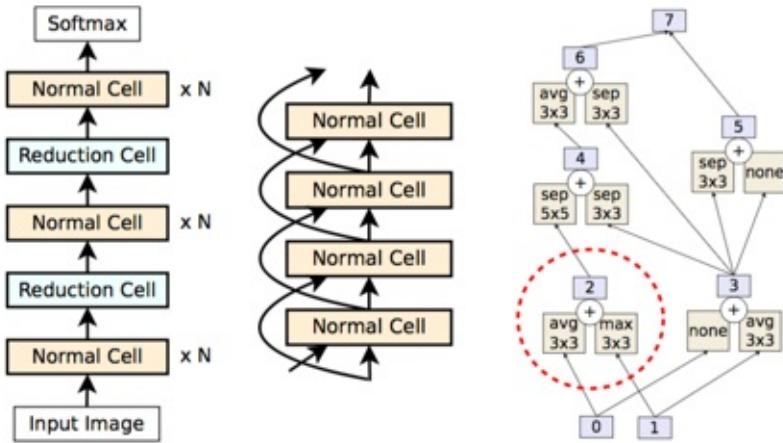


图1：AmoebaNet的搜索空间：(左)由Normal Cell和Reduction Cell构成的完整网络；
(中)Normal Cell内部的跳跃连接；(右)一个Normal/Reduction Cell 的内部结

在每个卷积操作中，我们需要学习两个参数：

1. 卷积操作的类型：类型空间参考NASNet。
2. 卷积核的输入：从该Cell中所有可以选择的Feature Map选择两个，每个Feature Map选择一个操作，通过合并这两个Feature Map得到新的Feature Map。最后将所有没有扇出的Feature Map合并作为最终的输出。上面所说的合并是单位加操作，因此Feature Map的个数不会改变。举例说明一下这个过程，根据图1中的跳跃连接，每个Cell有两个输入，对应图1右的0，1。那么第一个操作（红圈部分）选择0，1作为输入以及average池化和max池化作为操作构成新的Feature Map 2。接着第二个操作可以从（0，1，2）中选择两个作为输入，形成Feature Map 3，依次类推可以得到Feature Map 4，5，6，7。

最终AmoebaNet仅有两个变量需要决定，一个是每个Feature Map的卷积核数量 F ，另一个是堆叠的Normal Cell的个数 N ，这两个参数作为人工设定的超参数，作者也实验了 N 和 F 的各种组合。

1.2 Aging Evolution

AmoebaNet的进化算法Aging Evolution (AE) 如图2所示。

Algorithm 1 Aging Evolution

```

population ← empty queue           ▷ The population.
history ← ∅                         ▷ Will contain all models.
while |population| < P do        ▷ Initialize population.
    model.arch ← RANDOMARCHITECTURE()
    model.accuracy ← TRAINANDEVAL(model.arch)
    add model to right of population
    add model to history
end while
while |history| < C do          ▷ Evolve for C cycles.
    sample ← ∅                      ▷ Parent candidates.
    while |sample| < S do
        candidate ← random element from population
        ▷ The element stays in the population.
        add candidate to sample
    end while
    parent ← highest-accuracy model in sample
    child.arch ← MUTATE(parent.arch)
    child.accuracy ← TRAINANDEVAL(child.arch)
    add child to right of population
    add child to history
    remove dead from left of population    ▷ Oldest.
    discard dead
end while
return highest-accuracy model in history

```

在介绍代码之前，我们先看三条血淋淋的社会现实：

1. 优秀的父代更容易留下后代；
2. 年轻人比岁数大的更受欢迎；
3. 无论多么优秀的人都会有死去的一天。

这三个现实正是我从图2中的代码总结出来的，也是AE拿来进化网络的动机，现在我们来看看AE是如何反应这三点的。

第1行是使用队列（queue）初始化一个 population 变量。在AE中每个变量都有一个固定的生存周期，这个生存周期便是通过队列来实现的，因为队列的“先进先出”的特征正好符合AE的生命周期的特征。population 的作用是保存当前的存活模型，而只有存活的模型才有产生后代的能力。

第2行的 history 是用来保存所有训练好的模型。

第3行的作用是使用随机初始化的形式产生第一代存活的模型，个数正是循环的终止条件P。P的值在实验中给出的个数有20，64，100三个，其中P = 100的时候得到了最优解。

while 循环中（4-7行）便是随机初始化一个网络，然后训练并在验证集上测试这个网络的精度，最后将网络的架构和精度保存到 population 和 history 变量中。这里所有的模型评估都是在CIFAR-10上完成的。首先注意保存的是架构而不是模型，所以保存的变量的内容不会很多，因此并不会占用特别多的内存。其次由于 population 是一个队列，所以需要从右侧插入。而 history 插入变量时则没有这个要求。

第9行的第二个 `while` 循环表示的是进化的时长，即不停的向 `history` 中添加产生的优秀模型，直到 `history` 中模型的数量达到 C 个。 C 的值越大就越有可能进化出一个性能更为优秀的模型，我们也可以选择在模型开始收敛的结束进化。在作者的实验中 $C = 20,000$ 。

第10行的 `sample` 变量用于从存活的样本中随机选取 S 个模型进行竞争，第三个 `while` 循环中的代码（11-15行）便是用于随机选择候选父带。

第16行代码是从这 S 个模型只有精度最高的产生后代。这个有权利产生后代的变量命名为 `parent`。论文实验中 S 的值设定的值有 2, 16, 20, 25, 50，其中效果最好的值是 25。

第17行是使用变异（mutation）操作产生父代的子代，变量名是 `child`。变异的操作包括随机替换卷积操作（op mutation）和随机替换输入Feature Map（hidden state mutation），如图3所示。在每次变异中，只会进行一次变异操作，亦或是操作变异，亦或是输入变异。

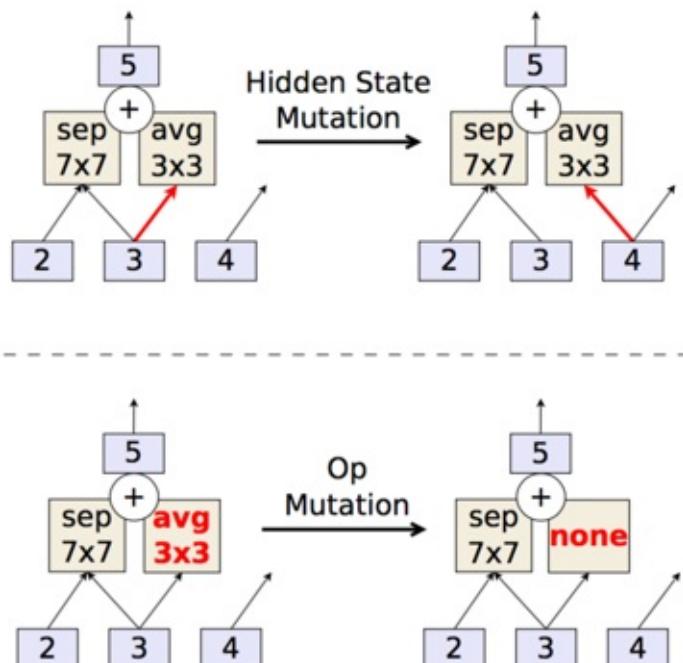


图3：AmoebaNet的变异操作：(上)Hidden State Mutation改变模型的输入Feature Map；(下)Op Mutation改变一个卷积操作

第18-20行依次是训练这个子代网络架构并将它依次插入 `population` 和 `history` 中。

第21-22行是从 `population` 顶端移除最老的架构，这一行也是AE最核心的部分。另外一种很多人想要使用的策略是移除效果最差的那个，这个方法在论文中叫做Non Aging Evolution (NAE)。作者这么做的动机是如果一个模型效果足够好，那么他有很大概率在他被淘汰之前已经在 `population` 中留下了自己的后代。如果按照NAE的思路淘汰差样本的话，`population` 中留下的样本很有可能是来自一个共同祖先，所以AE的方法得到的架构具有更强大的多样性。而NAE得到的架构由于多样性非常差，使得架构非常容易陷入局部最优值。这种情况在遗传学中也有一个名字：近亲繁殖。

最后一行代码是从所有训练过的模型中选择最好的那个作为最终的输出。

再回去看看开始的那个故事，讲的就是AE算法。

1.3 AmoebaNet 网络结构

通过上面的进化策略，产生的网络结构如图4所示，作者将其命名为AmoebaNet-A：

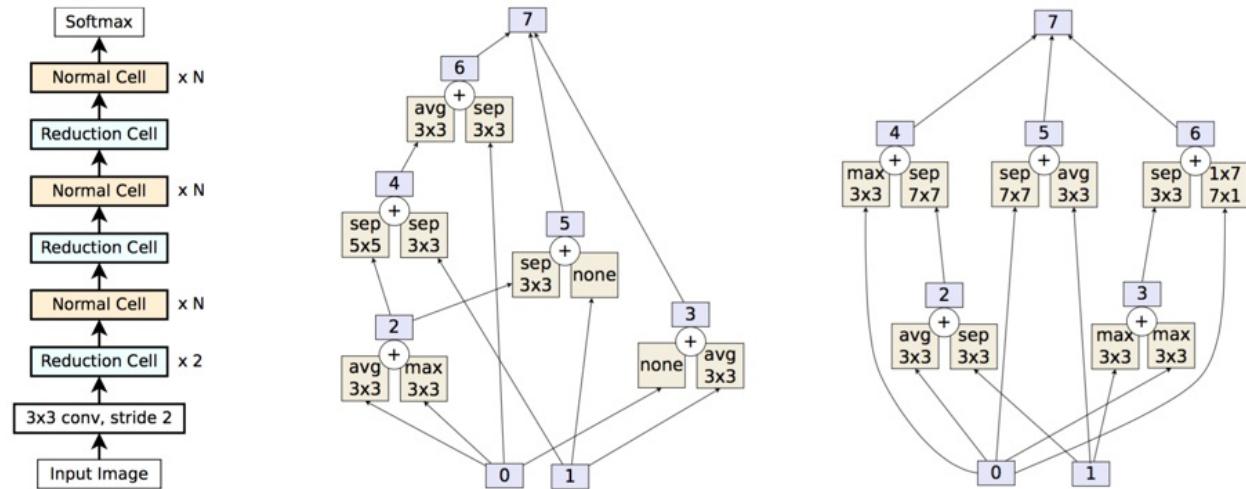


图4：AmoebaNet-A结构：(左)由Normal Cell和Reduction Cell构成的AmoebaNet-A；
(中)Normal Cell；(右)Reduction Cell

在图4中还有两个要手动设置的参数，一个参数是连续堆叠的Normal Cell的个数 N ，另外一个是卷积核的数量。在第一个Reduction之前卷积核的数量是 F ，后面每经过一次Reduction，卷积核的数量 $\times 2$ 。这两个参数是需要人工设置的超参数。

实验结果表明，当AmoebaNet的参数数量 ($N = 6, F = 190$) 达到了NASNet以及PNASNet[103]的量级 (80MB+) 时，AmoebaNet和其它两个网络在ImageNet上的精度是非常接近的。虽然AmoebaNet得到的网络和NASNet以及PNASNet非常接近，但是其基于AE的收敛速度是要明显快于基于强化学习的收敛速度的。

而最好的AmoebaNet的参数数量达到了469M时，AmoebaNet-A取得了目前在ImageNet上最优的测试结果。但是不知道是得益于AmoebaNet的网络结构还是其巨大的参数数量带来的模型容量的巨大提升。

最后作者通过一些传统的进化算法得到了AmoebaNet-B，AmoebaNet-C，AmoebaNet-D三个模型。由于它们的效果并不如AmoebaNet-A，所以这里不再过多介绍，感兴趣的同学去读论文的附录D部分。

从模型的精度上来看Aging Evolution (AE) 和RL (Reinfrocement Learning) 得到的同等量级参数的架构在ImageNet上的表现是几乎相同的，因此我们无法冒然的下结论说AE得到的模型要优于AL。但是AE的收敛速度快于RL是非常容易从实验结果中看到的。另外作者也添加了

一个Random Search (RS) 做对照实验，三个方法的收敛曲线图如图5所示：

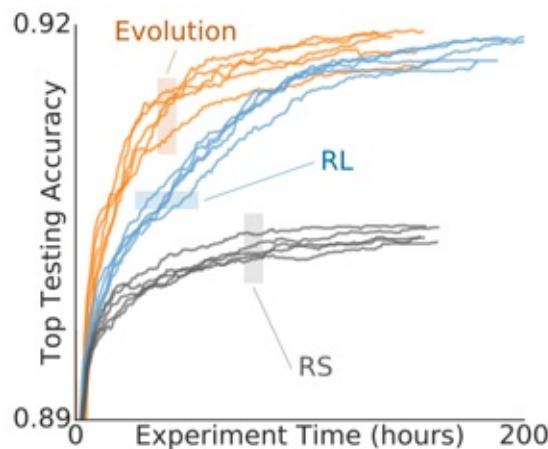


图5： AE ， RL 及 RS 在收敛速度上的对比曲线

2. 总结

这篇文章在NASNet的搜索空间的基础上尝试着使用进化策略来搜索网络的架构，并提出了一个叫做Aging Evolution (AE) 的进化策略。 AE 可以看做一个带有正则项的进化策略，它使得训练过程可以更加关注于网络架构而非模型参数。无论是拿强化学习和 AE 对比还是拿NAE和 AE 对比， AE 在收敛速度上均有明显的优势。同时 AE 的算法非常简单，正如我们在图2中的伪代码所示的它只有 P, C, S 三个参数，对比之下 RL 需要构建一个由LSTM构成的控制器， AE 的超参数明显少了很多。

最后，作者得到了一个目前为止在ImageNet上分类效果最好的AmoebaNet-A，虽然它的参数到达了4.69亿个。

实例解析：12306验证码破解

注意：本文章只适用于技术交流，切勿用于商业用途！

前言

又到一年一度的抢票季，12306是一个让人头疼的网站，一大因素是因为其变态的验证码机制。在这篇文章中我们将使用深度学习来破解12306验证码，并使用深度学习常见的策略，例如Dropout，迁移学习，数据增强等trick一步步提升模型识别率。

这里我们使用简单易用的Keras作为开源工具。上面提到的Trick也都是Keras自带的功能，所以在这里我们也会介绍一些Keras的基本用法，下面全部实验见链接：https://github.com/senliuy/12306_crack。

1. 数据分析

12306的验证码是从8个图片中找到要求的物体，如图1所示。



图1：12306验证码样式

我统计了1000个样本，发现12306的类别数其实只有80类，它们的类别以及对应的统计个数如表1

安全帽: 18	本子: 18	鞭炮: 10	冰箱: 18	菠萝: 12	苍蝇拍: 12	茶几: 12	茶盅: 13
创可贴: 9	刺绣: 16	打字机: 10	档案袋: 11	电饭煲: 16	电线: 12	电子秤: 11	调色板: 15
订书机: 14	耳塞: 16	风铃: 20	高压锅: 9	公交卡: 14	挂钟: 15	锅铲: 10	海报: 8
海鸥: 16	海苔: 13	航母: 14	黑板: 14	红豆: 14	红酒: 11	红枣: 13	护腕: 7
话梅: 10	剪枝: 10	金字塔: 10	锦旗: 13	卷尺: 13	开瓶器: 15	口哨: 12	蜡烛: 11
辣椒酱: 7	篮球: 13	老虎: 9	铃铛: 15	龙舟: 12	漏斗: 15	路灯: 17	绿豆: 11
锣: 11	蚂蚁: 10	毛线: 15	蜜蜂: 7	棉棒: 15	排风机: 13	牌坊: 12	盘子: 14
跑步机: 20	啤酒: 14	热水袋: 11	日历: 14	沙包: 13	沙拉: 13	珊瑚: 8	狮子: 8
手掌印: 11	薯条: 12	双面胶: 17	拖把: 2	网球拍: 10	文具盒: 9	蜥蜴: 12	药片: 13
仪表盘: 18	印章: 15	樱桃: 12	雨靴: 13	蒸笼: 11	中国结: 6	钟表: 12	烛台: 15

从上面的统计中我们可以看出，12306的验证码的破解工作可以转换成一个80类的分类问题，而这正是我们所擅长的，因为在物体分类领域我们尝试了太多的实验，例如MNIST，CIFAR-10，CIFAR-100等。

很幸运的是我们不需要人工标注数据，Kaggle上提供了一份开源的12306已标注图片数据集，注册之后即可下载，链接见：<https://www.kaggle.com/libowei/12306-captcha-image>。

在搭建模型之前我们需要将数据集分成训练集，验证集和测试集三个部分，我采用的策略是分别随机的从每类物体中各随机选取20个作为验证集和测试集。为了保证实验结果的可复现，我已将分好的数据集上传到百度云，下载链接见：<https://pan.baidu.com/s/1LksQZes3C1bM8ubIKUF6ag>。

2. 破解过程

物体分类的代码可以简单分成三个部分：

1. 网络搭建；
2. 数据读取；
3. 模型训练。

但是在上面的三步中每一步都存在一些超参数，怎么设置这些超参数是一个有经验的算法工程师必须掌握的技能。我们会在下面的章节中介绍每一步的细节，并给出我自己的经验和优化策略。

2.1 网络搭建

我们搭建一个分类网络时，可以使用上面几篇文章中介绍的经典的网络结构，也可以自行搭建。当自行搭建分类网络时，可以使用下面几步：

1. 堆积卷积操作（Conv2D）和最大池化操作（MaxPooling2D），第一层需要指定输入图像的尺寸和通道数；
2. Flatten()用于将Feature Map展开成特征向量；
3. 之后接全连接层和激活层，注意多分类应该使用softmax激活函数。

自行搭建网络时，我有几个经验：

1. 通道数的数量取 2^n ；
2. 每次MaxPooling之后通道数乘2；
3. 最后一层Feature Map的尺寸不宜太大也不宜太小(7-20之间是个不错的选择)；
4. 输出层和Flatten()层往往需要加最少一个隐层用于过渡特征；
5. 根据计算Flatten()层的节点数量设计隐层节点的个数。

下面代码是我搭建的一个分类网络，结构非常简单。

```
model_simple = models.Sequential()
model_simple.add(layers.Conv2D(32, (3,3), padding='same', activation='relu', input_shape = (66,66,3)))
model_simple.add(layers.MaxPooling2D((2,2)))
model_simple.add(layers.Conv2D(64, (3,3), padding='same', activation='relu'))
model_simple.add(layers.MaxPooling2D((2,2)))
model_simple.add(layers.Conv2D(128, (3,3), padding='same', activation='relu'))
model_simple.add(layers.MaxPooling2D((2,2)))
model_simple.add(layers.Flatten())
model_simple.add(layers.Dense(1024, activation='relu'))
model_simple.add(layers.Dense(80, activation='softmax'))
```

或者我们也可以使用之前提到的经典卷积网络，这里以VGG-16为例。Keras提供了VGG-16在ImageNet-2012（1000类）上的分类网络，由于输出节点数不一样，这里我们只取VGG-16的表示层，代码如下。

```

model_rand_VGG16 = models.Sequential()
rand_VGG16 = VGG16(weights=None, include_top=False, input_shape=(224, 224, 3))
model_rand_VGG16.add(rand_VGG16)
model_rand_VGG16.add(layers.Flatten())
model_rand_VGG16.add(layers.Dense(1024, activation='relu'))
model_rand_VGG16.add(layers.Dropout(0.25))
model_rand_VGG16.add(layers.BatchNormalization()) # 梯度爆炸
model_rand_VGG16.add(layers.Dense(80, activation='softmax'))
model_rand_VGG16.summary()

```

在上面代码中 `VGG16()` 函数用于调用 Keras 自带的 VGG-16 网络，`weights` 参数指定网络是否使用迁移学习模型，值为 `None` 时表示随机初始化，值为 `ImageNet` 时表示使用 ImageNet 数据集训练得到的模型。`include_top` 参数表示是否使用后面的输出层，我们确定了只使用表示层，所以取值为 `False`。`input_shape` 表示输入图片的尺寸，由于 VGG-16 会进行 5 次降采样，所以我们使用它的默认输入尺寸 $224 \times 224 \times 3$ ，所以输入之前会将输入图片放大。

2.2 数据读取

Keras 提供了多种读取数据的方法，我们推荐使用生成器的方式。在生成器中，Keras 在训练模型的同时把下一批要训练的数据预先读取到内存中，这样会节约内存，有利于大规模数据的训练。Keras 的生成器的初始化是 `ImageDataGenerator` 类，它有一些自带的数据增强的方法，我们会在 2.5 节进行介绍。

在这个实验中我们将不同的分类置于不同的目录之下，因此读取数据时使用的是 `flow_from_directory()` 函数，训练数据读取代码如下（验证和测试相同）：

```

train_data_gen = ImageDataGenerator(rescale=1./255)
train_generator = train_data_gen.flow_from_directory(train_folder,
                                                    target_size=(66, 66),
                                                    batch_size=128,
                                                    class_mode='categorical')

```

我们已近确定了是分类任务，所以 `class_mode` 的值取 `categorical`。

2.3 模型训练

当我们训练模型时首先我们要确定的优化策略和损失函数，这里我们选择了 `Adagrad` 作为优化策略，损失函数选择多分类交叉熵 `categorical_crossentropy`。由于我们使用了生成器读取数据，所以要使用 `fit_generator` 来向模型喂数据，代码如下。

```
model_simple.compile(loss='categorical_crossentropy', optimizer=optimizers.Adagrad(lr=0.01), metrics=['acc'])
history_simple = model_simple.fit_generator(train_generator,
                                             steps_per_epoch=128,
                                             epochs=20,
                                             validation_data=val_generator)
```

经过20个Epoch之后，模型会趋于收敛，损失值曲线和精度曲线见图2，此时的测试集的准确率是0.8275。从收敛情况我们可以分析到模型此时已经过拟合，我们需要一些策略来解决这个问题。

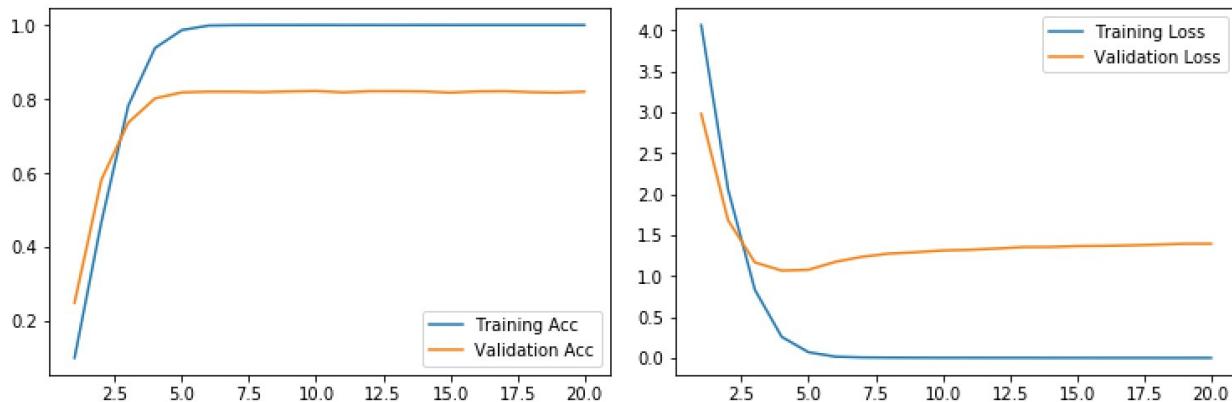


图2：自行搭建网络的损失曲线和精度曲线

2.4 Dropout

Dropout一直是解决过拟合非常有效的策略。在使用dropout时丢失率的设置是一个技术活，丢失率太小的话Dropout不能发挥其作用，丢失率太大的话模型会不容易收敛，甚至会一直震荡。在这里我在后面的全连接层和最后一层卷积层各加一个丢失率为0.25的Dropout。收敛曲线和精度曲线见图3，我们可以看出过拟合问题依旧存在，但是略有减轻，此时得到的测试集准确率是0.83375。

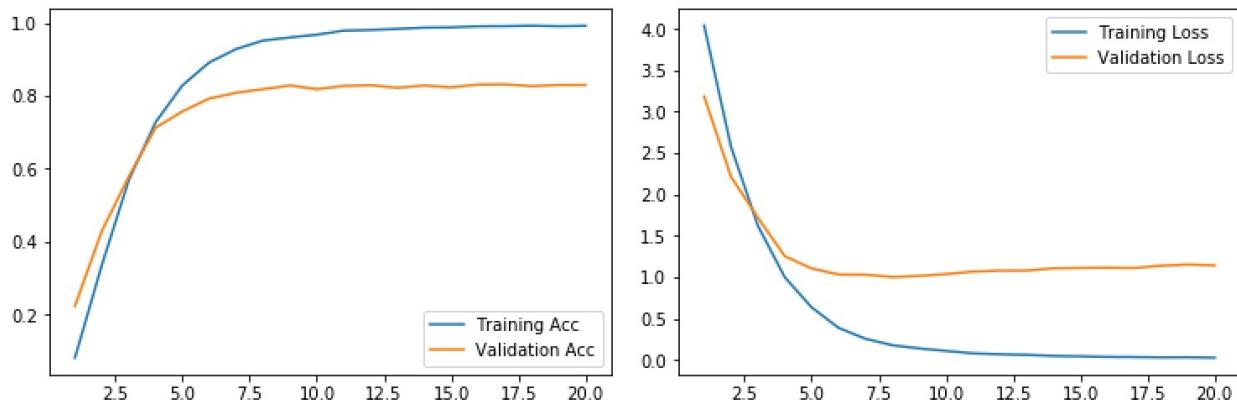


图3：加入Dropout之后的损失曲线和精度曲线

2.5 数据增强

Keras提供在调用`ImageDataGenerator`类的时候根据它的参数添加数据增强策略，在进行数据扩充时，我有几点建议：

1. 扩充策略的设置要建立在对数据集充分的观测和理解上；
2. 正确的扩充策略能增加样本数量，大幅减轻过拟合的问题；
3. 错误的扩充策略很有可能导致模型不好收敛，更严重的问题是使训练集和测试集的分布更加不一致，加剧过拟合的问题；
4. 往往开发者需要根据业务场景自行实现扩充策略。

下面代码是我使用的数据增强的几个策略。

```
train_data_gen_aug = ImageDataGenerator(rescale=1./255,
                                         horizontal_flip = True,
                                         zoom_range = 0.1,
                                         width_shift_range= 0.1,
                                         height_shift_range=0.1,
                                         shear_range=0.1,
                                         rotation_range=5)
train_generator_aug = train_data_gen_aug.flow_from_directory(train_folder,
                                                             target_size=(66, 66),
                                                             batch_size=128,
                                                             class_mode='categorical')
```

其中`rescale=1./255`参数的作用是对图像做归一化，归一化是一个在几乎所有图像问题上均有用的策略；`horizontal_flip = True`，增加了水平翻转，这个是适用于当前数据集的，但是在OCR等方向水平翻转是不能用的；其它的包括缩放，平移，旋转等都是常见的数据增强的策略，此处不再赘述。

结合Dropout，数据扩充可以进一步减轻过拟合的问题，它的收敛曲线和精度曲线见图4，此时得到的测试集准确率是0.84875。

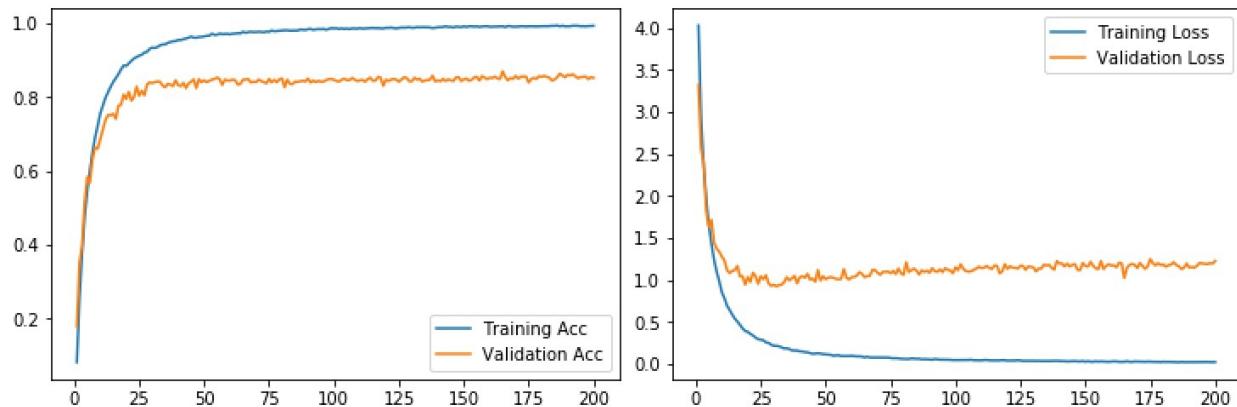


图4：加入数据增强之后的损失曲线和精度曲线

2.6 迁移学习

在2.1节中我们介绍了搭建模型中有自行搭建和使用经典模型两种策略，通过调用Keras的applications模块我们可以找到Keras中在ImageNet上训练过的几个模型，他们依次是：

- Xception
- VGG16
- VGG19
- ResNet50
- InceptionV3
- InceptionResNetV2
- MobileNet
- DenseNet
- NASNet

使用经典模型往往和迁移学习配合使用效果更好，所谓迁移学习是将训练好的任务A（最常用的是ImageNet）的模型用于当前任务的网络的初始化，然后在自己的数据上进行微调。该方法在数据集比较小的任务上往往效果很好。Keras提供用户自定义迁移学习时哪些层可以微调，哪些层不需要微调，通过layer.trainable设置。Keras使用迁移学习提供的模型往往比较深，容易产生梯度消失或者梯度爆炸的问题，建议添加BN层。最好的策略是选择好适合自己任务的网络后自己使用ImageNet数据集进行训练。

以VGG-16为例，其使用迁移学习的代码如下。第一次运行这段代码时需要下载供迁移学习的模型，因此速度会比较慢，请耐心等待。

```

model_trans_VGG16 = models.Sequential()
trans_VGG16 = VGG16(weights='imagenet', include_top=False, input_shape=(224, 224, 3))
model_trans_VGG16.add(trans_VGG16)
model_trans_VGG16.add(layers.Flatten())
model_trans_VGG16.add(layers.Dense(1024, activation='relu'))
model_trans_VGG16.add(layers.BatchNormalization())
model_trans_VGG16.add(layers.Dropout(0.25))
model_trans_VGG16.add(layers.Dense(80, activation='softmax'))
model_trans_VGG16.summary()

```

它的收敛曲线和精度曲线见图5，此时得到的测试集准确率是0.774375，此时迁移学习的效果反而不如我们前面随便搭建的网络。在这个问题上导致迁移学习模型表现效果不好的原因有两个：

1. VGG-16的网络过深，在12306验证码这种简单的验证码上容易过拟合；
2. 由于 `include_top` 的值为 `False`，所以网络的全连接层是随机初始化的，导致开始训练时损失值过大，带偏已经训练好的表示层。

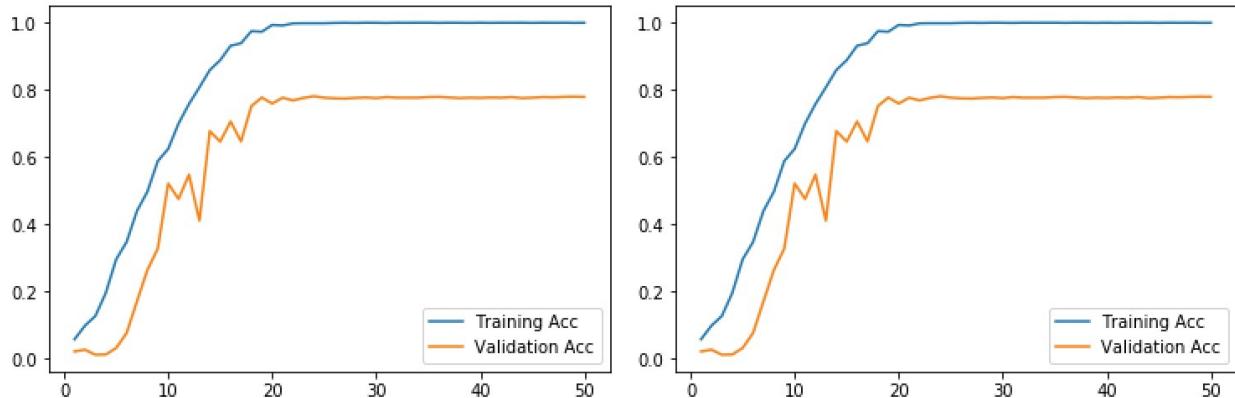


图5：VGG-16表示层可训练的损失曲线和精度曲线

为了防止表示层被带偏，我们可以将Keras中的层的 `trainable` 值设为 `False` 来达到此目的。结合之前介绍的数据增强和Dropout，最终我们得到的收敛曲线和精度曲线见图6，此时得到的测试集准确率是0.91625。

```

for layer in trans_VGG16.layers:
    layer.trainable = False

```

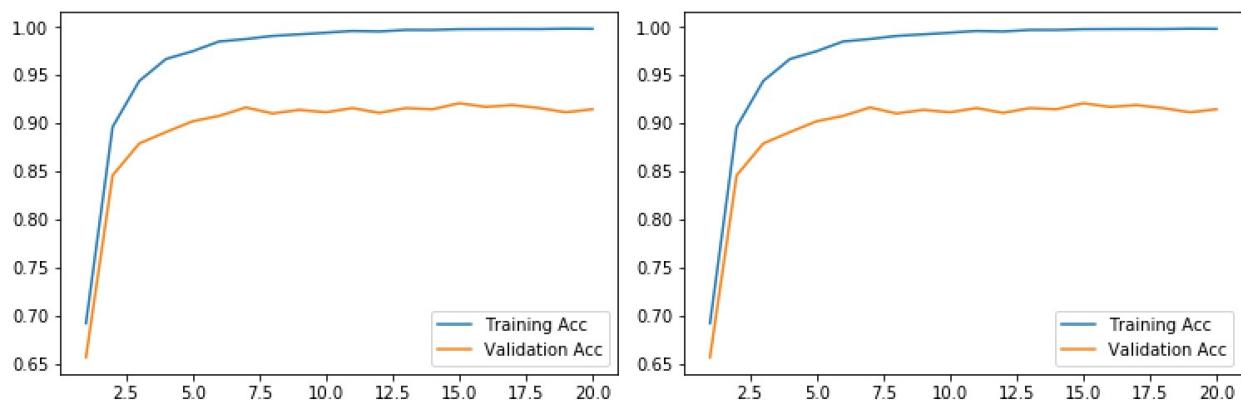


图6：VGG-16表示层不可训练的损失曲线和精度曲线

总结

在这篇文章中，我们将12306网站验证码的破解工作转换成了一个经典的多分类问题，并通过深度学习和一些trick将识别率提高到了91.625%。也许这个精度不能让您满意，此时你需要自己做一些工作来提升精度，以下是可能有用的几点：

1. 更合理的网络结构：网络层数，节点数量，卷积、池化、全连接的搭配；
2. 更好的缓解过拟合的策略：Dropout数量和位置，正则项；
3. 更合理的数据扩充策略；
4. 更合适的迁移学习模型以及冻结策略；
5. 者其它自己了解的其它优化方向的策略（例如自适应学习率，L1正则，Attention等）；
6. 采集并标注更多的数据。

91%的精度远远不是我们利用这批数据能达到的最高精度，写作这篇文章的目的是为了探讨深度学习在物体分类中的使用方法和针对训练日志优化模型的过程，如果你有更好的策略欢迎在评论区给出。

Recurrent Neural Network based Language Model

tags: NLP, Language Model

前言

在深度学习兴起之前，NLP领域一直是统计模型的天下，例如词对齐算法GIZA++，统计机器翻译开源框架MOSES等等。在语言模型方向，n-gram是当时最为流行的语言模型方法。一个最为常用的n-gram方法是回退（backoff）n-gram，因为n值较大时容易产生特别稀疏的数据，这时候回退n-gram会使用(n-1)-gram的值代替n-gram。

n-gram的问题是其捕捉句子中长期依赖的能力非常有限，解决这个问题的策略有cache模型和class-based模型，但是提升有限。另外n-gram算法过于简单，其是否有能力取得令人信服的效果的确要打一个大的问号。

一个更早的使用神经网络进行语言模型学习的策略是Bengio团队的使用前馈神经网络进行学习[83]。他们要求输入的数据由固定的长度，从另一个角度看它就是一个使用神经网络编码的n-gram模型，也无法解决长期依赖问题。基于这个问题，这篇文章使用了RNN作为语言模型的学习框架。

这篇文章介绍了如何使用RNN构建语言模型[82]，至此揭开了循环神经语言模型的篇章。由于算法比较简单，这里多介绍一些实验中使用的一些trick，例如动态测试过程等，希望能对你以后的实验设计有所帮助。（TODO：待之后对神经语言模型有系统的了解后，考虑将本文融合进综述的文章中）

1. 算法介绍

1.1 RNN

这篇文章中使用了最简单的RNN版本，而现在市场上普遍选择LSTM，GRU甚至NAS等具有捕捉更长时间长期依赖的节点。在RNN中，第t个时间片 $x(t)$ 读取的是 $t-1$ 时刻的状态 $s(t-1)$ 和t时刻的数据 $w(t)$ 。 $w(t)$ 是t时刻单词的one-hot编码，单词量在3万-20万之间； $s(t-1)$ 是 $t-1$ 时刻的隐藏层状态，实验中隐层节点数一般是30-500个， $t=0$ 时使用0.1进行初始化。上面过程表示为：

$$x(t) = w(t) + s(t-1)$$

t 时刻的隐藏层状态是 $x(t)$ 经过sigmoid激活函数 f 得到的值，其中 u_{ji} 是权值矩阵：

$$s_j(t) = f\left(\sum_i x_i(t)u_{ji}\right)$$

有的时候我们需要在每个时间片有一个输出，只需要在隐层节点 $s_j(t)$ 处添加一个softmax激活函数即可：

$$y_k(t) = g\left(\sum_j s_j(t)v_{kj}\right)$$

1.2 训练数据

训练语言模型的数据是不需要人工标注的，我们要做的就是寻找大量的单语言数据即可。在制作训练数据和训练标签时，我们通过取第0到 $t-1$ 时刻的单词作为网络输入，第 t 时刻的单词作为标签值。

由于输出使用了softmax激活函数，所以损失函数的计算使用的是交叉熵，输出层的误差向量为：

$$\text{error}(t) = \text{desired}(t) - y(t)$$

上式中 $\text{desired}(t)$ 是one-hot编码的模型预测值， $y(t)$ 是标签值。不知道上式的得来的同学自行搜索交叉熵的更新的推导公式，此处不再赘述。更新过程使用标准的SGD即可。

1.3 训练细节

初始化：使用的是均值为0，方差为0.1的高斯分布进行初始化。

学习率：初始值为0.1，当模型在验证集上的精度不再提升时将学习率减半，一般10-20个Epoch之后模型就收敛了。

正则：即使采用过大的隐藏层，网络也不会过度训练，并且实验结果表明添加正则项不会很有帮助。

动态模型：常见的机器/深度学习在测试的时候测试数据并不会用来更新模型。在这篇文章中作者认为测试数据也应该参与到模型的更新中，例如在测试数据中反复出现的一个人名等这种情况，作者将这种情况叫做动态模型。实验结果表明动态模型可以大大降低困惑度。

稀有（rare）类：为了提高模型的能力，作者将低于阈值的词合并到rare类中，词概率的计算方式如下：

$$y(t) \quad \text{if } w(t+1) \text{ is rare}$$

$$P(w_i(t+1)|w(t), s(t-1)) = \begin{cases} \frac{y_{rare}(t)}{C_{rare}} & \text{if } w_i(t+1) \text{ is rare} \\ y_i(t) & \text{otherwise} \end{cases}$$

其中 C_{rare} 是词表中词频低于阈值的单词的个数，所有的低频次都被平等对待，即它们的概率分布是均等的。

2. 总结

2019年的侧重点将会转移到NLP方向，首先拿一篇经典的RNN语言模型进行一下预热。毕竟发表在2010年，这篇文章的算法非常简单，RNN的效果必定不如LSTM或者GRU等，顺序语言模型也不如掩码语言模型能捕捉更多的上下文信息。这里只算抛砖引玉，在之后的文章中我们将介绍更多效果更好的语言模型。

Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation

简介

在很多时序分类 (Temporal Classification) 的应用中，输入数据 X 和输出数据 Y 的标签长度并不相等，而且不存在单调的映射关系，例如机器翻译，对话系统等等。为了解决这个问题，作者提出了 RNN Encoder-Decoder [81] 模型，RNN Encoder-Decoder 是由两个 RNN 模型级联而成的，通过 Encoder 将输入数据编码成特征向量，再通过 Decoder 将特征向量解码成输出数据。

这篇论文的第二个贡献就是 GRU (Gated Recurrent Unit) 的提出，GRU 和 LSTM 均是采用门机制的思想改造 RNN 的神经元，和 LSTM 相比，GRU 更加简单，高效，且不容易过拟合，但有时候在更加复杂的场景中效果不如 LSTM，算是 RNN 和 LSTM 在速度和精度上的一个折中方案。

论文的实现是对 SMT 中短语表的 rescore，即使用 MOSES (SMT 的一个开源工具) 根据平行语料产生短语表，使用 GRU 的 RNN Encoder-Decoder 对短语表中的短语对进行重新打分。

详解

1. RNN Encoder-Decoder

给定训练集 $D = (X, Y)$ ，我们希望最大化输出标签的条件概率，即：

$$p(y_1, y_2, \dots, y'_T | x_1, x_2, \dots, X_T)$$

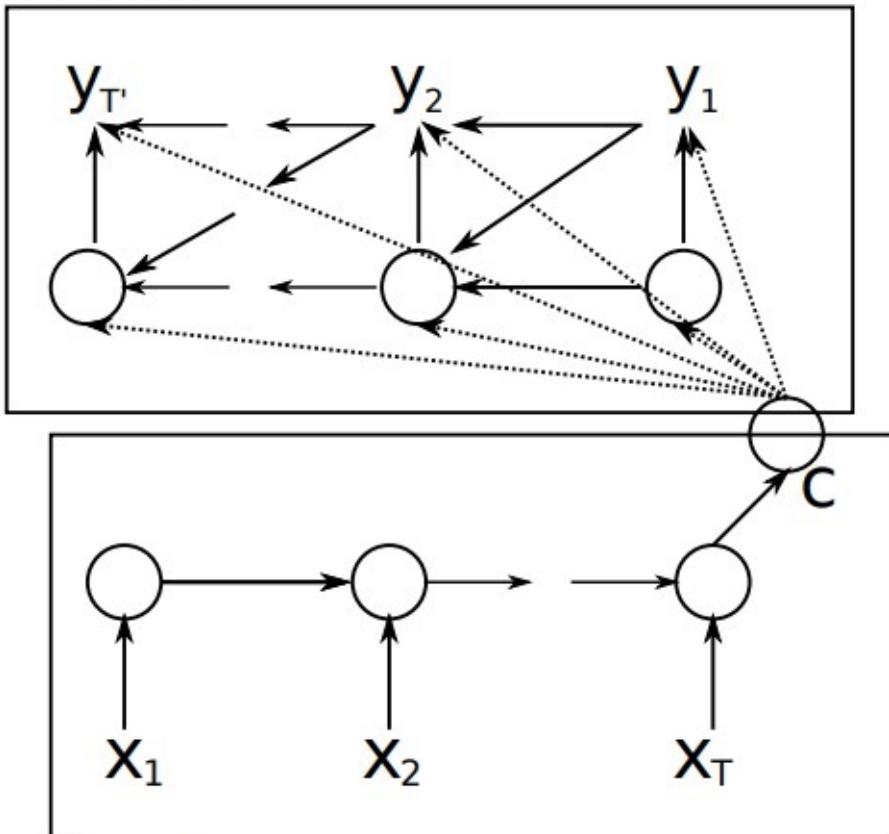
在上式中， $T \neq T'$ 。

1.1 编码

RNN Encoder-Decoder 的编码过程是先是通过一个 RNN 将变长的输入序列转换成固定长度的特征向量，再通过 RNN 将特征向量解码成需要的输出，如图 1.

图 1 : RNN Encoder-Decoder by Cho K

Decoder



Encoder

输入序列是一个标准的RNN，在计算时间片 t 的输出 $h_{<t>}$ 时，将 $h_{<t-1>}$ 和 $x_{<t>}$ 输入激活函数中，表示为

$$h_{<t>} = f(h_{<t-1>}, x_{<t>})$$

经过 T 个时间片后，得到一个 $h \times 1$ 的特征向量 \mathbf{c} ，其中 h 是隐层节点的节点数。 $f(\cdot)$ 是一个RNN单元，在这篇论文中，使用的是GRU，GRU的详细内容会在下面详细讲解。

1.2 解码

RNN Encoder-Decoder的解码过程是另外一个RNN，解码器的作用是将特征向量 \mathbf{c} ，前一个时间片的输出 $y_{<t-1>}$ ，以及前一个隐层节点 $h_{<t-1>}$ 作为输入，得到 $h_{<t>}$ ，表示为

$$h_{<t>} = f(h_{<t-1>}, y_{<t-1>}, \mathbf{c})$$

其中， $y_{<t>}$ 也是关于 \mathbf{c} ， $y_{<t-1>}$ 以及 $h_{<t-1>}$ 的条件分布

$$y_{<t>} = P(y_{<t>} | y_{<t-1>}, y_{<t-2>}, \dots, y_{<1>}, \mathbf{c}) = g(h_{<t>}, y_{<t-1>}, \mathbf{c})$$

RNN Encoder-Decoder的优化便是最大化 $p(y|x)$ 的log条件似然

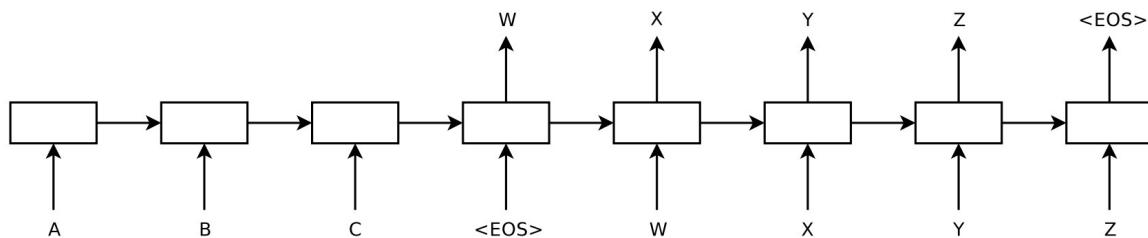
$$\max_{\theta} \frac{1}{N} \sum_{n=1}^N \log p_{\theta}(\mathbf{y}_n | \mathbf{x}_n)$$

其中 θ 是编码解码器的所有参数。

RNN Encoder-Decoder不仅可以用于产生输出数据，根据训练好的模型，使用条件概率模型 $p_{\theta}(\mathbf{y}_n | \mathbf{x}_n)$ ，也可以对现有的的标签进行评分。在这篇论文的实验中，作者便是对SMT中的短语表进行了重新打分。

另外一篇著名的Seq2Seq的论文 [80]几乎和这篇论文同时发表，在Seq2Seq中，编码器得到的特征向量仅用于作为解码器的第一个时间片的输入，结构如图2

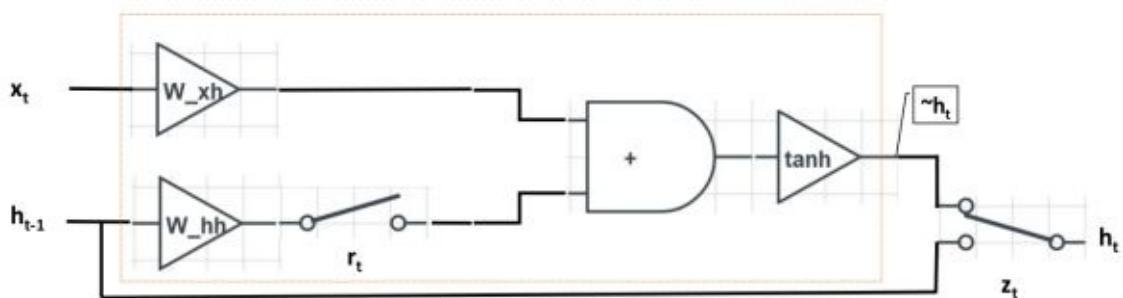
图2：RNN Encoder-Decoder by Sutskever |



2. GRU

对于短语rescore这个任务，作者最先使用的是RNN模型，遗憾的是RNN表现并不是非常理想。究其原因，在时间序列数据中， $\mathbf{h}_{<t>}$ 怎样更新，使用多少比例更新值，这些是可以仔细设计的，或者可以通过数据学习到的。根据LSTM中提出的门机制的思想，作者提出了一种更简单，更高效且更不容易过拟合的GRU，图3便是GRU的结构。

图3：GRU的结构



在上图中，有两个门：重置门（reset gate）以及更新门（update gate），两个门的计算均是通过当前时间片的输入数据 \mathbf{x}_t 以及上一个时间片的隐节点 $\mathbf{h}_{<t-1>}$ 计算而来：

重置门 r_j ：

$$r_j = \sigma([\mathbf{W}_r \mathbf{x}]_j + [\mathbf{U}_r \mathbf{h}_{}]_j)$$

更新门 z_j ：

$$z_j = \sigma([\mathbf{W}_z \mathbf{x}]_j + [\mathbf{U}_z \mathbf{h}_{}]_j)$$

其中， $[.]_j$ 表示向量的第j个元素， σ 是sigmoid激活函数。

重置门 r_j 用于控制前一时刻的状态 $\mathbf{h}_{}$ 对更新值的影响，当前一时刻的状态对当前状态的影响并不大时 $r_j = 0$ ，则更新值只受该时刻的输入数据 x_t 的影响：

$$\hat{h}_j^{} = \phi([\mathbf{W} \mathbf{X}]_j + [\mathbf{U}(\mathbf{r} \odot \mathbf{h}_{})]_j)$$

其中 ϕ 是tanh激活函数， \odot 是向量的按元素相乘。

而 z_t 用于控制该时间片的隐节点使用多少比例的上个状态，多少比例的更新值，当 $z_t = 1$ 时，则完全使用上个状态，即 $\mathbf{h}_{} = \mathbf{h}_{}$ ，相当于残差网络的short-cut。

$$h_j^{} = z_j h_j^{} + (1 - z_j) \hat{h}_j^{}$$

GRU的两个门机制是可以通过SGD和整个网络的参数共同调整的。

3. 总结

RNN Encoder-Decoder模型的提出和RNN门机制的隐层单元（LSTM/GRU）在解决长期依赖问题得到非常好的效果是分不开的。因为解码器使用的是编码器最后一个时间片的输出，加入我们使用的是经典RNN结构，则编码器得到的特征向量将包含大量的最后一个时间片的特征，而早期时间片的特征会在大量的计算过程中被抹掉。

参考文献

[1] <https://zhuanlan.zhihu.com/p/28297161>

Neural Machine Translation by Jointly Learning to Align and Translate

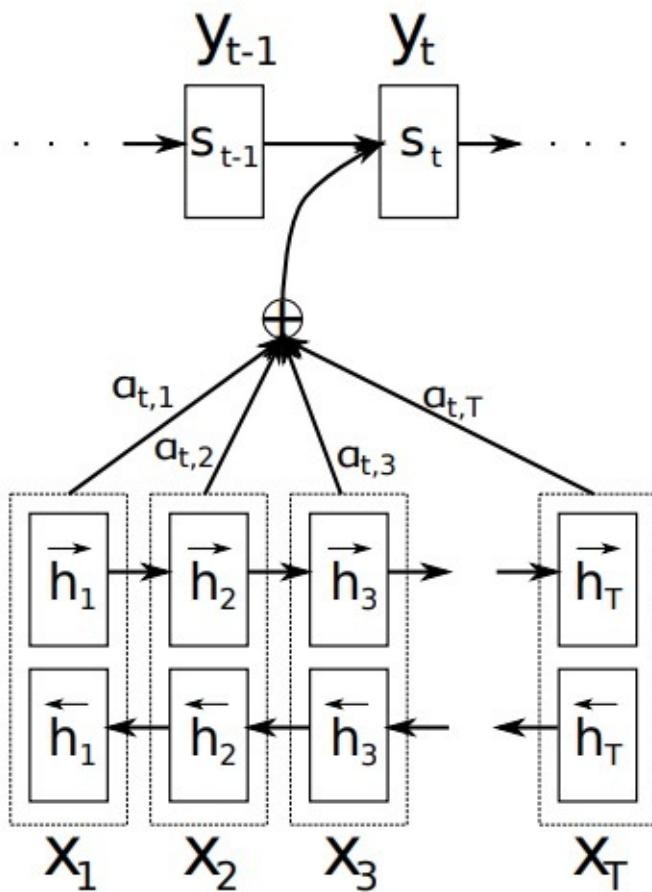
简介

在传统的RNN Encoder-Decoder模型中，在编码的过程中，将 $t-1$ 时的状态 $h_{}$ 和 t 时刻的数据 $x_{}$ 输入到 t 时刻的RNN单元中，得到 t 时刻的状态 $h_{}$ ，经过 T 个时间片后，得到长度等于隐节点数量的特征向量 \mathbf{c} 。在解码的过程中，将特征向量 \mathbf{c} 和上个时间片预测的输出 $y_{}$ 输入到RNN的单元中，得到该时刻的输出 $y_{}$ ，经过 T' 个时间片后得到输出结果。但在一些应用中，比如句子长度特别长的机器翻译场景中，传统的RNN Encoder-Decoder表现非常不理想。一个重要的原因是 t' 时刻的输出可能更关心输入序列的某些部分是什么内容而和其它部分是什么关系并不大。例如在机器翻译中，当前时间片的输出可能仅更注重原句子的某几个单词而不是整个句子。

这篇论文[88]率先提出了Attention的思想，通过Attention机制，模型可以同时学习原句子和目标句子的对齐关系和翻译关系。在编码过程中，将原句子编码成一组特征向量的一个集合，在翻译时，每个时间片会在该集合自行选择特征向量的一个子集用于产生输出结果。

详解

在这篇论文中，作者也是使用的RNN Encoder-Decoder结构。不同于传统的方式，在编码过程中，作者使用的是双向RNN（bi-RNN），每个RNN单元使用的是GRU。在解码过程中，使用的是基于Attention的GRU结构。算法结构如图1：



1.1 Encoder

双向RNN含有正向和反向两个方向，对于含有 T 个时间片的源句子 $X^T = \{x_1, x_2, \dots, x_T\}$ ，正向的输入数据是 $x_1 \rightarrow x_1 \rightarrow \dots \rightarrow x_T$ ，第 t 个时间片的隐节点 h_t 表示为

$$h_{<t>} = f(h_{<t-1>}, x_{<t>})$$

反向数据的输入序列是 $x_T \rightarrow x_{T-1} \rightarrow \dots \rightarrow x_1$ ，第 t 个时间片的隐节点 h'_t 表示为

$$h'_{<t>} = f(h'_{<t+1>}, x_{<t>})$$

其中 f 使用的是GRU的单元，详见上一篇论文的讲解。则第 t 个时间片的特征 $h_{<t>}$ 是前向和后向两个特征向量拼接到一起。

$$h_{<t>} = [h_t; h'_t]^T$$

1.2 Decoder

在解码的过程中，传统的RNN Encoder-Decoder的方式将整个句子的特征向量作为输入

$$s_{} = f(s_{}, y_{}, c)$$

Attention模型是使用所有特征向量的加权和，通过对特征向量的权值的学习，我们可以使用对当前时间片最重要的特征向量的子集 c_i ，即

$$s_{} = f(s_{}, y_{}, c_i)$$

其中 c_i 是 $h_{}$ 的加权和

$$c_i = \sum_{t=1}^T \alpha_{it} h_t$$

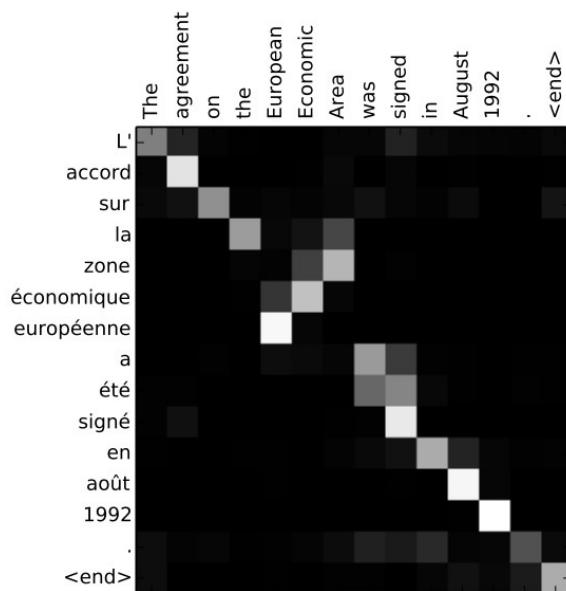
$$\alpha_{it} = \frac{\exp(e_{it})}{\sum_{k=1}^T \exp(e_{ik})}$$

$$e_{it} = a(s_{}, h_t)$$

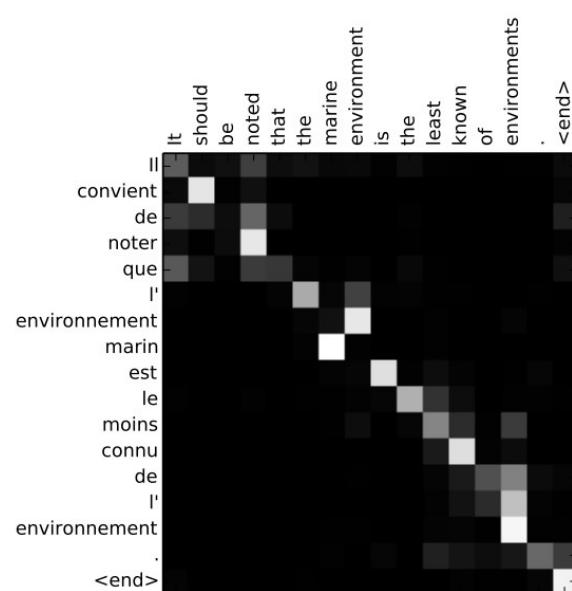
其中 e_{it} 是输出序列第*i*个时间片的对齐模型，表示的是该时刻和输入数据每个时间片的相关程度。使用前一时刻的状态 $s_{}$ 和第*t*个输入数据 h_t 计算得到，在作者的实验中， a 是使用的反正切tanh激活函数。

1.3 实验数据可视化

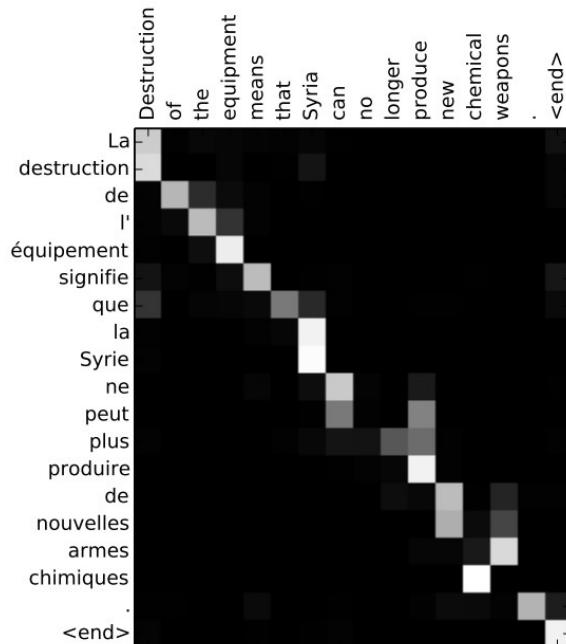
下图是通过可视化四组平行语料的 α 值得到的图， α 值越大，表示两个数据相关性越强，图中的颜色越浅。



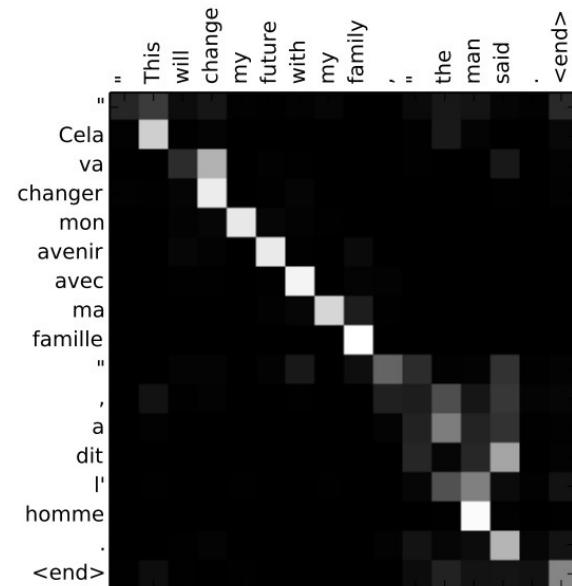
(a)



(b)



(c)



(d)

Hierarchical Attention Networks for Document Classification

tags: Attention

前言

本文提出了一个Hierarchical Attention Network (HAN) [79]模型用来做文章分类的任务，该算法提出的动机是考虑到在一个句子中，不同的单词对于决定这个句子的含义起着不同的作用；然后在一篇文章中，不同的句子又对于该文档的分类起着不同的作用。所以这篇层次Attention模型分别在单词层次和句子层次添加了一个Attention机制。实验结果表明这种机制可以提升文章分类的效果，同时通过Attention的权值向量的权值我们可以看出究竟哪些句子以及哪些单词对文档分类起着更重要的作用。

1. HAN算法详解

1.1 网络结构

HAN的网络结构如图1所示，它的核心结构由两个部分组成，下面是一个单词编码器加上基于单词编码的Attention层，上面是一个句子编码器和一个基于句子编码的Attention层。

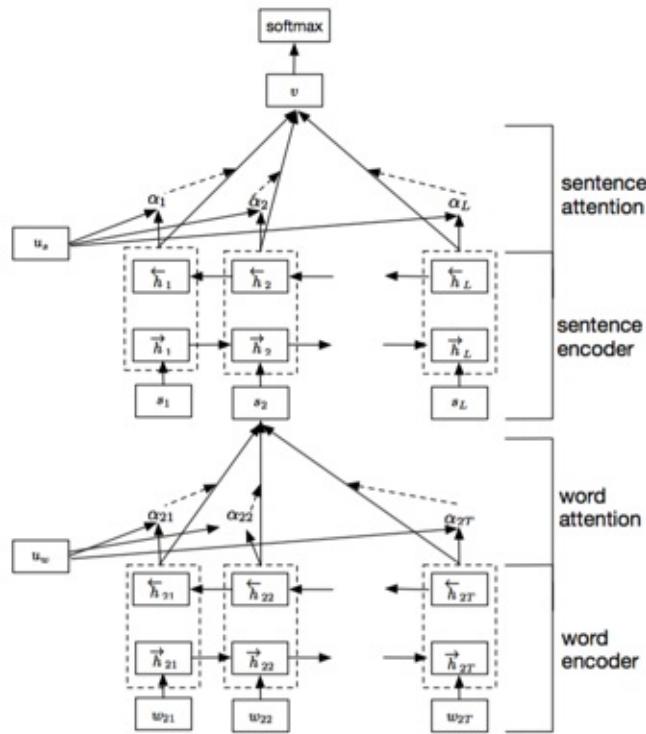


图1：HAN网络结构图

在详细介绍网络结构之前我们先给出几个重要参数的定义。假设一篇文章由 L 个句子组成，第 $s_i (i \in [1, L])$ 个句子包含 T_i 个单词， w_{it} 是第 i 个句子中的第 $t (t \in [1, T_i])$ 个单词。

1.2 单词编码器

图1中最底下的部分是个单词编码器，它的输入是一个句子。给定一个由单词 w_{it} 组成的句子 T_i ，它首先会经过一个嵌入矩阵编码成一个特征向量，例如word2vec等方法：

$$x_{it} = W_e w_{it}, t \in [1, T]$$

之后使用一个单层的双向GRU[88]对 x_{it} 进行编码：

$$h_{it} = \text{GRU}(x_{it}), t \in [1, T]$$

$$h'_{it} = \text{GRU}(x_{it}), t \in [T, 1]$$

双向GRU的输出是通过拼接前向GRU和反向GRU的方式得到的。

$$h_{it} = [h_{it}; h'_{it}]$$

1.3 单词Attention

单词编码器之上是一个单词Attention模块，它首先会将上一层得到的 h_{it} 输入一个MLP中得到它的非线性表示：

$$u_{it} = \tanh(W_w h_{it} + b_w)$$

接着便是Attention部分，首先需要使用softmax计算每个特征的权值。在论文中使用了Memory Network[78]，Memory Network是于2014年有FAIR提出的一种类似于神经图灵机的结构，它的核心部件是一个叫做记忆单元的部分，用来长期保存特征向量，也就是论文中的上下文向量 u_w ，它的值也会随着训练的进行而更新。Memory Network经过几年的发展也有很多性能更优的版本，但是由于坑比较深且业内没有广泛使用，暂时没有学习它的计划，感兴趣的同学请自行学习相关论文和代码。结合了Memory Network的权值的计算方式为：

$$\alpha_{it} = \frac{\exp(u_{it}^\top u_w)}{\sum_t \exp(u_{it}^\top u_w)}$$

最后得到的这个句子的编码 s_i 便是以 h_{it} 作为向量值， α_{it} 作为权值的加权和：

$$s_i = \sum_t \alpha_{it} h_{it}$$

1.4 句子编码器

句子编码器的也是使用了一个双向GRU，它的结构和单词编码器非常相似，数学表达式为：

$$h_i = \text{GRU}(s_i), t \in [1, T]$$

$$h'_i = \text{GRU}(s_i), t \in [T, 1]$$

$$h_i = [h_i; h'_i]$$

1.5 句子Attention

HAN的句子Attention部分也是使用了Memory Network带上下文向量的Attention结构，它的输入是句子编码器得到的特征向量，输出的是整个文本的特征向量 v 。

$$u_i = \tanh(W_s h_i + b_s)$$

$$\exp(u^\top u_w)$$

$$\alpha_i = \frac{\exp(u_i^\top u_s)}{\sum_i \exp(u_i^\top u_s)}$$

$$v = \sum_i \alpha_i h_i$$

1.5 句子分类

使用softmax激活函数我们可以根据文本向量 v 得到其每个类别的预测概率 p ：

$$p = \text{softmax}(W_c v + b_c)$$

由于使用了softmax激活函数，那么它的损失函数则应该是负log似然：

$$L = - \sum_d \log p_{dj}$$

其中 d 是批量中样本的下标， j 是分类任务中类别的下标。

2. 总结

结合要解决的问题的具体内容设计与之对应的算法流程和网络结构是一个合格算法工程师必须要掌握的技能之一。本文做了很好的示范，它在文档分类任务中将一个文档按照句子和单词进行了分层，并且在每层中使用了效果非常好的注意力机制。通过层次的注意力机制我们可以分析每个单词，每个句子在文档分类中扮演的作用，这对我们理解模型是非常有帮助的。

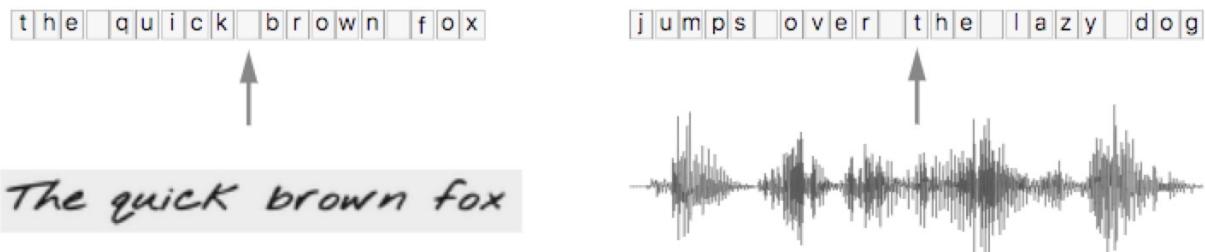
Connectionist Temporal Classification : Labelling Unsegmented Sequence Data with Recurrent Neural Networks

本文主要参考自Hannun等人在distill.pub发表的文章（<https://distill.pub/2017/ctc/>），感谢Hannun等人对CTC[77]的梳理。

简介

在语音识别中，我们的数据集是音频文件和其对应的文本，不幸的是，音频文件和文本很难再单词的单位上对齐。除了语言识别，在OCR，机器翻译中，都存在类似的Sequence to Sequence结构，同样也需要在预处理操作时进行对齐，但是这种对齐有时候是非常困难的。如果不使用对齐而直接训练模型时，由于人的语速的不同，或者字符间距离的不同，导致模型很难收敛。

CTC(Connectionist Temporal Classification)是一种避开输入与输出的一种方式，是非常适合语音识别或者OCR这种应用的。



给定输入序列 $X = [x_1, x_2, \dots, x_T]$ 以及对应的标签数据 $Y = [y_1, y_2, \dots, y_U]$ ，例如语音识别中的音频文件和文本文件。我们的工作是找到 X 到 Y 的一个映射，这种对时序数据进行分类的算法叫做 Temporal Classification。

对比传统的分类方法，时序分类有如下难点：

1. X 和 Y 的长度都是变化的；
2. X 和 Y 的长度是不相等的；
3. 对于一个端到端的模型，我们并不希望手动设计 X 和 Y 之间的对齐。

CTC提供了解决方案，对于一个给定的输入序列 X ，CTC给出所有可能的 Y 的输出分布。根据这个分布，我们可以输出最可能的结果或者给出某个输出的概率。

损失函数：给定输入序列 X ，我们希望最大化 Y 的后验概率 $P(Y|X)$, $P(Y|X)$ 应该是可导的，这样我们能执行梯度下降算法；

测试：给定一个训练好的模型和输入序列 X ，我们希望输出概率最高的 Y :

$$Y^* = \operatorname{argmax}_Y p(Y|X)$$

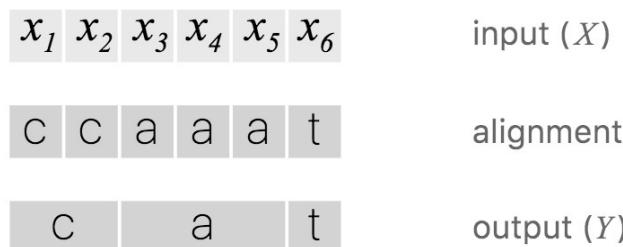
当然，在测试时，我们希望 Y^* 能够尽快的被搜索到。

算法详解

给定输入 X ，CTC输出每个可能输出及其条件概率。问题的关键是CTC的输出概率是如何考虑 X 和 Y 之间的对齐的，这种对齐也是构建损失函数的基础。所以，首先我们分析CTC的对齐方式，然后我们在分析CTC的损失函数的构造。

1.1 对齐

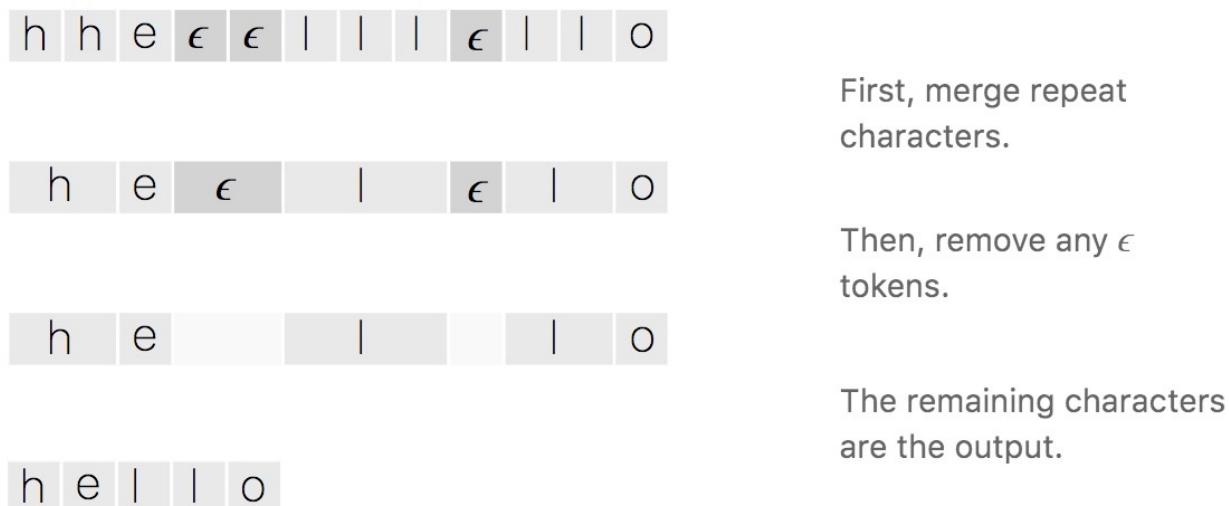
需要注意的是，CTC本身是不需要对齐的，但是我们需要知道 X 的输出路径和最终输出结果的对应关系，因为在CTC中，多个输出路径可能对应一个输出结果，举例来理解。例如在OCR的任务中，输入 X 是含有“CAT”的图片，输出 Y 是文本[C, A, T]。将 X 分割成若干个时间片，每个时间片得到一个输出，一个最简答的解决方案是合并连续重复出现的字母，如图2.



这个问题有两个缺点：

1. 几乎不可能将 X 的每个时间片都和输出 Y 对应上，例如OCR中字符的间隔，语音识别中的停顿；
2. 不能处理有连续重复字符出现的情况，例如单词“HELLO”，按照上面的算法，输出的是“HELO”而非“HELLO”。

为了解决上面的问题，CTC引入了空白字符 ϵ ，例如OCR中的字符间距，语音识别中的停顿均表示为 ϵ 。所以，CTC的对齐涉及去除重复字母和去除 ϵ 两部分，如图3。

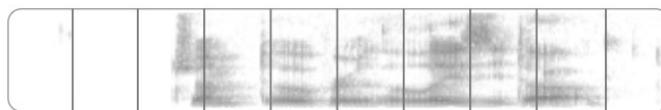


这种对齐方式有三个特征：

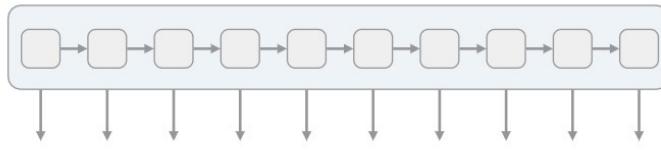
1. X 与 Y 之间的时间片映射是单调的，即如果 X 向前移动一个时间片， Y 保持不动或者也向前移动一个时间片；
2. X 与 Y 之间的映射是多对一的，即多个输出可能对应一个映射，反之则不成立，所以也有特征3；
3. X 的长度大于等于 Y 的长度。

1.2 损失函数

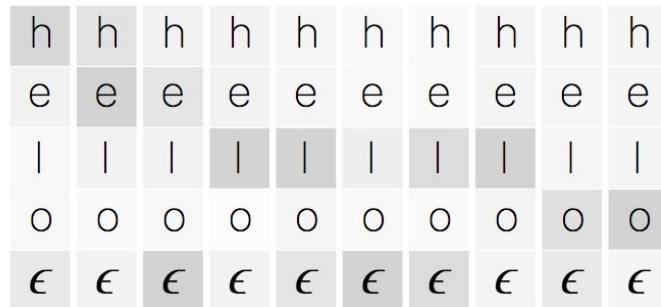
CTC的时间片的输出和输出序列的映射如图4：



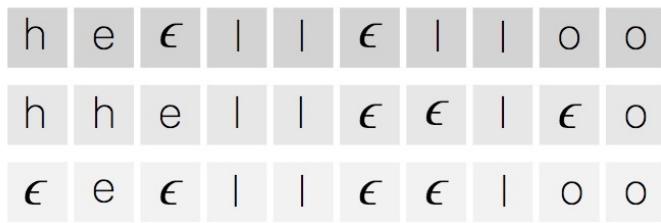
We start with an input sequence, like a spectrogram of audio.



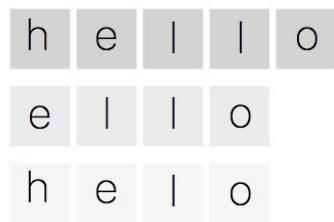
The input is fed into an RNN, for example.



The network gives $p_t(a | X)$, a distribution over the outputs $\{h, e, |, o, \epsilon\}$ for each input step.



With the per time-step output distribution, we compute the probability of different sequences



By marginalizing over alignments, we get a distribution over outputs.

也就是说，对应标签 Y ，其关于输入 X 的后验概率可以表示为所有映射为 Y 的路径之和，我们的目标就是最大化 Y 关于 $x = y$ 的后验概率 $P(Y|X)$ 。假设每个时间片的输出是相互独立的，则路径的后验概率是每个时间片概率的累积，公式及其详细含义如图5。

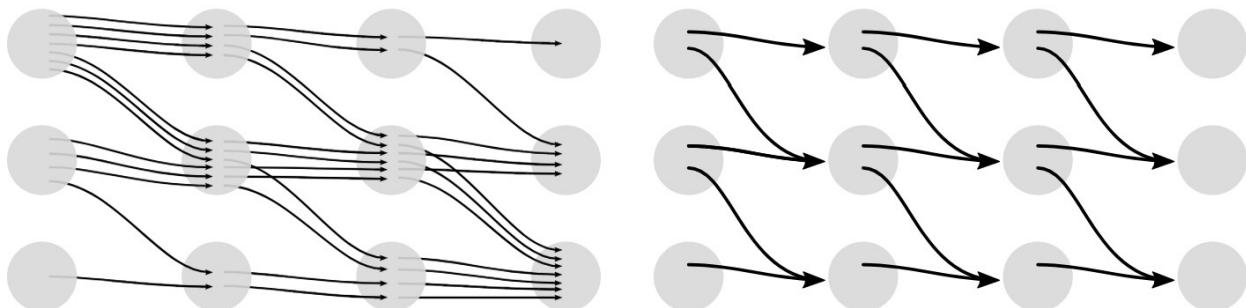
$$p(Y | X) = \sum_{A \in \mathcal{A}_{X,Y}} \prod_{t=1}^T p_t(a_t | X)$$

The CTC conditional probability

marginalizes over the set of valid alignments

computing the **probability** for a single alignment step-by-step.

上面的CTC算法存在性能问题，对于一个时间片长度为 T 的 N 分类任务，所有可能的路径数为 T^N ，在很多情况下，这几乎是一个宇宙级别的数字，用于计算Loss几乎是不现实的。在CTC中采用了动态规划的思想来对查找路径进行剪枝，算法的核心思想是如果路径 π_1 和路径 π_2 在时间片 t 之前的输出均相等，我们就可以提前合并他们，如图6。



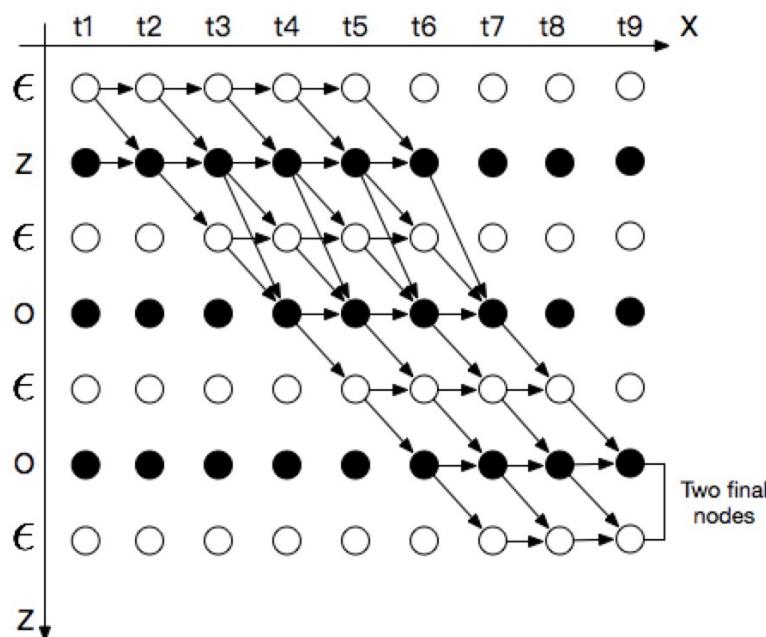
Summing over all alignments can be very expensive.

Dynamic programming merges alignments, so it's much faster.

其中，横轴的单位是 X 的时间片，纵轴的单位是 Y 插入 ϵ 的序列 Z 。例如对于单词“ZOO”，插入 ϵ 后为：

$$Z = \{\epsilon, Z, \epsilon, O, \epsilon, O, \epsilon\}$$

我们用 $\alpha_{s,t}$ 表示路径中已经合并的在横轴单位为 t ，纵轴单位为 s 的节点。根据CTC的对齐方式的三个特征，输入有9个时间片，标签内容是“ZOO”， $P(Y|X)$ 的所有可能的合法路径如下图



上图分成两种情况

Case 1 :

如果 $\alpha_{s,t} = \epsilon$ ，则 $\alpha_{s,t}$ 只能由前一个空格 $\alpha_{s-1,t-1}$ 或者其本身 $\alpha_{s,t-1}$ 得到，如果 $\alpha_{s,t}$ 不等于 ϵ ，但是 $\alpha_{s,t}$ 为连续字符的第二个，即 $\alpha_s = \alpha_{s-2}$ ，则 $\alpha_{s,t}$ 只能由前一个空格 $\alpha_{s-1,t-1}$ 或者其本身 $\alpha_{s,t-1}$ 得到，而不能由前一个字符得到，因为这样做会将连续两个相同的字符合并成一个。 $p_t(z_s|X)$ 表示在时刻 t 输出字符 z_s 的概率。

$$\alpha(s,t) = (\alpha(s,t-1) + \alpha(s-1,t-1)) \cdot p_t(z_s|X)$$

Case 2:

如果 $\alpha_{s,t}$ 不等于 ϵ ，则 $\alpha_{s,t}$ 可以由 $\alpha_{s,t-1}$ ， $\alpha_{s-1,t-1}$ 以及 $\alpha_{s-2,t-1}$ 得来，可以表示为：

$$\alpha(s,t) = (\alpha(s,t-1) + \alpha(s-1,t-1) + \alpha(s-2,t-1)) \cdot p_t(z_s|X)$$

从图7中我们可以看到，合法路径有两个起始点，合法路径的概率 $p(Y|X)$ 是两个final nodes 的概率之和。

现在，我们已经可以高效的计算损失函数，下一步的工作便是计算梯度用于训练模型。由于 $P(Y|X)$ 的计算只涉及加法和乘法，因此其一定是可导函数，进而我们可以使用SGD优化模型。

对于数据集 D ，模型的优化目标是最小化负对数似然

$$\sum_{(X,Y) \in D} -\log p(Y|X)$$

1.3 预测

当我们训练好一个RNN模型时，给定一个输入序列 X ，我们需要找到最可能的输出，也就是求解

$$Y^* = \arg \max_Y p(Y|X)$$

求解最可能的输出有两种方案，一种是Greedy Search，第二种是beam search

1.3.1 Greedy Search

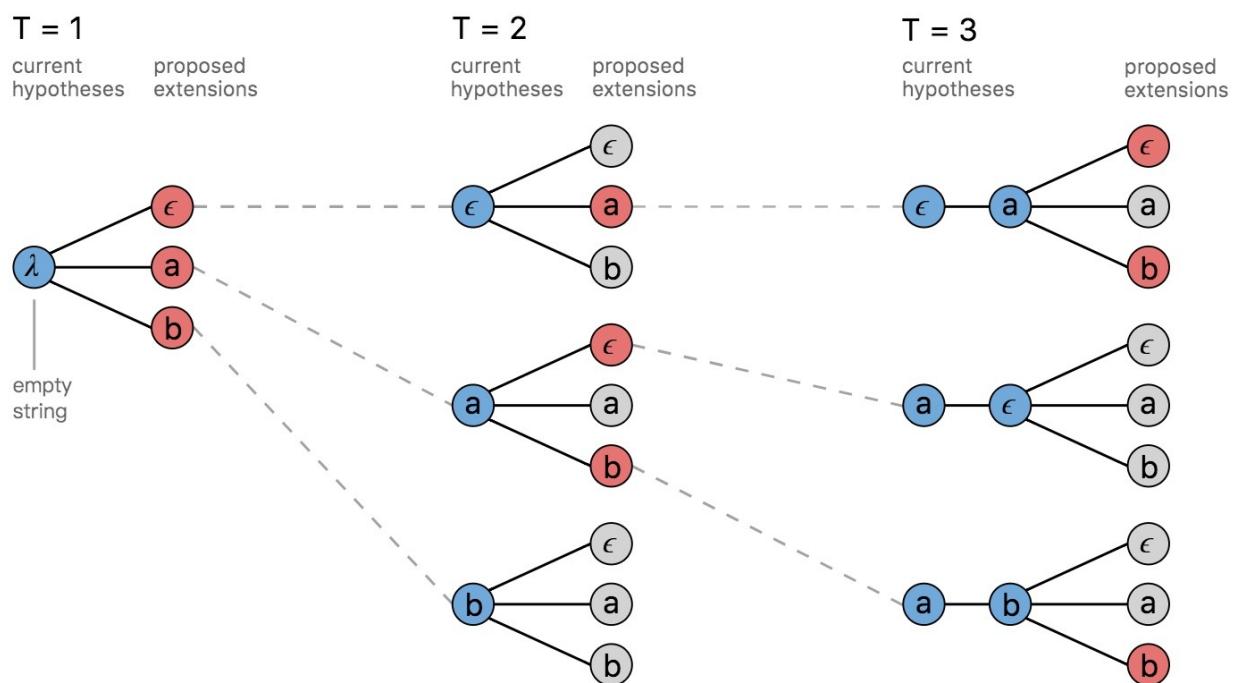
每个时间片均取该时间片概率最高的节点作为输出：

$$A^* = \arg \max_A \prod_{t=1}^T p_t(a_t|X)$$

这个方法最大的缺点是忽略了一个输出可能对应多个对齐方式。

1.3.2 Beam Search

Beam Search是寻找全局最优值和Greedy Search在查找时间和模型精度的一个折中。一个简单的beam search在每个时间片计算所有可能假设的概率，并从中选出最高的几个作为一组。然后再从这组假设的基础上产生概率最高的几个作为一组假设，依次进行，直到达到最后一个时间片，下图是beam search的宽度为3的搜索过程，红线为选中的假设。



A standard beam search algorithm with an alphabet of $\{\epsilon, a, b\}$ and a beam size of three.

CTC的特征

1. 条件独立：CTC的一个非常不合理的假设是其假设每个时间片都是相互独立的，这是一个非常不好的假设。在OCR或者语音识别中，各个时间片之间是含有一些语义信息的，所以如果能够在CTC中加入语言模型的话效果应该会有提升。
2. 单调对齐：CTC的另外一个约束是输入 X 与输出 Y 之间的单调对齐，在OCR和语音识别中，这种约束是成立的。但是在一些场景中例如机器翻译，这个约束便无效了。
3. 多对一映射：CTC的又一个约束是输入序列 X 的长度大于标签数据 Y 的长度，但是对于 Y 的长度大于 X 的长度的场景，CTC便失效了。

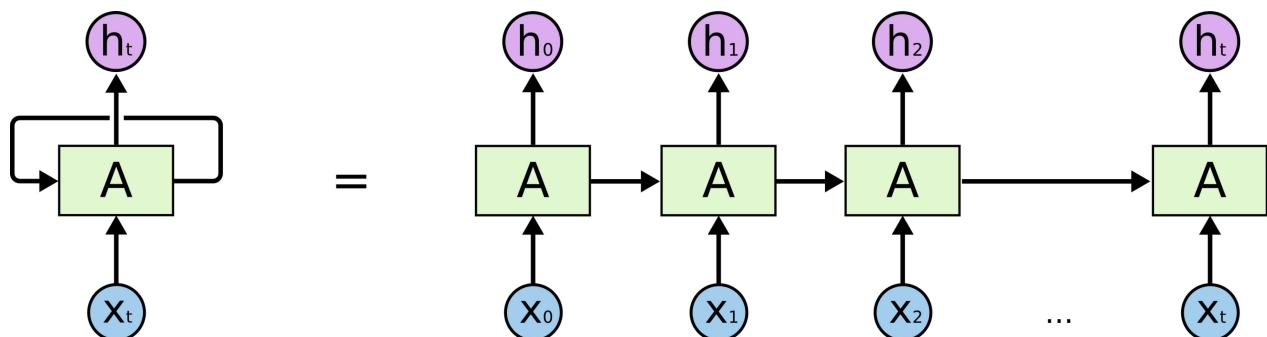
About Long Short Term Memory

1. 背景

Recurrent Neural Networks (RNN)

在使用深度学习处理时序问题时，RNN是最常使用的模型之一。RNN之所以在时序数据上有着优异的表现是因为RNN在 t 时间片时会将 $t-1$ 时间片的隐节点作为当前时间片的输出，也就是RNN具有图1的结构。这样有效的原因是之前时间片的信息也用于计算当前时间片的内容，而传统模型的隐节点的输出只取决于当前时间片的输入特征。

图1：RNN的链式结构



RNN的数学表达式可以表示为

$$h_t = \sigma(x_t * w_{xt} + h_{t-1} * w_{ht} + b)$$

而传统的DNN的隐节点表示为

$$h_t = \sigma(x_t * w_{xt} + b)$$

对比RNN和DNN的隐节点的计算方式，我们发现唯一不同之处在于RNN将上个时间片的隐节点状态 h_{t-1} 也作为了神经网络单元的输入，这也是RNN删除处理时序数据最重要的原因。

所以，RNN的隐节点 h_{t-1} 有两个作用

1. 计算在该时刻的预测值 \hat{y}_t
2. 计算下个时间片的隐节点状态 h_t

RNN的该特性也使RNN在很多学术和工业前景，例如OCR，语音识别，股票预测等领域上有了十足的进展。

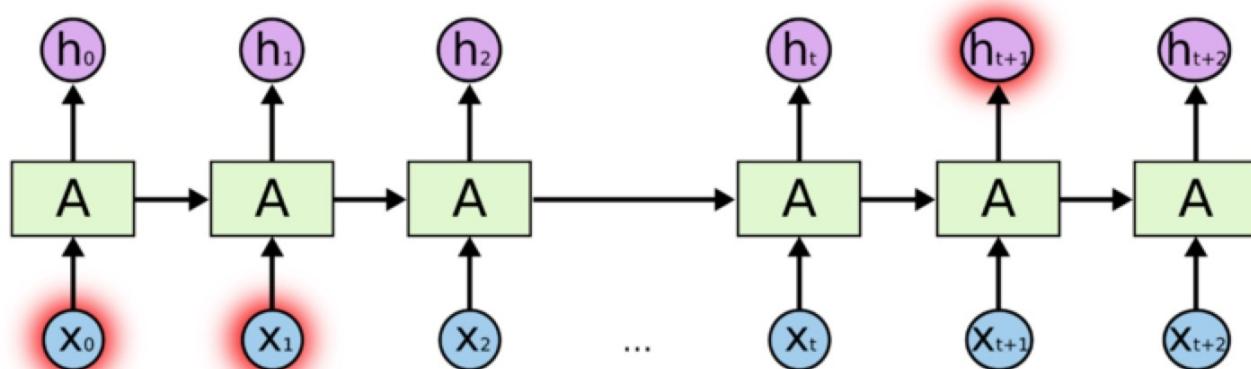
长期依赖(Long Term Dependencies)

在深度学习领域中（尤其是RNN），“长期依赖”问题是普遍存在的。长期依赖产生的原因是当神经网络的节点经过许多阶段的计算后，之前比较长的时间片的特征已经被覆盖，例如下面例子

```
eg1: The cat, which already ate a bunch of food, was full.
| | | | | | | | | |
t0 t1 t2 t3 t4 t5 t6 t7 t8 t9 t10
eg2: The cats, which already ate a bunch of food, were full.
| | | | | | | | | |
t0 t1 t2 t3 t4 t5 t6 t7 t8 t9 t10
```

我们想预测'full'之前系动词的单复数情况，显然full是取决于第二个单词'cat'的单复数情况，而非其前面的单词food。根据图1展示的RNN的结构，随着数据时间片的增加，RNN丧失了学习连接如此远的信息的能力（图2）。

图2：RNN的长期依赖问题



梯度消失/爆炸

梯度消失和梯度爆炸是困扰RNN模型训练的关键原因之一，产生梯度消失和梯度爆炸是由于RNN的权值矩阵循环相乘导致的，相同函数的多次组合会导致极端的非线性行为。梯度消失和梯度爆炸主要存在RNN中，因为RNN中每个时间片使用相同的权值矩阵。对于一个DNN，虽然也涉及多个矩阵的相乘，但是通过精心设计权值的比例可以避免梯度消失和梯度爆炸的问题[75]。

处理梯度爆炸可以采用梯度截断的方法。所谓梯度截断是指将梯度值超过阈值 θ 的梯度手动降到 θ 。虽然梯度截断会一定程度上改变梯度的方向，但梯度截断的方向依旧是朝向损失函数减小的方向。

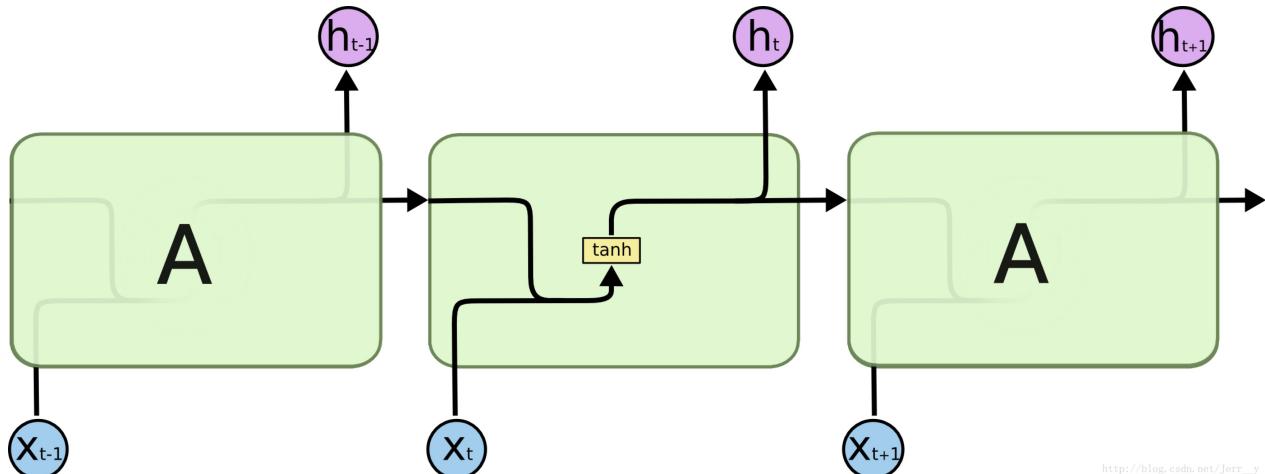
对比梯度爆炸，梯度消失不能简单的通过类似梯度截断的阈值式方法来解决，因为长期依赖的现象也会产生很小的梯度。在上面例子中，我们希望 t_9 时刻能够读到 t_1 时刻的特征，在这期间内我们自然不希望隐层节点状态发生很大的变化，所以 $[t_2, t_8]$ 时刻的梯度要尽可能的小才能保证梯度变化小。很明显，如果我们刻意提高小梯度的值将会使模型失去捕捉长期依赖的能力。

2. LSTM

LSTM的全称是Long Short Term Memory，顾名思义，它具有记忆长短期信息的能力的神经网络。LSTM首先在1997年由Hochreiter & Schmidhuber [76] 提出，由于深度学习在2012年的兴起，LSTM又经过了若干代大牛¹的发展，由此便形成了比较系统且完整的LSTM框架，并且在很多领域得到了广泛的应用。本文着重介绍深度学习时代的LSTM。

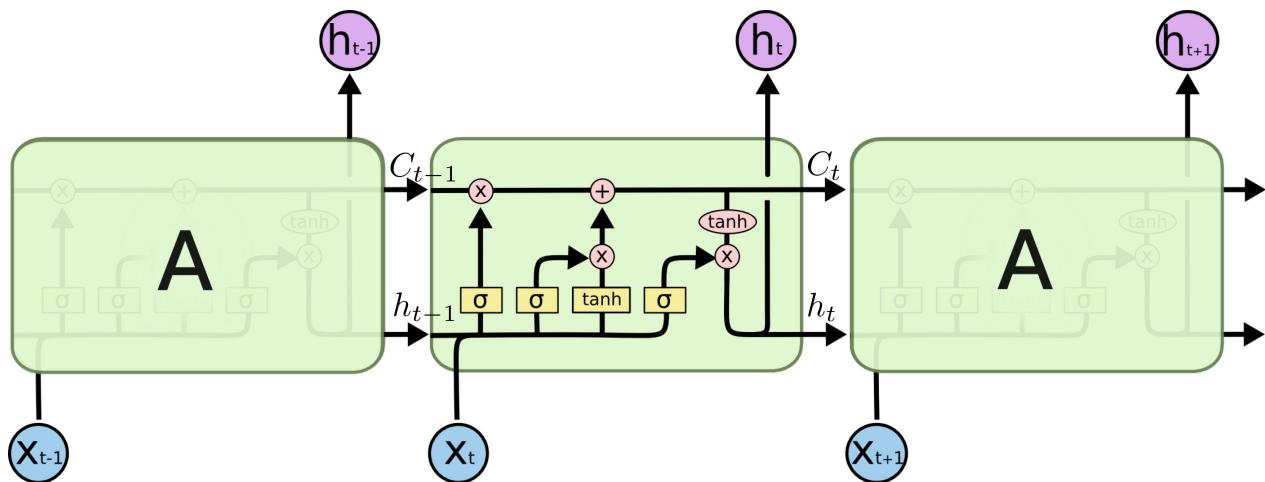
LSTM提出的动机是为了解决上面我们提到的长期依赖问题。传统的RNN节点输出仅由权值，偏置以及激活函数决定（图3）。RNN是一个链式结构，每个时间片使用的是相同的参数。

图3：RNN单元



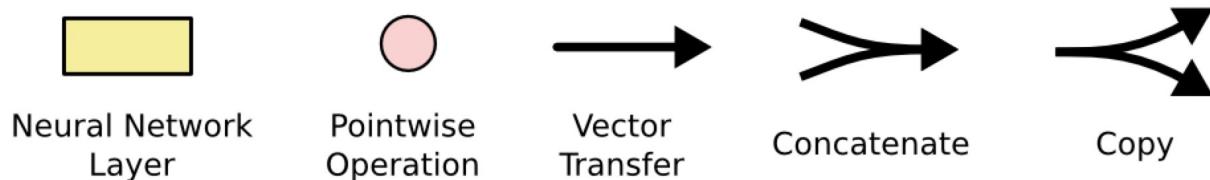
而LSTM之所以能够解决RNN的长期依赖问题，是因为LSTM引入了门（gate）机制用于控制特征的流通和损失。对于上面的例子，LSTM可以做到在 t_9 时刻将 t_2 时刻的特征传过来，这样就可以非常有效的判断 t_9 时刻使用单数还是负数了。LSTM是由一系列LSTM单元（LSTM Unit）组成，其链式结构如下图。

图4：LSTM单元



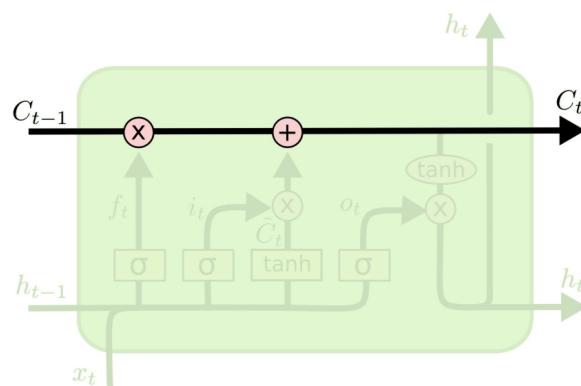
在后面的章节中我们再对LSTM的详细结构进行讲解，首先我们先弄明白LSTM单元中的每个符号的含义。每个黄色方框表示一个神经网络层，由权值，偏置以及激活函数组成；每个粉色圆圈表示元素级别操作；箭头表示向量流向；相交的箭头表示向量的拼接；分叉的箭头表示向量的复制。总结如图5.

图5：LSTM的符号含义



LSTM的核心部分是在图4中最上边类似于传送带的部分（图6），这一部分一般叫做单元状态（cell state）它自始至终存在于LSTM的整个链式系统中。

图6：LSTM的单元状态

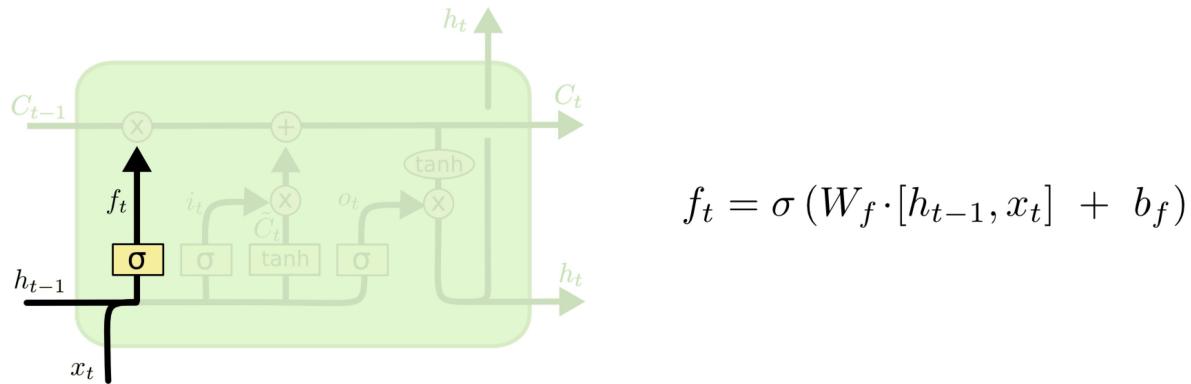


其中

$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$$

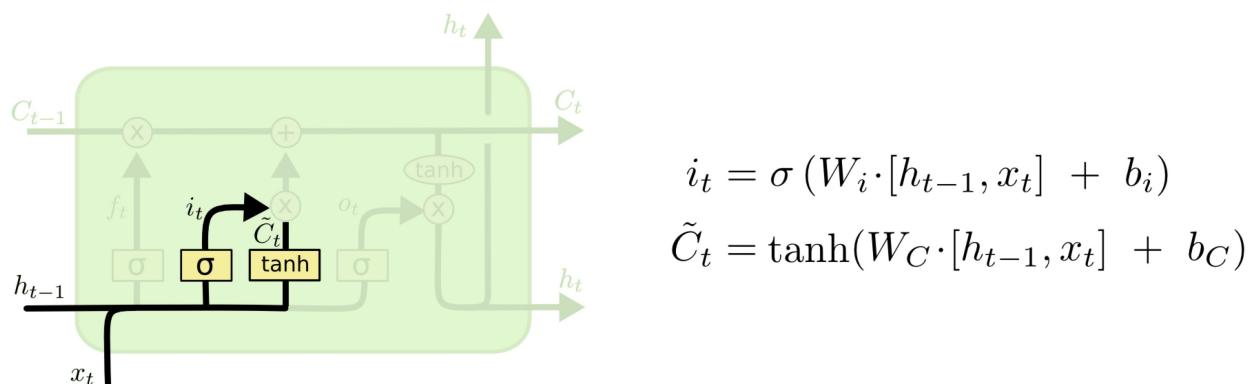
其中 f_t 叫做遗忘门，表示 C_{t-1} 的哪些特征被用于计算 C_t 。 f_t 是一个向量，向量的每个元素均位于 $[0, 1]$ 范围内，向量的维度）。通常我们使用 Sigmoid 作为激活函数，sigmoid 的输出是一个介于 $[0, 1]$ 区间内的值，但是当你观察一个训练好的 LSTM 时，你会发现门的值绝大多数都非常接近 0 或者 1，其余的值少之又少。其中 \otimes 是 LSTM 最重要的门机制，表示 f_t 和 C_{t-1} 之间的单位乘的关系。

图 7：LSTM 的遗忘门



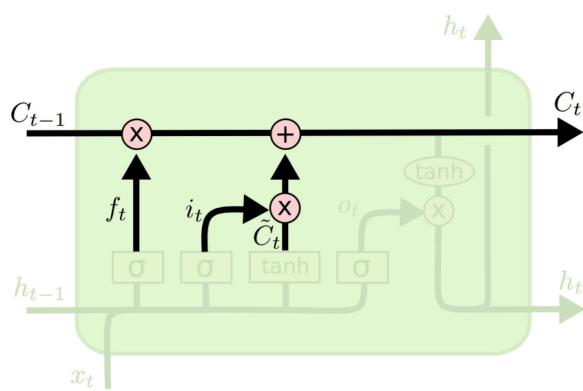
如图 8 所示， \tilde{C}_t 表示单元状态更新值，由输入数据 x_t 和隐节点 h_{t-1} 经由一个神经网络层得到，单元状态更新值的激活函数通常使用 \tanh 。 i_t 叫做输入门，同 f_t 一样也是一个元素介于 $[0, 1]$ 区间内的向量，同样由 x_t 和 h_{t-1} 经由 Sigmoid 激活函数计算而成。

图 8：LSTM 的输入门和单元状态更新值的计算方式



i_t 用于控制 \tilde{C}_t 的哪些特征用于更新 C_t ，使用方式和 f_t 相同（图 9）。

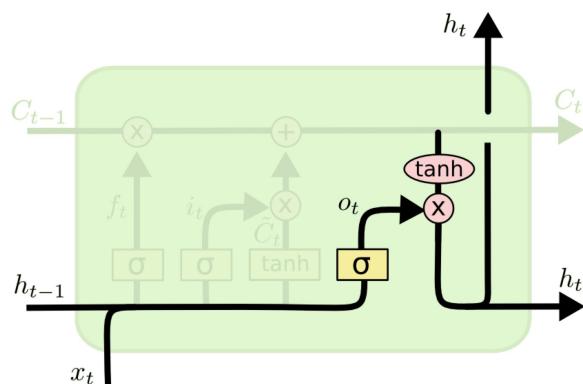
图 9：LSTM 的输入门的使用方法



$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$$

最后，为了计算预测值 \hat{y}_t 和生成下个时间片完整的输入，我们需要计算隐节点的输出 h_t （图10）。

图10：LSTM的输出门



$$o_t = \sigma (W_o [h_{t-1}, x_t] + b_o)$$

$$h_t = o_t * \tanh (C_t)$$

h_t 由输出门 o_t 和单元状态 C_t 得到，其中 o_t 的计算方式和 f_t 以及 i_t 相同。在[74]的论文中指出，通过将 b_o 的均值初始化为1，可以使LSTM达到同GRU近似的效果。

3. 其他LSTM

联想之前介绍的GRU [81]，LSTM的隐层节点的门的数量和工作方式貌似是非常灵活的，那么是否存在一个最好的结构模型或者比LSTM和GRU性能更好的模型呢？Rafal [73]等人采集了能采集到的100个最好模型，然后在这100个模型的基础上通过变异的形式产生了10000个新的模型。然后通过在字符串，结构化文档，语言模型，音频4个场景的实验比较了这10000多个模型，得出的重要结论总结如下：

1. GRU，LSTM是表现最好的模型；
2. GRU的在除了语言模型的场景中表现均超过LSTM；
3. LSTM的输出门的偏置的均值初始化为1时，LSTM的性能接近GRU；
4. 在LSTM中，门的重要性排序是遗忘门 > 输入门 > 输出门。

¹. Felix Gers, Fred Cummins, Santiago Fernandez, Justin Bayer, Daan Wierstra, Julian Togelius, Faustino Gomez, Matteo Gagliolo, and Alex Graves ↵

Attention Is All You Need

tags: NLP, Attention

前言

注意力（Attention）机制 由Bengio团队与2014年提出并在近年广泛的应用在深度学习中的各个领域[88]，例如在计算机视觉方向用于捕捉图像上的感受野，或者NLP中用于定位关键token或者特征。谷歌团队近期提出的用于生成词向量的BERT [71]算法在NLP的11项任务中取得了效果的大幅提升，堪称2018年深度学习领域最振奋人心的消息。而BERT算法的最重要的部分便是本文中提出的Transformer[72]的概念。

正如论文的题目所说的，Transformer中抛弃了传统的CNN和RNN，整个网络结构完全是由Attention机制组成。更准确地讲，Transformer由且仅由self-Attention和Feed Forward Neural Network组成。一个基于Transformer的可训练的神经网络可以通过堆叠Transformer的形式进行搭建，作者的实验是通过搭建编码器和解码器各6层，总共12层的Encoder-Decoder，并在机器翻译中取得了BLEU值得新高。

作者采用Attention机制的原因是考虑到RNN（或者LSTM，GRU等）的计算限制为是顺序的，也就是说RNN相关算法只能从左向右依次计算或者从右向左依次计算，这种机制带来了两个问题：

1. 时间片 t 的计算依赖 $t-1$ 时刻的计算结果，这样限制了模型的并行能力；
2. 顺序计算的过程中信息会丢失，尽管LSTM等门机制的结构一定程度上缓解了长期依赖的问题，但是对于特别长期的依赖现象,LSTM依旧无能为力。

Transformer的提出解决了上面两个问题，首先它使用了Attention机制，将序列中的任意两个位置之间的距离是缩小为一个常量；其次它不是类似RNN的顺序结构，因此具有更好的并行性，符合现有的GPU框架。论文中给出Transformer的定义是：Transformer is the first transduction model relying entirely on self-attention to compute representations of its input and output without using sequence aligned RNNs or convolution。

遗憾的是，作者的论文比较难懂，尤其是Transformer的结构细节和实现方式并没有解释清楚。尤其是论文中的 Q ， V ， K 究竟代表什么意思作者并没有说明。通过查阅资料，发现了一篇非常优秀的讲解Transformer的技术博客。本文中的大量插图也会从该博客中截取。首先感谢Jay Alammar详细的讲解，其次推荐大家去阅读原汁原味的文章。

1. Transformer 详解

1.1 高层 Transformer

论文中的验证Transformer的实验室基于机器翻译的，下面我们就以机器翻译为例子详细剖析Transformer的结构，在机器翻译中，Transformer可概括为如图1：



图1：Transformer用于机器翻译

Transformer的本质上是一个Encoder-Decoder的结构，那么图1可以表示为图2的结构：

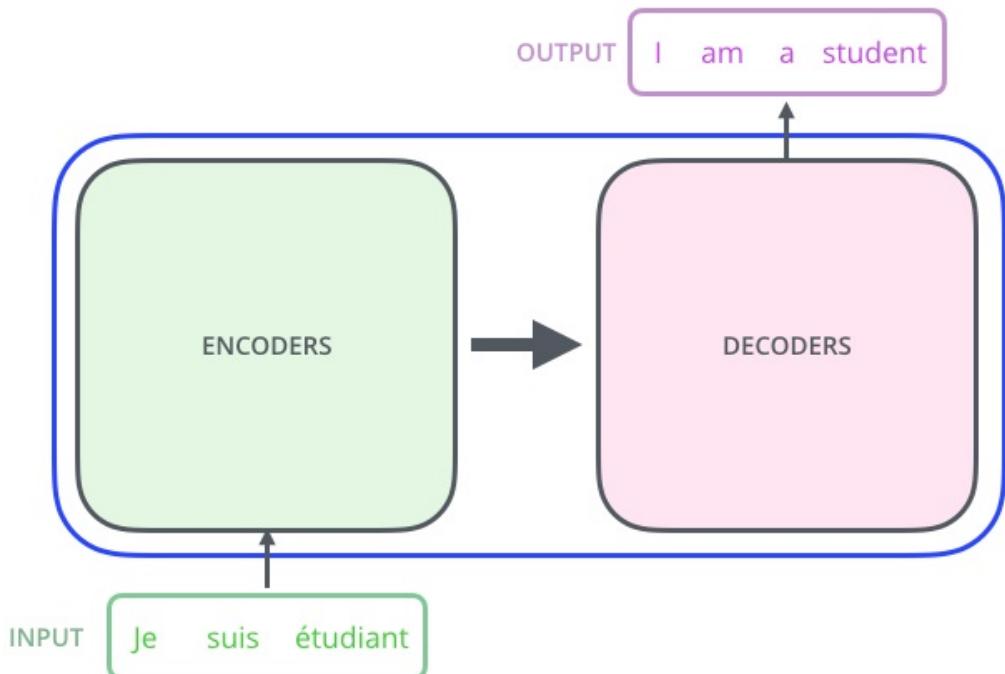


图2：Transformer的Encoder-Decoder结构

如论文中所设置的，编码器由6个编码block组成，同样解码器是6个解码block组成。与所有的生成模型相同的是，编码器的输出会作为解码器的输入，如图3所示：

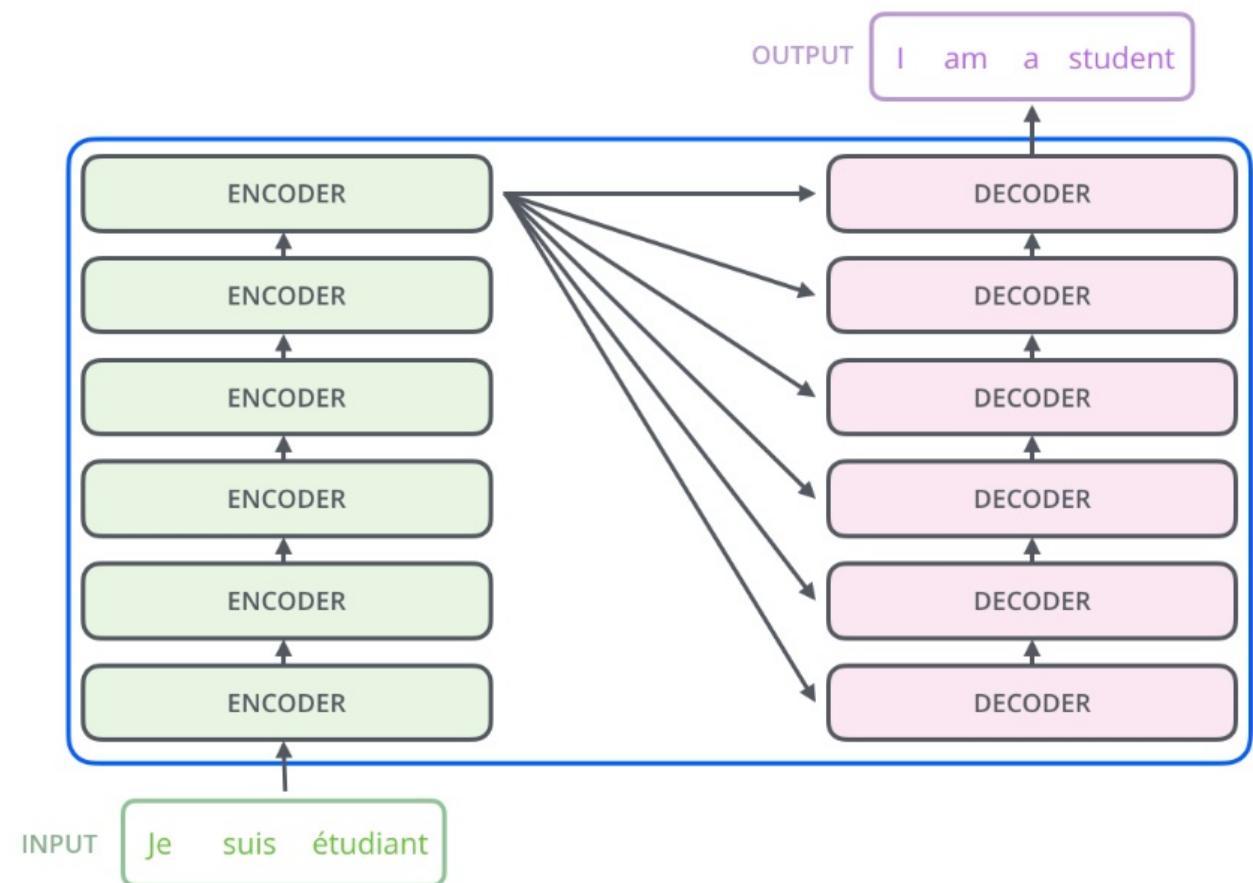


图3：Transformer的Encoder和Decoder均由6个block堆叠而成

我们继续分析每个encoder的详细结构：在Transformer的encoder中，数据首先会经过一个叫做‘self-attention’的模块得到一个加权之后的特征向量Z，这个Z便是论文公式1中的 $\text{Attention}(Q, K, V)$ ：

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

第一次看到这个公式你可能会一头雾水，在后面的文章中我们会揭开这个公式背后的实际含义，在这一段暂时将其叫做Z。

得到Z之后，它会被送到encoder的下一个模块，即Feed Forward Neural Network。这个全连接有两层，第一层的激活函数是ReLU，第二层是一个线性激活函数，可以表示为：

$$\text{FFN}(Z) = \max(0, ZW_1 + b_1)W_2 + b_2$$

Encoder的结构如图4所示：

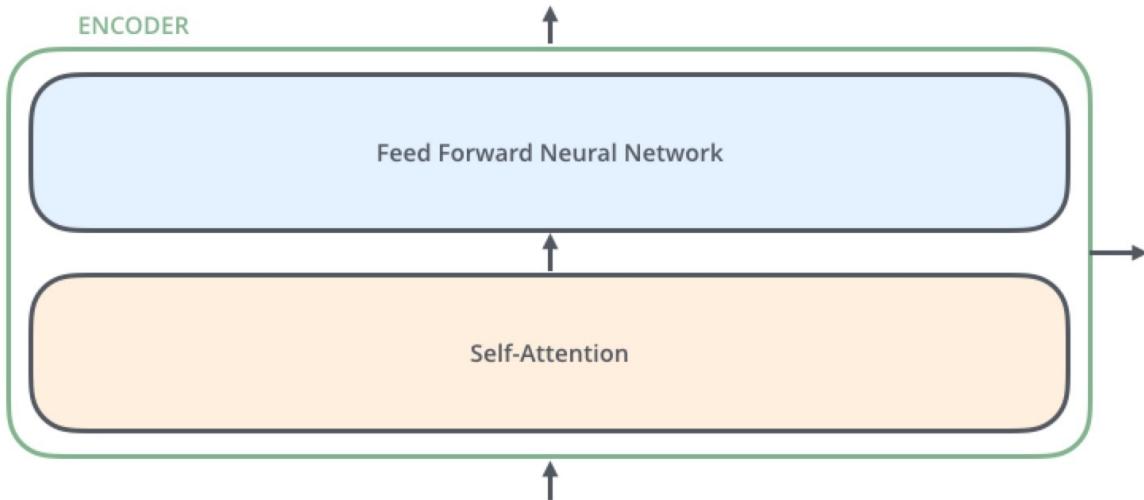


图4：*Transformer*由 *self-attention*和 *Feed Forward neural network*组成

Decoder的结构如图5所示，它和encoder的不同之处在于Decoder多了一个Encoder-Decoder Attention，两个Attention分别用于计算输入和输出的权值：

1. Self-Attention：当前翻译和已经翻译的前文之间的关系；
2. Encoder-Decnoder Attention：当前翻译和编码的特征向量之间的关系。

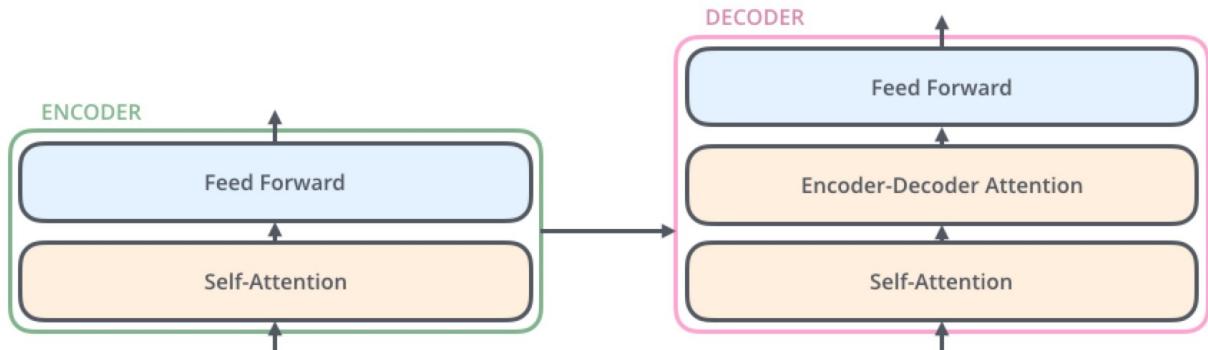


图5：*Transformer*的解码器由 *self-attention*，*encoder-decoder attention*以及 *FFNN*组成

1.2 输入编码

1.1节介绍的就是Transformer的主要框架，下面我们将介绍它的输入数据。如图6所示，首先通过Word2Vec等词嵌入方法将输入语料转化成特征向量，论文中使用的词嵌入的维度为

$$d_{model} = 512$$



图6：单词的输入编码

在最底层的block中， x 将直接作为Transformer的输入，而在其他层中，输入则是上一个block的输出。为了画图更简单，我们使用更简单的例子来表示接下来的过程，如图7所示：

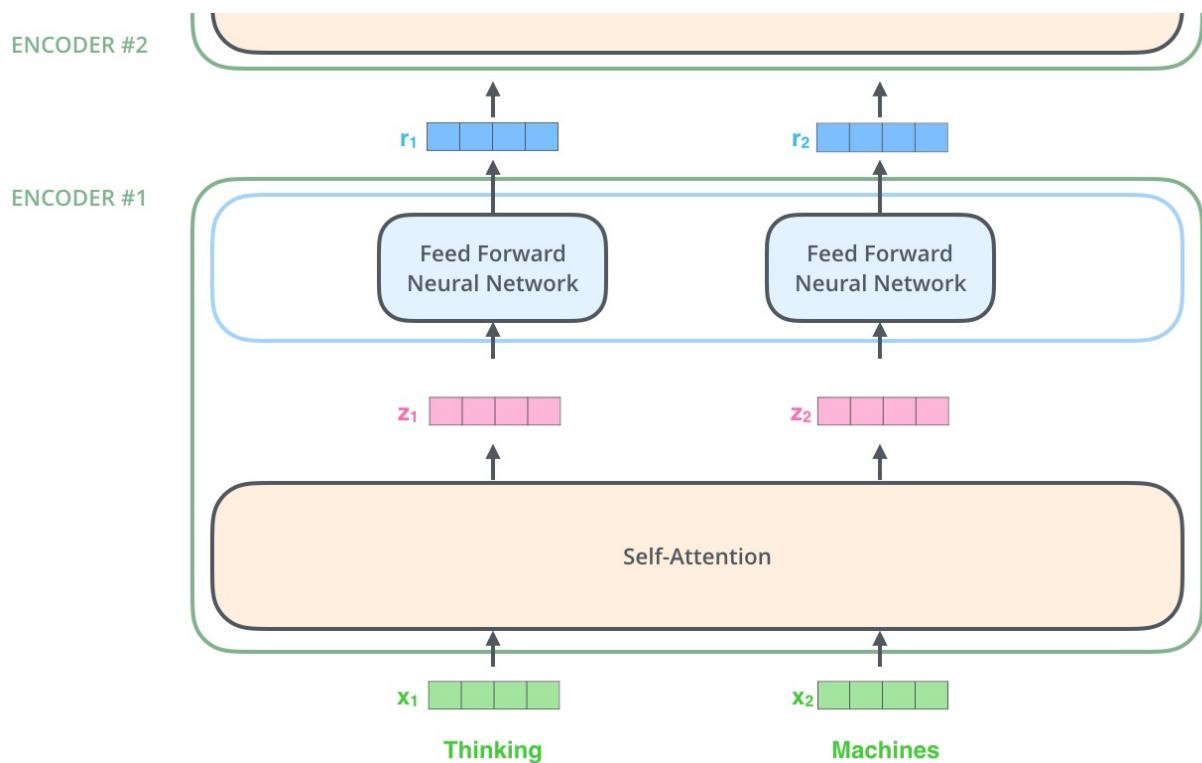


图7：输入编码作为一个tensor输入到encoder中

1.3 Self-Attention

Self-Attention是Transformer最核心的内容，然而作者并没有详细讲解，下面我们来补充一下作者遗漏的地方。回想Bahdanau等人提出的用Attention[2]，其核心内容是为输入向量的每个单词学习一个权重，例如在下面的例子中我们判断it代指的内容，

```
The animal didn't cross the street because it was too tired
```

通过加权之后可以得到类似图8的加权情况，在讲解self-attention的时候我们也会使用图8类似的表现方式

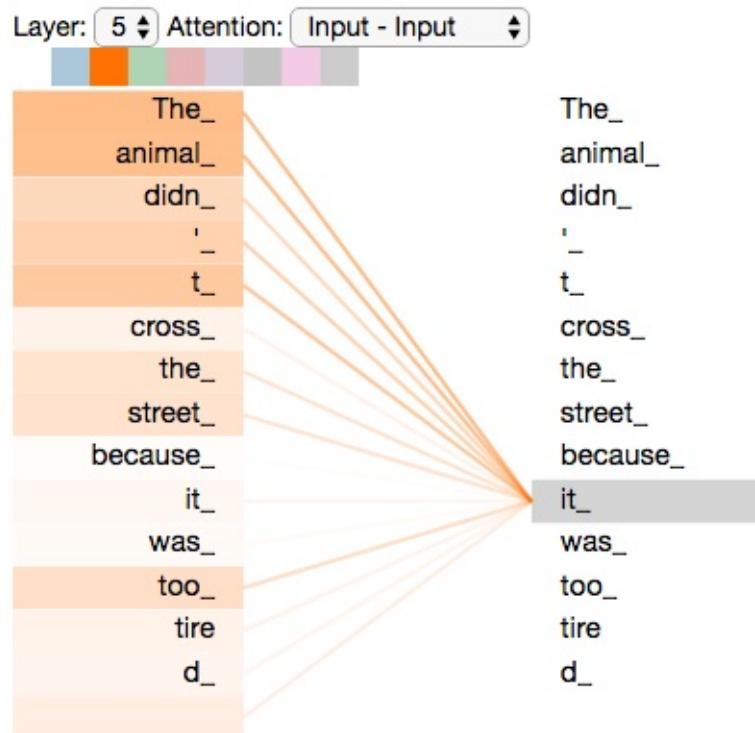
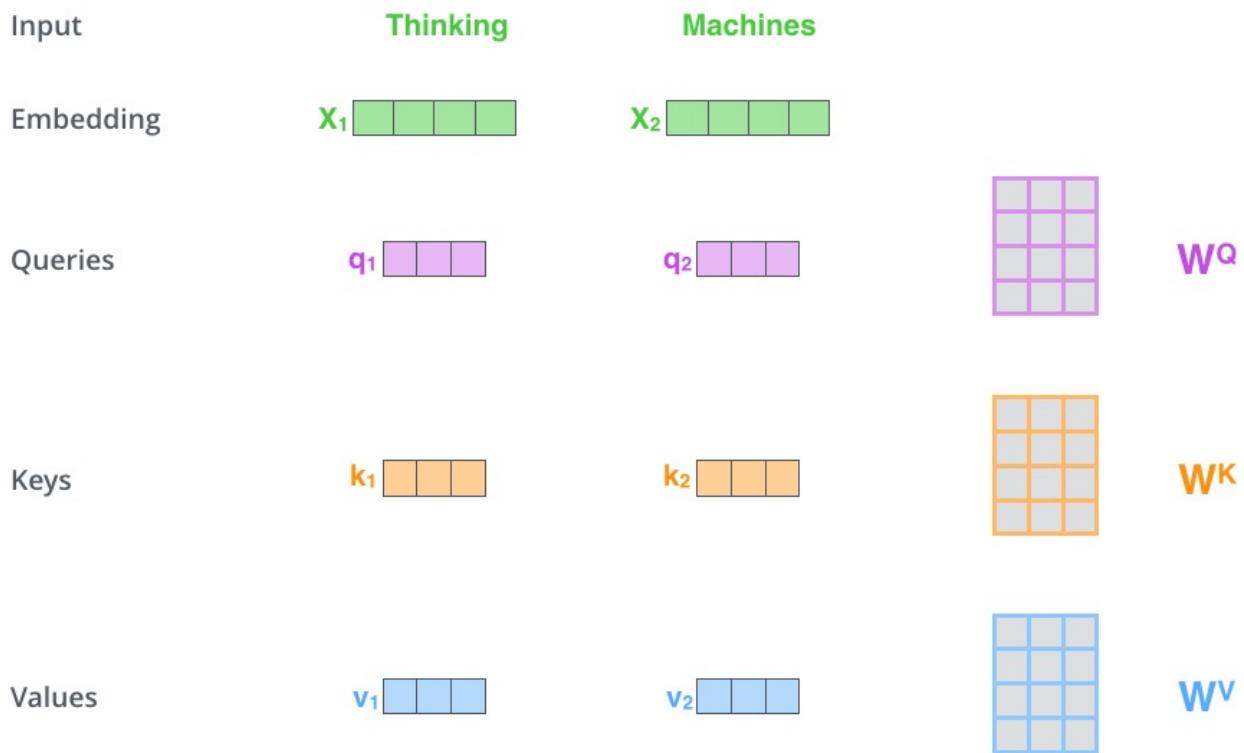


图8：经典Attention可视化示例图

在self-attention中，每个单词有3个不同的向量，它们分别是Query向量（Q），Key向量（K）和Value向量（V），长度均是64。它们是通过3个不同的权值矩阵由嵌入向量 X 乘以三个不同的权值矩阵 W^Q ， W^K ， W^V 得到，其中三个矩阵的尺寸也是相同的。均是 512×64 。

图9： Q ， K ， V 的计算示例图

那么Query，Key，Value是什么意思呢？它们在Attention的计算中扮演着什么角色呢？我们先看一下Attention的计算方法，整个过程可以分成7步：

1. 如上文，将输入单词转化成嵌入向量；
2. 根据嵌入向量得到 q ， k ， v 三个向量；
3. 为每个向量计算一个score： $score = q \cdot k$ ；
4. 为了梯度的稳定，Transformer使用了score归一化，即除以 $\sqrt{d_k}$ ；
5. 对score施以softmax激活函数；
6. softmax点乘Value值 v ，得到加权的每个输入向量的评分 v ；
7. 相加之后得到最终的输出结果 z ： $z = \sum v$ 。

上面步骤的可以表示为图10的形式。

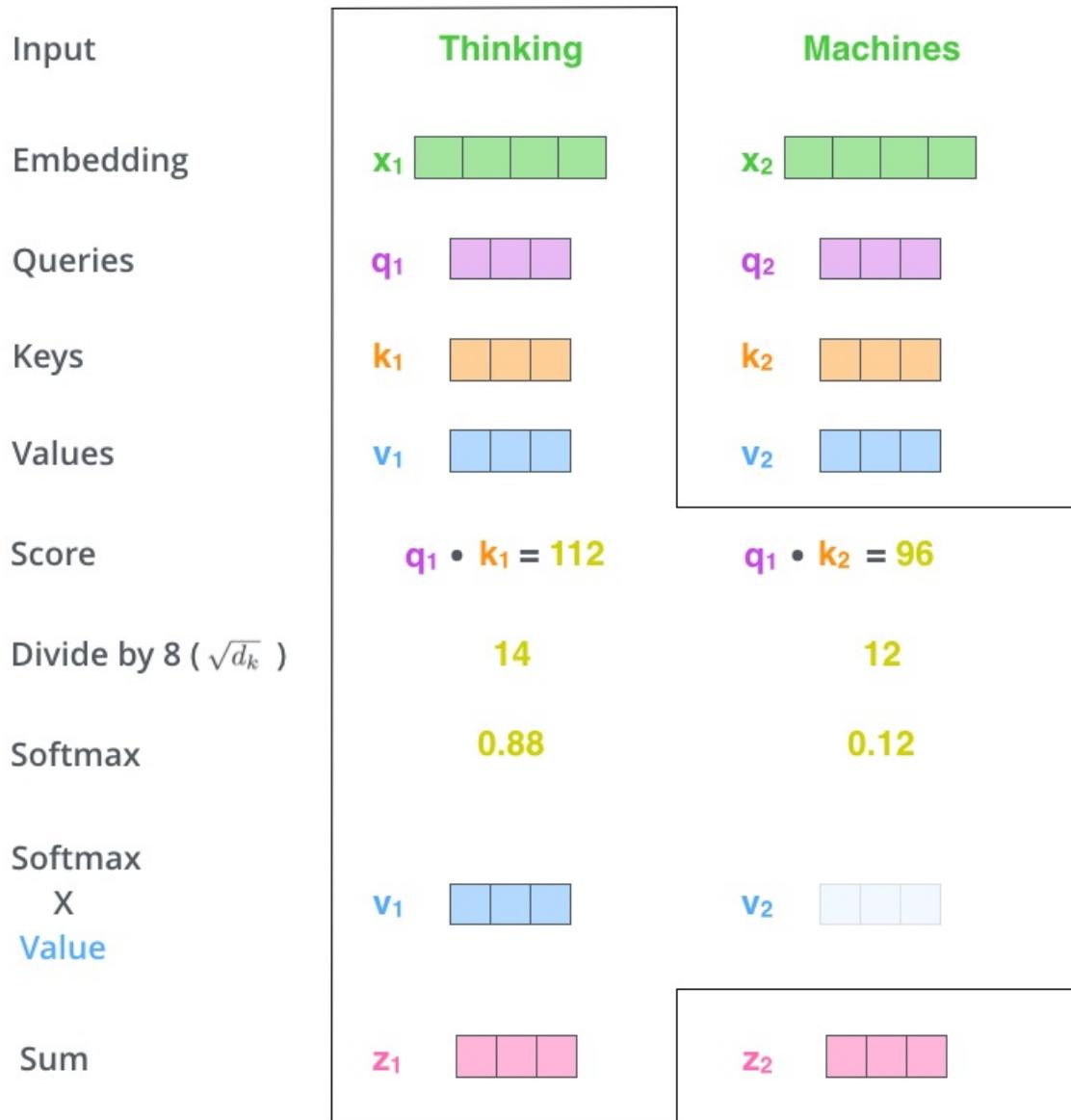


图 10：Self-Attention 计算示例图

实际计算过程中是采用基于矩阵的计算方式，那么论文中的 Q ， V ， K 的计算方式如图11：

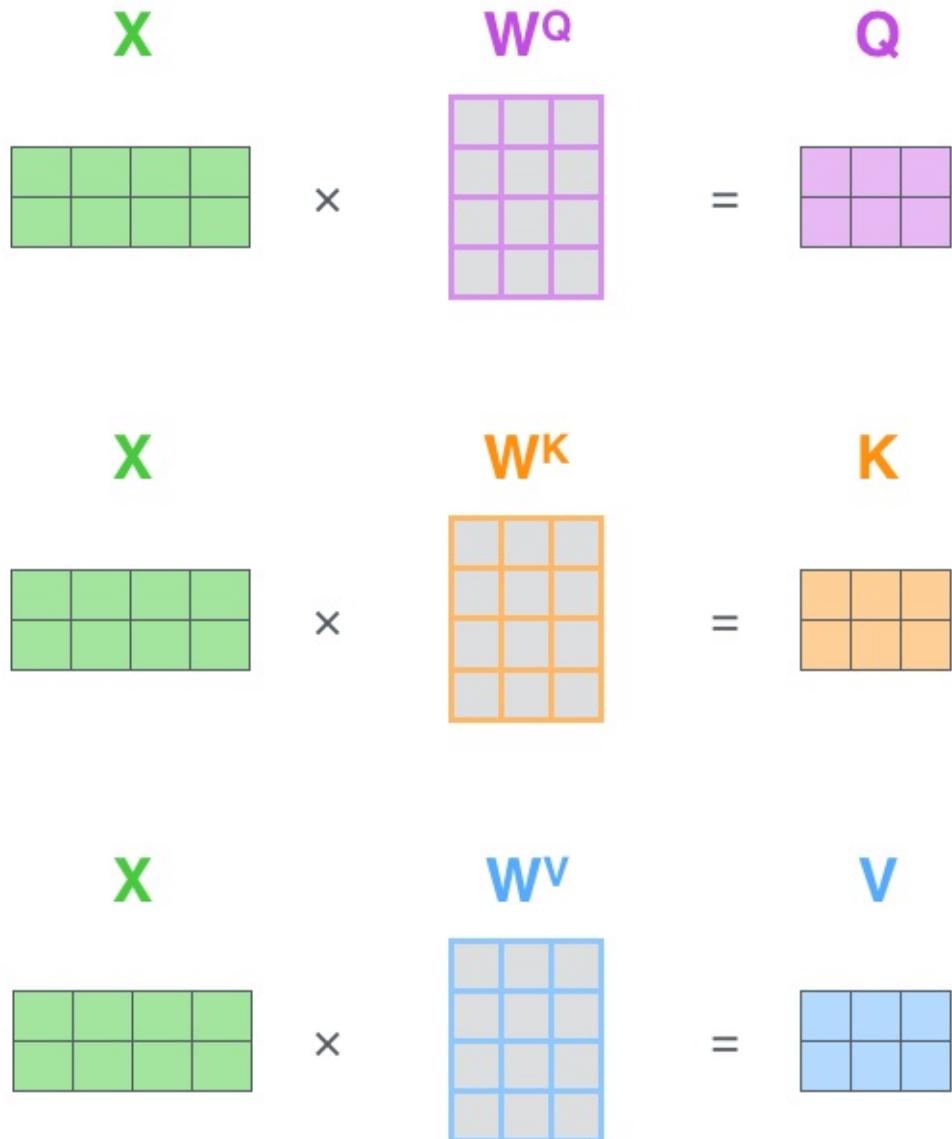
图 11 : Q , V , K 的矩阵表示

图10总结为如图12所示的矩阵形式：

$$\text{softmax} \left(\frac{\text{Q} \times \text{K}^T}{\sqrt{d_k}} \right) \text{V} = \text{Z}$$

Diagram illustrating the matrix representation of Self-Attention. The input matrix Q (purple) is multiplied by the transpose of matrix K (orange) and then scaled by $\sqrt{d_k}$. The result is multiplied by matrix V (blue) to produce the output matrix Z (pink).

图 12 : Self-Attention 的矩阵表示

这里也就是公式1的计算方式。

在self-attention需要强调的最后一点是其采用了[残差网络](#) [118] 中的short-cut结构，目的当然是解决深度学习中的退化问题，得到的最终结果如图13。

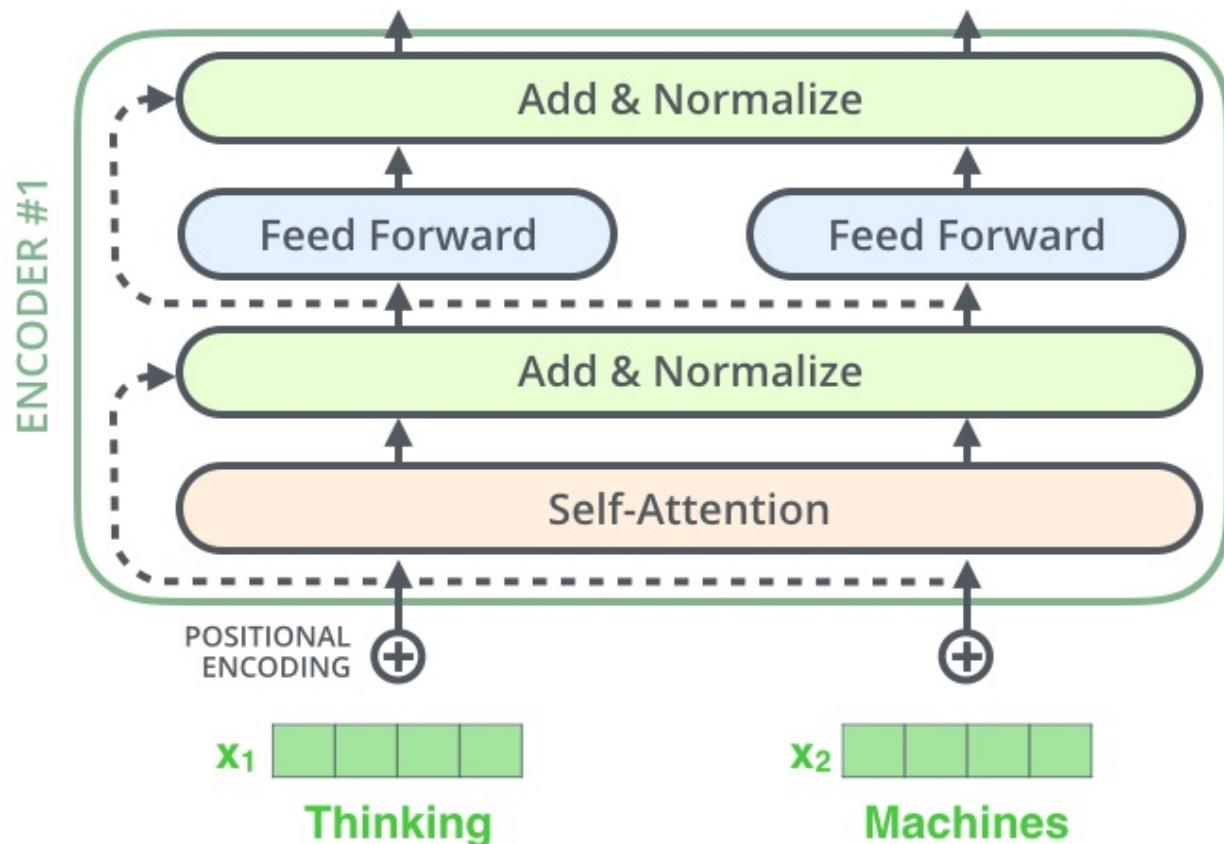


图 13 : Self-Attention 中的 short-cut 连接

1.3 Multi-Head Attention

Multi-Head Attention相当于 h 个不同的self-attention的集成（ensemble），在这里我们以 $h = 8$ 举例说明。Multi-Head Attention的输出分成3步：

1. 将数据 X 分别输入到图13所示的8个self-attention中，得到8个加权后的特征矩阵 $Z_i, i \in \{1, 2, \dots, 8\}$ 。
2. 将8个 Z_i 按列拼成一个大的特征矩阵；
3. 特征矩阵经过一层全连接后得到输出 Z 。

整个过程如图14所示：

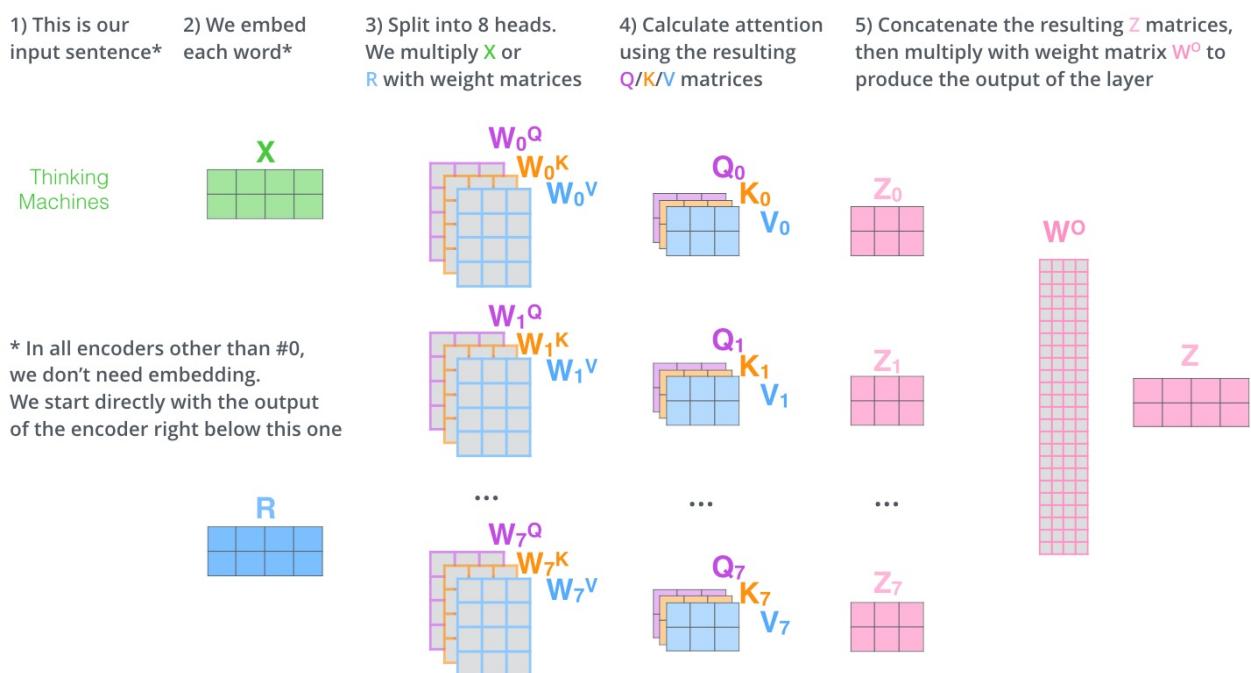


图 14 : Multi-Head Attention

同self-attention一样，multi-head attention也加入了short-cut机制。

1.4 Encoder-Decoder Attention

在解码器中，Transformer block比编码器中多了个encoder-decoder attention。在encoder-decoder attention中， Q 来自与解码器的上一个输出， K 和 V 则来自于与编码器的输出。其计算方式完全和图10的过程相同。

由于在机器翻译中，解码过程是一个顺序操作的过程，也就是当解码第 k 个特征向量时，我们只能看到第 $k-1$ 及其之前的解码结果，论文中把这种情况下的multi-head attention叫做masked multi-head attention。

1.5 损失层

解码器解码之后，解码的特征向量经过一层激活函数为softmax的全连接层之后得到反映每个单词概率的输出向量。此时我们便可以通过CTC等损失函数训练模型了。

而一个完整可训练的网络结构便是encoder和decoder的堆叠（各 N 个， $N = 6$ ），我们可以得到图15中的完整的Transformer的结构（即论文中的图1）：

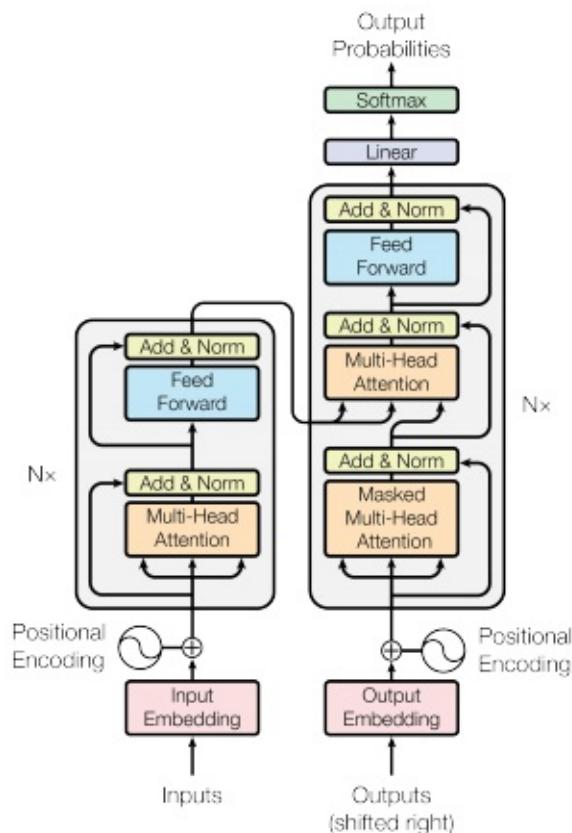


图15：Transformer的完整结构图

2. 位置编码

截止目前为止，我们介绍的Transformer模型并没有捕捉顺序序列的能力，也就是说无论句子的结构怎么打乱，Transformer都会得到类似的结果。换句话说，Transformer只是一个功能更强大的词袋模型而已。

为了解决这个问题，论文中在编码词向量时引入了位置编码（Position Embedding）的特征。具体地说，位置编码会在词向量中加入了单词的位置信息，这样Transformer就能区分不同位置的单词了。

那么怎么编码这个位置信息呢？常见的模式有：a. 根据数据学习；b. 自己设计编码规则。在这里作者采用了第二种方式。那么这个位置编码该是什么样子呢？通常位置编码是一个长度为 d_{model} 的特征向量，这样便于和词向量进行单位加的操作，如图 16。

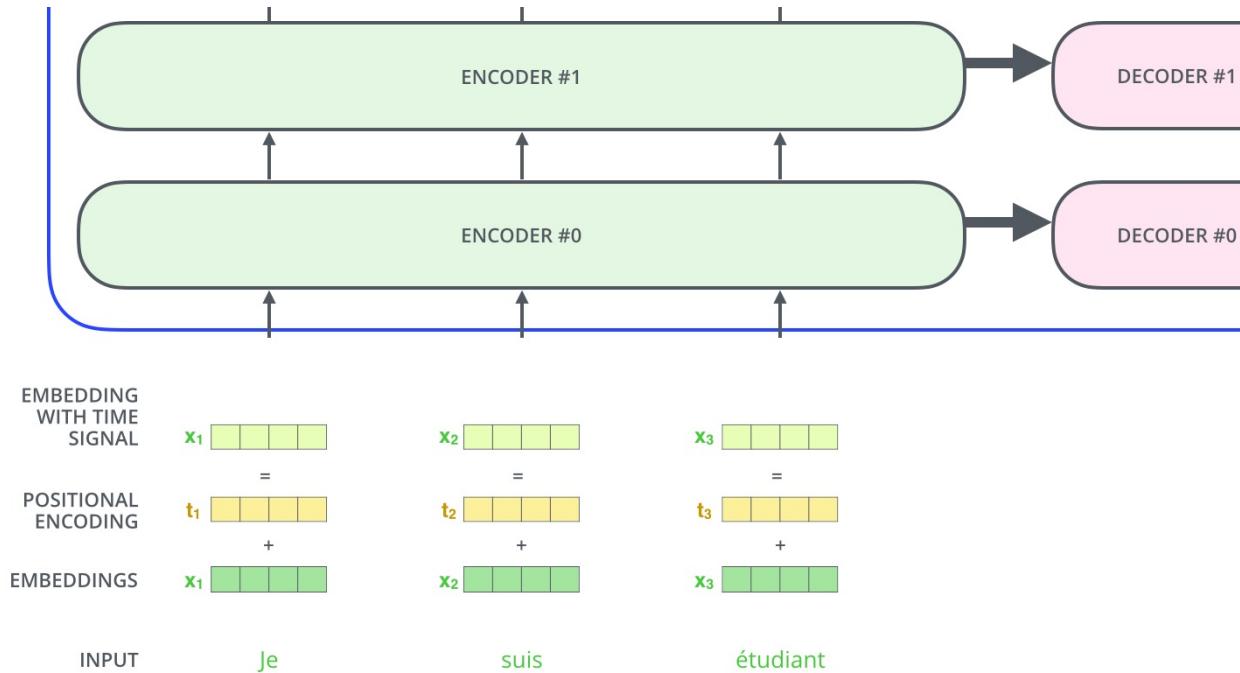


图 16 : Position Embedding

论文给出的编码公式如下：

$$PE(pos, 2i) = \sin\left(\frac{pos}{10000^{\frac{2i}{d_{model}}}}\right)$$

$$PE(pos, 2i + 1) = \cos\left(\frac{pos}{10000^{\frac{2i}{d_{model}}}}\right)$$

在上式中， pos 表示单词的位置， i 表示单词的维度。关于位置编码的实现可在 Google 开源的算法中 [get_timing_signal_1d\(\)](#) 函数找到对应的代码。

作者这么设计的原因是考虑到在 NLP 任务中，除了单词的绝对位置，单词的相对位置也非常重要。根据公式 $\sin(\alpha + \beta) = \sin\alpha\cos\beta + \cos\alpha\sin\beta$ 以及 $\cos(\alpha + \beta) = \cos\alpha\cos\beta - \sin\alpha\sin\beta$ ，这表明位置 $k + p$ 的位置向量可以表示为位置 k 的特征向量的线性变化，这为模型捕捉单词之间的相对位置关系提供了非常大的便利。

3. 总结

优点：（1）虽然Transformer最终也没有逃脱传统学习的套路，Transformer只是一个全连接（或者是一维卷积）加Attention的结合体。但是其设计已经足够有创新，因为其抛弃了在NLP中最根本的RNN或者CNN并且取得了非常不错的效果，算法的设计非常精彩，值得每个深度学习的相关人员仔细研究和品味。（2）Transformer的设计最大的带来性能提升的关键是将任意两个单词的距离是1，这对解决NLP中棘手的长期依赖问题是非常有效的。（3）Transformer不仅仅可以应用在NLP的机器翻译领域，甚至可以不局限于NLP领域，是非常有科研潜力的一个方向。（4）算法的并行性非常好，符合目前的硬件（主要指GPU）环境。

缺点：（1）粗暴的抛弃RNN和CNN虽然非常炫技，但是它也使模型丧失了捕捉局部特征的能力，RNN + CNN + Transformer的结合可能会带来更好的效果。（2）Transformer失去的位置信息其实在NLP中非常重要，而论文中在特征向量中加入Position Embedding只是一个权宜之计，并没有改变Transformer结构上的固有缺陷。

BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding

tags: NLP, Transformer, BERT

前言

BERT（Bidirectional Encoder Representations from Transformers）[71]近期提出之后，作为一个Word2Vec的替代者，其在NLP领域的11个方向大幅刷新了精度，可以说是近年来自残差网络最优突破性的一项技术了。论文的主要特点以下几点：

1. 使用了Transformer [72]作为算法的主要框架，Transformer能更彻底的捕捉语句中的双向关系；
2. 使用了Mask Language Model(MLM) [70] 和 Next Sentence Prediction(NSP) 的多任务训练目标；
3. 使用更强大的机器训练更大规模的数据，使BERT的结果达到了全新的高度，并且Google开源了BERT模型，用户可以直接使用BERT作为Word2Vec的转换矩阵并高效的将其应用到自己的任务中。

BERT的本质上是通过在海量的语料的基础上运行自监督学习方法为单词学习一个好的特征表示，所谓自监督学习是指在没有人工标注的数据上运行的监督学习。在以后特定的NLP任务中，我们可以直接使用BERT的特征表示作为该任务的词嵌入特征。所以BERT提供的是一个供其它任务迁移学习的模型，该模型可以根据任务微调或者固定之后作为特征提取器。BERT的源码和模型10月31号已经在Github上[开源](#)，简体中文和多语言模型也于11月3号开源。

1. BERT 详解

1.1 网络架构

BERT的网络架构使用的是《Attention is all you need》中提出的多层Transformer结构，其最大的特点是抛弃了传统的RNN和CNN，通过Attention机制将任意位置的两个单词的距离转换成1，有效的解决了NLP中棘手的长期依赖问题。Transformer的结构在NLP领域中已经得到了广泛应用，并且作者已经发布在TensorFlow的[tensor2tensor](#)库中。

Transformer的网络架构如图1所示，Transformer是一个encoder-decoder的结构，由若干个编码器和解码器堆叠形成。图1的左侧部分为编码器，由Multi-Head Attention和一个全连接组成，用于将输入语料转化成特征向量。右侧部分是解码器，其输入为编码器的输出以及已经

预测的结果，由Masked Multi-Head Attention, Multi-Head Attention以及一个全连接组成，用于输出最后结果的条件概率。关于Transformer的详细解析参考我之前总结的[文档](#)。

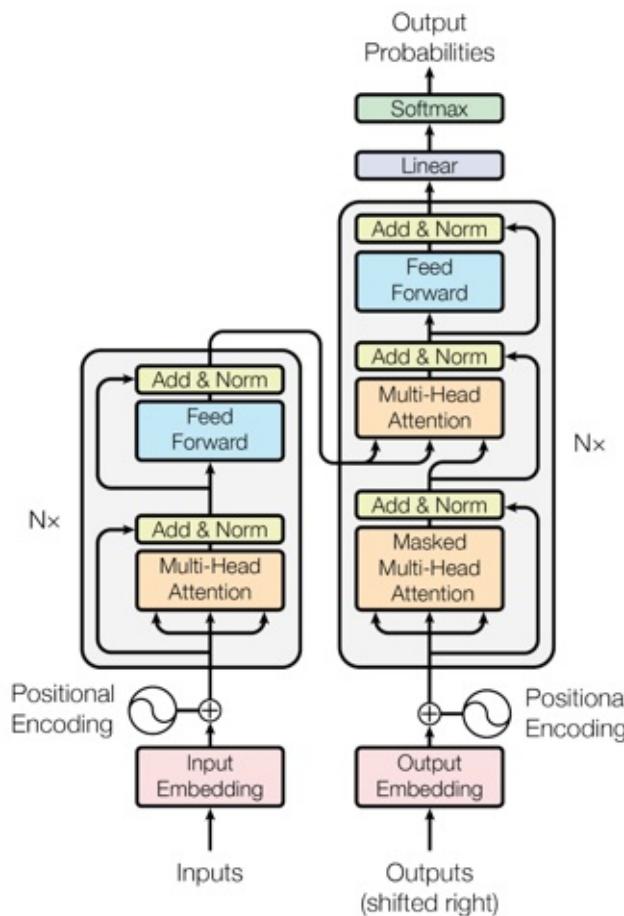


图1：BERT中采用的Transformer网络

图1中的左侧部分是一个Transformer Block，对应到图2中的一个“Trm”。

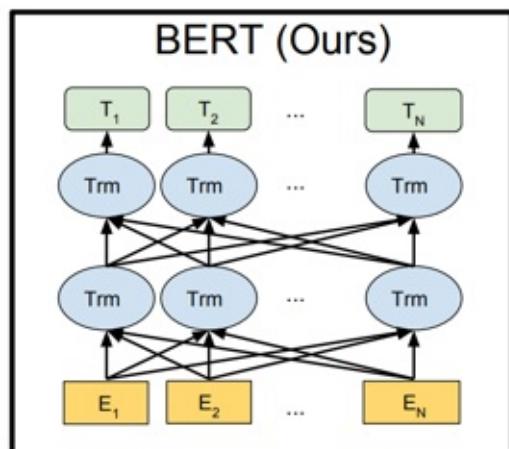


图2：BERT的网络结构

BERT提供了简单和复杂两个模型，对应的超参数分别如下：

- **BERT_{BASE}**: L=12, H=768, A=12, 参数总量110M；
- **BERT_{LARGE}**: L=24, H=1024, A=16, 参数总量340M；

在上面的超参数中，L表示网络的层数（即Transformer blocks的数量），A表示Multi-Head Attention中self-Attention的数量，filter的尺寸是4H。

论文中还对比了BERT和GPT[69]和ELMo[68]，它们两个的结构图如图3所示。

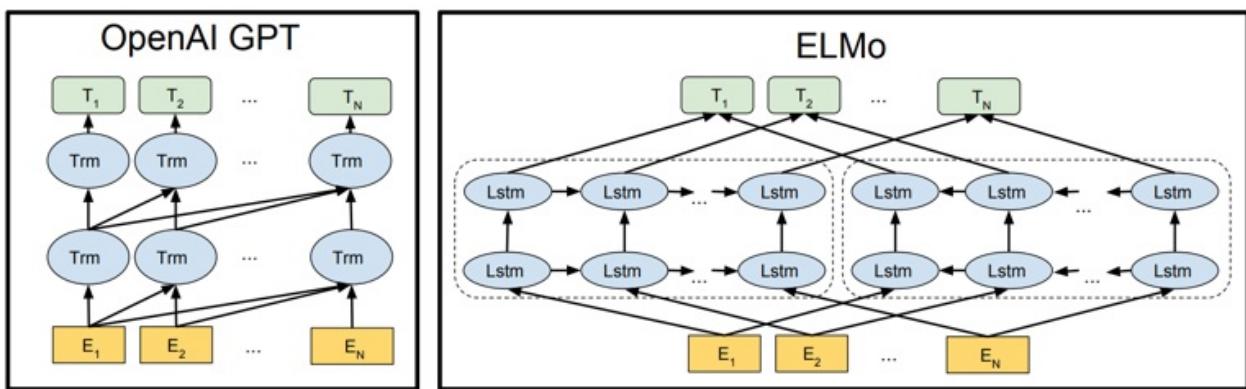


图3：OpenAI GPT和ELMo

BERT对比这两个算法的优点是只有BERT表征会基于所有层中的左右两侧语境。BERT能做到这一点得益于Transformer中Attention机制将任意位置的两个单词的距离转换成了1。

1.2 输入表示

BERT的输入的编码向量（长度是512）是3个嵌入特征的单位和，如图4，这三个词嵌入特征是：

1. WordPiece 嵌入[67]：WordPiece是指将单词划分成一组有限的公共子词单元，能在单词的有效性和字符的灵活性之间取得一个折中的平衡。例如图4的示例中‘playing’被拆分成了‘play’和‘ing’；
2. 位置嵌入（Position Embedding）：位置嵌入是指将单词的位置信息编码成特征向量，位置嵌入是向模型中引入单词位置关系的至关重要的一环。位置嵌入的具体内容参考我之前的[分析](#)；
3. 分割嵌入（Segment Embedding）：用于区分两个句子，例如B是否是A的下文（对话场景，问答场景等）。对于句子对，第一个句子的特征值是0，第二个句子的特征值是1。

最后，说明一下图4中的两个特殊符号 `[CLS]` 和 `[SEP]`，其中 `[CLS]` 表示该特征用于分类模型，对非分类模型，该符合可以省去。`[SEP]` 表示分句符号，用于断开输入语料中的两个句子。

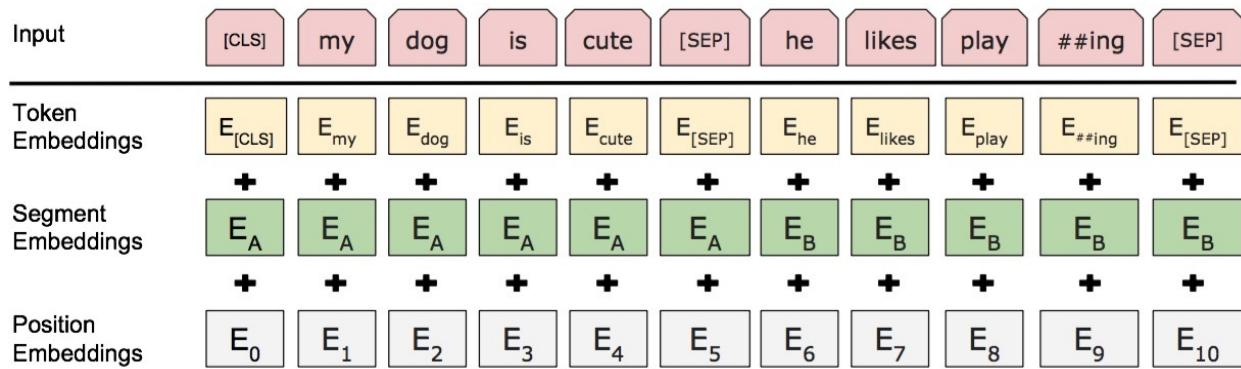


图4：*BERT*的输入特征。特征是*token*嵌入，位置嵌入和分割嵌入的单位和

1.3 预训练任务

*BERT*是一个多任务模型，它的任务是由两个自监督任务组成，即MLM和NSP。

1.3.1 Task #1 : Masked Language Model

Masked Language Model (MLM) 和核心思想取自Wilson Taylor在1953年发表的一篇论文。所谓MLM是指在训练的时候随即从输入预料上mask掉一些单词，然后通过的上下文预测该单词，该任务非常像我们在中学时期经常做的完形填空。正如传统的语言模型算法和RNN匹配那样，MLM的这个性质和Transformer的结构是非常匹配的。

在BERT的实验中，15%的WordPiece Token会被随机Mask掉。在训练模型时，一个句子会被多次喂到模型中用于参数学习，但是Google并没有在每次都mask掉这些单词，而是在确定要Mask掉的单词之后，80%的时候会直接替换为`[Mask]`，10%的时候将其替换为其它任意单词，10%的时候会保留原始Token。

- 80% : `my dog is hairy -> my dog is [mask]`
- 10% : `my dog is hairy -> my dog is apple`
- 10% : `my dog is hairy -> my dog is hairy`

这么做的原因是如果句子中的某个Token100%都会被mask掉，那么在fine-tuning的时候模型就会有一些没有见过的单词。加入随机Token的原因是因为Transformer要保持对每个输入token的分布式表征，否则模型就会记住这个`[mask]`是token 'hairy'。至于单词带来的负面影响，因为一个单词被随机替换掉的概率只有 $15\% * 10\% = 1.5\%$ ，这个负面影响其实是可以忽略不计的。

另外文章指出每次只预测15%的单词，因此模型收敛的比较慢。

1.3.2 Task #2: Next Sentence Prediction

Next Sentence Prediction (NSP) 的任务是判断句子B是否是句子A的下文。如果是的话输出'IsNext'，否则输出'NotNext'。训练数据的生成方式是从平行语料中随机抽取的连续两句话，其中50%保留抽取的两句话，它们符合IsNext关系，另外50%的第二句话是随机从预料中提取的，它们的关系是NotNext的。这个关系保存在图4中的 [CLS] 符号中。

1.4 微调

在海量单预料上训练完BERT之后，便可以将其应用到NLP的各个任务中了。对于NSP任务来说，其条件概率表示为 $P = \text{softmax}(CW^T)$ ，其中 C 是BERT输出中的 [CLS] 符号， W 是可学习的权值矩阵。

对于其它任务来说，我们也可以根据BERT的输出信息作出对应的预测。图5展示了BERT在11个不同任务中的模型，它们只需要在BERT的基础上再添加一个输出层便可以完成对特定任务的微调。这些任务类似于我们做过的文科试卷，其中有选择题，简答题等等。图5中其中 Tok 表示不同的Token， E 表示嵌入向量， T_i 表示第 i 个 Token 在经过BERT处理之后得到的特征向量。

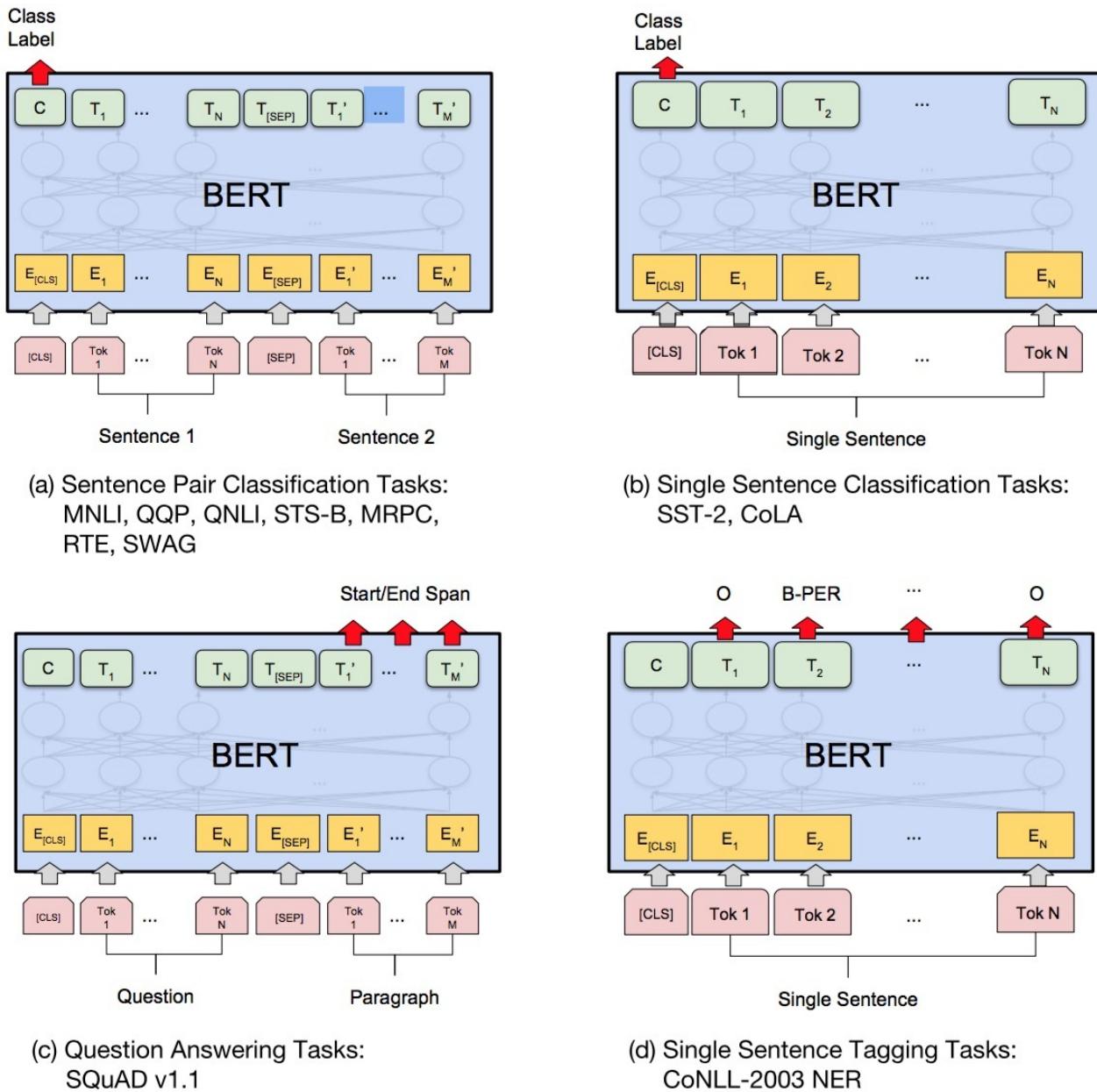


图 5 : BERT 用于模型微调

微调的任务包括 (a) 基于句子对的分类任务：

- **MNLI**：给定一个前提 (Premise)，根据这个前提去推断假设 (Hypothesis) 与前提的关系。该任务的关系分为三种，蕴含关系 (Entailment)、矛盾关系 (Contradiction) 以及中立关系 (Neutral)。所以这个问题本质上是一个分类问题，我们需要做的是去发掘前提和假设这两个句子对之间的交互信息。
- **QQP**：基于 Quora，判断 Quora 上的两个问题句是否表示的是一样的意思。
- **QNLI**：用于判断文本是否包含问题的答案，类似于我们做阅读理解定位问题所在的段落。
- **STS-B**：预测两个句子的相似性，包括 5 个级别。
- **MRPC**：也是判断两个句子是否是等价的。

- RTE：类似于MNLI，但是只是对蕴含关系的二分类判断，而且数据集更小。
- SWAG：从四个句子中选择为可能为前句下文的那个。

(b) 基于单个句子的分类任务

- SST-2：电影评价的情感分析。
- CoLA：句子语义判断，是否是可接受的（Acceptable）。

对于GLUE数据集的分类任务（MNLI，QQP，QNLI，SST-B，MRPC，RTE，SST-2，CoLA），BERT的微调方法是根据 [CLS] 标志生成一组特征向量 C ，并通过一层全连接进行微调。损失函数根据任务类型自行设计，例如多分类的softmax或者二分类的sigmoid。

SWAG的微调方法与GLUE数据集类似，只不过其输出是四个可能选项的softmax：

$$P_i = \frac{e^{V \cdot C_i}}{\sum_{j=1}^4 e^{V \cdot C_i}}$$

(c) 问答任务

- SQuAD v1.1给定一个句子（通常是一个问题）和一段描述文本，输出这个问题的答案，类似于做阅读理解的简答题。如图5.(c)表示的，SQuAD的输入是问题和描述文本的句子对。输出是特征向量，通过在描述文本上接一层激活函数为softmax的全连接来获得输出文本的条件概率，全连接的输出节点个数是语料中Token的个数。

$$P_i = \frac{e^{S \cdot T_i}}{\sum_{j=1}^n e^{S \cdot T_i}}$$

(d) 命名实体识别

- CoNLL-2003 NER：判断一个句子中的单词是不是Person，Organization，Location，Miscellaneous或者other（无命名实体）。微调CoNLL-2003 NER时将整个句子作为输入，在每个时间片输出一个概率，并通过softmax得到这个Token的实体类别。

2. 总结

BERT近期火得一塌糊涂不是没有原因的：

1. 使用Transformer的结构将已经走向瓶颈期的Word2Vec走向了一个新的方向，并再一次炒火了《Attention is All you Need》这篇论文；
2. 11个NLP任务的精度大幅提升足以震惊整个深度学习领域；
3. 无私的开源了多种语言的源码和模型，具有非常高的商业价值。
4. 迁移学习又一次胜利，而且这次是在NLP领域的连胜，狂胜。

BERT算法还有很大的优化空间，例如我们在Transformer中讲的如何让模型有捕捉Token序列关系的能力，而不是简单依靠位置嵌入。BERT的训练在目前的计算资源下很难完成，论文中说BERT_{LARGE}的训练需要在64块TPU芯片上训练4天完成，而一块TPU的速度约是目前主流GPU的7-8倍。非常幸运的是谷歌开源了各种语言的模型，免去了我们自己训练的工作。

最后，改用一句名言：谷歌大法好！

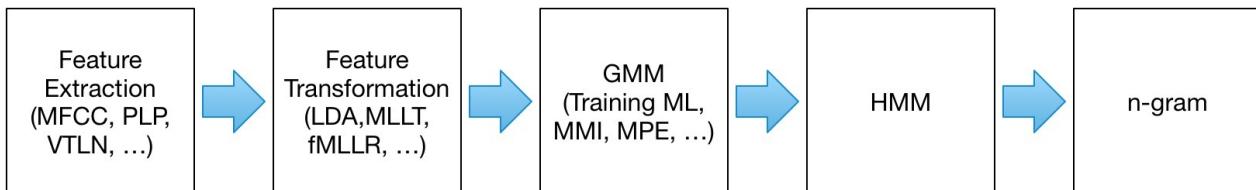
Speech Recognition with Deep Recurrent Neural Network

介绍

语音识别是具有巨大市场和研究潜力的一个领域，语音识别已经有了几十年的研究历史了。2000年前，涌现了大量的语音识别技术，例如：混合高斯模型（GMM），隐马尔科夫模型（HMM），梅尔倒谱系数（MFCC），n元语义语言模型（n-gram LM）等等（图1）。在21世纪的第一个十年，这些技术被成功应用到实际系统中。但同时，语音识别的技术仿佛遇到了瓶颈期，不论科研进展还是实际应用均进展非常缓慢。

2012年，深度学习兴起。仅仅一年之后，Hinton的著名弟子Alex Graves的这篇使用深度学习思想解决语音识别问题的这篇文章[66]引起了广泛关注，为语音识别开辟了新的研究方向，这篇文章可以说是目前所有深度学习解决语音识别方向的奠基性文章了。目前深度学习均采用和其类似的RNN+CTC的框架，甚至在OCR领域也是采用了同样的思路。

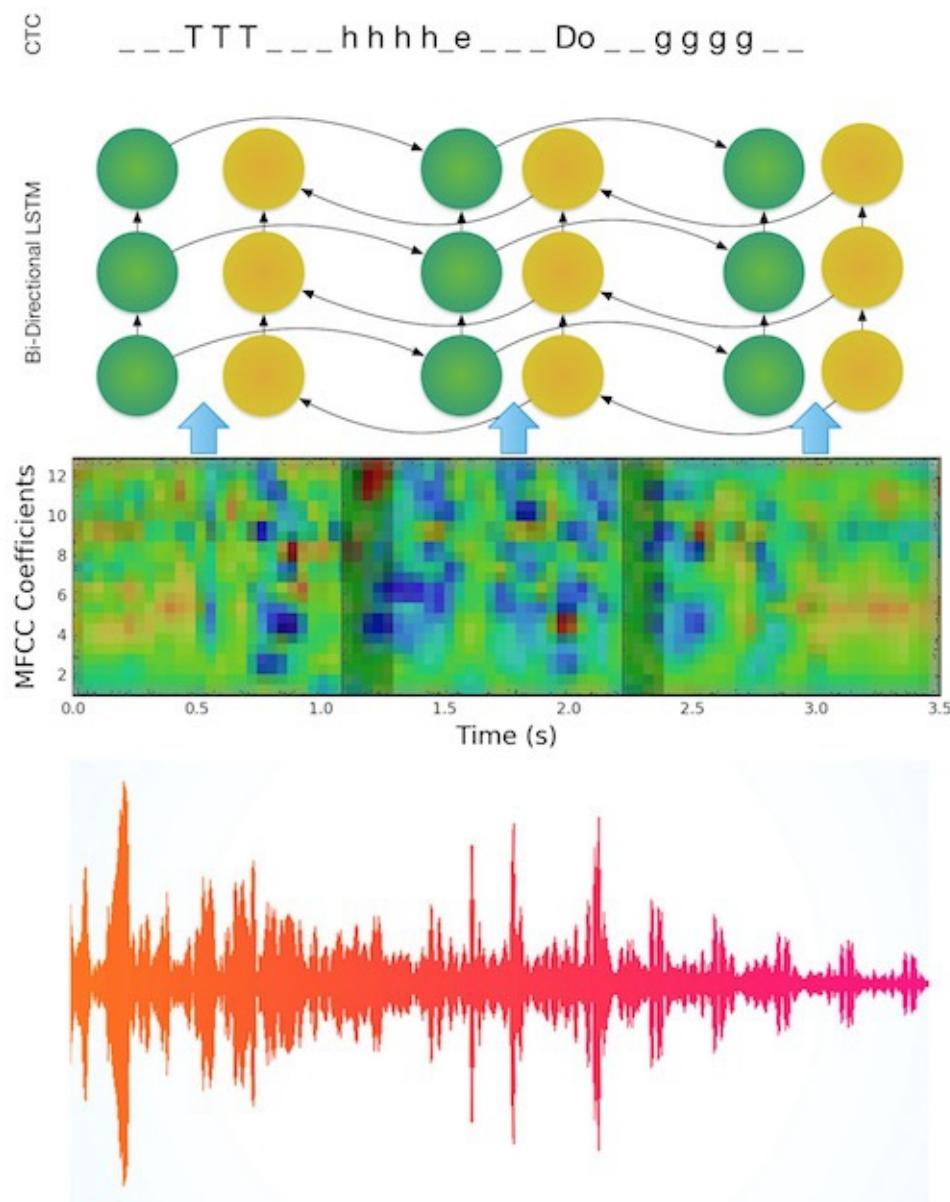
图1：语音识别传统模型



算法细节

在这篇论文涉及的实验里，使用了MFCC提取音频特征，多层次双向RNN [65] 编码特征（节点使用LSTM），CTC构建声学模型。由于CTC没有构建语音模型的能力，论文使用了RNN Transducer [64] 联合训练声学模型和语言模型。结构如图2。

图2：基于深度学习的语音识别架构



MFCC

首先，作者使用MFCC将音波的每个时间片转换成一个39维的特征向量。MFCC（Mel-Frequency Cepstral Coefficients）的全称是梅尔频率倒谱系数，是一种基于傅里叶变换的提取音频特征的方法。之后也有使用一维卷积提取特征的方法，由于MFCC和深度学习关系不大，需要详细了解的可以自行查阅相关文档，在这里可以简单理解为一种对音频的特征提取的方法。

多层双向LSTM

在这篇实验中，作者使用多层双向LSTM提取音频的时序特征，关于LSTM能解决RNN的梯度消失/爆炸以及长期依赖的问题已在本书2.4节分析过，此处不再赘述。双向LSTM（BLSTM）和双向RNN（BRNN）的不同之处仅在于BLSTM的隐节点使用和LSTM带有三个门机制的节点。所以我们首先讲解一下BRNN。

BRNN添加了一个沿时间片反向传播的节点，计算方式和RNN隐节点相同，但是第t个时间片的计算需要使用第t+1个时间片的隐节点

$$\text{正向} : h_t = \sigma(W_{xh}x_t + W_{hh}h_{t-1} + b_h)$$

$$\text{反向} : h'_t = \sigma(W_{xh'}x_t + W_{h'h'}h'_{t+1} + b_{h'})$$

$$y_t = W_{hy}h_t + W_{h'y}h'_t + b_y$$

其中， h_t 和 h' 分别表示正向和反向传输的隐层节点的输出。多层RNN的实现是通过stacking的形式完成的，即第n层，第t个时间片的节点使用第n-1层和第t-1个时间片的隐层节点的输出作为输入，

$$h_t^n = \sigma(W_{h^{n-1}h^n}h_t^{n-1} + W_{h^nh^n}h_{t-1}^n + b_h^n)$$

对于一个N层的RNN，第t个时间片的输出是

$$y_t = W_{h^Ny}h_t^N + b_y$$

RNN Transducer

CTC使用RNN得到的特征向量作为输入，所以CTC建模的是声学模型，但是很多时候我们还需要在模型中加入语言模型。RNN Transducer便是一种联合建立声学模型和语言模型的一种方法。更具体的讲，CTC建模的是每个时间片 y_t 的概率分布

$$Pr(k|t) = \frac{\exp(y_t[k])}{\sum_{k'=1}^K \exp(y_t[k'])}$$

而RNN Transducer建模的是当前时间片 y_t 和上个时间片输出的概率分布 p_u 的联合概率

$$l_t = W_{hN_l}h_t + W_{h'N_l}h'_t + b_l$$

$$h_{t,u} = \tanh(W_{lh}l_{t,u} + W_{pb}p_u + b_h)$$

$$y_{t,u} = W_{hy}h_{t,u} + b_y$$

$$Pr(k|t, u) = \frac{\exp(y_{t,u}[k])}{\sum_{k'=1}^K \exp(y_{t,u}[k'])}$$

RNN Transducer的解码依然可以使用beam search，CTC和beam search的讲解在2.3节已详细分析。

总结

作为深度学习进行语音识别的开山之作，这篇文章提供了MFCC+LSTM+CTC的框架流程。后面几年的深度学习相关论文均是在其基础上的扩展，虽然内容很简单，但是熟悉这一套流程框架还是有必要的。

第四章：物体检测

在2012年之前，物体检测一直是计算机视觉加上机器学习的天下。当时主要的研究方向是如何设计高质量的特征提取器和高效的分类器和回归器。当时特征提取器比较有代表性的算法有HoG，SIFT等，而分类任务几乎是SVM的天下。

早期的物体检测基本上也遵循四步走的流程：

1. Selective Search选取候选区域；
2. 特征提取器提取特征；
3. 分类器和回归器预测类别和位置四要素；
4. non maximum suppression合并检测框。

2012年，Hinton团队的AlexNet [123] 在ILSVRC大赛中将精度提高了约10个百分点。从此，计算机视觉的各个方向都开始考虑使用深度学习解决他们的问题。

卷积网络作为特征提取器首先引起了业内研究者的注意，2014年，使用深度学习解决物体检测问题的开山之作R-CNN [63]应运而生，论文的一作Ross B. Girshick (RBG) 不仅是该方向的鼻祖，而且其一系列的论文也引领了物体检测的发展方向。R-CNN的目的也很单纯，只是将特征提取器简单的换成了CNN。

Ross B. Girshick



骨干网络：同时在2014年，网络模型方向诞生了两个非常经典的网络结构，一是牛津大学计算机视觉组的VGG [120]，另外一个是谷歌公司的GoogLeNet [121]。VGG使用了当时最流行的深度学习框架Caffe，并非常有先见之明的开源了其训练好的网络模型。VGG也作为骨干网络成为了之后3-4年的检测算法的主要使用骨干网络，代表算法便是RBG的R-CNN系列的三篇文章。随着数据量的增大和人们对高精度的追求，骨干网络更深的深度成为了一个最容易想到的方向。很幸运，2016年深度学习领域的另外一尊大神，我国广东省2003年的高考理科状元何恺明的残差网络[118]使用short-cut解决了深度学习中的退化问题，因为其无限深度的能力成为了近几年物体检测算法骨干网络主要使用的算法，经典算法包括R-FCN[59]，Mask R-CNN[57]以及YOLOv3[52]等。

何恺明



端到端模型：物体检测的一个非常重要的优化方向是优化传统方法四步走的流程，2015年RBG的Fast R-CNN [61]使用softmax替代了SVM，进而将特征提取和分类模型的训练合二为一，算是第一个端到端的物体检测算法。在R-FCN [59]中使用了更为快速的投票机制替代了Fast R-CNN中的softmax，因为softmax前往往要接最少一层全连接，这也成了制约Fast R-CNN速度的一个重要瓶颈。YOLOv3 [52]则是使用 C 路sigmoid的多标签模型增强了对覆盖样本的检测能力。

同是在2015年，RBG和何恺明强强联手，推出了使用RPN替代了Selective Search的Faster R-CNN [60] 算法。Faster R-CNN因为其最高的算法精度和在显卡环境下的近实时的速度性能，也成了今年最为流行的算法之一。Faster R-CNN因其巧妙的设计也是深度学习面试官最爱问的算法之一。

Faster R-CNN系列虽然在实现上实现了端到端训练，但是其两步走（候选区域提取+位置精校）的策略也被一些人诟病。2016年，Joseph Redmon提出了更为革命性的YOLO[56] 系列算法。不同于R-CNN系列分两步走的策略，YOLO是单次检测检测的算法，YOLO可以看做是高精度的RPN。其更彻底的端到端训练将物体检测的速度大幅提升，在非顶端显卡环境下也实现了实时检测。

Joseph Redmon



降采样池化：无论是Selective Search还是RPN，得到的候选区域在尺寸和比例上都是不固定的，由此输入到网络中得到的Feature Map大小是不同的，最后展开成的特征向量长度也不固定，在目前的开源框架下，暂不支持变长的特征向量作为输入。在SPP [62] 中，作者提出了

金字塔池化的方式，通过多尺度分bin的形式得到长度固定的特征向量，在Fast R-CNN中将其简化为单尺度并命名为ROI Pooling。Mask R-CNN [57]发现当ROI Pooling应用到语义分割任务中会存在若干个像素的偏移误差，由此设计了更为精确的ROIAlign。

锚点：Faster R-CNN最大的特点是在RPN网络中引入了锚点机制，对锚点一个更好的解释是先验框，即对检测框的先验假设。在早期阶段，锚点是根据开发者的经验手动写死的。在YOLOv2 [55]中，作者在训练集对锚点进行了k-means聚类，进而产生了一组更优代表性的锚点。DSSD中锚点的设置则是根据聚类的结果分析得到的。

小尺寸物体检测困难：在所有的检测算法中都普遍存在着小尺寸物体检测困难的问题。究其原因，是因为在深层网络中随着语义信息的增强，位置信息也越来越弱，这是深度网络的固有问题。SSD [54]率先提出使用各个阶段的Feature Map都参与损失函数的计算，在FPN [58]中则是通过将各个阶段的Feature Map融合到一起的方式，融合的方式有FPN中从小尺寸向大尺寸融合的双线性插值上采样算法，也是目前最为广泛使用的融合方法；DSSD [53]则是通过反卷积得到不仅将小尺寸Feature Map上采样，而且包含语义信息的Feature Map；而YOLOv2采用的是中的大尺寸向小尺寸融合的 space_to_depth() 算法。而YOLOv3则是接合了FPN和锚点机制的思想，为不同深度的Feature Map赋予了不同比例，不同尺寸的锚点。

YOLOv2中采用的另外一个解决方案则是在训练过程中，不同批使用不同尺寸的输入图像。

半监督学习：系列算法中一个非常有商业前景的方向便是通过半监督学习的方式增加模型可处理的类别。半监督学习即是通过少量的带标签数据和大量的无标签数据，将模型的能力扩展到无标签数据中。YOLO9000 [55]通过WordTree融合了80类的检测数据集COCO和9418

类的分类数据集ImageNet，生成了可以检测9418类物体的模型。Mask^X R-CNN [51]则是通过权值迁移函数融合了80类的分割数据COCO和3000类的检测数据集Visual Gnome，生成了可以分割3000类物体的模型。

物体检测和语义分割：近几年物体检测和语义分割的距离越来越小，双方都在汲取对方的算法来获得灵感和优化算法。最典型的算法便是Mask R-CNN中融合了分类，检测和分割的三任务模型。DSSD使用反卷积进行上采样也非常有意思

最后预测一下未来一段时间物体检测的发展方向：

1. 小尺寸物体检测困难至今尚未有效解决，更有效的多尺度Feature Map，或者针对小尺寸物体的特定算法是研究的一个热点和难点；
2. 半监督学习：能否将语义分割任务扩展到ImageNet类别中，提升非子类或父类物体的无监督学习能力是一大热点；
3. 嵌入式平台的物体检测算法：目前最快的YOLOv3的实时运行依然依赖GPU环境，能否将检测算法实时的应用到嵌入式平台，例如手机，扫地机器人，无人机等都是有急切需求的场景；
4. 特定领域的物体检测算法：目前在单一领域发展较靠前的是场景文字检测算法。但在一些特定的场景中，例如医学，安检，微生物等依然很有研究前景，也是比较容易有研究成果和应用场景的方向。

如果您是一个有一定物体检测研究基础的读者，我从各个方向帮您梳理了近年来物体检测的发展历程。如果您对上面所说的算法不知所云，不要着急，我会在后面的章节中详细的解析上面提到过的所有算法。了解完本章的所有内容之后，再来重读这一部分，你一定会构建更清晰的知识体系。

读到这，你可能觉得有些枯燥和乏味。我在《复仇者联盟3》的预告片中运行了一下YOLOv3物体检测算法，也许能提高提高你继续读下去的兴趣，视频见百度云：<https://pan.baidu.com/s/1rv7QhuUAWleW5XjeEka8kQ>。

Rich feature hierarchies for accurate object detection and semantic segmentation

tags: R-CNN, Object Detection

简介

论文发表于2014年，自2012年之后，物体检测的发展开始变得缓慢，一个重要的原因是基于计算机视觉的方法（SIFT，HOG）进入了一个瓶颈期。生物学上发现人类的视觉是一个多层次的流程，而SIFT或者HOG只相当于人类视觉的第一层，这是导致瓶颈期的一个重要原因。

2012年，基于随机梯度下降的卷机网络在物体识别领域的突破性进展充分展现了CNN在提取图片特征上的巨大优越性 [123]。CNN的一个重要的特点是其多层次的结构更符合人类的生物特征。

但大规模深度学习网络的应用对数据量提出了更高的需求。在数据量稀缺的数据集上进行训练，迭代次数太少会导致模型拟合能力不足，迭代次数太多会导致过拟合。为了解决该问题，作者使用了在海量数据上的无监督学习的预训练加上稀缺专用数据集的fine-tune。

在算法设计上，作者采用了“Recognition Using Regions” [50]的思想，R-CNN [63]使用 Selective Search [49]提取了2k-3k个候选区域，对每个候选区域单独进行特征提取和分类器训练，这也是R-CNN命名的由来。

同时，为了提高检测精度，作者使用了岭回归对检测位置进行了精校。以上方法的使用，使得算法在PASCAL数据集上的检测到达了新的高度。

算法详解

1. R-CNN流程

R-CNN测试过程可分成五个步骤

1. 使用Selective Search在输入图像上提取候选区域；
2. 使用CNN对每个wrap到固定大小（227 * 227）的候选区域上提取特征；
3. 将CNN得到的特征Pool5层的特征输入N个（类别数量）SVM分类器对物体类别进行打分
4. 根据Pool5的特征输入岭回归器进行位置精校。

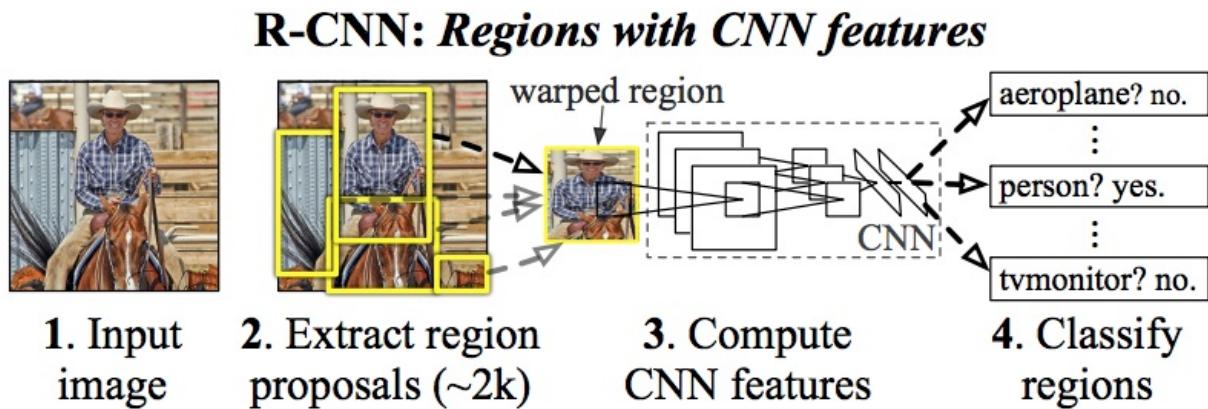
5. 使用贪心的非极大值抑制（NMS）合并候选区域，得到输出结果

所以，R-CNN的训练过程也涉及

- CNN特征提取器
- SVM分类器
- 岭回归位置精校器

三个模块的学习。

论文中给出的图（图1）没有画出回归器部分。



2. 候选区域提取

R-CNN输入网络的并不是原始图片，而是经过Selective Search选择的候选区域。

1. Selective Search 使用 [48] 的方法，将图像分成若干个小区域
2. 计算相似度，合并相似度较高的区域，直到小区域全部合并完毕
3. 输出所有存在过的区域，即候选区域 如下面伪代码：

Algorithm 1: Hierarchical Grouping Algorithm

```

Input: (color) image
Output: Set of object location hypotheses L

Obtain initial regions R = {r1, ..., r13}
Initial similarity set S = []

foreach Neighbouring region pair(ri, rj) do
    Calculate similarity s(ri, rj)
    S.insert(s(ri, rj))

while S != [] do
    Get highest similarity s(ri, rj) = max(S)
    Merge corresponding regions ri = Union(ri, rj)
    Remove similarities regarding ri: S = S.delete(ri, r*)
    Remove similarities regarding sj: S = S.delete(r*, rj)
    Calculate similarity set St between rt and its neighbours
    S = Union(S, St)
    R = Union(R, rt)

Extract object location boxes L from all regions in R

```

Selective Search 伪代码 区域的合并规则是：

1. 优先合并颜色相近的
2. 优先合并纹理相近的
3. 优先合并合并后总面积小的
4. 合并后，总面积在其BBOX中所占比例大的优先合并

图2是通过Selective Search得到的一候选区域



3. 训练数据准备

3.1 CNN的数据准备

1. 预训练：使用ILSVRC 2012的数据，训练一个N类任务的分类器。在该数据集上，top-1的error是2.2%，达到了比较理想的初始化效果。
2. 微调：每个候选区域是一个N+1类的分类任务（在PASCAL上，N=20；ILSVRC，N=200）。表示该候选区域是某一类或者是背景。当候选区域和某一类物体的Ground Truth box的重合度（IoU）大于0.5时，该样本被判定为正样本，否则为负样本。

3.2 SVM分类器的数据准备

标签

由于SVM只能做二分类，所以在N分类任务中，作者使用了N个SVM分类器。对于第K类物体，与该物体的Ground Truth box的IoU大于0.3的视为正样本，其余视为负样本。论文中指出，0.3是通过Grid Search得到的最优阈值。

通过实验结果选取IoU阈值是一方面。作者在附录B¹中给了解释，其实不太理解其思路，希望明白的大神能够帮忙给出解释。

特征

作者通过对比CNN网络中的Pool5，fc6，fc7三层的特征在PASCAL VOC 2007数据集上的表现，发现Pool5层得到的error更低，所以得出结论Pool5更能表达输入数据的特征，所以SVM使用的是从Pool5提取的特征。原因可能是图像的特征更容易通过卷积而非全连接来表示。

3.3 岭回归精校器的数据准备

特征

位置精校和[47]的思路类似，不同之处是使用CNN提取的特征而非DNN。同SVM一样，回归器也是使用的从Pool5层提取的特征。候选区域是选取的样本是和Ground Truth的IoU大于0.6的样本。

标签

回归器使用的是相对位置， $G = \{G_x, G_y, G_w, G_h\}$ 表示Ground Truth的坐标和长宽，

$P = \{P_x, P_y, P_w, P_h\}$ 表示候选区域的大小和长宽。相对的回归目标 $T = \{t_x, t_y, t_w, t_h\}$ 的计算方式如下：

$$t_x = (G_x - P_x)/P_w$$

$$t_y = (G_y - P_y)/P_h$$

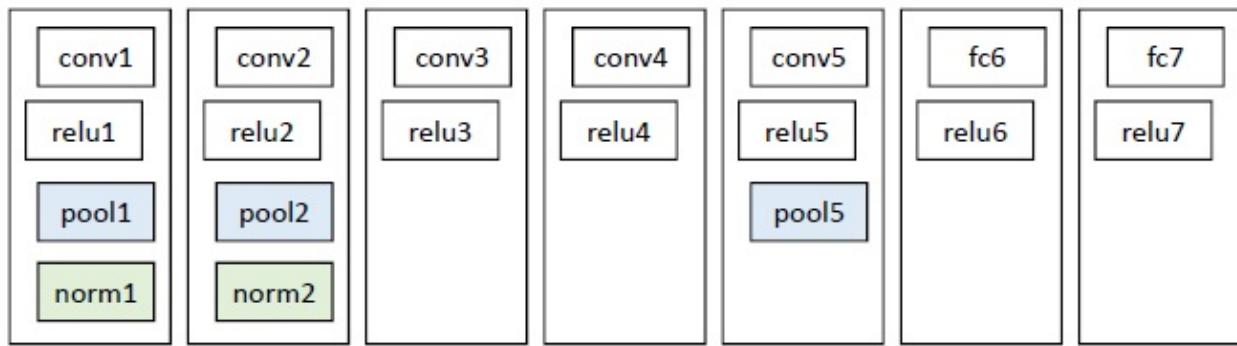
$$t_w = \log(G_x/P_w)$$

$$t_h = \log(G_y/P_h)$$

4. 训练

4.1 CNN

作者通过对比Alex-Net5(论文中叫做T-Net), VGG (论文中叫做O-Net) , 通过折中考虑mAP和训练时间，最终采用了Alex-Net。Alex-Net的网络结构如下图：



预训练就是在ILSVRC训练分类网络，不再赘述。

微调训练使用了mini-batch的SGD进行优化，batchsize的大小是128，其中32个正样本，96个负样本。CNN使用的loss是SOFTMAX loss。

4.2 SVM训练

SVM的训练使用了Hard Negative Mining, 对于目标检测中我们会事先标记处ground truth，然后在算法中会生成一系列proposal，这些proposal有跟标记的ground truth重合的也有没重合的，那么重合度 (IOU) 超过一定阈值 (通常0.5) 的则认定为是正样本，以下的则是负样本。然后扔进网络中训练。然而，这也许会出现一个问题那就是正样本的数量远远小于负样本，这样训练出来的分类器的效果总是有限的，会出现许多false positive，把其中得分较高的这些false positive当做所谓的Hard negative，既然mining出了这些Hard negative，就把这些扔进网络再训练一次，从而加强分类器判别假阳性的能力。

4.3 岭回归训练

精校器的作用是找到一组映射，是后续区域的位置信息P通过某种映射，能够转化为G。也可以理解为将Pool5层的图像特征，学习G和P的相对位置关系(3.3 中的t)，然后根据相对位置关系，便可以将候选区域还原成Ground Truth。所以可以有下面目标函数

$$w_{\star} = \operatorname{argmin}_{\hat{w}_{\star}} \sum_i^N (t_{\star}^i - \hat{w}_{\star}^T \phi_5(P^i))^2 + \lambda \|\hat{w}_{\star}\|^2$$

其中 $\phi_5(P^i)$ 表示候选区域 P^i 对应的 Pool5 层特征向量。

¹ 原文：historically speaking, we arrived at these definition because we started by training SVMs on features computed by the ImageNet pre-trained CNN, and so fine-tuning was not a consideration at that point in time. ↪

Spatial Pyramid Pooling in Deep Convolutional Networks for Visual Recognition

tag: SPP-Net, Objection Detection, R-CNN

简介

SPP-Net [62] 的初衷是解决令人头疼的输入图像的尺寸固定的要求，无论是裁剪，拉伸或者是加边都会对模型的效果带来负面影响。那是什么原因导致的需要固定输入图像的尺寸呢？一个CNN网络结构通常由卷积层和全连接层组成，卷积层通过滑动窗口的形式得到下一层，卷积对输入图像的尺寸并没有要求，只是不同尺寸的输入会产生不同尺寸的特征层。但是全连接要求的输入向量的尺寸是固定的，这也进而导致了图像特征层尺寸的固定，从而影响可对输入图像的尺寸要求。SPP是介于特征层（卷积层的最后一层）和全连接之间的一种pooling结构，不同尺寸的特征图通过金字塔式多个层次的pooling，可以得到固定尺寸的输入，从而满足全连接的要求。SPP的思想无论是对于图像分类还是物体检测，都是适用的。

从生物学的角度讲，SPP也是更符合人类的视觉特征的，因为当我们看一个物体时，是不考虑物体的尺寸的，而是再更深的视觉系统中进行处理。

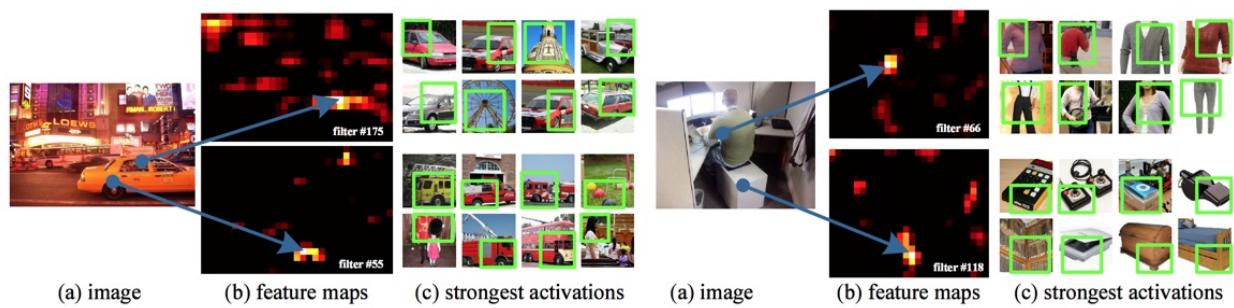
下面我们回到物体检测，在SPP-Net之前，物体检测效果最好的是R-CNN [63]。但R-CNN非常耗时，因为其再使用Selective Search提取候选区域后，对每张图的几千个候选区域重复的进行卷积操作。SPP-Net只需要在整张图片上运行一次卷积网络层，然后使用SPP-Net的金字塔的pooling思想在特征图上提取特征，这一操作将运行速度提升了上百倍。

作者的这一工作在ILSVRC2014上也取得了非常优秀地成绩（检测第二名，分类第三名）。

算法详解

分类

作者通过可视化卷积网络的Conv5层，发现卷积操作其实保存了输入图像的空间特征，且不同的卷积核可能响应不同的图像语义特征。如论文中提供的图1，通过第一张图片的特征图，我们可以看到filter-175倾向于响应多边形特征，filter-55倾向于响应圆形特征，通过第二张图可以看出，filter-66倾向于响应Λ形状，而118则倾向于响应V形状。上面这些响应是与输入图像的尺寸没有关系的，而是取决于图像的内容。



在传统的计算机视觉系列的方法中，我们首先可以通过SIFT或者HOG等方法提取图像特征，然后通过词袋或者空间金字塔的方法聚集这些特征 [45][46]。同样我们也可以用类似的方法聚集卷机网络得到的特征，这便是SPP-Net的算法思想。

首先通过卷积网络提取输入图像（尺寸无要求）的特征，然后通过空间金字塔池化（SPP）的方法将不同的特征图聚集成相同尺寸的特征向量，这些长度相同的特征向量便可以用于训练全连接或者SVM。与传统的词袋方法对比，SPP保存了图像的空间特征。得到尺寸相同的特征向量后。

图2阐明了SPP的网络结构

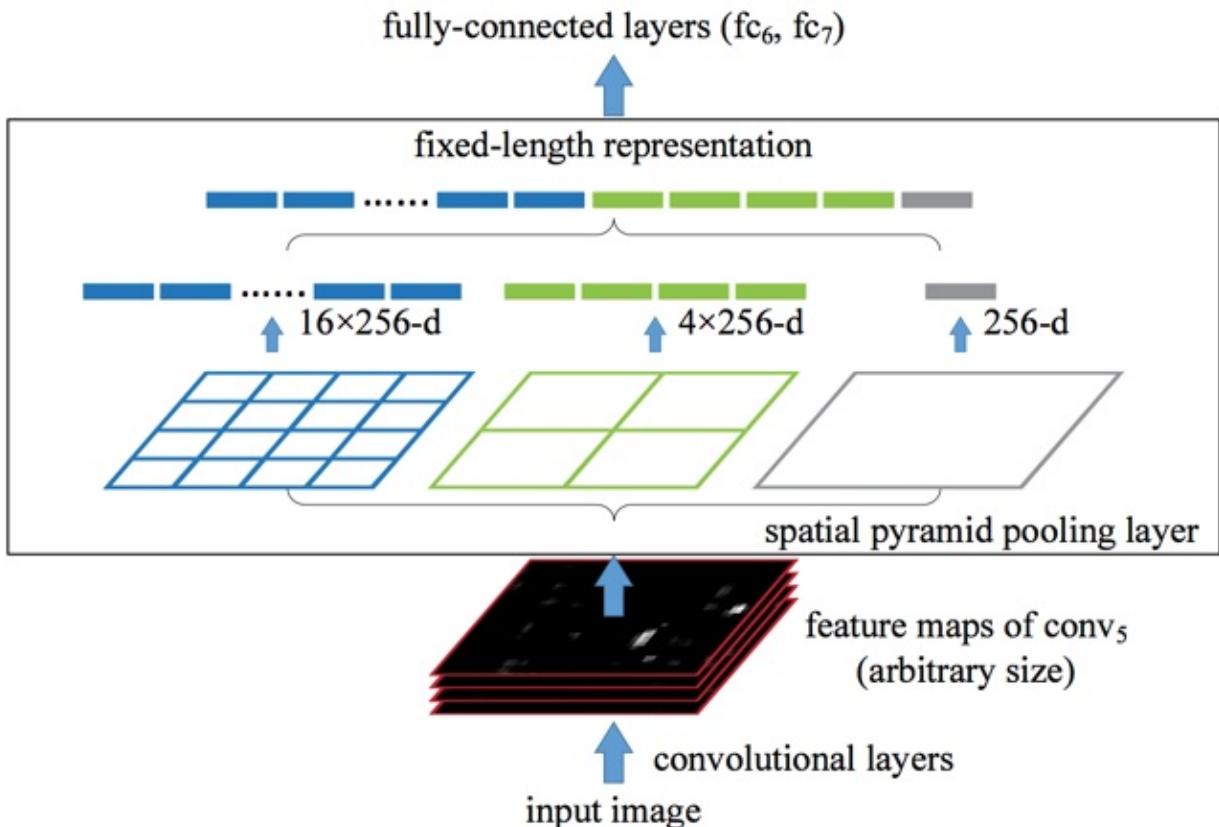


图2中conv5层有256个卷积核，论文中使用了 4×4 , 2×2 , 1×1 三个尺度的金字塔，在每个尺度的grid（大小和输入图像的尺寸有关）使用的是max pooling得到特征向量。将所有尺度的特征向量连在一起就得到了一个 5376^1 的特征的特征向量，该特征向量便是全连接的输入。通过分析可以看出，尽管输入到网络的图像的尺寸不一样，经过SPP-NET后，都会得到相同长度的特征向量。

该方法可以通过标准的BP进行调参，然而实际训练过程中，GPU更倾向于尺寸固定的输入图像（例如Mini-batch训练），为了能够使用现有的框架（Caffe）并同时考虑多尺度的因素，作者使用了多个不同固定输入图像尺寸的网络，这些网络是共享参数的。对于任意不同输入尺寸的卷积网络，经过卷积层得到conv5的尺度是 $a*a$ ，如果我们要使用金字塔的某层取一个 $n*n$ 的特征向量，则pooling层的窗口大小是 $[a/n]$ ，步长是 $[a/n]$ 。可见，参数的是和输入图像的尺寸没有关系的，因此不同的网络之间权值是可以共享的。

在实验中，作者使用了224*224和180*180两个网络，将图像resize固定到其中一个尺寸后训练该网络，并将学到的参数共享到另外一个网络中。更具体的，作者每个epoch更换一种图像尺寸，训练结束后共享权值。

注：作者根本没有实现训练不同大小输入的CNN的BP算法，只是针对各种各样大小不同的输入，定义出不同的网络，但这些网络实际上参数都相同，于是就可以用现有工具来训练。

在测试时，由于不存在mini-batch，所以输入图像的尺寸是任意的。

检测

简单的回顾一下R-CNN。首先R-CNN利用Selective Search在输入图像上提取2000个左右的候选区域，将每个候选区域拉伸到227*227的尺寸。使用标准的卷机网络训练这些候选区域，提取conv5特征层的特征用于训练n个2分类的SVM作为分类模型，一个回归器用于位置精校。R-CNN的性能瓶颈之一是在同一张图像上的2000个左右的候选区域上重复的进行卷积操作，这在测试过程是非常耗时的。

SPP-NET首先在输入图像上提取2000个候选区域。按照图像的短边（缩小到s）将图片resize²后，使用卷积网络提取整张图片的特征（提升时间的最关键部分）。找到每个候选区域对应的conv5特征图的部分，使用金字塔池化的方法提取长度固定的特征向量。特征向量后经过一个全连接后输入到二分类SVM中用于训练SVM分类器。同R-CNN一样，SPP-NET也使用了n个二分类的SVM。过程如图3.

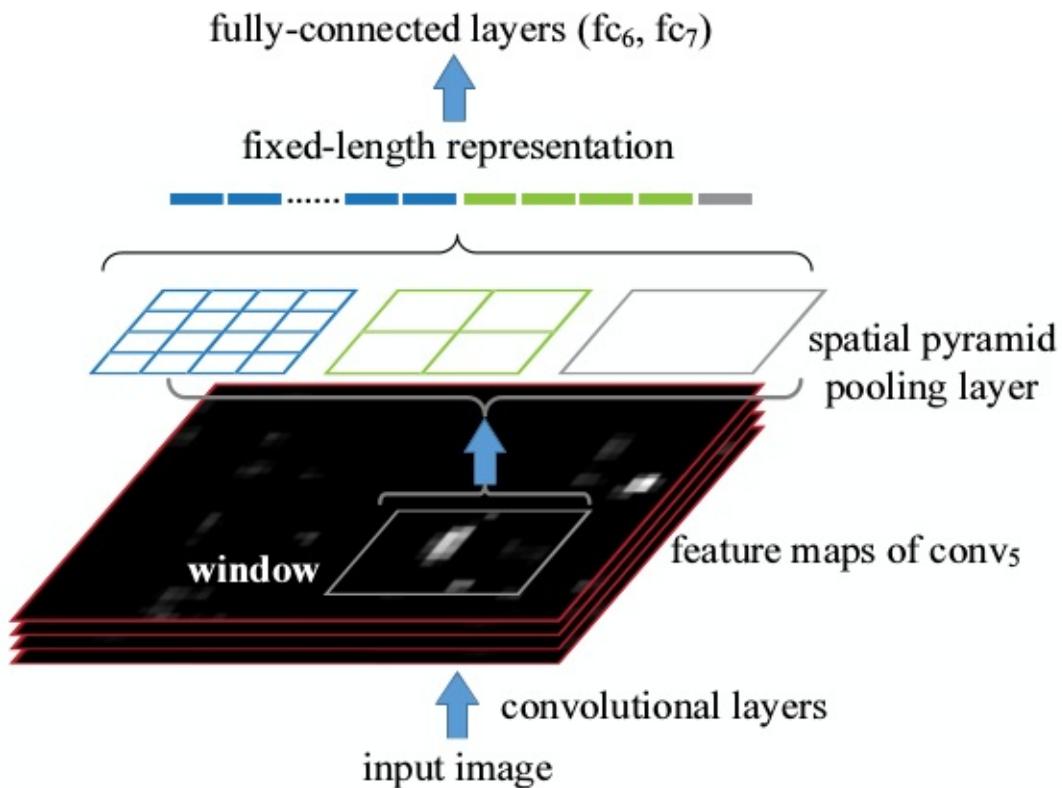


Figure 5: Pooling features from arbitrary windows on feature maps. The feature maps are computed from the entire image. The pooling is performed in candidate windows.

在上面一段中，我们提到了要找到原图的候选区域在特征层对应的相对位置。由于卷积操作并不影响物体在图像中的相对位置，这就涉及到感受野（Receptive Field）的计算问题。感知野是从第一层全连接从后往前推，公式是

$$rfsiz = (out - 1) \times stride + ksize$$

其中out是上一层感知野的大小，stride是步长，ksize和核函数的大小。

根据论文中给出的ZF-5Net的网络结构，便得出了论文附录A中感知野139的计算方法：

model	conv ₁	conv ₂	conv ₃	conv ₄	conv ₅
ZF-5	96×7^2 , str 2 LRN, pool 3^2 , str 2 map size 55×55	256×5^2 , str 2 LRN, pool 3^2 , str 2 map size 27×27	384×3^2 map size 13×13	384×3^2 map size 13×13	256×3^2 map size 13×13

1. conv₅: $fsize = (1 - 1) \times 1 + 3 = 3$
2. conv₄: $fsize = (3 - 1) \times 1 + 3 = 5$
3. conv₃: $fsize = (5 - 1) \times 1 + 3 = 7$
4. conv₂(LRN): $fsize = (7 - 1) \times 2 + 3 = 15$
5. conv₂: $fsize = (15 - 1) \times 2 + 5 = 33$

6. conv1(LRN): $fsize = (33 - 1) \times 2 + 5 = 67$

7. conv1: $fsize = (67 - 1) \times 2 + 7 = 139$

每经过一次 **stride=2** 的操作，相当于进行一次降采样，共四次 **stride**，也就是特征层的一个像素相当于原图的 16 的步长。根据论文 [\[44\]](#)，可以得到类似的结果，后面有时间的话，会给出这篇论文的总结。

同时，作者指出，**SPP-NET** 的卷积网络是可以使用候选区域进行微调的。针对候选区域的类别特征 (**N+1**) 作者在全连接的最后一层又接了一个 **n+1** 类的全连接，在实验中，作者通过 conv5 层提取的特征层没有经过微调，只是微调了一下全连接层。使用的数据是 25% 的正样本（和 **ground truth** 的重合度大于 50%）。

和 **R-CNN** 一样，作者也使用了回归器用于位置矫正，特征同样是从 conv5 层提取的特征。

1. $256 * (16+4+1) = 2576 \leftrightarrow$

2. 在实验中，作者使用了 $s \in \{480, 576, 688, 864, 1200\}$ 了多个 **resize** 尺度，原始策略是将这 5 个尺度的特征连在一起作为特征向量，但是作者发现将图像 **resize** 到像素点个数接近 $224*224$ 的那个尺度得到的效果最好 \leftrightarrow

Fast R-CNN

简介

在之前介绍的R-CNN [63]和SPP-net [62] 中，它们都是先通过卷积网络提取特征，然后根据特征训练SVM做分类和回归器用于位置矫正。这种多阶段的流程有两个问题

1. 保存中间变量需要使用大量的硬盘存储空间
2. 不能根据分类和矫正结果调整卷积网络权值，会一定程度的限制网络精度。

作者通过多任务的方式将R-CNN和SPP-net整合成一个流程，同时也带来分类和检测精度的提升吗，通过Softmax替代SVM的分类任务省去了中间变量的使用。Fast R-CNN的代码也是非常优秀的一份代码，强烈推荐参考学习：<https://github.com/rbgirshick/fast-rcnn>。

同SPP-net一样，Fast R-CNN [61]将整张图像输入到卷积网络用语提取特征，将Selective Search选定的候选区域坐标映射到卷积层。使用ROI 池化层 (单尺度的SPP层)将不同尺寸的候选区域特征窗口映射成相同尺寸的特征向量。经过两层全连接后将得到的特征分支成两个输出层，一个N+1类的softmax用于分类，一个bbox 回归器用于位置精校。这两个任务的损失共同用于调整网络的参数。

和SPP-net对比，fast R-CNN最大的优点是多任务的引进，在优化训练过程的同时也避免了额外存储空间的使用并在一定程度上提升了精度。

算法详解

1. 数据准备

1.1 候选区域选择

数据准备工作集中在 `/lib/datasets` 目录下面，下面会着重介绍几个重要部分

通过Selective Search¹选取候选区域，在5.5中论文指出，随着候选区域的增多，mAP呈先上升后下降的趋势。在fast-rcnn的源码中，作者选取了2000个候选区域。

```
# PASCAL specific config options
self.config = {'cleanup' : True,
               'use_salt' : True,
               'top_k'    : 2000}
```

1.2 输入图片尺度

作者在5.2中讨论了输入图片的尺寸问题，通过对比多尺度{480, 576, 688, 864, 1200}和单尺度的精度，作者发现单尺度和多尺度的精度差距并不明显。这也从一个角度证明了深度卷积网络有能力直接学习到输入图片的尺寸不变性。

尺度选项可以在 `lib/fast-rcnn/config.py` 里面设计，如下面代码，`scales`可以为单值或多个值。

```
# Scales to use during training (can list multiple scales)
# Each scale is the pixel size of an image's shortest side
__C.TRAIN.SCALES = (600,)
```

综合时间，精度等各种因素，作者在实验中使用了最小边长600，最大边长不超过1000的 `resize` 图像方法，通过下面函数实现。

```
def prep_im_for_blob(im, pixel_means, target_size, max_size):
    """Mean subtract and scale an image for use in a blob."""
    im = im.astype(np.float32, copy=False)
    im -= pixel_means
    im_shape = im.shape
    im_size_min = np.min(im_shape[0:2])
    im_size_max = np.max(im_shape[0:2])
    im_scale = float(target_size) / float(im_size_min)
    # Prevent the biggest axis from being more than MAX_SIZE
    if np.round(im_scale * im_size_max) > max_size:
        im_scale = float(max_size) / float(im_size_max)
    im = cv2.resize(im, None, None, fx=im_scale, fy=im_scale, interpolation=cv2.INTER_LINEAR)
    return im, im_scale
```

1.3 图像扩充

数据扩充往往对深度卷积网络的训练能够起到正面作用，尤其是在数据量不足的情况下。在实验中，作者仅使用了反转图片的这种扩充方式，在 `/lib/fast-rcnn-train` 里面调用了反转图片的函数。

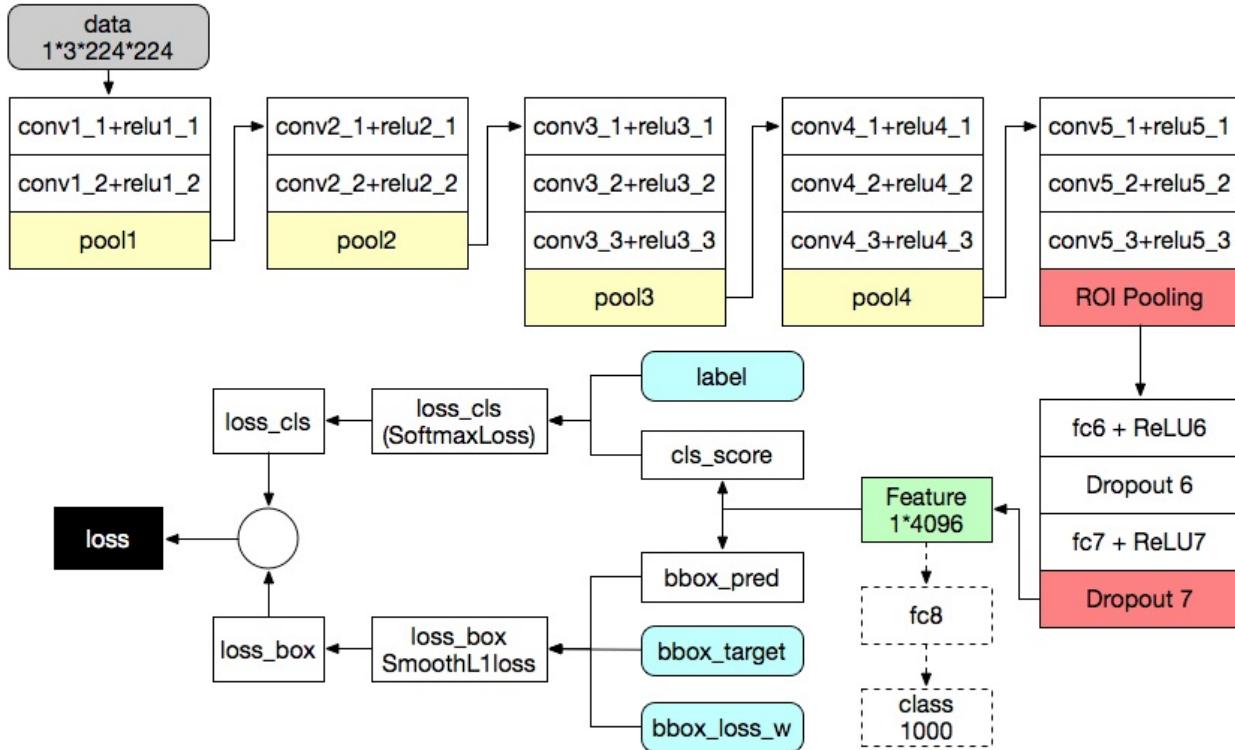
```
if cfg.TRAIN.USE_FLIPPED:
    print 'Appending horizontally-flipped training examples...'
    imdb.append_flipped_images()模拟
    print 'done'
```

2. 模型训练

2.1 网络结构

Fast-RCNN选择了VGG-16的卷积网络结构，并将最后一层的max pooling换成了roi pooling。经过两层全连接和Dropout后，接了一个双任务的loss，分别用于分类和位置精校。

图1：Fast-RCNN算法流程



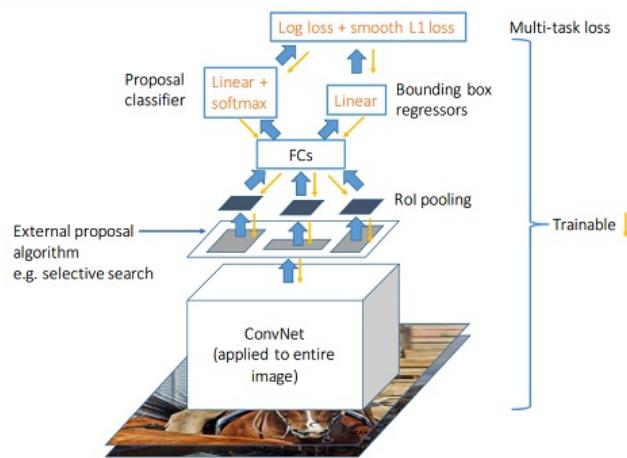
2.2 ROI pooling 层

ROI Pooling是一层的SPP-net。由 (r, c, w, h) 定义， (r, c) 表示候选区域的左上角， (w, h) 表示候选区域的高和宽。假设我们要将特征层映射到 $H * W$ 的矩阵。ROI pooling通过将特征层分成 $h/H * w/W$ 的窗格，每个窗格进行max pooling得到，在作者的实验中 $W = H = 7$ 。ROI pooling layer是在caffe源码中 `src/caffe/layers/roi_pooling_layer.cpp` 通过C++语言实现的。

既然ROI pooling层是自己定义的，当然我们也应向caffe中定义其它层一样，给出ROI pooling层的反向的计算过程。在Fast-RCNN中，ROI 使用的是固定grid数量的max pooling，在调参时，只对grid中选为最大值的像素点更新参数。可以表示为

$$\frac{\partial L}{\partial x_i} = \sum_r \sum_j [i = i^*(r, j)] \frac{\partial L}{\partial y_{r,j}}$$

selective search是在输入图像上完成的，由于所有的候选区域会共享计算，即对整张图进行卷积操作，然后将SS选择的候选区域映射到conv5特征层，最后在每一个候选区域上做ROI Pooling，如下图。



所以存在一个图像到conv5候选区域的映射过程，在Fast R-CNN源码中通过卷积后，图像的相对位置不变这一特征完成的。在Fast R-CNN使用的VGG网络中，通过max pooling做了4次stride=2的降采样，而VGG的卷积都是same卷积（卷积后图像的尺寸不变），所以特征图的尺寸变成了原来的 $1/16=0.0625$ ，在ROI pooling层中，`spatial_ratio`便是记录的这个数据。

```
layer {
  name: "roi_pool5"
  type: "ROIPooling"
  bottom: "conv5_3"
  bottom: "rois"
  top: "pool5"
  roi_pooling_param {
    pooled_w: 7
    pooled_h: 7
    spatial_scale: 0.0625 # 1/16
  }
}
```

原图的候选区域 (x_1, y_1, x_2, y_2) 对应的特征图的区域 (x'_1, y'_1, x'_2, y'_2) 是：

$$x'_1 = \text{round}(x_1 \times \text{spatial_scale})$$

$$y'_1 = \text{round}(y_1 \times \text{spatial_scale})$$

$$x'_2 = \text{round}(x_2 \times \text{spatial_scale})$$

$$y'_2 = \text{round}(y_2 \times \text{spatial_scale})$$

2.3 多任务

Fast-RCNN最重要的贡献是多任务模型的提出，多任务将原来物体检测的多阶段训练简化成一个端到端（end-to-end）的模型。Fast-RCNN有两个任务组成，一个任务是用来对候选区域进行分类，另外一个回归器是用来矫正候选区域的位置。

2.3.1 分类任务 L_{cls}

设 $p = \{p_0, p_1, \dots, p_n\}$ 是候选区域集合，则 L_{cls} 是一个 $K+1$ 类的分类任务，其中输入数据是经过卷积和全连接之后提取的特征向量，输出数据是候选区域的类别 (u)，包括 K 类物体 ($u \geq 1$) 和 1 类背景 ($u = 0$)。分类任务的损失函数是 softmax 损失。

2.3.2 位置精校任务 L_{loc}

对于候选区域所属的类别 u ， $v = \{v_x, v_y, v_w, v_h\}$ 表示候选区域的 **ground-truth**，

$t^u = (t_x^u, t_y^u, t_w^u, t_h^u)$ 表示对候选区域的类别 u ($u \geq 1$) 预测的位置。损失函数是 **smooth L1** 损失，表示为

$$L_{loc}(t^u, v) = \sum_{i \in x, y, w, h} smooth_{L_1}(t_i^u, v_i)$$

其中

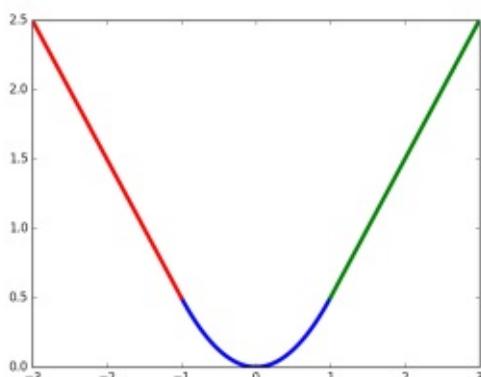
$$smooth_{L_1}(x) = \begin{cases} 0.5x^2 & \text{if } |x| < 1 \\ |x| - 0.5 & \text{otherwise} \end{cases}$$

smooth L1 的形状类似于二次曲线。

smooth L1 同样以 layer 的形式定义在了 caffe 的源码中

`/src/caffe/layers/smooth_L1_loss_layer.cpp`

图2 : smooth L1 曲线



2.3.3 多任务

多任务学习是由两个损失函数和权重 λ 组成，表示为

$$L(p, u, t^u, v) = L_{cls}(p, u) + \lambda[u \leq 1]L_{loc}(t^u, v)$$

在实验中，作者将 λ 统一设成了1。在模型文件中，定义了这两个损失

```
layer {
  name: "loss_cls"
  type: "SoftmaxWithLoss"
  bottom: "cls_score"
  bottom: "labels"
  top: "loss_cls"
  loss_weight: 1
}
layer {
  name: "loss_bbox"
  type: "SmoothL1Loss"
  bottom: "bbox_pred"
  bottom: "bbox_targets"
  bottom: "bbox_loss_weights"
  top: "loss_bbox"
  loss_weight: 1
}
```

根据笔者的经验，训练**Fast-RCNN**时根据两个损失函数的收敛情况适当的调整权值能得到更好的结果。调整的经验是给收敛更慢的那个任务更大的比重。

2.4 SGD训练详解

2.4.1 迁移学习

同**Fast-RCNN**一样，作者同样使用**ImageNet**的数据对模型进行了预训练。详细的讲，首先使用1000类的**ImageNet**训练一个1000类的分类器，如图2的虚线部分。然后提取模型中的特征层以及其以前的所有网络，使用**Fast-RCNN**的多任务模型训练网络，即图2所有的实线部分。

2.4.2 Minibatch training

在**Fast-RCNN**中，设每个batch的大小是R。在抽样时，每次随机选择N个图片，每张图片中随机选择R/N个候选区域，在实验中N=2，R=128。对候选区域进行抽样时，选取25%的正样本（和ground truth的IoU大于0.5），75%的负样本。

3. 物体检测

使用**selective search**输入图像中提取2000个候选区域，按照同训练样本相同的**resize**方法调整候选区域的大小。将所有的候选区域输入到训练好的神经网络，得到每一类的后验概率p和相对偏移r。通过预测概率给每一类一个置信度，并使用**NMS**对每一类确定最终候选区域。
Fast-RCNN使用了奇异值分解来提升矩阵乘法的运算速度。

1. Selective Search 无法通过GPU执行，这是造成Fast R-CNN无法实时的一个重要性能瓶颈。在Faster-RCNN中，对这一瓶颈进行了优化 ↵

Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks

简介

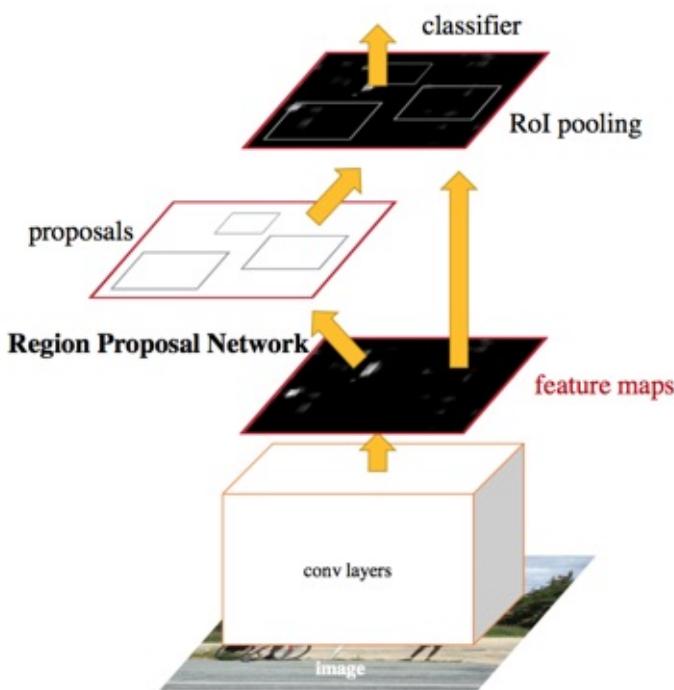
Fast-RCNN [61] 虽然实现了端到端的训练，而且也通过共享卷积的形式大幅提升了R-CNN的计算速度，但是其仍难以做到实时。其中一个最大的性能瓶颈便是候选区域的计算。在之前的物体检测系统中，Selective Search [49] 是最常用候选区域提取方法，它贪心的根据图像的低层特征合并超像素（SuperPixel）。另外一个更快速的版本是EdgeBoxes [43]，虽然EdgeBoxes的提取速度达到了0.2秒一张图片，当仍然难以做到实时，而且EdgeBoxes为了速度牺牲了提取效果。Selective Search速度慢的一个重要原因是不同于检测网络使用GPU进行运算，SS使用的是CPU。从工程的角度讲，使用GPU实现SS是一个非常有效的方法，但是其忽视了共享卷积提供的非常有效的图像特征。

由于卷积网络具有强大的拟合能力，很自然的我们可以想到可以使用卷积网络提取候选区域，由此，便产生了Faster R-CNN最重要的核心思想：RPN (Region Proposal Networks)。通过SPP-net [62] 的实验得知，卷积网络可以很好的提取图像语义信息，例如图像的形状，边缘等等。所以，这些特征理论上也应该能够用提取候选区域（这也符合深度学习解决一切图像问题的思想）。在论文中，作者给RPN的定义如下：RPN是一种可以端到端训练的全卷积网络 [42]，主要是用来产生候选区域。

RPN是通过一个叫做锚点（anchor）的机制实现的。锚点是通过在conv5上进行3*3，步长为1的滑窗，在输入图像上取得的，在取锚点时，同一个中心点取了3个尺度，3个比例共9个锚点。Faster R-CNN的候选区域便是用RPN网络标注了标签的锚点。RPN的思想类似于Attention机制，Attention中where to look要看的地方便是锚点。

RPN产生候选区域，Fast R-CNN使用RPN产生的候选区域进行物体检测，且二者共享卷积网络，这便是Faster R-CNN的框架（图1）。由此可见，RPN和Fast R-CNN是相辅相成的。在论文中，作者使用了Alternative Training的方法训练该网络。

图1：Faster R-CNN结构



算法详解

Faster R-CNN分成两个部分：

1. 使用RPN产生候选区域
2. 使用这些候选区域的Fast R-CNN

Fast R-CNN已经在上一篇文章分析过，下面我们结合论文和源码仔细分析Faster R-CNN。

1. Region Proposal Networks

首先我们要确定RPN网络输入与输出，RPN网络输入是任意尺寸的图像，输出是候选区域和它们的评分（可以理解为置信度），当然，由于RPN是一个多任务的监督学习，所以我们也需要图片的Ground Truth。谈到RPN的多任务，RPN的任务有两个，任务一是用来判断当前锚点是前景的概率和是背景的概率，所以是两个二分类问题¹；任务二用来预测锚点中前景区域的坐标 (x, y, w, h) ，所以是一个回归任务，该回归任务预测四个值。RPN对每一组不同尺度的锚点区域，都会单独的训练一组多损失任务，且这些任务参数不共享。这么做的原因我们会在锚点的生成部分进行讲解。所以，假设有 k 个锚点，RPN网络是一个有 $6 \times k$ 个输出的模型。反映到下面源码中

```

layer {
  name: "rpn_cls_score"
  type: "Convolution"
  bottom: "rpn/output"
  top: "rpn_cls_score"
  param { lr_mult: 1.0 }
  param { lr_mult: 2.0 }
  convolution_param {
    num_output: 18    # 2(bg/fg) * 9(anchors)
    kernel_size: 1 pad: 0 stride: 1
    weight_filler { type: "gaussian" std: 0.01 }
    bias_filler { type: "constant" value: 0 }
  }
}

layer {
  name: "rpn_bbox_pred"
  type: "Convolution"
  bottom: "rpn/output"
  top: "rpn_bbox_pred"
  param { lr_mult: 1.0 }
  param { lr_mult: 2.0 }
  convolution_param {
    num_output: 36    # 4 * 9(anchors)
    kernel_size: 1 pad: 0 stride: 1
    weight_filler { type: "gaussian" std: 0.01 }
    bias_filler { type: "constant" value: 0 }
  }
}

```

1.1 数据准备

理论上输入的图像是任意尺寸，但是在Faster R-CNN中，我们将图像的最短边resize到了600，且最长边不超过1000，在resize的过程中，保持输入图像的长宽比不变。最短边固定为600是为了防止经过若干次降采样后特征图的尺寸过小，最大边不超过1000是为了防止图像太大影响检测速度。

1.2 多任务损失

1.2.1 分类任务 L_{cls}

在分类任务中，如果锚点和Ground Truth的IoU大于0.7，该锚点为正样本；如果IoU小于0.3，则该样本为负样本。可见，正负样本并不是互斥的，因为还存在一些介于正负样本直接的锚点（IoU在0.3到0.7之间），所以并不能简单的化成一个二分类任务。设 p_i 是预测的类别，如

果锚点为正， $p_i^* = 0$ ，如果锚点为负 $p_i^* = 1$ ，其它介于正负锚点之间的样本不参与模型训练。

$L_{cls}(p_i, p_i^*)$ 是log损失。

1.2.2 回归任务 L_{reg}

RPN的另外一个任务是预测四个值的回归任务，即预测 (x, y, w, h) ，假设锚点表示为

(x_a, y_a, w_a, h_a) ，ground truth是 (x^*, y^*, w^*, h^*) ，将这些坐标信息参数化为

$$t_x = (x - x_a)/w_a, t_y = (y - y_a)/h_a$$

$$t_w = \log(w/w_a), t_h = (h/h_a)$$

$$t_x^* = (x^* - x_a)/w_a, t_y^* = (y^* - y_a)/h_a$$

$$t_w^* = \log(w^*/w_a), t_h^* = (h^*/h_a)$$

损失函数 $L_{reg}(t_i, t_i^*)$ 使用的是Fast R-CNN中定义的smooth L1 loss。上面的参数化可以理解为预测ground truth和锚点位置的相对尺寸和位置。

1.2.3 损失函数

多任务损失的损失函数表示为 (i表示minibatch中第i个样本)

$$L(p_i, t_i) = \frac{1}{N_{cls}} \sum_i L_{cls}(p_i, p_i^*) + \lambda \frac{1}{N_{reg}} \sum_i p_i^* L_{reg}(t_i, t_i^*)$$

为了平衡两个任务， λ 的值设为了10。同Fast R-CNN一样，为了达到更好的训练结果， λ 需要根据损失函数的下降情况以及模型在验证集上面的表现针对性的调整。

1.3 锚点生成

如何生成锚点是Faster R-CNN最重要也是最难理解的部分，网上很多博客的理解并不正确或者讲解的不够透彻，下面我们来开始详细分析这一部分。

首先，RPN的滑窗（步长是1）是在特征层即conv5层进行的，然后通过3*3*256(ZF-Net是256，VGG-16是512)的卷积核将该窗口内容映射为特征向量（下图中的256-d即为特征向量）。

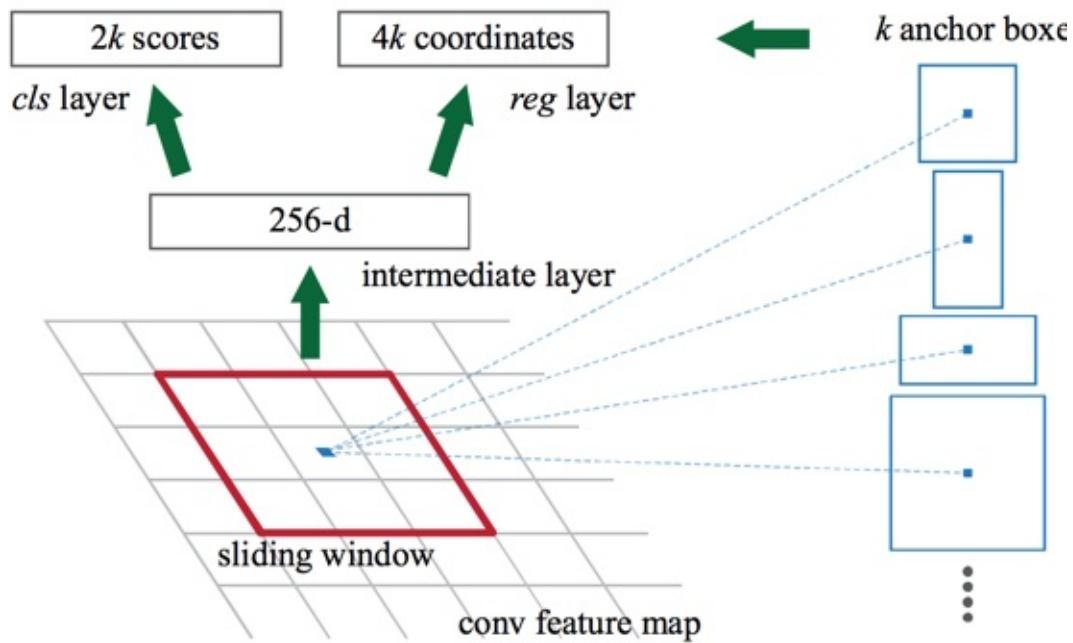


图2：RPN的滑窗

根据SPP-net所介绍的感受野的知识，特征图上的一点对应的感受野是输入图像的一个区域，该区域可以根据卷积网络结构反向递推得出。递推公式为：

$$rfsiz = (out - 1) \times stride + ksize$$

根据这个递推公式我们可以计算出VGG-16的感受野的大小是228（论文中有这个数字），在VGG的卷积层中

$$rfsiz = (out - 1) \times 1 + 3 = out + 2$$

在pooling层中

$$rfsiz = (out - 1) \times 2 + 2 = out * 2$$

递推过程总结如下图

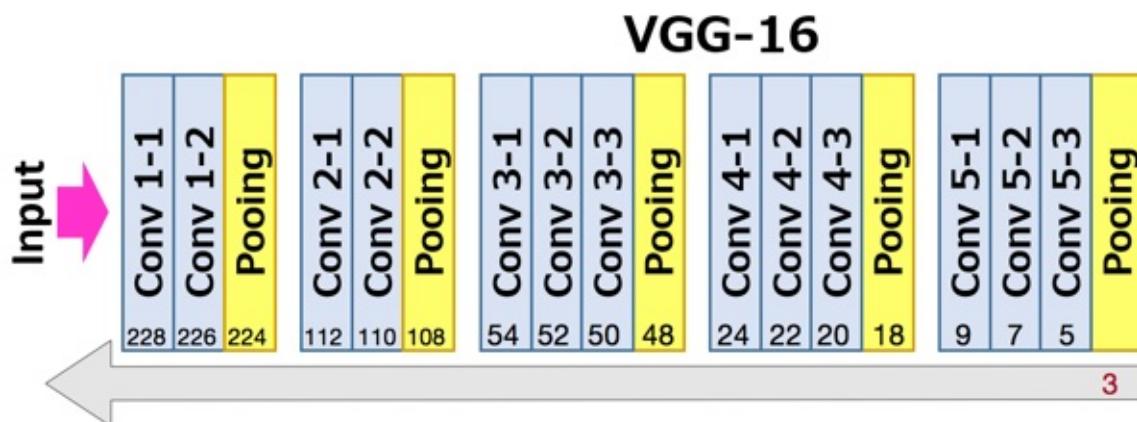


图3：VGG感受野的计算

由于RPN的特征向量是从conv5经过卷积核为3的卷积获得，所以应该从3开始向前推。

根据卷积的位移不变性，将conv5映射到输入图像感受野的中心点只需要乘以降采样尺度即可，由于VGG使用的都是same卷积，降采样尺度等于所有pooling的步长的积，即：

$$_feat_stride = \prod_i pool_stride = 2 * 2 * 2 * 2 = 16$$

相对位移便是特征图上的位移除以降采样尺度

```
shift_x = np.arange(0, width) * self._feat_stride # _feat_stride = 16
shift_y = np.arange(0, height) * self._feat_stride
```

所以，在特征图的步长为1的滑窗也可以理解为在输入图像上步长为`_feat_stride`的滑窗。例如一个最短边resize到600的4:3的输入图像，经过4次降采样后，特征图的大小为

$W * H = (600/16) * (800/16) = 38 * 50 \approx 2k$ 。步长为1的滑窗后，得到了 $W \times H \times k$ 个锚点，这便是论文中锚点个数的由来，由于部分锚点边界超过了图像，这部分锚点会被忽略，所以并不是所有锚点都参与抽样。

根据感受野的中心，每个中心取9个锚点，这9个锚点有3个尺度 128^2 , 256^2 和 512^2 ，每个尺度有3个比例1:1, 1:2, 2:1。代码中锚点的坐标为：

```
[[ -84.  -40.   99.   55.]
 [-176. -88.  191.  103.]
 [-360. -184.  375.  199.]
 [-56.  -56.   71.   71.]
 [-120. -120.  135.  135.]
 [-248. -248.  263.  263.]
 [-36.  -80.   51.   95.]
 [-80. -168.   95.  183.]
 [-168. -344.  183.  359.]]
```

可视化该锚点，得到下图，黄色部分代表中心点的感受野。

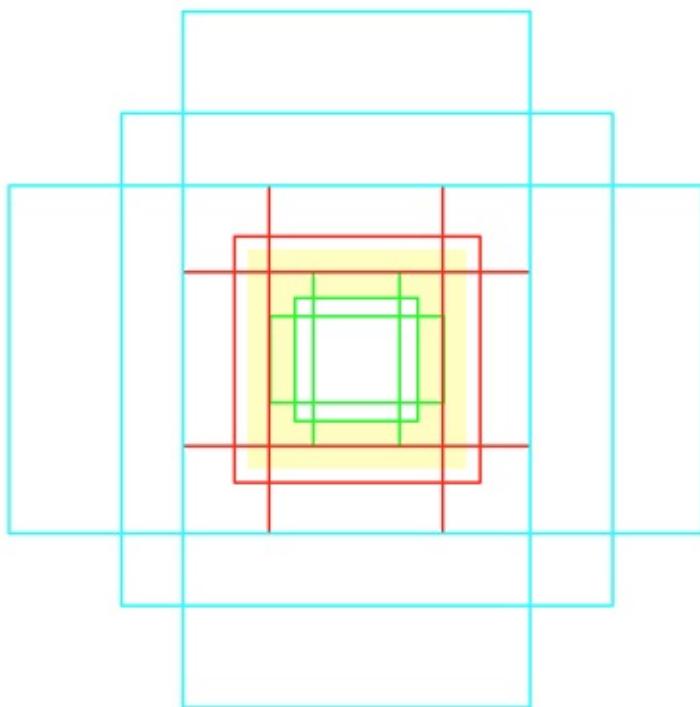


图4：锚点可视化

作者在论文中说这种锚点并没有经过精心设计，我认为，这批锚点表现好不是没有原因的，三中锚点分别包括被感受野包围，和感受野类似以及将感受野覆盖三种情况，可见这样设计锚点覆盖的情况还是非常全面的。

由于每个中心对应一个256维的特征向量，而1个中心对应了9个不同的锚点，进而产生不同的标签。这似乎是一个1 vs n的映射，而这种方程是无解的。实际上，作者根据9种不同尺寸和比例的锚点，独立的训练9个不同的回归模型，这些模型的参数是不共享的。这就是RPN的模型为什么有 $6 \times k$ 个输出的原因，如图5。

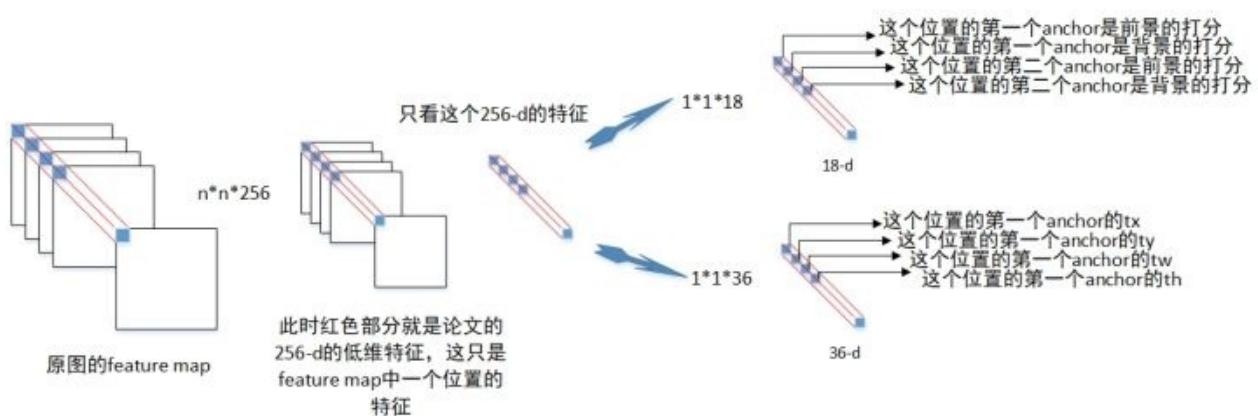


图5：RPN的输出层

一个重要的问题是同一个中心点产生9个不同尺度的锚点，每个锚点对应一个任务的分支，那么怎么做到不同尺寸的锚点训练不同的分支的呢？在损失函数的计算中，所有锚点的特征向量均参与计算概率和位移，只是根据采样的样本来调整一部分分支的权值。

```

def train_model(self, sess, max_iters):
    """Network training loop."""

    data_layer = get_data_layer(self.roidb, self.imdb.num_classes)

    # RPN
    # classification loss
    rpn_cls_score = tf.reshape(self.net.get_output('rpn_cls_score_reshape'), [-1, 2])
)
    rpn_label = tf.reshape(self.net.get_output('rpn-data')[0], [-1])
    # 提取采样的样本
    rpn_cls_score = tf.reshape(tf.gather(rpn_cls_score, tf.where(tf.not_equal(rpn_label, -1))), [-1, 2])
    rpn_label = tf.reshape(tf.gather(rpn_label, tf.where(tf.not_equal(rpn_label, -1))), [-1])
    rpn_cross_entropy = tf.reduce_mean(tf.nn.sparse_softmax_cross_entropy_with_logits(logits=rpn_cls_score, labels=rpn_label))

```

能这样做的原因是由于 1×1 卷积替代了全连接，这种只有卷积的网络结构叫做全卷积（PS: 先挖一坑）。

1.4 RPN的训练

RPN使用的是"Image-centric"的采样方法，即每次采样少量的图片，然后从图片中随机采样正负锚点样本。具体的，RPN每次随机采样一张图片，然后再每张图片中采样256个锚点，并尽量保证正负样本的比例是1:1。由于负样本的数量基本是上是超过128的，所以当正样本数量不够时，使用负样本补充。

一些超参数设置如下，使用Image-Net的结果初始化网络，前6k代，学习率是0.001，后20k代，学习率0.0001。遗忘因子momentum是0.9，权值衰减系数是0.0006。

2. RPN和Fast R-CNN的训练

由于RPN使用Fast R-CNN的网络模型可以更好的提取候选区域，而Fast R-CNN可以使用RPN产生的候选区域进行物体检测，两者相辅相成，作者尝试了多种模型训练策略，并最终采用了Alternating Training。

Alternating Training 可以分成4个步骤

1. 使用无监督学习即Imagenet的训练结果初始化网络训练RPN；
2. 使用RPN产生的候选区域训练Fast R-CNN，Fast R-CNN和RPN使用的是两个独立的网络，也是由Image-Net任务进行初始化；
3. 使用Fast R-CNN的网络初始化RPN，但是共享的卷积层固定，只调整RPN独有的网络；
4. 固定共享的卷积部分，训练Fast R-CNN。

第2条中我们讲到了使用RPN产生候选区域，下面介绍这一过程。在第一部分我们指出RPN的输出是锚点的正负评分以及预测坐标，在计算候选区域时

1. 所有的在图像内部的锚点均输入训练好的网络模型，得到样本评分和预测坐标；
2. 使用NMS根据评分过滤锚点，NMS的IoU阈值固定为0.7，之后产生的便是候选区域。

在作者开源的代码中，作者使用的是近似联合训练（Approximate joint training），即将RPN和Fast R-CNN的损失函数简单的加在一起，作为一个多任务的损失函数进行学习。作者也指出，这种方法忽略了Fast R-CNN将RPN的输出作为其输入的这一事实。实际上，Faster R-CNN的RPN和Fast R-CNN并不是一个并行的多任务的关系，而是一个串行级联的关系，图6说明了并行多任务和串行级联的区别，在何凯明的另外一篇论文中专门介绍了该如何处理这种多任务的问题（PS：又挖一坑）。

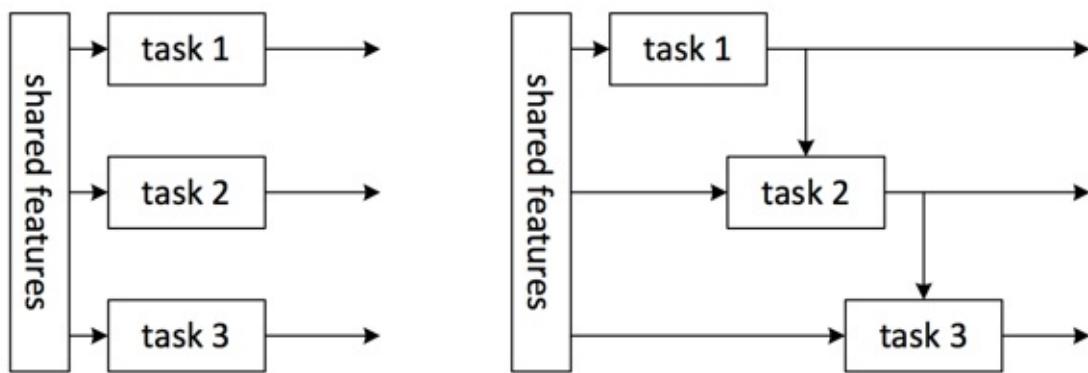


图6：并行多任务和串行多任务的区别

3. Faster R-CNN的检测

和之前讲到的一样，使用RPN产生完候选区域后，剩下的便和Fast R-CNN一样了。

R-FCN: Object Detection via Region-based Fully Convolutional Networks

简介：

位移不变性是卷积网络一个重要的特征，该特征使得卷积网格在图像分类任务上取得了非常好的效果，所谓唯一不变性，是指图片中物体的位置对图片的分类没有影响。但是在物体检测的场景中，我们需要知道检测物体的具体位置，这时候我们需要网络对物体的位置非常敏感，即我们需要网络具有“位移可变性”。R-FCN [59] 的提出便是解决分类任务中位移不变性和检测任务中位移可变性直接的矛盾的。

同时，作者分析了Faster R-CNN [60] 存在的性能瓶颈，即ROI之后使用Fast R-CNN [61] 对RPN提取的候选区域进行分类和位置精校。在R-FCN中，ROI之后便不存在可学习的参数，从而将Faster-RCNN的速度提高了2.5-20倍。

1. 动机

在R-CNN系列论文中，物体检测一般分成两步：

1. 提取候选区域；
2. 候选区域分类和位置精校。

在R-FCN之前，state-of-the-art的Faster-RCNN使用RPN网络进行候选区域（Proposal Region）选择，然后再使用Fast R-CNN进行分类。在Faster R-CNN中，首先使用ROI层将不同大小的候选区域归一化到统一大小，之后接若干个全连接层，最后使用一个任务作为损失函数。多任务包含两个子任务：

1. softmax的分类任务；
2. 用于位置精校的回归任务

Faster R-CNN之所以这样做主要是因为其使用了VGG [120] 作为特征提取器。在第一章中，我们了解到VGG之后的GoogLeNet [121] 和ResNet [118] 均是使用了全卷积的结构，即使用 1×1 卷积代替全连接。 1×1 卷积具备全连接层增加非线性性的作用，同时还保证了特征点的位置敏感性。可见在物体检测任务中引入 1×1 卷积会非常有帮助的。

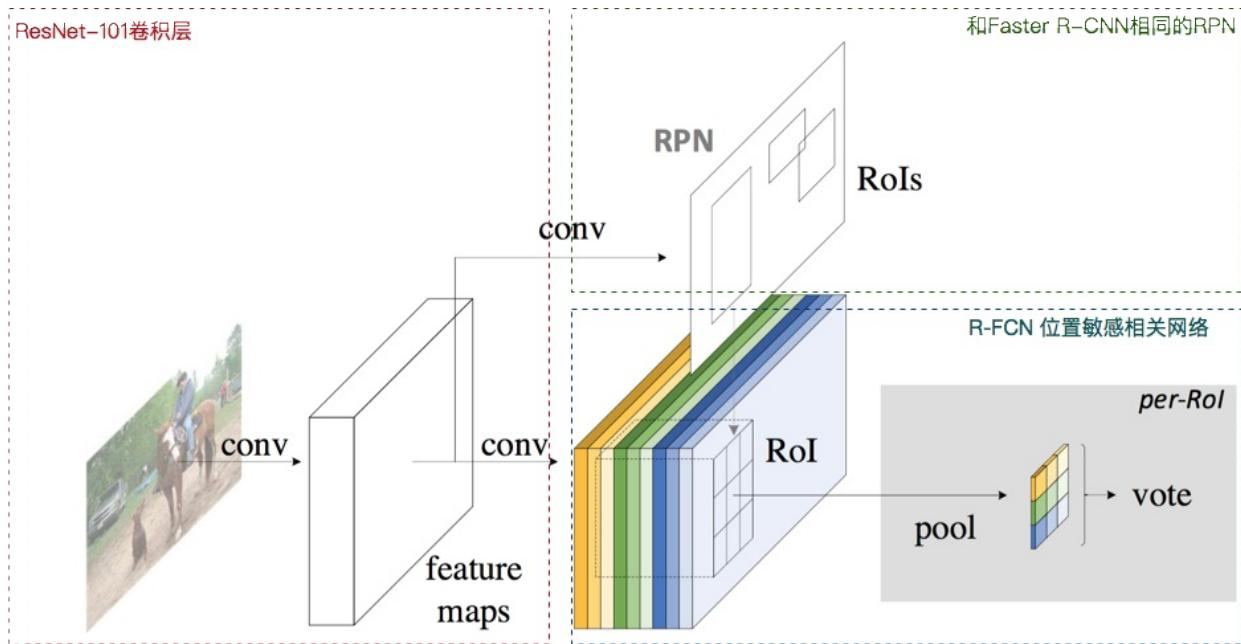
在Faster R-CNN中，为了保证特征的“位移敏感性”，作者根据RPN提取了约2000个候选区域，然后使用全连接层计算损失函数，然而候选区域有大量的特征冗余，造成了一部分计算资源的浪费。

R-FCN采用了和Faster R-CNN相同的过程，在R-FCN中做了如下改进

1. 模仿FCN，R-FCN采用了全卷积的结构；
2. R-FCN的两个阶段的网络参数全部共享；
3. 使用位置敏感网络产生检测框；
4. 位置敏感网络无任何可学习的参数。

R-FCN的结构如图1

图1：R-FCN核心思想

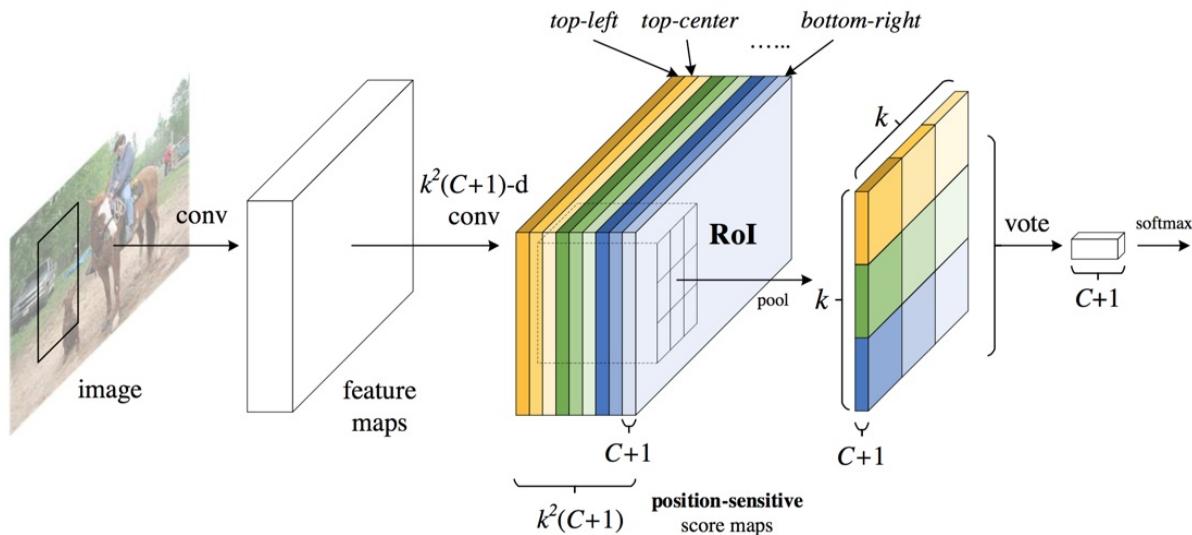


在图1中， C 表示物体检测中物体的类别数目。在R-FCN中，一个ROI会被分成 $k \times k$ 个bin。下面我们将详细解析R-FCN。

2. R-FCN详解

R-FCN采用了和Faster R-CNN相同的框架（图2），关于Faster R-CNN的解释，可以参考论文或者我的[解析](#)。

图2：R-FCN流程图 $x = y$



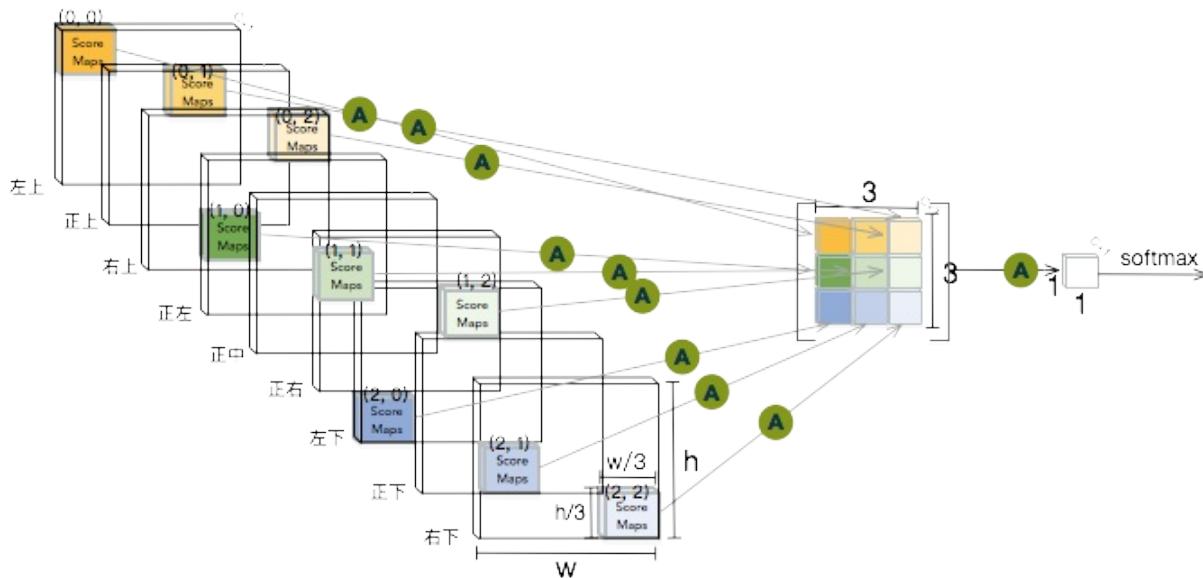
2.1 骨干架构 (backbone architecture)

R-FCN使用的是残差网络的ResNet-101[6]结构，ResNet-101采用的是100层卷积+Global Averaging Pooling (GAP) +fc分类器的结构，ResNet101卷积的最后一层的Feature Map的个数是2048。在R-FCN中，去掉了ResNet的GAP层和fc层，并在最后一个卷积层之后使用1024个 $1 \times 1 \times 2048$ 卷积降维到1024-d，然后再使用 $k^2 \times (C + 1)$ 个 $1 \times 1 \times 1024 - d$ 的卷积生成 $k^2 \times (C + 1) - d$ 的位置敏感卷积层。其中ResNet部分使用在ImageNet上训练好的模型作为初始化参数。

2.2 位置敏感网络

图1和图2中ResNet之后接的便是位置敏感网络。该层的大小和ResNet-101最后一层的大小相同，维度是 $k^2 \times (C + 1)$ 。 $C + 1$ 为类别数，表示 C 类物体加上1类背景。 k 是一个超参数，表示把ROI划分grid的单位，一般情况下， $k = 3$ 。在R-FCN中，一个ROI区域会被等比例划分成一个 $k \times k$ 的grid，每个位置为一个bin，分别表示该grid对应的物体的敏感位置（左上，正上，右上，正左，正中，正右，左下，正下，右下）编码（图3）。

图3：图解位置敏感ROI Pooling过程($k=3$)



对于一个尺寸为 $w \times h$ 的 ROI 区域，每个 bin 的大小约为 $\frac{w}{k} \times \frac{h}{k}$ 。在第 $(i, j)^{th}$ bin 中 $(0 \leq i, j < k)$ 中，定义了一个只作用于该 bin 的位置敏感 ROI 池化 (position-sensitive ROI pooling)，即求位置敏感分值图 (position-sensitive score maps) 中每个 bin 的均值

$$r_c(i, j | \theta) = \frac{1}{n} \sum_{(x, y) \in \text{bin}(i, j)} z_{i, j, c}(x + x_0, y + y_0 | \theta)$$

在上式中， θ 表示整个网络所有需要学习的参数， $r_c(i, j | \theta)$ 表示第 c 类物体在第 (i, j) 个 bin 处的响应值， $z_{i, j, c}(x + x_0, y + y_0 | \theta)$ 表示在位置敏感分值图中每个 bin 对应的横跨特征图中 $\lfloor i \frac{w}{k} \rfloor \leq x < \lceil (i+1) \frac{w}{k} \rceil$ 和 $\lfloor j \frac{h}{k} \rfloor \leq y < \lceil (j+1) \frac{h}{k} \rceil$ 的部分特征值。

如图3所示，一个维度为 $w \times h \times [k^2 \times (C + 1)]$ 的 ROI 区域可以展开成 k^2 个 $w \times h \times (C + 1)$ 个 ROI 区域，每个 ROI 区域的第 (i, j) 个 grid 对应物体的一个不同的敏感位置，这样我们可以提取 k^2 个维度为 $\frac{w}{k} \times \frac{h}{k} \times (C + 1)$ 的分值图，每个分值图求均值¹之后再整合到一起便得到了一个 $k^2 \times (C + 1)$ 的位置敏感分值。对该位置敏感分值的 k^2 个区域求均值得到一个 $1 \times 1 \times (C + 1)$ 的向量，使用 softmax 函数（注意不是 softmax 分类器）便可以得到每个类别的概率值。至此，R-FCN 的分类任务介绍完毕。

现在开始介绍分类任务，在 ResNet101 之后，使用 4 个 $1 \times 1 \times 1024$ 卷积得到一个维度维 4 的卷积。该卷积层对应的 ROI 区域根据物体的位置分成 k^2 个 bin，这样便得到了一个 $4 \times k^2$ 的特征向量，物体的位置信息 (x, y, w, h) 通过平均或者投票的方式可以得到。注意 R-FCN 得到的位置信息和物体的类别没有关系，这一点和 Faster R-CNN 是不同的。

2.3 R-FCN 的训练

R-FCN也是采用了分类和回归的多任务损失函数：

$$L(s, t_{x,y,w,h}) = L_{cls} + \lambda[c^* > 0]L_{reg}(t, t*)$$

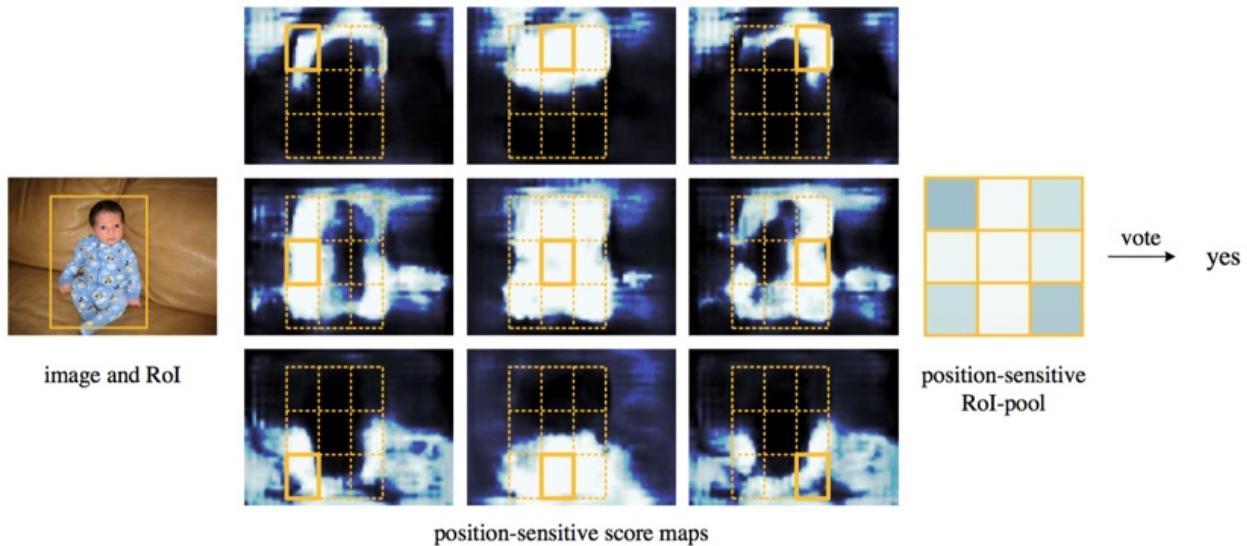
其中 c^* 表示分类得到的ROI区域的类别； $[c^* > 0]$ 表示如果括号内的判断满足，结果为1，否则结果为0； λ 为多任务的比重，是一个需要根据模型的收敛效果调整的超参数； L_{cls} 为softmax分类损失函数， L_{reg} 为bounding box回归损失函数。

2.4 R-FCN结果可视化

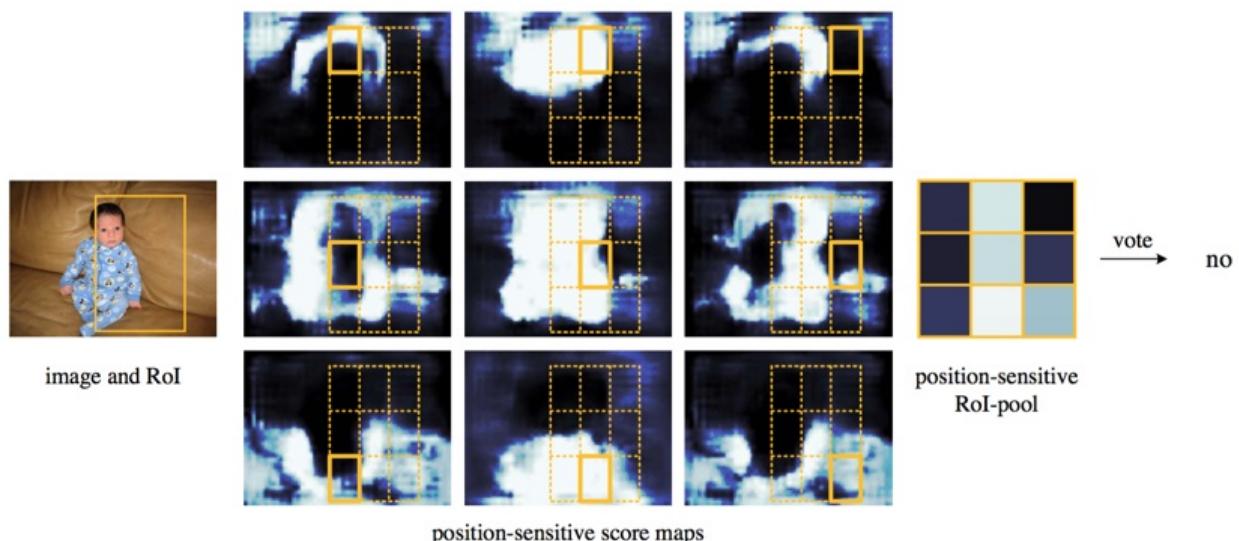
图4中的可视化展示了R-FCN的bin的工作原理，如果ROI区域能够比较精确的框住物体的位置（图4.a），那么每个bin对应的Feature Map都应该能得到非常高的响应；如果ROI区域（图4.b）的定位的不是非常准确，那么部分bin的响应就不是很明显，那么通过投票或者求均值的方法就能筛选出更精确的检测框。

图4：R-FCN可视化

(a)



(b)



Mask R-CNN

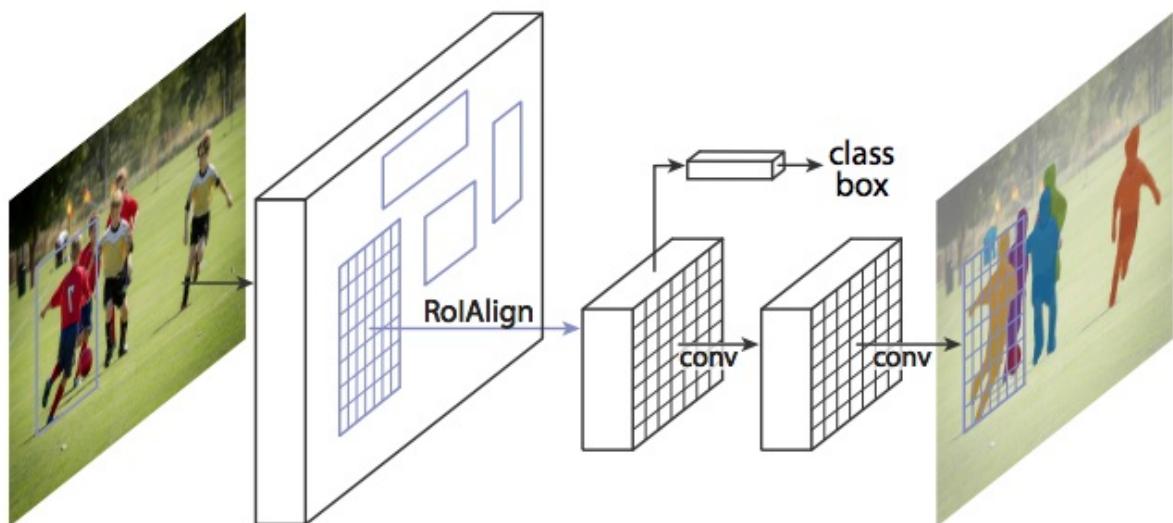
前言

个人非常喜欢何凯明的文章，两个原因，1) 简单，2) 好用。对比目前科研届普遍喜欢把问题搞复杂，通过复杂的算法尽量把审稿人搞蒙从而提高论文的接受率的思想，无论是著名的残差网络还是这篇 Mask R-CNN，大神的论文尽量遵循著名的奥卡姆剃刀原理：即在所有能解决问题的算法中，选择最简单的那个。霍金在出版《时间简史》中说“书里每多一个数学公式，你的书将会少一半读者”。Mask R-CNN [57] 更是过分到一个数学公式都没有，而是通过对问题的透彻的分析，提出针对性非常强的解决方案，下面我们来一睹Mask R-CNN的真容。

动机

语义分割和物体检测是计算机视觉领域非常经典的两个重要应用。在语义分割领域，FCN [42] 是代表性的算法；在物体检测领域，代表性的算法是Faster R-CNN [60]。很自然的会想到，结合FCN和Faster R-CNN不仅可以是模型同时具有物体检测和语义分割两个功能，还可以是两个功能互相辅助，共同提高模型精度，这便是Mask R-CNN的提出动机。Mask R-CNN的结构如图1

图1：Mask R-CNN框架图



如图1所示，Mask R-CNN分成两步：

1. 使用RPN网络产生候选区域；
2. 分类，bounding box，掩码预测的多任务损失。

在Fast R-CNN的解析文章中，我们介绍Fast R-CNN采用ROI池化来处理候选区域尺寸不同的问题。但是对于语义分割任务来说，一个非常重要的要求便是特征层和输入层像素的一对一，ROI池化显然不满足该要求。为了改进这个问题，作者仿照STN [41]中提出的双线性插值提出了ROIAlign，从而使Faster R-CNN的特征层也能进行语义分割。

下面我们结合代码详细解析Mask R-CNN，代码我使用的是基于TensorFlow和Keras实现的版本：https://github.com/matterport/Mask_RCNN。

Mask R-CNN详解

1. 骨干架构 (FPN)

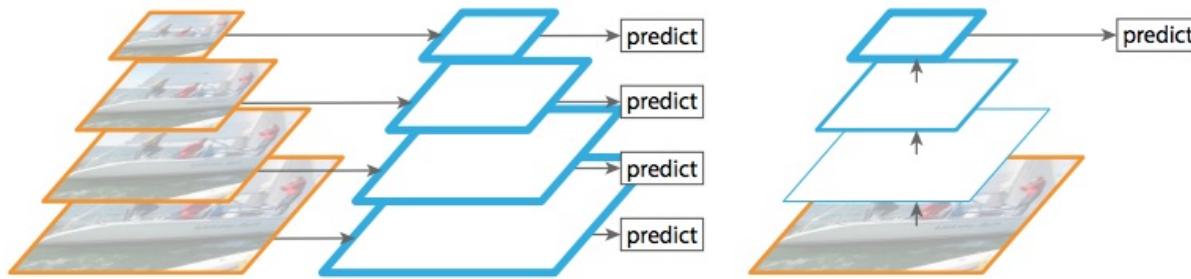
在第一章中，我们介绍过卷积网络的一个重要特征：深层网络容易响应语义特征，浅层网络容易响应图像特征。但是到了物体检测领域，这个特征便成了一个重要的问题，高层网络虽然能响应语义特征，但是由于Feature Map的尺寸较小，含有的几何信息并不多，不利于物体检测；浅层网络虽然包含比较多的几何信息，但是图像的语义特征并不多，不利于图像的分类，这个问题在小尺寸物体检测上更为显著和，这也就是为什么物体检测算法普遍对小物体检测效果不好的最重要原因之一。很自然地可以想到，使用合并了的深层和浅层特征来同时满足分类和检测的需求。

Mask R-CNN的骨干框架使用的是该团队在CVPR2017的另外一篇文章FPN [58]。FPN使用的是图像金字塔的思想以解决物体检测场景中小尺寸物体检测困难的问题，传统的图像金字塔方法（图2.a）采用输入多尺度图像的方式构建多尺度的特征，该方法的最大问题便是识别时间为单幅图的 k 倍，其中 k 是缩放的尺寸个数。Faster R-CNN等方法为了提升检测速度，使用了单尺度的Feature Map（图2.b），但单尺度的特征图限制了模型的检测能力，尤其是训练集中覆盖率极低的样本（例如较大和较小样本）。不同于Faster R-CNN只使用最顶层的Feature Map，SSD [54]利用卷积网络的层次结构，从VGG的第conv4_3开始，通过网络的不同层得到了多尺度的Feature Map（图2.c），该方法虽然能提高精度且基本上没有增加测试时间，但没有使用更加低层的Feature Map，然而这些低层次的特征对于检测小物体是非常有帮助的。

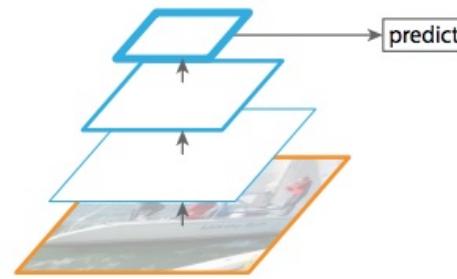
针对上面这些问题，FPN采用了SSD的金字塔内Feature Map的形式。与SSD不同的是，FPN不仅使用了VGG中层次深的Feature Map，并且浅层的Feature Map也被应用到FPN中。并通过自底向上（bottom-up），自顶向下（top-down）以及横向连接（lateral connection）将这些Feature Map高效的整合起来，在提升精度的同时并没有大幅增加检测时间（图2.d）。

通过将Faster R-CNN的RPN和Fast R-CNN的骨干框架换成FPN，Faster R-CNN的平均精度从51.7%提升到56.9%。

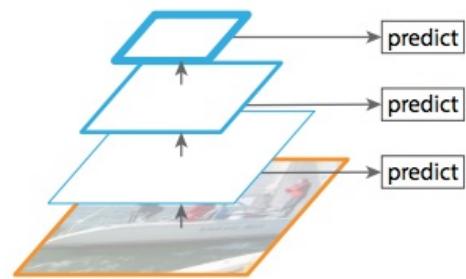
图2：金字塔特征的几种形式。



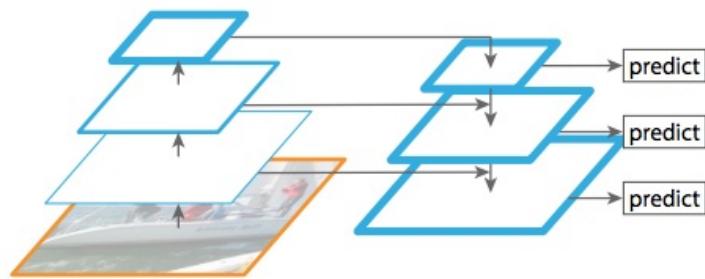
(a) Featurized image pyramid



(b) Single feature map



(c) Pyramidal feature hierarchy



(d) Feature Pyramid Network

FPN的代码出现在 `./mrcnn/model.py` 中，核心代码如下：

代码片段1：FPN结构

```

# Build the shared convolutional layers.
# Bottom-up Layers
# Returns a list of the last layers of each stage, 5 in total.
# Don't create the thead (stage 5), so we pick the 4th item in the list.
if callable(config.BACKBONE):
    _, C2, C3, C4, C5 = config.BACKBONE(input_image, stage5=True, train_bn=config.TRAIN_BN)
else:
    _, C2, C3, C4, C5 = resnet_graph(input_image, config.BACKBONE, stage5=True, train_bn=config.TRAIN_BN)
# Top-down Layers
# TODO: add assert to verify feature map sizes match what's in config
P5 = KL.Conv2D(config.TOP_DOWN_PYRAMID_SIZE, (1, 1), name='fpn_c5p5')(C5)
P4 = KL.Add(name="fpn_p4add")([
    KL.UpSampling2D(size=(2, 2), name="fpn_p5upsampled")(P5),
    KL.Conv2D(config.TOP_DOWN_PYRAMID_SIZE, (1, 1), name='fpn_c4p4')(C4)])
P3 = KL.Add(name="fpn_p3add")([
    KL.UpSampling2D(size=(2, 2), name="fpn_p4upsampled")(P4),
    KL.Conv2D(config.TOP_DOWN_PYRAMID_SIZE, (1, 1), name='fpn_c3p3')(C3)])
P2 = KL.Add(name="fpn_p2add")([
    KL.UpSampling2D(size=(2, 2), name="fpn_p3upsampled")(P3),
    KL.Conv2D(config.TOP_DOWN_PYRAMID_SIZE, (1, 1), name='fpn_c2p2')(C2)])
# Attach 3x3 conv to all P layers to get the final feature maps.
P2 = KL.Conv2D(config.TOP_DOWN_PYRAMID_SIZE, (3, 3), padding="SAME", name="fpn_p2")(P2)
P3 = KL.Conv2D(config.TOP_DOWN_PYRAMID_SIZE, (3, 3), padding="SAME", name="fpn_p3")(P3)
P4 = KL.Conv2D(config.TOP_DOWN_PYRAMID_SIZE, (3, 3), padding="SAME", name="fpn_p4")(P4)
P5 = KL.Conv2D(config.TOP_DOWN_PYRAMID_SIZE, (3, 3), padding="SAME", name="fpn_p5")(P5)
# P6 is used for the 5th anchor scale in RPN. Generated by
# subsampling from P5 with stride of 2.
P6 = KL.MaxPooling2D(pool_size=(1, 1), strides=2, name="fpn_p6")(P5)

# Note that P6 is used in RPN, but not in the classifier heads.
rpn_feature_maps = [P2, P3, P4, P5, P6]
mrcnn_feature_maps = [P2, P3, P4, P5]

```

1.1 自底向上路径

自底向上方法反映在上面代码的第6行或者第8行，自底向上即是卷积网络的前向过程，在Mask R-CNN中，用户可以根据配置文件选择使用ResNet-50或者ResNet-101。代码中的`resnet_graph`就是一个残差块网络，其返回值C2，C3，C4，C5，是每次池化之后得到的Feature Map，该函数也实现在`./mrcnn/model.py`中（代码片段2）。需要注意的是在残差网络中，C2，C3，C4，C5经过的降采样次数分别是2，3，4，5即分别对应原图中的步长分别是4，8，16，32。

代码片段2：残差网络

```

def resnet_graph(input_image, architecture, stage5=False, train_bn=True):
    """Build a ResNet graph.
    architecture: Can be resnet50 or resnet101
    stage5: Boolean. If False, stage5 of the network is not created
    train_bn: Boolean. Train or freeze Batch Norm layres
    """
    assert architecture in ["resnet50", "resnet101"]
    # Stage 1
    x = KL.ZeroPadding2D((3, 3))(input_image)
    x = KL.Conv2D(64, (7, 7), strides=(2, 2), name='conv1', use_bias=True)(x)
    x = BatchNorm(name='bn_conv1')(x, training=train_bn)
    x = KL.Activation('relu')(x)
    C1 = x = KL.MaxPooling2D((3, 3), strides=(2, 2), padding="same")(x)
    # Stage 2
    x = conv_block(x, 3, [64, 64, 256], stage=2, block='a', strides=(1, 1), train_bn=train_bn)
    x = identity_block(x, 3, [64, 64, 256], stage=2, block='b', train_bn=train_bn)
    C2 = x = identity_block(x, 3, [64, 64, 256], stage=2, block='c', train_bn=train_bn)
    # Stage 3
    x = conv_block(x, 3, [128, 128, 512], stage=3, block='a', train_bn=train_bn)
    x = identity_block(x, 3, [128, 128, 512], stage=3, block='b', train_bn=train_bn)
    x = identity_block(x, 3, [128, 128, 512], stage=3, block='c', train_bn=train_bn)
    C3 = x = identity_block(x, 3, [128, 128, 512], stage=3, block='d', train_bn=train_bn)
    # Stage 4
    x = conv_block(x, 3, [256, 256, 1024], stage=4, block='a', train_bn=train_bn)
    block_count = {"resnet50": 5, "resnet101": 22}[architecture]
    for i in range(block_count):
        x = identity_block(x, 3, [256, 256, 1024], stage=4, block=chr(98 + i), train_bn=train_bn)
    C4 = x
    # Stage 5
    if stage5:
        x = conv_block(x, 3, [512, 512, 2048], stage=5, block='a', train_bn=train_bn)
        x = identity_block(x, 3, [512, 512, 2048], stage=5, block='b', train_bn=train_bn)
        C5 = x = identity_block(x, 3, [512, 512, 2048], stage=5, block='c', train_bn=train_bn)
    else:
        C5 = None
    return [C1, C2, C3, C4, C5]

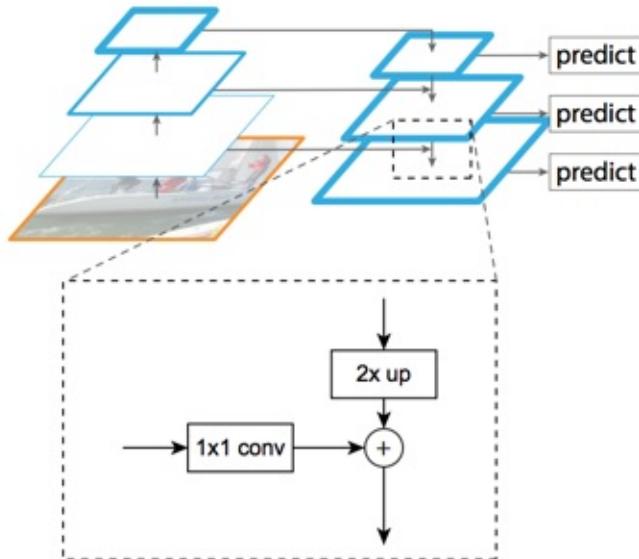
```

这里之所以没有使用C1，是考虑到由于C1的尺寸过大，训练过程中会消耗很多的显存。

1.2 自顶向下路径和横向连接

通过自底向上路径，FPN得到了四组Feature Map。浅层的Feature Map如C2含有更多的纹理信息，而深层的Feature Map如C5含有更多的语义信息。为了将这四组倾向不同特征的Feature Map组合起来，FPN使用了自顶向下及横向连接的策略，图3。

图3：FPN的自顶向上路径和横向连接



残差网络得到的C1-C5由于经历了不同的降采样次数，所以得到的Feature Map的尺寸也不同。为了提升计算效率，首先FPN使用1*1进行了降维，得到P5，然后使用双线性插值进行上采样，将P5上采样到和C4相同的尺寸。

之后，FPN也使用1*1卷积对P4进行了降维，由于降维并不改变尺寸大小，所以P5和P4具有相同的尺寸，FPN直接把P5单位加到P4得到了更新后的P4。基于同样的策略，我们使用P4更新P3，P3更新P2。这整个过程是从网络的顶层向下层开始更新的，所以叫做自顶向下路径。

FPN使用单位加的操作来更新特征，这种单位加操作叫做横向连接。由于使用了单位加，所以P2，P3，P4，P5应该具有相同数量的Feature Map（源码中该值为256），所以FPN使用了1*1卷积进行降维。

在更新完Feature Map之后，FPN在P2，P3，P4，P5之后均接了一个3*3卷积操作（代码片段1第22-25行），该卷积操作是为了减轻上采样的混叠效应（aliasing effect）¹。

2. 两步走策略

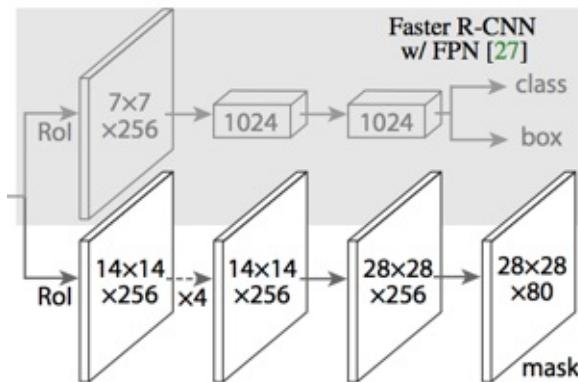
Mask R-CNN采用了和Faster R-CNN相同的两步走策略，即先使用RPN提取候选区域，关于RPN的详细介绍，可以参考Faster R-CNN一文。不同于Faster R-CNN中使用分类和回归的多任务回归，Mask R-CNN在其基础上并行添加了一个用于语义分割的Mask损失函数，所以Mask R-CNN的损失函数可以表示为下式。

$$L = L_{cls} + L_{box} + L_{mask}$$

上式中， L_{cls} 表示bounding box的分类损失值， L_{box} 表示bounding box的回归损失值， L_{mask} 表示mask部分的损失值，图4。在这份源码中，作者使用了近似联合训练（Approximate Joint Training），所以损失函数会由也会加上RPN的分类和回归loss。这一部分代码

在 `./mrcnn/model.py` 的2004-2025行。 L_{cls} 和 L_{box} 的计算方式与Faster R-CNN相同，下面我们重点讨论 L_{mask} 。

图4：Mask R-CNN的损失函数



在进行掩码预测时，FCN的分割和预测是同时进行的，即要预测每个像素属于哪一类。而Mask R-CNN将分类和语义分割任务进行了解耦，即每个类单独的预测一个掩码，这种解耦提升了语义分割的效果，从图5上来看，提升效果还是很明显的。

图5：Mask R-CNN解耦分类和分割的精度提升

	AP	AP ₅₀	AP ₇₅
softmax	24.8	44.1	25.1
sigmoid	30.3	51.2	31.5
	+5.5	+7.1	+6.4

(b) Multinomial vs. Independent Masks

(ResNet-50-C4): *Decoupling* via per-class binary masks (sigmoid) gives large gains over multinomial masks (softmax).

所以Mask R-CNN基于FCN将ROI区域映射成为一个 $m \times m \times nb_class$ (FCN是 $m \times m$) 的特征层，例如图4中的 $28 \times 28 \times 80$ 。由于每个候选区域的分割是一个二分类任务，所以 L_{mask} 使用的是二值交叉熵 (`binary_crossentropy`) 损失函数，对应的代码为 (1182-1184行)

代码片段3： L_{mask}

```
loss = K.switch(tf.size(y_true) > 0,
                K.binary_crossentropy(target=y_true, output=y_pred),
                tf.constant(0.0))
```

顾名思义，二值交叉熵即用于二分类的交叉熵损失函数，该损失一般配合sigmoid激活函数使用（第1006行）。

3. ROIAlign

ROIAlign的提出是为了解决Faster R-CNN中ROI Pooling的区域不匹配的问题，下面我们来举例说明什么是区域不匹配。ROI Pooling的区域不匹配问题是由于ROI Pooling过程中的取整操作产生的（图6），我们知道ROI Pooling是Faster R-CNN中必不可少的一步，因为其会产生长度固定的特征向量，有了长度固定的特征向量才能进行softmax计算分类损失。

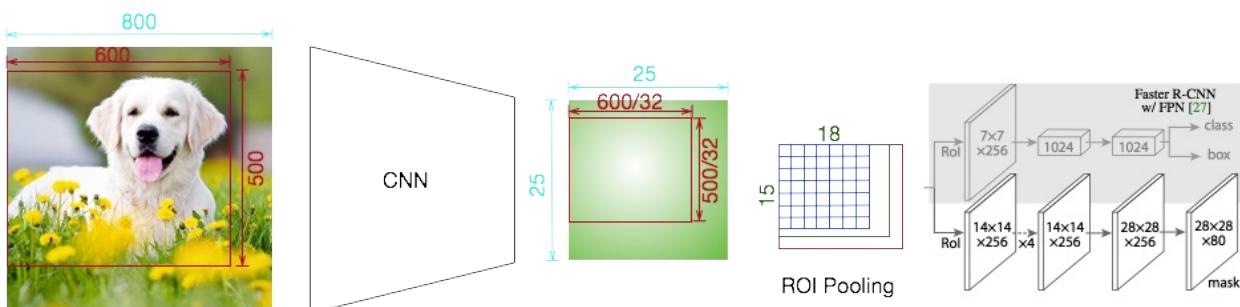
如下图，输入是一张 800×800 的图片，经过一个有5次降采样的卷机网络，得到大小为 25×25 的Feature Map。图中的ROI区域大小是 600×500 ，经过网络之后对应的区域为

$\frac{600}{32} \times \frac{500}{32} = 18.75 \times 15.625$ ，由于无法整除，ROI Pooling采用向下取整的方式，进而得到ROI区域的Feature Map的大小为 18×15 ，这就造成了第一次区域不匹配。

ROI Pooling的下一步是对Feature Map分bin，加入我们需要一个 7×7 的bin，每个bin的大小为 $\frac{18}{7} \times \frac{15}{7}$ ，由于不能整除，ROI同样采用了向下取整的方式，从而每个bin的大小为 2×2 ，即整个ROI区域的Feature Map的尺寸为 14×14 。第二次区域不匹配问题因此产生。

对比ROI Pooling之前的Feature Map，ROI Pooling分别在横向和纵向产生了4.75和1.625的误差，对于物体分类或者物体检测场景来说，这几个像素的位移或许对结果影响不大，但是语义分割任务通常要精确到每个像素点，因此ROI Pooling是不能应用到Mask R-CNN中的。

图6：ROI Pooling的区域不匹配问题

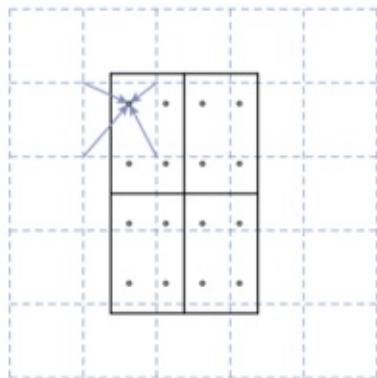


为了解决这个问题，作者提出了RoIAlign。RoIAlign并没有取整的过程，可以全程使用浮点数操作，步骤如下：

1. 计算ROI区域的边长，边长不取整；
2. 将ROI区域均匀分成 $k \times k$ 个bin，每个bin的大小不取整；
3. 每个bin的值为其最邻近的Feature Map的四个值通过双线性插值得到；
4. 使用Max Pooling或者Average Pooling得到长度固定的特征向量。

上面步骤如图7所示。

图7：RoIAlign可视化



RoIAlign操作通过 `tf.image.crop_and_resize` 一个函数便可以实现，在`./mrcnn/model.py`的第421-423行。由于Mask R-CNN使用了FPN作为骨干架构，所以使用了循环保存每次Pooling之后的Feature Map。

代码片段4：RoIAlign

```
tf.image.crop_and_resize(feature_maps[i], level_boxes, box_indices, self.pool_shape, method="bilinear")
```

总结

Mask R-CNN是一个很多state-of-the-art算法的合成体，并非常巧妙的设计了这些模块的合成接口：

1. 使用残差网络作为卷积结构；
2. 使用FPN作为骨干架构；
3. 使用Faster R-CNN的物体检测流程：RPN+Fast R-CNN；
4. 增加FCN用于语义分割。

Mask R-CNN设计的主要接口有：

1. 将FCN和Faster R-CNN合并，通过构建一个三任务的损失函数来优化模型；
2. 使用RoIAlign优化了RoI Pooling，解决了Faster R-CNN在语义分割中的区域不匹配问题。

附录A：双线性插值

双线性插值即在二维空间上按维度分别进行线性插值。

线性插值：已知在直线上两点 $(x_0, y_0), (x_1, y_1)$ ，则在 $[x_0, x_1]$ 区间内任意一点 $[x, y]$ 满足等式

$$y = y_0 + \frac{y_1 - y_0}{x_1 - x_0} (x - x_0)$$

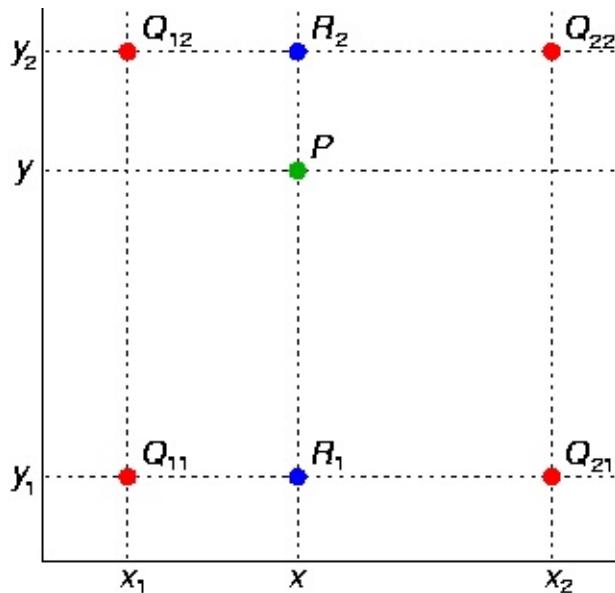
$$\frac{y - y_0}{x - x_0} = \frac{y_1 - y_0}{x_1 - x_0}$$

即已知 x 的情况下， y 的计算方式为：

$$y = \frac{x_1 - x}{x_1 - x_0} y_0 + \frac{x - x_0}{x_1 - x_0} y_1$$

双线性插值：双线性插值即在二维空间的每个维度分别进行线性插值，如图8

图8：双线性插值



已知二维空间中4点 $Q_{11} = (x_1, y_1)$ ， $Q_{12} = (x_1, y_2)$ ， $Q_{21} = (x_2, y_1)$ ， $Q_{22} = (x_2, y_2)$ ，我们要求的是空间中一点中 $P = (x, y)$ 的值 $f(P)$ 。

首先在 y 轴上进行线性插值据得到 R_1 和 R_2 ：

$$f(R_1) = f(x, y_1) = \frac{x_2 - x}{x_2 - x_1} f(Q_{11}) = \frac{x - x_1}{x_2 - x_1} f(Q_{21})$$

$$f(R_2) = f(x, y_2) = \frac{x_2 - x}{x_2 - x_1} f(Q_{12}) = \frac{x - x_1}{x_2 - x_1} f(Q_{22})$$

在根据 R_1 和 R_2 在 x 轴上进行线性插值

$$f(P) = \frac{y_2 - y}{y_2 - y_1} f(x, y_1) = \frac{y - y_1}{y_2 - y_1} f(x, y_2)$$

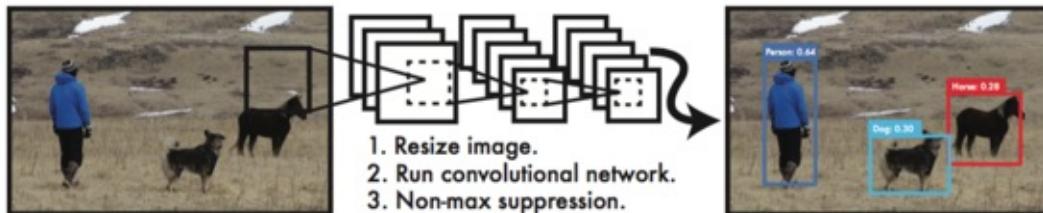
You Only Look Once: Unified, Real-Time Object Detection

前言

在R-CNN [63]系列的论文中，目标检测被分成了候选区域提取和候选区域分类及精校两个阶段。不同于这些方法，YOLO将整个目标检测任务整合到一个回归网络中。对比Fast R-CNN [61]提出的两步走的端到端方案，YOLO [56]的单阶段的使其是一个更彻底的端到端的算法（图1）。YOLO的检测过程分为三步：

1. 图像Resize到448*448；
2. 将图片输入卷积网络；
3. NMS得到最终候选框。

图1：YOLO算法框架



虽然在一些数据集上的表现不如Fast R-CNN及其后续算法，但是YOLO带来的最大提升便是检测速度的提升。在YOLO算法中，检测速度达到了45帧/秒，而一个更快速的Fast Yolo版本则达到了155帧/秒。另外在YOLO的背景检测错误率要低于Fast R-CNN。最后，YOLO算法具有更好的通用性，通过Pascal数据集训练得到的模型在艺术品问检测中得到了比Fast R-CNN更好的效果。

YOLO是可以在Fast R-CNN中的，结合YOLO和Fast R-CNN两个算法，得到的效果比单Fast R-CNN要更好。

YOLO源码是使用DarkNet框架实现的，由于本人对DarkNet并不熟悉，所以这里我使用YOLO的TensorFlow源码详细解析YOLO算法的每个技术细节和算发动机。

YOLO算法详解

YOLO检测速度远远超过R-CNN系列的重要原因是YOLO将整个物体检测统一成了一个回归问题。YOLO的输入是整张待检测图片，输出则是得到的检测结果，整个过程只经过一次卷积网络。Faster R-CNN [60]虽然使用全卷积的思想实现了候选区域的权值共享，但是每个候选区域的特征向量任然要单独的计算分类概率和bounding box。

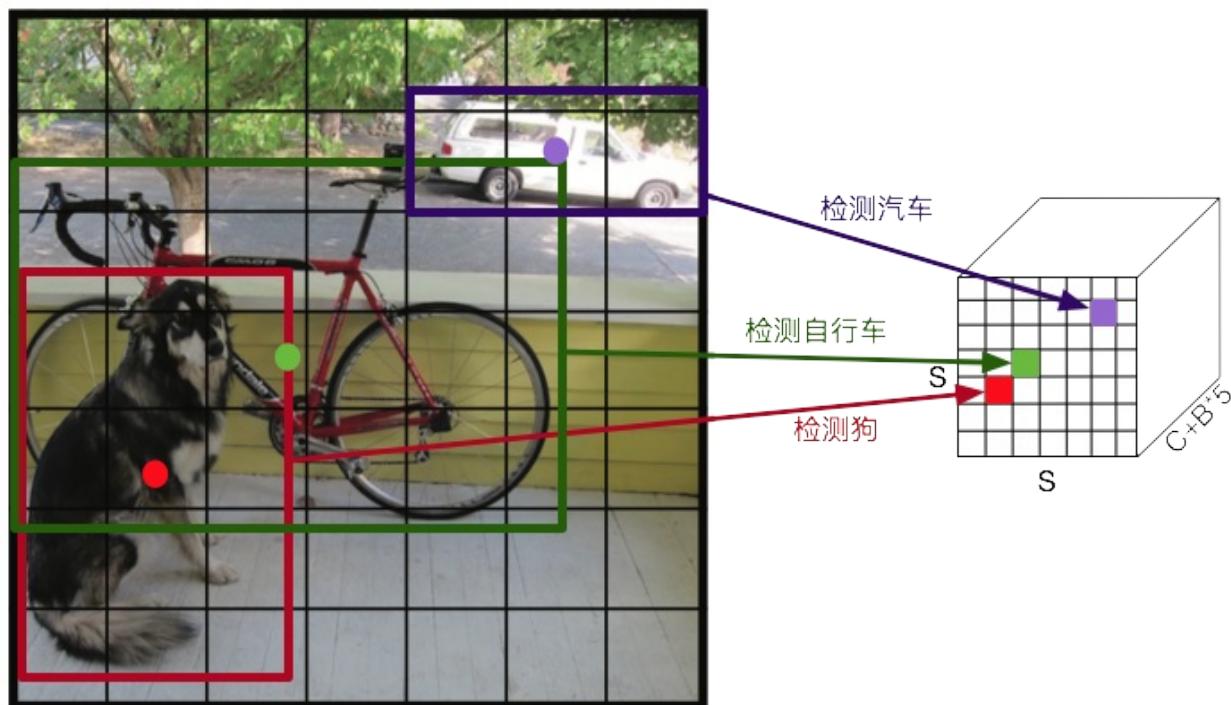
YOLO实现统一检测的方法是增加网络的输出节点数量，其实也算是空间换时间的一种策略。在Faster R-CNN的Fast R-CNN部分，网络有分类和回归两个任务，网络输出节点个数是 $C + 5$ ，其中 C 是数据集的类别个数。而YOLO的输出层 O 节点个数达到了 $S \times S \times (C + B \times 5)$ ，下面我们来讲解输出节点每个字符的含义。

1. YOLO输出层详解

1.1 $S \times S$ 窗格

YOLO将输入图像分成 $S \times S$ 的窗格（Grid），如果Ground Truth的中心落在某个单元（cell）内，则该单元负责该物体的检测，如图2所示。

图2： $S \times S$ 窗格



什么是某个单元负责落在该单元内的物体检测呢？举例说明一下，首先我们将输出层

$O_{S \times S \times (C+B \times 5)}$ 看做一个三维矩阵，如果物体的中心落在第 (i, j) 个单元内，那么网络只优化一个 $C + B \times 5$ 维的向量，即向量 $O[i, j, :]$ 。 S 是一个超参数，在源码中 $S = 7$ ，即配置文件 `./yolo/config.py` 的`CELL_SIZE`变量。

```
CELL_SIZE = 7
```

1.2 Bounding Box

B 是每个单元预测的bounding box的数量， B 的个数同样是一个超参数。

在 `./yolo/config.py` 文件中 $B = 2$ ，YOLO使用多个bounding box是为了每个cell计算top-B个可能的预测结果，这样做虽然牺牲了一些时间，但却提升了模型的检测精度。

```
BOXES_PER_CELL = 2
```

注意不管YOLO使用了多少个bounding box，每个单元的bounding box均有相同的优化目标值。在 `./yolo/yolo_net.py` 中，Ground Truth的label值被复制了 B 次。每个bounding box要预测5个值：bounding box (x, y, w, h) 以及置信度 P 。其中 (x, y) 是bounding box相对于每个cell中心的相对位置， (w, h) 是物体相对于整幅图的尺寸。

代码片段1：bounding box预处理

```
boxes = tf.tile(boxes, [1, 1, 1, self.boxes_per_cell, 1]) / self.image_size
classes = labels[:, :, :, 5:]
offset = tf.reshape(
    tf.constant(self.offset, dtype=tf.float32),
    [1, self.cell_size, self.cell_size, self.boxes_per_cell])
offset = tf.tile(offset, [self.batch_size, 1, 1, 1])
offset_tran = tf.transpose(offset, (0, 2, 1, 3))
...
boxes_tran = tf.stack(
    [boxes[:, :, 0] * self.cell_size - offset,
     boxes[:, :, 1] * self.cell_size - offset_tran,
     tf.sqrt(boxes[:, :, 2]),
     tf.sqrt(boxes[:, :, 3])], axis=-1)
```

labels需要往前追溯到Pascal voc文件的解析代码中，位于文件 `./utils/pascal_voc.py` 的139和145行

```
boxes = [(x2 + x1) / 2.0, (y2 + y1) / 2.0, x2 - x1, y2 - y1]
...
label[y_ind, x_ind, 1:5] = boxes
```

置信度 P 表示bounding box中物体为待检测物体的概率以及bounding box对该物体的覆盖程度的乘积。所以 $P = Pr(\text{Object}) \times IOU_{pred}^{truth}$ 。如果bounding box没有覆盖物体， $P = 0$ ，否则 $P = IOU_{pred}^{truth}$ 。

代码片段2：bounding box的标签值处理

```

predict_boxes = tf.reshape(
    predicts[:, self.boundary2:], 
    [self.batch_size, self.cell_size, self.cell_size, self.boxes_per_cell, 4])

response = tf.reshape(labels[...], [self.batch_size, self.cell_size, self.cell_size, self.cell_size, 1])
...
predict_boxes_tran = tf.stack(
    [(predict_boxes[..., 0] + offset) / self.cell_size,
     (predict_boxes[..., 1] + offset_tran) / self.cell_size,
     tf.square(predict_boxes[..., 2]),
     tf.square(predict_boxes[..., 3])], axis=-1)

iou_predict_truth = self.calc_iou(predict_boxes_tran, boxes)

# calculate I tensor [BATCH_SIZE, CELL_SIZE, CELL_SIZE, BOXES_PER_CELL]
object_mask = tf.reduce_max(iou_predict_truth, 3, keep_dims=True)
object_mask = tf.cast((iou_predict_truth >= object_mask), tf.float32) * response

# calculate no_I tensor [CELL_SIZE, CELL_SIZE, BOXES_PER_CELL]
noobject_mask = tf.ones_like(object_mask, dtype=tf.float32) - object_mask
...
coord_mask = tf.expand_dims(object_mask, 4)

```

同时，YOLO也预测检测物体为某一类C的条件概率： $Pr(Class_i|Object)$ 。对于每一个单元，YOLO值计算一个分类概率，而与B的值无关。在测试时，将条件概率 $Pr(Class_i|Object)$ 乘以P便得到了每个为每一类的概率：

$$Pr(Class_i|Object) \times Pr(Object) \times IOU_{pred}^{truth} = Pr(Class_i) \times IOU_{pred}^{truth}$$

该部分代码在 `./test.py` 文件中：

```

for i in range(self.boxes_per_cell):
    for j in range(self.num_class):
        probs[:, :, i, j] = np.multiply(class_probs[:, :, j], scales[:, :, i])

```

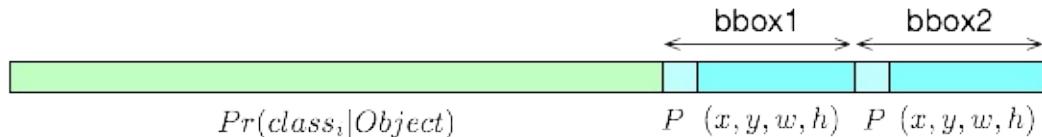
1.3 类别C

不同于Faster R-CNN添加了背景类，YOLO仅使用了数据集提供的物体类别，在Pascal VOC中，待检测物体有20类，具体类别内容列在了配置文件 `./yolo/config.py` 中

```
CLASSES = ['aeroplane', 'bicycle', 'bird', 'boat', 'bottle', 'bus',
           'car', 'cat', 'chair', 'cow', 'diningtable', 'dog', 'horse',
           'motorbike', 'person', 'pottedplant', 'sheep', 'sofa',
           'train', 'tvmonitor']
```

对于输出层的两个超参数， $S = 7$ ， $B = 2$ 。则输出层的结构如图3所示。

图3：YOLO的输出层



2. 输入层

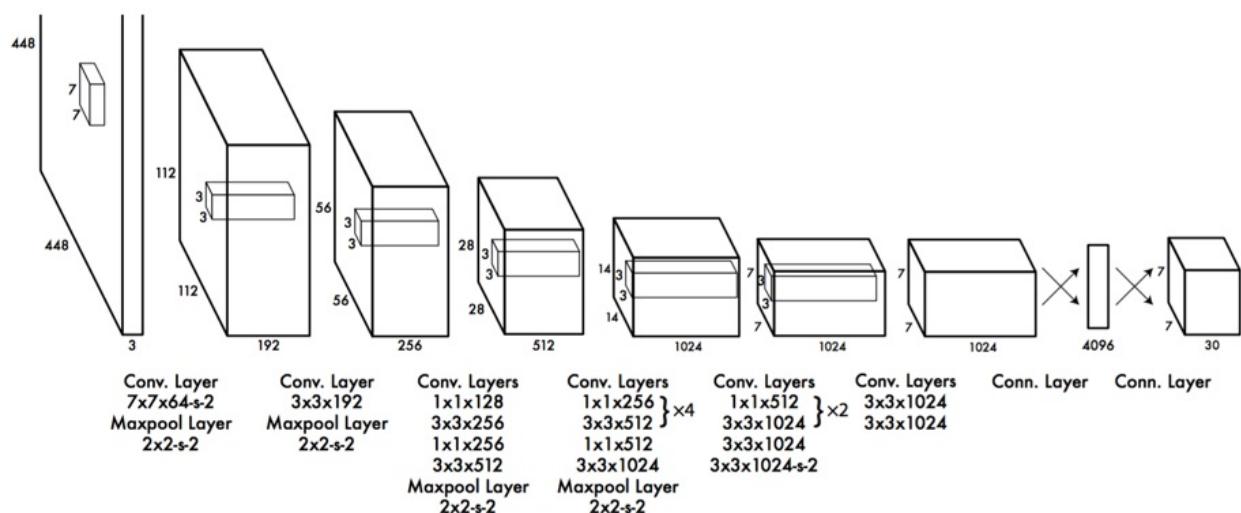
YOLO作为一个统计检测算法，整幅图是直接输入网络的。因为检测需要更细粒度的图像特征，YOLO将图像Resize到了448*448而不是物体分类中常用的224*224的尺寸。resize在 `./utils/pascal_voc.py` 中，需要注意的是YOLO并没有采用VGG中先将图像等比例缩放再裁剪的形式，而是直接将图片非等比例resize。所以YOLO的输出图片的尺寸并不是标准比例的。

```
image = cv2.resize(image, (self.image_size, self.image_size))
```

3. 骨干架构

YOLO使用了GoogLeNet作为骨干架构，但是使用了更少的参数，同时YOLO也不像GoogLeNet有3个输出层，图4。为了提高模型的精度，作者也使用了在ImageNet进行预训练的迁移学习策略。

图4：YOLO的骨干架构



研究发现，在AlexNet中提出的ReLU存在Dead ReLU的问题，所谓Dead ReLU是指由于ReLU的x负数部分的导数永远为0，会导致一部分神经元永远不会被激活，从而一些参数永远不会被更新。

为了解决这个问题，Andrew NG团队提出了leaky ReLU，即在负数部分给与一个很小的梯度，leaky ReLU拥有ReLU的所有优点，但同时不会有Dead ReLU的问题。YOLO中的leaky ReLU ($\phi(x)$) 表示为

$$\phi(x) = \begin{cases} x & \text{if } x > 0 \\ 0.1 \times x & \text{otherwise} \end{cases}$$

```
def leaky_relu(alpha):
    def op(inputs):
        return tf.nn.leaky_relu(inputs, alpha=alpha, name='leaky_relu')
    return op
```

然而现在的一些文章指出leaky ReLU并不是那么理想，现在尝试网络超参数时ReLU依旧是首选。

4. 损失函数

YOLO的输出层包含标签种类决定了YOLO的损失函数必须是一个多任务的损失函数。根据1.2节的介绍我们已知YOLO的输出层包含分类信息，置信度 P 和bounding box的坐标信息 (x, y, w, h) 。我们先给出YOLO的损失函数的表达式再逐步解析损失函数这样设计的动机。

$$\begin{aligned} & \lambda_{\text{coord}} \sum_{i=0}^{S^2} \sum_{j=0}^B \mathbb{1}_{ij}^{\text{obj}} \left[(x_i - \hat{x}_i)^2 + (y_i - \hat{y}_i)^2 \right] \\ & + \lambda_{\text{coord}} \sum_{i=0}^{S^2} \sum_{j=0}^B \mathbb{1}_{ij}^{\text{obj}} \left[\left(\sqrt{w_i} - \sqrt{\hat{w}_i} \right)^2 + \left(\sqrt{h_i} - \sqrt{\hat{h}_i} \right)^2 \right] \\ & + \sum_{i=0}^{S^2} \sum_{j=0}^B \mathbb{1}_{ij}^{\text{obj}} \left(C_i - \hat{C}_i \right)^2 \\ & + \lambda_{\text{noobj}} \sum_{i=0}^{S^2} \sum_{j=0}^B \mathbb{1}_{ij}^{\text{noobj}} \left(C_i - \hat{C}_i \right)^2 \\ & + \sum_{i=0}^{S^2} \mathbb{1}_i^{\text{obj}} \sum_{c \in \text{classes}} (p_i(c) - \hat{p}_i(c))^2 \end{aligned}$$

4.1 noobj

根据图2和图4所示，YOLO的 $S \times S$ 的窗格形式必然导致输出层含有大量的不包含物体的区域（也就是背景区域）。YOLO并不是直接将这一部分丢弃而是直接将其作为noobj一个分支进行优化。也是因为这个分支导致YOLO在检测背景时的错误率要比Fast R-CNN低近乎3倍。

4.2 λ_{coord} 和 λ_{noobj}

YOLO并没有使用深度学习常用的均方误差（MSE）而是使用和方误差（SSE）作为损失函数，作者的解释是SSE更好优化。但是SSE作为损失函数时会使模型更倾向于优化输出向量长度更长的任务（也就是分类任务）。为了提升bounding box边界预测的优先级，该任务被赋予了一个超参数 λ_{coord} ，在论文中 $\lambda_{coord} = 5$ 。

作者在观察数据集时发现Pascal VOC中包含样本的单元要远远少于包含背景区域的单元，为了解决前/背景样本的样本不平衡的问题，作者给非样本区域的分类任务一个更小的权值

λ_{noobj} ，在论文中 $\lambda_{noobj} = 0.5$ 。

需要注意的是TF的源码（`./yolo/config.py`）使用的并不是论文和DarkNet源码中给出的超参数。对于损失函数的四个任务，坐标预测，前景预测，背景预测和分类预测的权值使用的权值分别是1，1，2，5。该值并不是非常重要，通常需要根据模型在验证集上的表现调整。

```
OBJECT_SCALE = 1.0
NOOBJECT_SCALE = 1.0
CLASS_SCALE = 2.0
COORD_SCALE = 5.0
```

4.3 $I_{i,j}^{obj}$ ， I_i^{obj} 和 $I_{i,j}^{noobj}$

根据1.1节的定义，当bounding box $Box_{i,j}$ 负责检测某个物体时， $I_{i,j}^{obj} = 1$ ，否则 $I_{i,j}^{obj} = 0$ 。其中 i 用于遍历图像的单元， j 用于遍历每个cell的bounding box。而 $I_{i,j}^{noobj}$ 的定义则与 $I_{i,j}^{obj}$ 相反。在1.2节中我们介绍分类是以单元为单位的而与bounding box无关，所以 I_i^{obj} 表示物体出现在 $Cell_i$ 中。

$I_{i,j}^{obj}$ ， I_i^{obj} 和 $I_{i,j}^{noobj}$ 分别是代码片段2中的参数`object_mask`，`coord_mask`和`noobject_mask`。由于使用了`mask`，当网络遇到一个正样本时，只有一个单元的权值被调整，这也就是1.1节说的“该单元负责该物体的检测”。

4.4 $\sqrt{\cdot}$

最后还有一个小知识点，为了平衡短边和长边对损失函数的影响，YOLO使用了边长的平方根来减小长边的影响。

YOLO的损失函数见代码片段3

代码片段3：YOLO的损失函数

```

# class_loss
class_delta = response * (predict_classes - classes)
class_loss = tf.reduce_mean(
    tf.reduce_sum(tf.square(class_delta), axis=[1, 2, 3]), name='class_loss') * self.class_
scale

# object_loss
object_delta = object_mask * (predict_scales - iou_predict_truth)
object_loss = tf.reduce_mean(
    tf.reduce_sum(tf.square(object_delta), axis=[1, 2, 3]),
    name='object_loss') * self.object_scale

# noobject_loss
noobject_delta = noobject_mask * predict_scales
noobject_loss = tf.reduce_mean(
    tf.reduce_sum(tf.square(noobject_delta), axis=[1, 2, 3]),
    name='noobject_loss') * self.noobject_scale

# coord_loss
coord_mask = tf.expand_dims(object_mask, 4)
boxes_delta = coord_mask * (predict_boxes - boxes_tran)
coord_loss = tf.reduce_mean(
    tf.reduce_sum(tf.square(boxes_delta), axis=[1, 2, 3, 4]),
    name='coord_loss') * self.coord_scale

```

5. 后处理

测试样本时，有些物体会被多个单元检测到，NMS用于解决这个问题。

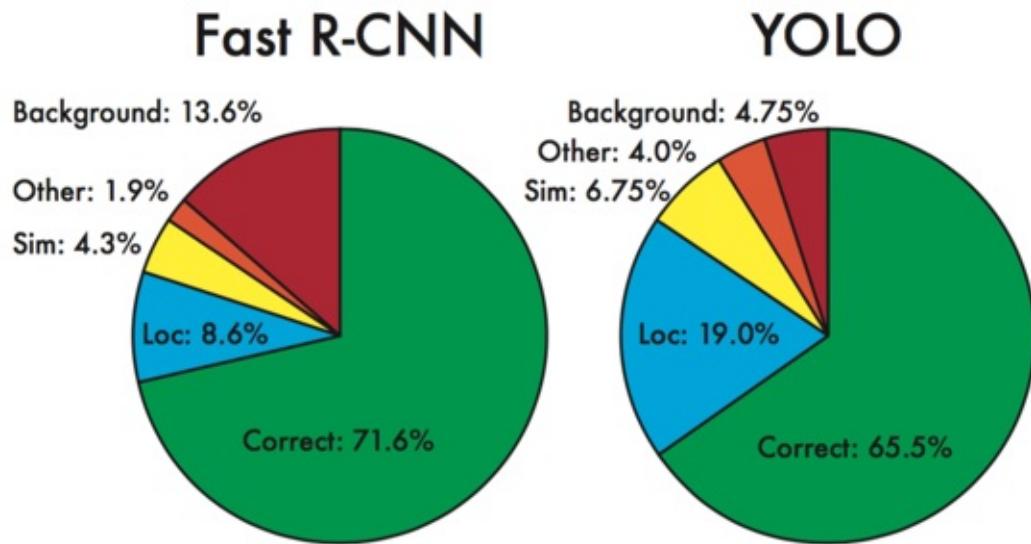
YOLO的优点

YOLO的快速我们已经一再重复，其性能的提升是因为YOLO统一检测框架的提出。同时YOLO在检测背景和通用性上表现也比Fast R-CNN要好。关于为什么YOLO比Fast R-CNN更擅长检测背景我们在4.1节进行了说明。从图5中我们可以看出YOLO的主要问题在于bounding box的精确检测。

- **Correct**: 正确分类且 $\text{IoU} > 0.5$ ；
- **Localization**：正确分类且 $0.1 < \text{IoU} < 0.5$
- **Similar**：类别近似且 $\text{IoU} > 0.1$

- Other : 分类错误且IoU>0.1
- Background : IoU<0.1的所有样本

图5 : Fast R-CNN vs YOLO



YOLO的论文中也指出YOLO的通用性更强，例如在人类画作的数据集上YOLO的表现要优于Fast R-CNN。但是为什么通用性更好至今我尚未想通，等待大神补充。

YOLO的缺点

YOLO的缺点也非常明显，首先其精确检测的能力比不上Fast R-CNN更不要提和其更早提出的Faster R-CNN了。

YOLO的另外一个重要问题是对于小物体的检测，其为了提升速度而粗粒度划分单元而且每个单元的bounding box的功能过度重合导致模型的拟合能力有限，尤其是其很难覆盖到的小物体。YOLO检测小尺寸问题效果不好的另外一个原因是因其只使用顶层的Feature Map，而顶层的Feature Map已经不会包含很多小尺寸的物体的特征了。

Faster R-CNN之后的算法均趋向于使用全卷积代替全连接层，但是YOLO依旧笨拙的使用了全连接不仅会使特征向量失去对于物体检测非常重要的位置信息，而且会产生大量的参数，影响算法的速度。

不过暂时不用着急，YOLO也在不断进化，YOLOv2，YOLOv3将会在速度和精度上进行优化，我们会在后续的文章中介绍。

SSD: Single Shot MultiBox Detector

前言

在YOLO [56] 的文章中我们介绍到YOLO存在三个缺陷：

1. 两个bounding box功能的重复降低了模型的精度；
2. 全连接层的使用不仅使特征向量失去了位置信息，还产生了大量的参数，影响了算法的速度；
3. 只使用顶层的特征向量使算法对于小尺寸物体的检测效果很差。

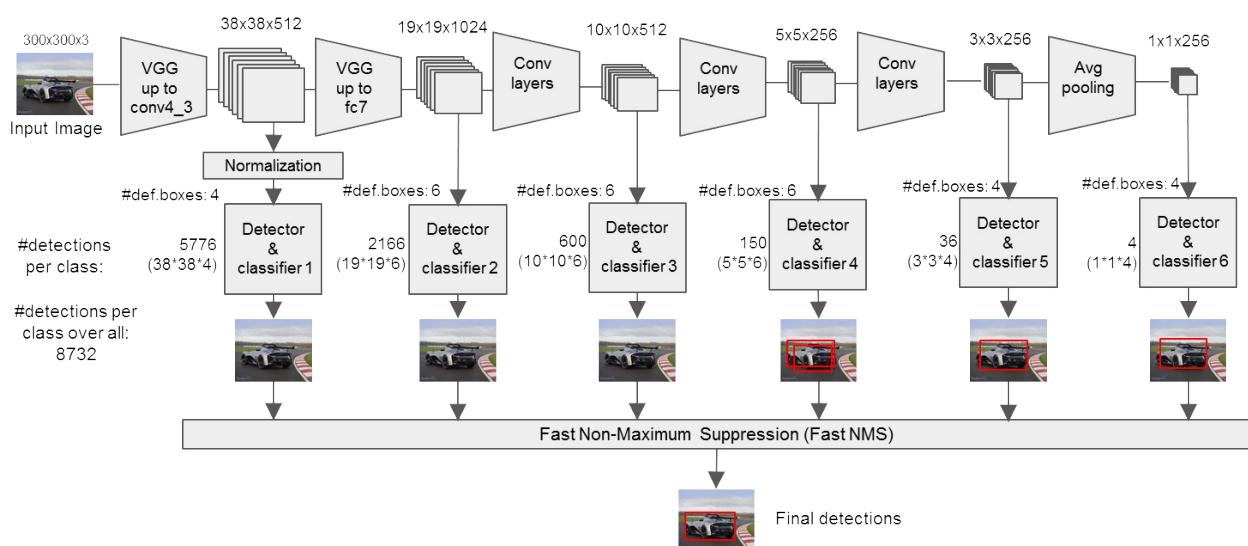
为了解决这些问题，SSD [54] 应运而生。SSD的全称是Single Shot MultiBox Detector，Single Shot表示SSD是像YOLO一样的单次检测算法，MultiBox指SSD每次可以检测多个物体，Detector表示SSD是用来进行物体检测的。

针对YOLO的三个问题，SSD做出的改进如下：

1. 使用了类似Faster R-CNN中RPN网络提出的锚点（Anchor）机制，增加了bounding box的多样性；
2. 使用全卷积的网络结构，提升了SSD的速度；
3. 使用网络中多个阶段的Feature Map，提升了特征多样性。

SSD的算法如图1。

图1：SSD算法流程



从某个角度讲，SSD和RPN的相似度也非常高，网络结构都是全卷积，都是采用了锚点进行采样，不同之处有下面两点：

1. RPN只使用卷积网络的顶层特征，不过在FPN和Mask R-CNN中已经对这点进行了改进；
2. RPN是一个二分类任务（前/背景），而SSD是一个包含了物体类别的多分类任务。

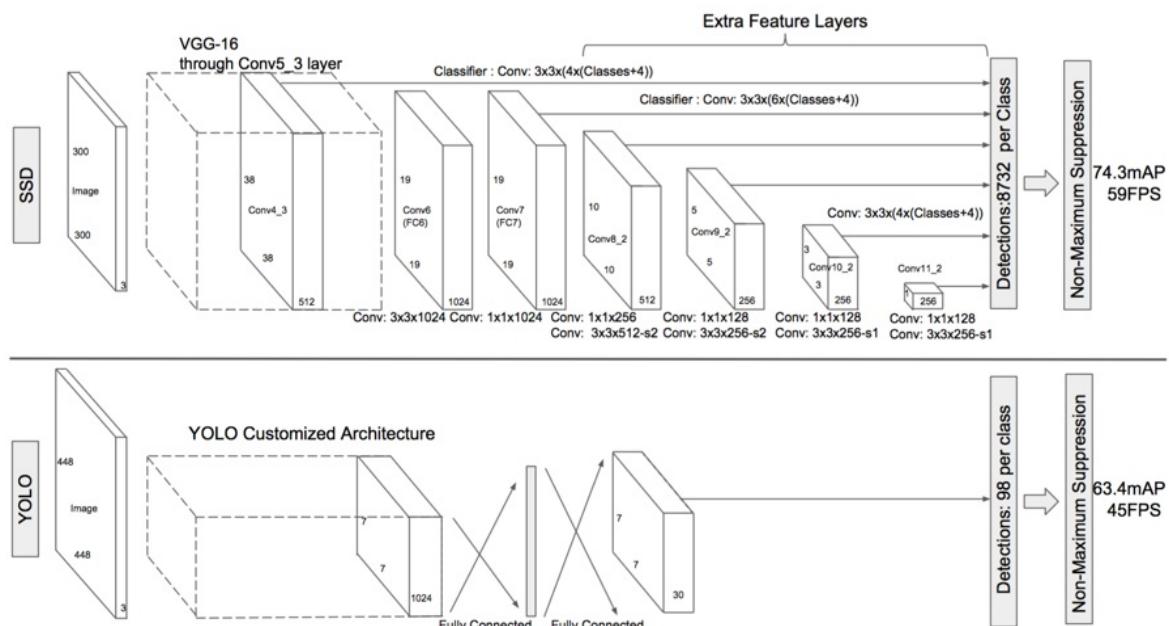
在论文中作者说SSD的精度超过了Faster R-CNN，速度超过了YOLO。下面我们将结合基于Keras的[源码](#)和论文对SSD进行详细剖析。

SSD详解

1. 算法流程

SSD的流程和YOLO是一样的，输入一张图片得到一系列候选区域，使用NMS得到最终的检测框。与YOLO不同的是，SSD使用了不同阶段的Feature Map用于检测，YOLO和SSD的对比如图2所示。

图1：SSD vs YOLO



在详解SSD之前，我先在代码片段1中列出SSD的超参数（`./models/keras_ssd300.py`），随后我们会在下面的章节中介绍这些超参数是如何使用的。

代码片段1：SSD的超参数

```

def ssd_300(image_size,
            n_classes,
            mode='training',
            l2_regularization=0.0005,
            min_scale=None,
            max_scale=None,
            scales=None,
            aspect_ratios_global=None,
            aspect_ratios_per_layer=[[1.0, 2.0, 0.5],
                                      [1.0, 2.0, 0.5, 3.0, 1.0/3.0],
                                      [1.0, 2.0, 0.5, 3.0, 1.0/3.0],
                                      [1.0, 2.0, 0.5, 3.0, 1.0/3.0],
                                      [1.0, 2.0, 0.5],
                                      [1.0, 2.0, 0.5]],
            two_boxes_for_ar1=True,
            steps=[8, 16, 32, 64, 100, 300],
            offsets=None,
            clip_boxes=False,
            variances=[0.1, 0.1, 0.2, 0.2],
            coords='centroids',
            normalize_coords=True,
            subtract_mean=[123, 117, 104],
            divide_by_stddev=None,
            swap_channels=[2, 1, 0],
            confidence_thresh=0.01,
            iou_threshold=0.45,
            top_k=200,
            nms_max_output_size=400,
            return_predictor_sizes=False)

```

1.1 SSD的骨干网络

首先我们先看一下SSD的骨干网络的源码（`./models/keras_ssd300.py`），再结合源码和图2我们来剖析SSD的算法细节。

代码片段2：SSD骨干网络源码。注意源码中的变量名称和图2不一样，我在代码中进行了更正。

```

conv1_1 = Conv2D(64, (3, 3), activation='relu', padding='same', kernel_initializer='he_normal', kernel_regularizer=l2(l2_reg), name='conv1_1')(x1)
conv1_2 = Conv2D(64, (3, 3), activation='relu', padding='same', kernel_initializer='he_normal', kernel_regularizer=l2(l2_reg), name='conv1_2')(conv1_1)
pool1 = MaxPooling2D(pool_size=(2, 2), strides=(2, 2), padding='same', name='pool1')(conv1_2)

conv2_1 = Conv2D(128, (3, 3), activation='relu', padding='same', kernel_initializer='he_normal', kernel_regularizer=l2(l2_reg), name='conv2_1')(pool1)
conv2_2 = Conv2D(128, (3, 3), activation='relu', padding='same', kernel_initializer='he_normal', kernel_regularizer=l2(l2_reg), name='conv2_2')(conv2_1)

```

```

pool2 = MaxPooling2D(pool_size=(2, 2), strides=(2, 2), padding='same', name='pool2')(conv2_2)

conv3_1 = Conv2D(256, (3, 3), activation='relu', padding='same', kernel_initializer='he_normal', kernel_regularizer=l2(l2_reg), name='conv3_1')(pool2)
conv3_2 = Conv2D(256, (3, 3), activation='relu', padding='same', kernel_initializer='he_normal', kernel_regularizer=l2(l2_reg), name='conv3_2')(conv3_1)
conv3_3 = Conv2D(256, (3, 3), activation='relu', padding='same', kernel_initializer='he_normal', kernel_regularizer=l2(l2_reg), name='conv3_3')(conv3_2)
pool3 = MaxPooling2D(pool_size=(2, 2), strides=(2, 2), padding='same', name='pool3')(conv3_3)

conv4_1 = Conv2D(512, (3, 3), activation='relu', padding='same', kernel_initializer='he_normal', kernel_regularizer=l2(l2_reg), name='conv4_1')(pool3)
conv4_2 = Conv2D(512, (3, 3), activation='relu', padding='same', kernel_initializer='he_normal', kernel_regularizer=l2(l2_reg), name='conv4_2')(conv4_1)
conv4_3 = Conv2D(512, (3, 3), activation='relu', padding='same', kernel_initializer='he_normal', kernel_regularizer=l2(l2_reg), name='conv4_3')(conv4_2)
pool4 = MaxPooling2D(pool_size=(2, 2), strides=(2, 2), padding='same', name='pool4')(conv4_3)

conv5_1 = Conv2D(512, (3, 3), activation='relu', padding='same', kernel_initializer='he_normal', kernel_regularizer=l2(l2_reg), name='conv5_1')(pool4)
conv5_2 = Conv2D(512, (3, 3), activation='relu', padding='same', kernel_initializer='he_normal', kernel_regularizer=l2(l2_reg), name='conv5_2')(conv5_1)
conv5_3 = Conv2D(512, (3, 3), activation='relu', padding='same', kernel_initializer='he_normal', kernel_regularizer=l2(l2_reg), name='conv5_3')(conv5_2)
pool5 = MaxPooling2D(pool_size=(3, 3), strides=(1, 1), padding='same', name='pool5')(conv5_3)

fc6 = Conv2D(1024, (3, 3), dilation_rate=(6, 6), activation='relu', padding='same', kernel_initializer='he_normal', kernel_regularizer=l2(l2_reg), name='fc6')(pool5)

fc7 = Conv2D(1024, (1, 1), activation='relu', padding='same', kernel_initializer='he_normal', kernel_regularizer=l2(l2_reg), name='fc7')(fc6)

conv8_1 = Conv2D(256, (1, 1), activation='relu', padding='same', kernel_initializer='he_normal', kernel_regularizer=l2(l2_reg), name='conv6_1')(fc7)
conv8_1 = ZeroPadding2D(padding=((1, 1), (1, 1)), name='conv6_padding')(conv8_1)
conv8_2 = Conv2D(512, (3, 3), strides=(2, 2), activation='relu', padding='valid', kernel_initializer='he_normal', kernel_regularizer=l2(l2_reg), name='conv6_2')(conv8_1)

conv9_1 = Conv2D(128, (1, 1), activation='relu', padding='same', kernel_initializer='he_normal', kernel_regularizer=l2(l2_reg), name='conv7_1')(conv8_2)
conv9_1 = ZeroPadding2D(padding=((1, 1), (1, 1)), name='conv7_padding')(conv9_1)
conv9_2 = Conv2D(256, (3, 3), strides=(2, 2), activation='relu', padding='valid', kernel_initializer='he_normal', kernel_regularizer=l2(l2_reg), name='conv7_2')(conv9_1)

conv10_1 = Conv2D(128, (1, 1), activation='relu', padding='same', kernel_initializer='he_normal', kernel_regularizer=l2(l2_reg), name='conv8_1')(conv9_2)
conv10_2 = Conv2D(256, (3, 3), strides=(1, 1), activation='relu', padding='valid', kernel_initializer='he_normal', kernel_regularizer=l2(l2_reg), name='conv8_2')(conv10_1)

```

```

conv11_1 = Conv2D(128, (1, 1), activation='relu', padding='same', kernel_initializer='he_normal', kernel_regularizer=l2(l2_reg), name='conv9_1')(conv10_2)
conv11_2 = Conv2D(256, (3, 3), strides=(1, 1), activation='relu', padding='valid', kernel_initializer='he_normal', kernel_regularizer=l2(l2_reg), name='conv9_2')(conv11_1)

```

从图1中我们可以看出，SSD输入图片的尺寸是 300×300 ，另外SSD也由一个输入图片尺寸是 512×512 的版本，这个版本的SSD虽然慢一些，但是检测精度达到了76.9%。

SSD采用的是VGG-16的作为骨干网络，VGG的详细内容参考文章[Very Deep Convolutional NetWorks for Large-Scale Image Recognition](#)。使用标准网络的目的是为了使用训练好的模型进行迁移学习，SSD使用的是在ILSVRC CLS-LOC数据集上得到的模型进行的初始化。目的是在更高的采样率上计算Feature Map。

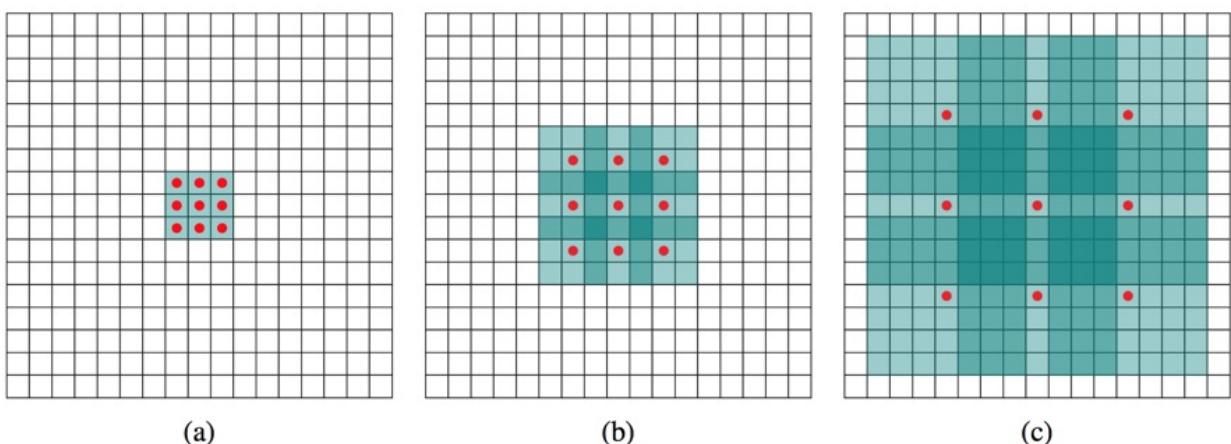
第一点不同的是在block5中，`max_pool2d`的步长`stride = 1`，此时图像将不会进行降采样，也就是说输入到block6的Feature Map的尺寸仍然是 38×38 。

SSD的 3×3 的conv6和 1×1 的conv7的卷积核是通过预训练模型的fc6和fc7采样得到，这种从全连接层中采样卷积核的方法参考的是[DeepLab-LargeFov \[39\]](#)的方法。具体细节在[DeepLab-LargeFov](#)的论文中进行分析。

在VGG的卷积部分之后，全连接被换成了卷积操作，在block6的卷积含有一个参数 `rate=6`。此时的卷积操作为空洞卷积（Dilation Convolution）[\[37\]](#)，在TensorFlow中使用 `tf.nn.atrous_conv2d()` 调用。

空洞卷积可以在不增加模型复杂度的同时扩大卷积操作的视野，通过在卷积核中插值0的形式完成的。如图3所示，(a)是膨胀率为1的卷积，也就是标准的卷积，其感受野的大小是 3×3 。(b)的膨胀率为2，卷积核变成了 7×7 的卷积核，其中只有9个红点处的值不为0，在不增加复杂度的同时感受野变成了 7×7 。(c)的膨胀率是4，感受野的大小变成了 15×15 。在设置感受野的膨胀率时要谨慎设计，否则如果卷积核大于Feature Map的尺寸之后程序会报错。

图3：空洞卷积示例图



fc7之后输出的Feature Map的大小是 19×19 ，经过block8的一次padding和一次valid卷积之后（即相当于一次same卷积），再经过一次步长为2的降采样，输入到block 9的Feature Map的尺寸是 10×10 。block 9的操作和block 8相同，即输入到block 8的Feature Map的尺寸是 5×5 。block 10和block 11使用的是valid卷积，所以图像的尺寸分别是3和1。这样我们便得到了图2中Feature Map尺寸的变化过程。

1.2 多尺度预测

在卷积网络中，不同深度的Feature Map趋向于响应不同程度的特征，SDD使用了骨干网络中的多个Feature Map用于预测检测框。通过图1和图2我们可以发现，SSD使用的是conv4_3, fc7, conv8_2, conv9_2, conv10_2, conv11_2分别用于检测尺寸从小到大的物体，如代码片段3（`./models/keras_ssd300.py`）。

代码片段3：SSD使用全卷积预测检测框

```

# Feed conv4_3 into the L2 normalization layer
conv4_3_norm = L2Normalization(gamma_init=20, name='conv4_3_norm')(conv4_3)

### Build the convolutional predictor layers on top of the base network

# We predict `n_classes` confidence values for each box, hence the confidence predictors have depth `n_boxes * n_classes`
# Output shape of the confidence layers: `(batch, height, width, n_boxes * n_classes)`
conv4_3_norm mbox conf = Conv2D(n_boxes[0] * n_classes, (3, 3), padding='same', kernel_initializer='he_normal', kernel_regularizer=l2(l2_reg), name='conv4_3_norm mbox conf')(conv4_3_norm)
fc7_mbox_conf = Conv2D(n_boxes[1] * n_classes, (3, 3), padding='same', kernel_initializer='he_normal', kernel_regularizer=l2(l2_reg), name='fc7_mbox_conf')(fc7)
conv8_2_mbox_conf = Conv2D(n_boxes[2] * n_classes, (3, 3), padding='same', kernel_initializer='he_normal', kernel_regularizer=l2(l2_reg), name='conv8_2_mbox_conf')(conv8_2)
conv9_2_mbox_conf = Conv2D(n_boxes[3] * n_classes, (3, 3), padding='same', kernel_initializer='he_normal', kernel_regularizer=l2(l2_reg), name='conv9_2_mbox_conf')(conv9_2)
conv10_2_mbox_conf = Conv2D(n_boxes[4] * n_classes, (3, 3), padding='same', kernel_initializer='he_normal', kernel_regularizer=l2(l2_reg), name='conv10_2_mbox_conf')(conv10_2)
conv11_2_mbox_conf = Conv2D(n_boxes[5] * n_classes, (3, 3), padding='same', kernel_initializer='he_normal', kernel_regularizer=l2(l2_reg), name='conv11_2_mbox_conf')(conv11_2)

# We predict 4 box coordinates for each box, hence the localization predictors have depth `n_boxes * 4`
# Output shape of the localization layers: `(batch, height, width, n_boxes * 4)`
conv4_3_norm mbox loc = Conv2D(n_boxes[0] * 4, (3, 3), padding='same', kernel_initializer='he_normal', kernel_regularizer=l2(l2_reg), name='conv4_3_norm mbox loc')(conv4_3_norm)
fc7_mbox_loc = Conv2D(n_boxes[1] * 4, (3, 3), padding='same', kernel_initializer='he_normal', kernel_regularizer=l2(l2_reg), name='fc7_mbox_loc')(fc7)
conv8_2_mbox_loc = Conv2D(n_boxes[2] * 4, (3, 3), padding='same', kernel_initializer='he_normal', kernel_regularizer=l2(l2_reg), name='conv8_2_mbox_loc')(conv8_2)
conv9_2_mbox_loc = Conv2D(n_boxes[3] * 4, (3, 3), padding='same', kernel_initializer='he_normal', kernel_regularizer=l2(l2_reg), name='conv9_2_mbox_loc')(conv9_2)
conv10_2_mbox_loc = Conv2D(n_boxes[4] * 4, (3, 3), padding='same', kernel_initializer='he_normal', kernel_regularizer=l2(l2_reg), name='conv10_2_mbox_loc')(conv10_2)
conv11_2_mbox_loc = Conv2D(n_boxes[5] * 4, (3, 3), padding='same', kernel_initializer='he_normal', kernel_regularizer=l2(l2_reg), name='conv11_2_mbox_loc')(conv11_2)

```

其中第二行的L2Normalization使用的是ParseNet [40] 中提出的全局归一化。即对像素点在通道维度上进行归一化，其中gamma是一个可训练的放缩变量。

SSD对于第*i*个Feature Map的每个像素点都会产生n_boxes[i]个锚点进行分类和位置精校，其中n_boxes的值为[4,6,6,6,4,4]，我们在1.3节会介绍n_boxes值的计算方法。SSD相当于预测M个bounding box，其中：

$$M = 38 \times 38 \times 4 + 19 \times 19 \times 6 + 10 \times 10 \times 6 + 5 \times 5 \times 6 + 3 \times 3 \times 4 + 1 \times 1 \times 4 = 8732$$

上式便是图2中最右侧8732的计算方式。也就是对于一张300*300的输入图片，SSD要预测8732个检测框，所以SSD本质上可以看做是密集采样。SSD的分类有 $C + 1$ 个值包括C类前景和1类背景，回归包括物体位置的四要素(y,x,h,w)。对于20类的Pascal VOC来说SSD是一个含有 $8732 \times (21 + 4)$ 的多任务模型。

通过代码片段3，我们可以看出SSD并没有使用全连接产生预测结果，而是使用的3*3的卷积操作分别产生了分类和回归的预测结果。对于一个分类任务来说，Feature Map的数量是 $(C+1)*n_boxes[i]$ ，而回归任务的Feature Map的数量是 $4*n_boxes[i]$ 。

1.3 SSD中的锚点

在1.2节中，我们介绍了SSD的 $n_boxes=[4, 6, 6, 6, 4, 4]$ ，下面我们就来详细解析SSD锚点是什么样子的。

SSD使用多尺度的Feature Map的原因是使用不同层次的Feature Map检测不同尺寸的物体，所以conv4_3, fc7, conv8_2, conv9_2, conv10_2, conv11_2的锚点的尺寸也是从小到大。论文中给出的值是从0.2到0.9间一个线性变化的值：

$$s_k = s_{min} + \frac{s_{max} - s_{min}}{m - 1}(k - 1), k \in [1, m]$$

s_{min} 和 s_{max} 是两个超参数，需要根据不同的数据集自行调整。论文中给出的例子是

$s_{min} = 0.2$ ， $s_{max} = 0.9$ ， $m = 6$ 。 s_k 表示的是锚点大小相对于Feature Map的比例，通过上式得出的值依次是 $[0.2, 0.34, 0.48, 0.62, 0.76, 0.9]$ 。

对于6组Feature Map，SSD分别产生 $[4, 6, 6, 6, 4, 4]$ 个不同比例的锚点。锚点的比例是超参

数 `aspect_ratios_per_layer` 中给出的值加上一组比例为 $s'_k = \sqrt{\frac{s_k}{s_{k+1}}}$ 的框，其中

$s_{k+1} = s_k + (s_k - s_{k-1})$ 。根据 s_k 和长宽比 a_r 我们便可以得到不同样式的锚点，其中锚点的宽

$$w_k^a = s_k \sqrt{\frac{1}{a_r}}, \text{ 高 } h_k^a = s_k / \sqrt{\frac{1}{a_r}}。a_r \in \{1, 2, 3, \frac{1}{2}, \frac{1}{3}\}。$$

a_r 的取值也是一个超参数，在源码中，定义在 `aspect_ratios_per_layer` 中。根据 `aspect_ratios_per_layer` 的变量个数，我们便可以得到 n_boxes 的值。

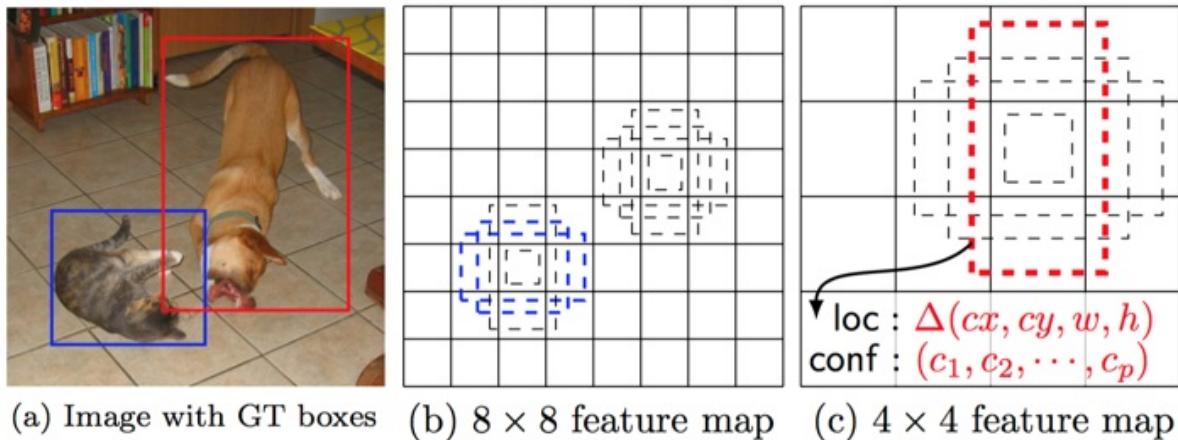
举个例子，在conv4_3中，要产生 $38 \times 38 \times 4$ 个锚点，其中有三个锚点的尺度分别是 $(1, 2.0, 0.5)$ ，再加上一组 $1:1$ 的尺度为 $s'_k = \sqrt{0.2 \times 0.34} = 0.2608$ 的锚点，得到四组锚点分别是 $[(0.2, 0.2), (0.2608, 0.2608), (0.2828, 0.1414), (0.1414, 0.2828)]$ 。等比例换算到原图中得到的锚点的大小（取整）为 $[(60, 60), (78, 78), (85, 42), (42, 85)]$ 。

通过上面的介绍，我们得到了锚点四要素中的 w 和 h ，锚点的 x, y 通过下式得到

$$(x, y) = \left(\frac{i + 0.5}{|f_k|}, \frac{j + 0.5}{|f_k|} \right), i, j \in [0, |f_k|]$$

i, j 即Feature Map像素点的坐标， f_k 是Feature Map的尺寸。图4便是在 8×8 和 4×4 的Feature Map上得到不同尺度的锚点的示例。

图4：锚点示例，改图也展示了锚点对Ground Truth的响应。



锚点如何设计是一种见仁见智的方案，例如源码中锚点的尺度便和论文不同，在源码中，尺度定义在jupyter notebook文件 `./ssd300_training.ipynb` 中。关于具体如何定义这些锚点其实不必太过在意，这些锚点的作用是为检测框提供一个先验假设，网络最后输出的候选框还是要经过Ground Truth纠正的。

```
scales_pascal = [0.1, 0.2, 0.37, 0.54, 0.71, 0.88, 1.05] # The anchor box scaling factors used in the original SSD300 for the Pascal VOC datasets
scales_coco = [0.07, 0.15, 0.33, 0.51, 0.69, 0.87, 1.05] # The anchor box scaling factors used in the original SSD300 for the MS COCO datasets
```

除了锚点的尺度以外，源码中锚点的中心点的实现也和论文不同。源码使用预先计算好的步长加上位移进行预测的，即超参数中的变量 `steps=[8, 16, 32, 64, 100, 300]`。`conv4_3`经过了3次降采样，即Feature Map的一步相当于原图的8步。但是对于这种方案存在一个问题，即75降采样到38时是不能整除的，也就是最后一列并没有参加降采样，这样步长非精确的计算经过多次累积会被放大到很大。例如经过源码中步长为64的`conv9_2`层的最后一行和最后一列的锚点的中心点将会取到图像之外，有兴趣的读者可以打印一下。

源码中，锚点是在 `keras_layers/keras_layer_AnchorBoxes` 中实现的，通过`AnchorBoxes`函数调用。网络中的6个Feature Map会产生6组共8732个先验box，如代码片段4所示。

代码片段4：计算先验box

```

# Output shape of anchors: `(batch, height, width, n_boxes, 8)`
conv4_3_norm_mbox_priorbox = AnchorBoxes(img_height, img_width, this_scale=scales[0],
                                         next_scale=scales[1], aspect_ratios=aspect_ratios[0],
                                         two_boxes_for_ar1=two_boxes_for_ar1, this_steps=steps[0],
                                         this_offsets=offsets[0], clip_boxes=clip_boxes,
                                         variances=variances, coords=coords, normalize_coords=normalize_coords,
                                         name='conv4_3_norm_mbox_priorbox')(conv4_3_norm_mbox_loc)
fc7_mbox_priorbox = AnchorBoxes(img_height, img_width, this_scale=scales[1], next_scale=scales[2],
                                 aspect_ratios=aspect_ratios[1],
                                 two_boxes_for_ar1=two_boxes_for_ar1, this_steps=steps[1],
                                 this_offsets=offsets[1], clip_boxes=clip_boxes,
                                 variances=variances, coords=coords, normalize_coords=normalize_coords,
                                 name='fc7_mbox_priorbox')(fc7_mbox_loc)
conv6_2_mbox_priorbox = AnchorBoxes(img_height, img_width, this_scale=scales[2], next_scale=scales[3],
                                    aspect_ratios=aspect_ratios[2],
                                    two_boxes_for_ar1=two_boxes_for_ar1, this_steps=steps[2],
                                    this_offsets=offsets[2], clip_boxes=clip_boxes,
                                    variances=variances, coords=coords, normalize_coords=normalize_coords,
                                    name='conv6_2_mbox_priorbox')(conv6_2_mbox_loc)
conv7_2_mbox_priorbox = AnchorBoxes(img_height, img_width, this_scale=scales[3], next_scale=scales[4],
                                    aspect_ratios=aspect_ratios[3],
                                    two_boxes_for_ar1=two_boxes_for_ar1, this_steps=steps[3],
                                    this_offsets=offsets[3], clip_boxes=clip_boxes,
                                    variances=variances, coords=coords, normalize_coords=normalize_coords,
                                    name='conv7_2_mbox_priorbox')(conv7_2_mbox_loc)
conv8_2_mbox_priorbox = AnchorBoxes(img_height, img_width, this_scale=scales[4], next_scale=scales[5],
                                    aspect_ratios=aspect_ratios[4],
                                    two_boxes_for_ar1=two_boxes_for_ar1, this_steps=steps[4],
                                    this_offsets=offsets[4], clip_boxes=clip_boxes,
                                    variances=variances, coords=coords, normalize_coords=normalize_coords,
                                    name='conv8_2_mbox_priorbox')(conv8_2_mbox_loc)
conv9_2_mbox_priorbox = AnchorBoxes(img_height, img_width, this_scale=scales[5], next_scale=scales[6],
                                    aspect_ratios=aspect_ratios[5],
                                    two_boxes_for_ar1=two_boxes_for_ar1, this_steps=steps[5],
                                    this_offsets=offsets[5], clip_boxes=clip_boxes,
                                    variances=variances, coords=coords, normalize_coords=normalize_coords,
                                    name='conv9_2_mbox_priorbox')(conv9_2_mbox_loc)

```

1.4 SSD的匹配准则

从Feature Map得到锚点之后，我们要确定Ground Truth和哪个锚点匹配，与之匹配的锚点将负责该Ground Truth的预测。在YOLO中，Ground Truth的中心点落在哪个单元内，则该单元的bounding box负责预测其准确的边界。SSD的锚点匹配采用了‘bipartite’和‘multi’两种策略，匹配源码位于 `./ssd_encoder_decoder/` 目录下面。

在bipartite模式中，每个Ground Truth选择与其IoU（论文用的是Jaccard Overlap）最大的锚

点进行匹配。如果一个锚点被多个Ground Truth匹配，那么该锚点只匹配与其IoU最大的Ground Truth，其它Ground Truth从剩下的锚点中选择IoU最大的那个进行匹配。`bipartite`可以保证每个Ground Truth都会有唯一的一个锚点进行匹配。`bipartite`的源码见代码片段5。

代码片段5：`bipartite`匹配

```
def match_bipartite_greedy(weight_matrix):
    """
    Parameters:
        weight_matrix (array): A 2D Numpy array that represents the weight matrix
            for the matching process. If `(m,n)` is the shape of the weight matrix,
            it must be `m <= n`. The weights can be integers or floating point
            numbers. The matching process will maximize, i.e. larger weights are
            preferred over smaller weights.

    Returns:
        A 1D Numpy array of length `weight_matrix.shape[0]` that represents
        the matched index along the second axis of `weight_matrix` for each index
        along the first axis.
    """
    weight_matrix = np.copy(weight_matrix)
    num_ground_truth_boxes = weight_matrix.shape[0]
    all_gt_indices = list(range(num_ground_truth_boxes))
    matches = np.zeros(num_ground_truth_boxes, dtype=np.int)
    for _ in range(num_ground_truth_boxes):
        # Find the maximal anchor-ground truth pair in two steps: First, reduce
        # over the anchor boxes and then reduce over the ground truth boxes.
        anchor_indices = np.argmax(weight_matrix, axis=1) # Reduce along the anchor bo
        x axis.
        overlaps = weight_matrix[all_gt_indices, anchor_indices]
        ground_truth_index = np.argmax(overlaps) # Reduce along the ground truth box a
        xis.
        anchor_index = anchor_indices[ground_truth_index]
        matches[ground_truth_index] = anchor_index # Set the match.

        # Set the row of the matched ground truth box and the column of the matched
        # anchor box to all zeros. This ensures that those boxes will not be matched a
        gain,
        # because they will never be the best matches for any other boxes.
        weight_matrix[ground_truth_index] = 0
        weight_matrix[:, anchor_index] = 0

    return matches
```

在`bipartite`策略中被匹配的锚点数量是非常少的，这就造成了训练时的正负样本的不平衡。所以需要`multi`策略进行纠正，源码中也是使用的`multi`策略。`multi`在`bipartite`策略的基础上增加了所有与Ground Truth的IoU大于阈值 θ （源码中 $\theta = 0.5$ ）的锚点作为匹配锚点。SSD中一个Ground Truth是可以有多个锚点与其匹配的，但是反过来是不行的，一个锚点只能与和它IoU最大的Ground Truth进行匹配。`multi`策略的源码见代码片段6

代码片段6：multi匹配

```

def match_multi(weight_matrix, threshold):
    """
    Returns:
        Two 1D Numpy arrays of equal length that represent the matched indices. The first
        array contains the indices along the first axis of `weight_matrix`, the second
        array contains the indices along the second axis.
    """

    num_anchor_boxes = weight_matrix.shape[1]
    all_anchor_indices = list(range(num_anchor_boxes))
    # Find the best ground truth match for every anchor box.
    ground_truth_indices = np.argmax(weight_matrix, axis=0)
    overlaps = weight_matrix[ground_truth_indices, all_anchor_indices]

    # Filter out the matches with a weight below the threshold.
    anchor_indices_thresh_met = np.nonzero(overlaps >= threshold)[0]
    gt_indices_thresh_met = ground_truth_indices[anchor_indices_thresh_met]

    return gt_indices_thresh_met, anchor_indices_thresh_met

```

尽管通过multi匹配策略增加了正样本的数量，但是在8732个锚点中，正负样本的比例还是非常不均衡的。所以SSD使用了难分样本挖掘（Hard Negative Mining）的策略对负样本进行采样。即对负样本的置信度进行排序，在保证正负样本1:3的前提下抽取top-k个负样本。

1.5 SSD的损失函数

由于SSD也是一个由分类任务和检测任务多任务模型，所以SSD的损失函数将由置信度误差

L_{conf} 和位置误差 L_{loc} 组成：

$$L(x, c, l, g) = \frac{1}{N} (L_{conf}(x, c) + \alpha L_{loc}(x, l, g))$$

其中 N 是正锚点的数量， α 是两个任务的侧重比重，经过交叉验证之后 α 被设置成了1。

$x_{i,j}^p = \{0, 1\} \in x$ 用于指示该锚点是否和Ground Truth进行了匹配。

对于分类任务，SSD使用的是softmax多类别的损失函数，上式中的 c 表示分类置信度：

$$L_{conf}(x, c) = - \sum_{i \in Pos}^N x_{i,j}^p \log(\hat{c}_i^p) - \sum_{i \in Neg} \log(\hat{c}_i^0), \hat{c}_i^p = \frac{\exp(c_i^p)}{\sum_p \exp(c_i^p)}$$

对于回归任务，SSD预测的是正锚点和Ground Truth的相对位移，损失函数使用的是Smooth L1损失函数。 l 表示预测的锚点和Ground Truth的相对位移，而 g 表示实际的相对位移。其中 l 和 g 包含物体位置的四要素 $(\hat{g}_j^{cx}, \hat{g}_j^{cy}, \hat{g}_j^w, \hat{g}_j^h)$ 。

$$\hat{g}_j^{cx} = (g_j^{cx} - d_i^{cx})/d_i^w$$

$$\hat{g}_j^{cy} = (g_j^{cy} - d_i^{cy})/d_i^h$$

$$\hat{g}_j^w = \log\left(\frac{g_j^w}{d_i^w}\right)$$

$$\hat{g}_j^h = \log\left(\frac{g_j^h}{d_i^h}\right)$$

损失函数表示为实际偏移和预测偏移的Smooth L1损失：

$$L_{loc}(x, l, g) = - \sum_{i \in Pos}^N \sum_{m \in cx, cy, w, h} x_{i,j}^k \text{smooth}_{L1}(l_i^m - \hat{g}_j^m)$$

与Faster R-CNN的 (x, y) 表示左上角不同，SDD的 (cx, cy) 表示的是锚点的中心点。

1.6 SSD的检测过程

1. 根据预测类别过滤掉背景类别的候选框；
2. 根据置信度过滤掉置信度低于阈值的候选框；
3. 置信度降序排列，保留top-k的候选框；
4. 解码相对位移，得出预测框四要素；
5. 使用NMS得到最终的候选区域。

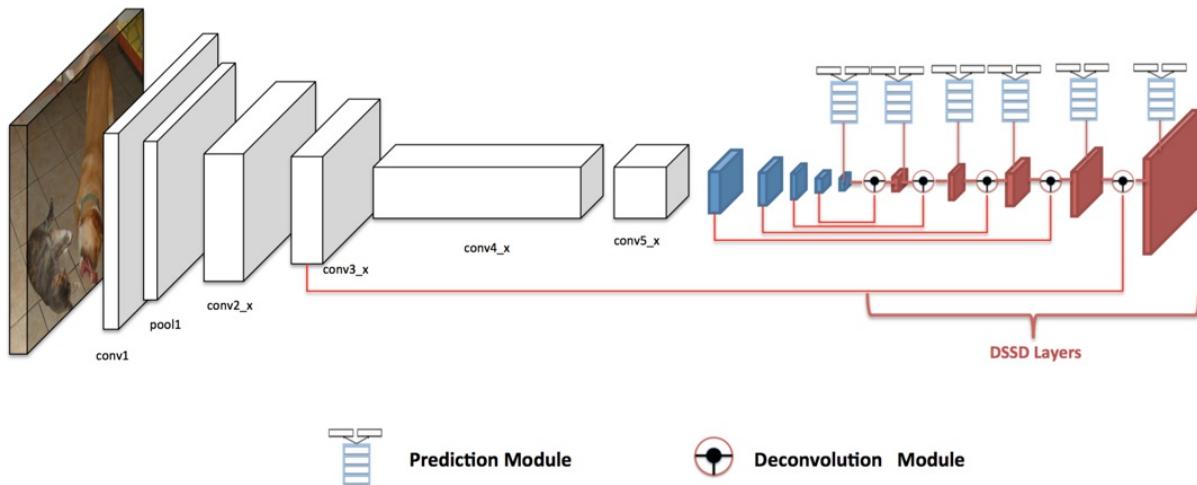
2. DSSD

SSD一个非常有意思的变种是使用反卷积增加了上下文信息的DSSD [53]，或者说用反卷积代替了基于双线性插值的上采样过程。下面我们来讲解DSSD是怎么进一步优化SSD的。

2.1 DSSD的骨干网络

在骨干网络方面，DSSD使用了层数更深的Residual Net-101，检测模块的网络是从conv5_x之后开始的，用于进行检测的则包括conv3_x，conv5_x和添加的检测模块，如图5。

图5：DSSD的骨干网络



DSSD并没有把反卷积部分构造的非常深，的原因有二：

1. 过多的反卷积会影响检测的速度，这与SSD的初衷不符；
2. 模型的训练依赖于迁移学习的初始化，而反卷积部分是没有模型可工迁移的。随机初始化部分如果过深的话会降低模型的收敛速度。

单纯的网络替换并不能带来检测效果的提升，DSSD的最大特点是图5右侧红色的反卷积部分。

2.2 反卷积

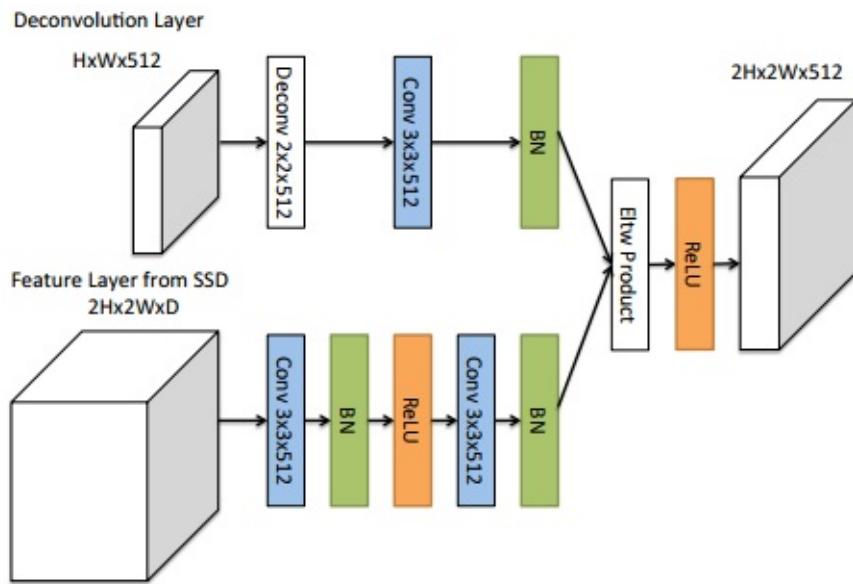
反卷积 [38]，又被叫做逆卷积，是在语义分割中应用中最常见的算法之一。下面通过一个例子来说明反卷积的工作原理：对于一个 4×4 的输入 x ，经过 3×3 卷积核的有效卷积，得到一个 2×2 的特征向量 y ，设卷积运算为 $y = Cx$ 。 C 的本质上是一个稀疏矩阵(很多开源框架卷积操作的实现方式)：

$$\begin{pmatrix} w_{0,0} & w_{0,1} & w_{0,2} & 0 & w_{1,0} & w_{1,1} & w_{1,2} & 0 & w_{2,0} & w_{2,1} & w_{2,2} & 0 & 0 & 0 & 0 \\ 0 & w_{0,0} & w_{0,1} & w_{0,2} & 0 & w_{1,0} & w_{1,1} & w_{1,2} & 0 & w_{2,0} & w_{2,1} & w_{2,2} & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & w_{0,0} & w_{0,1} & w_{0,2} & 0 & w_{1,0} & w_{1,1} & w_{1,2} & 0 & w_{2,0} & w_{2,1} & w_{2,2} & 0 \\ 0 & 0 & 0 & 0 & 0 & w_{0,0} & w_{0,1} & w_{0,2} & 0 & w_{1,0} & w_{1,1} & w_{1,2} & 0 & w_{2,0} & w_{2,1} & w_{2,2} \end{pmatrix}$$

反卷积相当于卷积网络的正向和反向的传播中做相反的运算，即正向的时候左乘 C^T ，反向的时候左乘 $(C^T)^T = C$ 的运算，所以有些人更喜欢把反卷积叫做转置卷积。

图5中的Deconvolution Module（反卷积模块）展开如图6所示。

图6：DSSD的反卷积模块

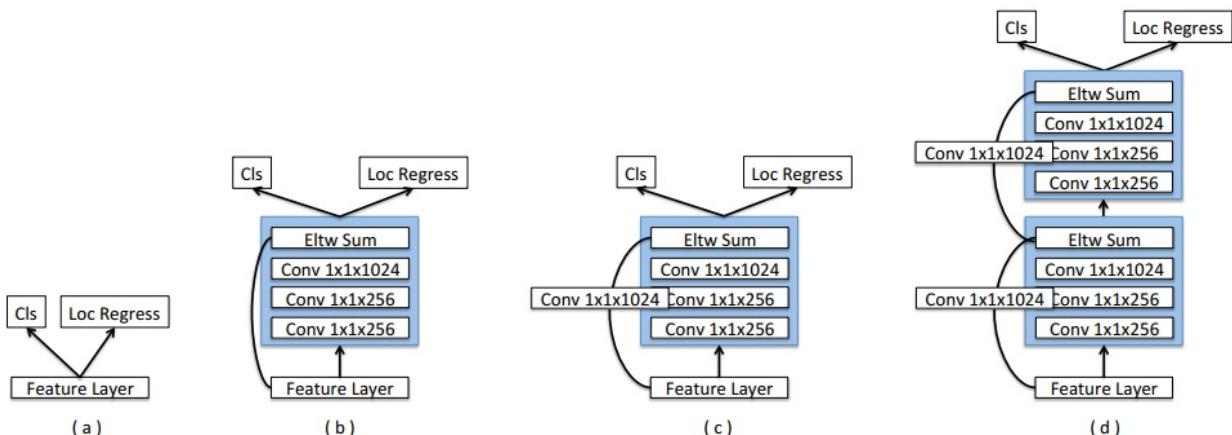


DSSD的反卷积模块分成两部分：图6的上半部分是反卷积Feature Map，其尺寸为 $H \times W$ ；图6的下半部分是SSD的Feature Map，其尺寸是反卷积Feature Map的二倍，即 $2H \times 2W$ ，进行了两组卷积和BN操作，得到一组 $2H \times 2W$ 的Feature Map。在反卷积部分中，通过步长为2的反卷积操作和一组 3×3 卷积得到 $2H \times 2W$ 的Feature Map。最后通过单位积操作和一个ReLU激活函数得到最终 $2W \times 2H$ 的Feature Map。同时作者也尝试了单位和操作，但是效果并不如单位积。

2.3 预测模块

作者在反卷积模块之后尝试了几种预测模块，图7。其中(a)是最常见的预测模块，例如SSD，YOLO；(b)和(c)分别是YOLOv2和YOLOv3采用的模块，不同的是YOLO需要上采样或者将采样到相同的尺寸。(c)是DSSD采用的预测模块，作者同时尝试了图7所有模块，实验结果表明(c)在DSSD中表现最好。

图7：DSSD中预测模块的几个变种



2.4 DSSD的锚点聚类

DSSD的锚点比例也采用了YOLOv2的思想对Ground Truth进行了聚类分析的方式得到。由于大部分Ground Truth的比例都在 $[1, 3]$ ，所以作者设置了三个比例的锚点 $(1.6, 2.0, 3.0)$ 。

小结

SSD算法的核心点在于

1. 使用多尺度的Feature Map提取特征；
2. 利用Faster R-CNN的锚点机制改进候选框。

DSSD的提出时间则较晚，其主要特别是反卷积的引入，从最近的趋势可以看出，物体检测和语义分割的交集越来越多，双方都不断的从对方汲取灵感来源来优化对应任务。

YOLO9000: Better, Faster, Stronger

前言

在这篇像极了奥运格言 (Faster, Higher, Stronger) 的论文 [55] 中，作者提出了YOLOv2和YOLO9000两个模型。

其中YOLOv2采用了若干技巧对YOLOv1 [56] 的速度和精度进行了提升。其中比较有趣的有以下几点：

1. 使用聚类产生的锚点代替Faster R-CNN[60] 和SSD[54] 手工设计的锚点；
2. 在高分辨率图像上进行迁移学习，提升网络对高分辨图像的响应能力；
3. 训练过程图像的尺寸不再固定，提升网络对不同训练数据的泛化能力。

除了以上三点，YOLO还使用了残差网络的直接映射的思想，R-CNN系列的预测相对位移的思想，Batch Normalization，全卷积等思想。YOLOv2将算法的速度和精度均提升到了一个新的高度。正是所谓的速度更快 (Faster)，精度更高 (Better/Higher)

论文中提出的另外一个模型YOLO9000非常巧妙的使用了WordNet [36]的方式将检测数据集COCO和分类数据集ImageNet整理成一个多叉树，再通过提出的联合训练方法高效的训练多叉树对应的损失函数。YOLO9000是一个非常强大 (Stronger) 且有趣的模型，非常具有研究前景。

在下面的章节中，我们将论文分成YOLOv2和YOLO9000两个部分并结合论文和源码对算法进行详细解析。

1. YOLOv2: Better, Faster

YOLOv2使用的是和YOLOv1相同的思路，算法流程参考YOLOv1的介绍，但看这篇文章肯定会感到一头雾水，因为论文并没有详细介绍YOLOv2的详细流程。而且我也不打算介绍，因为这只是对YOLOv1无畏的重复，强烈推荐读者能搞懂YOLOv1之后再来读这篇文章。

1.1. 更好 (Better)

YOLOv1之后，一系列算法和技巧的提出极大的提高了深度学习在各个领域的泛化能力。作者总结了可能在物体检测中有用的方法和技巧 (图1) 并将它们结合成了我们要介绍的YOLOv2。所以YOLOv2并没有像SSD或者Faster R-CNN具有很大的难度，更多的是在YOLOv1基础上的技巧方向的提升。在下面的篇幅中，我们将采用和论文相同的结构并结合基于Keras的源码对YOLOv2中涉及的技巧进行讲解。

图1：YOLOv2中使用的技巧及带来的性能提升

	YOLO							YOLOv2
batch norm?	✓	✓	✓	✓	✓	✓	✓	✓
hi-res classifier?		✓	✓	✓	✓	✓	✓	✓
convolutional?			✓	✓	✓	✓	✓	✓
anchor boxes?				✓	✓			
new network?					✓	✓	✓	✓
dimension priors?						✓	✓	✓
location prediction?						✓	✓	✓
passthrough?							✓	✓
multi-scale?								✓
hi-res detector?								✓
VOC2007 mAP	63.4	65.8	69.5	69.2	69.6	74.4	75.4	76.8
								78.6

1.1.1. BN替代Dropout

YOLOv2中作者舍弃了Dropout而使用Batch Normalization (BN) 来减轻模型的过拟合问题，从图1中我们可以看出BN带来了2.4%的mAP的性能提升。

Batch Normalization和Dropout均有正则化的作用。但是Batch Normalization具有提升模型优化的作用，这点是Dropout不具备的。所以BN更适合用于数据量比较大的场景。

关于BN和Dropout的异同，可以参考[Ian Goodfellow在Quora上的讨论](#)。

1.1.2. 高分辨率的迁移学习

之前的深度学习模型很多均是生搬在ImageNet上训练好的模型做迁移学习。由于迁移学习的模型是在尺寸为 224×224 的输入图像上进行训练的，进而限制了检测图像的尺寸也是 224×224 。在ImageNet上图像的尺寸一般在500左右，降采样到 224 的方案对检测任务的负面影响要远远大于分类任务。

为了提升模型对高分辨率图像的响应能力，作者先使用尺寸为 448×448 的ImageNet图片训练了10个Epoch（并没有训练到收敛，可能考虑 448×448 的图片的一个Epoch时间要远长于 224×224 的图片），然后再在检测数据集上进行模型微调。图1显示该技巧带来了3.7%的性能提升

1.1.3. 骨干网络Darknet-19

YOLOv2使用了DarkNet-19作为骨干网络（图2），在这里我们需要注意两点：

1. YOLOv2输入网络的图像尺寸并不是图2画的 224×224 ，而是使用了 416×416 的输入图像，原因我们随后会介绍；
2. 在 3×3 卷积中间添加了 1×1 卷积，Feature Map之间的一层非线性变化提升了模型的表

现能力；

3. Darknet-19进行了5次降采样，但是在最后一层卷积并没有添加池化层，目的是为了获得更高分辨率的Feature Map；
4. Darknet-19中并不含有全连接，使用的是全局平均池化的方式产生长度固定的特征向量。

图2：Darknet网络结构

Type	Filters	Size/Stride	Output
Convolutional	32	3×3	224×224
Maxpool		$2 \times 2/2$	112×112
Convolutional	64	3×3	112×112
Maxpool		$2 \times 2/2$	56×56
Convolutional	128	3×3	56×56
Convolutional	64	1×1	56×56
Convolutional	128	3×3	56×56
Maxpool		$2 \times 2/2$	28×28
Convolutional	256	3×3	28×28
Convolutional	128	1×1	28×28
Convolutional	256	3×3	28×28
Maxpool		$2 \times 2/2$	14×14
Convolutional	512	3×3	14×14
Convolutional	256	1×1	14×14
Convolutional	512	3×3	14×14
Convolutional	256	1×1	14×14
Convolutional	512	3×3	14×14
Maxpool		$2 \times 2/2$	7×7
Convolutional	1024	3×3	7×7
Convolutional	512	1×1	7×7
Convolutional	1024	3×3	7×7
Convolutional	512	1×1	7×7
Convolutional	1024	3×3	7×7
Convolutional	1000	1×1	7×7
Avgpool		Global	1000
Softmax			

首先，YOLOv2使用的是 416×416 的输入图像，考虑到很多情况下待检测物体的中心点容易出现在图像的中央，所以使用 416×416 经过5次降采样之后生成的Feature Map的尺寸是 13×13 ，这种奇数尺寸的Feature Map获得的中心点的特征向量更准确。其实这也和YOLOv1产生 7×7 的理念是相同的。

其次，YOLOv2也学习了Faster R-CNN和RPN中的锚点机制，锚点也可以被叫做先验框，即给出一个检测框可能的形状，向先验框的收敛总是比向固定box的收敛要容易的多。不同于以上两种算法的是YOLOv2使用的是在训练集上使用了K-means聚类产生的候选框，而上面两种方法的候选框是人工设计的。

最后，关于全卷积的作用， 1×1 卷积带来的非线性变化我们已经在之前的文章中多次提及，这里便不再说明。

1.1.4. 锚点聚类

在1.1.3节中我们介绍到YOLOv2使用的k-means聚类产生的锚点，该方法提出的动机是考虑到人工设计的锚点具有太强的主观性，与其主管设计，不如根据训练集学习一组更具有代表性的锚点。

由于锚点的中心即是Grid的中心点（YOLOv1是 7×7 的Grid，YOLOv2是 13×13 的Grid），所以所需要聚类的只有锚点的宽（ w ）和高（ h ）。更形象的解释就是对训练集的Ground Truth的聚类，聚类的分类目标是Ground Truth不同的大小和不同的长宽比。

类别数目为 k 的k-means可以简单总结为四步：

1. 随机初始化 k 个中心点；
2. 根据样本和中心点间的欧式距离确定样本所属的类别；
3. 根据样本的类别更新样本的中心点；
4. 循环执行2，3直到中心点的位置不再变化。

YOLOv2并没有直接使用欧式距离作为聚类标准，因为大尺寸的样本产生的误差要比小样本大。锚点作为候选框的先验，当然是希望锚点与Ground Truth的IoU越大越好，所以这里使用了IoU作为分类标准，即：

$$d(box, centroid) = 1 - IOU(box, centroid)$$

聚类的数目 k 是一个超参数，经过一系列的对比试验（图3），出于对速度和精度的折中考虑，YOLOv2使用的是 $k = 5$ 的值。

图3：k-means的 k 和IoU均值的实验对比



遗憾的是我并没有在源码中找到k-means的实现，在darknet源码中找到了两组值：

```

# coco
anchors = 0.57273, 0.677385, 1.87446, 2.06253, 3.33843, 5.47434, 7.88282, 3.52778, 9.
77052, 9.16828

# voc
anchors = 1.3221, 1.73145, 3.19275, 4.00944, 5.05587, 8.09892, 9.47112, 4.84053, 11.2
364, 10.0071

```

上面的值应该是锚点在 13×13 的Feature Map上的尺寸，可以等比例换算到原图中（图3右侧部分）。从上面的两组值我们也可以看出coco数据集的物体尺寸更偏小一些。

为了验证上面总结的思想，我用python实现了一份用于锚点聚类的k-means，源码见[链接](#)。核心算法见代码片段1：

代码片段1：用于锚点聚类的k-means

```

def kmeans(boxes, k, dist=np.median):
    """
    Calculates k-means clustering with the Intersection over Union (IoU) metric.
    """

    rows = boxes.shape[0]
    distances = np.empty((rows, k))
    last_clusters = np.zeros((rows,))

    # initiate centroids
    clusters = boxes[np.random.choice(rows, k, replace=False)]
    while True:
        for row in range(rows):
            distances[row] = 1 - iou(boxes[row], clusters)
        nearest_clusters = np.argmin(distances, axis=1)
        if (last_clusters == nearest_clusters).all():
            break
        for cluster in range(k):
            clusters[cluster] = dist(boxes[nearest_clusters == cluster], axis=0)
        last_clusters = nearest_clusters
    return clusters

```

跑了一下在pascal voc上的聚类，得到的 $k = 5$ 和 $k = 9$ 的实验结果如下：

```

# k=5
Accuracy: 60.07%
Boxes:
[[4.446      6.48266667]
 [2.106      3.95502646]
 [1.17       1.8744186 ]
 [9.81066667 9.91591592]
 [0.494      0.90133333]]
Ratios:
[0.53, 0.55, 0.62, 0.69, 0.99]

# k=9
Accuracy: 66.66%
Boxes:
[[10.738     10.192      ]
 [ 1.56       1.456      ]
 [ 0.884      2.704      ]
 [ 3.562      3.12       ]
 [ 5.75612121 7.67      ]
 [ 0.494      0.69333333]
 [ 2.938      6.526      ]
 [ 0.572      1.42133333]
 [ 1.638      3.98666667]]
Ratios:
[0.33, 0.4, 0.41, 0.45, 0.71, 0.75, 1.05, 1.07, 1.14]

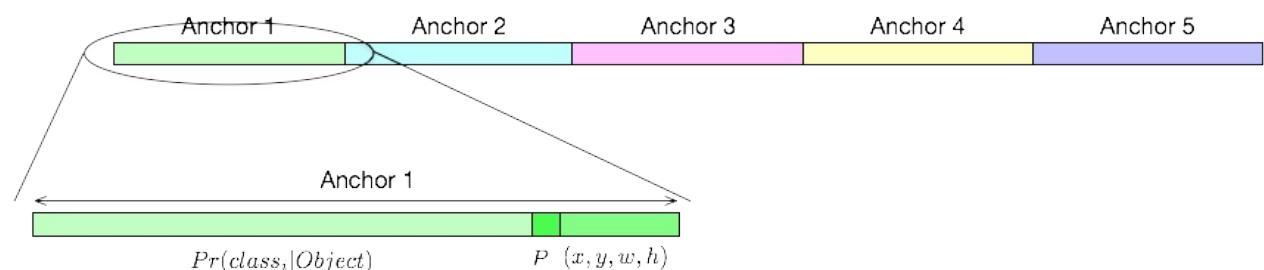
```

虽然和源码提供的值不完全一样，但是取得的先验框和源码的差距很小，而且IoU也基本符合图3给出的实验结果。

1.1.5. 直接位置预测

YOLOv2使用了和YOLOv1类似的损失函数，不同的是YOLOv2将分类任务从cell中解耦。因为在YOLOv1中，cell负责预测与之匹配的类别，bounding box负责位置精校，也就是预测位置。YOLOv1的输出层我们在[YOLOv1](#)的图3中进行了描述。但是在YOLOv2中使用了锚点机制，物体的类别和位置均是由锚点对应的特征向量决定的，如图4。

图4：YOLOv2的输出层



在Keras源码中使用的是80类的COCO数据集，锚点数 $k = 5$ ，所以YOLOv2的每个cell的输出层有 $(80 + 5) \times 5 = 425$ 个节点。

直接将锚点机制添加到YOLO中（也就是SSD）会产生模型不稳定的问题，尤其在早期迭代的时候，这些不稳定大部分是发生在预测 (x, y) 的时候。

回顾一下SSD的损失函数，相对位移 (x, y) 的计算方式为：

$$\hat{g}_j^{cx} = (g_j^{cx} - d_i^{cx})/d_i^w$$

$$\hat{g}_j^{cy} = (g_j^{cy} - d_i^{cy})/d_i^h$$

即

$$g_j^{cx} = \hat{g}_j^{cx} * d_i^w + d_i^{cx}$$

$$g_j^{cy} = \hat{g}_j^{cy} * d_i^h + d_i^{cy}$$

注意论文在这个地方是减号，应该是一个错误。

其中 \hat{g}_j^{cx} 和 \hat{g}_j^{cy} 是预测值，在模型训练初期，由于使用的是迁移学习和随机初始化得到的网络，此时网络必然收敛的不是特别好，这就造成了 \hat{g}_j^{cx} 和 \hat{g}_j^{cy} 的随机性。再加上并没有对 \hat{g} 的值加以限制，使得预测的检测框可能出现在网络中的任何位置而且这些事和锚点如何初始化无关的。正确的做法应该是预测的检测框应该也是由该锚点负责的，也就预测的检测框的中心点应该落到该锚点的内部。

为了解决这个问题，YOLOv2采用了YOLOv1中使用的相对一个cell的位移，同时使用了logistic函数将预测值限制在了 $[0, 1]$ 的范围内。YOLOv2的输出层会产生5个值，

$(t_x, t_y, t_w, t_j, t_o)$ ，该输出层对应cell的左上角为 (c_x, c_y) ，宽和高分别为 (p_w, p_h) ，那么对应的预测的相对位移为：

$$b_x = \sigma(t_x) + c_x$$

$$b_y = \sigma(t_y) + c_y$$

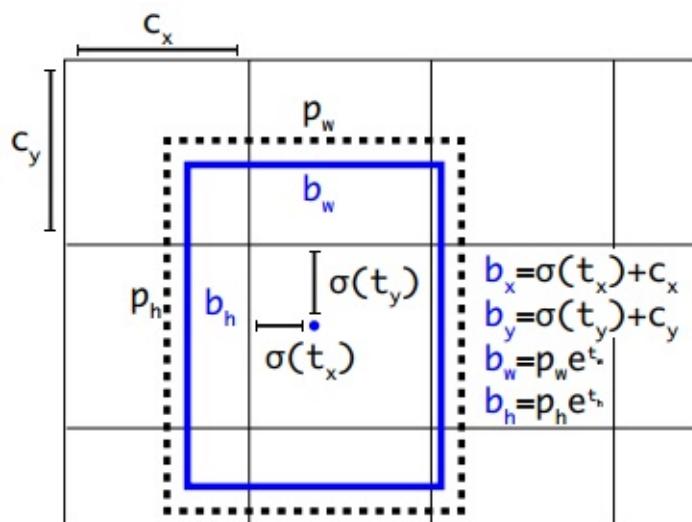
$$b_w = p_w e^{t_w}$$

$$b_h = p_h e^{t_h}$$

$$Pr(object) \times IoU(b, object) = \sigma(t_o)$$

上式表示的几何关系见图5。

图5：YOLOv2的预测值和匹配cell的几何关系



YOLOv2的损失函数 `./utils/loss_util.py` 和YOLOv1的是相同的，均是由5个任务组成的多任务损失函数。源码中各个模型的权重也和YOLOv1中提到的权重一致。

从图1中我们可以看出，1.1.4节的Dimension Clusters加上1.1.5节的Direct Location Prediction非常有效的将mAP提高了4.8%。

1.1.6. 细粒度特征

在特征在物体检测和语义分割中的研究中表明，使用多尺度的是非常有效的。YOLOv2的多尺度是通过将最后一个 26×26 的Feature Map直接映射到最后一个 13×13 的Feature Map上形成的。

这里面有两个技术细节需要详细说明一下：

1. $26 \times 26 \times 512$ 的Feature Map首先通过TensorFlow的 `space_to_depth()` 函数转换成 $13 \times 13 \times 2048$ 的Feature Map（图6）然后再和后面的Feature Map进行映射的；
2. 论文中采用的是类似残差网络的映射方式，也就是将Feature Map执行单位加操作，但是源码中使用的是DenseNet的方式，也就是Merge成 $13 \times 13 \times (2048 + 1024)$ 的 Feature Map。

图6 : `tf.space_to_depth()`

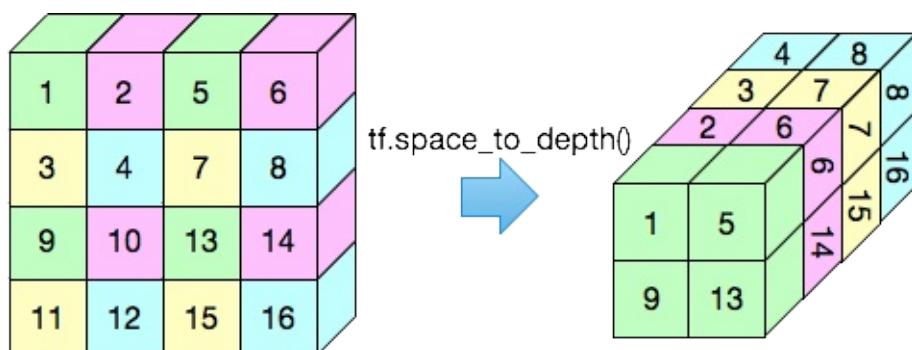


图1显示该方法带来了1%的性能提升。

1.1.7. 多尺度训练

全卷积的使用是网络在单词测试环境中每张输入图像的尺寸可以不同。但是当使用mini-batch方式训练的时候，图像的尺寸要是相同的，因为TensorFlow等框架要求输入网络的是一个维度为 $N \times W \times H \times C$ 的矩阵。其中 N 是批（batch）的大小， W, H, C 分别是图片的宽，高和通道数。所以虽然每个batch之内图像的尺寸必须是相同的，但是不同的batch之间图像的尺寸是不受限于框架的。YOLOv2便是基于这点实现了其训练过程中的多尺度。

在YOLOv2中，每隔10个batch便随机从320, 352, 384, ..., 608选择一个新的尺度作为输入图像的尺寸，多尺度训练将mAP提高了1.4%。

1.2. 更快 (Faster)

YOLOv2用于提速的技术我们已经在1.1节中介绍过，这里仅列出技术和提速的关系：

1. 使用全卷积网络代替全连接，网络具有更少的参数，速度更快；
2. 使用Batch Normalization代替Dropout的正则化的功能，使用BN训练的模型更稳定；
3. DarkNet-19将VGG-16运算数量从306.9亿降低到55.8亿。

文至此处，一个更快，更好的YOLOv2已介绍完毕，虽然不像SSD对YOLOv1的提升在技术上那么惊艳，但其使用的若干技术确实是非常有效。在下一部分我们将开始介绍YOLO9000，一个无论在技术，还是再创新点上都非常惊艳的模型。

2. YOLO9000: 更强 (Stronger)

在80类的COCO数据集中，物体的类别都是比较抽象的，例如类别‘dog’并没有精确到具体狗的品种（哈士奇或者柯基等）。而ImageNet中包含的类别则更具体，不仅包含‘dog’类，还包括‘poodle’和‘Corgi’类。我们将COCO数据集的狗的图片放到训练好的ImageNet模型中理论上是能判断出狗的品种的，同理我们将ImageNet中的狗的图片（不管是贵宾犬，还是柯基犬）放在COCO训练好的检测模型中，理论上是能够检测出来的。但是生硬的使用两个模型是非常愚蠢且低效的。YOLO9000的提出便是巧妙地利用了COCO数据集提供的检测标签和ImageNet强大的分类标签，使得训练出来的模型具有强大的检测和分类的能力。

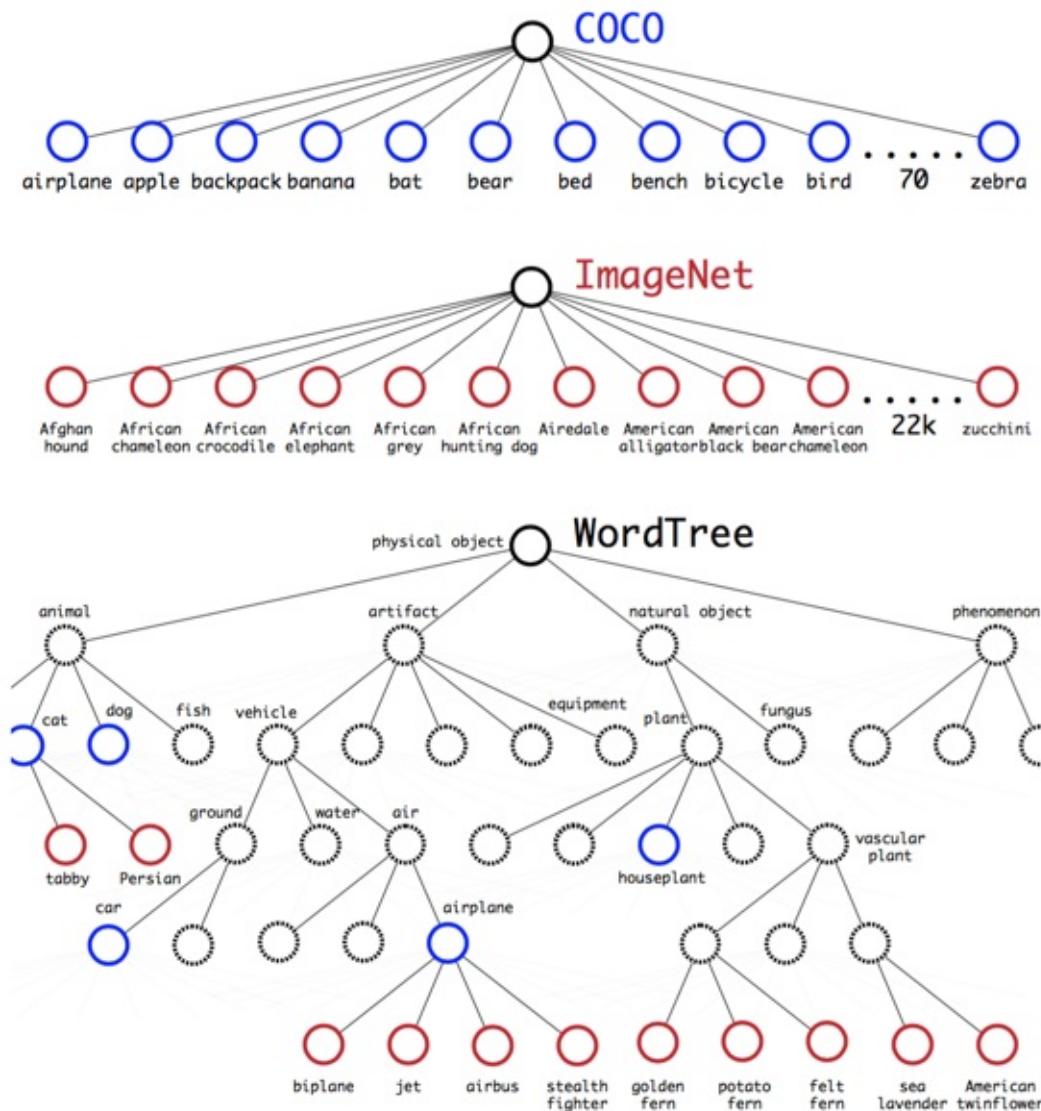
遗憾的是并没有找打YOLO9000的TensorFlow或是Keras源码，暂且用基于DarkNet的[源码](#)分析之：

2.1. 分层分类

ImageNet的数据集的标签是通过WordNet[5]的方式组织的，WordNet反应了物体类别之间的语义关系，例如‘dog’类既是‘canine’的子类，也是‘domestic animal’的子类，由于一个子节点有两个父节点，所以WordNet本质上是一个图模型。

在YOLOv2中，作者将WordNet简化成了一个分层的树结构，即WordTree。WordTree的生成方式也很简单，如果一个节点含有多个父节点，只需要保存到根节点路径最短的那条路径即可，生成的层次树模型见图7。在DarkNet的源码中，WordTree以二进制文件的形式保存在[./data/9k.tree](#)文件中。在9k.tree中，第一列表示类别的标签，标签的类别可以在[./data/9k.names](#)通过行数（从0开始计数）对应上，9k.tree的第二列表示该节点的父节点，值为-1的话表示父节点为空。

图7：YOLO9000的WordTree



例如从8888(military officer)行开始向上回溯到根节点，走过的路径依次是：

6920(corgi) -> 6912(dog) -> 6856(canine) -> 6781(carnivore) -> 6767(placental) ->
6522(mammal) -> 6519(vertebrate) -> 6468(chordate) -> 5174(animal) -> 5170(worsted) ->
1042(living thing) -> 865(whole) -> 2(object)-> -1

貌似问题不大。

现在问题是如果标签是以WordTree的形式组织的，我们如何确定检测的物体属于哪一类呢？在使用WordTree进行分类时，我们预测每个节点的条件概率，以得到同义词集合（synset）中每个同义词的下义词（hyponym）的概率，例如在‘dog’节点处我们要预测

$$Pr(poodle|dog)$$

$$Pr(Corgi|dog)$$

$$Pr(griffon|dog)$$

...

当我们要预测一只狗是不是柯基时， $Pr(Corgi)$ 是一系列条件概率的乘积：

$$Pr(Corgi) = Pr(Corgi|dog) \times Pr(dog|canine) \times \dots \times Pr(livingthing|whole) \times Pr(whole|object)$$

其中 $Pr(object) = 1$ 。 $Pr(Corgi|dog)$ 则是在‘dog’的所有下义词中为‘Corgi’的概率，由softmax激活函数求得，其它情况依次类推（图8）。

图8：在ImageNet和在WordTree下的预测。

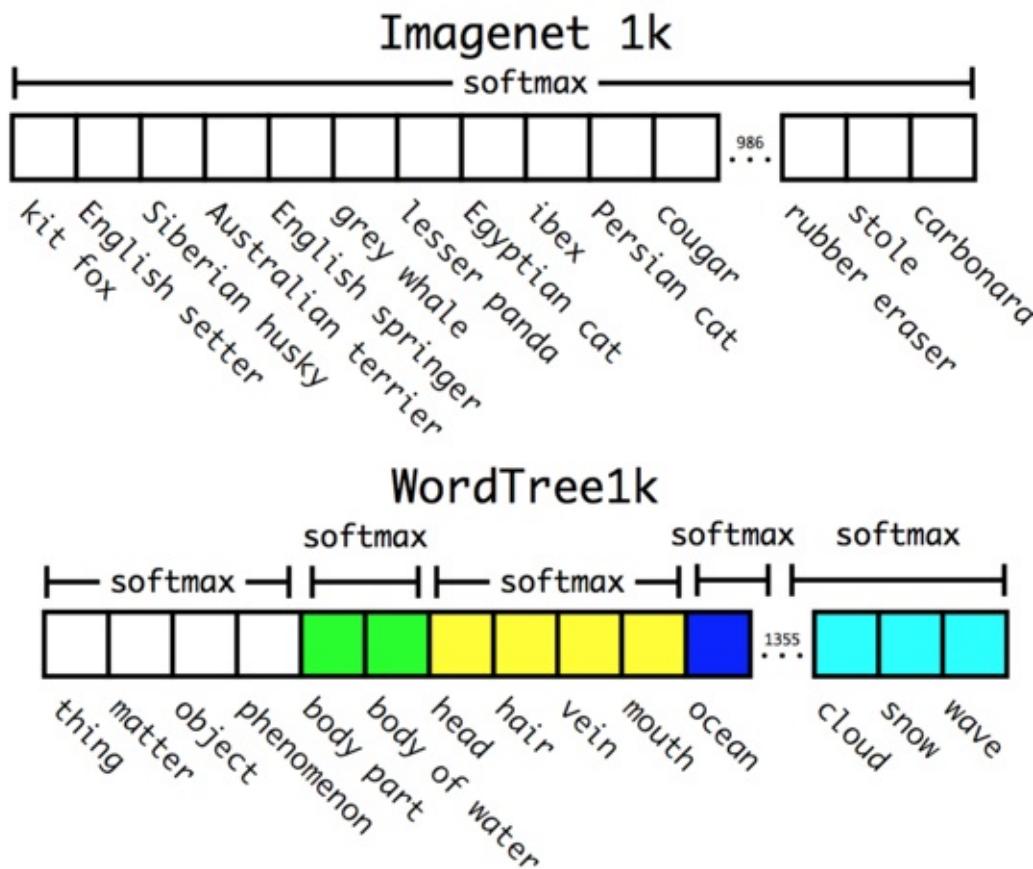


图8是作者为了验证其想法建立的WordTree 1k模型，在构建WordTree时添加了369个中间节点以便构成一个完整的WordTree。根据上面的分析，YOLO9000是一个多标签分类的模型，例如‘Corgi’则是一个含有1369个标签的数据，其one-hot编码的形式为在第 (6920, 6912, 6856, 6781, 6767, 6522, 6519, 6468, 5174, 5170, 1042, 865, 2) 共13个位置处为1，其余的位置均为0。

在预测物体的类别时，我们遍历整个WordTree，在每个分割中采用最高的置信度路径，直到分类概率小于某个阈值（源码给的是0.6）时，然后预测结果。

2.2. 使用WordTree合并数据集

WordTree的类别数量非常大，基本可以囊括目前所有的检测数据集，只需要在WordTree中标明哪些节点是检测数据集上的即可，图7显示的是COCO数据集合并到WordTree的结果。其中蓝色节点表示COCO中可以检测的类别。

2.3. 检测和分类的联合训练

WordTree 1k的实验证了作者的猜测，作者更大胆的将分类任务扩大到了整个ImageNet数据集。YOLO9000提取了ImageNet最常出现的top-9000个类别并在构建WordTree时将类别数扩大到了9418类。由于同时使用了ImageNet和COCO数据集进行训练，为了平衡分类和检测任务，YOLO9000将COCO上采样到和ImageNet的比例为1:4。

YOLO9000使用了YOLOv2的框架但是有以下改进：

1. 每个锚点的输出不再是85个，而是 $9418 + 5 = 9423$ 个；
2. 为了减少每个cell的输出节点数，YOLO9000使用了3个锚点，但每个cell的输出也达到了 $3 \times 9423 = 28269$ 个；
3. 当运行分类任务时，要更新该节点和其所有父节点的权值，且不更新检测任务的权值，因为我们此时根本没有任何关于检测的信息；
4. 锚点和Ground Truth的IoU大于0.3时便被判为正锚点，而YOLOv2的阈值是0.5。

总结

YOLO9000这篇论文算是干货满满的一篇文章，首先YOLOv2通过一系列非常有效的Trick将物体检测的速度和精度刷新到了新的高度。这些Trick不仅在YOLOv2中非常有效，而且对我们的其它任务也很有参考价值，例如高分辨率迁移学习应用到语义分割，多尺度训练应用到图像分类任务等。

YOLO9000更是强大到令人发指，其对COCO的80类的子类和父类能进行检测并不让我感到意外，YOLO9000强大之处在于路径中没有COCO类别的156类中也取得了非常不错的效果。这种半监督学习是非常有研究和应用前景的一个方向，因为在我们的大部分场景中获得大量数据集难度非常大，但我们又可以多少搞到写数据，这时候就要发挥半监督学习的作用了。

Focal Loss for Dense Object Detection

tags: FPN, ResNet, Focal Loss, RetinaNet

前言

何凯明，RBG等人一直是Two-Stage方向的领军人，在这篇论文中，他们也开始涉足One-Stage的物体检测算法。大牛就是牛，一次就刷新了精度。下面我们就来分析这几个大牛的作品。

目前主流的检测算法分为两个方向：（1）以[R-CNN\[63\]](#)系列为代表的two-stage方向；（2）以[YOLO\[56\]](#)系列为代表的one-stage方向。虽然one-stage方向的速度更快，但是其精度往往比较低。究其原因，有两个方面：

1. 正样本（Positive Example）和负样本（Negative Example）的不平衡；
2. 难样本（Hard Example）和易样本（Easy Example）的不平衡。

这些不平衡造成模型的效果不准确的原因如下：

1. Negative example的数量过多，导致Positive example的loss被覆盖，就算Positive example的loss非常大也会被数量庞大的 negative example中和掉，这些positive example往往是我们要检测的前景区域；
2. Hard example往往是前景和背景区域的过渡部分，因为这些样本很难区分，所以叫做Hard Example。剩下的那些Easy example往往很好计算，导致模型非常容易就收敛了。但是损失函数收敛了并不代表模型效果好，因为我们其实更需要把那些hard example训练好。

四种example的情况见图1。

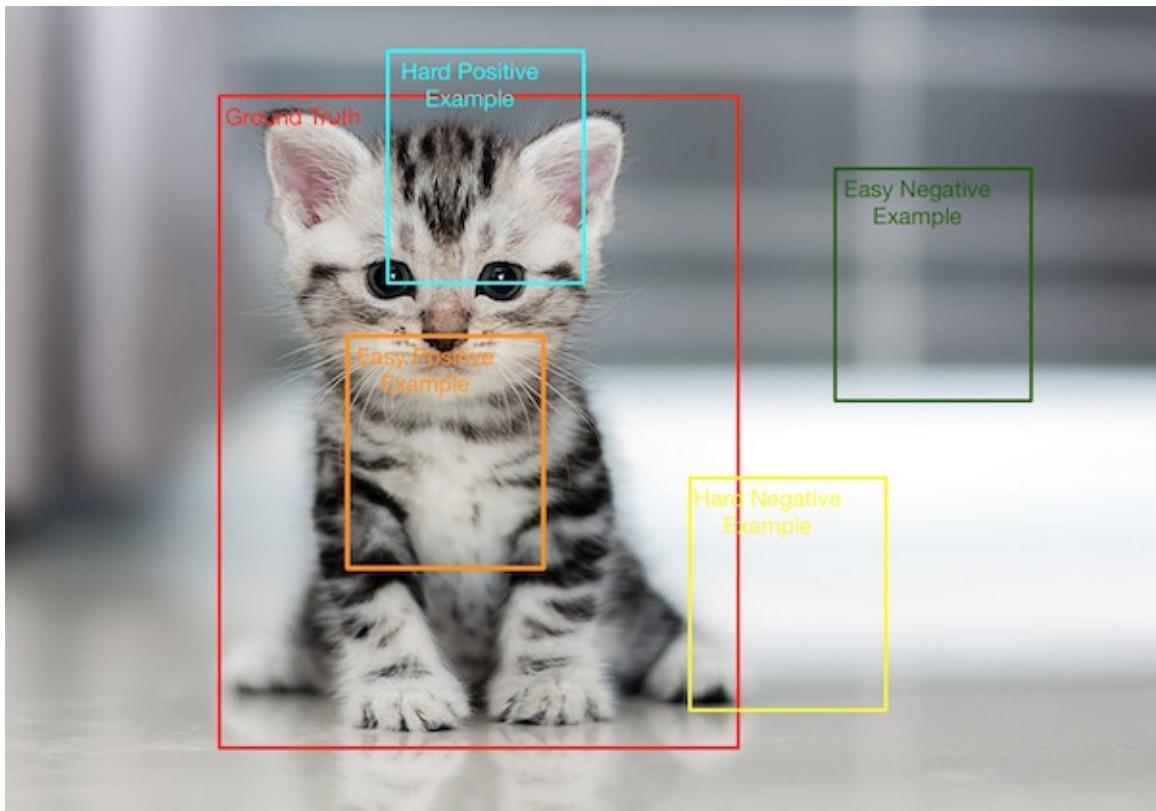


图1：物体检测中的四种Example

Faster R-CNN之所以能解决两个不平衡问题是因为其采用了下面两个策略：

1. 根据IoU采样候选区域，并将正负样本的比例设置成1:1。这样就解决了正负样本不平衡的问题；
2. 根据score过滤掉easy example，避免了训练loss被easy example所支配的问题。

而在这篇论文中他们采用的解决方案是基于交叉熵提出了一个新的损失函数Focal Loss (FL)。

$$FL(p_t) = -\alpha_t(1 - p_t)^\gamma \log(p_t)$$

FL是一个尺度动态可调的交叉熵损失函数，在FL中有两个参数 α_t 和 γ ，其中 α_t 主要作用是解决正负样本的不平衡， γ 主要是解决难易样本的不平衡。

最后，作者基于[残差网络\[118\]](#)，[FPN\[118\]](#)搭建了检测网络RetinaNet，该网络使用的策略都是他们自己提出的而且目前效果非常好的基础结构，再结合Focal Loss，刷新检测算法的精度也不意外[\[58\]](#)。

1. Focal Loss

Focal Loss是交叉熵损失的改进版本，一个二分类交叉熵可以表示为：

$$\text{CE}(p, y) = \begin{cases} -\log(p) & \text{if } y = 1 \\ -\log(1 - p) & \text{otherwise} \end{cases}$$

上面公式可以简写成：

$$\text{CE}(p, y) = \text{CE}(p_t) = -\log(p_t)$$

其中：

$$p_t = \begin{cases} p & \text{if } y = 1 \\ 1 - p & \text{otherwise} \end{cases}$$

1.1 α ：解决正负样本不平衡

平衡交叉熵的提出是为了解决正负样本不平衡的问题的。它的原理很简单，为正负样本分配不同的权重比值 $\alpha \in [0, 1]$ ，当 $y = 1$ 时取 α ，为 -1 时取 $1 - \alpha$ 。我们使用和 p_t 类似的方法将上面 α 的情况表示为 α_t ，即：

$$\alpha_t = \begin{cases} \alpha & \text{if } y = 1 \\ 1 - \alpha & \text{otherwise} \end{cases}$$

那么这个 α - balanced 交叉熵损失可以表示为式(6)。

$$\text{CE}(p_t) = -\alpha_t \log(p_t)$$

α 的值往往需要根据验证集进行调整，论文中给出的是 0.25。

1.2 γ ：解决难易样本不平衡

FL 中 γ 的引入是为了解决难易样本不平衡的问题的。图 2 是 FL 中 example 预测概率和 loss 值之间的关系。其中蓝色曲线是交叉熵 ($\gamma = 0$ 时 Focal Loss 退化为交叉熵损失) 的曲线。

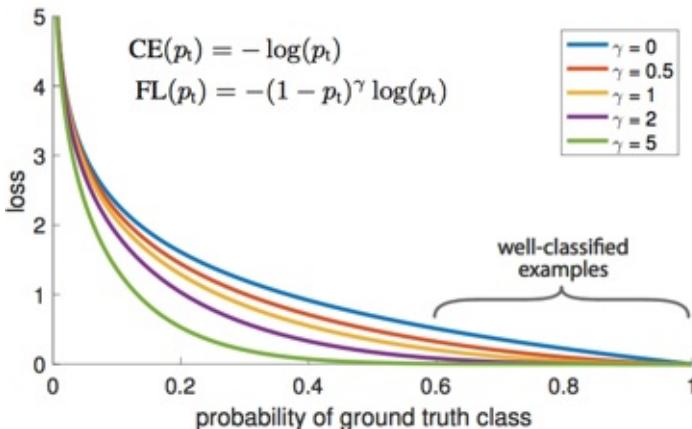


图2：CE损失和FL损失曲线图

从图2的曲线中我们可以看出对于一些well-classified examples (easy examples) 虽然它们单个example的loss可以收敛到很小，但是由于它们的数量过于庞大，把一些hard example的loss覆盖掉。导致求和之后他们依然会支配整个批次样本的收敛方向。

一个非常简单的策略是继续缩小easy examples的训练比重。作者的思路很简单，给每个乘以 $(1 - p_t)^\gamma$ 。因为easy example的score p_t 往往接近1，那么 $(1 - p_t)^\gamma$ 值会比较小，因此example得到了抑制，相对的hard example得到了放大，例如图2中 $\gamma > 0$ 的那四条曲线。

FL的求导结果如公式(7):

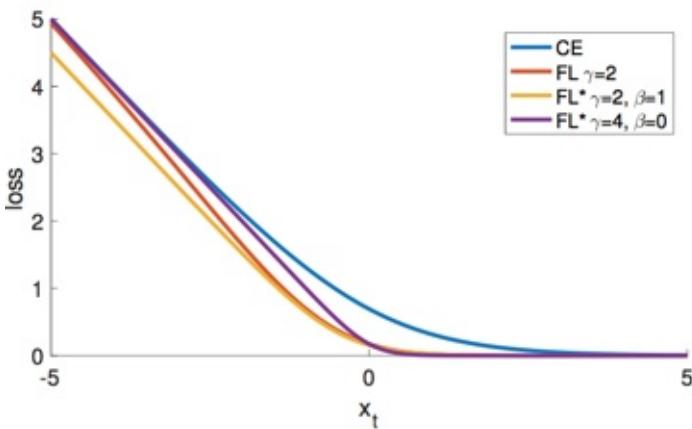
$$\frac{d\text{FL}}{dx} = y(1 - p_t)^\gamma(\gamma p_t \log(p_t) + p_t - 1)$$

γ 的值也可以根据验证集来调整，论文中给出的值是2。

1.3 FL的最终形式

结合1.1的 α 和1.2的 γ ，我们便有了公式(1)中FL的最终形式。作者也通过实验验证了结合两个策略的实验效果最好。

Focal Loss的最终形式并不是一定要严格的是(1)的情况，但是它应满前文的分析，即能缩小easy example的比重。例如在论文附录A中给出的另外一种Focal Loss：FL*，曲线见图3。它能取得和FL类似的效果。

图3：CE损失和 FL^* 损失曲线图

$$FL^* = -\frac{\log(\sigma(\gamma yx + \beta))}{\gamma}$$

最后作者指出如果将单标签softmax换成多标签的sigmoid效果会更好，这里应该和我们在YOLOv3中分析的情况类似。

2. RetinaNet

算法使用的检测框架RetinaNet并没有特别大的创新点，基本上是残差网络+FPN的最state-of-the-art的方法，如图4。

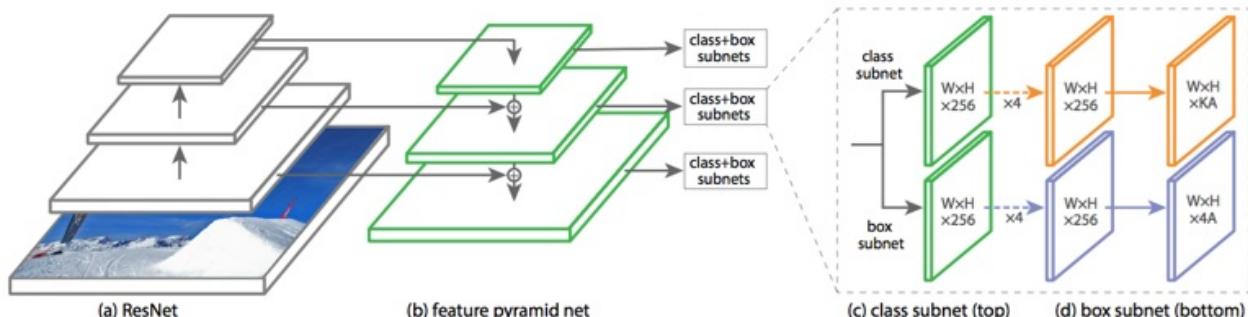


图4：RetinaNet网络结构图

对于残差网络和FPN不清楚的参考论文或者我之前的分析。这里我们列出RetinaNet的几个重点：

1. 融合的特征层是P3-P7；
2. 每个尺度的Feature Map有一组锚点（ $3*3=9$ ）；
3. 分类任务和预测任务的FPN部分的参数共享，其它参数不共享。

3. 测试

测试的时候计算所有锚点的score，再从其中选出top-1000个进行NMS，NMS的阈值是0.5。

4. 总结

Focal Loss的论文非常简单有效，非常符合何凯明等人的风格。FL中引入的两个参数 α 和 γ 分别用于抑制正负样本和难易样本的不平衡，动机明确。

Focal Loss几乎可以应用到很多imbalance数据的领域，还是非常有实用价值的。

YOLOv3: An Incremental Improvement

tags: YOLOv3, YOLOv2, YOLO

前言

YOLOv3[52]论文的干货并不多，用作者自己的话说是一篇“Tech Report”。这篇主要是在YOLOv2[55]的基础上的一些Trick尝试，有的Trick成功了，包括：

1. 考虑到检测物体的重叠情况，用多标签的方式替代了之前softmax单标签方式；
2. 骨干架构使用了更为有效的残差网络，网络深度也更深；
3. 多尺度特征使用的是FPN[58]的思想；
4. 锚点聚类成了9类。

也有一些尝试失败了，在介绍完YOLOv3的细节后我们在说明这些尝试会更好理解。在分析论文时，我们依然会使用一份Keras的[源码](#)辅助理解。

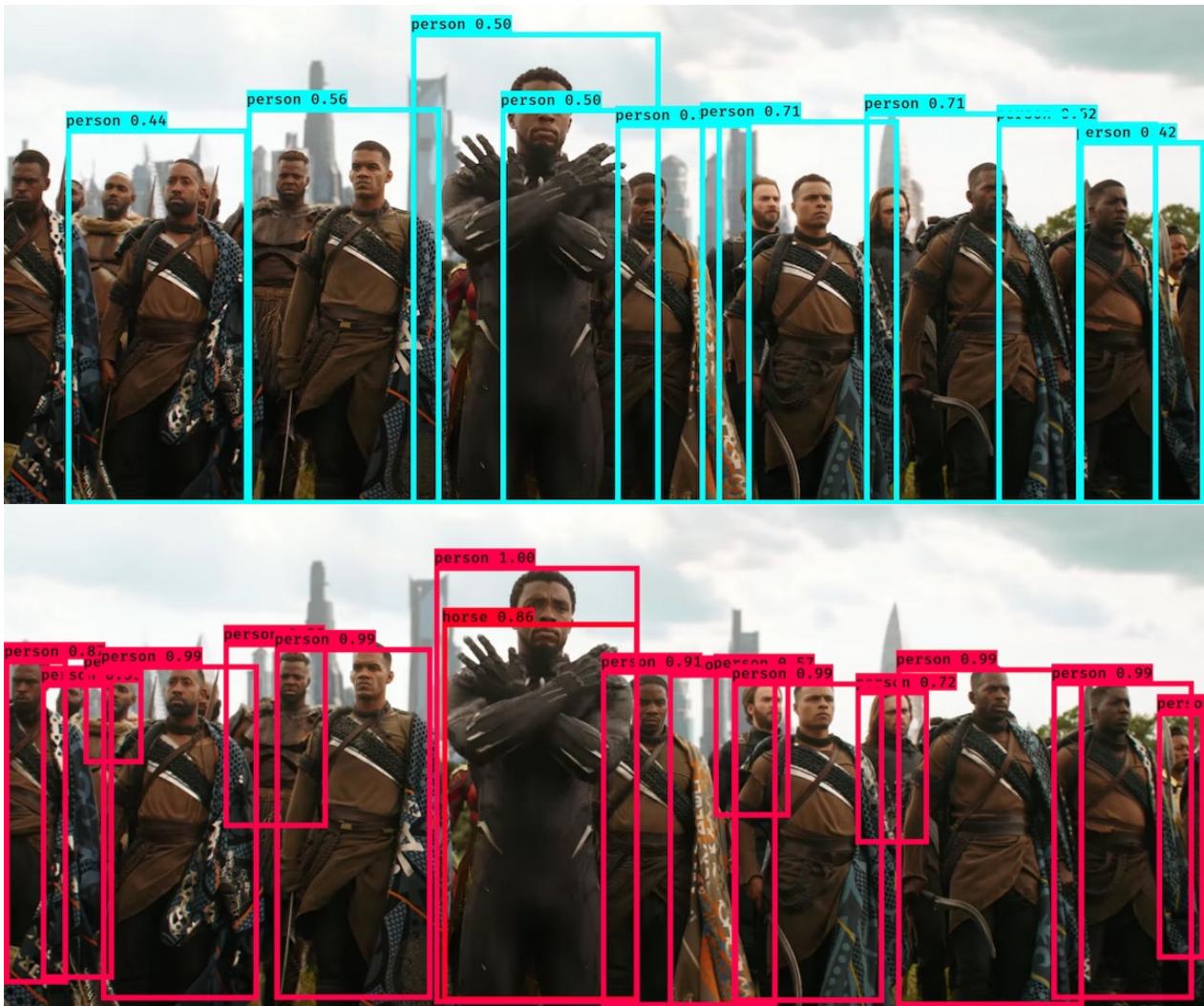
YOLOv3详解

1. 多标签任务

不管是在检测任务的标注数据集，还是在日常场景中，物体之间的相互覆盖都是不能避免的。因此一个锚点的感受野肯定会有包含两个甚至更多个不同物体的可能，在之前的方法中是选择和锚点IoU最大的Ground Truth作为匹配类别，用softmax作为激活函数。

YOLOv3多标签模型的提出，对于解决覆盖率高的图像的检测问题效果是十分显著的，图1是同一幅图在YOLOv2和YOLOv3下得到的检测结果。可以明显的看出YOLOv3的效果好很多，不仅检测的更精确，最重要的是在后排被覆盖很多的物体（例如美队和冬兵）也能很好的在YOLOv3中检测出来。

图1：YOLOv2 vs YOLOv3



YOLOv3提供的解决方案是将一个 N 路softmax分类器替换成 N 个sigmoid分类器，这样每个类的输出仍是 $[0, 1]$ 之间的一个值，但是他们的和不再是1。

虽然YOLOv3改变了输出层的激活函数，但是其锚点和Ground Truth的匹配方法仍旧采用的是YOLOv1[56]的方法，即每个Ground Truth匹配且只匹配唯一一个与其IoU最大的锚点。但是在输出的时候由于各类的概率之和不再是1，只要置信度大于阈值，该锚点便被作为检测框输出。

训练标签的制作和测试过程候选框的输出分别

在 `./yolo3/model.py` 的 `yolo_eval` 和 `preprocess_true_boxes` 函数中实现的。

2. 骨干网络

YOLOv3使用了由残差块构成的全卷积网络作为骨干网络，网络深度达到了53层，因此作者将其命名为Darknet-53。Darknet-53的详细结构见图2。

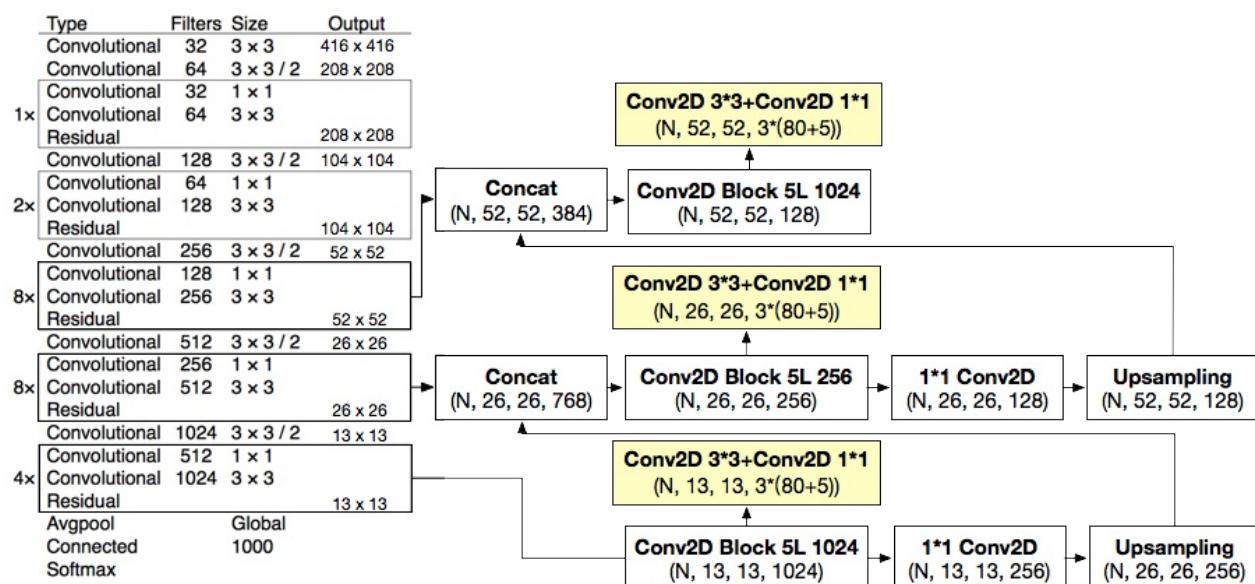
图2：Darknet-53

Type	Filters	Size	Output
Convolutional	32	3×3	416×416
Convolutional	64	$3 \times 3 / 2$	208×208
1x	Convolutional	32	1×1
1x	Convolutional	64	3×3
	Residual		208×208
	Convolutional	128	$3 \times 3 / 2$
	Convolutional	64	1×1
2x	Convolutional	128	3×3
	Residual		104×104
	Convolutional	256	$3 \times 3 / 2$
	Convolutional	128	1×1
8x	Convolutional	256	3×3
	Residual		52×52
	Convolutional	512	$3 \times 3 / 2$
	Convolutional	256	1×1
8x	Convolutional	512	3×3
	Residual		26×26
	Convolutional	1024	$3 \times 3 / 2$
	Convolutional	512	1×1
4x	Convolutional	1024	3×3
	Residual		13×13
	Avgpool		Global
	Connected		1000
	Softmax		

3. 多尺度特征

YOLOv3汲取了FPN的思想，从不从尺度上提取了特征。对比YOLOv2的只在最后两层提取特征，YOLOv3则将尺度扩大到了最后三层，图3是在图2的基础上加上多尺度特征提取部分的图示。

图3 : Darknet-53 with FPN



在多尺度特征部分强调几个关键点：

1. YOLOv2采用的是降采样的形式进行Feature Map的拼接，YOLOv3则是采用同SSD相同

- 的双线性插值的上采样方法拼接的Feature Map；
- 每个尺度的Feature Map负责对3个先验框（锚点）的预测，源码中的掩码（Mask）负责完成此任务。

4. 锚点聚类

在YOLOv2的文章中我们介绍了锚点是聚类的，作者尝试了折中考虑了速度和精度之后选择的类别数 $k = 5$ 。但是在YOLOv3中， $k = 9$ ，得到的9组锚点是：

$(10 \times 13), (16 \times 30), (33 \times 23), (30 \times 61), (62 \times 45), (59 \times 119), (116 \times 90), (156 \times 198), (373 \times 326)$

其中 13×13 的卷积核分配的尺度是 $(116 \times 90), (156 \times 198), (373 \times 326)$ ， 26×26 的卷积核分配的尺度是 $(30 \times 61), (62 \times 45), (59 \times 119)$ ， 52×52 的卷积核分配的尺度是 $(10 \times 13), (16 \times 30), (33 \times 23)$ 。这么做的原因是深度学习中层数越深，Feature Map对小尺寸物体的响应能力越弱。

5. YOLOv3一些失败的尝试

- 尝试捕捉位移 (x, y) 和检测框边长 (w, h) 的线性关系，这时方式得到的效果并不好且模型不稳定；
- 使用线性激活函数代替sigmoid激活函数预测位移 (x, y) ，该方法导致模型的mAP下降；
- 使用focal loss[35]，mAP也降了。

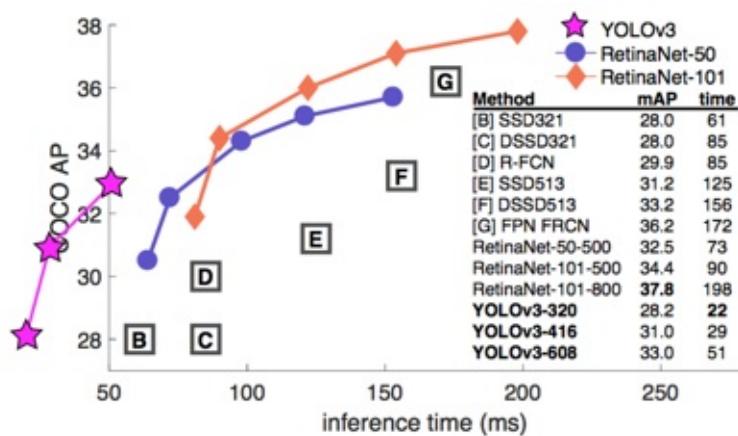
总结

至此，YOLO系列的算法整理完毕，作者的兴趣点也转向了GAN，感觉短期内不会有大的进展了。

从算法的角度讲，当业界都沉迷于R-CNN系列的方法时，作者另辟蹊径引入了单次检测的YOLO，虽然效果略差，但是其速度优势也占据了很大市场。但是作者并就看不起R-CNN系列，在YOLOv2中引入了RPN的锚点机制，在YOLOv3中引入了FPN，正所谓师夷长技以制夷，一段时间内从精度和时间实现了对R-CNN系列的全面压制。

从框架的角度讲，作者并没有使用流行的例如TensorFlow或是Caffe，而是自己开发了一套框架：Darknet。Darknet使用C语言开发在速度上大幅领先于脚本语言，一方面可以看出作者强大的编程功底，另一方面也看出了作者当初还是具有一些野心的。

从应用的角度讲，R-CNN系列虽然效果更好，但是其速度制约了其应用场景，因为其在强大的GPU环境下才勉强的实现了实时检测，R-FCN作为R-CNN系列的速度最快的算法，YOLOv3将其远远的甩在了第一象限。YOLO系列强大的性能优势使其在市场上尤其是计算资源受限的嵌入式等环境得到了广泛的应用，据我了解，一些安检系统就采用的是YOLO系列的算法。



YOLO9000是我认为在算法和应用上一个非常典型的结合体。在目前的市场上，高质量数据是深度学习领域最值钱的资源。YOLO9000采用半监督学习的方式高效的使用了目前质量最高的ImageNet数据集，设计了WordTree，从而实现了对未标注物体的高精度检测。半监督学习是未来市场上非常有前景的研究方向。

最后扯一点有意思的题外话，作者作为一个有野心的技术大牛，性格反而有点可爱和萌，给开发的网络取名“Darknet”便透露着浓浓的中二风。在写YOLOv1和YOLOv2的论文时文风还比较正经，但是到了YOLOv3文风便奔放起来，还时不时的调戏一下读者和审稿人。这些事儿你可能想不到是一个满脸大胡子的光头大汉（仔细一看还挺帅的）做的，感兴趣的话可以欣赏一下作者的[主页](#)和他在TED上的[演讲](#)。

Learning to Segment Every Thing

tags: Mask^X R-CNN, Mask R-CNN

前言

首先回顾一下YOLO系列论文，[YOLO9000\[55\]](#)通过半监督学习的方式将模型可检测的类别从80类扩展到了9418类，YOLO9000能成功的原因之一是物体分类和物体检测使用了共享的特征，而这些特征是由分类和检测的损失函数共同训练得到的。之所以采用半监督学习的方式训练YOLO9000，一个重要原因就是检测数据的昂贵性。所以作者采用了数据量较小的COCO的检测标签，数据量很大的ImageNet的分类标签做半监督学习的样本，分别训练多任务模型的检测分支和分类分支，进而得到了可以同时进行分类和检测的特征。

之所以在最开始回顾YOLO9000，是因为我们这里要分析的[Mask^X R-CNN \[51\]](#)和YOLO9000的动机和设计是如此的相同：

1. 他们都是在多任务模型中使用半监督学习来完成自己的任务，YOLO9000是用来做检测，[Mask^X R-CNN](#)是用来做语义分割；
2. 之所以使用半监督学习，因为他们想实现一个通用的模型，所以面临了数据量不够的问题：对比检测任务，语义分割的数据集更为稀缺（COCO的80类，Pascal VOC的20类），但是Visual Genome（VG）数据集却有3000类，108077张带有bounding box的样本；
3. 它们的框架算法都是继承自另外的框架：YOLO9000继承自YOLOv2，[Mask^X R-CNN](#)继承自[Mask R-CNN\[63\]](#)。

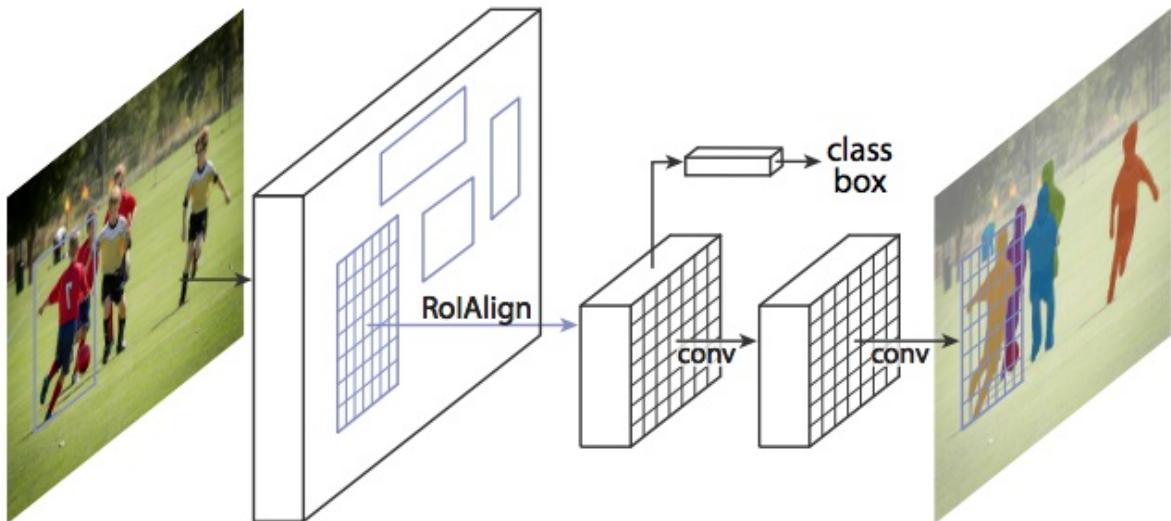
不同于YOLO9000通过构建WordTree的数据结构来使用两个数据集，[Mask^X R-CNN](#)提出了一个叫做权值迁移函数(weight transfer function)的迁移学习方法，将物体检测的特征迁移到语义分割任务中，进而实现了对VG数据集中3000类物体的语义分割。这个权值传递函数便是[Mask^X R-CNN](#)的精华所在。

Mask^X R-CNN 详解

1. 权值迁移函数： \mathcal{T}

Mask^X R-CNN 基于 Mask R-CNN (图1)。Mask R-CNN 通过向 Faster R-CNN[57] 中添加了一路 FPN 的分支来达到同时进行语义分割和目标检测的目的。在 RPN 之后，FPN 和 Fast R-CNN[61] 是完全独立的两个模块，此时若直接采用数据集 C 分别训练两个分支的话是行得通的，其实这也正是 YOLO9000 的训练方式。

图1：Mask R-CNN 流程



但是 Mask^X R-CNN 不会这么简单就结束的，它在检测分支(Fast R-CNN)和分割分支中间加了一条叫做权值迁移函数的线路，用于将检测的信息传到分割任务中，如图2所示。

图2：Mask^X R-CNN 流程

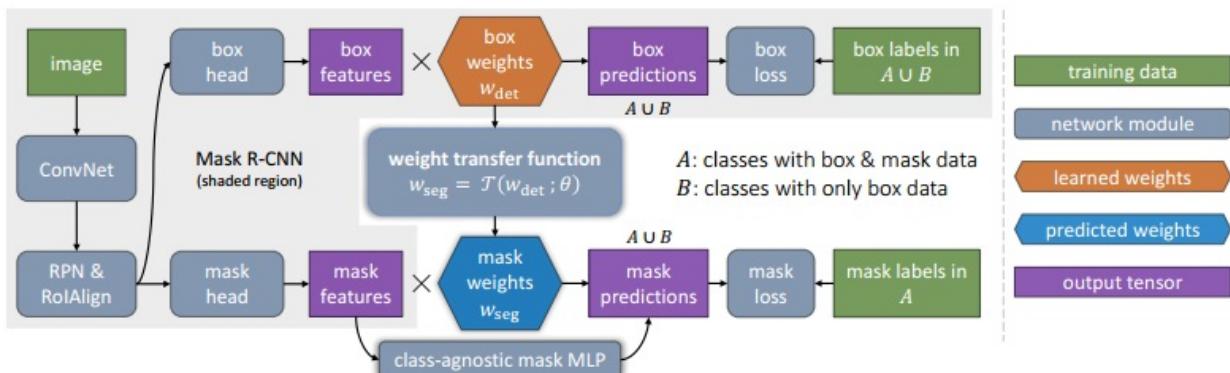


图2的整个框架是搭建在 Mask R-CNN 之上的，除了最重要的权值迁移函数之外，还有几点需要强调一下：

1. 着重强调， T 的输入参数是权值（图2中的两个六边形），而非 Feature Map；
2. 虽然 Mask R-CNN 中解耦了检测和分割任务，但是权值迁移函数 T 是类别无关的；

对于一个类别 c ， w_{det}^c 表示检测任务的权值， w_{seg}^c 表示分割任务的权值。权值迁移函数将 w_{det}^c 看做自变量， w_{seg}^c 看做因变量，学习两个权值的映射函数 \mathcal{T} ：

$$W_{seg}^c = \mathcal{T}(w_{det}^c; \theta)$$

其中 θ 的是类别无关的，可学习的参数。 \mathcal{T} 可以使用一个小型的MLP。 w_{det}^c 可以使分类的权值 w_{cls}^c ，bounding box的预测权值 w_{reg}^c 或是两者拼接到一起 $[w_{cls}^c, w_{reg}^c]$ 。

2. Mask^X R-CNN 的训练

含有掩码标签和检测标签的COCO数据集定义为 A ，只含有检测标签的VG数据集定义为 B ，则所有的数据集 C 便是 A 和 B 的并集： $C = A \cup B$ 。图2显示Mask^X R-CNN的损失函数由Fast R-CNN的检测任务和RPN的分类任务组成。当训练检测任务时，使用数据集C；当训练分割任务时，仅使用包括分割标签的COCO数据集，即 A 。

训练Mask^X R-CNN时，有两种类型：

1. 多阶段训练：首先使用数据集 C 训练Faster R-CNN，得到 w_{det}^c ；然后固定 w_{det}^c 和卷积部分，在使用 A 训练 \mathcal{T} 和分割任务的卷积部分。在这里 w_{det}^c 可以看做分割任务的特征向量。在Fast R-CNN中就指出多阶段训练的模型不如端到端训练的效果好；
2. 端到端联合训练：理论上是可以直接在 C 上训练检测任务，在 A 上训练分割任务，但是这会使模型偏向于数据集 A ，这个问题在论文中叫做discrepancy。为了解决这个问题，Mask^X R-CNN在反向计算mask损失函数时停止 w_{det}^c 相关的梯度更新，只更新权值迁移函数中的 θ 。

总结

截止到目前，尚无高质量的Mask^X R-CNN源码公布，有很多论文中没有涉及的细节尚无处可考，等作者开源之后会继续补充该文章。

仿照YOLO9000的思路，Mask^X R-CNN使用半监督学习的方式将分割类别扩大到3000类。采用WordTree将分类数据添加到Mask R-CNN中，将分割类别扩大到ImageNet中的类别应该是未来一个不错的研究方向，推测应该很快就有相关进展发表。

然后一个更精确，更快的的权值迁移函数也是一个非常有研究前景的一个方向，毕竟现在计算机视觉方向的趋势是去掉低效的全连接。

SNIPER: Efficient Multi-Scale Training

前言

图像金字塔是传统的提升物体检测精度的策略之一，其能提升精度的一个原因是尺寸多样性的引入。但是图像金字塔也有一个非常严重的缺点：即增加了模型的计算量，一个经过3个尺度放大（ $1x, 2x, 3x$ ）的图像金子塔要多处理14倍的像素点。

SNIPER（Scale Normalization for Image Pyramid with Efficient Resampling）[34]的提出动机便是解决图像金字塔计算量大的问题，图像金字塔存在一个固有的问题：对于一张放大之后的高分辨率的图片，其物体也随之放大，有时甚至待检测物体的尺寸超过了特征向量的感受野，这时候对大尺寸物体的检测便很难做到精确，因为这时候已经没有合适的特征向量了；对于一个缩小了若干倍的图像中的一个小尺寸物体，由于网络结构中降采样的存在，此时也很难找到合适的特征向量用于该物体的检测。因此作者在之前的SNIP[33]论文中便提出了高分辨率图像中的大尺寸物体和低分辨率图像中的小尺寸物体时应该忽略的。

为了阐述SNIPER的提出动机，作者用很大的篇幅来对比R-CNN和Fast R-CNN的异同，一个重要的观点就是R-CNN具有尺度不变性，而Fast R-CNN不具有该特征。R-CNN先通过Selective Search选取候选区域，然后无论候选区域的尺寸是多少都会将其归一化到 224×224 的尺寸，该策略虽然备受诟病，但是它却保证了R-CNN具有尺度不变性的特征。Fast R-CNN将resize的部分移到了使用了卷积之后，也就是使用RoI池化产生长度固定的特征向量，也就是说Fast R-CNN是将原始图像作为输入，无论里面的尺寸大小，均使用相同的卷积核计算特征向量。但是这样做真的合理吗？不同尺寸的待检测物体使用相同的卷积核来训练真的好吗？答案当然是否定的。

SNIPER策略最重要的贡献是提出了尺寸固定（ 512×512 ）的chips的东西，chips是图像金字塔中某个尺度图像的一个子图。chips有正负之分，其中正chip中包含了很多尺寸合适的标注样本，而负chip则不包含标注样本或者如第二段中所阐述的包含的样本不适合这个尺度的图像。

需要注意的是SNIPER只是一个采样策略，是对图像金字塔的替代。在论文中作者将SNIPER应用到了Faster R-CNN和Mask R-CNN中，同理，SNIPER也几乎可以应用到其它任何检测甚至识别算法中。

作者已经开源了基于MXNet的[源码](#)，我们在后面的分析中会结合源码进行讲解。

1. SNIPER 详解

作为一个采样策略，SNIPER的输入数据是原始的数据集，输出的是在图像上采样得到的子图（chips），如图1的虚线部分所示，而这些chips会直接作为s。当然，chips上的物体的Ground Truth也需要针对性的修改。那么这些chips是怎么计算的呢，下面我们详细分析之。

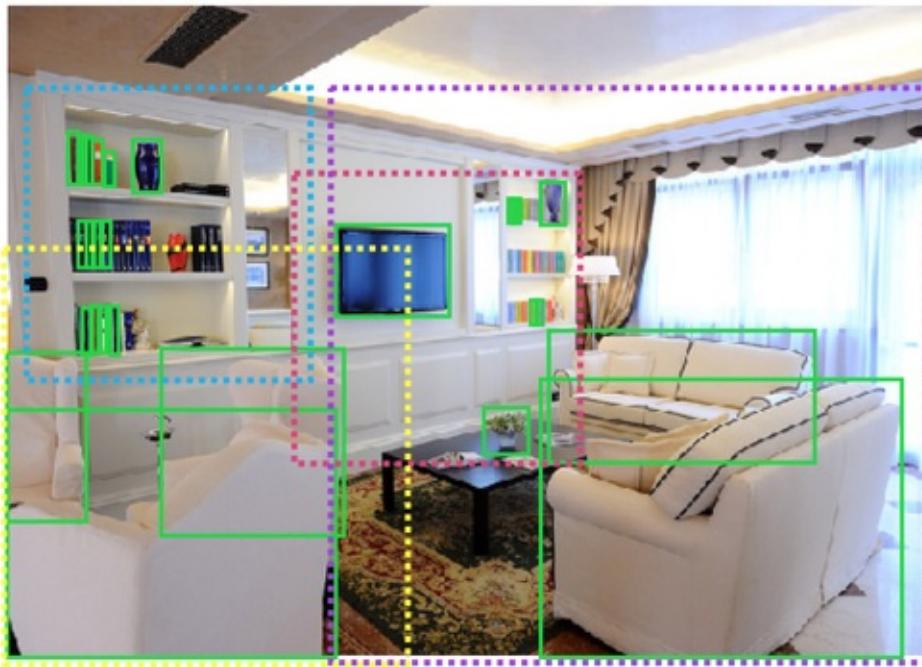


图1：SNIPER的chips示意图

1.1 Chips生成

按照图像金字塔的思想，一个原始输入图片会通过构建多个尺度 $\{s_1, s_2, \dots, s_n\}$ 的形式图像金字塔，源码中使用的是3个尺度(3.0, 1.667, 512.0/ms)，分别表示把图像放大3倍，1.667倍以及最大边长固定为512。

在图像金字塔的某个的图像上（假设大小为 $W_i \times H_i$ ）通过步长为32的滑窗的方法得到约

$\frac{W_i}{32} \times \frac{H_i}{32}$ 个 512×512 的chip。

那么整个图像金字塔的chips的总数约为：

$$\sum_i^n \frac{W_i}{32} \times \frac{H_i}{32}$$

1.2 正chips

对于1.1部分提到的 n 个尺度，图像的Ground Truth也会进行对应的放大或者缩小。对于每个尺度，它都有更合适的检测框。论文中给出的判断标准是对于三个尺度，它们理想的检测区域的面积介于 $\mathcal{R}^i = [r_{max}^i, r_{min}^i]$ 之间（用换算之前的Ground Truth计算面积），源码中给出三个尺度对应的检测面积的区域依次是 $(0, 80^2), (32^2, 150^2), (120^2, inf)$ 。

如果一个Ground Truth完全位于一个chip内，那么我们就说这个Ground Truth被这个chip包围（Covered）。在实际采样中，处于计算量和样本平衡的角度考虑，所有的chip不可能都被采样到。在论文中，作者使用了贪心的策略根据chip包围的Ground Truth的数量的多少，从每个尺度中抽取了前 K 个chip作为正chip，记做 C_{pos}^i

最后在放大或者缩小的图像中，将采样的chips裁剪出来，得到一系列大小固定（ 512×512 ）的子图，这些子图将作为后续检测算法的训练样本。由于得到的chip的尺寸比较小，且大多数没用的背景都都被忽略了（思想类似于Attention），这对训练的速度的提升是非常有帮助的，同时每个chip只包含合适尺寸的检测物体，这使得模型更加容易收敛。

因为多尺度的chips之间是互相覆盖的，所以可以保证了一个Ground Truth至少被一个Chip采样得到，一个Ground Truth既可以被不同尺度的chips所共同包围，也可以被相同尺度的不同chips所共同包围。

图2中得到的chips便是从图1中的虚线部分裁剪出来的。如图2所示，绿色的Ground Truth代表的是和该chips匹配的待检测物体，而红色的Ground Truth由于面积不在范围内，因此不会被标注出来，在检测的时候等同的看做背景区域。

图2：由图1得到的*chips*

需要注意一点，源码中生成*chips*的方式并不是之前所说的先生成图像金字塔，再从图像金字塔中裁剪出 512×512 的*chips*。源码中采用的方式是从原图中采样出等比例的*chips*，再将其resize到 512×512 。这两种策略是等价的，但是第二种无疑实现起来更为简单。

1.3 负 *chips*

如果只用正*chips*作为样本进行训练，模型的假正利率往往很高（不包含任何物体的*chip*判断为包含包含物体的*chip*）。具体原因是因为参与模型训练的数据都是包围着Ground Truth的*chips*，而在测试的时候（第2节详细介绍SNIPER的测试部分），输入的是整张图像的图像金字塔，这时候必然包含不包围任何Ground Truth的背景区域，也就是说训练集和测试集的分布是不一样的。

为了弥补训练集合测试集之间的分布差距，SNIPER提出了从图像金字塔中采样出一批 512×512 负 chips，它们将作为训练数据共同参与模型的训练。所谓负 chips，是指不包含 Ground Truth 或者包含的 Ground Truth 比较少的 chip。

论文中给出的策略是首先只使用正 chip 训练一个只有几个 Epoch 的弱 RPN。在这里我们对 RPN 的精度并没有特别高的要求，因为它只是我们用来选择 chip 的一个工具，对最终的结果影响十分微弱。尽管这个 RPN 检测能力很弱，但是其并不是随机初始化的一个模型，它得到的检测框还是有一定的置信度的。所以策略的第二步是根据弱 RPN 的检测结果选择那些“假正”的样本。详细的说，首先去掉 C_{pos}^i 中的正 chips，然后根据弱 RPN 的检测结果，从每个尺度选择至少包含 M 个候选区域的 chips 组成负 chips 池，最后在训练的时候从中随机选择固定数量的组成训练的负样本，表示为 $\bigcup_{i=1}^n C_{neg}^i$ 。

如图3所示，上排绿色部分表示 Ground Truth，下排红色区域表示弱 RPN 预测的候选区域，橙色区域表示根据弱 RPN 得到的负 chips。



图3：SNIPER中的负 chips

1.4 SNIPER的训练

通过上面的分析，我们可以根据 Ground Truth 和弱 RPN 得到一批正 chips 和一批负 chips，这批 chips 将作为训练样本直接输入给检测算法。SNIPER 采用了 Faster R-CNN 作为这些 chips 的检测算法框架，并且 SNIPER 后面模型训练的细节也基本和 Faster R-CNN 相同，除了一点，SNIPER 接的 Faster R-CNN 的 RPN 和 Fast R-CNN 使用了不同的标签系统。

Faster R-CNN [60] 系列论文中我们讲过，Faster R-CNN 是由 RPN 和 Fast R-CNN 组成的多任务模型，在 SNIPER 中，两个任务会使用两个不同的标签，首先训练 RPN 时，chips 中的待检测物体的 Ground Truth 并不会受范围 R 的限制。但是再根据 RPN 提取的候选区域训练 Fast R-CNN 时，在范围 R 之外的候选区域并不会参与 Fast R-CNN 的训练。这么做的原因作者没有指

出，猜测是RPN需要检测的候选区域覆盖范围更广，因此需要更多的，范围更大的Ground Truth。而Fast R-CNN需要检测的更准确，因此把这个任务交给了能提取到更合适的特征向量的对应尺度。

2. SNIPER的测试过程

SNIPER的测试过程有一个致命的缺点，它必须要求输入的图像是图像金字塔，因为它参与训练的Fast R-CNN的chip在范围 R 之外已经过滤掉了，因此模型并不擅长检测不在这个范围内的物体，也就是如果只使用单尺度的话，那些太大或者太小的物体都很难被检测到。

SNIPER使用的图像金字塔的尺度依次是(480, 512)(800, 1280)(1400, 2000)，其中第一个值表示resize后短边的大小，但是当长边大于第二个值时，应该将长边固定到第二个值，第一个值随意。

在图像金字塔提取完检测框之后，使用soft-NMS得到最终的候选区域。

3. 总结

小物体检测一直是困扰物体检测领域的一个重要难题，传统的图像金字塔式解决该问题的一个常见的传统策略，但是速度太慢，SNIPER的提出便是动机便是解决图像金字塔的速度问题。

需要注意SNIPER并不是一个检测算法，而是对输入图像的一个采样策略，其采样的结果(chips)将作为输入输入到物体检测算法中。

算法虽然使用了RPN，但是并不是离开了RPN就无法工作了，RPN提供了一个提取假正利率的功能，这个可以通过Selective Search或者Edge Box近似替代。

另外，SNIPER仅仅是对训练速度的提升，往往更重要的检测速度并没有提升，反而是模型必须依赖图像金字塔，这反而降低了模型的通用性。

最后，作者开源的源码和论文出入较大，读起来比较费劲，等之后有时间的话再详细学习这份源码。

第五章：光学字符识别

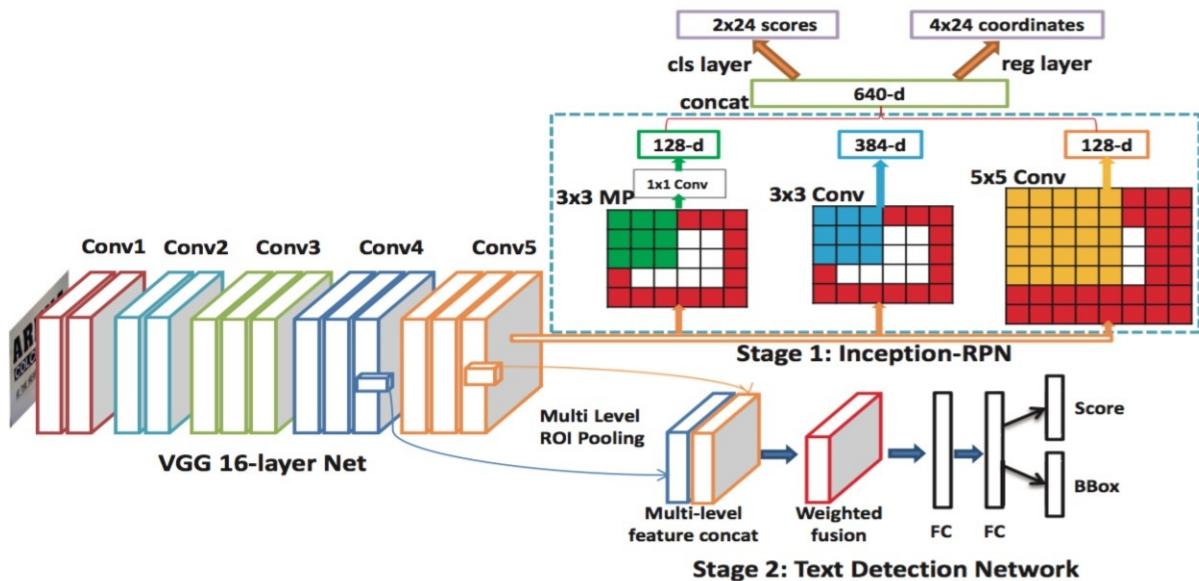
DeepText: A Unified Framework for Text Proposal Generation and Text Detection in Natural Images

前言

16年那段时间的文字检测的文章，多少都和当年火极一时的Faster R-CNN[2]有关，DeepText（图1）也不例外，整体上依然是Faster R-CNN的框架，并在其基础上做了如下优化：

1. **Inception-RPN**：将RPN的 3×3 卷积划窗换成了基于Inception[3]的划窗。这点也是这篇文章的亮点；
2. **ATC**：将类别扩展为‘文本区域’，‘模糊区域’与‘背景区域’；
3. **MLRP**：使用了多尺度的特征，ROI提供的按Grid的池化的方式正好融合不同尺寸的Feature Map。
4. **IBBV**：使用多个Iteration的bounding boxes的集合使用NMS

图1：DeepText网络结构图



在阅读本文前，一定要先搞清楚Faster R-CNN，本文只会对DeepText对Faster R-CNN的改进进行说明，相同部分不再重复。

1. DeepText详解

DeepText的结构如Faster R-CNN如出一辙：首先特征层使用的是VGG-16，其次算法均由用于提取候选区域的RPN和用于物体检测的Fast R-CNN。

下面我们将对DeepText优化的四点进行讲解。

1.1 Inception-RPN

首先DeepText使用了GoogLeNet提出的Inception结构代替Faster R-CNN中使用的 3×3 卷积在Conv5_3上进行滑窗。Inception的作用参照GoogLeNet中的讲解。

DeepText的Inception由3路不同的卷积构成：

- padding=1的 3×3 的Max Pooling后接128个用于降维的 1×1 卷积；
- 384个padding=1的 3×3 卷积；
- 128个padding=2的 5×5 卷积。

由于上面的Inception的3路卷积并不会改变Feature Map的尺寸，经过Concatenate操作后，Feature Map的个数变成了 $128 + 384 + 128 = 640$ 。

针对场景文字检测中Ground Truth的特点，DeepText使用了和Faster R-CNN不同的锚点： $(32, 48, 64, 80)$ 四个尺寸及 $(0.2, 0.5, 0.8, 1.0, 1.2, 1.5)$ 六种比例共 $4 \times 6 = 24$ 个锚点。

DeepText的采样阈值也和Faster R-CNN不同：当 $\text{IoU} > 0.5$ 时，锚点为正； $\text{IoU} < 0.3$ ，锚点为负。

Inception-RPN使用了阈值为0.7的NMS过滤锚点，最终得到的候选区域是top-2000的样本。

1.2 Ambiguous Text Classification (ATC)

DeepText将样本分成3类：

- Text: $\text{IoU} > 0.5$;
- Ambiguous: $0.2 < \text{IoU} < 0.5$;
- Background: $\text{IoU} < 0.2$.

这样做的目的是让模型在训练过程中见过所有IoU的样本，该方法对于提高模型的召回率作用非常明显。

1.3 Multi Layer ROI Pooling (MLRP)

DeepText使用了VGG-16的Conv4_3和Conv5_3的多尺度特征，使用基于Grid的ROI Pooling将两个不同尺寸的Feature Map变成 $7 \times 7 \times 512$ 的大小，通过 1×1 卷积将Concatenate后的1024维的Feature Map降维到512维，如图1所示。

1.4 Iterative Bounding Box Voting (IBBV)

在训练过程中，每个Iteration会预测一组检测框： $D_c^t = \{B_{i,c}^t, S_{i,c}^t\}_{i=1}^{N_{c,t}}$ ，其中 $t = 1, 2, \dots, T$ 表示训练阶段， $N_{c,t}$ 表示类别 c 的检测框， B 和 S 分别表示检测框和置信度。**NMS**合并的是每个训练阶段的并集： $D_c = \bigcup_{t=1}^T U_c^t$ 。**NMS**使用的合并阈值是0.3。

在**IBBV**之后，**DeepText**接了一个过滤器用于过滤多余的检测框，过滤器的具体内容不详，后续待补。

总结

结合当时的研究现状，**DeepText**结合了当时**state-of-the-art**的**Faster R-CNN**，**Inception**设计了该算法。算法本身的技术性和创新性并不是很强，但是其设计的**ATC**和**MLRP**均在后面的物体检测算法中多次使用，而**IBBV**也在实际场景中非常值得测试。

Detecting Text in Natural Image with Connectionist Text Proposal Network

tags: CTPN, Faster-RCNN, LSTM

CTPN[31]和Faster R-CNN[60]出自同系，根据文本区域的特点做了专门的调整，一个重要的地方是RNN的引入，笔者在实现CTPN的时候也是直接在Faster-RCNN基础上改的。理解了Faster R-CNN之后，CTPN理解的难度也不大，下面开始分析这篇论文。

作者提供的CTPN demo源码：<https://github.com/tianzhi0549/CTPN>

基于TensorFlow的CTPN开源代码：<https://github.com/eragonruan/text-detection-ctpn>

简介

传统的文字检测是一个自底向上的过程，总体上的处理方法分成连通域和滑动窗口两个方向。基于连通域的方法是先通过快速滤波器得到文字的像素点，再根据图像的低维特征（颜色，纹理等）贪心的构成线或者候选字符。基于滑动窗口的方法是通过在图像上通过多尺度的滑窗，根据图像的人工设计的特征（HOG，SIFT等），使用预先训练好的分类器判断是不是文本区域。但是这两个途径在健壮性和可信性上做的并不好，而且滑窗是一个非常耗时的过程。

卷积网络在2012年的图像分类上取得了巨大的成功，在2015年Faster R-CNN在物体检测上提供了非常好的算法框架。所以用深度学习的思想解决场景文字检测自然而然的成为研究热点。对比发现，场景文字检测和物体检测存在两个显著的不同之处

1. 场景文字检测有明显的边界，例如Wolf准则 [30]，而物体检测的边界要求较松，一般IoU为0.7便可以判断为检测正确；
2. 场景文字检测有明显的序列特征，而物体检测没有这些特征；
3. 和物体检测相比，场景文字检测含有更多的小尺寸的物体。

针对以上特点，CTPN做了如下优化：

1. 在CTPN中使用更符合场景文字检测特点的锚点；
2. 针对锚点的特征使用新的损失函数；
3. RNN（双向LSTM）的引入用于处理场景文字检测中存在的序列特征；
4. Side-refinement的引入进一步优化文字区域。

算法详解

1. 算法流程

CTPN的流程和Faster R-CNN的RPN网络类似，首先使用VGG-16提取特征，在conv5进行 3×3 ，步长为1的滑窗。设conv5的尺寸是 $W \times H$ ，这样在conv5的同一行，我们可以得到 W 个256维的特征向量。将同一行的特征向量输入一个双向LSTM中，在双向LSTM后接一个512维的全连接后便是CTPN的3个任务的多任务损失，结构如图1。任务1的输出是 $2 \times k$ ，用于预测候选区域的起始y坐标和高度h；任务2是用来对前景和背景两个任务的分类评分；任务3是 k 个输出的side-refinement的偏移（offset）预测。在CTPN中，任务1和任务2是完全并行的任务，而任务3要用到任务1，2的结果，所以理论上任务3和其他两个是串行的任务关系，但三者放在同一个损失和函数中共同训练，也就是我们在Faster R-CNN中介绍的近似联合训练。

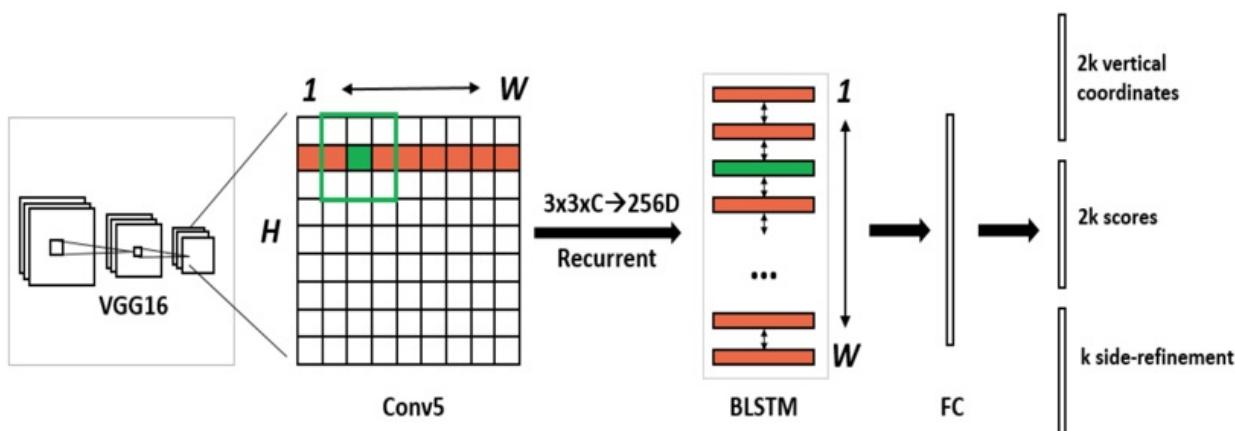


图1：CTPN的结构

2. 数据准备

和RPN的要求一样，CTPN输入图像的尺寸无硬性要求，只是为了保证特征提取的有效性，在保证图片比例不变的情况下，CTPN将输入图片的resize到600，且保证长边不大于1000。

```
def resize_im(im, scale, max_scale=None):
    f=float(scale)/min(im.shape[0], im.shape[1])
    if max_scale!=None and f*max(im.shape[0], im.shape[1])>max_scale:
        f=float(max_scale)/max(im.shape[0], im.shape[1])
    return cv2.resize(im, (0, 0), fx=f, fy=f), f
```

3. CTPN的锚点机制

作者通过分析RPN在场景文字检测的实验发现RPN的效果并不是特别理想，尤其是在定位文本区域的横坐标上存在很大的误差。因为在一串文本中，在不考虑语义的情况下，每个字符都是一个独立的个体，这使得文字区域的边界是很难确定的。显然，文本区域检测和物体检测最大的区别是文本区域是一个序列，如图2。如何我们能根据文本的序列特征捕捉到文本区域的边界信息，应该能够对文本区域的边界识别能够很好的预测。



图2：文本区域的序列特征。

而在目前的神经网络中，RNN在处理序列数据上占有垄断性的优势地位。在RNN的训练过程中，数据是以时间片为单位输入到模型中的。所以，如何将文本区域变成可以序列化输入的顺序成为了CTPN一个重要的要求。如图2所展示的，每一个蓝色矩形是一个锚点，那么一个文本区域便是由一系列宽度固定，紧密相连的锚点构成。所以，CTPN有如下的锚点设计机制：

由于CTPN是使用的VGG-16进行特征提取，VGG-16经过4次max pooling的降采样，得到的_feature_stride=16，_feature_stride体现在在conv5上步长为1的滑窗相当于在输入图像上步长为16的滑窗。所以，根据VGG-16的网络结构，CTPN的锚点宽度w必须为16¹。对于一个输入序列中的所有锚点，如果我们能够判断出锚点的正负，把这一排正锚点连在一起便构成了文本区域，所以，锚点的起始坐标x也不用预测。所以在CTPN中，网络只需要预测锚点的起始y坐标以及锚点的高度h即可。

在RPN网络中，一个特征向量对应的多个尺寸和比例的锚点，同样的，CTPN也对同一个特征向量设计了10个锚点。在CTPN中，锚点的高度依次是[11,16,23,33,48,68,97,139,198,283]，即高度每次除以0.7。根据笔者的经验，根据你的CTPN的应用场景，比如要做一些文本区域较小的检测，这时候你可能需要设计更小的锚点。

4. CTPN 中的RNN

我们多次强调场景文字检测一个重要的不同是文本区域具有序列特征。在上面一段，我们已经可以根据锚点构造序列化的数据。通过在W*H的conv5层进行步长为的滑窗，每一次横向滑动得到的便是W个长度为256的特征向量 X_t 。设RNN的隐层节点是 H_t ，则RNN模型可以表示为

$$H_t = \varphi(H_{t-1}, X_t), t = 1, 2, \dots, W$$

其中 φ 是非线性的激活函数。隐层节点的数量是128，RNN使用的是双向的LSTM，因此通过双向LSTM得到的特征向量是256维的。

```

layer {
  name: "lstm"
  type: "Lstm"
  bottom: "lstm_input"
  top: "lstm"
  lstm_param {
    num_output: 128
    weight_filler {
      type: "gaussian"
      std: 0.01
    }
    bias_filler {
      type: "constant"
    }
    clipping_threshold: 1
  }
}

...
layer {
  name: "rlstm"
  type: "Lstm"
  bottom: "rlstm_input"
  top: "rlstm-output"
  lstm_param {
    num_output: 128
  }
}

...
# merge lstm and r lstm
layer {
  name: "merge_lstm_rlstm"
  type: "Concat"
  bottom: "lstm"
  bottom: "rlstm"
  top: "merge_lstm_rlstm"
  concat_param {
    axis: 2
  }
}

```

5. side-refinement

将side-refinement从CTPN独立出来似乎更好理解，side-refinement对于CTPN的其它部分相当于Faster R-CNN中的RPN对于Fast R-CNN。不同之处在于side-refinement根据CTPN预测的锚点信息得到文本行，从中选择边界锚点进行位移优化，而Fast R-CNN优化的是根据RPN

的输出通过NMS得到的候选区域。所以，side-refinement一个重要的步骤是如何根据锚点信息构造文本行。

5.1 文本行的构造

通过CTPN可以得到候选区域的得分，如果判定为文本区域的得分大于阈值 θ_1 ，则该区域用来构造文本行。文本行是由一系列大于0.7的候选区域的邻居对构成的，如果区域 B_j 是区域 B_i 的邻居对，需要满足如下条件：

1. B_j 是距离 B_i 最近的正文本区域；
2. B_j 和 B_i 的距离小于 θ_2 个像素值；
3. B_i 和 B_j 的竖直方向的重合率大于 θ_3 。

在源码提供的配置文件中， $\theta_1 = 0.7, \theta_2 = 50, \theta_3 = 0.7$ 。文本区域是由一系列邻居对构成的。

```
def get_successions(self, index):
    box = self.text_proposals[index]
    results = []
    for left in range(int(box[0]) + 1, min(int(box[0]) + cfg.MAX_HORIZONTAL_GAP + 1, self.im_size[1])):
        adj_box_indices = self.boxes_table[left]
        for adj_box_index in adj_box_indices:
            if self.meet_v_iou(adj_box_index, index):
                results.append(adj_box_index)
    if len(results) != 0:
        return results
    return results
```

5.2 side-refinement的损失函数

构造完文本行后，我们根据文本行的左端和右端两个锚点的特征向量计算文本行的相对位移(o)：

$$o = (x_{side} - c_x^a) / w_a$$

$$o^* = (x_{side}^* - c_x^a) / w_a$$

其中 x_{side} 便是由CTPN构造的文本行的左侧和右侧两个锚点的x坐标，即文本行的起始坐标和结尾坐标。所以 x_{side}^* ，便是对应的ground truth的坐标， c_x^a 是锚点的中心点坐标， w_a 是锚点的宽度，所以是16。设 k 是锚点的下标，则 side-refinement 使用的损失函数是 smooth L1 函数。

6. CTPN的损失函数

CTPN使用的是Faster R-CNN的近似联合训练，即将分类，预测，side-refinement作为一个多任务的模型，这些任务的损失函数共同决定模型的调整方向。

6.1 文本区域得分损失 L_s^{cl}

分类损失函数 $L_s^{cl}(s_i, s_i^*)$ 是 softmax 损失函数，其中 $s_i^* = \{0, 1\}$ 是 ground truth，即如果锚点为正锚点（前景）， $s_i^* = 1$ ，否则 $s_i^* = 0$ 。在 tf 的源码中 (lib/rpn_msr/anchor_target_layer_tf.py)，一个锚点是正锚点的条件如下：

1. 每个位置上的9个anchor中 overlap 最大的认为是前景；
2. overlap 大于 0.7 的认为是前景

如果 overlap 小于 0.3，则被判定为背景。在源码中参数 RPN_CLOBBER_POSITIVES 为 true 则表示如果一个样本的 overlap 小于 0.3，且同时满足正样本的条件 1，则该样本被判定为负样本。

s_i 是预测锚点 i 为前景的概率。

6.2 纵坐标损失 L_v^{re}

纵坐标的损失函数 $L_v^{re}(v_j, v_j^*)$ 使用的是 smooth L1 损失函数， v_j 和 $x = y$ 使用的是相对位移，表示如下

$$v_c = (c_y - c_y^a)/h^a, v_h = \log(h/h^a)$$

$$v_c^* = (c_y^* - c_y^a)/h^a, v_h^* = \log(h^*/h^a)$$

$v = \{v_c, v_h\}$, $v^* = \{v_c^*, v_h^*\}$ 分别是预测的坐标和 ground truth 的坐标

综上，CTPN 的损失函数表示为

$$L(s_i, v_j, o_k) = \frac{1}{N_s} \sum_i L_s^{cl}(s_i, s_i^*) + \frac{\lambda_1}{N_v} \sum_j L_v^{re}(v_j, v_j^*) + \frac{\lambda_2}{N_o} L_o^{re}(o_k, o_k^*)$$

λ_1 ， λ_2 是各任务的权重系数， N_s ， N_v ， N_o 是归一化参数，表示对应任务的样本数量。

7. CTPN的训练细节

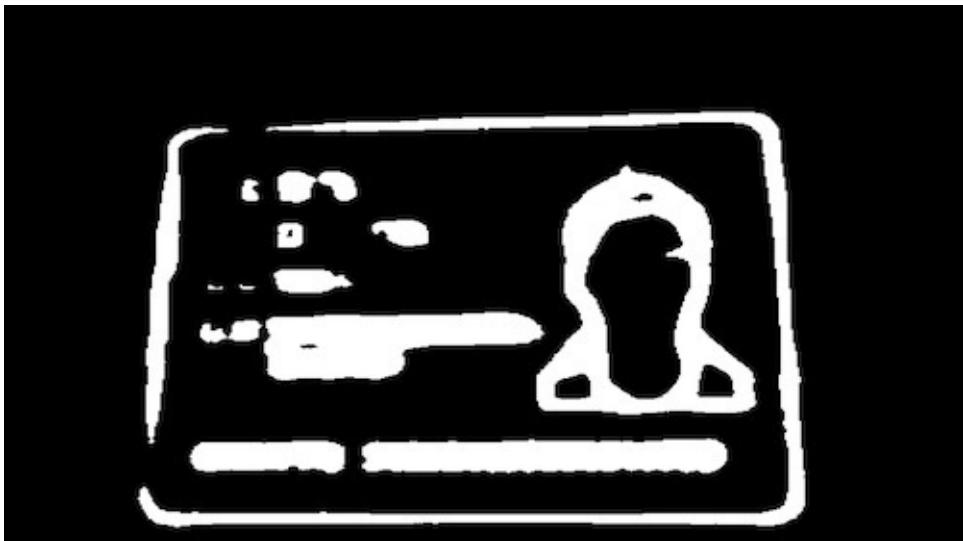
每个minibatch同样采用“Image-centric”的采样方法，每次随机取一张图片，然后在这张图片中采样128个样本，并尽量保证正负样本数量的均衡。卷积层使用的是Image-Net上无监督训练得到的结果，权值初始化使用的是均值为0，标准差为0.01的高斯分布。SGD的参数中，遗忘因子是0.9，权值衰减系数是0.0006。前16k次迭代的学习率是0.001，后4k次迭代的学习率是0.0001。

Scene Text Detection via Holistic, Multi-Channel Prediction

前言

本文是在边缘检测经典算法HED[28]之上的扩展，在这篇论文中我们讲过HED算法可以无缝转移到语义分割场景中。而这篇论文正是将场景文字检测任务转换成语义分割任务来实现HED用于文字检测的。图1是HED在身份证上进行边缘检测得到的掩码图，从图1中我们可以看出HED在文字检测场景中也是有一定效果的。

图1：HED在身份证上得到的掩码图



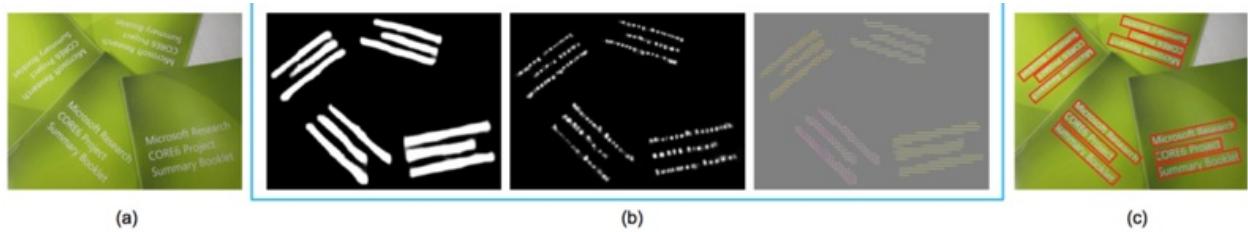
HED之所以能用于场景文字检测一个重要的原因是文字区域具有很强的边缘特征。

论文的题目为 Holistic, Multi-Channel Prediction (HMCP) [29]，其中 Holistic 表示算法基于 HED，Multi-Channel 表示该算法使用多个 Channel 的标签训练模型。也就是为了提升 HED 用于文字检测的精度，这篇文章做的改进是将模型任务由单任务模型变成是由文本行分割，字符分割和字符间连接角度构成的多任务系统。由于 HMCP 采用的是语义分割的形式，所以其检测框可以扩展到多边形或者是带旋转角度的四边形，这也更符合具有严重仿射变换的真实场景。

1. HMCP详解

HMCP的流程如图2：(a)是输入图像，(b)是预测的三个mask，分别是文本行掩码，字符掩码和字符间连接角度掩码，(c)便是根据(b)的三个掩码得到的检测框。

图2：HMCP流程



那么问题来了：

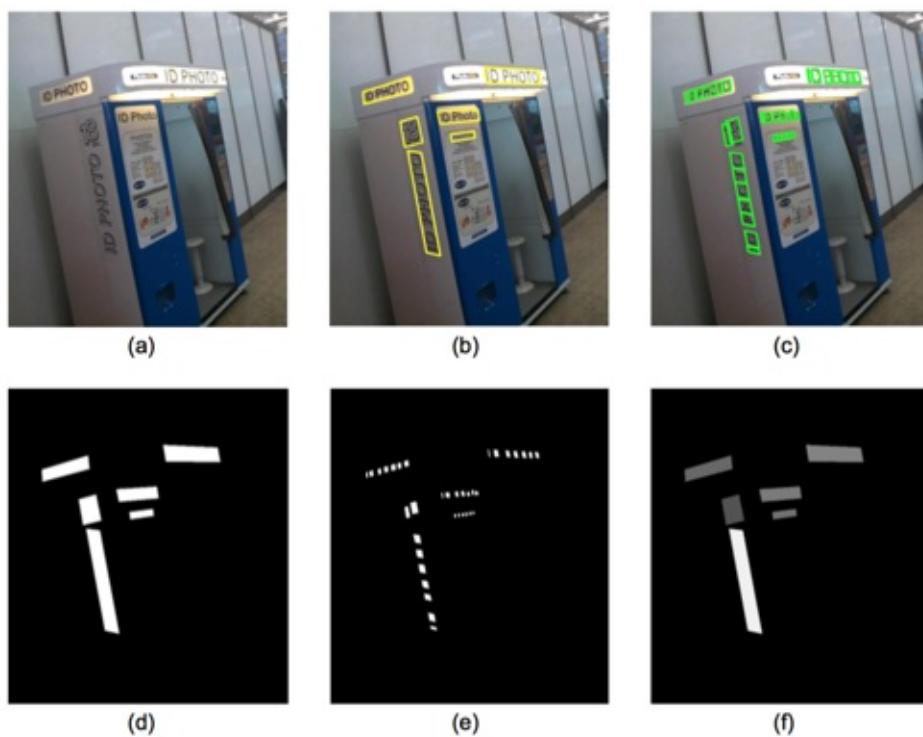
1. 如何构建三个channel掩码的标签值；
2. 如何根据预测的掩码构建文本行。

1.1 HMCP的标签值

在文本检测数据集中，常见的标签类型有QUAD和RBOX两种形式。其中QUAD是指标签值包含四个点 $\mathbf{G} = \{(x_i, y_i) | i \in \{1, 2, 3, 4\}\}$ ，由这四个点构成的不规则四边形（Quadrangle）便是文本区域。QBOX则是由一个规则矩形 \mathbf{R} 和其旋转角度 θ 构成，即 $\mathbf{G} = \{\mathbf{R}, \theta\}$ 。QUAD和RBOX是可以相互转换的。

HMCP的数据的Ground Truth分别包含基于文本行和基于字符的标签构成(QUAD或RBOX)，如图3.(b)和图3.(c)。数据集中只有基于文本行的Ground Truth，其基于单词的Ground Truth是通过SWT[27]得到的。(d)是基于文本行Ground Truth得到的二进制掩码图，文本区域的值为1，非文本区域的值为0。(e)是基于单词的二进制掩码图，掩码也是0或者1。由于字符间的距离往往比较小，为了能区分不同字符之间的掩码，正样本掩码的尺寸被压缩到了Ground Truth的一半。(f)是字符间连接角度掩码，其值为 $[-\pi/2, \pi/2]$ ，然后再通过归一化映射到 $[0, 1]$ 之间。角度的值是由RBOX形式的Ground Truth得到的值。

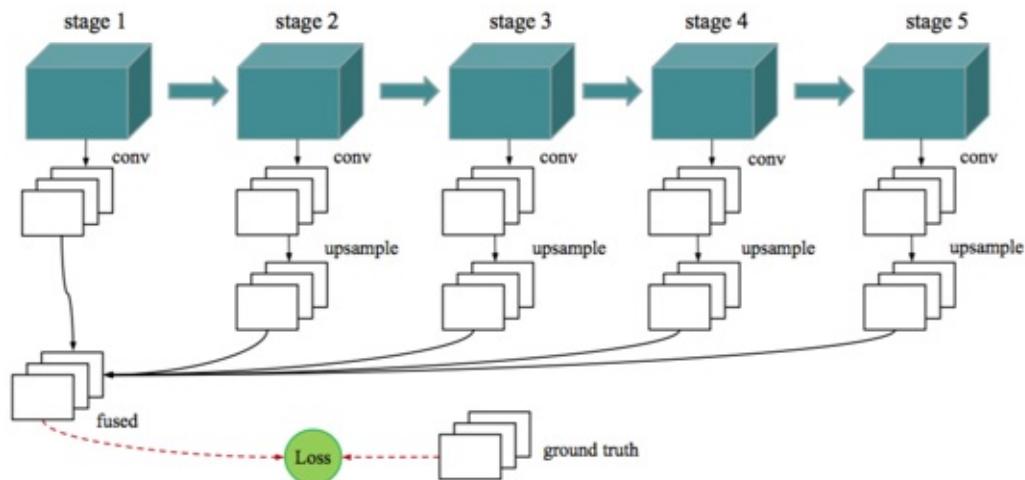
图3：HMCP的Ground Truth以及三种Mask



1.2 HMCP的骨干网络

HMCP的骨干网络继承自HED，如图4所示。HMCP的主干网络使用的是VGG-16，在每个block降采样之前通过反卷积得到和输入图像大小相同的Feature Map，最后通过fuse层将5个side branch的Feature Map拼接起来并得到预测值。HMCP和HED的不同之处是HMCP的输出节点有三个任务。

图4：HMCP的骨干网络



1.3 HMCP的损失函数

1.3.1 训练

设HMCP的训练集为 $S = \{(X_n, y_n), n = 1, \dots, N\}$ ，其中 N 是样本的数量。标签 Y_n 由三个掩码图构成，即 $Y_n = \{R_n, C_n, \Theta_n\}$ ，其中 $R_n = \{r_j^{(n)} \in \{0, 1\}, j = 1, \dots, |R_n|\}$ 表示文本区域的二进制掩码图， $C_n = \{c_j^{(n)} \in \{0, 1\}, j = 1, \dots, |C_n|\}$ 是字符的二进制掩码图， $\Theta_n = \{\theta_j^{(n)} \in \{0, 1\}, j = 1, \dots, |\Theta_n|\}$ 是相邻字符的连接角度。注意只有当 $r_j^{(n)} = 1$ 时 $\theta_j^{(n)}$ 才有效。

与HED不同的是HMCP的损失函数没有使用side branch，即损失函数仅由fuse层构成：

$$\mathcal{L} = \mathcal{L}_{\text{fuse}}(\mathbf{W}, \mathbf{w}, Y, \hat{Y})$$

其中 \mathbf{W} 为VGG-16部分的参数， \mathbf{w} 为fuse层部分的参数。 $\hat{Y} = \{\hat{R}, \hat{C}, \hat{\Theta}\}$ 是预测值：

$$\hat{Y} = \text{CNN}(X, \mathbf{W}, \mathbf{w})$$

$\mathcal{L}_{\text{fuse}}(\mathbf{W}, \mathbf{w}, Y, \hat{Y})$ 由三个子任务构成：

$$\mathcal{L}_{\text{fuse}}(\mathbf{W}, \mathbf{w}, Y, \hat{Y}) = \lambda_1 \Delta_r(\mathbf{W}, \mathbf{w}, R, \hat{R}) + \lambda_2 \Delta_c(\mathbf{W}, \mathbf{w}, C, \hat{C}) + \lambda_3 \Delta_o(\mathbf{W}, \mathbf{w}, \Theta, \hat{\Theta}, R)$$

其中 $\lambda_1 + \lambda_2 + \lambda_3 = 1$ 。 $\Delta_r(\mathbf{W}, \mathbf{w})$ 表示基于文本掩码的损失值， $\Delta_c(\mathbf{W}, \mathbf{w})$ 是基于字符掩码的损失值，两个均是使用HED采用过的类别平衡交叉熵损失函数：

$$\Delta_r(\mathbf{W}, \mathbf{w}, R, \hat{R}) = -\beta_R \sum_{j=1}^{|R|} R_j \log Pr(\hat{R}_j = 1; \mathbf{W}, \mathbf{w}) + (1 - \beta_R) \sum_{j=1}^{|R|} (1 - R_j) \log Pr(\hat{R}_j = 0; \mathbf{W}, \mathbf{w})$$

上式中的 β 为平衡因子 $\beta_R = \frac{|R_-|}{|R|}$ ， $|R_-|$ 为文本区域Ground Truth中负样本个数， $|R|$ 为所有样本的个数。

基于字符掩码的损失值与 $\Delta_r(\mathbf{W}, \mathbf{w}, R, \hat{R})$ 类似：

$$\Delta_c(\mathbf{W}, \mathbf{w}, C, \hat{C}) = -\beta_C \sum_{j=1}^{|C|} C_j \log Pr(\hat{C}_j = 1; \mathbf{W}, \mathbf{w}) + (1 - \beta_C) \sum_{j=1}^{|C|} (1 - C_j) \log Pr(\hat{C}_j = 0; \mathbf{W}, \mathbf{w})$$

$\Delta_o(\mathbf{W}, \mathbf{w}, \Theta, \hat{\Theta}, R)$ 定义为：

$$\Delta_o(\mathbf{W}, \mathbf{w}, \Theta, \hat{\Theta}, R) = \sum_{j=1}^{|R|} R_j (\sin(\pi |\hat{\Theta}_j - \Theta_j|))$$

1.4 检测

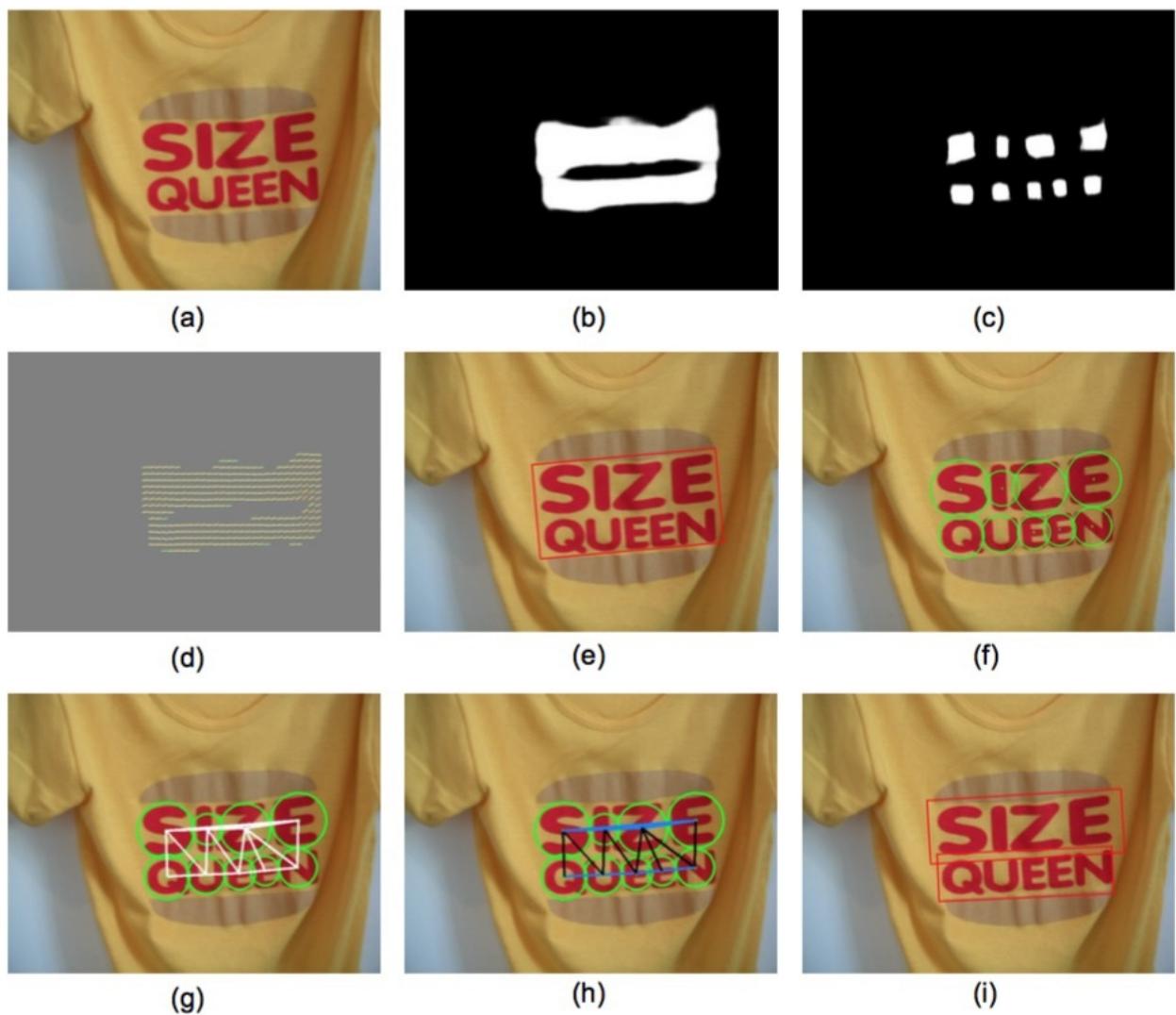
1.4.1 预测掩码

HMCP的预测的三个Map均是由fuse层得到，因为作者发现side branch的引入反而会伤害模型的性能。

1.4.2 检测框生成

HMCP的检测过程如图5：给定输入图像(a)得到(b), (c), (d)三组掩码。通过自适应阈值，我们可以得到(e)以及(f)的分别基于文本区域和基于字符的检测框，需要注意的是我们在制作字符掩码的时候掩码区域被压缩了一半，所以在这里我们需要将它们还原回来。

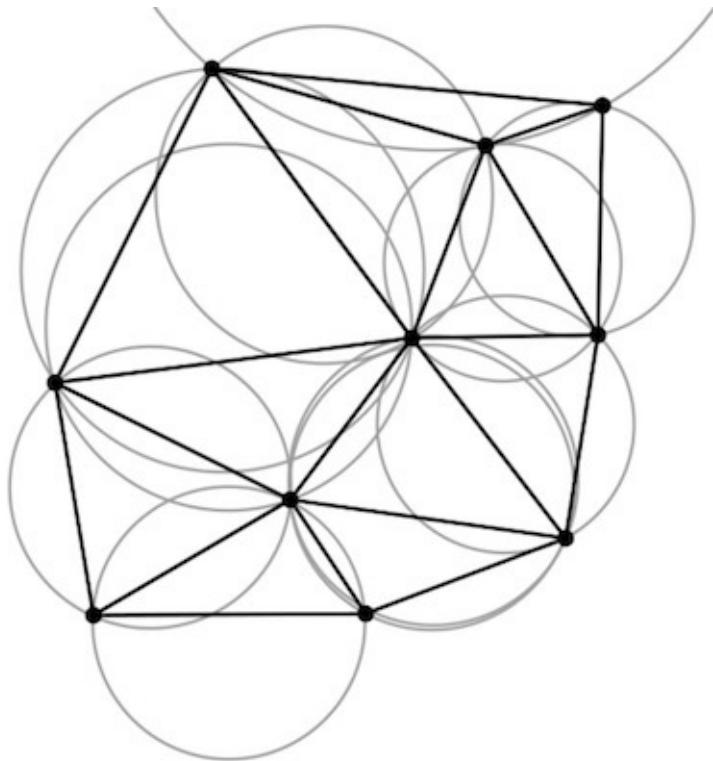
图5：HMCP的检测框生成流程



对于一个文本区域，假设其中有 m 个字符区域： $U = \{u_i, i = 1, \dots, m\}$ ，通过德劳内三角化 (Delaunay Triangulation) [26]得到的三角形 T 我们可以得到一个由相邻字符间连接构成的图 $G = \{U, E\}$ ，如图(g)。

德劳内三角化能够有效的去除字符区域之间不必要的链接，维基百科给的德劳内三角化的定义是指德劳内三角化是一种三角剖分 $DT(P)$ ，使得在 P 中没有点严格处于 $DT(P)$ 中任意一个三角形外接圆的内部。德劳内三角化最大化了此三角剖分中三角形的最小角，换句话，此算法尽量避免出现“极瘦”的三角形，如图6。

图6：德劳内三角化



在图 $G = \{U, E\}$ 中， U 表示图的顶点表示字符的位置。 E 表示图的边表示两个字符之间的相似度，边的权值 w 的计算方式为：

$$w = \begin{cases} s(i, j) & \text{if } e \in T \\ 0 & \text{otherwise} \end{cases}$$

$s(i, j)$ 有空间相似性 $a(i, j)$ 和角度相似性 $o(i, j)$ 计算得到：

$$s(i, j) = \frac{2a(i, j)o(i, j)}{a(i, j) + o(i, j)}$$

空间相似性定义为

$$a(i, j) = \exp\left(-\frac{d^2(i, j)}{2D^2}\right)$$

其中 $d(i, j)$ 是 (u_i, u_j) 之间的欧氏距离， D 是整个德劳内三角形的边长均值。

角度相似性 $o(i, j)$ 定义为：

$$o(i, j) = \cos(\Lambda(\phi(i, j) - \psi(i, j)))$$

其中 $\phi(i, j)$ 表示 (u_i, u_j) 形成的直线与水平方向的夹角， $\psi(i, j)$ 即两个节点之间的区域的所有像素点的夹角的平均值， Λ 表示两个角度的夹角。

综上可以看出， $s(i, j)$ 和 $d(i, j)$ 成反比，也就是两个字符的距离越近，这两个字符的相似度越高； $s(i, j)$ 和 $\phi(i, j)$ 也是成反比，即两个字符的角度越小（越接近水平文本行），两个字符的相似度越高。

得到带权值的无向图之后，使用 Kruskal 等方法生成最大生成树 M 。由图到树的生成过程是一个剪枝的过程，如若在树的基础上再进行剪枝，此时树便会分裂成由若干棵个树组成的森林。在最大生成树的基础上剪枝 $K - 1$ 次会生成 K 棵树， $K \geq 1$ 。如图 5.(a) 所示，文本区域由两行文本构成，显然我们需要在 M 的基础上进行一次剪枝才能生成两棵树，从而通过两棵树分别确定一个文本区域。

那么，给定一张图片，我们如何其文本行的个数（树的个数） K 呢？论文给出的策略是最大化 S_{vm} ， S_{vm} 的计算方式为：

$$S_{vm} = \sum_{i=1}^K \frac{\lambda_{i1}}{\lambda_{i2}}$$

其中 λ_{i1} 和 λ_{i2} 是协方差矩阵 C_i 最大和第二大的两个特征值。而 C_i 是由一个文本区域（论文中叫做 cluster）的字符中心点坐标构成的矩阵。

在一些样本中，会存在如图 7 所示的弧形文本区域，而传统的方法是基于一条直线上的文本行设计的。为了解决这个问题，HMCP 引入了阈值 τ ，权值大于 τ 的边既不会被删除，也不会被选中。

图 7 : HMCP 用于弧形文字区域检测



其实关于上面这两段不是很理解工作原理，带后续补充。

总结

这篇论文巧妙的将语义分割用于场景文字检测领域，其三个掩码图的多任务模型的设计非常漂亮。最后通过三个掩码图生成检测框的算法技术性非常高，是一篇非常值得学习的论文。

Arbitrary-Oriented Scene Text Detection via Rotation Proposals

tags: OCR, RRPN, Faster R-CNN

前言

在场景文字检测中一个最常见的问题便是倾斜文本的检测，现在基于候选区域的场景文字检测方法，例如[CTPN\[31\]](#)，[DeepText\[32\]](#)等，其检测框均是与坐标轴平行的矩形区域，其根本原因在于数据的标签采用了 (x, y, w, h) 的形式。另外一种方法是基于语义分割，例如[HMCP\[29\]](#)，[EAST\[24\]](#)等，但是基于分割算法的场景文字检测效率较低且并不擅长检测长序列文本。

作者提出的RRPN（Rotation Region Proposal Network）[\[25\]](#)可以归结到基于候选区域的类别当中，算法的主要贡献是提出了带旋转角度的锚点，并锚点的角度特征重新设计了IoU，NMS以及ROI池化等算法，RRPN的角度特征使其非常适合对倾斜文本进行检测。

RRPN的这个特征使其不仅可以应用到场景文字检测，在一些存在明显角度特征的场景中，例如建筑物检测，也非常适用。

1.RPN回顾

关于RPN的详细内容可参考[Faster R-CNN\[60\]](#)一文，在这里我们只进行简单的回顾。

RPN是一个全卷积网络，其首先通过3个尺寸，3个尺度的锚点在Feature Map上对输入图像进行密集采样。然后通过一个由判断锚点是前景还是背景的二分类任务和一个用于预测锚点和Ground Truth的位置相对距离的回归模型组成。

RPN的一个位置的特征向量采样 $3 \times 3 = 9$ 个锚点，每个锚点的损失函数由分类任务（2）和回归任务（4）组成，因此一个特征向量有 $9 \times 6 = 54$ 个输出，RPN的损失函数可以表示为：

$$L(p_i, t_i) = \frac{1}{N_{cls}} \sum_i L_{cls}(p_i, p_i^*) + \lambda \frac{1}{N_{reg}} \sum_i p_i^* L_{reg}(t_i, t_i^*)$$

其中 L_{cls} 是分类任务，损失函数是softmax，用于计算该锚点为前景或者背景的概率； L_{reg} 是回归任务，损失函数是Smooth L1，用于计算锚点和Ground Truth的相对关系。

2.RRPN详解

2.1. RRPN网络结构

RRPN的网络结构如图1所示，检测过程可以分成三步：

1. 使用卷积网络产生Feature Map，论文中使用的是VGG-16，也可以替换成物体检测的主流框架，例如基于残差网络的FPN；
2. 使用RRPN产生带角度的候选区域；
3. 使用RRoI Pooling产生长度固定的特征向量，之后接两层全连接用于候选区域的类别精校。

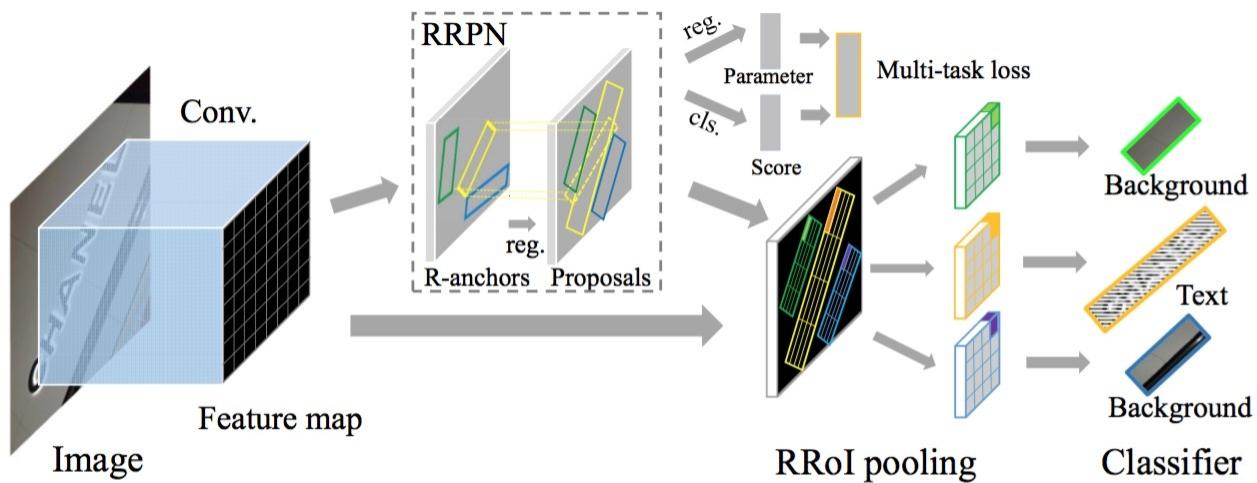


图1：RRPN网络结构图

2.2. R-Anchor

传统的RPN的锚点均是与坐标轴平行的矩形，而RRPN中添加了角度信息，我们将这样的锚点叫做R-Anchor。R-Anchor由 (x, y, w, h, θ) 五要素组成，其中 (x, y) 表示bounding box的几何中心（RPN中是左上角）。 (w, h) 分别是bounding box的长边和短边。 θ 是锚点的旋转角度，通过 $\theta + k\pi$ 将 θ 的范围控制在 $[-\frac{\pi}{4}, \frac{3\pi}{4}]$ 。

对比另外一种用4个点 $(x_1, y_1, x_2, y_2, x_3, y_3, x_4, y_4)$ 表示任意四边形的策略相比，R-Anchor有以下3条优点：

1. 两个四边形的相对角度更好计算；
2. 回归的值更少，模型更好训练；
3. 更容易进行图像扩充（2.3节）。

R-Anchor的锚点由3个尺寸，3个比例以及6个角度组成：3个尺寸分别是8，16，32；3个比例分别是 $1:2, 1:5, 1:8$ ；6个角度分别是 $-\frac{\pi}{6}, 0, \frac{\pi}{6}, \frac{\pi}{3}, \frac{\pi}{2}, \frac{2\pi}{3}$ 。锚点的形状如图2所示。因此在RRPN中每个特征向量共有 $3 \times 3 \times 6 = 54$ 个锚点。

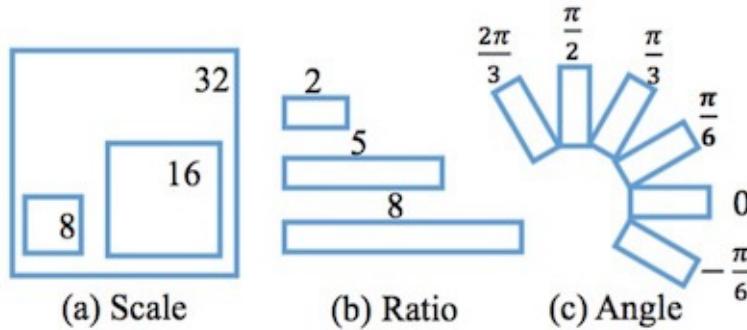


图2：RRPN的锚点

2.3. RRPN的图像扩充

为了缓解过拟合的问题，并增加模型对选择区域的检测能力，RRPN使用了数据扩充的方法增加样本的数量。RRPN使用的扩充方法之一是将输入图像选择 α 。

对于一张尺寸为 $I_W \times I_H$ 的输入图像，设其中一个Ground Truth表示为 (x, y, w, h, θ) ，旋转 α 后得到的Ground Truth为 $(x', y', w', h', \theta')$ ，其中Ground Truth的尺寸并不会改变，即 $w' = w$ ， $h' = h$ 。 $\theta' = \theta + \alpha + k\pi$ ， $k\pi$ 用于将 θ' 的范围控制到 $[-\frac{\pi}{4}, \frac{3\pi}{4}]$ 之间。 (x', y') 的计算方式为：

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \mathbf{T}\left(\frac{I_W}{2}, \frac{I_H}{2}\right) \mathbf{R}(\alpha) \mathbf{T}\left(-\frac{I_W}{2}, -\frac{I_H}{2}\right) \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

$\mathbf{T}(\delta_x, \delta_y)$ 和 $\mathbf{R}(\alpha)$ 的定义分别是：

$$\mathbf{T}(\delta_x, \delta_y) = \begin{bmatrix} 1 & 0 & \delta_x \\ 0 & 1 & \delta_y \\ 0 & 0 & 1 \end{bmatrix}$$

$$\mathbf{R}(\alpha) = \begin{bmatrix} \cos \alpha & \sin \alpha & 0 \\ -\sin \alpha & \cos \alpha & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

2.4. RRPN中正负锚点的判断规则

由于RRPN中引入了角度标签，传统RPN的锚点正负的判断方法是不能应用到RRPN中的。

RRPN的正锚点的判断规则满足下面两点（and关系）：

1. 当锚点与Ground Truth的IoU大于0.7；

2. 当锚点与Ground Truth的夹角小于 $\frac{\pi}{12}$ 。

RRPN的负锚点的判断规则满足1或者2（or关系）：

1. 与锚点的IoU小于0.3；

2. 与锚点的IoU大于0.7，但是夹角也大于 $\frac{\pi}{12}$ 。

在训练时，只有判断为正锚点和负锚点的样本参与训练，其它不满足的样本并不会训练。

2.5. RRPN中IoU的计算方法

RRPN中IoU的计算和RPN思路相同，但是由于引入了角度信息，两个旋转矩形的交集就比较复杂了，如图3所示，两个相交旋转矩形的交集可根据交点的个数分为三种情况，分别是4个，6个，8个交点：

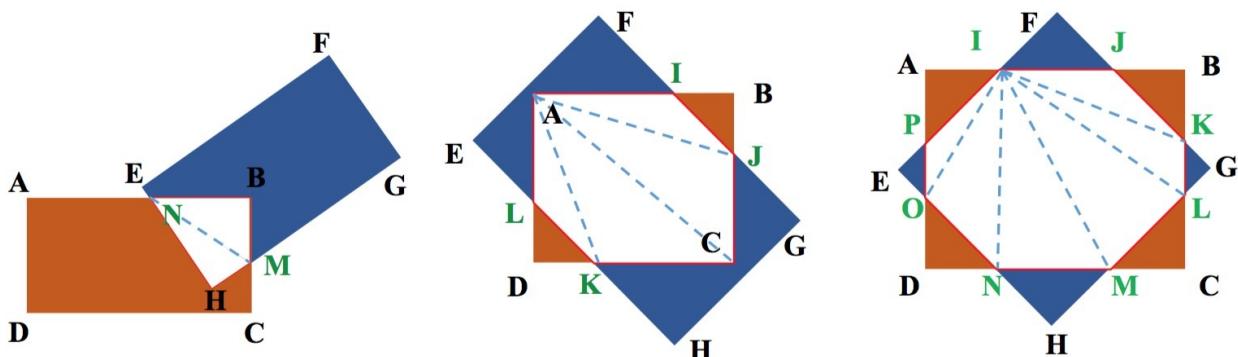


图3：旋转矩形交集情况

两个矩形的交集的计算方式见下面伪代码：

Algorithm 1 IoU computation

```

1: Input: Rectangles  $R_1, R_2, \dots, R_N$ 
2: Output: IoU between rectangle pairs  $IoU$ 
3: for each pair  $\langle R_i, R_j \rangle$  ( $i < j$ ) do
4:   Point set  $PSet \leftarrow \emptyset$ 
5:   Add intersection points of  $R_i$  and  $R_j$  to  $PSet$ 
6:   Add the vertices of  $R_i$  inside  $R_j$  to  $PSet$ 
7:   Add the vertices of  $R_j$  inside  $R_i$  to  $PSet$ 
8:   Sort  $PSet$  into anticlockwise order
9:   Compute intersection  $I$  of  $PSet$  by triangulation
10:   $IoU[i, j] \leftarrow \frac{Area(I)}{Area(R_i) + Area(R_j) - Area(I)}$ 
11: end for

```

首先根据矩形四条边的带定义域的一元一次方程求出所有交点，然后补充完整交集的顶点（即添加位于矩形B中的矩形A的所有顶点），顺时针排序之后最后根据三角形的三个顶点计算相交区域的面积。

2.6. RRPN的损失函数

RRPN的损失函数由分类任务和回归任务组成：

$$L(p, l, v^*, v) = L_{cls}(p, l) + \lambda l L_{reg}(v^*, v)$$

在上面的公式中， l 表示前/背景的指示值，前景（文字区域）时 $l = 1$ ，背景时 $l = 0$ 。

$p = (p_0, p_1)$ 表示的样本为前景或者背景的概率，使用了softmax作为激活函数，因此和是1。

$v = (v_x, v_y, v_w, v_h, v_\theta)$ 表示预测值，它计算的是bounding box和锚点的相对关系，因此对尺度不敏感。 $v = (v_x^*, v_y^*, v_w^*, v_h^*, v_\theta^*)$ 表示Ground Truth，也是计算的和锚点的相对关系。 λ 表示的是两个任务之间的平衡参数。

$L_{cls}(p, l)$ 使用的是log损失函数：

$$L_{cls}(p, l) = -\log p_l$$

$L_{reg}(v^*, v)$ 使用的是smooth- L_1 损失函数：

$$L_{reg}(v^*, v) = \sum_{i \in \{x, y, h, w, \theta\}} \text{smooth}_{L_1}(v_i^*, v_i)$$

尺度不敏感是通过对 \mathbf{v} 进行归一化实现的,设 $(x_a, y_a, w_a, h_a, \theta_a)$ 为当前锚点 ,

$\mathbf{v} = (v_x, v_y, v_w, v_h, v_\theta)$ 的换算为 :

$$v_x = \frac{x - x_a}{w_a}, v_y = \frac{y - y_a}{h_a}$$

$$v_h = \log \frac{h}{h_a}, v_w = \log \frac{w}{w_a}$$

$$v_\theta = \theta^* \ominus \theta_a$$

其中 $a \ominus b = a - b + k\pi$, k 用于控制 $a \ominus b$ 的值在 $[-\frac{\pi}{4}, \frac{3\pi}{4})$ 的范围内。

和RPN相比 , RRPN的最大变化在于在回归任务中添加了对相对角度 θ 的预测。 θ 的变化范围是 $[-\frac{\pi}{4}, \frac{3\pi}{4})$, 而锚点的6个角度分别是 $-\frac{\pi}{6}, 0, \frac{\pi}{6}, \frac{\pi}{3}, \frac{\pi}{2}, \frac{2\pi}{3}$ (逐次增加 $\frac{\pi}{12}$) 。结合正负锚点的判断规则 , 我们可以知道每个锚点都有一个对应的匹配范围 , 论文中将其命名为匹配域 (fit domain) , 匹配域有两个重要特点 :

1. 不同角度的锚点的匹配域是不相交的 ;
2. 同一个向量的不同角度的锚点的并集是全集。

上面两个特征产生了一个非常重要的性质 : 当 $(\mathbf{x}, \mathbf{y}, \mathbf{w}, \mathbf{h})$ 确定时 , 并且 $\text{IoU} > 0.7$, 一个**Ground Truth**有且只有一个正锚点与之匹配。

图4是对该损失函数的可视化 , 线段的角度代表预测的bounding box的旋转角度 , 长度代表置信度。

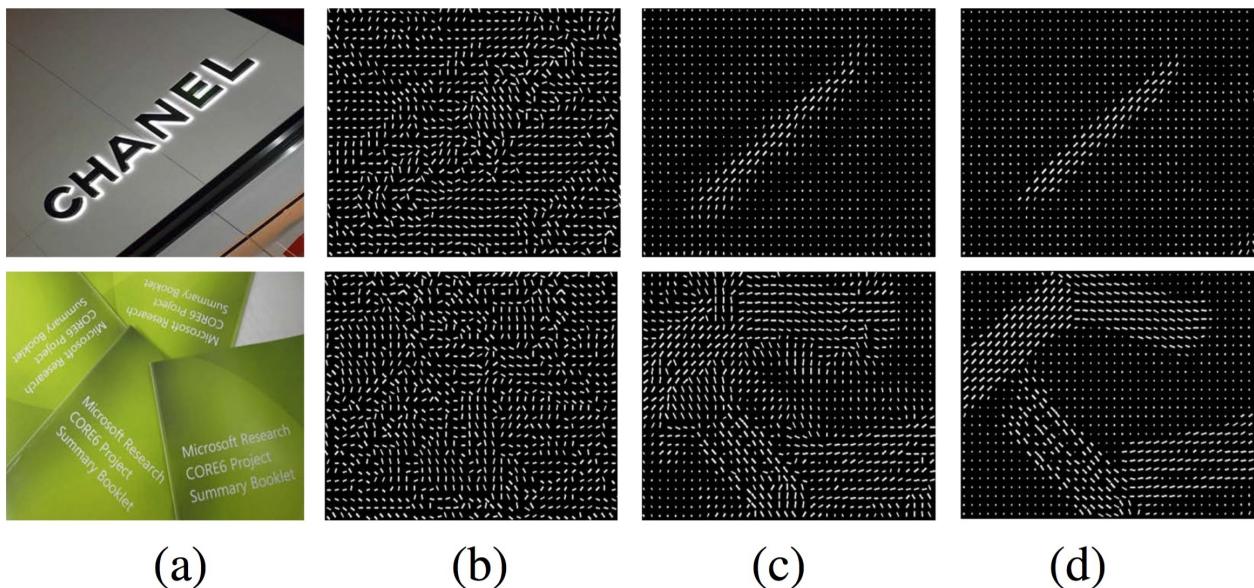


图4：RRPN损失函数的可视化：(a)input image, (b)0 iters, (c)1500 iters, (d)15000 iters

3. 位置精校部分详解

3.1 Skew NMS

传统的NMS只考虑IoU一个因素，而这点在RRPN中是行不通的，考虑一个偏移了 $\frac{\pi}{12}$ 的矩形，虽然IoU只有0.31，但是由于偏移角度比较小，这种情况也应该考虑进去。

Skew-NMS在NMS的基础上加入了IoU信息：

1. 保留IoU大于0.7的最大的候选框；
2. 如果所有的候选框均位于 $[0.3, 0.7]$ 之间，保留小于 $\frac{\pi}{12}$ 的最小候选框。

3.2 RRol Pooling

如图1所示，RRPN得到的候选区域是旋转矩形，而传统的RoI池化只能处理与坐标轴平行的候选区域，因此作者提出了RRol Pooling用于RRPN中的旋转矩形的池化。

如图5所示，首先需要设置超参数 H_r 和 W_r ，分别表示池化后得到的Feature Map的高和宽。然后将RRPN得到的候选区域等分成 $H_r \times W_r$ 个小区域，每个子区域的大小是 $\frac{w}{W_r} \times \frac{h}{H_r}$ ，这时每个区域仍然是带角度的，如5.(a)所示。接着通过仿射变换将子区域转换成平行于坐标轴的矩形，最后通过Max Pooling得到长度固定的特征向量。

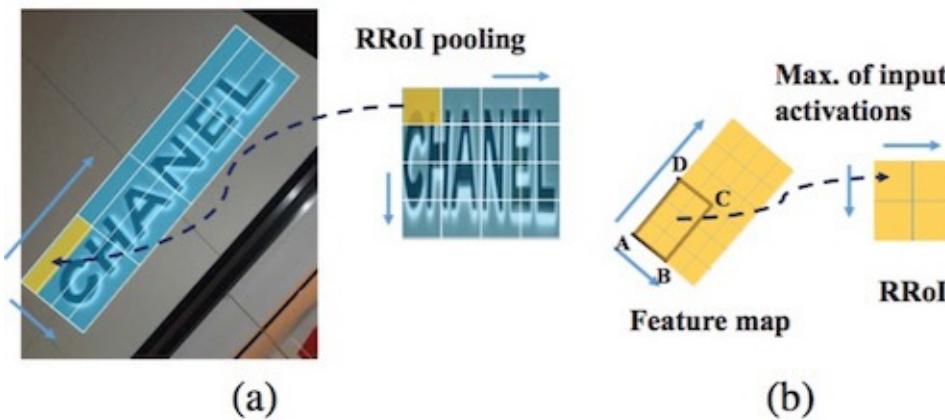


图 5 : RRoi Pooling

RROI Pooling的计算流程如算法2的伪代码。其中第一层for循环是遍历候选区域的所有子区域，5-7行通过仿射变换将子区域转换成标准矩形，第二个for循环用于取得每个子区域的最大值，10-11行由于对标准矩形中元素的插值，使用了向下取整的方式。

Algorithm 2 RRoi pooling

```

1: Input: Proposal  $(x, y, h, w, \theta)$ , pooled size  $(H_r, W_r)$ , input
   feature map  $InFeatMap$ , spatial scale  $SS$ 
2: Output: Output feature map  $OutFeatMap$ 
3:  $Grid_w, Grid_h \leftarrow \frac{w}{W_r}, \frac{h}{H_r}$ 
4: for  $\langle i, j \rangle \in \{0, \dots, H_r - 1\} \times \{0, \dots, W_r - 1\}$  do
5:    $L, T \leftarrow x - \frac{w}{2} + jGrid_w, y - \frac{h}{2} + iGrid_h$ 
6:    $L_{rotate} \leftarrow (L - x) \cos \theta + (T - y) \sin \theta + x$ 
7:    $T_{rotate} \leftarrow (T - y) \cos \theta - (L - x) \sin \theta + y$ 
8:    $value \leftarrow 0$ 
9:   for  $\langle k, l \rangle \in \{0, \dots, \lfloor Grid_h \cdot SS - 1 \rfloor\} \times \{0, \dots, \lfloor Grid_w \cdot SS - 1 \rfloor\}$  do
10:     $P_x \leftarrow \lfloor L_{rotate} \cdot SS + l \cos \theta + k \sin \theta + \frac{1}{2} \rfloor$ 
11:     $P_y \leftarrow \lfloor T_{rotate} \cdot SS - l \sin \theta + k \cos \theta + \frac{1}{2} \rfloor$ 
12:    if  $InFeatMap[P_y, P_x] > value$  then
13:       $value \leftarrow InFeatMap[P_y, P_x]$ 
14:    end if
15:  end for
16:   $OutFeatMap[i, j] \leftarrow value$ 
17: end for

```

在RRoI Pooling之后，模型接来两个全连接层用于判断待检测区域是前景区域还是背景区域。

总结

RRPN非常创新的提出了使用带角度的锚点处理场景文字检测中最常见的倾斜问题，为了配合R-Anchor，论文中对RoI，NMS的计算也做了正对性的修改，另外RRoI Pooling层的提出将池化的目标区域扩展到了不仅仅局限于标准矩形。

结合目标检测中的一些Trick，应该能将检测精度进一步提高，使RRPN在特定场景的比赛中也非常有用。

最后还有一点疑问：论文中说RRPN的位置精校部分只进行了二分类，但个人感觉更好的策略是Faster R-CNN中的多任务模型，粗略的看了一下代码好像也是使用的Faster R-CNN的策略。

Spatial Transformer Networks

Tags: OCR, STN

前言

自LeNet-5的结构被提出之后，其“卷积+池化+全连接”的结构被广泛的应用到各处，但是这并不代表其结构没有了优化空间。传统的池化方式（Max Pooling/Average Pooling）所带来卷积网络的位移不变性和旋转不变性只是局部的和固定的。而且池化并不擅长处理其它形式的仿射变换。

Spatial Transformer Network (STN) [41]的提出动机源于对池化的改进，即与其让网络抽象的学习位移不变性和旋转不变性，不如设计一个显示的模块，让网络线性的学习这些不变性，甚至将其范围扩展到所有仿射变换乃至非放射变换。更加通俗的将，STN可以学习一种变换，这种变换可以将进行了仿射变换的目标进行矫正。这也为什么我把STN放在了OCR这一章，因为在OCR场景中，仿射变换是一种最为常见的变化情况。

基于这个动机，作者设计了Spatial Transformer (ST)，ST具有显示学习仿射变换的能力，并且ST是可导的，因此可以直接整合进卷积网络中进行端到端的训练，插入ST的卷积网络叫做STN。

下面根据一份STN的keras源码：https://github.com/oarriaga/spatial_transformer_networks 详解STN的算法细节。

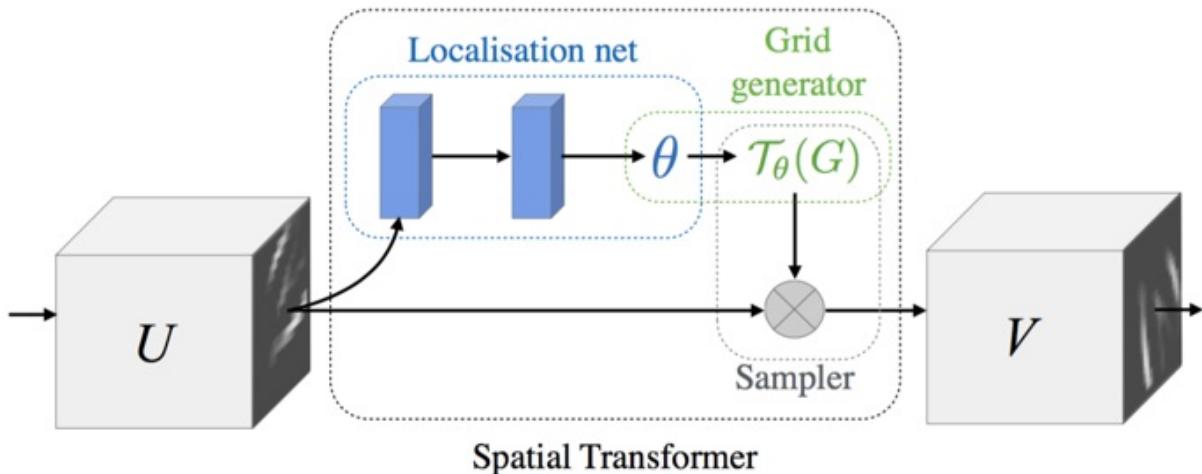
1. ST

ST由三个模块组成：

1. Localisation Network：该模块学习仿射变换矩阵（附件A）；
2. Parameterised Sampling Grid：根据Localisation Network得到仿射变换矩阵，得到输出Feature Map和输入Feature Map之间的位置映射关系；
3. Differentiable Image Sampling：计算输出Feature Map的每个像素点的值。

STM的结构见图1：

图1：STM的框架图



ST使用的插值方法属于“后向插值”的一种，即给定输出Feature Map上的一个点

$G_i = (x_i^t, y_i^t)$ ，我们某种变化呢反向找到其在输入Feature Map中对应的位置 (x_i^s, y_i^s) ，如果 (x_i^s, y_i^s) 为整数，则输出Feature Map在 (x_i^t, y_i^t) 处的值和输入Feature Map在 (x_i^t, y_i^t) 处的值相同，否则需要通过插值的方法得到输出Feature Map在 (x_i^t, y_i^t) 处的值。

说了后向插值，当然还有一种插值方式叫做前向插值，例如我们在[Mask R-CNN](#)中介绍的插值方法。

1.1 Localisation Network

Localisation Network是一个小型的卷积网络 $\Theta = f_{loc}(U)$ ，其输入是Feature Map ($U \in R^{W \times H \times C}$)，输出是仿射矩阵 Θ 的六个值。因此输出层是一个有六个节点回归器。

$$\theta = \begin{bmatrix} \theta_{11} & \theta_{12} & \theta_{13} \\ \theta_{21} & \theta_{22} & \theta_{23} \end{bmatrix}$$

下面的是源码中给出的Localisation Network的结构。

```
locnet = Sequential()
locnet.add(MaxPooling2D(pool_size=(2,2), input_shape=input_shape))
locnet.add(Conv2D(20, (5, 5)))
locnet.add(MaxPooling2D(pool_size=(2,2)))
locnet.add(Conv2D(20, (5, 5)))

locnet.add(Flatten())
locnet.add(Dense(50))
locnet.add(Activation('relu'))
locnet.add(Dense(6, weights=weights))
```

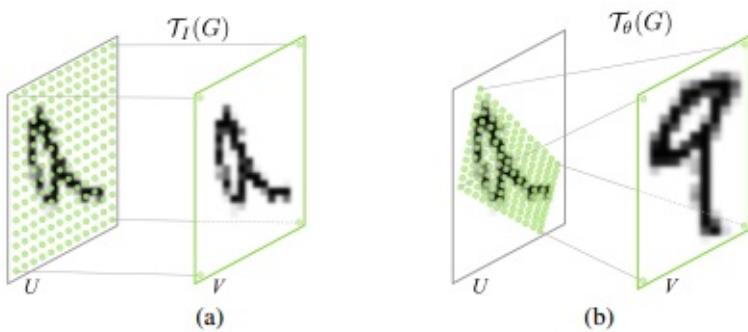
1.2 Parameterised Sampling Grid

Parameterised Sampling Grid利用Localisation Network产生的 Θ 进行仿射变换，即由输出Feature Map上的某一位置 $G_i = (x_i^t, y_i^t)$ 根据变换参数 θ 得到输入Feature Map的某一位置 (x_i^s, y_i^s) ：

$$\begin{pmatrix} x_i^s \\ y_i^s \\ 1 \end{pmatrix} = \mathcal{T}_\theta(G_i) = \Theta \begin{pmatrix} x_i^t \\ y_i^t \\ 1 \end{pmatrix} = \begin{bmatrix} \theta_{11} & \theta_{12} & \theta_{13} \\ \theta_{21} & \theta_{22} & \theta_{23} \end{bmatrix} \begin{pmatrix} x_i^t \\ y_i^t \\ 1 \end{pmatrix}$$

图2展示了ST中的一次仿射变换 (b) 和直接映射的区别。

图2: ST的仿射变换和普通卷积的直接映射



这里需要注意两点：

1. Θ 可以是一个更通用的矩阵，并不局限于仿射变换，甚至不局限于6个值；
2. 映射得到的 (x_i^s, y_i^s) 一般不是整数，因此不能 (x_i^t, y_i^t) 不能使用 (x_i^s, y_i^s) 的值，而是根据它进行插值，也就是我们下一节要讲的东西。

1.3 Differentiable Image Sampling

如果 (x_i^s, y_i^s) 为一整数，那么输出Feature Map的 (x_i^t, y_i^t) 处的值便可以从输入Feature Map上直接映射过去。然而在的1.2节我们讲到， (x_i^s, y_i^s) 往往不是整数，这时我们需要进行插值才能确定输出其值，在这个过程叫做一次插值，或者一次采样 (Sampling)。插值过程可以用下式表示：

$$V_i^c = \sum_n^H \sum_m^W U_{nm}^c k(x_i^s - m; \Phi_x) k(y_i^s - m; \Phi_y), \quad \text{where } \forall i \in [1, \dots, H'W'], \forall c \in [1, \dots, C]$$

在上式中，函数 $f()$ 表示插值函数，本文将以双线性插值为例进行解析， Φ 为 $f()$ 中的参数， U_{nm}^c 为输入Feature Map上点 (n, m, c) 处的值， V_i^c 便是插值后输出Feature Map的 (x_i^t, y_i^t) 处的值。

H', W' 分别为输出 Feature Map 的高和宽。当 $H' = H$ 并且 $W' = W$ 时，则 ST 是正常的仿射变换，当 $H' = H/2$ 并且 $W' = W/2$ 时，此时 ST 可以起到和池化类似的功能。

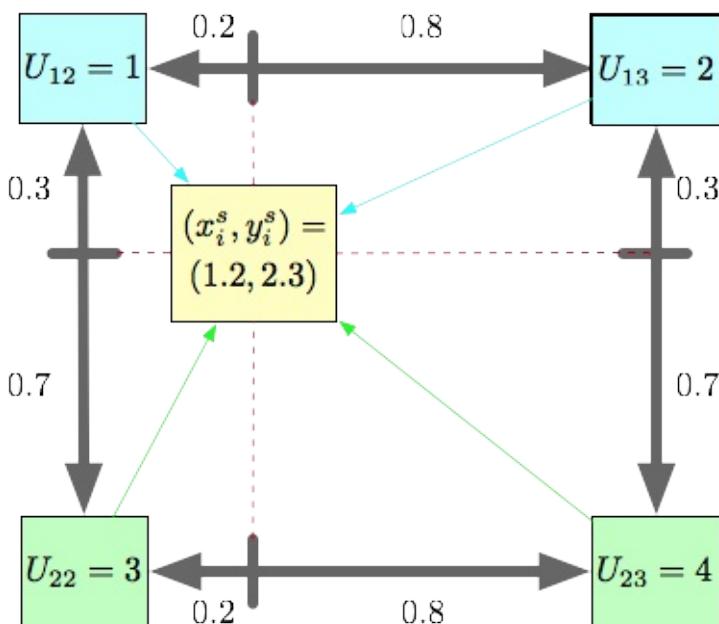
以双线性插值为例，插值过程即为：

$$V_i^c = \sum_n \sum_m U_{nm}^c \max(0, 1 - |x_i^s - m|) \max(0, 1 - |y_i^s - m|)$$

上式可以这么理解：遍历整个输入 Feature Map，如果遍历到的点 (n, m) 距离大于 1，即 $|x_i^s - m| > 1$ ，那么 $\max(0, 1 - |x_i^s - m|) = 0$ （ n 处同理），即只有距离 (x_i^s, y_i^s) 最近的四个点参与计算。且距离与权重成反比，也就是距离越小，权值越大，也就是双线性插值的过程，如图 3。其中 $(x_i^s, y_i^s) = (1.2, 2.3)$, $U_{12} = 1, U_{13} = 2, U_{22} = 3, U_{23} = 4$ ，则

$$V_{(x_i^s, y_i^s)} = 0.8 \times 0.7 \times 1 + 0.2 \times 0.7 \times 2 + 0.8 \times 0.3 \times 3 + 0.2 \times 0.3 \times 4 = 1.8$$

图 3：STN 中的双线性插值示例



上式中的几个值都是可偏导的：

$$\frac{\partial V_i^c}{\partial U_{nm}^c} = \sum_n \sum_m \max(0, 1 - |x_i^s - m|) \max(0, 1 - |y_i^s - m|)$$

$$\frac{\partial V_i^c}{\partial x_i^s} = \sum_n \sum_m U_{nm}^c \max(0, 1 - |y_i^s - m|) \begin{cases} 0 & \text{if } |m - x_i^s| > 1 \\ 1 & \text{if } m \geq x_i^s \\ -1 & \text{if } m < x_i^s \end{cases}$$

$$\frac{\partial V_i^c}{\partial y_i^s} = \begin{cases} 0 & \text{if } |n - y_i^s| > 1 \end{cases}$$

$$\frac{\partial V_i^c}{\partial y_i^s} = \sum_n^H \sum_m^W U_{nm}^c \max(0, 1 - |x_i^s - n|) \begin{cases} 0 & \text{if } |n - y_i^s| > 1 \\ 1 & \text{if } n \geq y_i^s \\ -1 & \text{if } n < y_i^s \end{cases}$$

在对 θ 求导为：

$$\frac{\partial V_i^c}{\partial \theta} = \begin{pmatrix} \frac{\partial V_i^c}{\partial x_i^s} \cdot \frac{\partial x_i^s}{\partial \theta} \\ \frac{\partial V_i^c}{\partial y_i^s} \cdot \frac{\partial y_i^s}{\partial \theta} \end{pmatrix}$$

ST的可导带来的好处是其可以和整个卷积网络一起端到端的训练，能够以layer的形式直接插入到卷积网络中。

2. STN

1.3节中介绍过，将ST插入到卷积网络中便得到了STN，在插入ST的时候，需要注意以下几点：

1. 在输入图像之后接一个ST是最常见的操作，也是最容易理解的，即自动图像矫正；
2. 理论上讲ST是可以以任意数量插入到网络中的任意位置，ST可以起到裁剪的作用，是一种高级的Attention机制。但多个ST无疑增加了网络的深度，其带来的收益价值值得讨论；
3. STM虽然可以起到降采样的作用，但一般不这么使用，因为基于ST的降采样产生了对其的问题；
4. 可以在同一个卷积网络中并行使用多个ST，但是一般ST和图像中的对象是1:1的关系，因此并不是具有非常广泛的通用性。

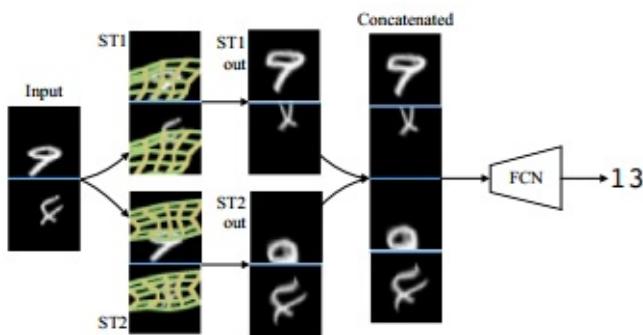
3. STN的应用场景

3.1 并行ST

在这个场景中，输入是两张有仿射变换的MNIST的图片，然后直接输出这两个图片的数字的和（是一个19类的分类任务，不是两个10分类任务），如图3右侧图。

图4：并行ST

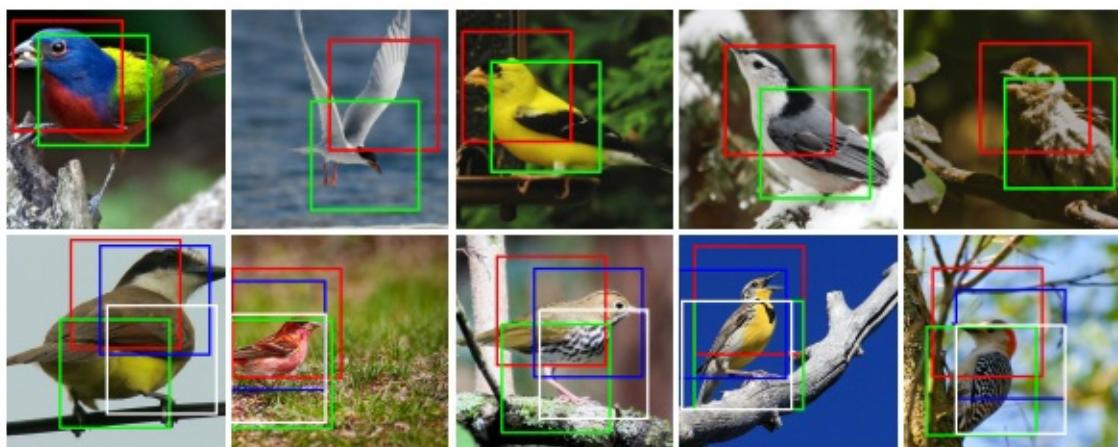
Model	RTS
FCN	47.7
CNN	14.7
ST-FCN	22.6
Proj	18.5
TPS	19.1
Aff	9.0
2×ST-FCN	5.9
Proj	5.8
TPS	5.8



具体的将，给定两张图像，初始化两个ST，将两个ST分别作用于两张图片，得到四个Feature Map，将这个四个通道的图片作为FCN的输入，预测0-18间的一个整数值。图3左边的实验结果显示了两个并行ST的效果明显强于单个ST。

在鸟类分类的任务上，作者并行使用了两个和四个ST，得到了图5的实验结果：

图5：STN用于鸟类分类



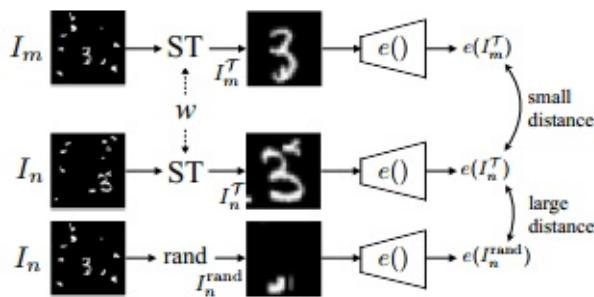
在这里STN可以理解为一种Attention机制，即不同的Feature Map注意小鸟的不同部分，例如上面一排明显可以看出红色Feature Map比较注意小鸟的头部，而绿色则比较注重小鸟的身体。

3.2 STN用于半监督学习的co-localisation

在co-localisation中，给出一组图片，这些图片中包含一些公共部分，但是这组公共部分在什么地方，长什么样子我们都不知道，我们的任务时定位这些公共部分。

STN解决这个任务的方案是STN在图片m检测到的部分与在图片n中检测到的部分的相似性应该小于STN在n中随机采样的部分，如图6。

图6：STN用于半监督学习的co-localisation



Loss使用的是Hinge损失函数:

$$\sum_n^N \sum_{n \neq m}^M \max(0, \|e(I_n^T) - e(I_m^T)\|_2^2 - \|e(I_n^T) - e(I_n^{rand})\|_2^2 + \alpha)$$

其中 I_n^T 和 I_m^T 是 STN 裁剪得到的图像， I_n^{rand} 是随机采样的图像， $e()$ 是编码函数， α 是 hinge loss 的 margin，即裕度，相当于净赚多少。

3.3 高维ST

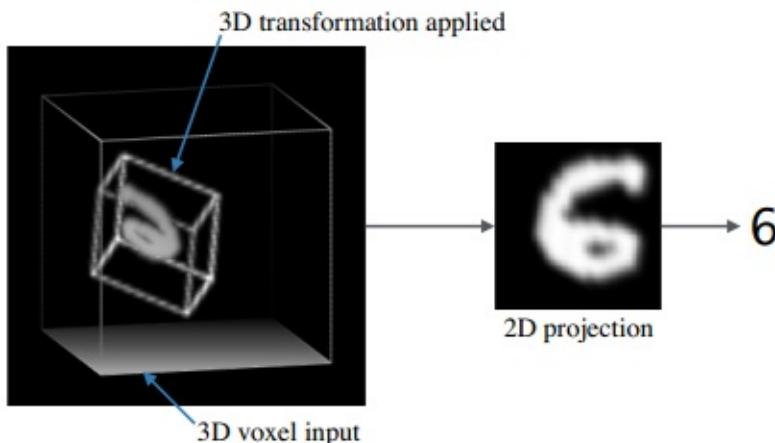
STN也可扩展到三维，此时的放射变换矩阵是的3行4列的，仿射变换表示为:

$$\begin{pmatrix} x_i^s \\ y_i^s \\ z_i^s \end{pmatrix} = \begin{bmatrix} \theta_{11} & \theta_{12} & \theta_{13} & \theta_{14} \\ \theta_{21} & \theta_{22} & \theta_{23} & \theta_{24} \\ \theta_{31} & \theta_{32} & \theta_{33} & \theta_{34} \end{bmatrix} = \begin{pmatrix} x_i^t \\ y_i^t \\ z_i^t \\ 1 \end{pmatrix}$$

此时 Localisation Network 需要回归预测 12 个值，插值则是使用的三线性插值。

STN 的另外一个有趣的方向是通过将图像在一个维度上展开，将 3 维物体压缩到二维，如图 7。

图7：STN用于高维映射



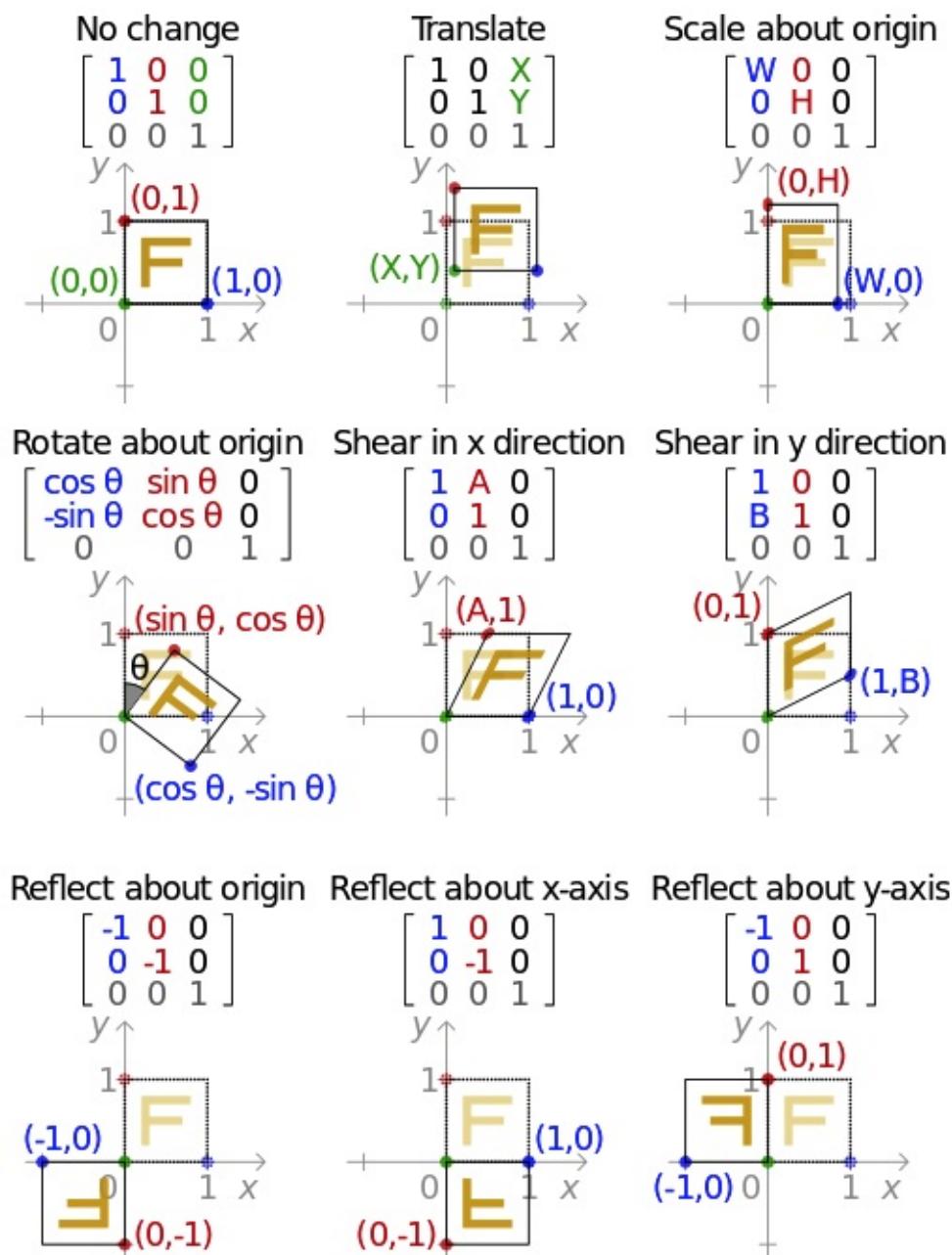
附件A：仿射变换矩阵

仿射变换(Affine Transformation)是一种二维坐标到二维坐标的变化，其保持了二维图形的平直性 (straightness，即变换后直线依旧是直线，不会变成曲线) 和平行性 (parallelness，平行线依旧平行，不会相交)。仿射变换可以由一系列原子变换构成，其中包括：平移 (Translation)，缩放 (Scale)，翻转 (Flip)，旋转 (Rotation) 和剪切 (Crop)。仿射变换可以用下面公式表示：

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} \theta_{11} & \theta_{12} & \theta_{13} \\ \theta_{21} & \theta_{22} & \theta_{23} \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

图8是一些常见的仿射变换的形式及其对应的仿射变换矩阵。

图8：常见仿射变换



Robust Scene Text Recognition with Automatic Rectification

tags: RARE, OCR, STN, TPS, Attention

前言

RARE[22]实现了对不规则文本的end-to-end的识别，算法包括两部分：

1. 基于STN[41]的不规则文本区域的矫正：与STN不同的是，RARE在Localisation部分预测的并不是仿射变换矩阵，而是K个TPS（Thin Plate Spline）[20]的基准点，其中TPS基于样条（spines）的数据插值和平滑技术，在1.1节中会详细介绍其在RARE中的计算过程。
2. 基于SRN的文字识别：SRN（Sequence Recognition Network）是基于Attention的序列模型，包括有CNN和LSTM构成的编码（Encoder）模块和基于Attention和GRU的解码（Decoder）模块构成，此部分会在1.2节介绍。

在测试阶段，RARE使用了基于贪心或Beam Search的方法寻找最优输出结果。

RARE的流程如图1。

图1：RARE的算法框架，其中实线表示预测流程，虚线表示反向迭代过程。



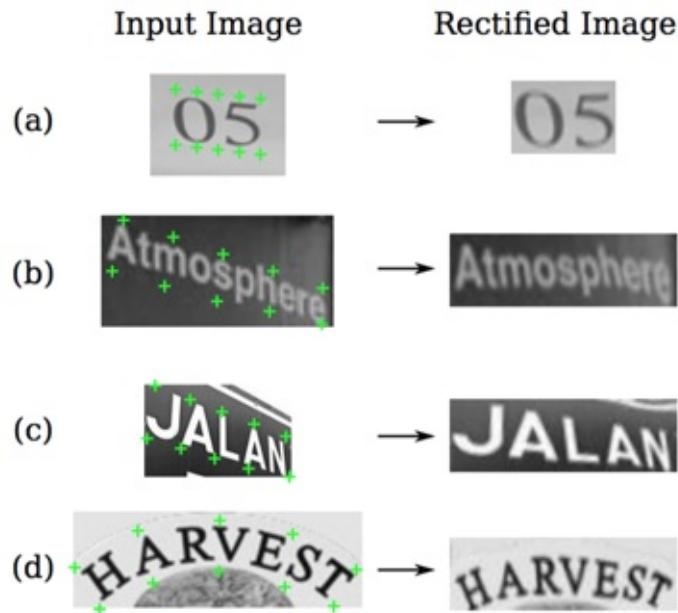
1. RARE详解

1.1 Spatial Transformer Network

场景文字检测的难点有很多，仿射变换是其中一种，Jaderberg[2]等人提出的STN通过预测仿射变换矩阵的方式对输入图像进行矫正。但是真实场景的不规则文本要复杂的多，可能包括扭曲，弧形排列等情况（图2），这种方式的变换是传统的STN解决不了的，因此作者提出了基于TPS的STN。TPS非常强大的一点在于其可以近似所有和生物有关的形变。

图2：自然场景中的变换，左侧是输入图像，右侧是矫正后的效果，其中涉及的变换包括：(a)

loosely-bounded text; (b) multi-oriented text; (c) perspective text; (d) curved text.



TPS是一种基于样条的数据插值和平滑技术。要详细了解STN的细节和动机，可以自行去看论文，我暂无计划解析这篇1989年提出的和深度学习关系不大的论文。对于TPS可以这么简单理解，给我们一块光滑的薄铁板，我们弯曲这块铁板使其穿过空间中固定的几个点，TPS得到的便是我们弯曲铁板所耗费的最小的功。

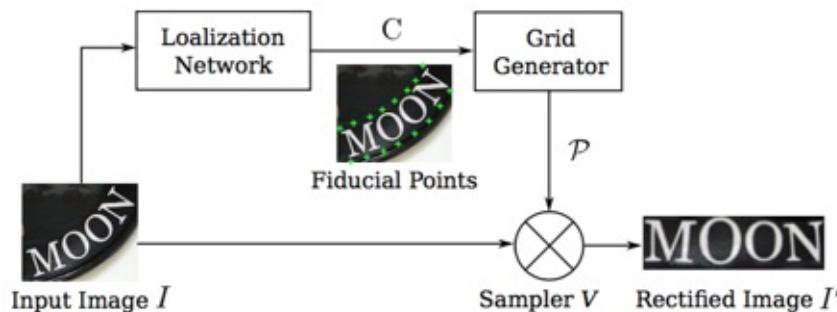
TPS也常用于对扭曲图像的矫正，1.1.2节中介绍计算流程，至于为什么能work，我也暂时没有搞懂。

纵观整个矫正算法，RARE的STN也分成3部分：

1. **localization network**: 预测TPS矫正所需要的 K 个基准点 (fiducial point) ；
2. **Grid Generator** : 基于基准点进行TPS变换，生成输出Feature Map的采样窗格 (Grid) ；
3. **Sampler** : 每个Grid执行双线性插值。

STN的算法流程如图3。

图3：RARE中的STN



1.1.1 localization network

Localization network是一个有卷积层，池化层和全连接构成的卷积网络（图4）。由于一个点由 (x, y) 定义，所以一个要预测 K 个基准点的卷积网络需要由 $2K$ 个输出。为了将基准点的范围控制到 $[-1, 1]$ ，输出层使用 $tanh$ 作为激活函数，在论文的实验部分给出 $K = 20$ 。如图2和图3所示的绿色'+'即为Localization network预测的基准点。

得到网络的输出后，其被`reshape`成一个 $2 \times K$ 的矩阵 \mathbf{C} ，即 $\mathbf{C} = [\mathbf{c}_1, \mathbf{c}_2, \dots, \mathbf{c}_K] \in \mathfrak{R}^{2 \times K}$ 。

图4: Localization network的结构，其中所有卷积的 $padding = 1$ ，所有 $stride = 1$

Type	Filters	Size	Output
Convolution	64	3*3	100*32
Convolution	128	3*3	100*32
Convolution	256	3*3	100*32
Convolution	512	3*3	100*32
Max-pooling	512	2*2	50*16
fc	-	-	1000
fc	-	-	1000
output	-	-	40

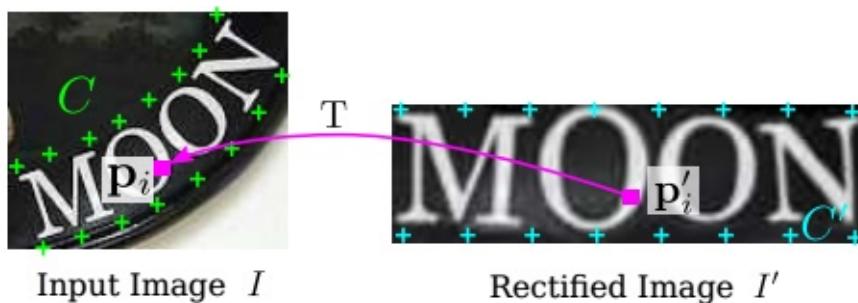
如图4所示，RARE的输入图片的尺寸是 100×32 ，在RARE的实现中，STN的输出层的Feature Map的尺寸同样使用了 100×32 的大小。

1.1.2 Grid Generator

当给定了输出Feature Map的时候，我们可以再其顶边和底边分别均匀的生成 K 个点，如图5，这些点便被叫做基-基准点（base fiducial point），表示为 $\mathbf{C}' = [\mathbf{c}'_1, \mathbf{c}'_2, \dots, \mathbf{c}'_K] \in \mathfrak{R}^{2 \times K}$ ，在RARE的STN中，输出的Feature Map的尺寸是固定的，所以 \mathbf{C}' 为一个常量。

那么，1.1.2节要介绍的Grid Generator的作用就是如何利用 C 和 C' ，将图5中的图像 I 和图5中的图像 I' 的转换关系 T 。

图5：TPS中的转换关系



当从localization network得到基准点 \mathbf{C} 和固定基-基准点 \mathbf{C}' 后，转换矩阵 $T \in \Re^{2 \times (K+3)}$ 的值已经可以确定：

$$\mathbf{T} = \left(\Delta_{\mathbf{C}'}^{-1} \begin{bmatrix} \mathbf{C}'^T \\ \mathbf{0}^{3 \times 2} \end{bmatrix} \right)^T$$

其中 $\Delta_{\mathbf{C}'} \in \Re^{(K+3) \times K+3}$ 是一个只由 \mathbf{C}' 计算得到的矩阵：

$$\Delta_{\mathbf{C}'} = \begin{bmatrix} \mathbf{1}^{K \times 1} & \mathbf{C}'^T & \mathbf{R} \\ \mathbf{0} & \mathbf{0} & \mathbf{1}^{1 \times K} \\ \mathbf{0} & \mathbf{0} & \mathbf{C}' \end{bmatrix}$$

其中 $\mathbf{1}^{K \times 1}$ 是一个 $K \times 1$ 的值全为1的行向量， $\mathbf{1}^{1 \times K}$ 同理。 $\mathbf{R} \in \Re^{K \times K}$ 是一个由 $r_{i,j}$ 组成的 $K \times K$ 的矩阵。其中

$$r_{i,j} = d_{i,j}^2 \ln(d_{i,j}^2)$$

$$d_{i,j} = \text{euclidean}(c'_i, c'_j)$$

上式中 $\text{euclidean}(a, b)$ 表示 a, b 两点之间的欧式距离。

由此可见，仅仅使用 \mathbf{C} 和 \mathbf{C}' 我们便可以得到转换矩阵 \mathbf{T} 。那么对于STN这个反向插值的算法来说，对于矫正图片中 $I' = \{\mathbf{p}'_i\}_{i=1,2,\dots,N}$ ($N = W \times H$, 即输出图像的像素点的个数) 的任意一点 $\mathbf{p}'_i = [x'_i, y'_i]^T$ ，我们怎样才能找到其在原图 I 中对应的点 $\mathbf{p}_i = [x_i, y_i]^T$ 呢？这就需要用到我们上面得到的 \mathbf{T} 了。

$$r'_{i,k} = d_{i,k}^2 \ln(d_{i,k}^2)$$

$$\hat{\mathbf{p}}'_i = [1, x'_i, y'_i, r'_{i,1}, r'_{i,2}, \dots, r'_{i,3}]$$

$$\mathbf{p}_i = \mathbf{T} \hat{\mathbf{p}}'_i$$

其中 $d_{i,k}^2$ 表示第 i 个像素点 \mathbf{p}'_i 和第 k 个基准点 c'_k 之间的欧式距离。

1.1.3 Sampler

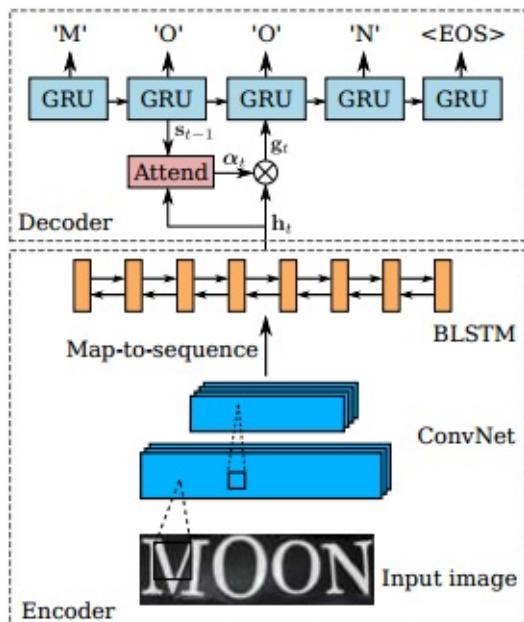
在1.1.2节中，我们得到了输出Feature Map上一点 $\mathbf{p}'_i = [x'_i, y'_i]^T$ 对应的输入Feature Map上像素点的坐标 $\mathbf{p}_i = [x_i, y_i]^T$ 的对应关系。在RARE中，使用了双线插值得到了输出Feature Map在 $[x'_i, y'_i]$ 上的值。

RARE中的STN和原始版本的STN都是一个可微分的模型，这也就意味着RARE也是一个可端到端训练的模型。不同点在于RARE将仿射变换矩阵变成了TPS，从而使模型有拥有矫正任何变换的能力，包括但不仅限于仿射变换，图2右侧部分是RARE的STN得到的实验结果。

1.2 Sequence Recognition Network

如图1的后半部分所示，RARE的SRN的输入是1.1节得到的校正后的图片，输出则是识别的字符串。SRN是一个基于Attention的序列到序列（Seq-to-Seq）的模型，包含编码器（Encoder）和解码器（Decoder）两部分，编码器用于将输入图像 I' 编码成特征向量 \mathbf{h} ，解码器则负责将特征向量 \mathbf{h} 解码成字符串 $\hat{\mathbf{y}}$ 。SRN的机构基本遵循Bahdanau在[5]中的结构，RARE的SRN的结构如图6所示。

图6：SRN框架图



1.2.1 编码器（Encoder）

RARE的编码器非常简单由一个7层的CNN和一个两层的双向LSTM组成。根据论文中给出的结构，以及1.1节确定的STN的输出层的大小(100×32)，Encoder的结构如图7所示。

图7：SRN Encoder网络结果即输出Feature Map的尺寸

Type	Filters	Size	Padding	Stride	Output
Convolution	64	3*3	1	(1,1)	100*32
Max-pooling	64	2*2	0	(2,2)	50*16
Convolution	128	3*3	1	(1,1)	50*16
Max-pooling	128	2*2	0	(2,2)	25*8
Convolution	256	3*3	1	(1,1)	25*8
Convolution	256	3*3	1	(1,1)	25*8
Max-pooling	256	2*1	0	(2,1)	25*4
Convolution	512	3*3	1	(1,1)	13*4
Convolution	512	3*3	1	(1,1)	13*4
Max-pooling	512	2*1	0	(2,1)	25*2
Convolution	512	2*2	0	(1,1)	24*1
Map-to-Sequence	-	-	-	-	24*512
B-LSTM	-	-	-	-	24*256*2
B-LSTM	-	-	-	-	24*256*2

注意此处论文中有点没有说明，通过和作者的讨论得知，最后两层只对高度进行降采样，因此才有了图7中的结构。

在卷积层之后，Encoder设置了两个双向LSTM，每个LSTM的隐层节点的数量都是256，第 t 个时间片的输出特征为 \mathbf{x}_t ，第 t 个时间片的正向LSTM的隐层节点为 \mathbf{h}_t^f ，反向LSTM的隐节点为 \mathbf{h}_t^b ， $f()$ 表示一个LSTM节点，则正向和反向传播可分别表示为：

$$\mathbf{h}_t^f = f(\mathbf{x}_t, \mathbf{h}_{t-1}^f)$$

$$\mathbf{h}_t^b = f(\mathbf{x}_t, \mathbf{h}_{t+1}^b)$$

上式中的 h_0 以及 h_7 可以自己定义或者用默认的0值。

Encoder的输出是正反向两个隐层节点拼接起来，这样每个时间片的特征向量的个数便是512：

$$\mathbf{h} = [\mathbf{h}_t^f; \mathbf{h}_t^b]$$

卷积之后 $W_{conv} = 6$ ，Encoder的输出特征序列 \mathbf{h} 由所有时间片拼接而成，因此

$\mathbf{h} = (\mathbf{h}_1, \dots, \mathbf{h}_L) \in \mathfrak{R}^{512 \times L}$ ，其中 $L = W_{conv} = 6$ 。

1.2.2 解码器 (Decoder)

Decoder是基于单向GRU的序列模型，其在第 t 个时间片的特征 \mathbf{s}_t 表示为：

$$\mathbf{s}_t = \text{GRU}(l_{t-1}, \mathbf{g}_t, \mathbf{s}_{t-1})$$

其中 $t = [1, 2, \dots, T]$ ， T 是输出标签的长度。

在训练时， l_{t-1} 是第 t 个时间片的标签，在测试时则是第 t 个时间片的预测结果。 \mathbf{g}_t 是Attention的一个叫做glimpse的参数，从数学上理解是特征 \mathbf{h} 的各个时间片的特征的加权和：

$$g_t = \sum_{i=1}^L \alpha_{ti} \mathbf{h}_i$$

$$\alpha_{ti} = \frac{\exp(\tanh(s_{i-1}, \mathbf{h}_t))}{\sum_{k=1}^T \exp(\tanh(s_{i-1}, \mathbf{h}_k))}$$

RARE的输出向量有37个节点，包括26个字母+10个数字+1个终止符，输出层使用softmax做激活函数，每个时间片预测一个值：

$$\hat{\mathbf{y}}_t = \text{softmax}(\mathbf{W}^T \mathbf{s}_t)$$

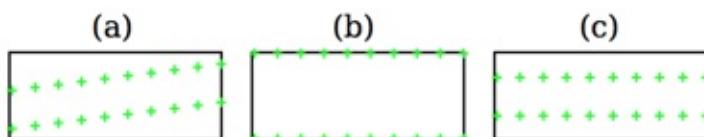
1.3 训练

RARE是一个端到端训练的模型，我们不需要STN专门的标签也可以训练它，所以损失函数仅是一个简单的log极大似然：

$$\mathcal{L} = \sum_{i=1}^N \log \prod_{t=1}^{|I^{(i)}|} p(l_t^{(i)} | I^{(i)}; \theta)$$

为了提高STN的收敛速度，作者使用了图8的三种方式和书籍初始化的方式初始化基准点，其中(a)的收敛效果最好：

图8：基准点的几种初始化方式



1.4 基于词典的测试

在测试时一个最简答的策略就是在每个时间片都选择概率最高的作为输出。另外一种方式是根据字典构建一棵先验树来缩小字符的搜索范围。使用字典时有贪心搜索和beam search搜索两个思路，在实验中，作者选择了宽度为7的beam search。

总结

RARE的特点是将STN和TPS结合起来使STN理论上具有矫正任何形变的能力，创新性和技术性上都非常值得参考。但是结合了TPS的STN带来的效果提升并没有设想的那么好，实验结果可以参考论文中给出的几个图，原因可能是问题本身的复杂性。但是稍微的一些纠正效果总比没有强，毕竟STN带来的速度损失还是很小的。

Reading Text in the Wild with Convolutional Neural Networks

tags: OCR, CNN

前言

这篇论文[\[19\]](#)出自著名的牛津大学计算机视觉组（Visual Geometry Group），没错，就是发明VGG网络的那个实验室。这篇论文是比较早的研究端到端文字检测和识别的经典算法之一。参考文献显示文章发表于2016年，但是该论文于2014年就开始投稿，正好也是那一年物体检测的开山算法R-CNN发表。

论文的算法在现在看起来比较传统和笨拙，算法主要分成两个阶段：

1. 基于计算机视觉和机器学习的场景文字检测；
2. 基于深度学习的文本识别。

虽然论文说自己是端到端的系统，但是算法的阶段性特征是非常明显的，并不是纯粹意义上的端到端。这里说这些并不是要否定这篇文章的贡献，结合当时深度学习的发展条件，算法涉及成这样也是可以理解的。

虽然方法比较笨拙，但是作为OCR领域的教科书式的文章，这篇文章还是值得一读的。

算法详解

1. 候选区域生成

算法中候选区域生成是采用的两种方案的并集， $B = \{B_e \cup B_d\}$ 它们分别是Edge Boxes [\[43\]](#) 和Aggregate Channel Feature Detector [\[18\]](#)。由于本书更偏重于深度学习领域的知识，对于上面两个算法只会粗略介绍方法，详细内容请自行查看参考文献中给出的论文。

1.1 Edge Boxes

Edge boxes的提出动机是bounding box内完全包含的的轮廓越多（Contours），那么该bounding box为候选区域的概率越高，这种特征到文字检测方向尤为明显。

Edge boxes的计算步骤如下：

1. 边缘检测；

2. 计算Edge Group：通过合并近似在一条直线上的边缘点得到的；
3. 计算两个Edge Group之间的亲密度（affinity）：

$$a(s_i, s_j) = |\cos(\theta_i, \theta_{i,j}) \cos(\theta_j, \theta_{ij})|^\gamma$$

其中 γ 为超参数，一般设置为 2。 (θ_i, θ_j) 是两组 Edge Group 的平均旋转角度， θ_{ij} 是两组 edge boxes 的平均位置 x_i, x_j 的夹角。

1. 计算 edge group 的权值：

$$w_b(s_i) = 1 - \max_T \prod_j^{|T|-1} a(t_j, t_{j+1})$$

1. 计算最终评分：

$$h_b = \frac{\sum_i w_b(s_i) m_i}{2(b_w + b_h)^\kappa}$$

其中 bounding box 通过多尺寸，多比例的滑窗方式得到。

计算完评分之后，通过 NMS 得到最终的候选区域， B_e 。

1.2 Aggregate Channel Feature Detector

该方法的核心思想是通过 AdaBoost 集成多尺度的 ACF 特征。ACF 特征不具有尺度不变性，而使用多尺度输入图像计算特征的方法又过于耗时，[4] 中只在每个 octave 重采样图像计算特征，每个 octave 之间的特征使用其它尺度进行估计：

$$C_s = R(C_{s'}, s/s')(s/s')^{-\lambda\Omega}$$

最后在通过 AdaBoost 集成由决策树构成的若分类器，通过计算阈值的方式得到最终的候选区域， B_d 。

2. 候选区域的精校

通过第一节的方法得到候选区域后，作者对这些候选区域进行了进一步的精校，包括对候选区域是否包含文本的二分类和 bounding box 位置的调整。

2.1 word/no word 分类

作者通过从训练集上采样得到了一批候选区域，其中和Ground Truth的overlap大于0.5的设为正样本，小于0.5的设为负样本。提取了候选区域的Hog特征，并使用随机森林分类器训练了一个判断候选区域是否包含文本的二分类的分类器。随机森林包括10棵决策树，每棵树的最大深度为64。

2.2 Bounding box回归

Bounding box回归器是通过四层CNN加上两层全连接得到的，四层卷积层卷积核的尺寸和个数分别是 $\{5, 64\}, \{5, 128\}, \{3, 256\}, \{3, 512\}$ ，全连接隐层节点的个数是 $4k$ ，网络有四个输出，分别用于预测bounding box的左上角和右下角坐标 $b = \{x_1, y_1, x_2, y_2\}$ 。损失函数则是使用的 L_2 损失函数：

$$\min_{\Phi} \sum_{b \in B_{brain}} \|g(I_b; \Phi) - q(b_{gt})\|_2^2$$

其中 Φ 表示由训练集得到的参数。

作者也尝试了基于CNN的分类和回归的多任务模型，但是效果并不如基于Hog特征的随机森林分类器。

3. 文本识别

OCR中，有基于字符的时序序列分类器和基于单词的分类器。前者的优点是分类器类别数目少(26类)，缺点是序列模型识别困难，一个字符错误导致整个单词的识别失败，基于序列的建模要比简单的分类任务复杂得多。后者的优点是无时序特征，缺点是类别数目太多，穷举所有类别不太可能，如果采用基于抽样的方法的话，抽样类别太少导致在测试时遇见未定义的单词的概率过高，抽样类别多的话导致训练集样本不够，不足以覆盖所有类别。

在这里，作者使用的是第二种方案，也就是基于单词的分类方法。为了解决未定义单词的问题，作者采样了90k个最常见的单词，样本不足的问题则是通过合成数据解决的。

3.1 合成数据

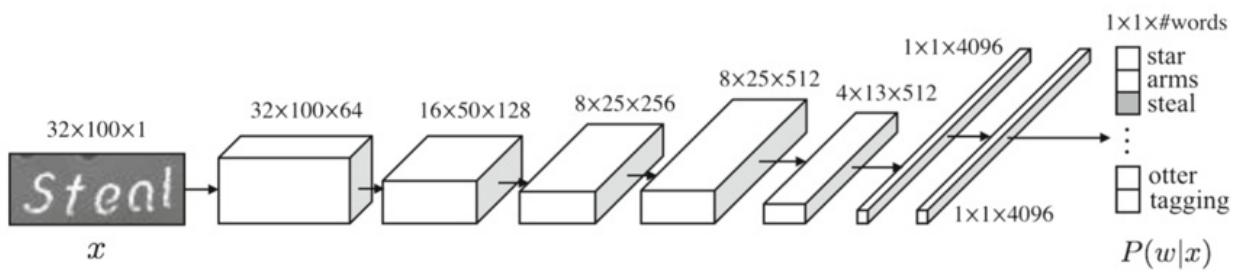
合成模型使用了90k的单词作为合成文本，在图像变化上，考虑了以下几点：

- 字体：从Google Fonts中下载了1400个不同的字体，文本随机随机添加下划线和弯曲效果；
- 边界，阴影；
- 文字颜色；
- 仿射变换；
- 噪音；
- 混合真实数据。

3.2 分类模型

作者使用了CNN训练识别模型，根据上面的描述。CNN是一个90k的多分类模型，网络具体参数见图1。

图1：Read Text in the Wild的识别网络



图片 resize 作者使用了 wrap 的方式，添加 padding 通过实验得知效果并不如 wrap。分类函数则使用了 softmax 损失函数。

总结

从这篇算法使用的技术可以看出这篇论文作为早期的深度学习的文章，传统方法和深度学习方法都在扮演着非常重要的作用。

文字检测部分依旧遵循 R-CNN 系列的 two-stage 思路，即基于 Edge boxes 和 ACF 的候选区域提取和基于随机森林和 CNN 的位置精校。

而分类作者则是使用了最为传统的 CNN 分类模型，并没有添加 RNN 等擅长处理序列特征的结构。

Deep TextSpotter: An End-to-End Trainable Scene Text Localization and Recognition Framework

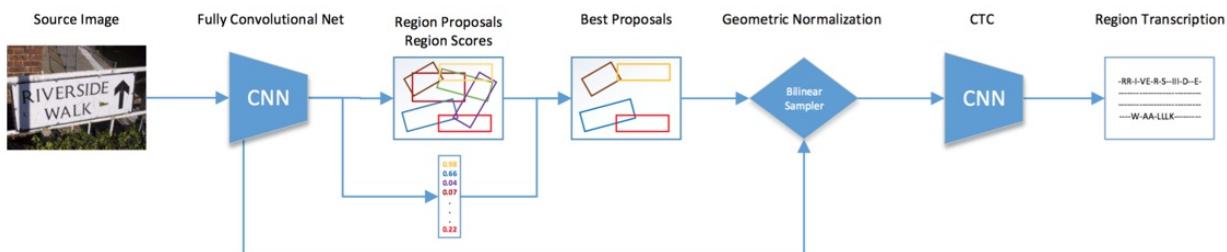
tags: Deep TextSpotter, OCR, YOLOv2, STN, CTC

前言

Deep TextSpotter[17]的创新点并不多，基本上遵循了传统OCR或者物体检测的两步走的流程（图1），即先进行场景文字检测，再进行文字识别。在这个算法中，检测模块基于YOLOv2[55]，识别模块基于STN[41]，损失函数则使用了精度的CTC[77]。这几个算法在当时都是state-of-the-art的，因此其效果达到了最优也不难理解。这三个知识点已分别在本书的第四章，第五章和第二章进行了解析，算法细节可参考具体内容或者阅读论文。这里不在对上面三个算法的细节再做重复，只会对Deep TextSpotter的流程做一下梳理和解释。

Deep TextSpotter的一个创新点是将NMS放到了识别之后，使用识别置信度替代了传统的检测置信度。

图1：Deep TextSpotter算法流程



1. Deep TextSpotter解析

1.1 全卷积网络

为什么使用YOLOv2：在YOLOv2的文章中我们讲过，YOLOv2使用了高分辨率的迁移学习提高了网络对高分辨率图像的检测效果，这个能力在端到端的文字检测及识别中非常重要。因为过分的降采样将造成文本区域的识别问题。

Feature Map的尺寸：网络的框架也采样去YOLOv2中在 3×3 卷积中插入 1×1 卷积进行非线性化的结构。对于一张尺寸为 $W \times H$ 的输入图像，在网络中会通过5个Max Pooling进行降采样，得到尺寸为 $\frac{W}{32} \times \frac{H}{32}$ 的的Feature Map。在Deep TextSpotter中，每隔20个Epoch会更换一次输入图像的尺寸，尺寸的变化范围是 $\{352, 416, 480, 544, 608\}$

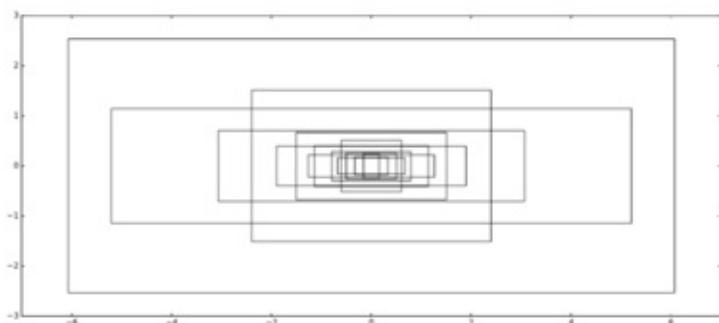
全卷积：Deep TextSpotter使用了Global Average Pooling代替全连接实现非线性化，从而使网络成为全卷积网络，原因已多次提及：保留特征向量的位置信息。

1.2 候选区域提取

输出向量：Deep TextSpotter的检测部分预测了6个值，它们分别是坐标 r_x , r_y ，尺寸 r_w , r_h ，检测置信度 r_p 以及比YOLOv2增加的一个旋转角度 r_θ 。其中角度使用了弧度值，即 $\theta \in (-\frac{\pi}{2}, \frac{\pi}{2})$ 。其它几个预测值则采用了YOLOv2中使用的预测相对值。

锚点聚类：Deep TextSpotter将锚点聚了14类，锚点形状见图2。

图2：Deep TextSpotter聚类得到的锚点



样本采样：匹配正负锚点时使用了YOLOv1中介绍的bipartite策略，即只有和Ground Truth的IOU最大的锚点为正样本，其余均为负样本。

NMS：Deep TextSpotter的一个创新点在于并没有这检测完之后就使用NMS，考虑到的一个问题是只覆盖部分文字区域的检测框的置信度有可能高于检测到完整文字区域的置信度要高。在这里，只使用阈值 $\theta = 0.1$ 过滤掉部分置信度非常低的样本。

1.3 双线性插值

经过YOLOv2得到的检测框的尺寸，角度，比例等都是不同的，为了产生长度固定的特征向量。Faster R-CNN等方法采用的是ROI Pooling，Deep TextSpotter则是使用STN的策略，STN不仅能产生长度固定的特征向量，还能学到图像的仿射变换矩阵，是非常适用于OCR领域的。

Deep TextSpotter产生的是长宽比不变，宽度固定位 $H' = 32$ 的 Feature Map，即对于一个检测到的区域 $U \in R^{w \times h \times C}$ ，其得到的 Feature Map 的为 $V \in R^{\frac{wH'}{h} \times H' \times C}$ 。

Feature Map 中位置 (x', y') 处的值为：

$$V_{x',y'}^c = \sum_{x=1}^w \sum_{y=1}^h \mathbf{U}_{x,y}^c \kappa(x - \mathcal{T}_x(x')) \kappa(y - \mathcal{T}_y(y'))$$

这个过程便是双线性插值，不理解的参考我在 STN 的解释。

其中 (x, y) 为检测框中的一点， (x', y') 为输出 Feature Map 上的一点，范围是

$x' \in [0, \frac{wH'}{h} - 1]$, $y' \in [0, H' - 1]$ 。 $\mathcal{T}(\cdot)$ 为位置转移函数，应该是仿射变换矩阵。

$\kappa(v) = \max(0, 1 - |v|)$ 为双线性插值函数。

在 STN 中我们讲过其双线性插值是可导的，因此到目前为止该过程是端到端的。

1.4 文本识别

Deep TextSpotter 使用的是基于字符序列识别方式，骨干网络使用的是基于图 3 的全卷积网络。网络支持宽的变长输入，但是高是固定的。图 3 中的 Recurrent Convolution 猜测是使用 [15] 的结构，论文中没有给出注释。

损失函数使用的是 CTC，参考第二章 CTC 一节，此处不再废话。

图 3 : Deep TextSpotter 识别部分的全卷积网络

Type	Channels	Size/Stride	Dim/Act
input	C	-	$W \times 32$
conv	32	3×3	leaky ReLU
conv	32	3×3	leaky ReLU
maxpool		$2 \times 2/2$	$\bar{W}/2 \times 16$
conv	64	3×3	leaky ReLU
BatchNorm			
recurrent conv	64	3×3	leaky ReLU
maxpool		$2 \times 2/2$	$\bar{W}/4 \times 8$
conv	128	3×3	leaky ReLU
BatchNorm			
recurrent conv	128	3×3	leaky ReLU
maxpool		$2 \times 2/2 \times 1$	$\bar{W}/4 \times 4$
conv	256	3×3	leaky ReLU
BatchNorm			
recurrent conv	256	3×3	leaky ReLU
maxpool		$2 \times 2/2 \times 1$	$\bar{W}/4 \times 2$
conv	512	3×2	leaky ReLU
conv	512	5×1	leaky ReLU
conv	$ \hat{\mathcal{A}} $	7×1	$\bar{W}/4 \times 1$
log softmax			

1.5 NMS

在检测中搁置的NMS将在识别完之后在使用，NMS中使用的置信度是识别的置信度，阈值给的是0.5。

总结

由于使用了STN连接检测和识别，Deep TextSpotter是一个真正的端到端模型，所以在训练的过程中，只需要针对分类的loss进行训练。

Deep TextSpotter，算法最核心的部件是STN，但是并没有cite该论文，识别中的RCNN也没cite，进而导致了理解上的困难，这毛病可不好。

实例解析：字符验证码破解

前言

如果MNIST数据集是计算机视觉届的“Hello, World”的话，那么破解字符验证码就是计算机视觉届的99乘法表。目前市场上的字符验证码一般是由大小写英文字母和阿拉伯数字组成，最常见的是4个字符，下面我们来一步步解析如何使用前面介绍的知识一步步破解字符验证码，尝试了近10个不同得网站，平均精度能够控制到99%上下。在文章中出于法律问题的原因，并不会列举破解了哪些公司的验证码。

1. 综述

破解一个网站的验证码主要分成以下步骤：

1. 数据爬取和标注；
2. 模型搭建和数据预处理；
3. 合成数据和迁移学习；
4. n 折交叉数据清洗；
5. 模型fine-tune。

以上的算法的开发环境我使用的是python 2.7.6 + keras 2.1.2 (TensorFlow做后端)，若使用截止最新的版本 python 3.7.0及keras 2.2.0，代码需要进行语法的微调后便可运行。

1.1 数据爬取和标注

破解一个网站的验证码，标注数据样本是必不可少的一步。为了减轻标注数据的成本，我一般标注11500张数据，其中10000张作为训练集，1000张作为测试集，500张作为验证集。标注数据可以采用亚马逊等平台提供的众包工具，当然不怕浪费时间的话也可以自己标。注意外包出去的验证码由于各种各样的原因导致存在标注错误的情况，一般用户常犯两种错误：

1. 近似字符的标注错误，例如‘U’和‘V’等，我一般叫这种错误为眼花错误；
2. 键盘临近字符敲错，例如‘S’和‘D’等，我一般叫做手抖错误。

清理训练集错误数据是将模型能力提升至接近100%的至关重要的一步，我们在1.4节介绍如何使用模型帮助我们清理训练集数据。但是对于验证集我建议还是自己手动清理，原因有三点：1. 测试集加验证集数据并不多，一般两三个小时便可手动清理完毕；2. 清理的过程中，我们顺便观察验证码的样式，为下一步迁移学习积累知识；3. 1.4节使用的 n 折数据交叉清洗的方法并不能保证100%清理干净。

1.2 模型搭建

我一般使用CNN+GRU+CTC损失的网络结构，这也是OCR场景中最流行的框架结构，其中我在第一章进行了CNN的介绍，GRU和CTC在第二章进行了介绍，算法详解参考具体章节的内容。

1.2.1 CNN

一般验证码的场景比较简单，使用简单的VGG模式便可以解决。即

$$m \times (n \times (conv_33) + max_pooling)$$

其中n是根据验证码的复杂程度决定，一般复杂程度表示的是验证码的干扰因素的多少，我们在1.3节会手动合成样本，验证码的复杂度你会在进行这一过程时深刻的感受到。一般n为2-4之间。

m的大小取决于验证码最短边的长度，我们在第一章介绍过，每次pooling便进行一次降采样，此时Feature Map的大小会减半。在输入RNN之前，对于验证码这种简单的场景，最后一层的Feature Map的尺寸在5-10之间是一个优先选择的尺寸。目前市场上的主流验证码的最短边（高度）一般在30-100之间，所以m优先取值2到5。

1.2.2 GRU和CTC

在经过CNN得到Feature Map之后，我们需要将Feature Map展开成一个向量，然后经过一个全连接的编码之后作为GRU的输入。全连接我一般使用的节点数目为32。

关于GRU这一部分介绍的并不多，在目前的所有网站中，我使用的均是两层双向GRU结构，其中隐节点数量为128。

1.2.3 分类函数

在GRU之后接的是使用softmax激活函数的全连接层。由于只有10000张训练数据，所以为了减轻轻过拟合，推荐添加Dropout，一般丢失率我设置为0.25。

为了提高精度，softmax一般不使用全部62个字母和数字字符，而是由训练集的标注数据统计得到。一般验证码字符的分布属于均匀分布，对于10000张训练集，一个字符平均出现1000-2000次，此时要注意那些出现概率很低的字符，极有可能是标注错误的样本。用户在标注样本时，可能犯的错误很多，推荐在统计字符时使用try...except...捕捉异常情况。

1.2.4 CTC

最后使用CTC构造损失函数。要非常注意的一点是Keras封装的decode函数存在内存泄漏的问题，长时间训练或者测试时会产生内存不足的问题。这里我根据单独重现了CTC的Beam Search函数，读者可以网上自行搜索相关源码。

代码片段1：网络模型keras实现

```
#hypter parameters
m = 2
n = 2
rnn_size = 128

input_tensor = Input((width, height, 3))
x = input_tensor
for i in range(m):
    for j in range(n):
        x = Convolution2D(32, (3, 3), activation='relu')(x)
        x = MaxPooling2D(pool_size=(2, 2))(x)

conv_shape = x.get_shape()
x = Reshape(target_shape=(int(conv_shape[1]), int(conv_shape[2] * conv_shape[3])))(x)
x = Dense(32, activation='relu')(x)

gru_1 = GRU(rnn_size, return_sequences=True, kernel_initializer='he_normal', name='gru_1')(x)
gru_1b = GRU(rnn_size, return_sequences=True, go_backwards=True, kernel_initializer='he_normal', name='gru1_b')(x)
gru1_merged = add([gru_1, gru_1b])
gru_2 = GRU(rnn_size, return_sequences=True, kernel_initializer='he_normal', name='gru_2')(gru1_merged)
gru_2b = GRU(rnn_size, return_sequences=True, go_backwards=True, kernel_initializer='he_normal', name='gru2_b')(
    gru1_merged)
x = concatenate([gru_2, gru_2b])
x = Dropout(0.25)(x)
x = Dense(n_class, kernel_initializer='he_normal', activation='softmax')(x)
base_model = Model(input=input_tensor, output=x)

labels = Input(name='the_labels', shape=[n_len], dtype='float32')
input_length = Input(name='input_length', shape=[1], dtype='int64')
label_length = Input(name='label_length', shape=[1], dtype='int64')
loss_out = Lambda(ctc_lambda_func, output_shape=(1,), name='ctc')([x, labels, input_length, label_length])

model = Model(input=[input_tensor, labels, input_length, label_length], output=[loss_out])
```

1.3 合成数据和迁移学习

当训练样本不足时，先使用迁移学习进行模型初始化再使用标注数据进行微调是目前最主流的算法。而迁移学习的样本推荐手动合成和标注样本尽可能像的数据，这样有点有三：

1. 和标注样本相似的合成样本能达到比较好的初始化效果，甚至合成样本的精度有可能超过90%；
2. 当任务比较复杂时，推荐每次训练样本均是重新合成的，这样由于每个训练样本均不相同，减轻了过拟合的问题。
3. 验证码的样式比较简单，合成算法并不会非常复杂。

常见的验证码随机样式包括，

1. 文字的随机，一般包括随机字符，随机大小，随机字体，随机颜色，随机位置，随机旋转角度，文字扭曲；
2. 背景的随机，一般使用随机颜色，随机背景图片；
3. 噪音干扰，一般有噪点干扰和干扰线；

合成数据一般由反向预处理和正向合成效果组成。

1.3.1 反向预处理

常见的预处理操作有

1. 二值化减少字符和背景颜色的干扰；
2. 滤波器过滤噪音；
3. 边缘增强加强文字和背景的对比度。

此处验证码便发挥非常重要的作用了，因为我们在训练和测试数据中也会采用相同的预处理方案，所以我们必须确保预处理在500张验证码上均起到了正向的作用，否则此预处理操作便不能直接使用。

图1：某网站验证码和预处理效果图



1.3.2 正向合成

此处为代码量最大的一部分，即自己制作和训练样本尽可能相似的样本，常用的python包有OpenCV和Pillow，一般的变化通过随机数便可以解决，在我接触的验证码中，有两类变化比较难模拟——扭曲的文字和光滑的干扰线。对于扭曲的文字，我们可以使用对文字区域施加波形变化得到，对于光滑曲线，这类曲线叫做贝泽尔曲线，关于波形变化和贝泽尔曲线我会在附录A中给出代码片段。

图2：某网站验证码及合成的验证码



1.3.3 迁移学习

对于使用合成数据进行迁移学习来说，一般有两种方案：

1. 边合成边学习：这种方法的优点是每次的训练样本都不一样，学习的模型不容易过拟合。问题是由于合成使用的是CPU，这一阶段往往成为性能瓶颈而不能充分发挥GPU的作用，从而导致收敛时间过长。
2. 先合成数据再学习：这种方法的优点是模型迭代的快，一般是方案1的几十倍。缺点是合成数据需要占用硬盘空间，尤其是硬盘节点数，而且有过拟合的潜在问题。

对于上面两种方案，我推荐方案2。因为在使用合成数据进行迁移学习之后，我们要继续使用标注数据进行微调，所以迁移学习的初始化并不是决定模型的关键，一定程度的误差是可以允许的。一般在任务2中，我会合成50万到100万张数据，然后loss收敛到小数点后两位即可。或者在睡觉时将服务挂着，第二天初始化的模型便可以用了。

1.4 n折交叉数据清洗

迁移学习之后便可以进行模型微调了，此时一般可以得到一个95%-97%准确率的模型，此时向上提升精度便变得非常困难。因此为了得到更高的精度，清洗训练数据便成了不可避免的一步。这里，我根据n折交叉验证的思想，设计了使用训练集清理自身数据的方法，所以该方法便叫做n折交叉数据清洗。

所谓n折交叉数据清洗，即将训练集随机分成个子集 S 。分别使用第 i 个子集 $s_i \in S$ 作为测试集，其它子集共同作为训练集。在迁移学习得到的模型的基础上独立训练n个模型，

$M = \{m_1, m_2, \dots, m_n\}$ 。然后使用 m_i 测试 s_i ，这时绝大多数的错误样本便会被检测出来，尤其是由于“手抖错误”导致的错误样本。最后根据模型的测试效果用户手动检查并清洗错误样本即可。

这样做的原因是用户标注错误 p_1 和模型识别错误 p_2 可近似看做两个相互独立的事件，那么一个样本同时被标注错误和识别错误的概率为 $p_1 \times p_2$ ，期望错误样本数便是 $10000 \times p_1 \times p_2$ 。假设 $p_1 \approx 0.05$, $p_2 \approx 0.05$ ，即有 $p_1 \times p_2 \approx 0.0025$ ，那么此时10000张训练集中的错误样本的期望便只有25张了。

为了进一步确保训练集的准确率，可以再使用一次n折交叉数据清洗，不过一般我只使用一次。

1.5 模型fine-tune

得到清洗后的训练数据后，便可以使用这些数据进行模型微调了，由于训练样本比较少而且比较干净，一般几个小时就收敛了。中间可以隔一段时间就保存一个模型，由于10000个训练样本非常容易导致过拟合，所以我们通过验证集选择一个早停版本的模型。

附录A

代码片段2：波形变换

```

def gene_wave(img, row, col):
    channel = 3
    img = img_as_float(img)
    img_out = img * 1.0
    alpha = 70.0
    beta = 30.0
    degree = 3.0

    center_x = (col - 1) / 2.0
    center_y = (row - 1) / 2.0

    xx = np.arange(col)
    yy = np.arange(row)

    x_mask = numpy.matlib.repmat(xx, row, 1)
    y_mask = numpy.matlib.repmat(yy, col, 1)
    y_mask = np.transpose(y_mask)

    xx_dif = x_mask - center_x
    yy_dif = center_y - y_mask

    x = degree * np.sin(2 * math.pi * yy_dif / alpha) + xx_dif
    y = degree * np.cos(2 * math.pi * xx_dif / beta) + yy_dif

    x_new = x + center_x
    y_new = center_y - y

    int_x = np.floor(x_new)
    int_x = int_x.astype(int)
    int_y = np.floor(y_new)
    int_y = int_y.astype(int)

    for ii in range(row):
        for jj in range(col):
            new_xx = int_x[ii, jj]
            new_yy = int_y[ii, jj]

            if x_new[ii, jj] < 0 or x_new[ii, jj] > col - 1:
                continue
            if y_new[ii, jj] < 0 or y_new[ii, jj] > row - 1:
                continue

            img_out[ii, jj, :] = img[new_yy, new_xx, :]

    img_out = Image.fromarray(np.uint8((img_out) * 255))
    return img_out

```

代码片段3：贝泽尔曲线

```

# bazier 曲线
def make_bezier(xys):

```

```

# xys should be a sequence of 2-tuples (Bezier control points)
n = len(xys)
combinations = pascal_row(n - 1)

def bezier(ts):
    # This uses the generalized formula for bezier curves
    # http://en.wikipedia.org/wiki/B%C3%A9zier_curve#Generalization
    result = []
    for t in ts:
        tpowers = (t ** i for i in range(n))
        upowers = reversed([(1 - t) ** i for i in range(n)])
        coefs = [c * a * b for c, a, b in zip(combinations, tpowers, upowers)]
        result.append(
            tuple(sum([coef * p for coef, p in zip(coefs, ps)]) for ps in zip(*xys
)))
    return result

return bezier

def pascal_row(n):
    # This returns the nth row of Pascal's Triangle
    result = [1]
    x, numerator = 1, n
    for denominator in range(1, n // 2 + 1):
        # print(numerator,denominator,x)
        x *= numerator
        x /= denominator
        result.append(x)
        numerator -= 1
    if n & 1 == 0:
        # n is even
        result.extend(reversed(result[:-1]))
    else:
        result.extend(reversed(result))
    return result

# 用来绘制干扰线
def gene_line(image):
    (width, height) = size
    rand = random.random()
    if rand < 0.5:
        line_color = (80, 90, 90)
    else:
        line_color = (0, 0, 12)
    xys = []
    for i in range(0, 4):
        xys.append((20 + i * 140, random.randint(0, height)))
    ts = [t / 100.0 for t in range(101)]
    bezier = make_bezier(xys)
    points = bezier(ts)
    draw = ImageDraw.Draw(image) # 创建画笔

```

```
draw.line(points, fill=line_color, width=5)  
return image
```

二维识别

二维图像识别包括多行文本识别，表格识别，公式识别甚至Image Caption等，它们的共同特点是图像的内容即要考虑左右顺序关系，同时也要考虑上下内容信息，因此不能再采用CTC作为损失函数。在深度学习中，二维识别的算法一般也是采用“编码器-解码器”的网络结构，这里的解码器叫做“生成器”其实更符合网络特征。在二维图像识别中，编码器一般采用CNN组成，用于将图像编码成特征向量，而解码器（或生成器）一般由RNN组成，用于根据图像的特征生成对应的标签，这个标签可以使文本的内容，或者是图像的描述，或者是公式的Latex代码等。

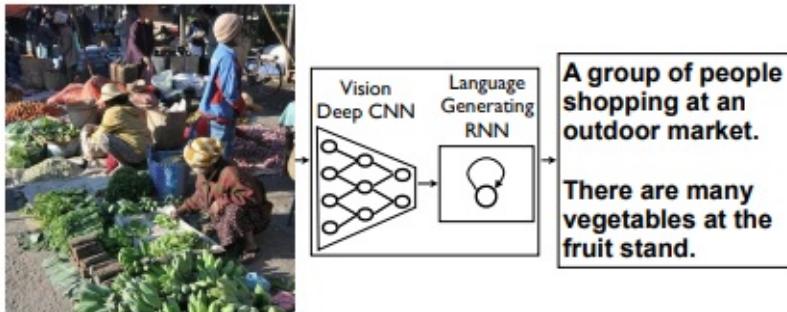
Show and Tell: A Neural Image Caption Generator

前言

图像描述（Image Caption）是非常经典的图像二维信息识别的一个英语。类似的还有表格识别，公式识别等。如图1所示，Image Caption的输入时衣服图像，输出是对这个图像的描述。Image Caption的难点有二：

1. 模型不仅要能够对图像中的每一个物体进行分类，还需要能够理解和描述它们的空间关系。
2. 描述的生成要考虑语义信息，当前的输出高度依赖之前生成的内容。

这篇论文提供了一个Image Caption的基础框架：即用CNN作为特征提取器用于将图像转换为特征向量，之后用一个RNN作为解码器（生成器），用于生成对图像的描述。



Show and Tell详解

Image Caption也是采用的Encoder-Decoder的算法框架，作者当初设计这个算法的时候，也是借鉴了神经机器翻译的思想，故而采用了类似的网络架构。

和机器翻译类似，Image Caption的目标也是最大化标签值得概率，这里的标签即使训练集的描述内容 S ，表示为：

$$\theta^* = \arg \max_{\theta} \sum_{(I, \theta)} \log pS|I; \theta$$

U-Net: Convolutional Networks for Biomedical Image Segmentation

tags: U-Net, Semantic Segmentation

前言

U-Net[14]是比较早的使用全卷积网络进行语义分割的算法之一，论文中使用包含压缩路径和扩展路径的对称U形结构在当时非常具有创新性，且一定程度上影响了后面若干个分割网络的设计，该网络的名字也是取自其U形形状。

U-Net的实验是一个比较简单的ISBI cell tracking数据集，由于本身的任务比较简单，U-Net紧紧通过30张图片并辅以数据扩充策略便达到非常低的错误率，拿了当届比赛的冠军。

论文源码已开源，可惜是基于MATLAB的Caffe版本。虽然已有各种开源工具的实现版本的U-Net算法陆续开源，但是它们绝大多数都刻意回避了U-Net论文中的细节，虽然这些细节现在看起来已无关紧要甚至已被淘汰，但是为了充分理解这个算法，笔者还是建议去阅读作者的源码，地址如下：<https://lmb.informatik.uni-freiburg.de/people/ronneber/u-net/>

1. 算法详解

1.1 U-Net的网络结构

直入主题，U-Net的U形结构如图1所示。网络是一个经典的全卷积网络（即网络中没有全连接操作）。网络的输入是一张 572×572 的边缘经过镜像操作的图片（input image tile），关于“镜像操作”会在1.2节进行详细分析，网络的左侧（红色虚线）是由卷积和Max Pooling构成的一系列降采样操作，论文中将这一部分叫做压缩路径（contracting path）。压缩路径由4个block组成，每个block使用了3个有效卷积和1个Max Pooling降采样，每次降采样之后Feature Map的个数乘2，因此有了图中所示的Feature Map尺寸变化。最终得到了尺寸为 32×32 的Feature Map。

网络的右侧部分（绿色虚线）在论文中叫做扩展路径（expansive path）。同样由4个block组成，每个block开始之前通过反卷积将Feature Map的尺寸乘2，同时将其个数减半（最后一层略有不同），然后和左侧对称的压缩路径的Feature Map合并，由于左侧压缩路径和右侧扩展路径的Feature Map的尺寸不一样，U-Net是通过将压缩路径的Feature Map裁剪到和扩展路径相同尺寸的Feature Map进行归一化的（即图1中左侧虚线部分）。扩展路径的卷积操作依旧使用的是有效卷积操作，最终得到的Feature Map的尺寸是 388×388 。由于该任务是一个二分类任务，所以网络有两个输出Feature Map。

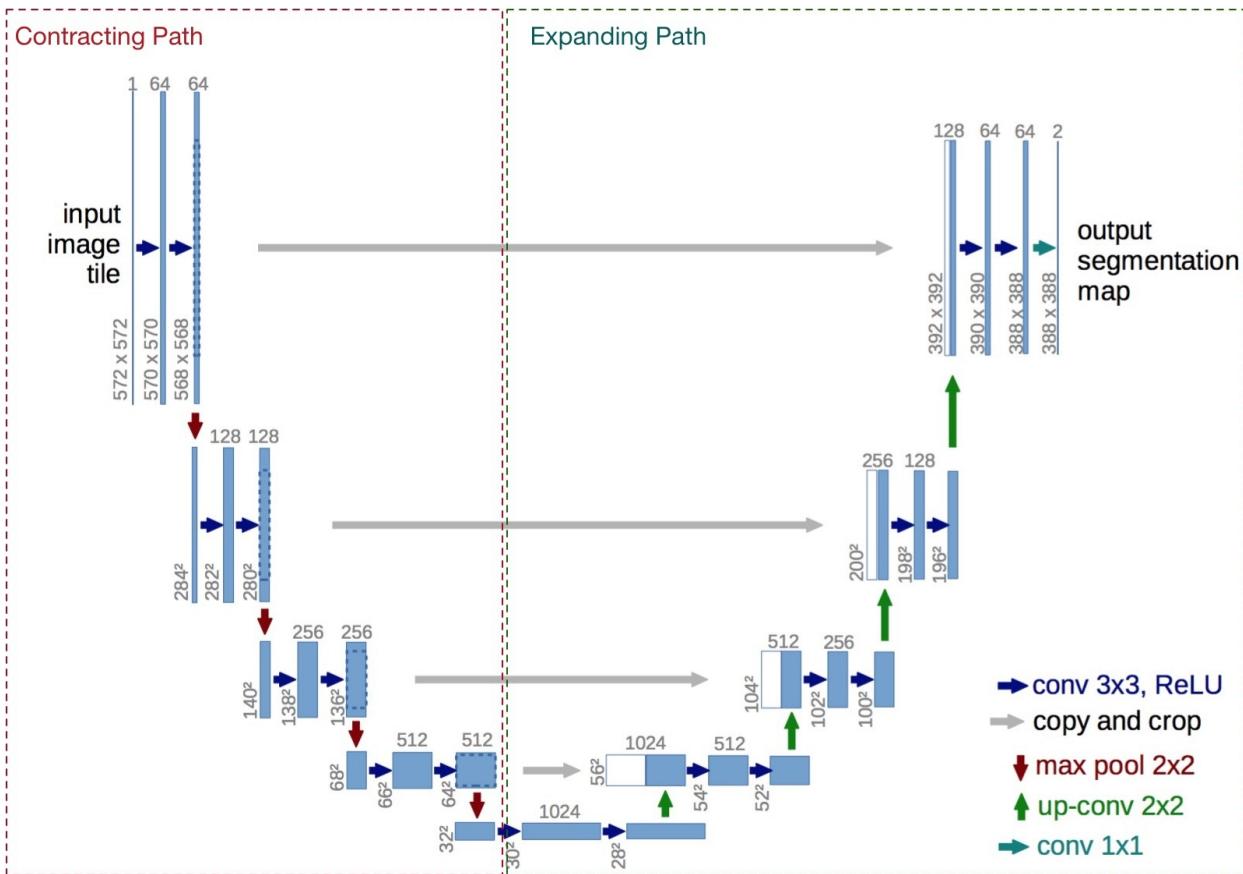


图1：U-Net网络结构图

如图1中所示，网络的输入图片的尺寸是 572×572 ，而输出Feature Map的尺寸是 388×388 ，这两个图像的大小是不同的，无法直接计算损失函数，那么U-Net是怎么操作的呢？

1.2 U-Net究竟输入了什么

首先，数据集我们的原始图像的尺寸都是 512×512 的。为了能更好的处理图像的边界像素，U-Net使用了镜像操作（Overlay-tile Strategy）来解决该问题。镜像操作即是给输入图像加入一个对称的边（图2），那么边的宽度是多少呢？一个比较好的策略是通过感受野确定。因为有效卷积是会降低Feature Map分辨率的，但是我们希望 512×512 的图像的边界点能够保留到最后一层Feature Map。所以我们需要通过加边的操作增加图像的分辨率，增加的尺寸即是感受野的大小，也就是说每条边界增加感受野的一半作为镜像边。

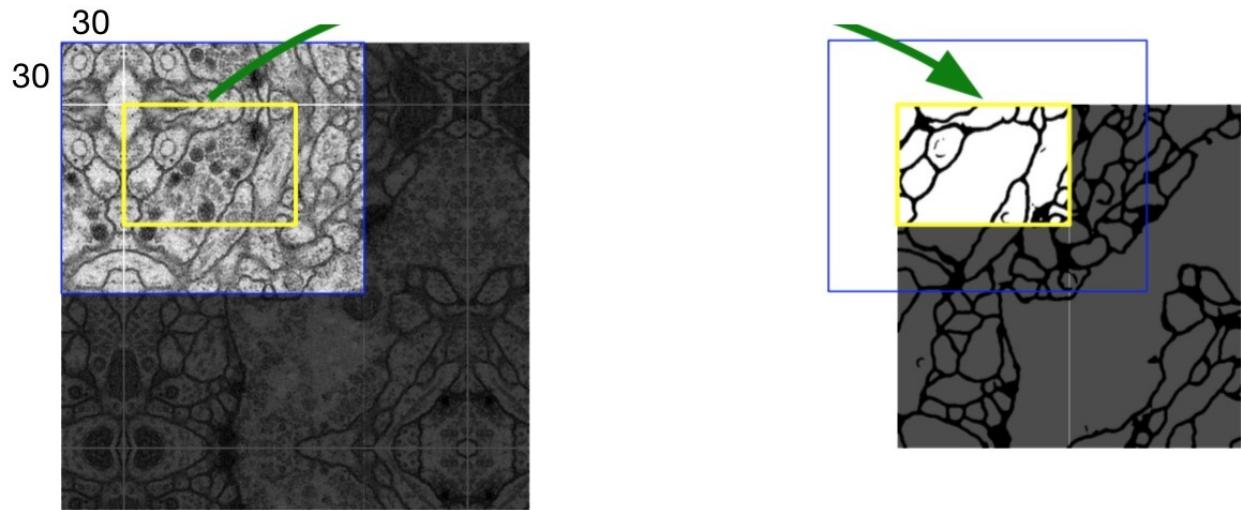


图2：U-Net镜像操作

根据图1中所示的压缩路径的网络架构，我们可以计算其感受野：

$$rf = (((0 \times 2 + 2 + 2) \times 2 + 2 + 2) \times 2 + 2 + 2) \times 2 + 2 + 2 = 60$$

这也就是为什么U-Net的输入数据是 572×572 的。572的卷积的另外一个好处是每次降采样操作的Feature Map的尺寸都是偶数，这个值也是和网络结构密切相关的。

1.3 U-Net的损失函数

ISBI数据集的一个非常严峻的挑战是紧密相邻的物体之间的分割问题。如图3所示，(a)是输入数据，(b)是Ground Truth，(c)是基于Ground Truth生成的分割掩码，(d)是U-Net使用的用于分离边界的损失权值。

图3：ISBI数据集样本示例

那么该怎样设计损失函数来让模型有分离边界的能力呢？U-Net使用的是带边界权值的损失函数：

362

$$E = \sum_{\mathbf{x} \in \Omega} w(\mathbf{x}) \log(p_{\ell(\mathbf{x})}(\mathbf{x}))$$

其中 $p_{\ell(\mathbf{x})}(\mathbf{x})$ 是 *softmax* 损失函数， $\ell : \Omega \rightarrow \{1, \dots, K\}$ 是像素点的标签值， $w : \Omega \in \mathbb{R}$ 是像素点的权值，目的是为了给图像中贴近边界点的像素更高的权值。

$$w(\mathbf{x}) = w_c(\mathbf{x}) + w_0 \cdot \exp\left(-\frac{(d_1(\mathbf{x}) + d_2(\mathbf{x}))^2}{2\sigma^2}\right)$$

其中 $w_c : \Omega \in \mathbb{R}$ 是平衡类别比例的权值， $d_1 : \Omega \in \mathbb{R}$ 是像素点到距离其最近的细胞的距离， $d_2 : \Omega \in \mathbb{R}$ 则是像素点到距离其第二近的细胞的距离。 w_0 和 σ 是常数值，在实验中 $w_0 = 10$ ， $\sigma \approx 5$ 。

2. 数据扩充

由于训练集只有30张训练样本，作者使用了数据扩充的方法增加了样本数量。并且作者指出任意的弹性形变对训练非常有帮助。

3. 总结

U-Net是比较早的使用多尺度特征进行语义分割任务的算法之一，其U形结构也启发了后面很多算法。但其也有几个缺点：

1. 有效卷积增加了模型设计的难度和普适性；目前很多算法直接采用了 `same` 卷积，这样也可以免去 Feature Map 合并之前的裁边操作
2. 其通过裁边的形式和 Feature Map 并不是对称的，个人感觉采用双线性插值的效果应该会更好。

DenseBox: Unifying Landmark Localization with End to End Object Detection

tags: OCR, DenseBox

前言

DenseBox[13]百度IDL的作品，提出的最初动机是为了解决普通的物体检测问题。其在2015年初就被提出来了，甚至比Fast R-CNN还要早，但是由于论文发表的比较晚，虽然算法上非常有创新点，但是依旧阻挡不了Fast R-CNN一统江山。

DenseBox的主要贡献如下：

1. 使用全卷积网络，任务类型类似于语义分割，并且实现了端到端的训练和识别，而R-CNN系列算法是从Faster R-CNN中使用了RPN代替了Selective Search才开始实现端到端训练的，而和语义分割的结合更是等到了2017年的Mask R-CNN才开始；
2. 多尺度特征，而R-CNN系列直到FPN才开始使用多尺度融合的特征；
3. 结合关键点的多任务系统，DenseBox的实验是在人脸检测数据集（MALF）上完成的，结合数据集中的人脸关键点可以使算法的检测精度进一步提升。

1. DenseBox详解

1.1 训练标签

DenseBox没有使用整幅图作为输入，因为作者考虑到一张图上的背景区域太多，计算时间会严重浪费在对没用的背景区域的卷积上。而且使用扭曲或者裁剪将不同比例的图像压缩到相同尺寸会造成信息的丢失。作者提出的策略是从训练图片中裁剪出包含人脸的patch，这些patch包含的背景区域足够完成模型的训练，详细过程如下：

1. 根据Ground Truth从训练数据集中裁剪出大小是人脸的区域的高的4.8倍的正方形作为一个patch，且人脸在这个patch的中心；
2. 将这个patch resize到 240×240 大小。

举例说明：一张训练图片中包含一个 60×80 的人脸，那么第一步会裁剪出大小是 384×384 的一个patch。在第二步中将这个patch resize到 240×240 。这张图片便是训练样本的输入。

通过上面方法采样得到的patch叫做正patch，除了这些正patch，DenseBox还随机采样到了等数量的随机patch，同时使翻转，位移和尺度变换三个数据增强的方法产生样本以增强模型的拟合能力。

训练集的标签是一个 $60 \times 60 \times 5$ 的热图（图1）， 60×60 表示热图的尺寸，从这个尺寸我们也可以看出训练样本经过了两次降采样。5表示热图的通道数，组成方式如下：

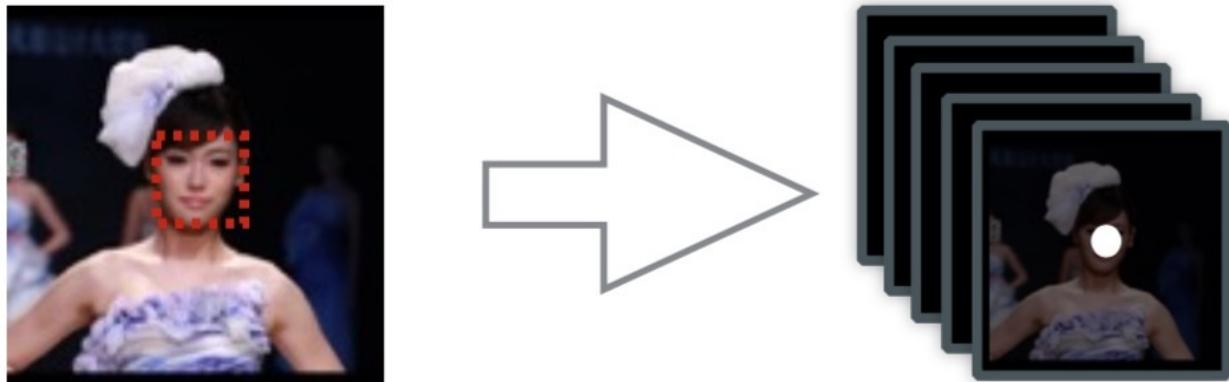


图1：*DenseBox*的*Ground Truth*

1. 图1中最前面的热图用于标注人脸区域置信度，前景为1，背景为0。*DenseBox*并没有使用左图的人脸矩形区域而是使用半径(r_c)为*Ground Truth*的高的0.3倍的圆作为标签值，而圆形的中心就是热图的中心，即有图中的白色圆形部分；
2. 图1中后面的四个热图表示像素点到最近的*Ground Truth*的四个边界的距离，如图2所示，*Ground Truth*为蓝色矩形，表示为 $d^* = (d_{x^t}^*, d_{x^b}^*, d_{y^t}^*, d_{y^b}^*)$ ，绿色为预测的矩形，表示为 $\hat{d} = (\hat{d}_{x^t}, \hat{d}_{x^b}, \hat{d}_{y^t}, \hat{d}_{y^b})$ 。（论文中符号的使用混乱且有错误，这里没有采用和论文完全相同的符号）。

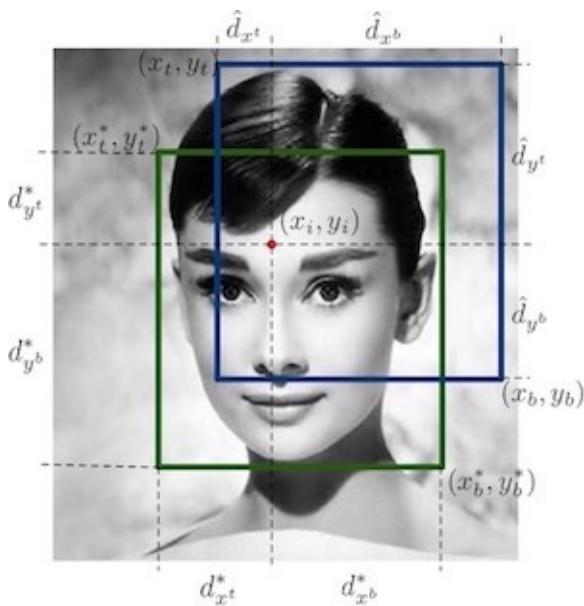


图2：DenseBox中距离热图示意图

如果训练样本中的人脸比较密集，一个patch中可能出现多个人脸，如果某个人脸和中心点处的人脸的高的比例在[0.8, 1.25]之间，则认为该样本为正样本。

作者认为DenseBox的标签设计是和感受野密切相关的，具体的讲，结合1.2节要分析的网络结构我们可以计算得到热图中每个像素的感受野是 48×48 ，这和我们每个patch中每个人脸的尺寸是非常接近的。在DenseBox中，每个像素点有5个预测值，而这5个预测值便可以确定一个检测框，所以DenseBox本质上也是一个密集采样，每个图片的采样个数是 $60 \times 60 = 3600$ 个。

1.2 网络结构

DenseBox使用了16层的VGG-19作为骨干网络，但是只使用了其前12层，如图3所示。

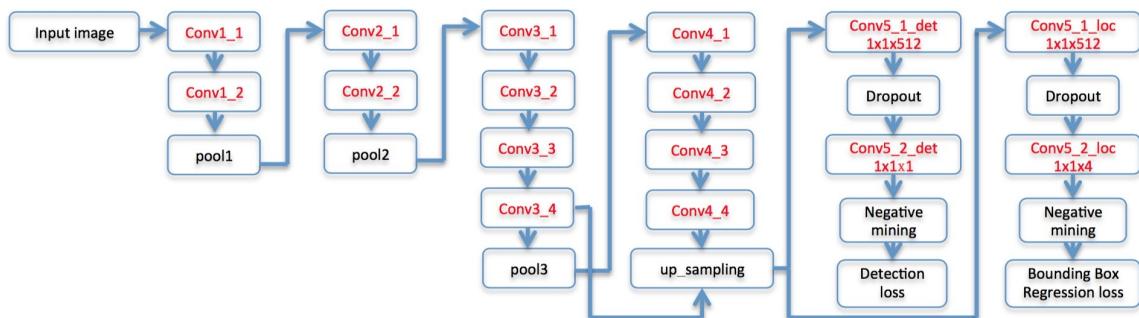


图3：DenseBox中的网络结构

首先需要注意的是在网络的Conv3_4和Conv4_4之间发生了一次特征融合，融合的方式是Conv4_4层的双线性插值上采样，因此得到的Feature Map和Conv3_4是相同的，即为 60×60 ，通过计算我们可以得知Conv3_4层的感受野的尺寸是 48×48 ，该层的尺寸和标签中的人脸尺寸接近，用于捕捉人脸区域的关键特征；Conv4_4层的感受野的大小是 118×118 ，用于捕捉人脸的上下文特征。

上采样之后网络有两个分支，分别用于计算检测损失和Bounding Box的回归损失，分支由 1×1 卷积核Dropout组成。从这里我们看出DenseBox也是一个多任务模型，下面我们开始介绍这个多任务模型。

网络在VGG-19部分的初始化使用的是ImageNet上得到的迁移学习的参数，其余部分使用的是Xavier初始化。

1.3 多任务模型

DenseBox的第一个损失为分类损失（检测损失）。在1.1节中我们知道，分类标签 $y^* \in \{0, 1\}$ ，设 \hat{y} 为模型的预测值。作者使用的是l2损失：

$$\mathcal{L}_{cls}(\hat{y}, y^*) = \|\hat{y} - y^*\|^2$$

第二个分支是bounding box回归损失，即计算图2中像素点分别到Ground Truth和到预测值之间的l2损失：

$$\mathcal{L}_{loc}(\hat{d}, d^*) = \sum_{i \in \{x^t, y^t, x^b, y^b\}} \|\hat{d}_i - d_i^*\|^2$$

1.4 平衡采样

在1.1节的最后我们讲到，DenseBox可被视为3600个样本的密集采样，其中每个像素点都可以看做是一个样本。在算法中并不是所有样本都会参与到训练中，且为了平衡正负样本，提高模型精度，DenseBox采用了以下策略。

1.4.1 忽略灰色区域

所谓灰色区域，是指正负样本边界部分的像素点，因为在这些区域由于标注的样本是很难区分的，让其参与训练反而会降低模型的精度，因此这一部分不会参与训练，在论文中，长度小于2的边界部分视为灰色区域。DenseBox使用 f_{ign} 对灰色样本进行标注， $f_{ign} = 1$ 表示为灰色区域样本。

1.4.2 Hard Negative Mining

DenseBox使用的Hard Negative Mining的策略和SVM类似，具体策略是：

1. 计算整个patch的3600个所有样本点，并根据loss进行排序；
2. 取其中的1%，也就是36个作为hard-negative样本；
3. 随机采样36个负样本和hard-negative构成72个负样本；
4. 随机采样72个正样本。

使用上面策略得到的144个样本参与训练，DenseBox使用掩码 f_{sel} 对参与训练的样本点进行标注

注：样本被选中 $f_{sel} = 1$ ，否则 $f_{sel} = 0$ 。

1.4.3 使用掩码的损失函数

损失函数使用掩码来控制哪些样本参与训练，其掩码 $M(\hat{t}_i)$ 是由1.4.1节的 f_{ign} 和1.4.2节的 f_{sel} 共同决定的：

$$M(\hat{t}_i) = \begin{cases} 0 & f_{ign}^i = 1 \text{ or } f_{sel}^i = 0 \\ 1 & \text{otherwise} \end{cases}$$

使用掩码后，得到的损失函数如下

$$\mathcal{L}_{det}(\theta) = \sum_i (M(\hat{t}_i) \mathcal{L}_{cls}(\hat{y}_i, y_i^*) + \lambda_{loc}[y_i^* > 0] M(\hat{t}_i) \mathcal{L}_{loc}(\hat{d}_i, d_i^*))$$

其中 θ 为卷积网络的参数， $[y_i^* > 0]$ 表示只有正样本参与bounding box的训练，其数学表达式为

$$[y_i^* > 0] = \begin{cases} 0 & y_i^* > 0 \\ 1 & \text{otherwise} \end{cases}$$

λ_{loc} 是平衡两个任务的参数，论文中值为3。位置 d_i 使用的是归一化的值。

1.5 结合关键点检测的多任务模型

论文中指出当DenseBox加入关键点检测的任务分支时模型的精度会进一步提升，这时只需要在图3的conv3_4和conv4_4融合之后的结果上添加一个用于关键点检测的分支即可，分支的详细结构如图4所示。

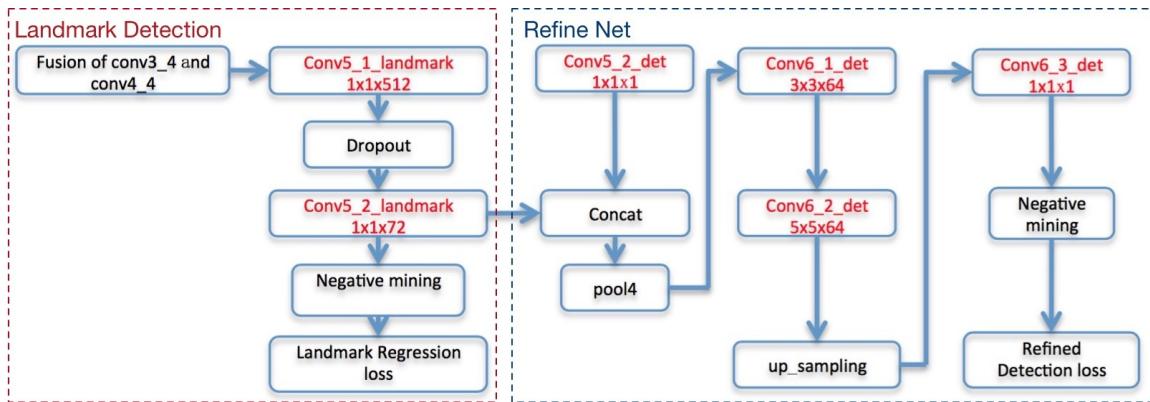


图4：Refine Network的网络结构

假设样本有 N 个关键点（在MALF中 $N = 72$ ），DenseBox的关键点检测的输出是 N 个热图，热图中的每个像素点表示改点为对应位置关键点的置信度。

关键点的标签值的生成方式也很简单，对于标签集中的第 i 个关键点 (x, y) ，在第 i 个feature map在 (x, y) 处的值是1，其它位置为0，有时也可以以 (x, y) 为圆心将值为1的区域扩展成一个圆，半径为 r_l 。

Landmark使用了1.4节中介绍的灰色区域和Hard Negative Mining方法进行采样，损失函数则是使用了采样样本之间的I2损失函数 \mathcal{L}_{lm} 。整个关键点检测如图4中红色虚线部分所示。

1.6 Refine Network

加入关键点检测分支之后，DenseBox根据关键点的置信度图和boudning box的置信度图构成了新的检测损失，并将其命名为Refine Network，如图4的蓝色虚线部分。更详细的讲，Refine Net通过拼接的方式融合了关键点检测的Conv5_2_landmark层和图2中bounding box的Conv5_2_det层，之后接了Max Pooling层，卷积层，上采样层最后生成新的预测值 \hat{y} 。

Refine Network也是使用了相同的I2损失函数，表示为 \mathcal{L}_{rf} 。

1.7 最终输出

最终得到的损失函数 \mathcal{L}_{full} 是1.4节中的 $\mathcal{L}_{det}(\theta)$ ，关键点检测任务和Refine之后的分类任务的加权和，表示为：

$$\mathcal{L}_{full}(\theta) = \lambda_{det}\mathcal{L}_{det}(\theta) + \lambda_{lm}\mathcal{L}_{lm}(\theta) + \mathcal{L}_{rf}(\theta)$$

注意 $\mathcal{L}_{det}(\theta)$ 是由检测任务和**bounding box**定位任务共两个任务组成，因此 $\mathcal{L}_{full}(\theta)$ 本质上是个四任务模型。 λ_{det} 和 λ_{lm} 是平衡各任务的权值，论文中的值分别是1和0.5，更好的策略是根据收敛情况进行调整。

1.8 测试

DenseBox的检测过程如图5所示，先考虑不带关键点检测的流程：

1. 图像金字塔作为输入；
2. 经过网络后产生5个通道的Feature Map；
3. 两次双线性插值上采样得到和输入图像相同尺寸的Feature Map；
4. 根据Feature Map得到检测框；
5. NMS合并检测框得到最终的检测结果。

总结

DenseBox在今天看来技术性依旧非常强，虽然作为一个人脸检测的论文被发表，但是其思想也可以迁移到通用的物体检测中。而且得到的效果几乎和Faster R-CNN旗鼓相当。由于采用了FCN的架构，DenseBox本身的速度应该不会太慢，唯一的性能瓶颈应该是图像金字塔的引入。在之后的研究中，DenseBox通过SPP-Net中的金字塔池化的方式将检测时间优化到了GPU的实时。本来DenseBox在物体检测中能有更大的价值的，但是由于其仅限于百度内部使用，并没有开源，论文投稿也比较晚造成了R-CNN系列的一统天下。当然R-CNN系列凭借其代码的规范性，算法的通用性等优点一统天下也不意外。

UnitBox: An Advanced Object Detection Network

前言

UnitBox[12]使用了和DenseBox[13]类似的方法进行人脸检测。在DenseBox中，bounding box的定位使用的是L2损失。L2损失的一个缺点是会使模型在训练过程中更偏向于尺寸更大的物体，因为大尺寸物体的L2损失更容易大于小物体。

为了解决这个问题，UnitBox中使用了IoU损失，顾名思义，IoU损失既是使用Ground Truth和预测bounding box的交并比作为损失函数。

1. UnitBox详解

首先回顾DenseBox中介绍的几个重要的知识点，明白了这些知识点才能理解下面要讲解的UnitBox。

1. DenseBox网络结构是全卷积网络，输出层是一个 $\frac{m}{4} \times \frac{n}{4}$ 的Feature Map，是一个image-to-image的任务；
2. 输出Feature Map的每个像素点 (x_i, y_i) 都是可以确定一个检测框的样本，包样本置信度 y 和该点到bounding box四条边的距离 (x_t, x_b, y_t, y_b) ，如图1所示

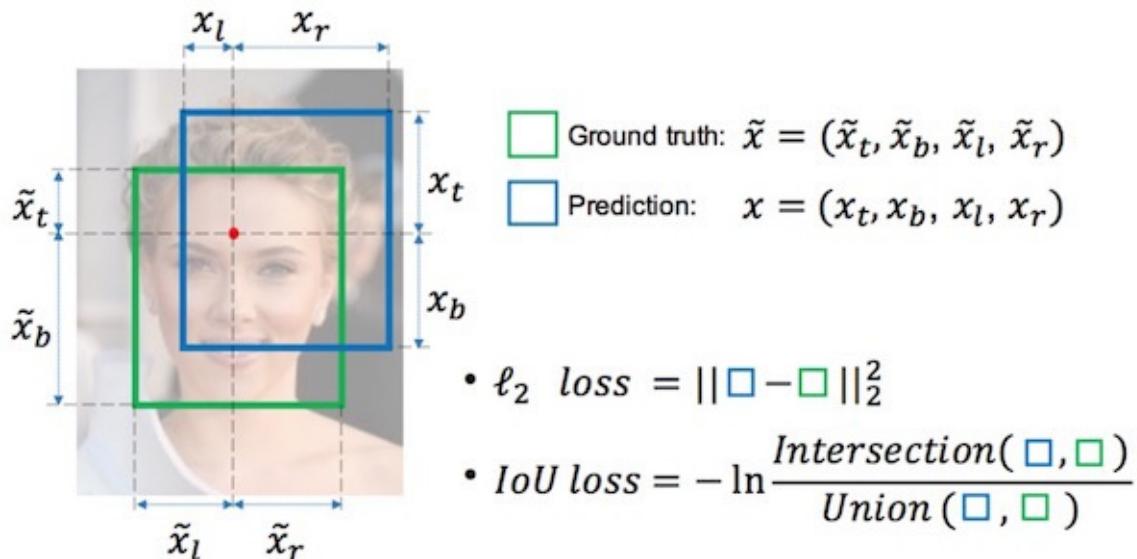


图1：UnitBox的Ground Truth

Unitbox的一个最重要的特征是使用IoU损失替代了传统的I2损失，下面我们先从IoU损失入手讲解UnitBox。

1.1 IoU损失的前向计算

前向计算非常简单，如图2中的伪代码所示：

Algorithm 1: IoU loss Forward

```

Input:  $\tilde{x}$  as bounding box ground truth
Input:  $x$  as bounding box prediction
Output:  $\mathcal{L}$  as localization error
for each pixel  $(i, j)$  do
    if  $\tilde{x} \neq 0$  then
         $X = (x_t + x_b) * (x_l + x_r)$ 
         $\tilde{X} = (\tilde{x}_t + \tilde{x}_b) * (\tilde{x}_l + \tilde{x}_r)$ 
         $I_h = \min(x_t, \tilde{x}_t) + \min(x_b, \tilde{x}_b)$ 
         $I_w = \min(x_l, \tilde{x}_l) + \min(x_r, \tilde{x}_r)$ 
         $I = I_h * I_w$ 
         $U = X + \tilde{X} - I$ 
         $IoU = \frac{I}{U}$ 
         $\mathcal{L} = -\ln(IoU)$ 
    else
         $\mathcal{L} = 0$ 
    end
end

```

图2：IoU损失前向计算伪代码

注意结合图1中的 x 和 \tilde{x} 的定义理解图2中的伪代码， X 计算的是预测bounding box的面积， \tilde{X} 则是ground truth的bounding box的面积， I 是两个区域的交集， U 是两个区域的并集。

$\mathcal{L} = -\ln(IoU)$ 本质上是对IoU的交叉熵损失函数：那么可以将IoU看做从伯努利分布中的随机采样，并且 $p(IoU = 1) = 1$ ，于是可以化简成源码中的公式，即

$$\mathcal{L} = -p\ln(IoU) - (1-p)\ln(1-IoU) = -\ln(IoU)$$

1.2 IoU损失的反向计算

这里我们推导一下IoU损失的反向计算公式，以变量 x_t 为例：

$$\begin{aligned} \frac{\partial \mathcal{L}}{\partial x_t} &= \frac{\partial}{\partial x_t}(-\ln(IoU)) \\ &= -\frac{1}{IoU} \frac{\partial}{\partial x_t}(IoU) \\ &= -\frac{1}{IoU} \frac{\partial}{\partial x_t}\left(\frac{I}{U}\right) \\ &= \frac{1}{IoU} \frac{I \times \frac{\partial U}{\partial x_t} - U \times \frac{\partial I}{\partial x_t}}{U^2} \\ &= \frac{I \times \frac{\partial}{\partial x_t}(X + \tilde{X} - I) - U \times \frac{\partial I}{\partial x_t}}{U^2 IoU} \\ &= \frac{I \times \left(\frac{\partial}{\partial x_t}X - \frac{\partial}{\partial x_t}I\right) - U \times \frac{\partial I}{\partial x_t}}{U^2 IoU} \\ &= \frac{1}{U} \frac{\partial X}{x_t} - \frac{U + I}{UI} \frac{\partial I}{x_t} \end{aligned}$$

其中：

$$\frac{\partial X}{x_t} = x_l + x_r$$

$$\frac{\partial I}{x_t} = \begin{cases} I_w, & \text{if } x_t < \tilde{x}_t (\text{or } x_b < \tilde{x}_b) \\ 0, & \text{otherwise} \end{cases}$$

其它三个变量的推导方法类似，这里不再重复。

从这个推导公式中我们可以看出三点信息：

1. 损失函数和 $\frac{\partial X}{x_t}$ 成正比，因此预测的面积越大，损失越多；
2. 同时损失函数和 $\frac{\partial I}{x_t}$ 成反比，因此我们希望交集尽可能的大；
3. 综合1, 2两条我们可以看出当bounding box等于ground truth值时检测效果最好。

因此可以看出优化IoU损失是正向促进物体检测的精度的。

1.3 UnitBox网络架构

UnitBox的网络结构如图3所示，下面分析几个重要的方面

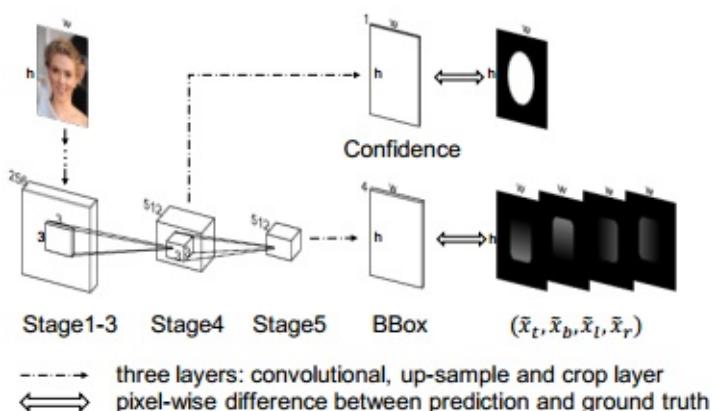


图3：UnitBox网络结构

1.3.1 网络的输入输出

输入：由于使用了全卷积结构，在测试时直接输入原始图片即可。在训练时每个batch的图像的尺寸相同即可。

输出：UnitBox的输出标签分成两部分，上半部分椭圆形为置信度热图，具体标签生成方法论文中没有讲，猜测应该是采样类似于DenseBox中的策略。另外一组是bounding box热图，生成策略应该也是类似于DenseBox。

1.3.2 骨干网络

骨干网络是VGG-16，用于计算置信度热图的是stage-4阶段的Feature Map，计算方式是先通过线性插值得到相同尺寸的Feature Map，再通过 1×1 卷积将Feature Map的通道数降到1，此时得到的Feature Map表示预测的置信度热图。网络的另外一个分支用于预测bounding box的热图，Feature Map取自VGG-16的stage-5。通过和上面类似的方法得到和原图尺寸相同的4个预测bounding box热图。并且在后面加入了ReLU将负值置0。

至于为什么两个任务使用不同的stage，论文中给出的解释是IoU损失计算的Bounding box是一个整体，因此需要更大的感受野，由于UnitBox仅添加了两组 1×1 卷积，因此速度要比DenseBox快很多。

2. 总结

UnitBox的提出虽然目的是为了解决人脸检测问题，但是从其算法角度讲也可以扩展到其它类型的检测任务中，但是像Pascal VOC或者是COCO这样的多类别检测任务，目前基于语义分割的方法还需要改进，因为其本身的置信度热图的设计是位于 $[0, 1]$ 直接的一个值，暂时是无

法进行多分类的。

写这篇文章的目的是介绍UnitBox中引入的IoU损失，IoU损失有如下优点

- IoU损失将位置信息作为一个整体进行训练，而L2损失把它们当做互相独立的四个变量进行训练，这样得到的结果更准确；
- 无论输入的样本是什么样子，IoU的值均介于[0, 1]，这种天然的归一化的损失使模型具有更强的处理多尺度图像的能力。

Batch Normalization

tags: Normalization

前言

Batch Normalization(BN)是深度学习中非常好用的一个算法，加入BN层的网络往往更加稳定并且BN还起到了一定的正则化的作用。在这篇文章中，我们将详细介绍BN的技术细节[114]以及其能工作的原因[11]。

在提出BN的文章中[114]，作者BN能工作的原因是BN解决了普通网络的内部协变量偏移 (Internel Covariate Shift, ICS) 的问题，所谓ICS是指网络各层的分布不一致，网络需要适应这种不一致从而增加了学习的难度。而在[11]中，作者通过实验验证了BN其实和ICS的关系并不大，其能工作的原因是使损失平面更加平滑，并给出了其结论的数学证明。

1. BN详解

1.1 内部协变量偏移

BN的提出是基于小批量随机梯度下降 (mini-batch SGD) 的，mini-batch SGD是介于one-example SGD和full-batch SGD的一个折中方案，其优点是比full-batch SGD有更小的硬件需求，比one-example SGD有更好的收敛速度和并行能力。随机梯度下降的缺点是对参数比较敏感，较大的学习率和不合适的初始化值均有可能导致训练过程中发生梯度消失或者梯度爆炸的现象的出现。BN的出现则有效的解决了这个问题。

在Sergey Ioffe的文章中，他们认为BN的主要贡献是减弱了内部协变量偏移 (ICS) 的问题，论文中对ICS的定义是：as the change in the distribution of network activations due to the change in network parameters during training。作者认为ICS是导致网络收敛的慢的罪魁祸首，因为模型需要学习在训练过程中会不断变化的隐层输入分布。作者提出BN的动机是企图在训练过程中将每一层的隐层节点的输入固定下来，这样就可以避免ICS的问题了。

在深度学习训练中，白化 (Whiten) 是加速收敛的一个小Trick，所谓白化是指将图像像素点变化到均值为0，方差为1的正态分布。我们知道在深度学习中，第 i 层的输出会直接作为第 $i+1$ 层的输入，所以我们能不能对神经网络的每一层的输入都做一次白化呢？其实BN就是这么做的。

1.2 梯度饱和

我们知道sigmoid激活函数和tanh激活函数存在梯度饱和的区域，其原因是激活函数的输入值过大或者过小，其得到的激活函数的梯度值会非常接近于0，使得网络的收敛速度减慢。传统的方法是使用不存在梯度饱和区域的激活函数，例如ReLU等。BN也可以缓解梯度饱和的问题，它的策略是在调用激活函数之前将 $WX + b$ 的值归一化到梯度值比较大的区域。假设激活函数为 g ，BN应在 g 之前使用：

$$z = g(\text{BN}(Wx + b))$$

1.3 BN的训练过程

如果按照传统白化的方法，整个数据集都会参与归一化的计算，但是这种过程无疑是非常耗时的。BN的归一化过程是以批量为单位的。如图1所示，假设一个批量有 n 个样本

$\mathcal{B} = \{x_1, x_2, \dots, x_m\}$ ，每个样本有 d 个特征，那么这个批量的每个样本第 k 个特征的归一化后的值为

$$\hat{x}^{(k)} = \underbrace{x^{(k)}}_{\sqrt{\text{Var}[x^{(k)}]}} E[x^{(k)}]$$

其中 E 和 Var 分别表示在第 k 个特征上这个批量中所有的样本的均值和方差。

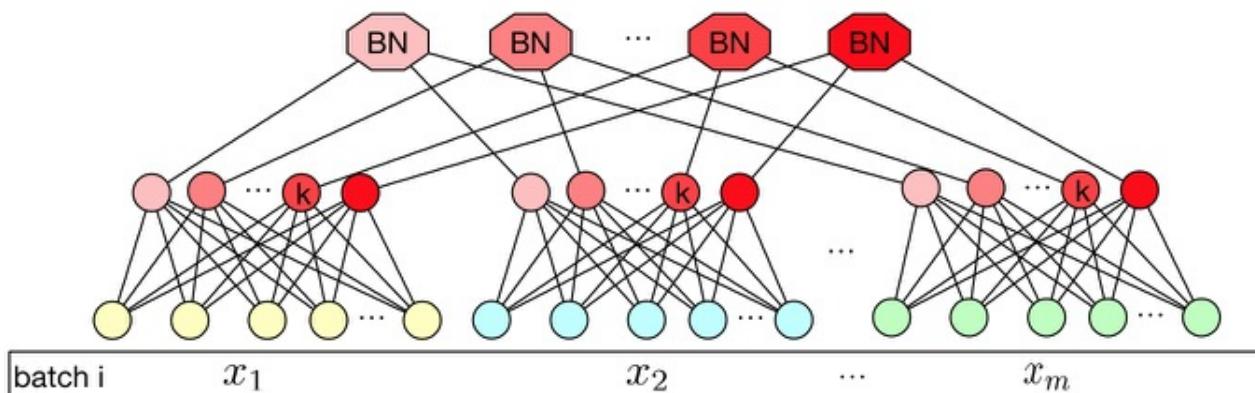


图1：神经网络的BN示意图

这种表示会对模型的收敛有帮助，但是也可能破坏已经学习到的特征。为了解决这个问题，BN添加了两个可以学习的变量 β 和 γ 用于控制网络能够表达直接映射，也就是能够还原BN之前学习到的特征。

$$y^{(k)} = \gamma^{(k)} \hat{x}^{(k)} + \beta^{(k)}$$

当 $\gamma^{(k)} = \sqrt{\text{Var}[x^{(k)}]}$ 并且 $\beta^{(k)} = E[x^{(k)}]$ 时， $y^{(k)} = x^{(k)}$ ，也就是说经过BN操作之后的网络容量是不小于没有BN操作的网络容量的。

综上所述，BN可以看做一个以 γ 和 β 为参数的，从 $x_{1\dots m}$ 到 $y_{1\dots m}$ 的一个映射，表示为

$$BN_{\gamma, \beta} : x_{1\dots m} \rightarrow y_{1\dots m}$$

BN的伪代码如算法1所示

```

Input: Values of  $x$  over a mini-batch:  $\mathcal{B} = \{x_{1\dots m}\}$ ;  

        Parameters to be learned:  $\gamma, \beta$   

Output:  $\{y_i = BN_{\gamma, \beta}(x_i)\}$ 

 $\mu_{\mathcal{B}} \leftarrow \frac{1}{m} \sum_{i=1}^m x_i$  // mini-batch mean  

 $\sigma_{\mathcal{B}}^2 \leftarrow \frac{1}{m} \sum_{i=1}^m (x_i - \mu_{\mathcal{B}})^2$  // mini-batch variance  

 $\hat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}}$  // normalize  

 $y_i \leftarrow \gamma \hat{x}_i + \beta \equiv BN_{\gamma, \beta}(x_i)$  // scale and shift

```

在训练时，我们需要计算BN的反向传播过程，感兴趣的同学可以自行推导，这里直接给出结论（ l 表示损失函数）。

$$\begin{aligned}
 \frac{\partial l}{\partial \hat{x}_i} &= \frac{\partial l}{\partial y_i} \cdot \gamma \\
 \frac{\partial l}{\partial \sigma_{\mathcal{B}}^2} &= \sum_{i=1}^m \frac{\partial l}{\partial \hat{x}_i} \cdot (x_i - \mu_{\mathcal{B}}) \cdot \frac{-1}{2} (\sigma_{\mathcal{B}}^2 + \epsilon)^{-\frac{3}{2}} \\
 \frac{\partial l}{\partial \mu_{\mathcal{B}}} &= \left(\sum_{i=1}^m \frac{\partial l}{\partial \hat{x}_i} \cdot \frac{-1}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}} \right) + \frac{\partial l}{\partial \sigma_{\mathcal{B}}^2} \cdot \frac{\sum_{i=1}^m -2(x_i - \mu_{\mathcal{B}})}{m} \\
 \frac{\partial l}{\partial x_i} &= \frac{\partial l}{\partial \hat{x}_i} \cdot \frac{1}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}} + \frac{\partial l}{\partial \sigma_{\mathcal{B}}^2} \cdot \frac{2(x_i - \mu_{\mathcal{B}})}{m} + \frac{\partial l}{\partial \mu_{\mathcal{B}}} \cdot \frac{1}{m} \\
 \frac{\partial l}{\partial \gamma} &= \sum_{i=1}^m \frac{\partial l}{\partial y_i} \cdot \hat{x}_i \\
 \frac{\partial l}{\partial \beta} &= \sum_{i=1}^m \frac{\partial l}{\partial y_i}
 \end{aligned}$$

通过上面的式子中我们可以看出BN是处处可导的，因此可以直接作为层的形式加入到神经网络中。

1.4 BN的测试过程

在训练的时候，我们采用SGD算法可以获得该批量中样本的均值和方差。但是在测试的时候，数据都是以单个样本的形式输入到网络中的。在计算BN层的输出的时候，我们需要获取的均值和方差是通过训练集统计得到的。具体的讲，我们会从训练集中随机取多个批量的数据集，每个批量的样本数是 m ，测试的时候使用的均值和方差是这些批量的均值。

$$\mathbb{E}(x) \leftarrow \mathbb{E}_{\mathcal{B}}[\mu_{\mathcal{B}}]$$

$$\text{Var}(x) \leftarrow \frac{m}{m-1} \mathbb{E}_{\mathcal{B}}[\sigma_{\mathcal{B}}^2]$$

上面的过程明显非常耗时，更多的开源框架是在训练的时候，顺便就把采样到的样本的均值和方差保留了下来。在Keras中，这个变量叫做滑动平均（moving average），对应的均值叫做滑动均值（moving mean），方差叫做滑动方差（moving variance）。它们均使用 `moving_average_update` 进行更新。在测试的时候则使用滑动均值和滑动方差代替上面的 $\mathbb{E}(x)$ 和 $\text{Var}(x)$ 。

滑动均值和滑动方差的更新如下：

$$\mathbb{E}_{\text{moving}}(x) = m \times \mathbb{E}_{\text{moving}}(x) + (1 - m) \times \mathbb{E}_{\text{sample}}(x)$$

$$\text{Var}_{\text{moving}}(x) = m \times \text{Var}_{\text{moving}}(x) + (1 - m) \times \text{Var}_{\text{sample}}(x)$$

其中 $\mathbb{E}_{\text{moving}}(x)$ 表示滑动均值， $\mathbb{E}_{\text{sample}}(x)$ 表示采样均值，方差定义类似。 m 表示遗忘因子 momentum，默认值是0.99。

滑动均值和滑动方差，以及可学参数 β ， γ 均是对输入特征的线性操作，因此可以这两个操作合并起来。

$$y = \frac{\gamma}{\sqrt{\text{Var}[x] + \epsilon}} \cdot x + \left(\beta - \frac{\gamma \mathbb{E}[x]}{\sqrt{\text{Var}[x] + \epsilon}} \right)$$

1.5 卷积网络中的BN

BN除了可以应用在MLP上，其在CNN网络中的表现也非常好，但是在RNN上的表现并不好，具体原因后面解释，这里详细介绍BN在卷积网络中的使用方法。

卷积网络和MLP的不同点是卷积网络中每个样本的隐层节点的输出是三维（宽度，高度，维度）的，而MLP是一维的，如图2所示。

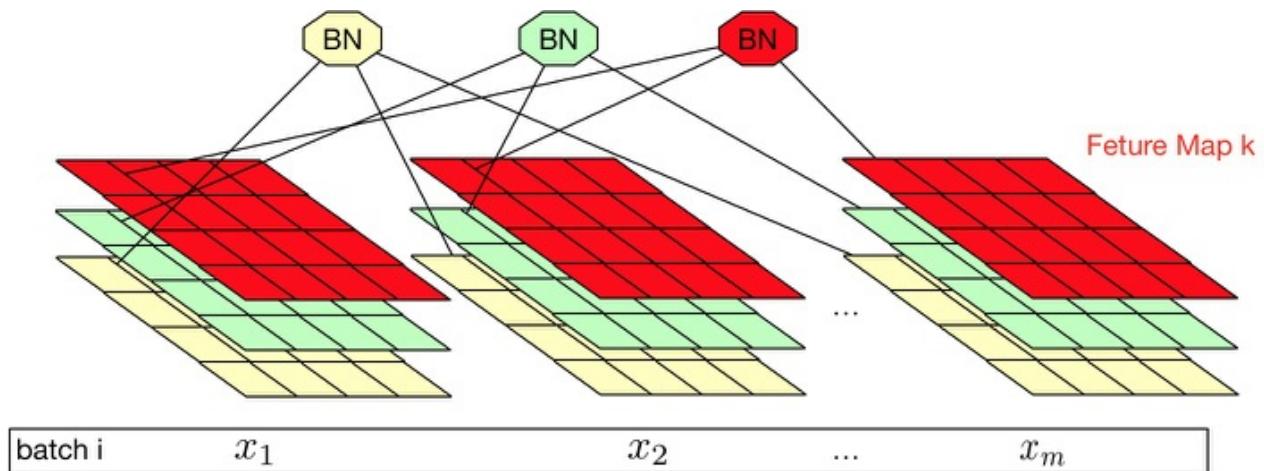


图2：卷积网络的BN示意图

在图2中，假设一个批量有 m 个样本，Feature Map的尺寸是 $p \times q$ ，通道数是 d 。在卷积网络的中，BN的操作是以Feature Map为单位的，因此一个BN要统计的数据个数为 $m \times p \times q$ ，每个Feature Map使用一组 γ 和 β 。

2. BN的背后原理

2.1 BN与ICS无关

最近MIT的一篇文章[11]否定了BN的背后原理是因为其减少了ICS的问题。在这篇文章中，作者通过两个实验验证了ICS和BN的关系非常小的观点。

第一个实验验证了ICS和网络性能的关系并不大，在这个实验中作者向使用了BN的网络中加入了随机噪声，目的是使这个网络的ICS更加严重。实验结果表明虽然加入了随机噪声的BN的ICS问题更加严重，但是它的性能是要优于没有使用BN的普通网络的，如图3所示。

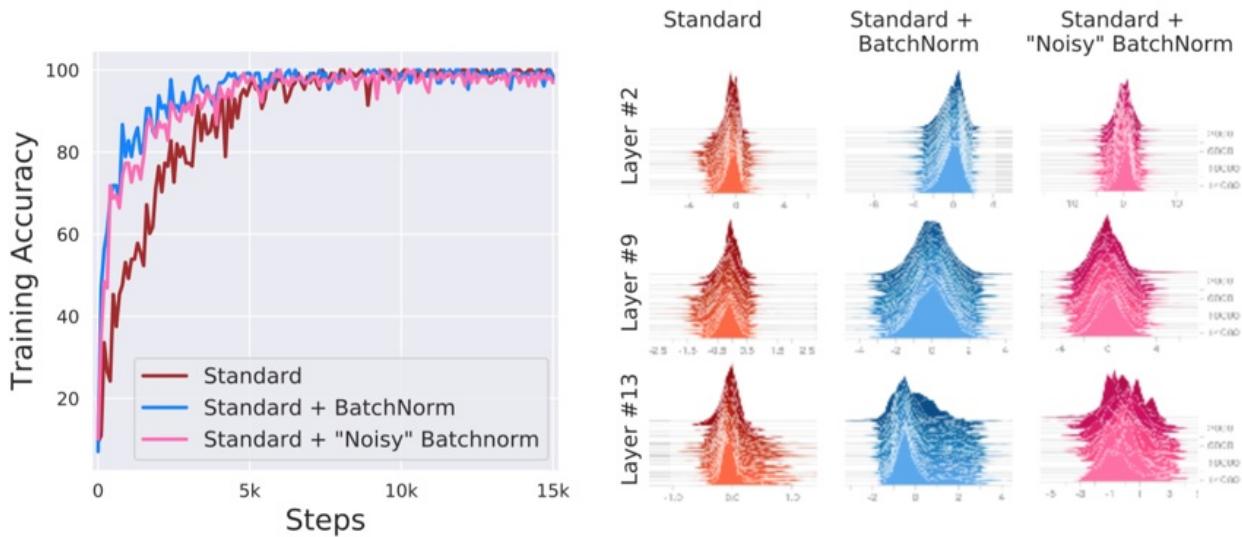


图3：BN，普通网络的，加入噪音的BN的ICS实验数据图

第二个实验验证了BN并不会减小ICS，有时候甚至还能提升ICS。在这个实验中，作者对ICS定义为：

定义：假设 \mathcal{L} 是损失值， $W_1^{(t)}, \dots, W_k^{(t)}$ 是在 k 个层中在时间 t 时的参数值， $(x^{(t)}, y^{(t)})$ 是在 t 时刻的输入特征和标签值，ICS定义为在时间 t 时，第 i 个隐层节点的两个变量的距离 $\|G_{t,i} - G'_{t,i}\|_2$ ，其中

$$G_{t,i} = \nabla_{w_i^{(t)}} \mathcal{L}(W_1^{(t)}, \dots, W_k^{(t)}; x^{(t)}, y^{(t)})$$

$$G'_{t,i} = \nabla_{w_i^{(t)}} \mathcal{L}(W_1^{(t+1)}, \dots, W_{i-1}^{(t+1)}, W_i^{(t)}, W_{i+1}^{(t)}, \dots, W_k^{(t)}; x^{(t)}, y^{(t)})$$

两个变量的区别在于 W_1, \dots, W_{i-1} 是 t 时刻的还是 $t+1$ 时刻的，其中 $G_{t,i}$ 表示更新梯度时使用的参数， $G'_{t,i}$ 表示使用这批样本更新后的新的参数。在上面提到的欧氏距离中，值越接近0说明ICS越小。另外一个相似度指标是cosine夹角，值越接近于1说明ICS越小。图4的实验结果（25层的DLN）表明BN和ICS的关系并不是很大。

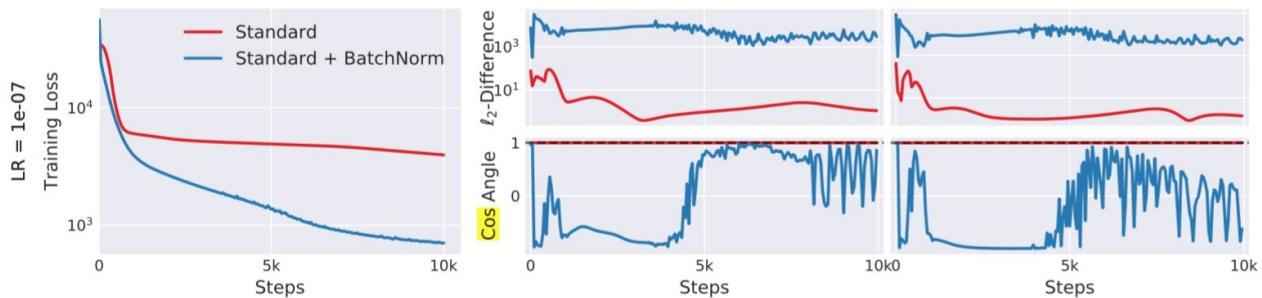


图4：普通网络和带BN的网络在两个ICS指标上的实验结果

2.2 BN与损失平面

通过上面两个实验，作者认为BN和ICS的关系不大，那么BN为什么效果好呢，作者认为BN的作用是平滑了损失平面（loss landscape），关于损失平面的介绍，参考文章[10]，这篇文章中介绍了损失平面的概念，并指出残差网络和DenseNet均起到了平滑损失平面的作用，因此他们具有较快的收敛速度。

作者证明了BN处理之后的损失函数满足Lipschitz连续，即损失函数的梯度小于一个常量，因此网络的损失平面不会震荡的过于严重。

$$||f(x_1) - f(x_2)|| \leq L||x_1 - x_2||$$

而且损失函数的梯度也满足Lipschitz连续，这里叫做 β -平滑，即斜率的斜率也不会超过一个常量。

$$||\nabla f(x_1) - \nabla f(x_2)|| \leq \beta ||x_1 - x_2||$$

作者认为当着两个常量的值均比较小的时候，损失平面就可以看做是平滑的。图5是加入没有跳跃连接的网络和加入跳跃连接（残差网络）的网络的损失平面的可视化，作者认为BN和残差网络对损失平面平滑的效果类似。

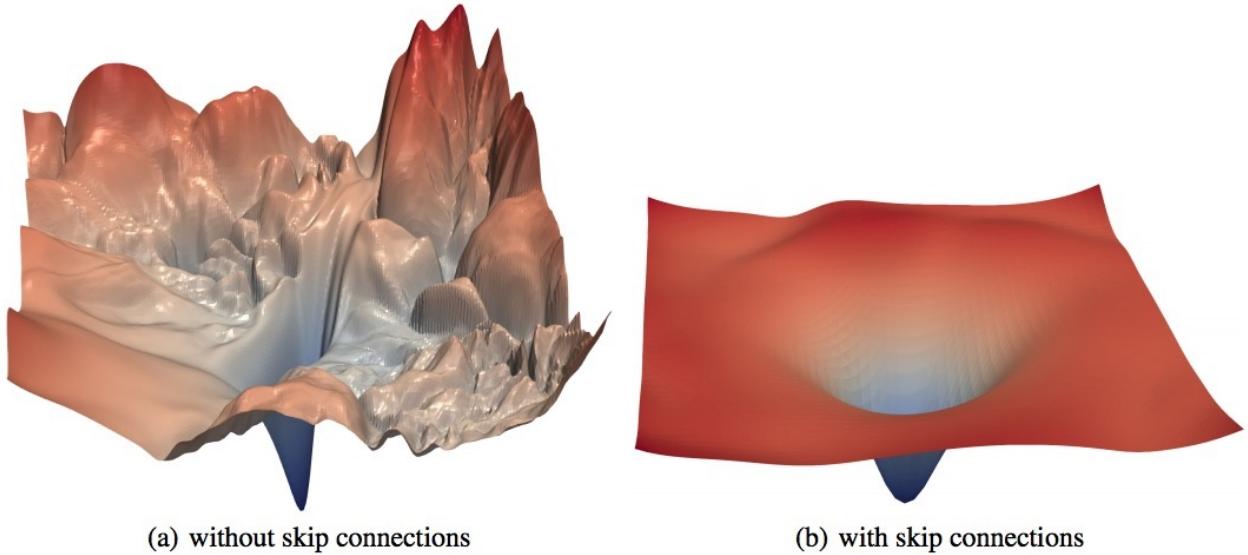


图5：损失平面可视化（无跳跃连接 vs ResNet）

通过上面的分析，我们知道BN收敛快的原因是由于BN产生了更光滑的损失平面。其实类似于BN的能平滑损失平面的策略均能起到加速收敛的效果，作者在论文中尝试了 l_p -norm的策略（即通过取它们 l_p -norm的均值的方式进行归一化）。实验结果表明它们均取得了和BN类似的效果。

2.3 BN的数学定理

作者对于自己的猜想，给出了5个定理，引理以及观察并在附录中给出了证明，由于本人的数学能力有限，这些证明有些看不懂，有需要的同学自行查看证明过程。

定理4.1：设 $\hat{\mathcal{L}}$ 为BN网络的损失函数， \mathcal{L} 为普通网络的损失函数，它们满足：

$$\|\nabla_{\mathbf{y}_j} \hat{\mathcal{L}}\|^2 \leq \frac{\gamma^2}{\sigma_j^2} (\|\nabla_{\mathbf{y}_j} \mathcal{L}\|^2 - \frac{1}{m} \langle \mathbf{1}, \nabla_{\mathbf{y}_j} \mathcal{L} \rangle^2 - \frac{1}{\sqrt{m}} \langle \nabla_{\mathbf{y}_j} \mathcal{L}, \hat{\mathbf{y}}_j \rangle^2)$$

在绝大多数场景中， σ 作为不可控的项往往值是要大于 γ 的，因此证明了BN可以使神经网络满足Lipschitz连续；

定理4.2：假设 $\hat{\mathbf{g}}_j = \nabla_{\mathbf{y}_j} \mathcal{L}$ ， $\mathbf{H}_{jj} = \frac{\partial \mathcal{L}}{\partial \mathbf{y}_j \partial \mathbf{y}_j}$ 是Hessian矩阵：

$$(\nabla_{\mathbf{y}_j} \hat{\mathcal{L}})^T \frac{\partial \hat{\mathcal{L}}}{\partial \mathbf{y}_j \partial \mathbf{y}_j} (\nabla_{\mathbf{y}_j} \hat{\mathcal{L}}) \leq \frac{\gamma^2}{\sigma_j^2} (\hat{\mathbf{g}}_j^T \mathbf{H}_{jj} \hat{\mathbf{g}}_j - \frac{1}{m\gamma} \langle \hat{\mathbf{g}}_j, \hat{\mathbf{y}}_j \rangle \|\frac{\partial \hat{\mathcal{L}}}{\partial \mathbf{y}_j}\|^2)$$

同理，4.2证明了BN是神经网络的损失函数的梯度也满足Lipschitz连续；

观察**4.3**：由于BN可以还原为直接映射，所以普通神经网络的最优损失平面也一定存在于带BN的网络中。

定理4.4：设带有BN的网络的损失为 $\hat{\mathcal{L}}$ ，与之等价的无BN的损失为 \mathcal{L} ，它们满足如果

$g_j = \max_{\|X\| \leq \lambda} \|\nabla_w \mathcal{L}\|^2$ ， $\hat{g}_j = \max_{\|X\| < \lambda} \|\nabla_w \hat{\mathcal{L}}\|^2$ ，我们可以推出：

$$\hat{g}_j \leq \frac{\gamma^2}{\sigma^2} (g_j^2 - m\mu_{g_j}^2 - \lambda^2 \langle \nabla_{\mathbf{y}_j}, \hat{\mathbf{y}}_j \rangle^2)$$

定理4.3证明了BN可以降低损失函数梯度的上界。

引理4.5：假设 W^* 和 \hat{W}^* 分别是普通神经网络和带BN的神经网络的局部最优解的权值，对于任意的初始化 W_0 ，我们有：

$$\|W_0 - \hat{W}^*\|^2 \leq \|W_0 - W^*\|^2 - \frac{1}{\|W^*\|^2} (\|W^*\|^2 - \langle W^*, W_0 \rangle)^2$$

引理4.5证明了BN对参数的不同初始化更加不敏感。

总结

BN是深度学习调参中非常好用的策略之一（另外一个可能就是Dropout），当你的模型发生梯度消失/爆炸或者损失值震荡比较严重的时候，在BN中加入网络往往能取得非常好的效果。

BN也有一些不是非常适用的场景，在遇见这些场景时要谨慎的使用BN：

- 受制于硬件限制，每个Batch的尺寸比较小，这时候谨慎使用BN；
- 在类似于RNN的动态网络中谨慎使用BN；
- 训练数据集和测试数据集方差较大的时候。

在Ioffe的论文中，他们认为BN能work的原因是因为减轻了ICS的问题，而在Santurkar的论文中则对齐进行了否定。它们结论的得出非常依赖他们自己给出的ICS的数学定义，这个定义是否不能说不对，但是感觉不够精确，BN真的和ICS没有一点关系吗？我是觉得不一定。

Layer Normalization

tags: Normalization

前言

在上一篇文章中我们介绍了BN[114]的计算方法并且讲解了BN如何应用在MLP以及CNN中如何使用BN。在文章的最后，我们指出BN并不适用于RNN等动态网络和batchsize较小的时候效果不好。Layer Normalization (LN) [9]的提出有效的解决BN的这两个问题。LN和BN不同点是归一化的维度是互相垂直的，如图1所示。在图1中N表示样本轴，C表示通道轴，F是每个通道的特征数量。BN如右侧所示，它是取不同样本的同一个通道的特征做归一化；LN则是如左侧所示，它取的是同一个样本的不同通道做归一化。

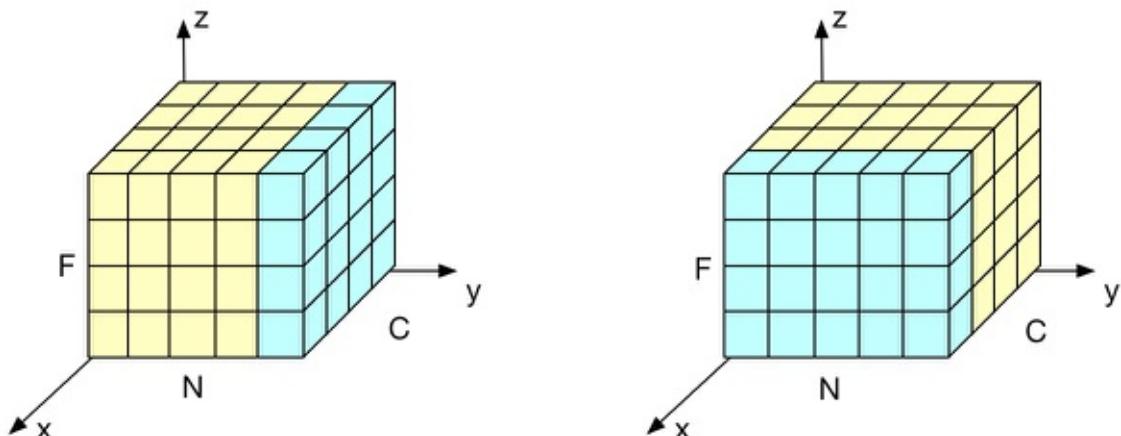


图1：LN(左)和BN(右)对比示意图

1. BN的问题

1.1 BN与Batch Size

如图1右侧部分，BN是按照样本数计算归一化统计量的，当样本数很少时，比如说只有4个。这四个样本的均值和方差便不能反映全局的统计分布信息，所以基于少量样本的BN的效果会变得很差。在一些场景中，比如说硬件资源受限，在线学习等场景，BN是非常不适用的。

1.2 BN与RNN

RNN可以展开成一个隐藏层共享参数的MLP，随着时间片的增多，展开后的MLP的层数也在增多，最终层数由输入数据的时间片的数量决定，所以RNN是一个动态的网络。

在一个batch中，通常各个样本的长度都是不同的，当统计到比较靠后的时间片时，例如图2中 $t > 4$ 时，这时只有一个样本还有数据，基于这个样本的统计信息不能反映全局分布，所以这时BN的效果并不好。

另外如果在测试时我们遇到了长度大于任何一个训练样本的测试样本，我们无法找到保存的归一化统计量，所以BN无法运行。

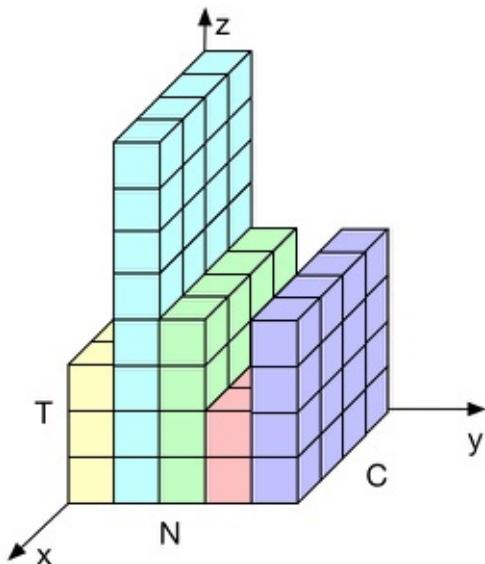


图2：RNN中使用BN会导致batchsize过小的问题

2. LN详解

2.1 MLP中的LN

通过第一节的分析，我们知道BN的两个缺点的产生原因均是因为计算归一化统计量时计算的样本数太少。LN是一个独立于batch size的算法，所以无论样本数多少都不会影响参与LN计算的数据量，从而解决BN的两个问题。LN的做法如图1左侧所示：根据样本的特征数做归一化。

先看MLP中的LN。设 H 是一层中隐层节点的数量， l 是MLP的层数，我们可以计算LN的归一化统计量 μ 和 σ ：

$$\mu^l = \frac{1}{H} \sum_{i=1}^H a_i^l \quad \sigma^l = \sqrt{\frac{1}{H} \sum_{i=1}^H (a_i^l - \mu^l)^2}$$

注意上面统计量的计算是和样本数量没有关系的，它的数量只取决于隐层节点的数量，所以只要隐层节点的数量足够多，我们就能保证LN的归一化统计量足够具有代表性。通过 μ^l 和 σ^l 可以得到归一化后的值 \hat{a}^l ：

$$\hat{a}^l = \frac{\mathbf{a}^l - \mu^l}{\sqrt{(\sigma^l)^2 + \epsilon}}$$

其中 ϵ 是一个很小的小数，防止除0（论文中忽略了这个参数）。

在LN中我们也需要一组参数来保证归一化操作不会破坏之前的信息，在LN中这组参数叫做增益（gain） g 和偏置（bias） b （等同于BN中的 γ 和 β ）。假设激活函数为 f ，最终LN的输出为：

$$\mathbf{h}^l = f(\mathbf{g}^l \odot \hat{\mathbf{a}}^l + \mathbf{b}^l)$$

合并公式(2), (3)并忽略参数 l ，我们有：

$$\mathbf{h} = f\left(\frac{\mathbf{g}}{\sqrt{\sigma^2 + \epsilon}} \odot (\mathbf{a} - \mu) + \mathbf{b}\right)$$

2.2 RNN中的LN

在RNN中，我们可以非常简单的在每个时间片中使用LN，而且在任何时间片我们都能保证归一化统计量统计的是 H 个节点的信息。对于RNN时刻 t 时的节点，其输入是 $t-1$ 时刻的隐层状态 h^{t-1} 和 t 时刻的输入数据 \mathbf{x}_t ，可以表示为：

$$\mathbf{a}^t = W_{hh}h^{t-1} + W_{xh}\mathbf{x}^t$$

接着我们便可以在 \mathbf{a}^t 上采取和1.1节中完全相同的归一化过程：

$$\mathbf{h}^t = f\left(\frac{\mathbf{g}}{\sqrt{(\sigma^t)^2 + \epsilon}} \odot (\mathbf{a}^t - \mu^t) + \mathbf{b}\right) \quad \mu^t = \frac{1}{H} \sum_{i=1}^H a_i^t \quad \sigma^t = \sqrt{\frac{1}{H} \sum_{i=1}^H (a_i^t - \mu^t)^2}$$

2.3 LN与ICS和损失平面平滑

LN能减轻ICS吗？当然可以，至少LN将每个训练样本都归一化到了相同的分布上。而在BN的文章中介绍过几乎所有的归一化方法都能起到平滑损失平面的作用。所以从原理上讲，LN能加速收敛速度的。

3. 对照实验

这里我们设置了一组对照试验来对比普通网络，BN以及LN在MLP和RNN上的表现。这里使用的框架是Keras：

3.1 MLP上的归一化

这里使用的是MNIST数据集，但是归一化操作只添加到了后面的MLP部分。Keras官方源码中没有LN的实现，我们可以通过 `pip install keras-layer-normalization` 进行安装，使用方法见下面代码

```
from keras_layer_normalization import LayerNormalization

# 构建LN CNN网络
model_ln = Sequential()
model_ln.add(Conv2D(input_shape = (28, 28, 1), filters=6, kernel_size=(5,5), padding='valid', activation='tanh'))
model_ln.add(MaxPool2D(pool_size=(2,2), strides=2))
model_ln.add(Conv2D(input_shape=(14, 14, 6), filters=16, kernel_size=(5,5), padding='valid', activation='tanh'))
model_ln.add(MaxPool2D(pool_size=(2,2), strides=2))
model_ln.add(Flatten())
model_ln.add(Dense(120, activation='tanh'))
model_ln.add(LayerNormalization()) # 添加LN运算
model_ln.add(Dense(84, activation='tanh'))
model_ln.add(LayerNormalization())
model_ln.add(Dense(10, activation='softmax'))
```

另外两个对照试验也使用了这个网络结构，不同点在于归一化部分。图3左侧是 `batchsize=128` 时得到的收敛曲线，从中我们可以看出BN和LN均能取得加速收敛的效果，且BN的效果要优于LN。图3右侧是 `batchsize=8` 是得到的收敛曲线，这时BN反而会减慢收敛速度，验证了我们上面的结论，对比之下LN要轻微的优于无归一化的网络，说明了LN在小尺度批量上的有效性。图3的完整代码见连

接：https://github.com/senliuy/keras_layerNorm_mlp_lstm/blob/master/mnist.ipynb。

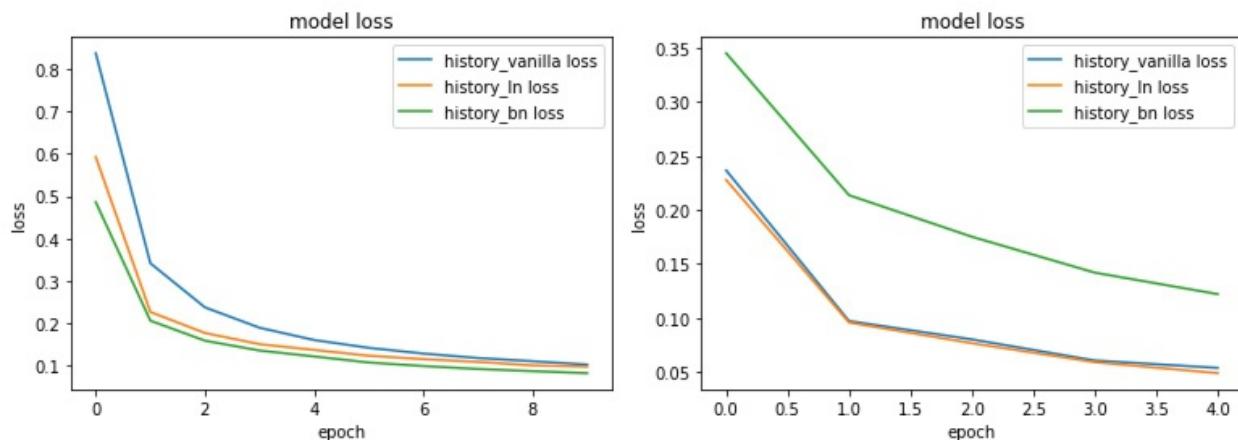


图3： $batchsize=128$ (左)和 $batchsize=8$ (右)损失收敛曲线示意图

3.2 LSTM上的归一化

另外一组对照实验是基于imdb的二分类任务，使用了glove作为词嵌入。这里设置了无LN的LSTM和带LN的LSTM的作为对照试验。LN_LSTM源码参考

<https://github.com/cleemesser/keras-layer-norm-work> 其网络结构如下面代码：

```
# https://github.com/cleemesser/keras-layer-norm-work
from lstm_ln import LSTM_LN
model_ln = Sequential()

model_ln.add(Embedding(max_features, 100))
model_ln.add(LSTM_LN(128))
model_ln.add(Dense(1, activation='sigmoid'))
model_ln.summary()
```

从图4的实验结果我们可以看出LN对于RNN系列动态网络的收敛加速上的效果是略有帮助的。LN的有点主要体现在两个方面：

1. LN得到的模型更稳定；
2. LN有正则化的作用，得到的模型更不容易过拟合。

至于论文中所说的加速收敛的效果，从我的实验上结果上看不到明显的加速。源码见：https://github.com/senliuy/keras_layerNorm_mlp_lstm/blob/master/imdb.ipynb。

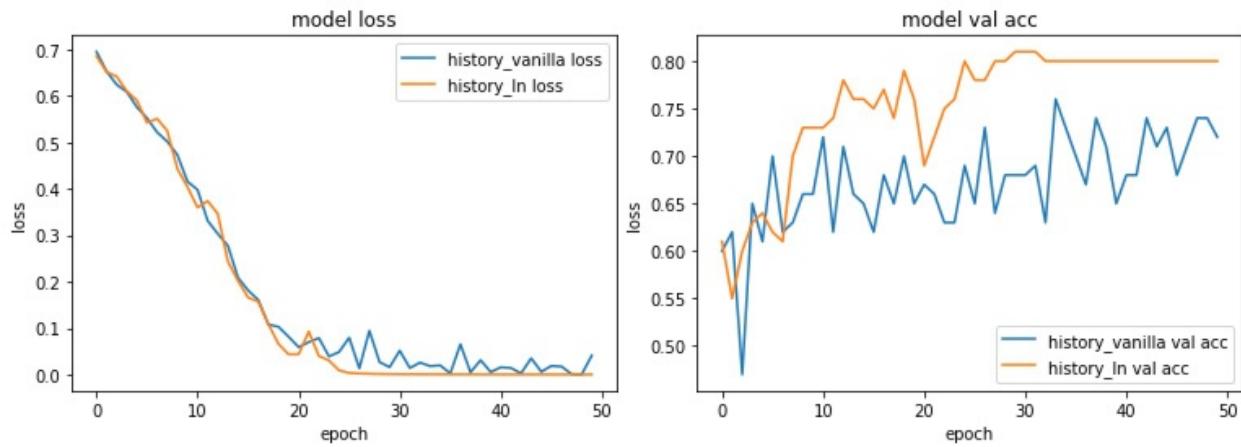


图4：训练集损失值(左)验证集准确率(右)示意图

3.3 CNN上的归一化

我们也尝试了将LN添加到CNN之后，实验结果发现LN破坏了卷积学习到的特征，模型无法收敛，所以在CNN之后使用BN是一个更好的选择。

4. 总结

LN是和BN非常近似的一种归一化方法，不同的是BN取的是不同样本的同一个特征，而LN取的是同一个样本的不同特征。在BN和LN都能使用的场景中，BN的效果一般优于LN，原因是基于不同数据，同一特征得到的归一化特征更不容易损失信息。

但是有些场景是不能使用BN的，例如batchsize较小或者在RNN中，这时候可以选择使用LN，LN得到的模型更稳定且起到正则化的作用。RNN能应用到小批量和RNN中是因为LN的归一化统计量的计算是和batchsize没有关系的。

Weight Normalization

tags : Normalization

前言

之前介绍的BN[2]和LN[3]都是在数据的层面上做的归一化，而这篇文章介绍的Weight Normalization (WN)是在权值的维度上做的归一化。WN的做法是将权值向量 \mathbf{w} 在其欧氏范数和其方向上解耦成了参数向量 \mathbf{v} 和参数标量 g 后使用SGD分别优化这两个参数。

WN也是和样本量无关的，所以可以应用在batchsize较小以及RNN等动态网络中；另外BN使用的基于mini-batch的归一化统计量代替全局统计量，相当于在梯度计算中引入了噪声。而WN则没有这个问题，所以在生成模型，强化学习等噪声敏感的环境中WN的效果也要优于BN。

WN没有一如额外参数，这样更节约显存。同时WN的计算效率也要优于要计算归一化统计量的BN。

1. WN详解

1.1 WN的计算

神经网络的一个节点计算可以表示为：

$$y = \phi(\mathbf{w} \cdot \mathbf{x} + b)$$

其中 \mathbf{w} 是一个 k -维的特征向量， y 是该神经节点的输出，所以是一个标量。在得到损失值后，我们会根据损失函数的值使用SGD等优化策略更新 \mathbf{w} 和 b 。WN提出的归一化策略是将 \mathbf{w} 分解为一个参数向量 \mathbf{v} 和一个参数标量 g ，分解方法为

$$\mathbf{w} = \frac{g}{\|\mathbf{v}\|} \mathbf{v}$$

上式中 $\|\mathbf{v}\|$ 表示 \mathbf{v} 的欧氏范数。当 $\mathbf{v} = \mathbf{w}$ 且 $g = \|\mathbf{w}\|$ 时，WN还原为普通的计算方法，所以WN的网络容量是要大于普通神经网络的。

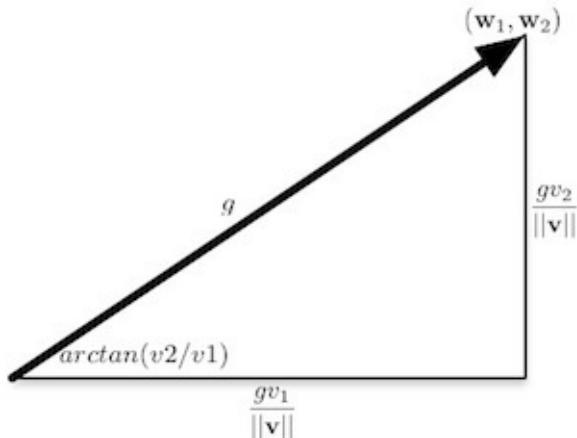


图1：权值向量的分解可视化

当我们将 g 固定为 $\|\mathbf{w}\|$ 时，我们只优化 \mathbf{v} ，这时候相当于只优化 \mathbf{w} 的方向而保留其范数。当 \mathbf{v} 固定为 \mathbf{w} 时，这时候相当于只优化 \mathbf{w} 的范数，而保留其方向，这样为我们优化权值提供了更多可以选择的空间，且解耦方向与范数的策略也能加速其收敛。

在优化 g 时，我们一般通过优化 g 的 \log 级参数 s 来完成，即 $g = e^s$ 。

\mathbf{v} 和 g 的更新值可以通过SGD计算得到：

$$\nabla_g L = \frac{\nabla_{\mathbf{w}} L \cdot \mathbf{v}}{\|\mathbf{v}\|} \quad \nabla_{\mathbf{v}} L = \frac{g}{\|\mathbf{v}\|} \nabla_{\mathbf{w}} L - \frac{g \nabla_g L}{\|\mathbf{v}\|^2} \mathbf{v}$$

其中 L 为损失函数， $\nabla_{\mathbf{w}} L$ 为 \mathbf{w} 在 L 下的梯度值。

从上面WN的计算公式中我们可以看出WN并没有引入新的参数，

1.2 WN的原理

1.1节的梯度更新公式也可以写作：

$$\nabla_{\mathbf{v}} L = \frac{g}{\|\mathbf{v}\|} M_{\mathbf{w}} \nabla_{\mathbf{w}} L \quad \text{with} \quad M_{\mathbf{w}} = I - \frac{\mathbf{w} \mathbf{w}'}{\|\mathbf{w}\|^2}$$

推导方式如下：

$$\begin{aligned}
\nabla_{\mathbf{v}} L &= \frac{g}{\|\mathbf{v}\|} \nabla_{\mathbf{w}} L - \frac{g \nabla_{\mathbf{w}} L}{\|\mathbf{v}\|^2} \mathbf{v} \\
&= \frac{g}{\|\mathbf{v}\|} \nabla_{\mathbf{w}} L - \frac{g}{\|\mathbf{v}\|^2} \frac{\nabla_{\mathbf{w}} L \cdot \mathbf{v}}{\|\mathbf{v}\|} \mathbf{v} \\
&= \frac{g}{\|\mathbf{v}\|} \left(I - \frac{\mathbf{v} \mathbf{v}'}{\|\mathbf{v}\|^2} \right) \nabla_{\mathbf{w}} L \\
&= \frac{g}{\|\mathbf{v}\|} \left(I - \frac{\mathbf{w} \mathbf{w}'}{\|\mathbf{w}\|^2} \right) \nabla_{\mathbf{w}} L \\
&= \frac{g}{\|\mathbf{v}\|} M_{\mathbf{w}} \nabla_{\mathbf{w}} L
\end{aligned}$$

倒数第二步的推导是因为 \mathbf{v} 是 \mathbf{w} 的方向向量。上面公式反应了WN两个重要特征：

1. $\frac{g}{\|\mathbf{v}\|}$ 表明WN会对权值梯度进行 $\frac{g}{\|\mathbf{v}\|}$ 的缩放；
2. $M_{\mathbf{w}} \nabla_{\mathbf{w}} L$ 表明WN会将梯度投影到一个远离于 $\nabla_{\mathbf{w}} L$ 的方向。

这两个特征都会加速模型的收敛。

具体原因论文的说法比较复杂，其核心思想有两点：1. 由于 \mathbf{w} 垂直于 $M_{\mathbf{w}}$ ，所以 $\nabla_{\mathbf{v}} L$ 非常接近于垂直参数方向 \mathbf{w} ，这样对于矫正梯度更新方向是非常有效的；2. \mathbf{v} 和梯度更新值中的噪声量成正比，而 \mathbf{v} 是和更新量成反比，所以当更新值中噪音较多时，更新值会变小，这说明WN有自稳定（self-stabilize）的作用。这个特点使得我们可以在WN中使用比较大的学习率。¹

另一个角度从新权值的协方差矩阵出发的，假设 \mathbf{w} 的协方差矩阵是 \mathbf{C} ，那么 \mathbf{v} 的协方差矩阵

$\mathbf{D} = (g^2 / \|\mathbf{v}\|^2) M_{\mathbf{w}} \mathbf{C} M_{\mathbf{w}}$ ，当去掉 \mathbf{D} 中的特征值后我们发现新的 \mathbf{D} 非常趋近于一个单位矩阵，这说明了 \mathbf{w} 是 \mathbf{C} 的主特征向量（dominant eigenvector），说明WN有助于提升收敛速度。

1.3 BN和WN的关系

假设 $t = \mathbf{v} \mathbf{x}$ ， $\mu[t]$ 和 $\sigma[t]$ 分别为 t 的均值和方差，BN可以表示为：

$$t' = \frac{t - \mu[t]}{\sigma[t]} = \frac{\mathbf{v}}{\sigma[t]} \mathbf{x} - \frac{\mu[t]}{\sigma[t]}$$

当网络只有一层且输入样本服从均值为0，方差为1的独立分布时，我们有 $\mu[t] = 0$ 且 $\sigma[t] = \|\mathbf{v}\|$ ，此时WN和BN等价。

1.4 WN的参数初始化

由于WN不像BN有规范化特征尺度的作用，所以WN的初始化需要慎重。作者建议的初始化策略是：

- \mathbf{v} 使用均值为0，标准差为0.05的正态分布进行初始化；

- g 和偏置 b 使用第一批训练样本的统计量进行初始化：

$$g \leftarrow \frac{1}{\sigma[t]} \quad b \leftarrow \frac{-\mu[t]}{\sigma[t]}$$

由于使用了样本进行初始化，所以这种初始化方法不适用于RNN等动态网络。

1.5 Mean-Only BN

基于WN的动机，文章提出了Mean-Only BN。这种方法是一个只进行减均值而不进行除方差的BN，动机是考虑到BN的除方差操作会引入额外的噪声，实验结果表明WN+Mean-Only BN虽然比标准BN收敛得慢，但它们在测试集的精度要高于BN。

2. 总结

和目前主流归一化方法不同的是，WN的归一化操作作用在了权值矩阵之上。从其计算方法上来看，WN完全不像一个归一化方法，更像是基于矩阵分解的一种优化策略，它带来了四点好处：

1. 更快的收敛速度；
2. 更强的学习率鲁棒性；
3. 可以应用在RNN等动态网络中；
4. 对噪声更不敏感，更适用在GAN，RL等场景中。

说WN不像归一化的原因是它并没有对得到的特征范围进行约束的功能，所以WN依旧对参数的初始值非常敏感，这也是WN一个比较严重的问题。

¹ 这是我对于论文的2.1节的比较主观的个人理解，当初看的时候就非常头疼，理解可能有偏差，希望各位读者给出正确的批评指正。 ↪

Instance Normalization

tags: Normalization

前言

对于我们之前介绍过的[图像风格迁移\[7\]](#)这类的注重每个像素的任务来说，每个样本的每个像素点的信息都是非常重要的，于是像[BN\[114\]](#)这种每个批量的所有样本都做归一化的算法就不太适用了，因为BN计算归一化统计量时考虑了一个批量中所有图片的内容，从而造成了每个样本独特细节的丢失。同理对于[LN\[9\]](#)这类需要考虑一个样本所有通道的算法来说可能忽略了不同通道的差异，也不太适用于图像风格迁移这类应用。

所以这篇文章提出了Instance Normalization (IN) [6]，一种更适合对单个像素有更高要求的场景的归一化算法 (IST, GAN等)。IN的算法非常简单，计算归一化统计量时考虑单个样本，单个通道的所有元素。IN (右) 和BN (中) 以及LN (左) 的不同从图1中可以非常明显的看出。

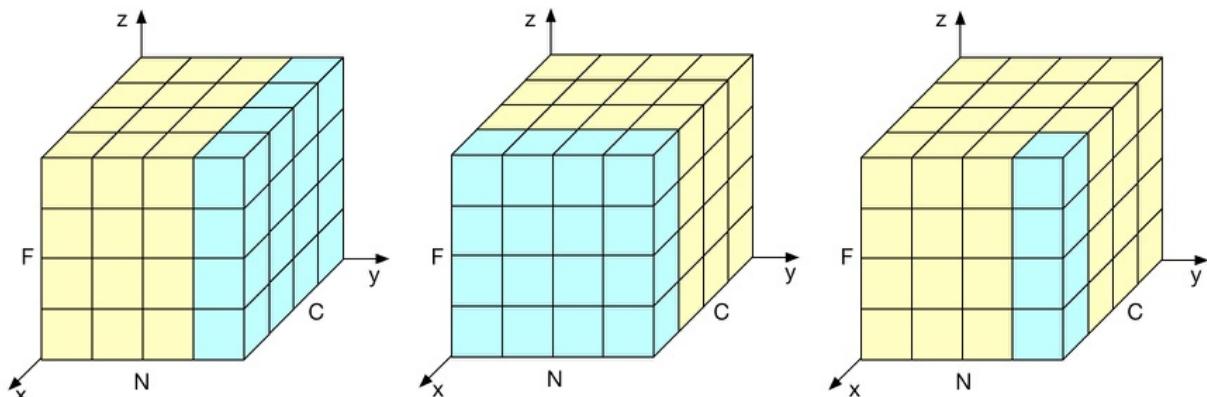


图1 : LN (左) , BN (中) , IN (右)

1. IN详解

1.1 IST中的IN

在Gatys等人的IST算法中，他们提出的策略是通过L-BFGS算法优化生成图片，风格图片以及内容图片再VGG-19上生成的Feature Map的均方误差。这种策略由于Feature Map的像素点数量过于多导致了优化起来非常消耗时间以及内存。IN的作者Ulyanov等人同在2016年提出了

Texture network[5] (图2) ,

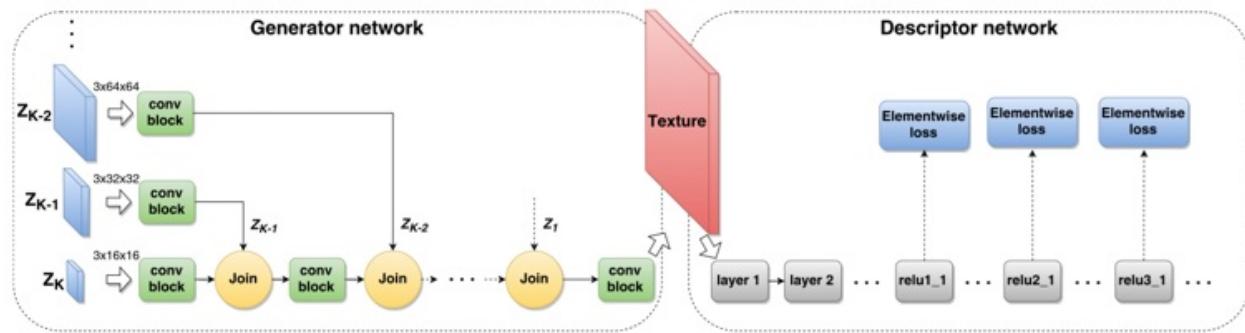


图2 : *Texture Networks*的网络结

图2中的生成器网络 (Generator Network) 是一个由卷积操作构成的全卷积网络，在原始的 Texture Network 中，生成器使用的操作包括卷积，池化，上采样以及BN。但是作者发现当训练生成器网络网络时，使用的样本数越少（例如16个），得到的效果越好。但是我们知道BN并不适用于样本数非常少的环境中，因此作者提出了IN，一种不受限于批量大小的算法专门用于Texture Network中的生成器网络。

1.2 IN vs BN

BN的详细算法我们已经分析过，这里再重复一下它的计算方式：

$$\mu_i = \frac{1}{HWT} \sum_{t=1}^T \sum_{l=1}^W \sum_{m=1}^H x_{tilm} \quad \sigma_i^2 = \frac{1}{HWT} \sum_{t=1}^T \sum_{l=1}^W \sum_{m=1}^H (x_{tilm} - \mu_i)^2 \quad y_{tijk} = \frac{x_{tijk}}{\sqrt{\sigma_i^2 + \epsilon}}$$

正如我们之前所分析的，IN在计算归一化统计量时并没有像BN那样跨样本、单通道，也没有像LN那样单样本、跨通道。它是取的单通道，单样本上的数据进行计算，如图1最右侧所示。所以对比BN的公式，它只需要它只需要去掉批量维的求和即可：

$$\mu_{ti} = \frac{1}{HW} \sum_{l=1}^W \sum_{m=1}^H x_{tilm} \quad \sigma_{ti}^2 = \frac{1}{HW} \sum_{l=1}^W \sum_{m=1}^H (x_{tilm} - \mu_{ti})^2 \quad y_{tijk} = \frac{x_{tijk}}{\sqrt{\sigma_{ti}^2 + \epsilon}}$$

对于是否使用BN中的可学习参数 β 和 γ ，从LN的TensorFlow中源码中我们可以看出这两个参数是要使用的。但是我们也可以通过将其值置为False来停用它们，这一点和其它归一化方法在TensorFlow中的实现是相同的。

1.3 TensorFlow 中的IN

IN在TensorFlow中的实现见[链接](#)，其函数声明如下：

```
def instance_norm(inputs,
                  center=True,
                  scale=True,
                  epsilon=1e-6,
                  activation_fn=None,
                  param_initializer=None,
                  reuse=None,
                  variables_collections=None,
                  outputs_collections=None,
                  trainable=True,
                  data_format=DATA_FORMAT_NHWC,
                  scope=None)
```

其中的 `center` 和 `scale` 便是分别对应BN中的参数 β 和 γ 。

归一化统计量是通过 `nn.moments` 函数计算的，决定如何从`inputs`取值的是 `axes` 参数，对应源码中的 `moments_axes` 参数。

```
# Calculate the moments (instance activations).
mean, variance = nn.moments(inputs, moments_axes, keep_dims=True)
```

下面我们提取源码中的核心部分，并通过注释的方法对齐进行解释（假设输入的Tensor是按NHWC排列的）：

```
inputs_rank = inputs.shape.ndims # 取Tensor的维度数，这里值是4
reduction_axis = inputs_rank - 1 # 取Channel维的位置，值为3
moments_axes = list(range(inputs_rank)) # 初始化moments_axes链表，值为[0,1,2,3]
del moments_axes[reduction_axis] # 删除第3个值（Channel维），moments_axes变为[0,1,2]
del moments_axes[0] # 删除第一个值（Batch维），moments_axes变为[1,2]
```

总结

IN本身是一个非常简单的算法，尤其适用于批量较小且单独考虑每个像素点的场景中，因为其计算归一化统计量时没有混合批量和通道之间的数据，对于这种场景下的应用，我们可以考虑使用IN。

另外需要注意的一点是在图像这类应用中，每个通道上的值是比较大的，因此也能够取得比较合适的归一化统计量。但是有两个场景建议不要使用IN:

1. MLP或者RNN中：因为在MLP或者RNN中，每个通道上只有一个数据，这时会自然不能使用IN；
2. Feature Map比较小时：因为此时IN的采样数据非常少，得到的归一化统计量将不再具有代表性。

Group Normalization

tags: Normalization

前言

Group Normalization (GN) [4]是何恺明团队提出的一种归一化策略，它是介于[Layer Normalization \(LN\) \[9\]](#)和[Instance Normalization \(IN\) \[6\]](#)之间的一种折中方案，图1最右。它通过将通道数据分成几组计算归一化统计量，因此GN也是和批量大小无关的算法，因此可以用在batchsize比较小的环境中。作者在论文中指出GN要比LN和IN的效果要好。

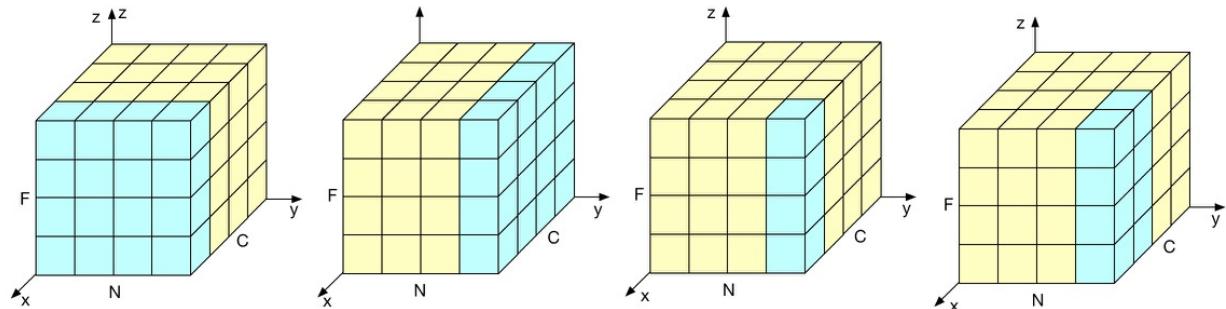


图1：从左到右依次是BN，LN，IN以及GN

1. GN详解

1.1 GN算法

和之前所有介绍过的归一化算法相同，GN也是根据该层的输入数据计算均值和方差，然后使用这两个值更新输入数据：

$$\mu_i = \frac{1}{m} \sum_{k \in \mathcal{S}_i} x_k \quad \sigma_i = \sqrt{\frac{1}{m} \sum_{k \in \mathcal{S}_i} (x_k - \mu_i)^2 + \epsilon} \quad \hat{x}_i = \frac{1}{\sigma_i} (x_i - \mu_i)$$

之前所介绍的所有归一化方法均可以使用上面式子进行概括，区别它们的是 \mathcal{S}_i 是如何取得的：

对于BN来说，它是取不同batch的同一个channel上的所有的值：

$$\mathcal{S}_i = \{k | k_C = i_C\}$$

而LN是从同一个batch的不同的channel上取所有的值：

$$\mathcal{S}_i = \{k | k_N = i_N\}$$

IN即不跨batch，也不跨channel：

$$\mathcal{S}_i = \{k | k_N = i_N, k_C = i_C\}$$

GN是将Channel分成若干组，只使用组内的数据计算均值和方差。通常组数G是一个超参数，TensorFlow中的默认值是32。

$$\mathcal{S}_i = \{k | k_N = i_N, \lfloor \frac{k_C}{C/G} \rfloor = \lfloor \frac{i_C}{C/G} \rfloor\}$$

我们可以看出，当GN的组数为1时，此时GN和LN等价；当GN的组数为通道数时，GN和IN等价。

GN和其它算法一样也可以添加参数 γ 和 β 来保证网络的容量。

1.2 GN的伪代码

论文中给出了基于TensorFlow的GN的源码：

```

1 def GroupNorm(x, gamma, beta, G, eps=1e-5):
2     # x: input features with shape [N,C,H,W]
3     # gamma, beta: scale and offset, with shape [1,C,1,1]
4     # G: number of groups for GN
5     N, C, H, W = x.shape
6     x = tf.reshape(x, [N, G, C // G, H, W])
7     mean, var = tf.nn.moments(x, [2, 3, 4], keep dims=True)
8     x = (x - mean) / tf.sqrt(var + eps)
9     x = tf.reshape(x, [N, C, H, W])
10    return x * gamma + beta

```

第6行代码将Tensor中添加一个‘组’的维度，形成一个五维张量。第7行的`axes`的值为[2,3,4]表明计算归一化统计量时即不会跨batch，也不会跨组。

1.2 GN的原理

在深度学习之前，传统的SIFT，HOG等算法均由按组统计特征的特性，它们一般将同一个种类的特征归为一组，然后在进行组归一化。在深度学习中，每个通道的Feature Map也可以看做结构化的特征向量。如果一个Feature Map的卷积数足够多，那么必然有一些通道的特征是

类似的，因此我们可以将这些类似的特征进行归一化处理。

作者认为，GN比LN效果好的原因是GN比LN的限制更少，因为LN假设了一个层的所有通道的数据共享一个均值和方差。而IN则丢失了探索通道之间依赖性的能力。

总结

作为一种介于IN和LN之间的归一化策略，GN的效果反而优于另外两个算法，这令我非常困惑。虽然作者也尝试给出解释，但总是感觉这个解释有些过于主观，有根据结果推导原因的嫌疑。另外我也做了一些归一化方法的对比实验，实验结果并不如作者所说的那么理想。所以我们在设计网络时，如果batchsize尺寸可以做的比较大，BN仍旧是最优的选择。但是如果batchsize比较小，也许通过对照实验选出最好的归一化策略是最优的选择。

Switchable Normalization

tags: Normalization

前言

在之前的文章中，我们介绍了BN[114]，LN[9]，IN[6]以及GN[4]的算法细节及适用的任务。虽然这些归一化方法往往能提升模型的性能，但是当你接收一个任务时，具体选择哪个归一化方法仍然需要人工选择，这往往需要大量的对照实验或者开发者优秀的经验才能选出最合适归一化方法。本文提出了Switchable Normalization (SN) [3]，它的算法核心在于提出了一个可微的归一化层，可以让模型根据数据来学习到每一层该选择的归一化方法，亦或是三个归一化方法的加权和，如图1所示。所以SN是一个任务无关的归一化方法，不管是LN适用的RNN还是IN适用的图像风格迁移(IST)，SN均能用到该应用中。作者在实验中直接将SN用到了包括分类，检测，分割，IST，LSTM等各个方向的任务中，SN均取得了非常好的效果。

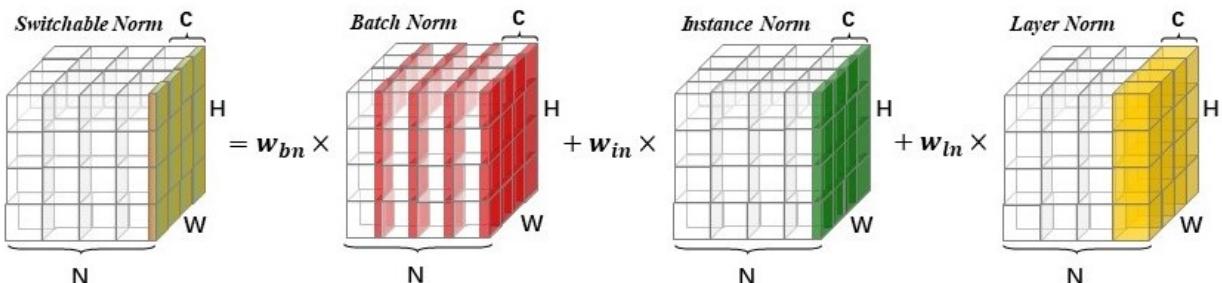


图1：SN是LN，BN以及IN的加权和

1. SN详解

1.1 回顾

SN实现了对BN，LN以及IN的统一。以CNN为例，假设一个4D Feature Map的尺寸为

(N, C, W, H) ，假设 h_{ncij} 和 \hat{h}_{ncij} 分别是归一化前后的像素点的值，其中 $n \in [1, N]$ ， $c \in [1, C]$ ， $i \in [1, H]$ ， $j \in [1, W]$ 。假设 μ 和 σ 分别是均值和方差，上面所介绍的所有归一化方法均可以表示为：

$$\hat{h}_{ncij} = \gamma \frac{h_{ncij} - \mu}{\sqrt{\sigma^2 + \epsilon}} + \beta$$

其中 β 和 γ 分别是位移变量和缩放变量， ϵ 是一个非常小的数用以防止除0。上面式子概括了BN，LN，以及IN三种归一化的计算公式，唯一不同是计算 μ 和 σ 统计的像素点不同。我们可以将 μ 和 σ 表示为：

$$\mu_k = \frac{1}{I_k} \sum_{(n,c,i,j) \in I_k} h_{ncij}, \quad \sigma_k^2 = \frac{1}{I_k} \sum_{(n,c,i,j) \in I_k} (h_{ncij} - \mu_k)^2$$

其中 $k \in \{in, ln, bn\}$ 。IN是统计的是单个批量，单个通道的所有像素点，如图1绿色部分。BN统计的是单个通道上所有像素点，如图1红色部分。LN统计的是单个批量上的所有像素点，如图1黄色部分。它们依次可以表示为 $I_{in} = \{(i, j) | i \in [1, H], j \in [1, W]\}$ ， $I_{bn} = \{(i, j) | n \in [1, N], i \in [1, H], j \in [1, W]\}$ ， $I_{ln} = \{(i, j) | c \in [1, C], i \in [1, H], j \in [1, W]\}$ 。

1.2 SN算法介绍

SN算法是为三组不同的 μ_k 以及 σ_k 分别学习三个总共6个标量值(w_k 和 w'_k)， \hat{h}_{ncij} 的计算使用的是它们的加权和：

$$\hat{h}_{ncij} = \gamma \frac{h_{ncij}}{\sum_{k \in \Omega} w_k \mu_k} + \beta \sqrt{\sum_{k \in \Omega} w'_k \sigma_k^2} + \epsilon$$

其中 $\Omega = \{in, ln, bn\}$ 。在计算 (μ_{ln}, σ_{ln}) 和 (μ_{bn}, σ_{bn}) 时，我们可以使用 (μ_{in}, σ_{in}) 作为中间变量以减少计算量。

$$\begin{aligned} \mu_{in} &= \frac{1}{HW} \sum_{i,j}^{H,W} h_{ncij} & \sigma_{in}^2 &= \frac{1}{HW} \sum_{i,j}^{H,W} (h_{ncij} - \mu_{in})^2 \\ \mu_{ln} &= \frac{1}{C} \sum_{c=1}^C \mu_{in} & \sigma_{ln}^2 &= \frac{1}{C} \sum_{c=1}^C (\sigma_{in}^2 + \mu_{in}^2) - \mu_{ln}^2 \\ \mu_{bn} &= \frac{1}{N} \sum_{n=1}^N \mu_{in} & \sigma_{bn}^2 &= \frac{1}{N} \sum_{n=1}^N (\sigma_{in}^2 + \mu_{in}^2) - \mu_{bn}^2 \end{aligned}$$

w_k 是通过softmax计算得到的激活函数：

$$w_k = \frac{e^{\lambda_k}}{\sum_{z \in \{in, ln, bn\}} e^{\lambda_z}} \quad \text{and} \quad k \in \{in, ln, bn\}$$

其中 $\{\lambda_{in}, \lambda_{bn}, \lambda_{ln}\}$ 是需要优化的3个参数，可以通过BP调整它们的值。同理我们也可以计算 w' 对应的参数值 $\{\lambda'_{in}, \lambda'_{bn}, \lambda'_{ln}\}$ 。

从上面的分析中我们可以看出，SN只增加了6个参数 $\Phi = \{\lambda_{in}, \lambda_{bn}, \lambda_{ln}, \lambda'_{in}, \lambda'_{bn}, \lambda'_{ln}\}$ 。假设原始网络的参数集为 Θ ，带有SN的网络的损失函数可以表示为 $\mathcal{L}(\Theta, \Phi)$ ，他可以通过BP联合优化 Θ 和 Φ 。对SN的反向推导感兴趣的同学参考论文附件H。

1.3 测试

在BN的测试过程中，为了计算其归一化统计量，传统的BN方法是从训练过程中利用滑动平均的方法得到的均值和方差。在SN的BN部分，它使用的是另一种叫做批平均batch average的方法，它分成两步：1. 固定网络中的SN层，从训练集中随机抽取若干个批量的样本，将输入输入到网络中；2. 计算这些批量在特定SN层的 μ 和 σ 的平均值，它们将会作为测试阶段的均值和方差。实验结果表明，在SN中批平均的效果略微优于滑动平均。

2. SN的优点

2.1 SN的普遍适用性

SN通过根据不同的任务调整不同归一化策略的权值使其可以直接应用到不同的任务中。图2可视化了在不同任务上不同归一化策略的权值比重：

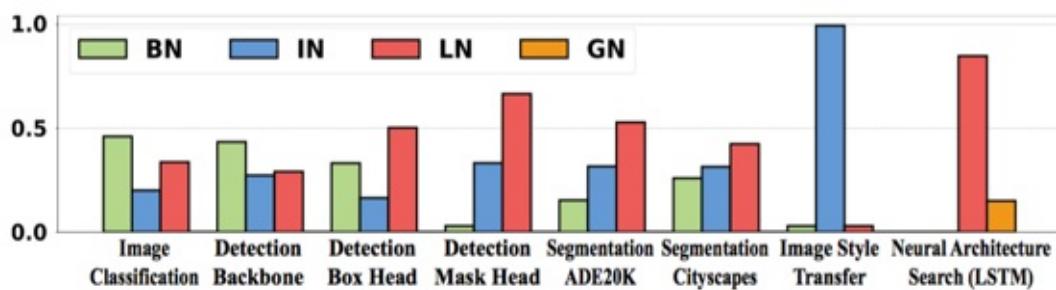


图2：SN在不同任务下的权值分布可视化图

从图2中我们可以看出LSTM以及IST都学到了最适合它们本身的归一化策略。

2.2 SN与BatchSize

SN也能根据batchsize的大小自动调整不同归一化策略的比重，如果batchsize的值比较小，SN学到的BN的权重就会很小，反之BN的权重就会很大，如图3所示：

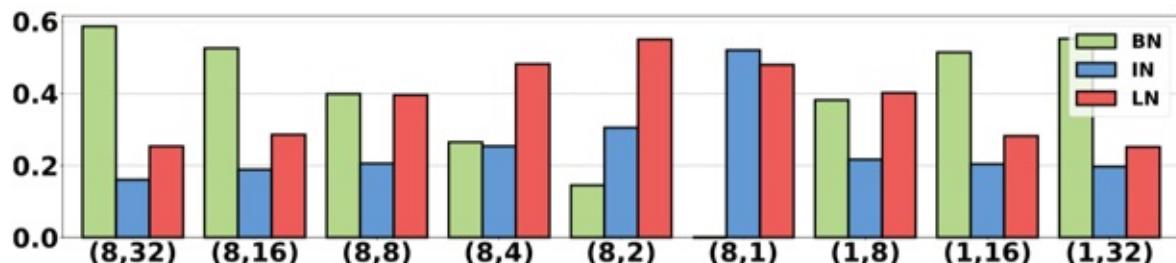


图3：SN在不同batchsize下的权值分布可视化图

图3中括号的意思是(#GPU, batchsize)。

3. 总结

这篇文章介绍了统一了BN, LN以及IN三种归一化策略的SN, SN具有以下三个有点：

1. 鲁棒性：无论batchsize的大小如何，SN均能取得非常好的效果；
2. 通用性：SN可以直接应用到各种类型的应用中，减去了人工选择归一化策略的繁琐；
3. 多样性：由于网络的不同层在网络中起着不同的作用，SN能够为每层学到不同的归一化策略，这种自适应的归一化策略往往要优于单一方案人工设定的归一化策略。

Generative Adversarial Nets

前言

生成对抗网络（Generative Adversarial Nets，GAN）[\[23\]](#)自2014年被提出以来便受到了业内广泛的关注，Yann LeCun将GAN赞誉为“这几年最棒的想法”。GAN作为无监督学习的一个分支被广泛的应用于数据集的生成。

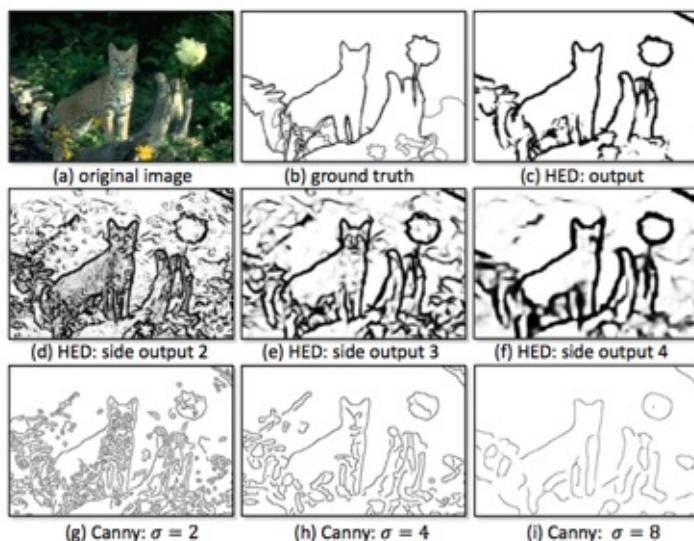
Holistically-Nested Edge Detection

tags: HED, Edge Detection

前言

本文提出了一个新的网络结构用于边缘检测，即本文的题目 Holistically-Nested Network (HED)。其中 Holistically 表示该算法试图训练一个 image-to-image 的网络；Nested 则强调在生成的输出过程中通过不断的集成和学习得到更精确的边缘预测图的过程。从图1中 HED 和传统 Canny 算法进行边缘检测的效果对比图我们可以看到 HED 的效果要明显优于 Canny 算子的。

图1：HED vs Canny



由于 HED 是 image-to-image 的，所以该算法也很容易扩展到例如语义分割的其它领域。此外在 OCR 中的文字检测中，文字区域往往具有比较强的边缘特征，因此 HED 也可以扩展到场景文字检测中，著名的 EAST [2] 算法便得到了 HED 的启发。

下面我们结合 HED 的 [Keras 源码](#) 对 HED 展开详细分析。

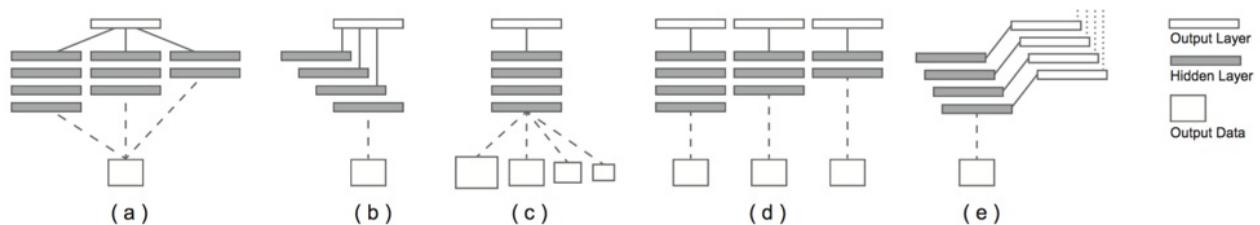
1.1 HED 的骨干网络

HED 创作于 2015 年，使用了当时 state-of-the-art 的 VGG-16 作为骨干网络，并且使用迁移学习初始化了网络权重。

HED 使用了多尺度的特征，类似多尺度特征的思想还有 Inception，SSD，FPN 等方法，对比如图 2。

- (a) Multi-stream learning: 使用不同结构，不同参数的网络训练同一副图片，类似的结构有Inception；
- (b) Skip-layer network learning: 该结构有一个主干网络，在主干网络中添加若干条到输出层的skip-layer，类似的结构有FPN；
- (c) Single model on multiple inputs: 该方法使用同一个网络，不同尺寸的输入图像得到不同尺度分Feature Map，YOLOv2采用了该方法；
- (d) Training independent network: 使用完全独立的网络训练同一张图片，得到多个尺度的结果，该方法类似于集成模型；
- (e) Holistically-Nested networks: HED采用的方法，下面详细介绍。

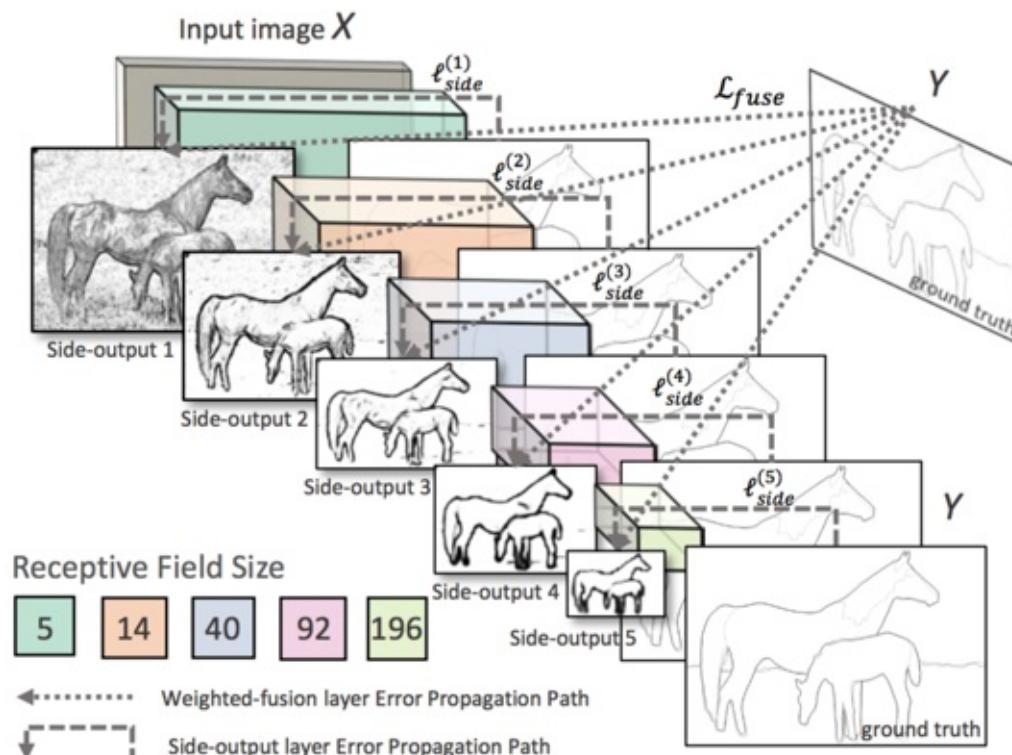
图2：几种提取多尺度特征的算法的网络结构



1.2 Holistically-Nested networks

Holistically-Nested networks的结构如图3以及下面代码：

图3：Holistically-Nested networks结构图



```

# Input
img_input = Input(shape=(480, 480, 3), name='input')
# Block 1
x = Conv2D(64, (3, 3), activation='relu', padding='same', name='block1_conv1')(img_input)
x = Conv2D(64, (3, 3), activation='relu', padding='same', name='block1_conv2')(x)
b1= side_branch(x, 1) # 480 480 1
x = MaxPooling2D((2, 2), strides=(2, 2), padding='same', name='block1_pool')(x) # 240
240 64
# Block 2
x = Conv2D(128, (3, 3), activation='relu', padding='same', name='block2_conv1')(x)
x = Conv2D(128, (3, 3), activation='relu', padding='same', name='block2_conv2')(x)
b2= side_branch(x, 2) # 480 480 1
x = MaxPooling2D((2, 2), strides=(2, 2), padding='same', name='block2_pool')(x) # 120
120 128
# Block 3
x = Conv2D(256, (3, 3), activation='relu', padding='same', name='block3_conv1')(x)
x = Conv2D(256, (3, 3), activation='relu', padding='same', name='block3_conv2')(x)
x = Conv2D(256, (3, 3), activation='relu', padding='same', name='block3_conv3')(x)
b3= side_branch(x, 4) # 480 480 1
x = MaxPooling2D((2, 2), strides=(2, 2), padding='same', name='block3_pool')(x) # 60
60 256
# Block 4
x = Conv2D(512, (3, 3), activation='relu', padding='same', name='block4_conv1')(x)
x = Conv2D(512, (3, 3), activation='relu', padding='same', name='block4_conv2')(x)
x = Conv2D(512, (3, 3), activation='relu', padding='same', name='block4_conv3')(x)
b4= side_branch(x, 8) # 480 480 1
x = MaxPooling2D((2, 2), strides=(2, 2), padding='same', name='block4_pool')(x) # 30
30 512
# Block 5
x = Conv2D(512, (3, 3), activation='relu', padding='same', name='block5_conv1')(x)
x = Conv2D(512, (3, 3), activation='relu', padding='same', name='block5_conv2')(x)
x = Conv2D(512, (3, 3), activation='relu', padding='same', name='block5_conv3')(x) # 3
0 30 512
b5= side_branch(x, 16) # 480 480 1
# fuse
fuse = Concatenate(axis=-1)([b1, b2, b3, b4, b5])
fuse = Conv2D(1, (1,1), padding='same', use_bias=False, activation=None)(fuse) # 480
480 1
# outputs
o1 = Activation('sigmoid', name='o1')(b1)
o2 = Activation('sigmoid', name='o2')(b2)
o3 = Activation('sigmoid', name='o3')(b3)
o4 = Activation('sigmoid', name='o4')(b4)
o5 = Activation('sigmoid', name='o5')(b5)
ofuse = Activation('sigmoid', name='ofuse')(fuse)
# model
model = Model(inputs=[img_input], outputs=[o1, o2, o3, o4, o5, ofuse])

```

无论从图3还是源码，VGG-16的骨干架构是非常明显的。在VGG-16的5个block的Max Pooling降采样之前，HED通过side_branch函数产生了5个分支，side_branch的源码如下

```

def side_branch(x, factor):
    x = Conv2D(1, (1, 1), activation=None, padding='same')(x)
    kernel_size = (2*factor, 2*factor)
    x = Conv2DTranspose(1, kernel_size, strides=factor, padding='same', use_bias=False
    , activation=None)(x)
    return x

```

其中Conv2DTranspose是反卷积操作，side_branch的输出特征向量的维度已反应在注释中。HED利用反卷积进行上采样的方法类似于DSSD。

HED的fuse branch层是由5个side_branch的输出通过Concatenate操作合并而成的。网络的5个side_branch和一个fuse branch通过sigmoid激活函数后共同作为网络的输出，每个输出的尺寸均和输入图像相同。

1.3 HED的损失函数

1.3.1 训练

设HED的训练集为 $S = \{(X_n, Y_n), n = 1, \dots, N\}$ ，其中 $X_n = \{x_j^{(n)}, j = 1, \dots, |X_n|\}$ 表示原始输入图像， $Y_n = \{y_j^{(n)}, j = 1, \dots, |X_n|\}$ 表示 X_n 的二进制边缘标签map，故 $y_j^{(n)} \in \{0, 1\}$ ， $|X_n|$ 是一张图像的像素点的个数。

假设VGG-16的网络的所有参数值为 \mathbf{W} ，如果网络有 M 个side branch的话，那么定义side branch的参数值为 $\mathbf{w} = (\mathbf{w}^{(1)}, \dots, \mathbf{w}^{(M)})$ ，则HED关于side branch的目标函数定义为：

$$\mathcal{L}_{\text{side}}(\mathbf{W}, \mathbf{w}) = \sum_{m=1}^M \alpha_m \ell_{\text{side}}^{(m)}(\mathbf{W}, \mathbf{w}^{(m)})$$

其中 α_m 表示每个side branch的损失函数的权值，可以根据训练日志进行调整或者均为 $1/5$ 。

$\ell_{\text{side}}^{(m)}(\mathbf{W}, \mathbf{w}^{(m)})$ 是每个side branch的损失函数，该损失函数是一个类别平衡的交叉熵损失函数：

$$\ell_{\text{side}}^{(m)}(\mathbf{W}, \mathbf{w}^{(m)}) = -\beta \sum_{j \in Y_+} \log \Pr(y_j = 1 | X; \mathbf{W}, \mathbf{w}^{(m)}) - (1 - \beta) \sum_{j \in Y_-} \log \Pr(y_j = 0 | X; \mathbf{W}, \mathbf{w}^{(m)})$$

其中 β 适用于平衡边缘检测的正负样本不均衡的类别平衡权值，其中 $\beta = \frac{|Y_-|}{|Y|}$ ， $1 - \beta = \frac{|Y_+|}{|Y|}$ 。

$|Y_+|$ 表示非边缘像素的个数，那么 $|Y_-|$ 则表示边缘像素的个数。

$\hat{Y}_{\text{side}}^{(m)} = \Pr(y_j = 1 | X; \mathbf{W}, \mathbf{w}^{(m)}) = \sigma(a_j^{(m)})$ 表示第 m 个 side branch 在第 j 个像素处预测的边缘值, $\sigma()$ 是 sigmoid 激活函数。

类别平衡损失函数实现如下

```
def cross_entropy_balanced(y_true, y_pred):
    _epsilon = _to_tensor(K.epsilon(), y_pred.dtype.base_dtype)
    y_pred = tf.clip_by_value(y_pred, _epsilon, 1 - _epsilon)
    y_pred = tf.log(y_pred / (1 - y_pred))
    y_true = tf.cast(y_true, tf.float32)
    count_neg = tf.reduce_sum(1. - y_true)
    count_pos = tf.reduce_sum(y_true)
    beta = count_neg / (count_neg + count_pos)
    pos_weight = beta / (1 - beta)
    cost = tf.nn.weighted_cross_entropy_with_logits(logits=y_pred, targets=y_true, pos_weight=pos_weight)
    cost = tf.reduce_mean(cost * (1 - beta))
    return tf.where(tf.equal(count_pos, 0.0), 0.0, cost)
```

如图3所示, fuse 层表示为 m 个 side branch 的加权和 (代码中的 1×1 卷积起到的作用), 即

$\hat{Y}_{\text{fuse}} \equiv \sigma(\sum_{m=1}^M h_m \hat{A}_{\text{side}}^{(m)})$, fuse 层的损失函数 1 定义为:

$$\mathcal{L}_{\text{fuse}}(\mathbf{W}, \mathbf{w}, \mathbf{h}) = \text{Dist}(Y, \hat{Y}_{\text{fuse}})$$

其中 $\text{Dist}(\cdot, \cdot)$ 表示交叉熵损失函数。源码中使用的是类别平衡的交叉熵损失函数, 个人认为源码中的方案更科学。

最后, 训练模型时的目标函数便是最小化 side branch 损失 $\mathcal{L}_{\text{side}}(\mathbf{W}, \mathbf{w})$ 以及 fuse 损失

$\mathcal{L}_{\text{fuse}}(\mathbf{W}, \mathbf{w}, \mathbf{h})$ 的和:

$$(\mathbf{W}, \mathbf{w}, \mathbf{h})^* = \operatorname{argmin}(\mathcal{L}_{\text{side}}(\mathbf{W} + \mathcal{L}_{\text{fuse}}(\mathbf{W}, \mathbf{w}, \mathbf{h})))$$

1.3.2 测试

给定一张图片 X , HED 预测 M 个 side branch 和一个 fuse layer :

$$(\hat{Y}_{\text{fuse}}, \hat{Y}_{\text{side}}^{(1)}, \dots, \hat{Y}_{\text{side}}^{(1)}) = \text{CNN}(X, (\mathbf{W}, \mathbf{w}, \mathbf{h})^*)$$

HED 的输出是所以 side branch 和 fuse layer 的均值:

$$\hat{Y}_{\text{HED}} = \text{Average}(\hat{Y}_{\text{fuse}}, \hat{Y}_{\text{side}}^{(1)}, \dots, \hat{Y}_{\text{side}}^{(1)})$$

总结

我是在研究EAST的时候读到的这篇论文，EAST算法的核心之一是使用语义分割构建损失函数，而其语义分割的标签便是由类似HED的结构得到的。

从HED的实验结果可以看出，其边缘检测的效果着实经验，且测试非常快，具有非常光明的应用前景。

HED的缺点是模型过于庞大，Keras训练的模型超过了100MB，原因是fuse layer合并了VGG-16每个block的Feature Map，且每个side branch的尺寸均为输入图像的大小。由此引发了HED训练过程中显存占用问题，不过在目前GPU环境下训练HED算法还是没有问题的。

Image Style Transfer Using Convolutional Nerual Networks

tags: Nueral Style Transfer

前言

Leon A.Gatys是最早使用CNN做图像风格迁移[7]的先驱之一，这篇文章还有另外一个版本[2]，应该是它投到CVPR之前的预印版，两篇文章内容基本相同。

我们知道在训练CNN分类器时，接近输入层的Feature Map包含更多的图像的纹理等细节信息，而接近输出层的Feature Map则包含更多的内容信息。这个特征的原理可以通过我们在[残差网络](#)中介绍的数据处理不等式（DPI）解释：越接近输入层的Feature Map经过的处理（卷积和池化）越少，则这时候损失的图像信息还不会很多。随着网络层数的加深，图像经过的处理也会增多，根据DPI中每次处理信息会减少的原理，靠后的Feature Map则包含的输入图像的信息是不会多余其之前的Feature Map的；同理当我们使用标签值进行参数更新时，越接近损失层的Feature Map则会包含越多的图像标签（内容）信息，越远则包含越少的内容信息。这篇论文正是利用了CNN的天然特征实现的图像风格迁移的。

具体的讲，当我们要在图片 p （content）的内容之上应用图片 a （style）的风格时，我们会使用梯度下降等算法更新目标图像 x （target）的内容，使其在较浅的层有和图片 a 类似的响应值，同时在较深的层和 p 也有类似的响应，这样就保证了 x 和 a 有类似的风格而且和 p 有类似的内容，这样生成的图片 x 就是我们要得到的风格迁移的图片。如图1所示。

在[Keras](#)官方源码中，作者提供了神经风格迁移的[源码](#)，这里对算法的讲解将结合源码进行分析。

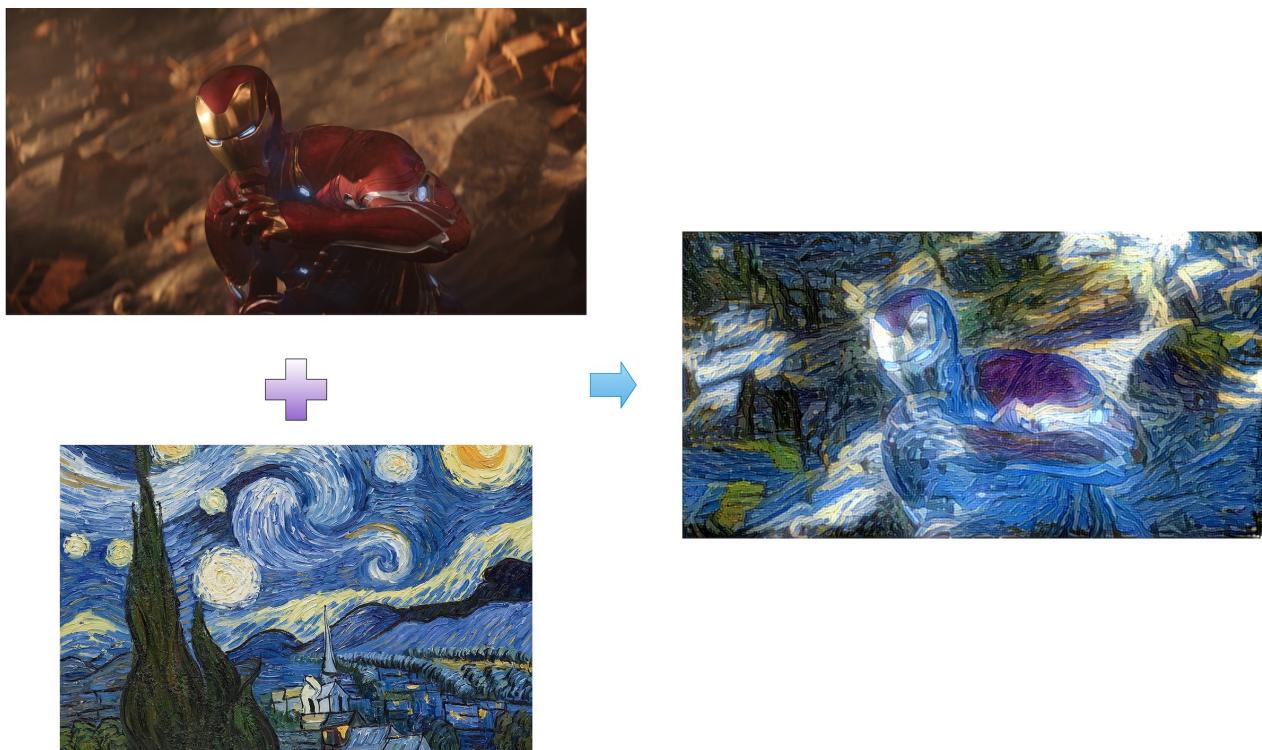


图1：图像风格迁移效果图

1. Image Style Transfer (IST) 算法详解

1.1 算法概览

IST的原理基于上面提到的网络的不同层会响应不同的类型特征的特点实现的。给定一个训练好的网络，源码中使用的是[VGG19](#)，下面是源码第142-143行，因此在运行该源码时如果你之前没有下载过训练好的VGG19模型文件，第一次运行会有下载该文件的过程，文件名为'vgg19_weights_tf_dim_ordering_tf_kernels_notop.h5'。

```
142 model = vgg19.VGG19(input_tensor=input_tensor,
143                      weights='imagenet', include_top=False)
```

论文中有两点在源码中并没有体现，一个是对权值进行了归一化，使用的方法是我们之前介绍的[Weight Normalization\[8\]](#)，另外一个是使用平均池化代替最大池化，使用了这两点的话会有更快的收敛速度。

图2有三个部分，最左侧的输入是风格图片 a ，将其输入到训练好的VGG19中，会得到一批它对应的Feature Map；最右侧则是内容图片 p ，它也会输入到这个网络中得到它对应的Feature Map；中间是目标图片 x ，它的初始值是白噪音图片，它的值会通过SGD进行更新，SGD的损

失函数时通过 x 在这个网络中得到的 Feature Map 和 a 的 Feature Map 以及 p 的 Feature Map 计算得到的。图2中所有的细节会在后面的章节中进行介绍。

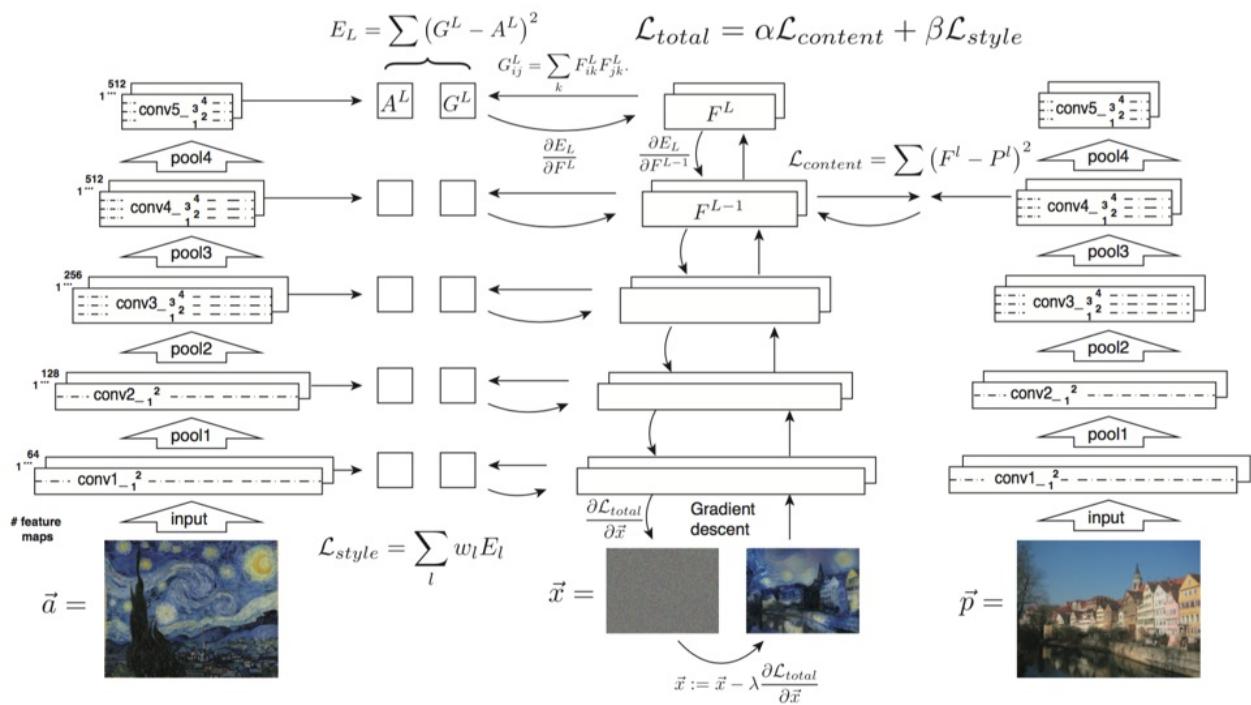


图2：图像风格迁移算法流程图

传统的深度学习方法是根据输入数据更新网络的权值。而IST的算法是固定网络的参数，更新输入的数据。固定权值更新数据还有几个经典案例，例如材质学习[1]，卷积核可视化等。

1.2 内容表示

内容表示是图2中右侧的两个分支所示的过程。我们先看最右侧， p 输入 VGG19 中，我们提取其在第四个 block 中第二层的 Feature Map，表示为 conv4_2（源码中提取的是 conv5_2）。假设其层数为 l ， N_l 是 Feature Map 的数量，也就是通道数， M_l 是 Feature Map 的像素点的个数。那么我们得到 Feature Map F^l 可以表示为 $F^l \in \mathcal{R}^{N_l \times M_l}$ ， F_{ij}^l 则是第 l 层的第 i 个 Feature Map 在位置 j 处的像素点的值。根据同样的定义，我们可以得到 x 在 conv4_2 处的 Feature Map P^l 。

如果 x 的 F_l 和 p 的 P^l 非常接近，那么我们可以认为 x 和 p 在内容上比较接近，因为越接近输出的层包含有越多的内容信息。这里我们可以定义 IST 的内容损失函数为：

$$\mathcal{L}_{\text{content}}(p, x, l) = \frac{1}{2} \sum_{i,j} (F_{i,j}^l - P_{i,j}^l)^2$$

下面我们来看一下源码，上面142行的 `input_tensor` 的是由 p, a, x 一次拼接而成的，见136-138行。

```
136 input_tensor = K.concatenate([base_image,
137                               style_reference_image,
138                               combination_image], axis=0)
```

通过对142行的 `model` 的遍历我们可以得到每一层的Feature Map的名字以及内容，然后将其保存在字典中，见147行。

```
147 outputs_dict = dict([(layer.name, layer.output) for layer in model.layers])
```

这样我们可以根据关键字提取我们想要的Feature Map，例如我们提取两个图像在 `conv5_2` 处的Feature Map P^l （源码中的 `base_image_features`）和 F^l 源码中的 `combination_features`，然后使用这两个Feature Map计算损失值，见208-212行：

```
208 layer_features = outputs_dict['block5_conv2']
209 base_image_features = layer_features[0, :, :, :]
210 combination_features = layer_features[2, :, :, :]
211 loss += content_weight * content_loss(base_image_features,
212                                         combination_features)
```

上式中的 `content_weight` 是内容损失函数的比重，源码中给出的值是0.025，内容损失函数的定义见185-186行：

```
185 def content_loss(base, combination):
186     return K.sum(K.square(combination - base))
```

有了损失函数的定义之后，我们便可以根据损失函数的值计算其关于 $F_{i,j}$ 的梯度值，从而实现从后向前的梯度更新。

$$\frac{\partial \mathcal{L}_{\text{content}}}{\partial F_{i,j}^l} = \begin{cases} (F^l - P^l)_{i,j} & \text{if } F_{i,j} > 0 \\ 0 & \text{if } F_{i,j} < 0 \end{cases}$$

如果损失函数只包含内容损失，当模型收敛时，我们得到的 x' 应该非常接近 p 的内容。但是它很难还原到和 p 一模一样，因为即使损失值为0时，我们得到的 x' 值也有多种形式。

为什么说 x' 具有 p 的内容呢，因为当 x' 经过 VGG19 的处理后，它的 conv5_2 层的输出了 p 几乎一样，而较深的层具有较高的内容信息，这也就说明了 x' 和 p 具有非常类似的内容信息。

1.3 风格表示

风格表示的计算过程是图2的左侧和中间两个分支。和计算 F^l 相同，我们将 a 输入到模型中便可得到它对应的 Feature Map S^l 。不同于内容表示的直接运算，风格表示使用的是 Feature Map 展开成 1 维向量的 Gram 矩阵的形式。使用 Gram 矩阵的原因是因为考虑到纹理特征是和图像的具体位置没有关系的，所以通过打乱纹理的位置信息来保证这个特征，Gram 矩阵的定义如下：

$$G_{i,j}^l = \sum_k F_{i,k}^l F_{j,k}^l$$

另外一点和内容表示不同的是，风格表示使用了每个 block 的第一个卷积来计算损失函数，作者认为这种方式得到的纹理特征更为光滑，因为仅仅使用底层 Feature Map 得到的图像较为精细但是比较粗糙，而高层得到的图像则含有更多的内容信息，损失了一些纹理信息，但他的材质更为光滑。所以，综合了所有层的样式表示的损失函数为：

$$L_{style} = \sum_l w_l E_l$$

其中 E_l 是 S^l 的 Gram 矩阵 A^l 和 F^l 的 Gram 矩阵 G^l 的均方误差：

$$E_l = \frac{1}{4N_l^2 M_l^2} \sum_{i,j} (G_{i,j}^l - A_{i,j}^l)^2$$

它关于 $F_{i,j}^l$ 的梯度的计算方式为：

$$\frac{\partial E_l}{\partial F_{i,j}^l} = \begin{cases} \frac{1}{N_l^2 M_l^2} ((F^l)^T (G^l - A^l))_{ji} & \text{if } F_{i,j}^l > 0 \\ 0 & \text{if } F_{i,j}^l < 0 \end{cases}$$

上面的更新同样使用 SGD。

下面我们继续来学习源码，从源码的 214-223 行我们可以看出样式表示使用了 5 个 block 的 Feature Map：

```

214 feature_layers = ['block1_conv1', 'block2_conv1',
215                 'block3_conv1', 'block4_conv1',
216                 'block5_conv1']
217 for layer_name in feature_layers:
218     layer_features = outputs_dict[layer_name]
219     style_reference_features = layer_features[1, :, :, :]
220     combination_features = layer_features[2, :, :, :]
221     sl = style_loss(style_reference_features, combination_features)
222     loss += (style_weight / len(feature_layers)) * sl
223 loss += total_variation_weight * total_variation_loss(combination_image)

```

从上面的代码中我们可以看出，样式表示使用了 `feature_layers` 中所包含的Feature Map，并且最后 `loss` 的计算把它们进行了相加。第221行的 `style_loss` 的定义见源码的171-178行：

```

171 def style_loss(style, combination):
172     assert K.ndim(style) == 3
173     assert K.ndim(combination) == 3
174     S = gram_matrix(style)
175     C = gram_matrix(combination)
176     channels = 3
177     size = img_nrows * img_ncols
178     return K.sum(K.square(S - C)) / (4.0 * (channels ** 2) * (size ** 2))

```

从174-175行我们可以看出损失函数的计算使用的是两个Feature Map的Gram矩阵，Gram矩阵的定义见155-162行：

```

155 def gram_matrix(x):
156     assert K.ndim(x) == 3
157     if K.image_data_format() == 'channels_first':
158         features = K.batch_flatten(x)
159     else:
160         features = K.batch_flatten(K.permute_dimensions(x, (2, 0, 1)))
161     gram = K.dot(features, K.transpose(features))
162     return gram

```

第158或者160行的 `batch_flatten` 验证了Feature Map要先展开成向量，第161行则是Gram矩阵的计算公式。

还有一些超参数在配置文件中进行了指定，`style_weight` 和 `total_variation_weight` 的默认值都是1。

1.4 风格迁移

明白了如何计算内容损失函数 $\mathcal{L}_{content}$ 和风格损失函数 \mathcal{L}_{style} 之后，整个风格迁移任务的损失函数就是两个损失值得加权和：

$$\mathcal{L}_{\text{total}}(p, a, x) = \alpha \mathcal{L}_{\text{content}}(p, x) + \beta \mathcal{L}_{\text{style}}(a, x)$$

其中 α 和 β 就是我们在 1.2 节和 1.3 节介绍的 `content_weight` 和 `total_variation_weight`。通过调整这两个超参数的值我们可以设置生成的图像更偏向于 p 的内容还是 a 的风格。 $\frac{\partial \mathcal{L}_{\text{total}}}{\partial x}$ 的值用来更新输入图像 x 的内容，作者推荐使用 L-BFGS 更新梯度。

另外对于 x 的初始化，论文中推荐使用白噪音进行初始化，这样虽然计算的时间要更长一些，但是得到的图像的样式具有更强的随机性。而论文使用的是使用 p 初始化 x ，这样得到的生成图像更加稳定。

下面继续学习这一部分的源码。在第 287-288 行的 `fmin_l_bfgs_b` 说明了计算梯度使用了 L-BFGS 算法，它是 `scipy` 提供：

```
287 x, min_val, info = fmin_l_bfgs_b(evaluator.loss, x.flatten(),
288                               fprime=evaluator.grads, maxfun=20)
```

`fmin_l_bfgs_b` 是 `scipy` 包中一个函数。第一个参数是定义的损失函数，第二个参数是输入数据，`fprime` 通常用于计算第一个损失函数的梯度，`maxfun` 是函数执行的次数。它的第一个返回值是更新之后的 x 的值，这里使用了递归的方式反复更新 x ，第二个返回值是损失值。

其中 `x` 的初始化使用的是内容图片 p ：

```
282 x = preprocess_image(base_image_path)
```

287 行的损失函数定义在 264-269 行：

```
264 def loss(self, x):
265     assert self.loss_value is None
266     loss_value, grad_values = eval_loss_and_grads(x)
267     self.loss_value = loss_value
268     self.grad_values = grad_values
269     return self.loss_value
```

其中最重要的函数是 `eval_loss_and_grads()` 函数，它定义在了 237-248 行：

```

237 def eval_loss_and_grads(x):
238     if K.image_data_format() == 'channels_first':
239         x = x.reshape((1, 3, img_nrows, img_ncols))
240     else:
241         x = x.reshape((1, img_nrows, img_ncols, 3))
242     outs = f_outputs([x])
243     loss_value = outs[0]
244     if len(outs[1:]) == 1:
245         grad_values = outs[1].flatten().astype('float64')
246     else:
247         grad_values = np.array(outs[1:]).flatten().astype('float64')
248     return loss_value, grad_values

```

其中 `f_outputs()` 是实例化的Keras函数，作用是使用梯度更新 x 的内容，见226-234行：

```

226 grads = K.gradients(loss, combination_image)
227
228 outputs = [loss]
229 if isinstance(grads, (list, tuple)):
230     outputs += grads
231 else:
232     outputs.append(grads)
233
234 f_outputs = K.function([combination_image], outputs)

```

2. 总结

图像风格迁移是一个非常好玩但是无法对齐效果量化的算法，我们可以得到和一些著名画家风格看起来非常类似的画作，但是很难从数学的角度去衡量一个画作的风格，得出的结论是非常主观的。但是算法的设计动机是出于CNN的底层**Feature Map**接近图像纹理而高层**Feature Map**接近图像内容的天然特性，也是对神经网络这个黑盒子从另外一个角度给与了解释。IST产生的结果非常有趣，由此诞生了一批商用的软件，例如Prisma等。

IST如果能迁移到音频领域也许会有帮助，例如在TTS中如果可以将合成的语音的内容应用到真实人类语音的风格上，这样也许可以得到更为平滑的语音。或者如果我们将音频内容应用到某个人说话的风格中，也许我们可以得到和这个人说话风格非常类似的音频输出。

算法另外一个缺点是对噪音比较敏感，尤其是当参与合成的风格图片和内容图片都是真实照片的时候。

Reference

1	Gatys, Leon <i>et al.</i> , Texture synthesis using convolutional neural networks, <i>Advances in neural information processing systems</i> , 2015.
2	Gatys, Leon A <i>et al.</i> , A neural algorithm of artistic style, 2015.
3	Luo, Ping and Ren, Jiamin and Peng, Zhanglin, Differentiable learning-to-normalize via switchable normalization, 2018.
4	Wu, Yuxin and He, Kaiming, Group normalization, <i>Proceedings of the European Conference on Computer Vision (ECCV)</i> , 2018.
5	Ulyanov, Dmitry <i>et al.</i> , Texture Networks: Feed-forward Synthesis of Textures and Stylized Images., <i>ICML</i> , 2016.
6	Ulyanov, Dmitry and Vedaldi, Andrea and Lempitsky, Victor, Instance normalization: The missing ingredient for fast stylization, 2016.
7	Gatys, Leon A <i>et al.</i> , Image style transfer using convolutional neural networks, <i>Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition</i> , 2016.
8	Salimans, Tim and Kingma, Durk P, Weight normalization: A simple reparameterization to accelerate training of deep neural networks, <i>Advances in Neural Information Processing Systems</i> , 2016.
9	Ba, Jimmy Lei and Kiros, Jamie Ryan and Hinton, Geoffrey E, Layer normalization, 2016.
10	Li, Hao <i>et al.</i> , Visualizing the loss landscape of neural nets, <i>Advances in Neural Information Processing Systems</i> , 2018.
11	Santurkar, Shibani <i>et al.</i> , How does batch normalization help optimization?, <i>Advances in Neural Information Processing Systems</i> , 2018.
12	Yu, Jiahui <i>et al.</i> , Unitbox: An advanced object detection network, <i>Proceedings of the 24th ACM international conference on Multimedia</i> , 2016.
13	Huang, Lichao <i>et al.</i> , Densebox: Unifying landmark localization with end to end object detection, 2015.
14	Ronneberger, Olaf and Fischer, Philipp and Brox, Thomas, U-net: Convolutional networks for biomedical image segmentation, <i>International Conference on Medical image computing and computer-assisted intervention</i> , 2015.
15	Lai, Siwei <i>et al.</i> , Recurrent convolutional neural networks for text classification, <i>Twenty-ninth AAAI conference on artificial intelligence</i> , 2015.
16	Liang, Ming and Hu, Xiaolin, Recurrent convolutional neural network for object recognition, <i>Proceedings of the IEEE conference on computer vision and pattern recognition</i> , 2015.

17	end trainable scene text localization and recognition framework, <i>Proceedings of the IEEE International Conference on Computer Vision</i> , 2017.
18	Doll{\`a}r, Piotr <i>et al.</i> , Fast feature pyramids for object detection, <i>IEEE</i> , 2014.
19	Jaderberg, Max <i>et al.</i> , Reading text in the wild with convolutional neural networks, <i>Springer</i> , 2016.
20	Warp, Fred L Bookstein Principal, Thin-Plate Splines and the Decompositions of Deformations, 1989.
21	Bookstein, Fred L., Principal warps: Thin-plate splines and the decomposition of deformations, <i>IEEE</i> , 1989.
22	Shi, Baoguang <i>et al.</i> , Robust scene text recognition with automatic rectification, <i>Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition</i> , 2016.
23	Goodfellow, Ian <i>et al.</i> , Generative adversarial nets, <i>Advances in neural information processing systems</i> , 2014.
24	Zhou, Xinyu <i>et al.</i> , EAST: an efficient and accurate scene text detector, <i>Proceedings of the IEEE conference on Computer Vision and Pattern Recognition</i> , 2017.
25	Ma, Jianqi <i>et al.</i> , Arbitrary-oriented scene text detection via rotation proposals, <i>IEEE</i> , 2018.
26	Kang, Le and Li, Yi and Doermann, David, Orientation robust text line detection in natural images, <i>Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition</i> , 2014.
27	Epshtain, Boris and Ofek, Eyal and Wexler, Yonatan, Detecting text in natural scenes with stroke width transform, <i>2010 IEEE Computer Society Conference on Computer Vision and Pattern Recognition</i> , 2010.
28	Xie, Saining and Tu, Zhuowen, Holistically-nested edge detection, <i>Proceedings of the IEEE international conference on computer vision</i> , 2015.
29	Yao, Cong <i>et al.</i> , Scene text detection via holistic, multi-channel prediction, 2016.
30	Wolf, Christian and Jolion, Jean-Michel, Object count area graphs for the evaluation of object detection and segmentation algorithms, <i>Springer</i> , 2006.
31	Tian, Zhi <i>et al.</i> , Detecting text in natural image with connectionist text proposal network, <i>European conference on computer vision</i> , 2016.
32	Zhong, Zhuoyao <i>et al.</i> , Deeptext: A unified framework for text proposal generation and text detection in natural images, 2016.
33	Singh, Bharat and Davis, Larry S, An analysis of scale invariance in object detection snip, <i>Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition</i> , 2018.
34	Singh, Bharat and Najibi, Mahyar and Davis, Larry S, SNIPER: Efficient multi-scale training, <i>Advances in Neural Information Processing Systems</i> , 2018.
	Lin, Tsung-Yi <i>et al.</i> , Focal loss for dense object detection, <i>Proceedings of the</i>

	<i>IEEE international conference on computer vision</i> , 2017.
36	Miller, George A <i>et al.</i> , Introduction to WordNet: An on-line lexical database, <i>Oxford University Press</i> , 1990.
37	Holschneider, Matthias <i>et al.</i> , A real-time algorithm for signal analysis with the help of the wavelet transform, <i>Wavelets</i> , Springer, 1990.
38	Pinheiro, Pedro O <i>et al.</i> , Learning to refine object segments, <i>European Conference on Computer Vision</i> , 2016.
39	Chen, Liang-Chieh <i>et al.</i> , Semantic image segmentation with deep convolutional nets and fully connected crfs, 2014.
40	Liu, Wei <i>et al.</i> , Parsenet: Looking wider to see better, 2015.
41	Jaderberg, Max <i>et al.</i> , Spatial transformer networks, <i>Advances in neural information processing systems</i> , 2015.
42	Long, Jonathan and Shelhamer, Evan and Darrell, Trevor, Fully convolutional networks for semantic segmentation, <i>Proceedings of the IEEE conference on computer vision and pattern recognition</i> , 2015.
43	Zitnick, C Lawrence and Doll{\'a}r, Piotr, Edge boxes: Locating object proposals from edges, <i>European conference on computer vision</i> , 2014.
44	Lenc, Karel and Vedaldi, Andrea, R-cnn minus r, 2015.
45	Lazebnik, Svetlana and Schmid, Cordelia and Ponce, Jean, Beyond bags of features: Spatial pyramid matching for recognizing natural scene categories, <i>2006 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'06)</i> , 2006.
46	Grauman, K and Darrell, T, The pyramid match kernel: Discriminative classification with sets of image features, 2005, 2006.
47	Felzenszwalb, Pedro F <i>et al.</i> , Object detection with discriminatively trained part-based models, <i>IEEE</i> , 2010.
48	Felzenszwalb, Pedro F and Huttenlocher, Daniel P, Efficient graph-based image segmentation, <i>Springer</i> , 2004.
49	Uijlings, Jasper RR <i>et al.</i> , Selective search for object recognition, <i>Springer</i> , 2013.
50	Gu, Chunhui, Recognition using regions, 2012.
51	Hu, Ronghang <i>et al.</i> , Learning to segment every thing, <i>Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition</i> , 2018.
52	Redmon, Joseph and Farhadi, Ali, Yolov3: An incremental improvement, 2018.
53	Fu, Cheng-Yang <i>et al.</i> , Dssd: Deconvolutional single shot detector, 2017.
54	Liu, Wei <i>et al.</i> , Ssd: Single shot multibox detector, <i>European conference on computer vision</i> , 2016.
55	Redmon, Joseph and Farhadi, Ali, YOLO9000: better, faster, stronger, <i>Proceedings of the IEEE conference on computer vision and pattern recognition</i> , 2017.

56	Redmon, Joseph <i>et al.</i> , You only look once: Unified, real-time object detection, <i>Proceedings of the IEEE conference on computer vision and pattern recognition</i> , 2016.
57	He, Kaiming <i>et al.</i> , Mask r-cnn, <i>Proceedings of the IEEE international conference on computer vision</i> , 2017.
58	Lin, Tsung-Yi <i>et al.</i> , Feature pyramid networks for object detection, <i>Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition</i> , 2017.
59	Dai, Jifeng <i>et al.</i> , R-fcn: Object detection via region-based fully convolutional networks, <i>Advances in neural information processing systems</i> , 2016.
60	Ren, Shaoqing <i>et al.</i> , Faster r-cnn: Towards real-time object detection with region proposal networks, <i>Advances in neural information processing systems</i> , 2015.
61	Girshick, Ross, Fast r-cnn, <i>Proceedings of the IEEE international conference on computer vision</i> , 2015.
62	He, Kaiming <i>et al.</i> , Spatial pyramid pooling in deep convolutional networks for visual recognition, <i>IEEE</i> , 2015.
63	Girshick, Ross <i>et al.</i> , Rich feature hierarchies for accurate object detection and semantic segmentation, <i>Proceedings of the IEEE conference on computer vision and pattern recognition</i> , 2014.
64	Graves, Alex, Sequence transduction with recurrent neural networks, 2012.
65	Schuster, Mike and Paliwal, Kuldip K, Bidirectional recurrent neural networks, <i>IEEE</i> , 1997.
66	Graves, Alex and Mohamed, Abdel-rahman and Hinton, Geoffrey, Speech recognition with deep recurrent neural networks, <i>2013 IEEE international conference on acoustics, speech and signal processing</i> , 2013.
67	Wu, Yonghui <i>et al.</i> , Google's neural machine translation system: Bridging the gap between human and machine translation, 2016.
68	Peters, Matthew E <i>et al.</i> , Semi-supervised sequence tagging with bidirectional language models, 2017.
69	Radford, Alec <i>et al.</i> , Improving language understanding with unsupervised learning, 2018.
70	Taylor, Wilson L, "Cloze procedure": A new tool for measuring readability, <i>SAGE Publications Sage CA: Los Angeles, CA</i> , 1953.
71	Devlin, Jacob <i>et al.</i> , Bert: Pre-training of deep bidirectional transformers for language understanding, 2018.
72	Vaswani, Ashish <i>et al.</i> , Attention is all you need, <i>Advances in Neural Information Processing Systems</i> , 2017.
73	Jozefowicz, Rafal and Zaremba, Wojciech and Sutskever, Ilya, An empirical exploration of recurrent network architectures, <i>International Conference on Machine Learning</i> , 2015.
	Gers, Felix A and Schmidhuber, J{"u}rgen and Cummins, Fred, Learning to

75	Sussillo, David, RANDOM WALKS: TRAINING VERY DEEP NONLIN-EAR FEED-FORWARD NETWORKS WITH SMARTINI, 2014.
76	Hochreiter, Sepp and Schmidhuber, J{"u}rgen, Long short-term memory, <i>MIT Press</i> , 1997.
77	Graves, Alex <i>et al.</i> , Connectionist temporal classification: labelling unsegmented sequence data with recurrent neural networks, <i>Proceedings of the 23rd international conference on Machine learning</i> , 2006.
78	Weston, Jason and Chopra, Sumit and Bordes, Antoine, Memory networks, 2014.
79	Yang, Zichao <i>et al.</i> , Hierarchical attention networks for document classification, <i>Proceedings of the 2016 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies</i> , 2016.
80	Sutskever, Ilya and Vinyals, Oriol and Le, Quoc V, Sequence to sequence learning with neural networks, <i>Advances in neural information processing systems</i> , 2014.
81	Cho, Kyunghyun <i>et al.</i> , Learning phrase representations using RNN encoder-decoder for statistical machine translation, 2014.
82	Bengio, Yoshua <i>et al.</i> , A neural probabilistic language model, 2003.
83	Mikolov, Tom{"a}{\w{s}} <i>et al.</i> , Recurrent neural network based language model, <i>Eleventh annual conference of the international speech communication association</i> , 2010.
84	Real, Esteban <i>et al.</i> , Regularized evolution for image classifier architecture search, 2018.
85	Clevert, Djork-Arn{"e} and Unterthiner, Thomas and Hochreiter, Sepp, Fast and accurate deep network learning by exponential linear units (elus), 2015.
86	Larsson, Gustav and Maire, Michael and Shakhnarovich, Gregory, Fractalnet: Ultra-deep neural networks without residuals, 2016.
87	Schulman, John <i>et al.</i> , Proximal policy optimization algorithms, 2017.
88	Bahdanau, Dzmitry and Cho, Kyunghyun and Bengio, Yoshua, Neural machine translation by jointly learning to align and translate, 2014.
89	Williams, Ronald J, Simple statistical gradient-following algorithms for connectionist reinforcement learning, <i>Springer</i> , 1992.
90	Loshchilov, Ilya and Hutter, Frank, Sgdr: Stochastic gradient descent with warm restarts, 2016.
91	Sifre, Laurent and Mallat, St{"e}phane, Rigid-motion scattering for image classification, <i>Citeseer</i> , 2014.
92	Guo, Yiwen and Yao, Anbang and Chen, Yurong, Dynamic network surgery for efficient dnns, <i>Advances In Neural Information Processing Systems</i> , 2016.
	Abdi, Masoud and Nahavandi, Saeid, Multi-residual networks: Improving the

	speed and accuracy of residual networks, 2016.
94	Veit, Andreas and Wilber, Michael J and Belongie, Serge, Residual networks behave like ensembles of relatively shallow networks, <i>Advances in neural information processing systems</i> , 2016.
95	He, Kaiming <i>et al.</i> , Identity mappings in deep residual networks, <i>European conference on computer vision</i> , 2016.
96	Goodfellow, Ian J <i>et al.</i> , Maxout networks, 2013.
97	Sermanet, Pierre <i>et al.</i> , Overfeat: Integrated recognition, localization and detection using convolutional networks, 2013.
98	Lin, Min and Chen, Qiang and Yan, Shuicheng, Network in network, 2013.
99	Ciresan, Dan Claudiu <i>et al.</i> , Flexible, high performance convolutional neural networks for image classification, <i>Twenty-Second International Joint Conference on Artificial Intelligence</i> , 2011.
100	Hinton, Geoffrey E <i>et al.</i> , Improving neural networks by preventing co-adaptation of feature detectors, 2012.
101	LeCun, Yann <i>et al.</i> , Gradient-based learning applied to document recognition, <i>Taipei, Taiwan</i> , 1998.
102	Huang, Gao <i>et al.</i> , Condensenet: An efficient densenet using learned group convolutions, <i>Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition</i> , 2018.
103	Liu, Chenxi <i>et al.</i> , Progressive neural architecture search, <i>Proceedings of the European Conference on Computer Vision (ECCV)</i> , 2018.
104	Zoph, Barret <i>et al.</i> , Learning transferable architectures for scalable image recognition, <i>Proceedings of the IEEE conference on computer vision and pattern recognition</i> , 2018.
105	Zoph, Barret and Le, Quoc V, Neural architecture search with reinforcement learning, 2016.
106	Ma, Ningning <i>et al.</i> , Shufflenet v2: Practical guidelines for efficient cnn architecture design, <i>Proceedings of the European Conference on Computer Vision (ECCV)</i> , 2018.
107	Zhang, Xiangyu <i>et al.</i> , Shufflenet: An extremely efficient convolutional neural network for mobile devices, <i>Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition</i> , 2018.
108	Xie, Saining <i>et al.</i> , Aggregated residual transformations for deep neural networks, <i>Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition</i> , 2017.
109	Howard, Andrew G <i>et al.</i> , Mobilenets: Efficient convolutional neural networks for mobile vision applications, 2017.
110	I <i>et al.</i> , SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and< 0.5 MB model size, 2016.
	Zhang, Xingcheng <i>et al.</i> , Polynet: A pursuit of structural diversity in very deep

111	Zhang, Xingcheng <i>et al.</i> , Polynet: A pursuit of structural diversity in very deep networks, <i>Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition</i> , 2017.
112	Szegedy, Christian <i>et al.</i> , Inception-v4, inception-resnet and the impact of residual connections on learning, <i>Thirty-First AAAI Conference on Artificial Intelligence</i> , 2017.
113	Szegedy, Christian <i>et al.</i> , Rethinking the inception architecture for computer vision, <i>Proceedings of the IEEE conference on computer vision and pattern recognition</i> , 2016.
114	Ioffe, Sergey and Szegedy, Christian, Batch normalization: Accelerating deep network training by reducing internal covariate shift, 2015.
115	S <i>et al.</i> , Mobilenetv2: Inverted residuals and linear bottlenecks, <i>Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition</i> , 2018.
116	Hu, Jie and Shen, Li and Sun, Gang, Squeeze-and-excitation networks, <i>Proceedings of the IEEE conference on computer vision and pattern recognition</i> , 2018.
117	Huang, Gao <i>et al.</i> , Densely connected convolutional networks, <i>Proceedings of the IEEE conference on computer vision and pattern recognition</i> , 2017.
118	He, Kaiming <i>et al.</i> , Deep residual learning for image recognition, <i>Proceedings of the IEEE conference on computer vision and pattern recognition</i> , 2016.
119	Chollet, Fran{\c{c}ois, Xception: Deep learning with depthwise separable convolutions, <i>Proceedings of the IEEE conference on computer vision and pattern recognition</i> , 2017.
120	Simonyan, Karen and Zisserman, Andrew, Very deep convolutional networks for large-scale image recognition, 2014.
121	Szegedy, Christian <i>et al.</i> , Going deeper with convolutions, <i>Proceedings of the IEEE conference on computer vision and pattern recognition</i> , 2015.
122	Zeiler, Matthew D and Fergus, Rob, Visualizing and understanding convolutional networks, <i>European conference on computer vision</i> , 2014.
123	Alex Krizhevsky and Sutskever, Ilya and Hinton, Geoffrey E, ImageNet Classification with Deep Convolutional Neural Networks , <i>Advances in Neural Information Processing Systems 25</i> , Curran Associates, Inc., 2012.