

# Papers I have read

宋明辉

July 17, 2018



# Contents

<b>1</b>	<b>Image processing based on CUDA</b>	<b>3</b>
1.1	Novel multi-scale retinex with color restoration on graphics processing unit	3
1.1.1	Abstract	3
1.1.2	Content	3
1.1.3	Parallel optimization strage	3
1.1.4	Conclusion	4
1.2	Image Convolution	4
1.2.1	Naïve Implementation	4
1.2.2	Naïve Shared Memory Implementation	6
1.2.3	Separable Gaussian Filtering	8
1.2.4	Optimizing for memory coalescence	9
1.3	CUDA C Best Practice Guide	10
1.3.1	Performance Metrics	10
1.4	Npp Library Image Filters	10
1.4.1	Image Data	10
1.5	PTX ISA 6.0	10
1.5.1	PTX Machine Model	10
<b>2</b>	<b>FPGAs</b>	<b>13</b>
2.1	Performance Comparison of FPGA, GPU and CPU in Image Processing 2009	13
2.1.1	Abstract	13
2.1.2	Content	14
2.1.3	Results	15
2.1.4	Conclusion	15
2.2	Fast FPGA Prototyping for real-time image processing with very high-level synthesis 2017	16
2.2.1	Abstract	16
2.2.2	Content	17
<b>3</b>	<b>Image Fusion</b>	<b>19</b>
3.1	Guided Image Filter 2013	19
3.1.1	Abstract	19
3.1.2	Content	19
3.1.3	Conclusion	20
3.2	Multiscale Image Fusion Through Guided Filtering	20

3.2.1	Abstract . . . . .	20
3.2.2	Contents . . . . .	20
3.2.3	Conclusion . . . . .	22
3.3	Image Fusion With Guided Filtering . . . . .	23
3.3.1	Abstract . . . . .	23
3.3.2	Contents . . . . .	23
3.3.3	Fusion Frame . . . . .	24
3.3.4	Conclusion . . . . .	24
<b>4</b>	<b>Saliency Detection</b>	<b>25</b>
4.1	Frequency-tuned Salient Region Detection . . . . .	25
4.1.1	Abstract . . . . .	25
4.1.2	Contents . . . . .	25
4.1.3	Conclusion . . . . .	26
<b>5</b>	<b>Semantic SLAM</b>	<b>27</b>
5.1	DeLS-3D: Deep Localization and Segmentation with a 2D Semantic Map[36] . . . . .	27
5.1.1	Abstract . . . . .	27
5.1.2	Introduction . . . . .	27
5.1.3	Framework . . . . .	28
5.1.4	Related Work . . . . .	28
5.1.5	Dataset . . . . .	29
5.1.6	Localizing camera and Scene Parsing . . . . .	29
5.1.7	Experiment . . . . .	30
5.1.8	Conclusion . . . . .	31
5.2	PAD-Net: Multi-Task Guided Prediction-and-Distillation Network for Simultaneous Depth and Scene Parsing [39] . . . . .	31
5.2.1	Abstract . . . . .	31
5.2.2	Analysis . . . . .	31
5.3	RNN for Learning Dense Depth and Ego-Motion from Video . . . . .	32
5.3.1	Abstract . . . . .	32
5.3.2	Introduction & Related Works . . . . .	32
5.3.3	Network Architecture . . . . .	33
5.3.4	Training . . . . .	34
5.3.5	Experiments . . . . .	35
5.3.6	Ablation Studies . . . . .	35
5.4	DA-RNN . . . . .	35
5.4.1	Related Works . . . . .	36
5.4.2	Methods . . . . .	36
5.4.3	Experiments . . . . .	36
5.4.4	Conclusion . . . . .	36
5.5	SemanticFusion: Dense 3D Semantic Mapping with CNNs . . . . .	36
5.5.1	Introduction & Related Works . . . . .	36
5.5.2	Method . . . . .	37
5.5.3	Experiments . . . . .	38
5.5.4	总结 . . . . .	38

## CONTENTS

---

5.6	Meaningful Maps with Object-Oriented Semantic Mapping . . . . .	39
5.6.1	Introduction & Related Works . . . . .	39
5.6.2	Object Oriented Semantic Mapping . . . . .	39
5.6.3	总结 . . . . .	42
5.7	LiteFlowNet . . . . .	42
5.7.1	背景知识 . . . . .	42
5.7.2	Related Works . . . . .	43
5.7.3	LiteFlowNet . . . . .	44
5.7.4	Ablation Study . . . . .	47
5.7.5	Regularization . . . . .	48
5.7.6	Conclusion . . . . .	48
5.8	小结 . . . . .	48
5.9	ExFuse: Enhancing Feature Fusion for Semantic Segmentation . . . . .	48
5.9.1	要解决的问题 . . . . .	48
5.9.2	Method . . . . .	48
5.10	Multi-View Deep Learning for Consistent Semantic Mapping with RGB-D Cameras . . . . .	50
5.10.1	Introduction & Related Works . . . . .	50
5.10.2	CNN Architecture For Semantic Segmentation . . . . .	51
5.10.3	Multi-View Consistent Learning and Prediction . . . . .	52
5.10.4	Experiments . . . . .	53
5.10.5	总结 . . . . .	53
5.11	MaskFusion . . . . .	53
5.11.1	背景及相关工作 . . . . .	53
5.11.2	System Design . . . . .	55
5.11.3	Evaluation . . . . .	57
5.11.4	总结 . . . . .	57
5.12	小结 2 . . . . .	58
5.13	CNN-SLAM . . . . .	58
5.14	图像语义分割之 FCN 和 CRF . . . . .	58
5.14.1	前端 FCN . . . . .	59
5.14.2	DeepLab . . . . .	59
5.14.3	后端优化 CRF/MRF . . . . .	61
5.14.4	小结 . . . . .	61
5.15	Learning Deconvolution Network for Semantic . . . . .	62
5.16	ReSeg 2015 . . . . .	63
5.16.1	背景 & 相关工作 . . . . .	63
5.16.2	Model Description . . . . .	63
5.16.3	实验结果 . . . . .	65
5.16.4	总结 . . . . .	65
5.17	U-Net . . . . .	65
5.17.1	Network Architecture . . . . .	65
5.17.2	Conclusion . . . . .	66
5.17.3	补充 . . . . .	66
5.18	Semantic Visual Localization . . . . .	67
5.18.1	背景 & 相关工作 . . . . .	67

---

5.18.2 Semantic Visual Localization . . . . .	68
5.18.3 Experiments . . . . .	69
5.18.4 Conclusions . . . . .	70
5.19 小结 3 . . . . .	70
5.20 StaticFusion . . . . .	70
5.20.1 背景 . . . . .	70
5.20.2 动态场景下 SLAM 系统的相关工作 . . . . .	70
5.20.3 Framework 和 Notation . . . . .	71
5.21 Convolutional CRFs for Semantic Segmentation . . . . .	72
5.21.1 背景 & 相关工作 . . . . .	72
5.21.2 Fully Connected CRFs . . . . .	73
5.21.3 Convolutional CRFs . . . . .	74
5.21.4 总结 . . . . .	75
5.22 Co-Fusion . . . . .	75
5.23 Squeeze-and-Excitation Networks . . . . .	75
5.24 Deep Multi-scale Architectures For Monocular Depth Estimation . . . . .	76
5.24.1 背景与相关工作 . . . . .	76
5.25 Depth Map Prediction from a single image usign a multi-Scale Deep Network . . . . .	77
5.26 Mask R-CNN . . . . .	78
5.26.1 研究现状与相关背景 . . . . .	79
5.26.2 Mask RCNN . . . . .	79
5.26.3 实验 . . . . .	80
5.26.4 总结 . . . . .	80
5.27 Learning to segment every thing . . . . .	80
5.28 Path aggregation network for instance segmentation . . . . .	82
5.29 End-to-End Instance Segmentation with Recurrent Attention . . . . .	82
5.30 Fully Convolutional Instance-aware Semantic Segmentation . . . . .	82
5.30.1 背景与相关现状 . . . . .	82
5.30.2 Our Approach . . . . .	83
5.31 TernausNetV2: Fully Convolutional Network for Instance Segmentation . . . . .	83
5.31.1 背景及相关工作 . . . . .	83
5.31.2 Model . . . . .	83
5.31.3 Training . . . . .	83
5.31.4 常用数据集 . . . . .	84
5.31.5 KITTI . . . . .	84
5.31.6 官网资源介绍 . . . . .	84
5.31.7 详述 . . . . .	85
5.31.8 Cityscape . . . . .	86
5.31.9 TUM . . . . .	86
5.32 实例分割 -图像分割 2018.06.11 博客总结 . . . . .	86
5.32.1 Mask R-CNN 狙击目标实例分割 . . . . .	86
5.33 无需 Proposal 的实例分割论文 . . . . .	87
5.34 Learning Rich Features from RGB-D Images for Object Detection and Segmentation . . . . .	88
5.35 Multimodal Deep Learning for Robust RGB-D Object Recognition . . . . .	88

## CONTENTS

---

5.36	3D Graph Neural Networks for RGBD Semantic Segmentation . . . . .	89
5.37	RefineNet . . . . .	89
5.37.1	引言中的重要几点 . . . . .	89
5.37.2	算法思想 . . . . .	90
5.37.3	实验部分 . . . . .	90
5.37.4	小结 . . . . .	90
5.38	RedNet: Residual Encoder-Decoder Network for indoor RGB-D Semantic Segmentation . . . . .	90
5.39	RDFNet: RGB-D Multi-level Residual Feature fusion for Indoor Semantic Segmentation . . . . .	90
5.40	Progressively Complementarity-aware Fusion Network for RGB-D Salient Object Detection . . . . .	90
5.41	Semantic-Guided Multi-level RGB-D Feature Fusion for Indoor Semantic segmentation . . . . .	90
5.42	Learning Common and Specific Features for RGB-D Semantic Segmentation with Deconvolutional Networks . . . . .	90
<b>6</b>	<b>Open Set Recognition</b>	<b>91</b>
6.1	GAN . . . . .	91
6.1.1	GAN 原理笔记 . . . . .	91
6.2	从头开始 GAN . . . . .	93
6.2.1	定义 . . . . .	94
6.2.2	DCGAN: Deep Convolution GAN . . . . .	94
6.2.3	CGAN: Conditional Generative Adversarial Nets . . . . .	95
6.2.4	InfoGAN . . . . .	95
6.3	Generative Adversarial Nets . . . . .	96
6.4	Towards Open Set Deep Networks . . . . .	96
6.4.1	Introduction & Related Works . . . . .	96
6.5	Probability Models for Open Set Recognition . . . . .	96
6.5.1	背景及相关工作 . . . . .	97
6.6	Meta Recognition . . . . .	97
6.6.1	该死的 Fisher-Tippet Theorem . . . . .	97
6.6.2	Weibull Distribution . . . . .	98
<b>7</b>	<b>MXNet</b>	<b>99</b>
7.1	MXNet System Overview . . . . .	99
7.1.1	MXNet System Architecture . . . . .	99
7.1.2	MXNet System Components . . . . .	100
7.2	Optimizing Memory Consumption in DL . . . . .	102
7.2.1	Computation Graph . . . . .	102
7.2.2	What Can be Optimized? . . . . .	104
7.2.3	Memory Allocation Algorithm . . . . .	105
7.2.4	Static vs. Dynamic Allocation . . . . .	106
7.2.5	Memory Allocation for Parallel Operations . . . . .	106
7.2.6	How Much Can we Save ? . . . . .	108
7.2.7	References . . . . .	108

---

7.3	Deep Learning Programming Style . . . . .	108
7.3.1	Symbolic vs. Imperative Program . . . . .	108
7.3.2	Imperative Programs Tend to be More Flexible . . . . .	108
7.3.3	Symbolic Programs Tend to be More Efficient . . . . .	109
7.3.4	Case Study: Backprop and AutoDiff . . . . .	109
7.3.5	Model Checkpoint . . . . .	110
7.3.6	Big vs. Small Operations . . . . .	110
7.3.7	Mix The Approaches . . . . .	111
7.4	Dependency Engine for Deep Learning . . . . .	112
7.4.1	Problems in Dependency Scheduling . . . . .	112
7.4.2	Implementing the Generic Dependency Engine . . . . .	113
7.4.3	Discussion . . . . .	114
7.5	Designing Efficient Data Loaders for DL . . . . .	114
7.5.1	Design Insight . . . . .	114
7.5.2	Data Format . . . . .	115
7.5.3	Data Loading and Preprocessing . . . . .	116
7.5.4	MXNet IO Python Interface . . . . .	116
7.6	Except Handling in MXNet . . . . .	117
7.7	MXNet-Gluon 创建模型 . . . . .	118
7.7.1	模型构造 . . . . .	118
7.7.2	自定义层 . . . . .	120
7.7.3	实际例子 . . . . .	120
7.8	MXNet 中的 Deconvolution . . . . .	122
7.9	MXNet 读取训练数据 . . . . .	122
7.9.1	使用 mx.io 读取数据 . . . . .	123
7.9.2	常用的系统提供的接口函数 . . . . .	125
7.9.3	使用 mx.image 读取数据 . . . . .	125
7.9.4	使用 Gluon 接口读取数据 . . . . .	126
7.10	Gluon 中的数据结构 . . . . .	127
7.11	MXNet 中的 Confusing 参数 . . . . .	128
7.11.1	mx.symbol.reshape . . . . .	128
7.11.2	mx.symbol.transpose . . . . .	128
7.12	UpSample In MXNet . . . . .	129
7.12.1	总结 . . . . .	131
7.13	MXNet 中设置层的学习率 . . . . .	131
7.14	SoftmaxCrossEntropyLoss 释疑 . . . . .	131
7.15	MXNet 中的 3D CNN . . . . .	132
7.15.1	应该怎么理解 3D CNN . . . . .	132
7.16	向 MXNet 添加新的操作 . . . . .	132
7.16.1	添加新的 Operator . . . . .	132
7.16.2	添加新的 Layer . . . . .	135
7.17	关于 MXNet 里面的 register 修饰符 . . . . .	138
7.17.1	调用顺序 -CSDN 博客 . . . . .	138
7.17.2	Python 中的装饰器之 register . . . . .	138

<b>8 GPU Optimization in DL</b>	<b>139</b>
8.1 In-Place Activated BatchNorm for Memory-Optimized Training of DNNs	139
8.2 Training deep nets with sublinear memory cost	139
8.3 Memory-efficient backpropagation through time	139
8.4 The Reversible Residual Network: Backpropagation Without Storing Activations	139
8.5	139
<b>9 Tips in DL</b>	<b>141</b>
9.1 Enlarge the FOV	141
9.2 Upsampling	141
9.3 Multiscale Ability	142
9.4 Dilated Convolution	142
9.5 Deconvolutional Network	143
9.5.1 Convolutional Spare Coding	143
9.5.2 CNN 可视化	143
9.5.3 Upsampling	144
9.5.4 补充	144
9.5.5 Deconvolution 与 Upsample 的区别	144
9.5.6 Deconvolution 输出的大小计算	146
9.5.7 Conv2DTranspose 用于成倍的提高分辨率的时候	146
9.6 Dilated Network 与 Deconv Network 之间的区别	147
9.7 目标检测中的 mAP 的含义	147
9.8 统计学习方法	148
9.9 Distillation Module	148
9.9.1 Knowledge Distillation	149
9.9.2 Recurrent Knowledge Distillation [30]	149
9.10 光流估计中的 Average end-point error	149
9.11 CNN 中的卷积方式汇总	149
9.11.1 Inception	149
9.11.2 空洞卷积, Dilation	150
9.11.3 深度可分离卷积, Depthwise Separable Convolution	150
9.11.4 可变性卷积	152
9.11.5 特征重标定卷积	154
9.11.6 小结 - 比较	154
9.12 全连接与卷积的异同	156
9.13 Pooling	156
9.14 Local Response Normalization	157
9.15 CNN 中感受野的计算	158
9.16 神经网络中的初始化	162
9.16.1 Xavier	162
9.17 计算图的后向传播计算	163
9.17.1 Notes on CNNs	164
9.17.2 Pooling 层的反向传播	168
9.18 神经网络中的 Attention 机制	168
9.18.1 Recurrent Models of Visual Attention	168

---

9.19 小型网络 . . . . .	170
9.19.1 理论分析 . . . . .	170
9.19.2 GCONV & DWCONV . . . . .	170
9.19.3 NiN 及相关 . . . . .	171
9.20 Deep Reinforcement Learning . . . . .	171
9.21 为什么 Python 中要继承 object 类 . . . . .	171
9.22 1*1 卷积 . . . . .	172
9.23 目标检测中完整流程 . . . . .	173
9.24 Batch Size 的影响 . . . . .	173
9.25 非极大值抑制 NMS . . . . .	173
9.26 Bilinear filler 初始化 ConvTranspose 层的权重 . . . . .	174
9.27 卷积层输出的尺寸 . . . . .	174
9.28 机器学习面试博客总结 2018.06.11 . . . . .	175
9.28.1 大疆 . . . . .	175
9.28.2 机器学习面试总结 . . . . .	176
9.29 花书前两部分总结 . . . . .	176
9.29.1 第三章 . . . . .	176
9.29.2 第四章 . . . . .	177
9.29.3 第五章 . . . . .	179
9.29.4 第六章 . . . . .	182
9.29.5 第七章 . . . . .	188
9.29.6 第八章 . . . . .	189
9.30 梯度消失 - 梯度爆炸专题 . . . . .	194
9.30.1 自圆其说的解释 . . . . .	195
9.31 ROI Pooling . . . . .	195
9.32 深度学习面试 100 题 . . . . .	195
9.32.1 Tensorflow 计算图 . . . . .	195
9.32.2 DL 调参 . . . . .	196
9.32.3 为什么 CNN 在 CV、NLP、Speech、AlphaGo 里面都有应用 . . . . .	196
9.32.4 为什么要引入非线性激励函数 . . . . .	197
9.32.5 为什么 ReLU 要好于 tanh 和 sigmoid function . . . . .	197
9.32.6 为啥 LSTM 模型中既存在 sigmoid 又存在 tanh 两种激活函数 . . . . .	197
9.32.7 如何解决 RNN 梯度爆炸和弥散问题 . . . . .	197
9.32.8 什么样的数据集不适合深度学习 . . . . .	198
9.32.9 如何解决梯度消失和梯度膨胀 . . . . .	198
9.32.10 激活函数的真正意义 . . . . .	198
9.32.11 梯度下降训练神经网络容易收敛到局部最优，为什么还应用广泛 . . . . .	199
9.33 数据的归一化 . . . . .	199
9.33.1 为什么需要归一化 . . . . .	199
9.33.2 归一化的方法 . . . . .	199
9.33.3 小结 . . . . .	200
9.34 待续 . . . . .	200

## CONTENTS

---

<b>10 Image Processing</b>	<b>201</b>
10.1 Feature Extraction . . . . .	201
10.1.1 SIFT . . . . .	201
<b>11 Feature Extraction</b>	<b>203</b>
11.1 Selective Search . . . . .	203
11.1.1 Efficient Graph-Based Image Segmentation . . . . .	203
11.1.2 Selective Search . . . . .	204
11.2 Region CNN . . . . .	204
11.2.1 概述 . . . . .	204
11.3 SPP Net . . . . .	204
11.3.1 背景与相关工作 . . . . .	204
11.3.2 网络结构 . . . . .	205
11.3.3 Object Detection Experiments . . . . .	206
11.3.4 Conclusion . . . . .	207
11.4 Fast RCNN . . . . .	208
11.4.1 ROI-Pooling 层的实现 -知乎 . . . . .	208
11.5 Faster RCNN . . . . .	209
11.6 R FCN . . . . .	209
11.7 Mask RCNN . . . . .	209
11.8 YOLO . . . . .	209
11.9 YOLO v2 . . . . .	209
11.10 YOLO v3 . . . . .	209
11.11 SSD . . . . .	209
11.12 DSSD . . . . .	209
11.13 Retina Net (Focal Loss) . . . . .	209
11.14 Feature Pyramid Network . . . . .	209
11.14.1 Nearest Neighbor Interpolation . . . . .	209
11.15 ResNet . . . . .	209

---

## CONTENTS

# List of Figures

1.1	Image Convolution based on Global Memory . . . . .	5
1.2	Image Convolution based on shared memory . . . . .	6
1.3	PTX Directives . . . . .	12
1.4	Reserved Instruction Keywords . . . . .	12
2.1	传统提升小波计算过程 . . . . .	14
2.2	Circuits for non-separable filters . . . . .	15
2.3	Performance of two-dimensional filters . . . . .	16
2.4	Comparison of RTL- and HLS-based design flows by using Gasjki-Kuhn's Y-chart: full lines indicate the automated cycles, while dotted lines the manual cycles. . . . .	17
3.1	Schematic diagram of the proposed image fusion method based on guided filtering. . . . .	24
5.1	DeLS Framework . . . . .	28
5.2	Segment Network in DeLS . . . . .	30
5.3	Dense SLAM 框架 . . . . .	33
5.4	Dense SLAM 中每一级的细节框架 . . . . .	33
5.5	粗糙的数据流图 . . . . .	37
5.6	算法的几个主要步骤 . . . . .	40
5.7	Semantic Mapping 系统概览 . . . . .	41
5.8	LiteFlowNet 结构框图 . . . . .	44
5.9	在 NetE 中的级联光流推理模块,M:S . . . . .	46
5.10	ExFusion 的实现框图 . . . . .	49
5.11	语义嵌入分支的结构图 . . . . .	50
5.12	基于 Encoder-Decoder CNN 的语义分割示意图 . . . . .	51
5.13	Overview of MaskFusion . . . . .	56
5.14	图像分割的算法框架 . . . . .	58
5.15	Atrous Convolution 示意图 . . . . .	60
5.16	Atrous Convolution 中的感受野变化示意图 . . . . .	60
5.17	论文提出的神经网络结构 . . . . .	62
5.18	ReNet layer 结构 . . . . .	64
5.19	U-Net 结构示意图 . . . . .	66
5.20	Generative Descriptor Learning 的示意图 . . . . .	69
5.21	StaticFusion 算法的三个主要流程 . . . . .	71
5.22	Mean field inference 算法步骤 . . . . .	74

---

5.23 SENet 结构示意图 . . . . .	76
5.24 网络结构示意图 . . . . .	78
5.25 Mask RCNN 示意图, 在 class box 的基础上增加了预测 mask 的分支。 . . . . .	79
5.26 ROI Align 的示意图, 如何将 region of interest 的原始图像和特征图精确对齐? . . . . .	80
5.27 论文的整体结构, 包括 Weight Transfer 结构 . . . . .	81
5.28 KITTI 官网截图 . . . . .	84
5.29 早期的数据文件组织格式 . . . . .	85
6.1 GAN 训练过程 . . . . .	92
6.2 CGAN 示意图, 在 G、N 网络中新增了数据 y . . . . .	95
7.1 The implicitly & explicitly back-propagation on Graph . . . . .	103
7.2 Dependencies can be found quickly. . . . .	103
7.3 Different backward path from forward path. . . . .	104
7.4 Standard Memory sharing between B & the result of E. . . . .	104
7.5 Standard Memory sharing between B & the result of E. . . . .	105
7.6 Standard Memory sharing between B & the result of E. . . . .	106
7.7 Standard Memory sharing between B & the result of E. . . . .	107
7.8 Color the longest paths in the Graph. . . . .	107
7.9 Operation Folding 示意图。 . . . . .	109
7.10 第一步, 给变量分配 tag . . . . .	112
7.11 把相应的 Function Closure push 进依赖分析引擎 . . . . .	113
7.12 一个具体的例子 . . . . .	113
7.13 Binary recordIO 数据结构 . . . . .	115
7.14 Binary recordIO 的一个例子 . . . . .	115
7.15 并行预处理例子 . . . . .	116
7.16 数据预取的示意图, 借助 Buffer 来实现 . . . . .	117
9.1 Dilated Convolution 示意图 . . . . .	142
9.2 Dilated Convolution 在 WaveNet 中的应用示意图 . . . . .	143
9.3 Transpose Convolution 过程示意图 . . . . .	144
9.4 一个例子, 用于卷积操作说明 . . . . .	146
9.5 卷积操作时的矩阵形式 . . . . .	146
9.6 三种不同的 Distillation Module . . . . .	148
9.7 Inception 卷积结构 . . . . .	150
9.8 Inception 结构与 Depthwise Separable Convolution 结构对比 . . . . .	151
9.9 Channel 分组的极限版本 . . . . .	152
9.10 Deformable Convolution 示意图 . . . . .	153
9.11 Deformable Convolution 的实现示意图 . . . . .	153
9.12 Squeeze-and-Excitation Block 示意图 . . . . .	154
9.13 SENet 与 Inception 以及 ResNet 结合的示意图 . . . . .	155
9.14 不同卷积策略的比较 . . . . .	155
9.15 Unpooling 示意图 . . . . .	156
9.16 文章的总结与说明 . . . . .	160
9.17 二维图像像素编号示意图 . . . . .	161
9.18 BP 计算过程示意图 . . . . .	164

## LIST OF FIGURES

---

9.19 目标函数对最后一层卷积层的输出, 即最后一层 Softmax 的输入的求导 . . . . .	165
9.20 则按照 mean-pooling, 首先得到的卷积层应该是 $4 \times 4$ 大小, 其值分布为 (等值复制) . . . . .	167
9.21 反向传播时的误差 . . . . .	167
9.22 Attention 模型示意图 . . . . .	169
9.23 $1 \times 1$ Convolution 的作用示意图 . . . . .	173
9.24 Confusion Matrix 示意图 . . . . .	176
9.25 激活函数对比 . . . . .	187
11.1 RCNN 整体思想 . . . . .	204
11.2 Pyramid Pooling 在网络结构中的示意图, 与传统 CNN 结构的比较 . . . . .	205
11.3 Pyramid Pooling 的计算过程示意图 . . . . .	207

---

**LIST OF FIGURES**

**Usage Instructions:**

- This book include all papers I have read from 2017.05.28.
- The **magenta** represent the online link.
- The **red** represents the links in this book including reference, figure, table and others.
- The **purple** represents the emphasize.

---

**LIST OF FIGURES**

# **Chapter 1**

## **Image processing based on CUDA**

This chapter include the Image processing acceleration based on CUDA.

### **1.1 Novel multi-scale retinex with color restoration on graphics processing unit**

#### **1.1.1 Abstract**

In this paper, a parallel application of the MSRCR+AL algorithm on a GPU is presented. For the various configurations in our test, the GPU-accelerated MSRCR+AL shows a scalable speedup as the resolution of an image increases. The up to  $45\times$  speed up ( $1024 \times 1024$ ) over the single-threaded CPU counterpart shows a promissing direction of using the GPU-based MSRCR+AL in large scale, time-critical applications. We also achieved 17 frames per second in video processing ( $1280 \times 720$ ).

#### **1.1.2 Content**

In our implementation, the CUFFT provides a simple interface for computing FFTs. After the plans of both forward and inverse FFTs are created according to the CUFFT requirements, the image data and the Gaussian filters can be parallel transformed to frequency domain. The multiplication between image data and Gaussian filters in frequency domain is finished by `ModulateandNormal- ize()` function, which is also provided by the CUFFT library.

The atomic function `atomicAdd()` provided by CUDA is used in the kernel histogram function to guarantee to be performed without interference from other threads.

#### **1.1.3 Parallel optimization strage**

##### **size of thread block and grid**

For example, the maximum number of threads on the lower capability version of CUDA is 512, but newer CUDA-enabled GPUs with 2.x compute capability can reach 1,024. But, each streaming multiprocessor (SM) can only execute 1,536 threads simul-

taneously. Therefore, we set the number of threads per block at 192, which means each SM can fully execute eight blocks to maximize resources.

### Memory access optimization

A thread needs 400–600 clock cycles to access the global memory, but only needs about 4 clock cycles to access fast memory units such as register and shared memory due to lower access latency. Therefore, taking full advantage of the multi-level GPU memory storage components can obtain quick data access to improve the execution performance effectively.

### Loop unrolling

After loop unrolling, the code only needs to run one time to write the result. The number of write processes decreases 66.7% compared with the serial code. Also, it only needs one time to read the result, compared with three times had we used in the serial code. The number of read processes decreases 66.7% as well. Furthermore, this loop unrolling strategy can be applied to sum different scale results together for the Multi-scale Retinex. More importantly, in the reduction algorithm for the summation process, this strategy is used to greatly increase the calculation speed and reduce the instruction overhead.

#### 1.1.4 Conclusion

see the Abstract subsection.

## 1.2 Image Convolution

This section includes all articles I have read relating to Image Convolution using CUDA. Image convolution is usually used in image filtering, like gaussian filter.

### 1.2.1 Naïve Implementation

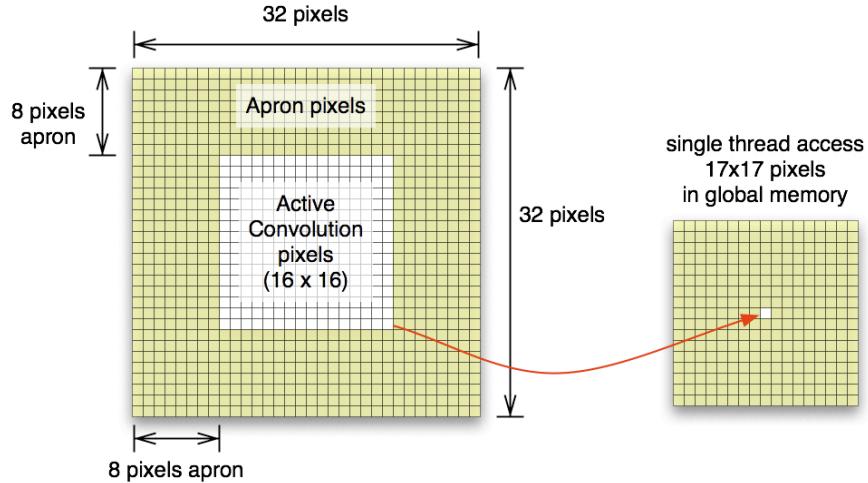
From the idea of convolution filter itself, the most naive approach is to use global memory to send data to device and each thread accesses this to compute convolution kernel. Our convolution kernel size is radius 8 (total  $17 \times 17$  multiplication for single pixel value). In image border area, reference value will be set to 0 during computation. This naive approach includes many of conditional statements and this causes very slow execution. The code is as shown below:

Listing 1.1: Image Convolution based on global memory

```
--global__ void gaussfilterGlo_kernel(float *d_imgOut, float *
d_imgIn, int wid, int hei,
```

## 1.2 Image Convolution

---



**Fig 1.1.** Image Convolution based on Global Memory

```
{  
    int idx = threadIdx.x + blockDim.x * blockIdx.x;  
    int idy = threadIdx.y + blockDim.y * blockIdx.y;  
  
    if(idx > wid || idy > hei)  
        return ;  
  
    int filterR = (filterW - 1) / 2;  
  
    float val = 0.f;  
  
    for(int fr = -filterR; fr <= filterR; ++fr)           // row  
        for(int fc = -filterR; fc <= filterR; ++fc)       // col  
    {  
        int ir = idy + fr;  
        int ic = idx + fc;  
  
        if((ic >= 0) && (ic <= wid - 1) && (ir >= 0) && (ir <= hei - 1))  
            val += d_imgIn[INDX(ir, ic, wid)] * d_filter[INDX(fr +filterR, fc+filterR, filterW)];  
    }  
    d_imgOut[INDX(idy, idx, wid)] = val;
```

}

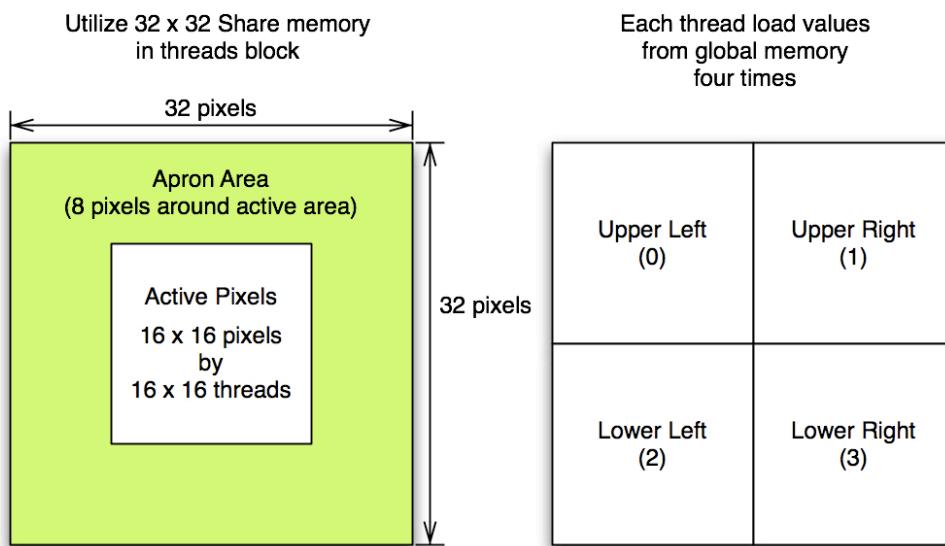
For  $396 \times 396$  input image, the time is 1.6ms. When the input filter is stored in constant memory or specified by 'restrict', the time is 1.7 or 1.8 ms.

### 1.2.2 Naïve Shared Memory Implementation

The simplest approach to implement convolution in CUDA is to load a block of the image into a shared memory array, do a point-wise multiplication of a filter-size portion of the block, and then write this sum into the output image in device memory. Each thread block processes one block in the image. Each thread generates a single output pixel.

The algorithm itself is somewhat complex. For any reasonable filter kernel size, the pixels at the edge of the shared memory array will depend on pixels not in shared memory. Around the image block within a thread block, there is an *apron* of pixels of the width of the kernel radius that is required in order to filter the image block. Thus, each thread block must load into shared memory the pixels to be filtered and the apron pixels.

Note: The apron of one block overlaps with adjacent blocks. The aprons of the blocks on the edges of the image extend outside the image – these pixels can either be clamped to the color of pixels at the image edge, or they can be set to zero.



**Fig 1.2.** Image Convolution based on shared memory

The first attempt was to keep active thread size as same as previous and increase block size for apron pixels. This did not work since convolution kernel radius is 8 and it make block size to  $32 \times 32$  (1024). This is bigger than G80 hardware limit (512 threads max per block).

Therefore, I changes scheme as all threads are active and each thread loads four pixels and keep the block size  $16 \times 16$ . Shared Memory size used is  $32 \times 32$  (this includes all necessary apron pixel values for  $16 \times 16$  active pixels). Below shows quite a bit of performance improve. This is almost  $\times 2.8$  speed up over naive approach (in 2048 resolution).

## 1.2 Image Convolution

---

Listing 1.2: Image convolution based on Shared memory and Apron

```
__global__ void convolutionGPU(
.....          float *d_Result,
.....          float *d_Data,
.....          int dataW,
.....          int dataH
.....)
{
....// Data cache: threadIdx.x , threadIdx.y
....__shared__ float data[TILE_W + KERNEL_RADIUS * 2][TILE_W +
    KERNEL_RADIUS * 2];

....// global mem address of this thread
....const int gLoc = threadIdx.x +
.....          IMUL(blockIdx.x, blockDim.x) +
.....          IMUL(threadIdx.y, dataW) +
.....          IMUL(blockIdx.y, blockDim.y) * dataW;

....// load cache (32x32 shared memory, 16x16 threads blocks)
....// each threads loads four values from global memory into shared
mem
....// if in image area, get value in global mem, else 0
....int x, y; // image based coordinate

....// original image based coordinate
....const int x0 = threadIdx.x + IMUL(blockIdx.x, blockDim.x);
....const int y0 = threadIdx.y + IMUL(blockIdx.y, blockDim.y);

....// case1: upper left
....x = x0 - KERNEL_RADIUS;
....y = y0 - KERNEL_RADIUS;
....if ( x < 0 || y < 0 )
.....data[threadIdx.x][threadIdx.y] = 0;
....else
.....data[threadIdx.x][threadIdx.y] = d_Data[ gLoc -
    KERNEL_RADIUS - IMUL(dataW, KERNEL_RADIUS)];

....// case2: upper right
....x = x0 + KERNEL_RADIUS;
....y = y0 - KERNEL_RADIUS;
....if ( x > dataW-1 || y < 0 )
.....data[threadIdx.x + blockDim.x][threadIdx.y] = 0;
....else
.....data[threadIdx.x + blockDim.x][threadIdx.y] = d_Data[gLoc +
    KERNEL_RADIUS - IMUL(dataW, KERNEL_RADIUS)];

....// case3: lower left
....x = x0 - KERNEL_RADIUS;
....y = y0 + KERNEL_RADIUS;
....if ( x < 0 || y > dataH-1)
.....data[threadIdx.x][threadIdx.y + blockDim.y] = 0;
....else
.....data[threadIdx.x][threadIdx.y + blockDim.y] = d_Data[gLoc -
    KERNEL_RADIUS + IMUL(dataW, KERNEL_RADIUS)];
```

```

.....// case4: lower right
.....x = x0 + KERNEL_RADIUS;
.....y = y0 + KERNEL_RADIUS;
.....if ( x > dataW-1 || y > dataH-1)
.....    data[threadIdx.x + blockDim.x][threadIdx.y + blockDim.y] =
0;
.....else
.....    data[threadIdx.x + blockDim.x][threadIdx.y + blockDim.y] =
d_Data[gLoc + KERNEL_RADIUS + IMUL(dataW, KERNEL_RADIUS)];
.....__syncthreads();

.....// convolution
.....float sum = 0;
.....x = KERNEL_RADIUS + threadIdx.x;
.....y = KERNEL_RADIUS + threadIdx.y;
.....for (int i = -KERNEL_RADIUS; i <= KERNEL_RADIUS; i++)
.....    for (int j = -KERNEL_RADIUS; j <= KERNEL_RADIUS; j++)
.....        sum += data[x + i][y + j] * d_Kernel[KERNEL_RADIUS + j]
* d_Kernel[KERNEL_RADIUS + i];
.....d_Result[gLoc] = sum;
}

```

Note: the value “ $gLoc - KERNEL\_RADIUS - IMUL(dataW, KERNEL\_RADIUS)$ ” is the shift address of the image data on the upper left corner. 在本方法中，主要是索引的问题，所以又分为 Share Memory 的索引以及图像数据的索引。在具体实现过程中，选择是固定 thread Block 的大小，同时在 share memory 中添加边界。所以将图像数据拷贝到 Share Memory 中时，分了四次，分别对应：左上角，右上角，左下角、右下角。也就是图1.2中的对应关系，其处理过程就是将小的图像块映射到大的 Share memory 中，从这个方面进行理解。

### 1.2.3 Separable Gaussian Filtering

#### Separable Convolution

A two-dimensional filter  $s$  is said to be separable if it can be written as the convolution of two one-dimensional filters  $v$  and  $h$ :

$$s = v * h$$

"How to determine if a matrix is an outer product of two vectors?"

"Go look at the **rank** function.". Of course. If a matrix is an outer product of two vectors, its rank is 1.

So the test is this: The rank of A is the number of nonzero singular values of A, with some numerical tolerance based on eps and the size of A.

So how can we determine the outer product vectors? The answer is to go back to the svd function. Here's a snippet from the doc:

$[U, S, V] = svd(X)$  produces a diagonal matrix  $S$  of the same dimension as  $X$ , with nonnegative diagonal elements in decreasing order, and unitary matrices  $U$  and  $V$  so that  $X = U * S * V'$

## 1.2 Image Convolution

---

A rank 1 matrix has only one nonzero singular value, so  $X = U * S * V'$  becomes  $U(:, 1) * S(1, 1) * V(:, 1)$ . This is basically the outer product we were seeking. Therefore, we want the first columns of  $U$  and  $V$ . (We have to remember also to use the nonzero singular value as a scale factor.)

### Separate Gaussian filter

First chose, somewhat arbitrarily to split the scale factor,  $S(1, 1)$ , "equally" between  $v$  and  $h$ . Except for normal floating-point roundoff differences, gaussian and  $v * h$  are equal. Just show as following :

$$\begin{aligned} [U, S, V] &= svd(X) \\ v &= U(:, 1) * \text{sqrt}(S(1, 1)) \\ h &= V(:, 1) * \text{sqrt}(S(1, 1)) \\ \text{GaussianFilter} &= v * h \end{aligned}$$

More details can be found at :[Separable Convolution](#).

### 1.2.4 Optimizing for memory coalescence

Base read/write addresses of the warps of 32 threads also must meet half-warp alignment requirement in order to be coalesced. If four-byte values are read, then the base address for the warp must be 64-byte aligned, and threads within the warp must read sequential 4-byte addresses. If the dataset with apron does not align in this way, then we must fix it so that it does.

The approach used in the row filter is to have additional threads on the leading edge of the processing tile, in order to make `threadIdx.x == 0` always reading properly aligned address and thus to meet global memory alignment constraints for all warps. This may seem like a waste of threads, but it is of little importance when the data block, processed by a single thread block is large enough, which decreases the ratio of apron pixels to output pixels.

Each image convolution pass in both row and column pass is separated into two sub stages within corresponding CUDA kernels. The first stage loads the data from global memory into shared memory, and the second stage performs the filtering and writes the results back to global memory. We mustn't forget about the cases when row or column processing tile becomes clamped by image borders, and initialize clamped shared memory array indices with correct values. Indices not lying within input image borders are usually initialized either with zeroes or with values, corresponding to clamped image coordinates. In this sample we opt for the former.

In between the two stages there is a `__syncthreads()` call to ensure that all threads have written to shared memory before any processing begins. This is necessary because threads are dependent on data loaded by other threads.

For both the loading and processing stages each active thread loads/outputs one pixel. In the computation stage each thread loops over a width of twice the filter radius plus 1, multiplying each pixel by the corresponding filter coefficient stored in constant memory. Each thread in a half-warp accesses the same constant address and hence there is no penalty due to constant memory bank conflicts. Also, consecutive threads always access consecutive shared memory addresses so no shared memory bank conflicts occur as well.

The column filter pass operates much like the row filter pass. The major difference is that thread IDs increase across the filter region rather than along it. As in the row filter pass, threads in a single half-warp always access different shared memory banks, but the calculation of the next/previous addresses involves increment/decrement by COL-UMN\_TILE\_W, rather than simply 1. In the column filter pass we do not have inactive “coalescing alignment” threads during the load stage, because we assume that the tile width is a multiple of the coalesced read size. In order to decrease the ratio of apron to output pixels we want image tile to be as tall as possible, so to have reasonable shared memory utilization we shoot for as thin image tiles as possible: 16 columns.

## 1.3 CUDA C Best Practice Guide

### 1.3.1 Performance Metrics

#### Timing

- Using CPU Timers  
Should call `cudaDeviceSynchronize()` immediately before starting and stopping the CPU timer.
- Using CUDA GPU Timers  
The device will record a timestamp for the event when it reaches that event in the stream. This value is expressed in milliseconds and has a resolution of approximately half a microsecond.

#### Bandwidth

Bandwidth - the rate at which data can be transferred - is one of the most important gating factors for performance.

## 1.4 Npp Library Image Filters

### 1.4.1 Image Data

#### Line Step

All image data passed to NPPI primitives requires a line step to be provided. It is important to keep in mind that this line step is always specified in terms of bytes, not pixels.

## 1.5 PTX ISA 6.0

### 1.5.1 PTX Machine Model

The *Multiprocessor* maps each thread to one *scalar processor* core, and each scalar thread executes independently with its own instruction address and register state.

Individual threads composing a SIMT warp start together at the same program address but are otherwise free to branch and execute independently. A warp executes one common instruction at a time, so full efficiency is realized when all threads of a warp agree on their execution path. If threads of a warp diverge via a data-dependent conditional branch, the warp serially executes each branch path taken, disabling threads that are not on that path, and when all paths complete, the threads converge back to the same execution path.

Each multiprocessor has **on-chip memory** of the four following types :

- Local 32-bit registers per processor;
- Shared memory (parallel data cache);
- Read-only *constant cache* that is shared by all scalar processor cores;
- Read-only *texture cache*

The local and global memory spaces are read-write regions of **device memory** and are not cached.

If there are not enough registers or shared memory available per multiprocessor to process at least one block, the kernel will fail to launch.

### Syntax

PTX programs are a collection of text source modules(files). PTX source modules have an assembly-language style syntax with instruction operation codes and operands. Pseudo-operations specify symbol and addressing management. The *ptxas* optimizing backend compiler optimizes and assembles PTX source modules to produce corresponding binary object files.

### Source Format

PTX is case sensitive and uses lowercase for keywords.

Each PTX module must begin with a **.version** directive specifying the PTX language version, followed by a **.target** directive specifying the target architecture assumed.

### Statements

A PTX statement is either a **directive** or an **instruction**. Statements begin with an optional label and end with a semicolon.

- Directive Statements

Directive keywords begin with a dot, so no conflict is possible with user-defined identifiers. 如图1.3所示。

- Instruction Statements

Instructions are formed from an instruction opcode followed by a **comma-separated** list of zero or more operands, and terminated with a semicolon.

Table 1 PTX Directives

.address_size	.file	.minnctapersm	.target
.align	.func	.param	.tex
.branchtargets	.global	.pragma	.version
.callprototype	.loc	.reg	.visible
.calltargets	.local	.reqntid	.weak
.const	.maxnctapersm	.section	
.entry	.maxnreg	.shared	
.extern	.maxntid	.sreg	

Fig 1.3. PTX Directives

Operands may be register variables, constant expressions, address expressions, or label names. The guard predicate follows the optional label and precedes the op-code, and is written as `@p`, where p is a predicate register. The guard predicate may be optionally negated, written as `@!p`.

The destination operand is first, followed by source operands. 如图1.4所示。

Table 2 Reserved Instruction Keywords

abs	div	or	sin	vavrg2, vavrg4
add	ex2	pmevent	slct	vmad
addc	exit	popc	sqrt	vmax
and	fma	prefetch	st	vmax2, vmax4
atom	isspacep	prefetchu	sub	vmin
bar	ld	prmt	subc	vmin2, vmin4
bfe	ldu	rcp	suld	vote
bfi	lg2	red	suq	vset
bfind	mad	rem	sured	vset2, vset4
bra	mad24	ret	sust	vshl
brev	madc	rsgrt	testp	vshr
brkpt	max	sad	tex	vsub
call	membar	selp	tld4	vsub2, vsub4
clz	min	set	trap	xor
cnot	mov	setp	txq	
copysign	mul	shf	vabsdiff	
cos	mul 24	shfl	vabsdiff2, vabsdiff4	
cvt	neg	shl	vadd	
cvt	not	shr	vadd2, vadd4	

Fig 1.4. Reserved Instruction Keywords

# Chapter 2

## FPGAs

Every section is arranged like follows:

- Abstract  
The abstract part of paper.
- Content  
The main idea of paper.
- Results  
The experiment implementation.
- Conclusion  
The conclusion part of paper.

### 2.1 Performance Comparison of FPGA, GPU and CPU in Image Processing 2009

#### 2.1.1 Abstract

**Many applications in image processing have high inherent parallelism.** FPGAs have shown very high performance in spite of their low operational frequency by fully extracting the parallelism. In recent micro processors, it also becomes possible to utilize the parallelism using multi-cores which support improved SIMD instructions, though programmers have to use them explicitly to achieve high performance. Recent GPUs support a large number of cores, and have a potential for high performance in many applications. **However, the cores are grouped, and data transfer between the groups is very limited.** Programming tools for FPGA, SIMD instructions on CPU and a large number of cores on GPU have been developed, but it is still difficult to achieve high performance on these platforms. In this paper, we compare the performance of FPGA, GPU and CPU using three applications in image processing; two-dimensional filters, stereo-vision and k-means clustering, and make it clear which platform is faster under which conditions.



Fig 2.1. 传统提升小波计算过程

### 2.1.2 Content

Compared three applications: two-dimension filters, stereo-vision, k-means clustering.

The high performance of FPGA comes from its flexibility which makes it possible to realize the fully optimized circuit for each application, and a large number of on-chip memory banks which supports the high parallelism. **FPGA can achieve extremely high performance in many applications in spite of its low operational frequency.**

GPU cores are grouped, the data transfer between groups is very slow.

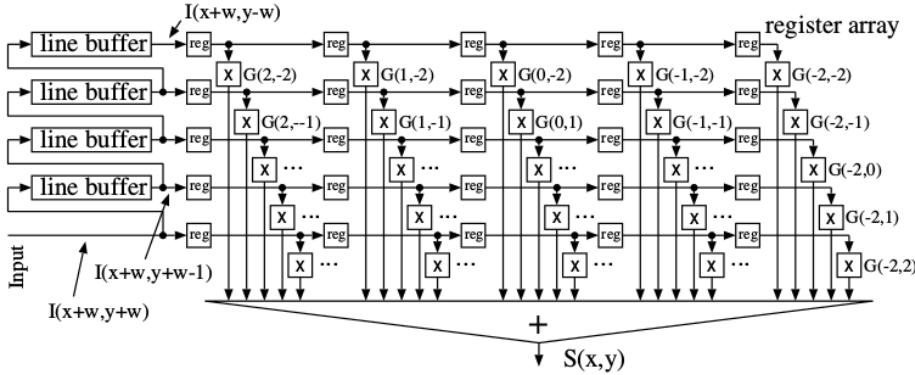
#### GPU Analysis

It consists of 10 thread processor clusters. A thread processor cluster has three streaming multiprocessors, eight texture filtering units, and one level-1 cache memory. Each streaming multiprocessor has one instruction unit, eight stream processors (SPs) and one local memory (16KB). Thus, GTX280 has 240 SPs in total. Eight SPs in a stream-

ing multiprocessor are connected to one instruction unit. This means that the eight SPs execute the same instruction stream on different data.

### Two-Dimensional Filters

The computational complexity of filters is  $O(w \times w)$ , and  $w$  is radius of filter.



**Fig 2.2.** Circuits for non-separable filters

Fig2.2 shows the filter is  $5 \times 5$  case.

### Stereo Vision

This application is to get the distance to the location obtained from the two camera's disparity. The sum of absolute difference (SAD) is widely used to compare the windows because of its simplicity. More details can be found in paper[1]. **K-means Clustering**

More details can be found in paper [1].

### 2.1.3 Results

- Xilinx XC4VLX160.
- GeForce 280GTX, 1 GB DDR3, CUDA version 2.1.
- Intel Core2 Extreme QX6859.

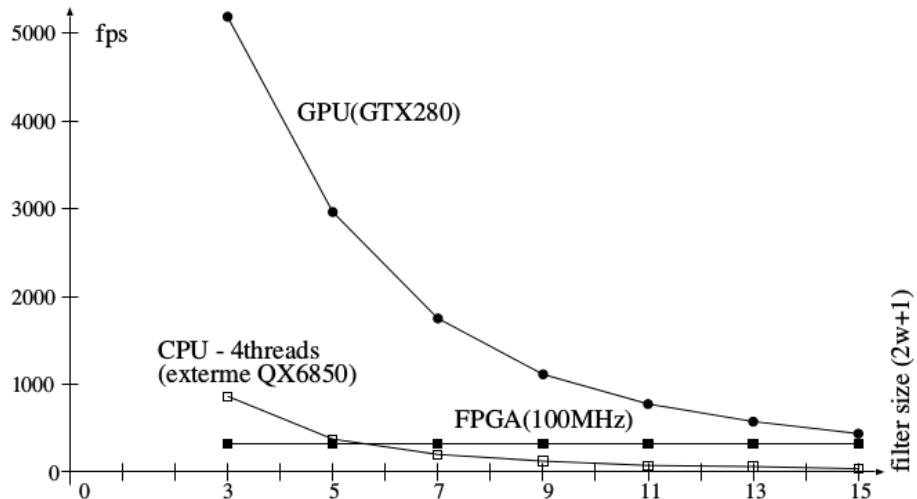
The time to download images from main memory is not included. CPU has four cores, FPGA is fixed to 100MHz. Fig is the performance of two-dimension filters.

GPU is the fastest for all tested filter size. In this problem, filters can be applied to each pixel in the image independently without using shared variables. So, GPU can show its best performance.

In the later two applications, the performance FPGA is much better than GPU.

### 2.1.4 Conclusion

We have compared the performance of GPU with FPGA and CPU (quad-cores) using three simple problems in image processing. GPU has a potential for achieving almost the same performance with FPGA. The number of cores in GTX280 is 240. Considering the trade-offs between the operational frequency of GPU (more than 10 times faster), and



**Fig 2.3.** Performance of two-dimensional filters

the fine-grained parallelism in FPGA, this seems to be a natural consequence. However, GPU can show its potential only for naive computation methods, in which all pixels can be processed independently. For more sophisticated algorithms which use shared arrays, GPU can not execute those algorithms because of its very small local memory, or can not show good performance because of the memory access limitation caused by its memory architecture. GPU is slower than CPU in those algorithms (it may be possible to realize much better performance if we can find algorithms which can get around the limitations, but we could not find them). The performance of CPU is 1/12 - 1/7 of FPGA, which means that CPU with quad-cores can executes about 1/10 operations of FPGA in a unit time (the same algorithms are executed on CPU and FPGA). The performance of FPGA is limited by the size of FPGA and the memory bandwidth. With a latest FPGA board with DDR-II DRAM and a larger FPGA, it possible to double the performance by processing twice the number of pixels in parallel.

We have the following issues which have to be considered. We have compared the performance using only three problems. The performances of the programs on GPU and CPU are not fully tuned up. In the comparison, power consumption and costs are not considered.

## 2.2 Fast FPGA Prototyping for real-time image processing with very high-level synthesis 2017

### 2.2.1 Abstract

Programming in high abstraction level can facilitate the development of digital signal processing systems. In the recent 20 years, HLS has made significantly progress. However, due to the high complexity and computational intensity, image processing algorithms usually necessitate a higher abstraction environment than C-synthesis, and the current HLS tools do not have the ability of this kind. This paper presents a conception of

## 2.2 Fast FPGA Prototyping for real-time image processing with very high-level synthesis 2017

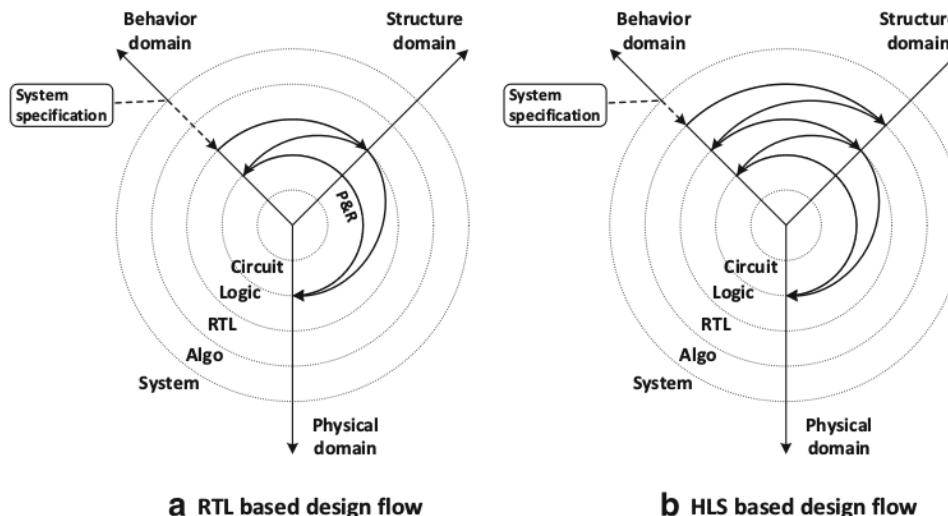
very high-level synthesis method which allows fast prototyping and verifying the FPGA-based image processing designs in the MATLAB environment. We build a heterogeneous development flow by using currently available tool kits for verifying the proposed approach and evaluated it within two real-life applications. Experiment results demonstrate that it can effectively reduce the complexity of the development by automatically synthesizing the algorithm behavior from the user level into the low register transfer level and give play to the advantages of FPGA related to the other devices.

### 2.2.2 Content

Advanced Digital Sciences Center(ADSC) of the University of Illinois reported that FPGA can achieve a speedup to 2-2.5x and save 84-92% of the energy consumption comparing to Graphics Processing Units(GPUs). ADSC indicates also that a manual FPGA design may consume 6-18 months and even years for a full custom hardware, while the GPUs(CUDA) based designed only 1-2 weeks.

Fig2.4 show the Gasjki-Kuhn's Y-chart comparing the conventional RTL with the HLS-based design flows.

The challenges of MATLAB-to-RTL synthesis include:



**Fig 2.4.** Comparison of RTL- and HLS-based design flows by using Gasjki-Kuhn's Y-chart: full lines indicate the automated cycles, while dotted lines the manual cycles.

- Operators in MATLAB perform different operations depending on the type of the operands, whereas the functions of the operators in RTL are fixed.
- MATLAB includes very simple and powerful vector operations such as the concatenation ``[ ]'' and column operators `` $x(:)$ '' or ``end'' construct, which can be quite hard to map to RTL.

- MATLAB supports ``polymorphism" whereas RTL does not. More precisely, functions in MATLAB are generic and can process different types of input parameters. In the behaviors of RTL, each parameter has only a single given type, which cannot change.
- MATLAB supports dynamic loop bounds or vector size, whereas RTL requires users to initialize explicitly them and cannot do any changes during the synthesis.
- The variables in MATLAB can be reused for different contents (different types), whereas RTL does not, as each variable has one unique type.

Two complex image processing applications: Kubelka-Munk genetic algorithm(KMGA) for the multispectral image based skin lesion assessments & level set method(LSM)-based algorithm for very high-resolution(VHR) satellite image segmentation..

# Chapter 3

## Image Fusion

### 3.1 Guided Image Filter 2013

#### 3.1.1 Abstract

In this paper, we propose a novel explicit image filter called guided filter. Derived from a local linear model, the guided filter computes the filtering output by considering the content of a guidance image, which can be the input image itself or another different image. The guided filter can be used as an edge-preserving smoothing operator like the popular bilateral filter, but it has better behaviors near edges. The guided filter is also a more generic concept beyond smoothing: It can transfer the structures of the guidance image to the filtering output, enabling new filtering applications like dehazing and guided feathering. Moreover, the guided filter naturally has a fast and nonapproximate linear time algorithm, regardless of the kernel size and the intensity range. Currently, it is one of the fastest edge-preserving filters. Experiments show that the guided filter is both effective and efficient in a great variety of computer vision and computer graphics applications, including edge-aware smoothing, detail enhancement, HDR compression, image matting/feathering, dehazing, joint upsampling, etc.

#### 3.1.2 Content

A general linear translation-variant filtering process, which involves a guidance image  $I$ , an filtering input image  $p$  and an output image  $q$ . Both  $I$  and  $p$  are given beforehand according to the application, and they can be **identical!** The filtering output at a pixel  $i$  is expressed as a weight average:

$$q_i = \sum_j W_{ij}(I)p_j \quad (3.1)$$

where  $i, j$  are pixel indexes. The filter kernel  $W_{ij}$  is a function of the guidance image  $I$  and independent of  $p$ .

### 3.1.3 Conclusion

## 3.2 Multiscale Image Fusion Through Guided Filtering

### 3.2.1 Abstract

We introduce a multiscale image fusion scheme based on GUided Filtering. Guided filtering can effectively reduce noise while preserving details boundaries. While restoring larger scale edges. The proposed multi-scale fusion scheme achieves optimal spatial consistency by using guided filtering both at the decomposing and at the recombination stage of the multiscale fusion process. First, size-selective iterative guided filtering is applied to decompose the source images into base and detail layers at multiple levels of resolution. Next, at each resolution level a binary weighting map is obtained as the pixelwise maximum of corresponding source saliency maps. Guided filtering of the binary weighting maps with their corresponding source images as guidance images serves to reduce noise and to restore spatial consistency. The final fused image is obtained as the weighted recombination of the individual detail layers and the mean of the lowest resolution base layers. Application to multiband visual (intensified) and thermal infrared imagery demonstrates that the proposed method obtains state-of-the-art performance for the fusion of multispectral nightvision images. The method has a simple implementation and is computationally efficient[33].

### 3.2.2 Contents

To date, a variety of image fusion algorithms have been proposed. A popular class of algorithms are the multi-scale image fusion schemes, which decompose the source images into spatial primitives at multiple spatial scales, then integrate these primitives to form a new multi-scale transform-based representation, and finally apply an inverse multi-scale transform to reconstruct the fused image. However, most of these techniques are computationally expensive and tend to oversharpen edges, which makes them less suitable for application in multiscale schemes

Bilateral Filter: It can reverse the intensity gradient near sharp edges.

Guided Filter:

The two filtering conditions are:

- the local filter output is a linear transform of the guidance image  $G$
- as similar as possible to the input image  $I$ .

The first condition implies that:

$$O_i = a_k G_i + b_k \quad \forall i \in \omega_k$$

where the  $\omega_k$  is a square window of size  $(2r + 1) \times (2r + 1)$ . **The local linear model ensures that the output image  $O$  has an edge only at locations where the guidance image  $G$  also has one.** Linear coefficients  $a_k$  and  $b_k$  are constant in  $\omega_k$ . They can be

### 3.2 Multiscale Image Fusion Through Guided Filtering

---

estimated by minimizing the squared difference between the output image  $O$  and the input image  $I$  in the window  $\omega_k$ , i.e. minimizing the cost function  $E$ :

$$E(a_k, b_k) = \sum_{i \in \omega_k} ((a_k G_i + b_k - I_i)^2 + \epsilon a_k^2)$$

where  $\epsilon_k$  is a regularization parameter penalizing large  $a_k$ . The coefficients  $a_k$  and  $b_k$  can directly be solved by linear regression. Since pixel  $i$  is contained in several different window  $\omega_k$ , the value of  $O_i$  depends on the window over which it is calculated:

$$O_I = \bar{a}_i G_i + \bar{b}_i$$

The abrupt intensity changes in the guiding image  $G$  are still largely preserved in the output image  $O$ . The guided filter is a computationally efficient, edge-preserving operator which avoids the gradient reversal artefacts of the bilateral filter.

In Iterative guided filtering: In such a scheme the result  $G^{t+1}$  of the  $t$ -th iteration is obtained from the joint bilateral filtering of the input image  $I$  using the result  $G^t$  of the previous iteration step:

$$G_i^{t+1} = \frac{1}{K_i} \sum_{j \in \omega} I_j \cdot f(\|i - j\|) \cdot g(\|G_i^t - G_j^t\|)$$

Note that the initial guidance image  $G^1$  can simply be a constant (e.g. zero) valued image since it updates to the Gaussian filtered input image in the first iteration step.

Proposed Method:

- Iterative guided filtering is applied to decompose the source images into base layers (representing large scale variations) and detail layers (containing small scale variations).
- Frequency-tuned filtering is used to generate saliency maps for the source images.
- Binary weighting maps are computed as the pixelwise maximum of the individual source saliency maps.
- Guided filtering is applied to each binary weighting map with its corresponding source as the guidance image to reduce noise and to restore spatial consistency.
- The fused image is computed as a weighted recombination of the individual source detail layers.

**Visual saliency** refers to the physical, bottom-up distinctness of image details. It is a relative property that depends on the degree to which a detail is visually distinct from its background. **Since saliency quantifies the relative visual importance of image details saliency maps are frequently used in the weighted recombination phase of multiscale image fusion schemes.** Frequency tuned filtering computes bottom-up saliency as local multiscale luminance contrast. The saliency map  $S$  for an image  $I$  is computed as

$$S(x, y) = \|I_\mu - I_f(x, y)\|$$

where

$I_\mu$  is the arithmetic mean image feature vector

$I_f$  represents a Gaussian blurred version of the original image, using a  $5 * 5$  separable binomial kernel

$\| \cdot \|$  is the  $L_2$  norm(Euclidian distance), and  $x, y$  are the pixel coordinates.

We compute saliency using frequency tuned filtering since a recent and extensive evaluation study comparing 13 state-of-the-art saliency models found that the output of this simple saliency model correlates more strongly with human visual perception than the output produced by any of the other available models.

Binary weight maps  $BW_{X_i}$  and  $BW_{Y_i}$  are then computed by taking the pixelwise maximum of corresponding saliency maps  $S_{X_i}$  and  $S_{Y_i}$ :

$$BW_{X_i}(x, y) = \begin{cases} 1 & \text{if } S_{X_i}(x, y) > S_{Y_i}(x, y) \\ 0 & \text{otherwise} \end{cases}$$

$$BW_{Y_i}(x, y) = \begin{cases} 1 & \text{if } S_{Y_i}(x, y) > S_{X_i}(x, y) \\ 0 & \text{otherwise} \end{cases}$$

The resulting binary weight maps are noisy and typically not well aligned with object boundaries, which may give rise to artefacts in the final fused image. Spatial consistency is therefore restored through guided filtering (GF) of these binary weight maps with the corresponding source layers as guidance images

$$W_{X_i} = GF(BW_{X_i}, X_i)$$

$$W_{Y_i} = GF(BW_{Y_i}, Y_i)$$

Fused detail layers are then computed as the normalized weighted mean of the corresponding source detail layers:

$$dF_i = \frac{W_{X_i} \cdot dX_i + W_{Y_i} \cdot dY_i}{W_{X_i} + W_{Y_i}}$$

The fused image  $F$  is finally obtained by adding the fused detail layers to the average value of the lowest resolution source layers:

$$F = \frac{X_3 + Y_3}{2} + \sum_{i=0}^2 dF_i$$

By using guided filtering both in the decomposition stage and in the recombination stage, this proposed fusion scheme optimally benefits from both the multi-scale edge-preserving characteristics (in the iterative framework) and the structure restoring capabilities (through guidance by the original source images) of the guided filter. The method is easy to implement and computationally efficient.

### 3.2.3 Conclusion

We propose a multiscale image fusion scheme based on guided filtering. Iterative guided filtering is used to decompose the source images into base and detail layers. Initial binary weighting maps are computed as the pixelwise maximum of the individual

### 3.3 Image Fusion With Guided Filtering

---

source saliency maps, obtained from frequency tuned filtering. Spatially consistent and smooth weighting maps are then obtained through guided filtering of the binary weighting maps with their corresponding source layers as guidance images. Saliency weighted recombination of the individual source detail layers and the mean of the lowest resolution source layers finally yields the fused image. The proposed multi-scale image fusion scheme achieves spatial consistency by using guided filtering both at the decomposition and at the recombination stage of the multiscale fusion process. Application to multiband visual (intensified) and thermal infrared imagery demonstrates that the proposed method obtains state-of-the-art performance for the fusion of multispectral nightvision images. The method has a simple implementation and is computationally efficient.

## 3.3 Image Fusion With Guided Filtering

### 3.3.1 Abstract

A fast and effective image fusion method is proposed for creating a highly informative fused image through merging multiple images. The proposed method is based on a two-scale decomposition of an image into a base layer containing large scale variations in intensity, and a detail layer capturing small scale details. A novel guided filtering-based weighted average technique is proposed to make full use of spatial consistency for fusion of the base and detail layers. Experimental results demonstrate that the proposed method can obtain state-of-the-art performance for fusion of multispectral, multifocus, multimodal, and multiexposure images.

### 3.3.2 Contents

#### Guided Filter

The filtering output  $O$  is linear transformation of the guidance image  $I$  in a local window  $\omega_k$  centered at pixel  $k$ :

$$O_i = a_k I_i + b_k \quad \forall i \in \omega_k$$

where  $\omega_k$  is a square window of size  $(2r+1) \times (2r+1)$ . The linear coefficients  $a_k$  and  $b_k$  are constant in  $\omega_k$  by minimizing the squared difference between the output image  $O$  and the input image  $P$ :

$$E(a_k, b_k) = \sum_{i \in \omega_k} ((a_k I_i + b_k - P_i) + \epsilon a_k^2)$$

where  $\epsilon$  is a regularization parameter given by the user. The coefficients can be directly solved by linear regression as follows:

$$a_k = \frac{\frac{1}{|\omega|} \sum_{i \in \omega_k} I_i P_i - \mu_k \bar{P}_k}{\delta_k + \epsilon}$$
$$b_k = \bar{P}_k - a_k \mu_k$$

where  $\mu_k$  and  $\delta_k$  are the mean and variance of  $I$  in  $\omega_k$  respectively.  $|\omega|$  is the number of pixels in  $\omega_k$ , and  $\bar{P}_k$  is the mean of  $P$  in  $\omega_k$ . Then the output image can be calculated according to above equation.

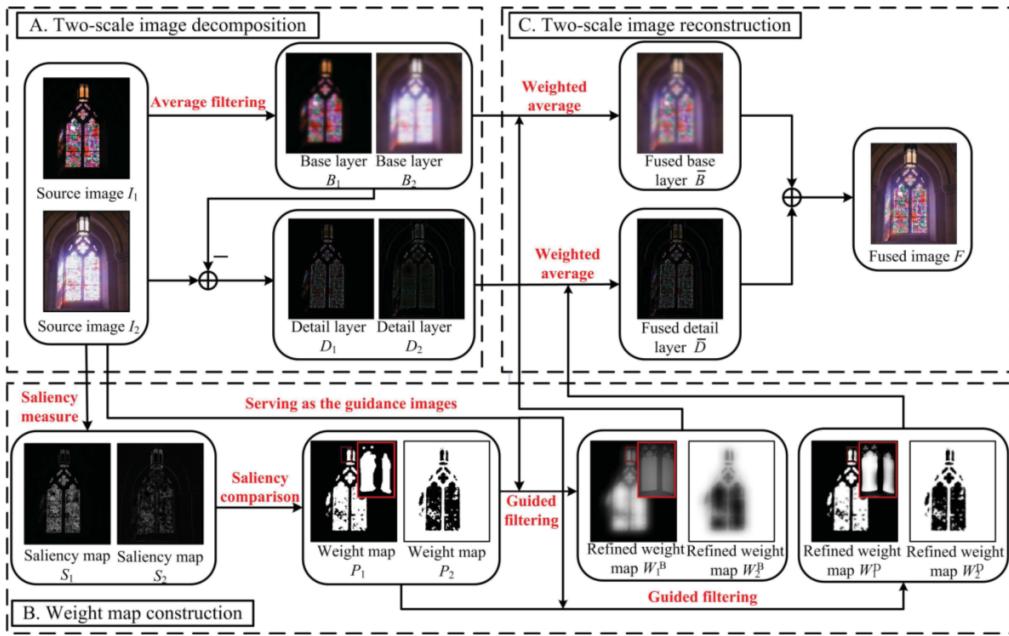
$$O_i = \bar{a}_i I_i + \bar{b}_i$$

where  $\bar{a}_i = \frac{1}{|\omega|} \sum_{k \in \omega_i} a_k$ ,  $\bar{b}_i = \frac{1}{|\omega|} \sum_{k \in \omega_i} b_k$ .

The color image situation, the  $a_i$  and other calculators become vector version. See [15].

### 3.3.3 Fusion Frame

See figure 3.1.



**Fig 3.1.** Schematic diagram of the proposed image fusion method based on guided filtering.

### 3.3.4 Conclusion

# Chapter 4

## Saliency Detection

This chapter includes papers about saliency detection.

### 4.1 Frequency-tuned Salient Region Detection

#### 4.1.1 Abstract

In this paper, we introduce a method for salient region detection that outputs full resolution saliency maps with well-defined boundaries of salient objects. These boundaries are preserved by retaining substantially more frequency content from the original image than other existing techniques. Our method exploits features of color and luminance, is simple to implement, and is computationally efficient. We compare our algorithm to five state-of-the-art salient region detection methods with a frequency domain analysis, ground truth, and a salient object segmentation application. Our method outperforms the five algorithms both on the ground-truth evaluation and on the segmentation task by achieving both higher precision and better recall.

#### 4.1.2 Contents

##### Related work

Saliency estimation methods can broadly be classified as:

- biologically based
- purely computational
- combination of above two approaches

Itti base their method on the biologically plausible architecture proposed by Koch and Ullman. They determine center-surround contrast using a **Difference of Gaussians** (DoG). Frintrop present a method inspired by Itti's method, but they compute **center-surround differences** with square filters and use integral images to speed up the calculations.

Other methods are purely computational and are not based on biological vision principles. Ma and Zhang and Achanta et al. estimate saliency using center-surround feature

distances. Hu et al. estimate saliency by applying heuristic measures on initial saliency measures obtained by histogram thresholding of feature maps. Gao and Vasconcelos maximize the mutual information between the feature distributions of center and surround regions in an image, while Hou and Zhang rely on frequency domain processing.

The third category of methods are those that incorporate ideas that are partly based on biological models and partly on computational ones. For instance, Harel et al. create feature maps using Itti's method but perform their normalization using a graph based approach. Other methods use a computational approach like maximization of information that represents a biologically plausible model of saliency detection.

### Limitations

The saliency maps generated by most methods have low resolution. Itti's method produces saliency maps that are just  $1/256^{th}$

### Frequency-tuned Saliency Detection

#### DoG

DoG : Difference of Gaussians. DoG filter is widely used in edge detection since it closely and efficiently approximates the Laplacian of Gaussian (LoG) filter, cited as the most satisfactory operator for detecting intensity changes when the standard deviations of the Gaussians are in the ratio  $1 : 1.6$ . The DoG has also been used for interest point detection and saliency detection. The DoG filter is given by :

$$\begin{aligned} DoG(x, y) &= \frac{1}{2\pi} \left[ \frac{1}{\delta_1^2} e^{-\frac{x^2+y^2}{2\delta_1^2}} - \frac{1}{\delta_2^2} e^{-\frac{x^2+y^2}{2\delta_2^2}} \right] \\ &= G(x, y, \delta_1) - G(x, y, \delta_2) \end{aligned} \quad (4.1)$$

where  $\delta_1$  and  $\delta_2$  are the standard deviations of the Gaussian ( $\delta_1 > \delta_2$ ).

A DoG filter is a simple band-pass filter whose passband width is controlled by the ratio  $\delta_1 : \delta_2$ .

#### 4.1.3 Conclusion

# Chapter 5

## Semantic SLAM

### 5.1 DeLS-3D: Deep Localization and Segmentation with a 2D Semantic Map[36]

#### 5.1.1 Abstract

Sensor fusion scheme: Integrates camera videos, Motion sensors (GPS/IMU), and a 3D semantic map.

步骤：

- Initial Coarse camera pose obtained from consumer-grade GPS/IMU
- A label map can be rendered from the 3D semantic map.
- Rendered label map and the RGB image are jointly fed into a pose CNN, yielding a corrected camera pose.
- A multi-layer RNN is further deployed improve the pose accuracy
- Based on pose from RNN, a new label map is rendered
- New label map and the RGB image is fed into a segment CNN which produces per-pixel sematnic label.

从结果可以看出，Scene Parsing 以及姿态估计两者可以相互改善，从而提高系统的鲁棒性以及精确度。

#### 5.1.2 Introduction

在 Localization 中，传统的做法是基于特征匹配来做，但这样的坏处是，如果纹理信息较少，那么系统就不稳定，会出错。一种改进办法是利用深度神经网络提取特征。实际道路中包含大量的相似场景以及重复结构，所以前者实用性较差。

在 Scene Parsing 中，深度神经网络用的很多，最好的基于 (FCN + ResNet) 的途径。在视频中，可以借助光流信息来提高计算速度以及时间连续性。对于静态场景，可以借助 SfM 技术来联合 Parse 以及 Reconstruction. 但这些方法十分耗时。

相机的姿态信息可以帮助 3D 语义地图与 2D 标签地图之间的像素对应。反过来，场景语义又会帮助姿态估计。

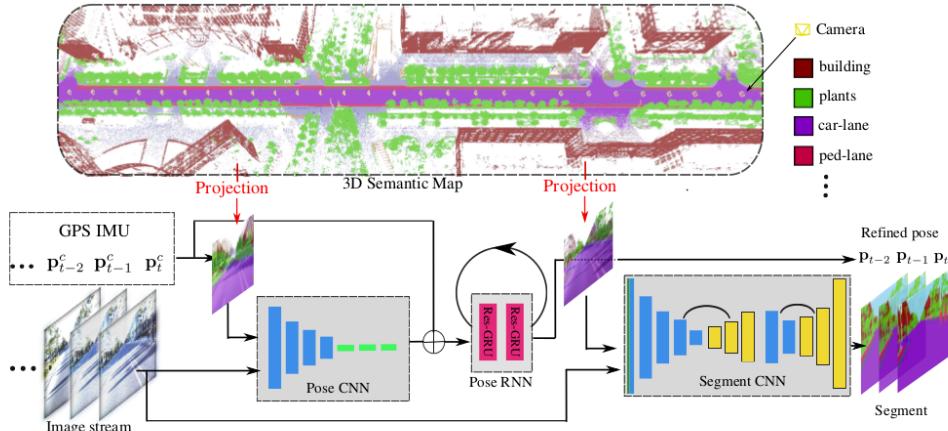


Figure 1: System overview. The black arrows show the testing process, and red arrows indicate the rendering (projection) operation in training and inference. The yellow frustum shows the location of cameras inside the 3D map. The input of our system contains a sequence of images and corresponding GPS/IMU signals. The outputs are the semantically segmented images, each with its refined camera pose.

**Fig 5.1.** DeLS Framework

### 5.1.3 Framework

总的工作流程，如图5.1所示：

从图中可以看出，RGB Images 以及根据 GPS/IMU 获得的 semantic label map 被输入到 Pose CNN，然后输出的 Pose 信息输入到 Pose RNN 来对 Pose 进一步提高，这里用 RNN 来获得前后帧的一致性！然后在利用新得到的 Pose 来获取更精确的 Semantic Label Map，最后，这个 label Map 以及 RGB 图像输入到 Segment CNN 来进一步提高语义地图的精度。这里标签地图被用于提高语义地图的空间精度以及时间一致性。

网络的训练是基于非常精确地相机姿态以及语义分割，所以可以采用监督学习。

### 5.1.4 Related Work

- Camera Pose Estimation

  - PnP

在大的范围内，可能需要提供姿态的先验信息。但对于城市环境中存在大量的 Points，这种方法不适用，且不适用于纹理少、结构重复、以及重叠的区域。

  - Deep learned features

PoseNet, LSTM-PoseNet, Bi-Directional LSTM, or Kalman filter LSTM. 但实际中由于存在植被等重复性的场景，所以十分有必要加入 GPS/IMU 等信息来获得鲁棒的定位结果。而在这里，我们采用结合 RGB 图像与 Online Rendered label map 的方式来提供更好的结果。

**这里问题来了，首先是 label map 的精度如何？其次，随着时间的变化，label map 与实际 RGB 图像可能完全不同，如季节改变了，这应该如何？**

- Scene Parsing

## 5.1 DeLS-3D: Deep Localization and Segmentation with a 2D Semantic Map[36]

FCN, Multi-scale context module with dilated convolution, Pooling, CRF, or Spatial RNN with hundreds of layers. 这些方法都太耗时了。

一些方法是利用小模型或者模型压缩来加速，但会降低精度。

当输入是 Video 时，需要考虑时空信息。当前，存在利用光流来帮助 label 以及 semantic 在相邻帧之间的传递。借助 3D 信息以及相机姿态把相邻帧联系起来，可以更好的处理静态背景下的表示。具体的，是使用 DeMoN 来提高推理效率。

- Joint 2D-3D for video parsing

CNN-SLAM 把传统的 3D 重建模块替换为深度预测网络，且借助语义分割网络来获取场景语义。同样比较耗时、仅适合静态背景，重建效果也不好。

### 5.1.5 Dataset

- Data collection

Mobile LIDAR to collect point clouds of the static 3D map. Cameras' resolution: 2018 \* 2432.

- 2D and 3D Labeling

- Over-segment the point clouds into point clusters based on spatial distances and normal directions, then label each point cluster manually.
- Prune out the points of static background, label the remaining points of the objects.
- After 3D-2D projection, only moving object remain unlabeled.

- 使用图形学中的 *Splatting techniques* 来优化未被标签的像素。

### 5.1.6 Localizing camera and Scene Parsing

#### Render a label map from a camera pose

初始的相机的姿态来自于 GPS/IMU 等传感器。

6-DOF 相机姿态:  $\mathbf{b} = [\mathbf{q}, \mathbf{t}] \in SE(3)$ . 其中  $\mathbf{q} \in SO(3)$  是四元数表示的旋转,  $\mathbf{t} \in \mathbb{R}^3$  表示 Translation。

在由 Point 经 Spalting 获取其面时，面积大小  $s_c$  根据 Point 所属的类别来决定，且与该类别与相机的平均距离的比例有关。

$$s_c \propto \frac{1}{|\mathcal{P}_c|} \sum_{x \in \mathcal{P}_c} \min_{\mathbf{t} \in \tau} d(x, \mathbf{t})$$

其中， $P_c$  是属于类别  $c$  的 3D 点云， $\tau$  是精确地相机姿态。如果面积过大，则会出现 Dilated edges, 而如果面积过小，则会形成 Holes。

## Camera Localization rectification with road prior

### CNN-GRU pose Network Architecture

文中的 Pose Network 包含一个 Pose CNN 以及一个 Pose GRU-RNN。其中 Pose CNN 的输入是 RGB 图像 I 以及一个标签地图 L。输出是一个 7 维的向量，表示输入图像 I 与输入标签地图 L(由较粗糙的姿态  $\mathbf{p}_i^c$  得到)之间的位姿关系，从而得到一个在 3D Map 中更精确的姿态： $\mathbf{P}_i = \mathbf{p}_i^c + \hat{\mathbf{p}}_i$ 。

CNN 结构借鉴 DeMoN，利用打的 Kernel Size 来获取更大的内容，同时保证运行效率，减少参数量。

由于输入的是图像流，为了保证时间一致性，所以在 Pose CNN 之后又加上一个多层的 GRU 网络，且该网络具有 Residual Connection 的连接结构。结果表明，RNN 相比于卡尔曼滤波可以获得更好的运动估计。

### Pose Loss

类似于 PoseNet 的选择，使用 Geometric Matching Loss 来训练。

## Video Parsing with Pose Guidance

上一步得到的 Pose 估计，不是完美的，因为存在 light poles 存在。由于反光，很多点消失了。此外，由于存在动态物体，这些物体可能在原来的标签地图中不存在，所以这些区域可能发生错误。因此，利用额外的一个 Segment CNN 来出来这些问题。且利用标签地图来指导分割过程。

### Segment Network Architecture

首先基于 RGB 图像对该网络训练，然后加入标签地图数据进行微调 (Fine Tune)。具体结构如图 5.2 所示。

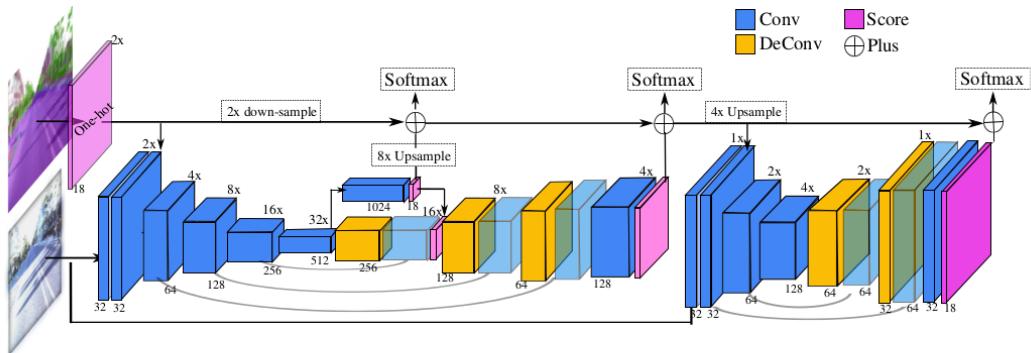


Figure 4: Architecture of the segment CNN with rendered label map as a segmentation priori. At bottom of each convolutional block, we show the filter size, and at top we mark the downsample rates of each block w.r.t. the input image size. The softmax' text box indicates the places a loss is calculated. Details are in Sec. 4.3.

**Fig 5.2. Segment Network in DeLS**

需要注意的是，当标签地图加入框架时，需要经过编码，即每一个像素经 One-hot 操作得到一个 32 维的 Feature Representation。然后得到的 32 维特征加入到 RGB 图像的第一层卷积输出中，且该层的 Kernel 数也是 32 个，从而平衡了两种数据的通道数 (Channel Number)。

### 5.1.7 Experiment

- Adopt OpenGL to efficiently render a label map with the z-buffer handling.

## 5.2 PAD-Net: Multi-Task Guided Prediction-and-Distillation Network for Simultaneous Depth and Scene Parsing [39]

---

- Implement all the networks by adopting the MXNet platform.
- 使用 RNN 可以提高 Pose 的精度，也可以提高 Segment 的精度，尤其对于纤细的物体。

### 5.1.8 Conclusion

基于已有的 3D 语义地图以及视频数据，实现相机的姿态、场景语义任务的实现。算法融合了多种传感器信息。实验表明，相机位姿估计与场景语义两类任务可以相互促进、提高。

## 5.2 PAD-Net: Multi-Task Guided Prediction-and-Distillation Network for Simultaneous Depth and Scene Parsing [39]

### 5.2.1 Abstract

利用同一个网络，完成深度估计与场景解析两个任务。具体来说，通过神经网络学习一系列的中间辅助任务 (Intermediate Auxiliary Tasks)，然后基于中间任务的输出，作为多模式数据 (Multi-modal input) 输入到下一层网络中，完成最终的深度估计以及场景解析两个任务。

其中，一系列的中间任务包括低层任务和高层任务。低层任务包括：Surface Normal, Contour; 高层任务包括：Depth Estimation, Scene Parsing.

### 5.2.2 Analysis

#### Effect of Direct Multi-task Learning

It can be observed that on NYUD-v2, the Front-end + DE + SP slightly outperforms the Front-end + DE, while on Cityscapes, the performance of Front-end + DE + SP is even decreased, which means that using a direct multi-task learning as traditional is probably not an effective means to facilitate each other the performance of different tasks. (DE: Depth Estimation, SP: Scene Parsing)

#### Effect of Multi-modal Distillation

这种 Multi-Modal Distillation 对结果十分有效。且：

By using the attention PAD-Net (Distillation C + SP) guided scheme, the performance of the module C is further improved over the module B.

#### Importance of Intermediate Supervision and Tasks

测试了选择不同的中间任务类型，如：(Multi-Task Deep Network)MTDN + inp2(depth + semantic map), MTDN+3inp3(depth + semantic + surface normal), MTDN + all(depth + semantic + surface + contour).

其中 MTDN + all 比 MTDN + 0-3 都好。

## 5.3 RNN for Learning Dense Depth and Ego-Motion from Video

参考文献: [37]

时间: 2018 年 05 月 19 日

### 5.3.1 Abstract

现在基于学习的单目深度估计, 在 Unseen Dataset 上泛化较差, 可以利用连续两帧之间的特征匹配来解决。本文提出了基于 RNN 的多目视觉深度估计以及运动估计。结果表明, 在远距离上的深度估计, 表现很好。本文方法可使用 both static and deformable scenes with either constant or inconsistent light conditions.

### 5.3.2 Introduction & Related Works

由于 CNN 只能捕获单帧内部的空间特征, 所以即使输入两帧图像, 实际效果也比较差。相关的 SLAM 框架有:

- ORB-SLAM, DSO: 稀疏 SLAM
- LSD-SLAM: semi-dense SLAM, 利用光强来实现跟踪以及构建地图
- DTAM: dense SLAM

上述框架仅适用于静态、光照恒定、足够的 camera motion baseline 的情景。进来, CNN 可被用于 SLAM 中的以下部分:

- Correspondence matching
- Camera Pose estimation
- Stereo

输出包括:

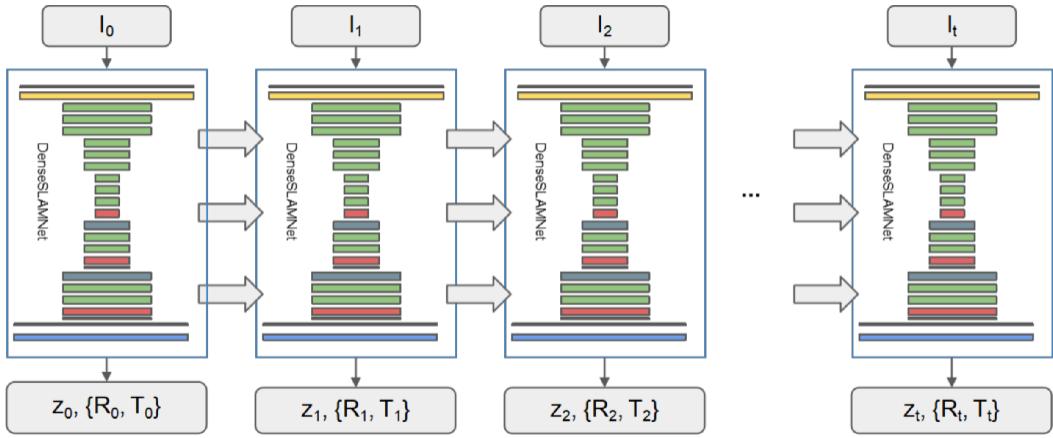
- Depth maps
- point clouds and voxels

CNN 的优势在于可以适用于纹理较少、表面覆盖、很细等场景下, 这些场景对纯使用几何技术来说很难实现。

Newcombe 研究表明, 多目视觉有助于提高深度估计的精度, 而本文作者认为采用相邻几帧可以实现类似的效果。

有作者利用一个非监督生成模型来学习复杂的 3D 到 2D 的投影。但这些手段不太适用于 Deformable Objects.

### 5.3 RNN for Learning Dense Depth and Ego-Motion from Video



**Fig 5.3.** Dense SLAM 框架

#### 5.3.3 Network Architecture

从图5.3可以看出，它接受当前 RGB 图像  $I_t$  以及来自迁移级的隐藏状态  $h_{t-1}$ 。  
 $h_{t-1}$  通过 LSTM 单元进行内部转移。网络的输出是稠密地图  $z_t$  以及相机的姿态  $R_t, T_t$  等。有没有在每一时刻，只输入一帧图像，且输出该帧对应的深度以及姿态，所以相比于 CNN 具有更高的灵活性。

具体的每一块的细节结构如下：

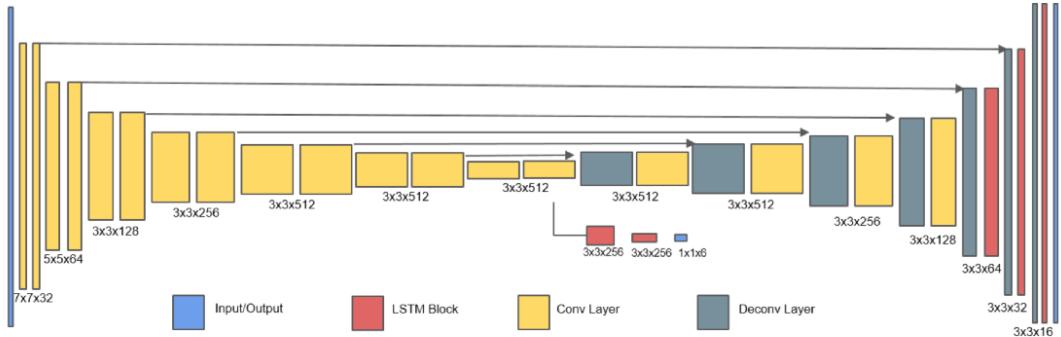


Fig. 4: (Best viewed in color) Our network architecture at a single time step. We use the DispNet architecture. The width and height of each rectangular block indicates the size and the number of the feature map at that layer. Each increase and decrease of size represents a change factor of 2. The first convolutional layer has 32 feature maps. The kernel size for all convolution layers is 3, except for the first two convolution layers, which are 7 and 5, respectively.

**Fig 5.4.** Dense SLAM 中每一级的细节框架

网络结构的另一个参数是时间窗口的大小  $N$ 。本文中  $N = 10$ ，也就是说，图5.4中的结构被重复十次。

### 5.3.4 Training

During training, we feed frames to the network and compute losses from all frames in a temporal window. However, there is no input length constraint at test time.

#### Loss Function

由三部分组成：

- A Point-wise depth loss

$$L_{depth} = \sum_t^N \sum_{i,j} \left\| \xi_t(i,j) - \hat{\xi}_t(i,j) \right\|_{L1}$$

其中， $i, j$  为索引， $t$  为 time step. 使用  $L1 - Norm$  的原因是，它对噪声更鲁棒。

- a camera pose loss

Use the Euler angle  $R$  和平移向量  $T$ 。

$$L_{rot} = \sum_t \|r_t - \hat{r}_t\|_2$$

$$L_{trans} = \sum_t \|t_t - \hat{t}_t\|_2$$

- a scale-invariant gradient loss

为了保证深度图的 Smoothness 和 Sharpness, 所以增加了这个 loss。具体如下：

$$L_{grad} = \sum_t \sum_{h \in \{1,2,4,8,16\}} \sum_{i,j} \|g_{h,t}(i,j) - \hat{g}_{i,j}(i,j)\|_2$$

其中， $h$  代表不同的尺度。 $g_{h,t}$  is a scale-normalized, discretized measurement of the local changes of  $\xi_t$ 。定义如下：

$$g_{h,t} = \left( \frac{\xi_t(i+h,j) - \xi_t(i,j)}{|\xi_t(i+h,j) + \xi_t(i,j)|}, \frac{\xi_t(i,j+h) - \xi_t(i,j)}{|\xi_t(i,j+h) + \xi_t(i,j)|} \right)$$

$L_{grad}$  emphasizes the depth discontinuities, such as occlusion boundaries and sharp edges, as well as the smoothness in homogeneous regions. This property encourages the estimated depth map to preserve more details and reduce noise. Therefore, we put highly weight on this component of the loss. 这一项在所有的 Loss 中比重较大。

最终的 Loss 由上面几项的和构成：

$$L = \alpha * L_{depth} + \beta * L_{pose} + \gamma * L_{grad}$$

其中， $\alpha, \beta, \gamma$  是系数，由经验决定。

注意，下面解释的是为什么使用 Disparity 而不是直接使用深度。

**Caution!** Disparity, the reciprocal of depth (深度的倒数),  $\xi = \frac{1}{z}$  as our direct estimation because it can represent points at infinity and account for the localization uncertainty of points at increasing distance.

Different datasets have different camera intrinsic parameters, so we explicitly crop and resize images to ensure uniform intrinsic parameters. This step assures that the non-linear mapping between color and depth is consistent across all training datasets.

### 5.3.5 Experiments

We evaluate DenseSLAMNet using five error metrics。具体可以参考文章。

- Sc-inv
- Abs-rel, Abs-inv
- RMSE, RMSE-log

### 5.3.6 Ablation Studies

比较了三种不同的网络结构。

1. CNN-SINGLE, 去掉图5.4中 LSTM 单元
2. CNN-STACK, 使用与 CNN-SINGLE 相同的网络, 但是输入 stack of ten images as input.

实验结果表明, LSTM 可有效保留时间域的信息, 从而深度预测的更好。

## Conclusions

引入了 LSTM 单元。Our method effectively utilizes the temporal relationships between neighboring frames through LSTM units, which we show is more effective than simply stack- ing multiple frames together as input.

比几乎所有的单帧 CNN-based 都好。Our DenseSLAMNet outperformed nearly all of the state-of-the-art CNN-based, single-frame depth estimation methods on both indoor and outdoor scenes and showed better generalization ability.

## 5.4 DA-RNN: Semantic Mapping with Data Associated Recurrent Neural Networks

参考文献: [38]

评语看得出来, RNN 在帮助建立相邻帧之间的一致性方面具有很大的优势。

本文利用可以实现 Data Associate 的 RNN 产生 Semantic Label。然后作用于 KinectFusion, 来插入语义信息。

所以本文利用了 RNN 与 KinectFusion 来实现语义地图的构建!

### 5.4.1 Related Works

本文提出的 DA-RU (Data Associated Recurrent Unit) 可以作为一个单独的模块加入到已有的 CNN 结构中！

### 5.4.2 Methods

需要注意的是，本文采用了 DA-RNN 与 KinectFusion 合作的方式来产生最终的 Semantic Mapping，而文章采用的则是 ORB-SLAM 与 CNN 结合，他们有一些共同的结构，如 Data Associate，虽然我现在还不明白这个 DA 具体得怎么实现！？

### 5.4.3 Experiments

实验表明，针对不同的实验输入场景，RGB 图像和 Depth 图像可能发挥不同的作用，有一些时候甚至让结果更差，而本文采用 Contatenated 把来自于 RGB 和 Depth 的 Feature 进行组合起来，在这里，是否可以采取 Attention 或者 [39] 中采用的 Knowledge Distillation 的方式进行多模态数据融合呢！？

此外，实验中也发现，有时候 RGB 图像的 Color 会影响实验结果，所以，是否可以利用一些其它的不那么 Confusing 的数据呢，比如 Shape 等，这也类似于 [39] 中多任务中的 Contour 类似吧！

### 5.4.4 Conclusion

- 将来的可以借助 Optical Flow 来生成图像帧之间的 Data Associate，来代替 KinectFusion 的作用！

## 5.5 SemanticFusion: Dense 3D Semantic Mapping with CNNs

参考文献：[20]

主要框架，用到了 ElasticFusion 和 CNN 来产生稠密的 Semantic Mapping.

### 5.5.1 Introduction & Related Works

本文的算法，利用 SLAM 来实现 2D Frame 与 3D Map 之间的匹配 (Corresponding)。通过这种方式，可以实现把 CNN 的语义预测以概率的方式融合到 Dense semantically annotated map.

为什么选择 ElasticFusion 呢，是因为这种算法是 surfel-based surface representation，可以支持每一帧的语义融合。

比较重要的一点是，通过实验说明，引入 SLAM 甚至可以提高单帧图像的语义分割效果。尤其在存在 wide viewpoint variation，可以帮助解决单目 2D Semantic 中的一些 Ambiguations.

之前存在方法使用 Dense CRF 来获得语义地图。

本文的主要不同是：利用 CNN 来产生语义分割，然后是在线以 Incremental 的方式生成 Semantic Map。即新来一帧就生成一帧的语义地图。

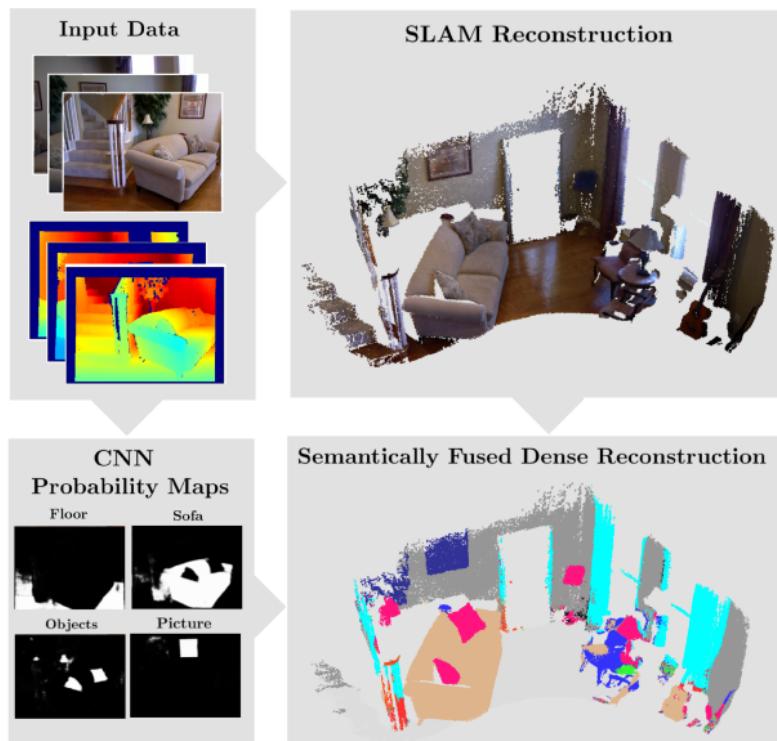
### 5.5.2 Method

系统由三部分组成:

- A real-tiem SLAM system ElasticFusion  
用于提供帧间的关联、全局一致的地图
- CNN  
产生语义分割
- Bayesian update scheme

基于 SLAM 提供的 Pose 关系、CNN 提供的每个像素的 Class Probability 对 Map 中的每一个 surfel 的 Class Probability 进行更新!

此外，作者还尝试了用 CRF 来利用 SLAM 提供的 Geometry 帮助语义分割。  
非常粗糙的流程图如下图所示：



**Fig. 2: An overview of our pipeline:** Input images are used to produce a SLAM map, and a set of probability prediction maps (here only four are shown). These maps are fused into the final dense semantic map via Bayesian updates.

**Fig 5.5.** 粗糙的数据流图

### SLAM Mapping

就是使用 ElasticFusion.

## CNN Architecture

使用 Deconvolutional Semantic Segmentation network，与 FCN 同年提出的另一个分割网络。

在本实验中，输入由 RGB 变为 RGBD，多了一个 Channel。然而，深度相关的训练数据比较少，为了提高利用率，作者利用其它三个输入的光强的平均值对 Depth Filter 进行初始化。

对输入数据，作者还进行了 Scaling，RGB 是用 Bilinear Interpolation，Depth 数据是 Nearest neighbour。

## Incremental Semantic Label Fusion

在生成的地图  $\mathcal{M}$  中，每一个 Surfel(ElasticFusion 的结果) 不仅代表了 Location、Normal 等信息，还包含一个类别 ( $\mathcal{L}$ ) 的分布。

输入数据为  $\mathbf{I}_k$  表示 RGB 图像与 Depth 等。

在对每一个 Surfel 进行类别分布概率更新时，采用的 Recursive Bayesian update，这个算法就是 Probability Robotics 里面最基础的算法，也就是分为 Prediction 和 Correlation 两个步骤。

$$P(l_i|I_{1,\dots,k}) = \frac{1}{Z} P(l_i|I_{1,\dots,k-1}) P(O_{u(s,k)=l_i|I_k})$$

其中，第一个  $P$  表示根据过去的信息的预测，在这里也就是已有的  $\mathcal{M}$  里面的一个 surfel 的 class probability，如果，是一个全新的 Surfel，则初始化它为均匀分布，因此这样熵最大；而第二个  $P$  表示来自 CNN 输入是  $I_k$  时输出的 class probability，然后对已有地图中的 surfel 的 class probability 进行 Correlation!

## Map Regularisation

作者尝试了利用 CRF 来借助 map geometry 对预测的 Semantic surfel 进行 Regularise。

在这里，把每一个 Surfel 当做 CRF 的一个 Node。

这一部分，参考论文吧。

### 5.5.3 Experiments

本文用到的 Semantic 分割的方法 (CNN + SLAM) 与普通的单 CNN 相比，具有很大的优势。本文的方法是，得到 3D 的 Semantic Map 后，重投影到 2D 图像中。

时间方面，SLAM 需要 29.3ms，CNN 的前向传播用了 51.2ms，Bayesian 更新用了 41.1ms。

### 5.5.4 总结

未来也还是 SLAM 与语义分割相互促进吧，达到 Semantic SLAM。

# 5.6 Meaningful Maps with Object-Oriented Semantic Mapping

参考文献: [31]

主要的框架是, 利用 CNN 与 ORB-SLAM2 来实现 Semantic Mapping。但是都用到了 Data Association 的操作! 而且本文还是 Object-oriented (Instance level) !

## 5.6.1 Introduction & Related Works

**重要:**

与已有的方式不同的是, 本文的算法不仅分割独立的 3D Point, 也就是把语义信息投影到 3D Point 中, 而是投影到 3D Structure。这样会更有利于场景理解。

已有的 SLAM 算法得到的结果都是一些几何上的概念, 比如: 点、面、表面等。另一方面, 为了实现与环境交互, 必须基于语义地图才行。

### Semantic Mapping

语义信息与地图构建可能属于两个不同的过程得到。

有一些算法用到了 HMM 或 Dense CRF 等。

### Object Detection and Semantic Segmentation

FCN 的缺点是, 形成的语义地图一般缺少 Notion of independent object instances。如果只是 Pixel-level 的 labels 不能辨别有重叠时物体的身份。

也有一些 Instance-level 的分割算法, 但精度、速度都有待提高。

## 5.6.2 Object Oriented Semantic Mapping

算法的主要步骤:

算法使用 RGB-D 作为输入, 算法是用 ORB-SLAM2 提供相机的姿态、地图等。

### Object Detection for Semantic Mapping

本文使用 SSD 来完成 Object 的 Location 和 Recognition。

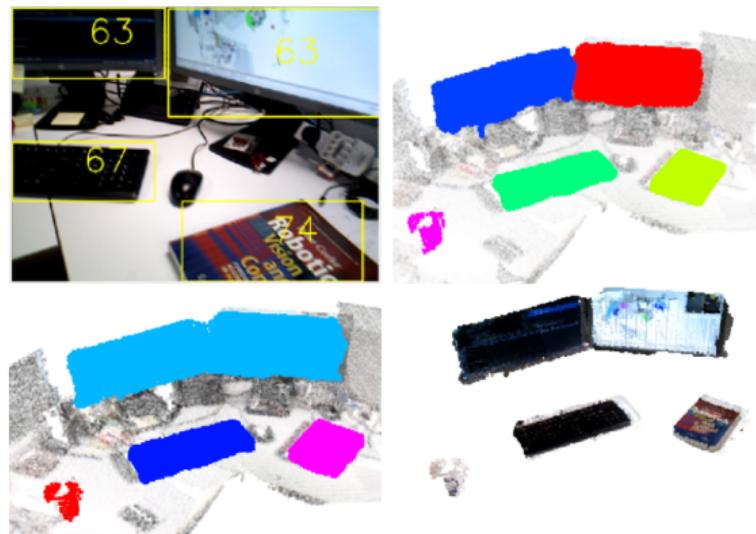
### 3D Segmentation

由于需要非常精确的图像分割, 所以本文利用 Depth 图来帮助分割。所以需要对 Depth 进行分割的算法, 本文采用了文章 [4][23] 等人的算法。也是涉及到基于图的分割的过程。

### Data Association

重点。

当完成了把 3D Point 投影到识别的物体后, 数据关联要做的事: 判断检测到的物体是否已经在已经构建的地图中存在了呢, 如果不存在的话, 就需要新增这个物体。



**Fig. 2:** Illustration of key steps in our proposed approach: (top-left) SSD [23] generates object proposals consisting of bounding boxes, class labels, and confidence scores. (top-right) Our unsupervised 3D segmentation algorithm creates a 3D point cloud segment for each of these objects detected in the current RGB-D frame. (bottom row) We obtain a map that contains semantically meaningful entities: objects that carry a semantic label, confidence, as well as geometric information. The semantic label is color coded in the bottom left image. light blue: monitor, pink: book, red: cup, dark blue: keyboard.

**Fig 5.6.** 算法的几个主要步骤

## 5.6 Meaningful Maps with Object-Oriented Semantic Mapping

通过一个二阶的流程来实现：

- 对于每一个检测到的 Object，根据点云的欧式距离，来选择一系列的 Landmarks(已经检测到并在地图里面已经有的 Object)
- 对 Object 和 Point Cloud of landmark 进行最近邻搜索，使用了 k-d tree 来提高效率，其实这一步也就是判断当前图像检测到的 Object 与已有的地图中的 Landmark(Object) 是否相匹配。

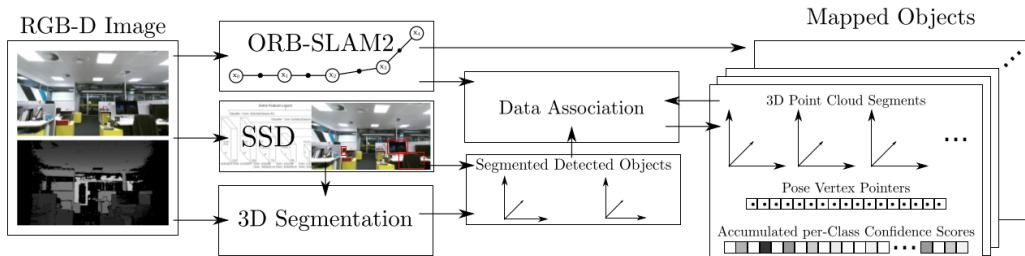
在第二步中，如果多余 50 % 的 Points 的都距离小于 2cm 的话，就说明这个检测到的 Object 已经存在了。

这个也就是把 CNN 的分割结果与地图中的 Object 进行关联起来，并用颜色表示 Map 中 Object 的类别。

### Object Model Update

地图中的目标保存：

- 与目标相关联的分割的 3D 点云
- ORB-SLAM 中因子图的各个位姿的索引
- 有 SSD 提供的各个目标的置信度



**Fig. 3:** Overview of our semantic mapping system. While ORB-SLAM2 performs camera localisation and mapping on every RGB-D frame, SSD [23] detects objects in every RGB keyframe. Our own adapted 3D unsupervised segmentation approach assigns a 3D point cloud segment to every detection. Data association based on an ICP-like matching score decides to either create a new object in the map or associate the detection with an existing one. In the latter case, the 3D model of the map object is extended with the newly detected 3D structure. Every object stores 3D point cloud segments, pointers into the pose graph of ORB-SLAM and per-class confidence scores that are updated on the fly whenever new observations are available.

**Fig 5.7. Semantic Mapping 系统概览**

上图中可以看出，

**评语：**在上一篇 SematicFusion 中，采用的 Recursive Bayesian 的更新规则来完成地图更新的！看来，这个是 CNN 与传统的 SLAM 框架结合的时候的一个需要解决的问题，那就是如何把新来的物体与已有的地图中的物体相互关联起来，并更新！

### Map Generation

通过保存 Keyframe 中物体的 3D point clouds、每一个物体的 3D 分割以及执行位姿图中的一个指针。

### 5.6.3 总结

未来可以的发展方向:

- 语义 Landmark 如何提高 SLAM 的精度, 从而实现 Semantic SLAM
- 把稀疏的语义地图改进为稠密地图

## 5.7 LiteFlowNet: A Lightweight Convolutional Neural Network for Optical Flow Estimation

参考文献: [12], CVPR 2018

### 5.7.1 背景知识

FlowNet2 是基于 CNN 进行光流估计的 SOTA 算法 (?), 需要 160M 的参数量。LiteFlowNet 实现了 30 倍的轻量化, 并且比 FlowNet2 块 1.36 倍。

主要的贡献:

- 在每一层金字塔 (Pyramid Level) 预测光流更高效的轻量级联网络。通过早前矫正 (Early Correction) 提高了精度, 同时无缝支持网络中的描述子匹配。
- 提出了一个新型的光流正则化层, 可以改善野值点、光流边界模糊等问题, 是基于特征驱动的区域卷积实现
- 网络结构可以高效提取 Pyramidal Feature 以及 Feature Warping, 而不是像 FlowNet2 中的 Image Warping。

FlowNet2 通过级联 FlowNet, 来不断调优光流场, by contributing on the flow increment between the first image and the warped second image?

后来, SPyNet 通过在每一 Pyramid level 采用 Image Warping 实现了只有 1.2M 大小的网络, 但精度只有 FlowNet 大小, 而达不到 FlowNet2 的水平。

**Image Warping:** 图像扭转, 是一种数字图像处理过程, 在任何图像中所描述的任何形状都会产生显著有损。扭曲可用于矫正图像有损, 同时可用于某种创意目的。纯粹的图像扭曲意味着点到点的映射, 而不改变其颜色。

提高 FlowNet2 以及 SPyNet 的两种准则 (Principles):

- Pyramid feature extraction

Consists of an encoder and a decoder. Encoder 把输入的图像对分别映射到多尺度高维特征空间中; Decoder 以 Coarse-to-fine 的框架估计光流场。这比 FlowNet2 采用 U-Net 更轻量化。相比于 SPyNet, 我们的模型把特征提取与光流估计两个过程分离, 可以更好的处理精度与复杂度之间的矛盾。

- Feature Warping

FlowNet2 与 SPyNet 将输入图像对中的第二幅图像基于先前估计的光流进行 Warp, 然后使用 Warped 的第二幅图像与第一幅图像的特征图谱 Refine 估计的光流。

所以这个过程中，首先把第二幅图像进行 Warp，然后提取 Warp 图像的特征，这个过程十分繁琐。所以，本文提出直接对 Feature Map 进行 warp。保证模型更精确以及高效。

更详细的细节一定要看原文 [12]。

此外，除了上面两个主要改进的 Principle，作者还提出第三个比较重要的改进措施，那就是 Flow Regularization.

- Flow Regularization

级联的光流估计类似于能量最小化方法中的保真度 (Data Fidelity) 的作用。为了消除边界的模糊以及野值点，Regularize flow Field 的常用 Cues:

- Local flow consistency
- Co-occurrence between flow boundaries
- Co-occurrence between intensity edges

对应的代表方法包括：

- Anisotropic image-driven
- image- and flow-driven
- Complementary regularizations

在本文中，提出的是 Feature-driven local convolution layer at each pyramid level. 该方法对 Flow- 以及 Image- 敏感。

**评语：**看起来，一个是提高精度，一个是提高效率，这是两个最终的目的。提高精度可以通过设计网络结构以及增加其它考虑来实现；提高效率的一个重要表现是降低模型的参数数量，提高运算速度。不过，本文虽然参数数量减少了 30 倍，但速度却只提高了 1.36 倍，这里面的原因是什么？

### 5.7.2 Related Works

#### Variational Methods

Address illumination changes by combining the brightness and gradient constancy assumptions.

DeepFlow, propose to correlate multi-scale patches and incorporate this as the matching term in functional.

PatchMatch Filter, EpicFlow.

本文提出的网络结构，是受 Variational methods 中的 Data Fidelity 以及正则化启发。

#### Machine Learning Methods

PCA-Flow. 这些参数化模型可以通过 CNN 高效的实现。

### CNN-based Methods

FlowNet, 使用能量最小化作为后处理步骤, 来降低在光流边界的平滑效应。不能端到端训练?

FlowNet2, 通过 FlowNet 的级联实现。虽然提高了精度, 但模型更大, 计算更复杂。

SPyNet, 受 Spatial Pyramid 启发, 模型更紧凑, 但效果远不如 FlowNet2。

InterpoNet, 借助第三方系数光流但需要 off-the-shelf 的边缘检测。

DeepFlow, 使用 Correlation 而不是真正意义的 CNN, 参数不能训练。

### Establish Point Correspondence

Establishing point correspondence, 一种方式是 Match Image Patches

CNN-Feature Matching, 首先被 Zagoruyko 等人提出 (Learning to compare image patches via CNN, 2015 CVPR)。

MRF-based, Güney 等人提出利用 Feature Representation 以及利用 MRF 来估计光流。

Bailer 使用多尺度 Feature, 然后以类似于 Flow Fields 的方式进行特征匹配。

Fischer 和 Ilg 等人为了提高计算效率, 仅在稀疏空间维度进行特征匹配。

本文中, We reduce the computational burden of feature matching by using a short-ranged matching of warped CNN features at sampled positions and a sub-pixel refinement at every pyramid level.

类似于 Spatial Transformer, 本文利用 f-warp layer 来区分不同个 Channel. 本文的决策网络是一个更普用的 Warping Network, 可以用来 Warp 高层次的 CNN Features, 而不仅仅是对 Image 进行 Warpping.

### 5.7.3 LiteFlowNet

整体结构如下:

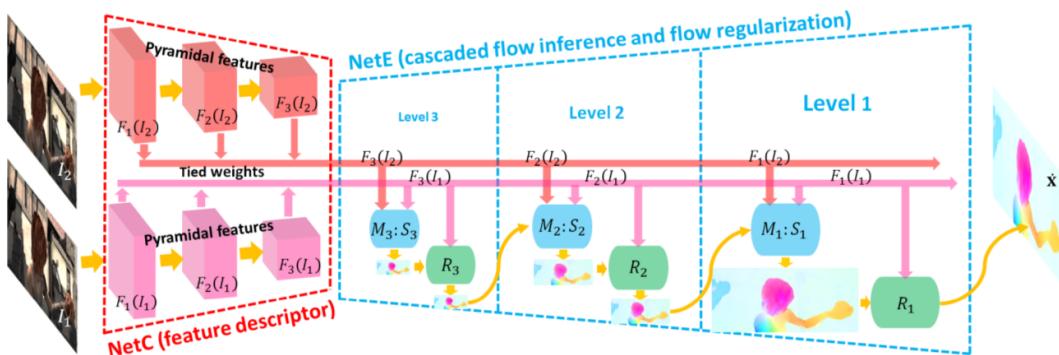


Figure 2: The network structure of LiteFlowNet. For the ease of representation, only a 3-level design is shown. Given an image pair ( $I_1$  and  $I_2$ ), NetC generates two pyramids of high-level features ( $\mathcal{F}_k(I_1)$  in pink and  $\mathcal{F}_k(I_2)$  in red,  $k \in [1, 3]$ ). NetE yields multi-scale flow fields that each of them is generated by a cascaded flow inference module  $M:S$  (in blue color, including a descriptor matching unit  $M$  and a sub-pixel refinement unit  $S$ ) and a regularization module  $R$  (in green color). Flow inference and regularization modules correspond to data fidelity and regularization terms in conventional energy minimization methods respectively.

**Fig 5.8.** LiteFlowNet 结构框图

### Pyramid Feature Extraction

进行 stride- $s$  的卷积操作，得到的 Feature Map 表示为： $\mathcal{F}_k(I_i)$ ，即第  $i$  个图像的第  $k$  层 Feature Map。简化写成  $\mathcal{F}_i$ 。

### Feature Warping (f-warp)

假设  $\dot{x}$  为预测的光流，则 Feature Warping 是指：

$$\tilde{\mathcal{F}}_2(x) \triangleq \mathcal{F}_2(x + \dot{x}) \sim \mathcal{F}_1(x)$$

注意，Warping 是作用于  $\mathcal{F}$  上的，而不是输入图像上的。

为了使上述过程可以支持 end-to-end 训练，这里采用 Bilinear Interpolation 进行插值的技术实现 Warping。Bilinear Interpolation 是支持后向传播训练的！修改后 Warping 实现公式如下：

$$\tilde{\mathcal{F}} = \sum_{x_s^i \in N(x_s)} \mathcal{F}(x_s^i)(1 - |x_s - x_s^i|)(1 - |y_s - y_s^i|)$$

其中， $x_s = x + \dot{x} = (x_s, y_s)^T$  是输入的源 Feature Map 中的坐标。 $x$  denotes the target coordinates of the regular grid in the interpolated feature map  $\mathcal{F}$ ， $N(x)$  代表 the four pixel neighbors of  $x_s$ 。

自己的理解：首先  $x$  是插值后的图像索引，而  $x_s = x + \dot{x}$  是插值前 Feature Map 中对应 Object Feature 的  $x$  索引处的索引。 $x_s^i$  是在  $x_s$  周围的四个紧邻像素。也就是个 Bilinear Interpolated。

### Cascaded Flow Interface

**评语：**这一块看的比较吃力。为什么会吃力？因为新的概念么？那么作者为什么提出这么麻烦的概念呢？实际效果又怎么样呢？

通过计算高层特征向量的 Correlation 来实现输入图像的点对应。

$$c(x, d) = \mathcal{F}_1(x) \cdot \mathcal{F}_2(x + d) / N$$

这个公式跟 FlowNet 中的计算方式区别不大。只不过这里的  $N$  是指 Feature 的长度。

**M Module** 在 Descriptor Matching unit M, Residual flow  $\Delta\dot{x}_m$ . A complete flow field  $\dot{x}_m$  is computed as follows, 这句话没懂

$$\dot{x}_m = \underbrace{M(C(\mathcal{F}_1, \tilde{\mathcal{F}}_2; d))}_{\Delta\dot{x}_m} + s\dot{x}^{\uparrow s}$$

其中， $\dot{x}$  是上一层的最开始的光流估计！

**S Module** 为了进一步提高 flow estimate  $\dot{x}_m$  的精度，即达到亚像素级别。作者引入了 Second flow inference。这可以防止错误光流的放大并传到下一级。Sub-pixel refinement unit S，会产生一个更准确的光流场，这通过最小化  $\mathcal{F}_1$  和  $\tilde{\mathcal{F}}_2$  之间的距离来实现。

$$\dot{x}_s = \underbrace{S(C(\mathcal{F}_1, \tilde{\mathcal{F}}_2, \dot{x}_m))}_{\Delta\dot{x}_s} + \dot{x}_m$$

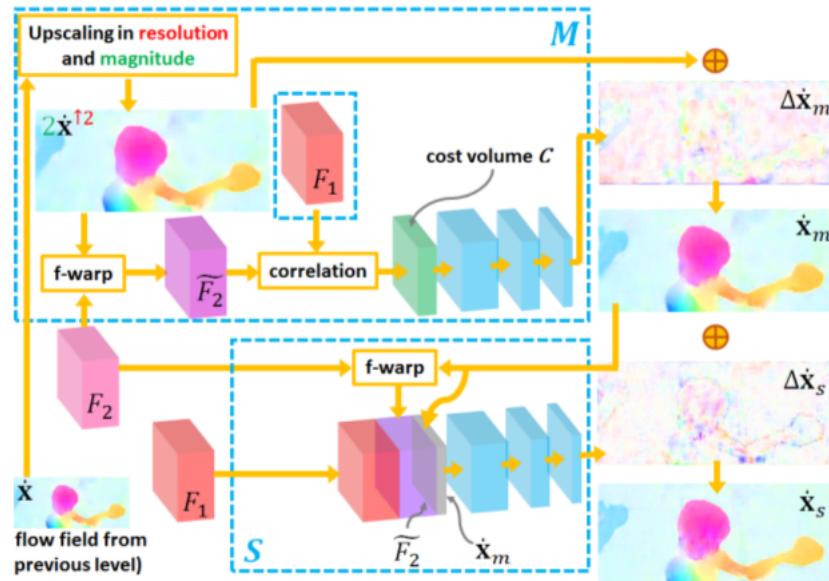


Figure 3: A cascaded flow inference module  $M:S$  in NetE. It consists of a descriptor matching unit  $M$  and a sub-pixel refinement unit  $S$ . In  $M$ , f-warp transforms high-level feature  $\mathcal{F}_2$  to  $\widetilde{\mathcal{F}}_2$  via upsampled flow field  $2\dot{x}^{t2}$  estimated at previous pyramid level. In  $S$ ,  $\mathcal{F}_2$  is warped by  $\dot{x}_m$  from  $M$ . In comparison to residual flow  $\Delta\dot{x}_m$ , more flow adjustment exists at flow boundaries in  $\Delta\dot{x}_s$ .

**Fig 5.9.** 在 NetE 中的级联光流推理模块,M:S

所以总的来说 M 模块是为了计算  $\Delta\dot{x}_m$ , 而 S 模块是为了计算  $\Delta\dot{x}_s$ 。对最开始  $\dot{x}$  估计的光流进行两次的 Refinement.

### Flow Regularization

这一部分主要消除光流边界的模糊、存在的 artifacts 等。提出用 feature-driven local convolution (f-lcon)

假设 Feature Map (F) 的尺寸为:  $M * N * c$ , 定义 f-lcon 的滤波器为  $\mathbf{G} = g$

对于输入为  $\dot{x}_s$ , Flow Regularization 值的是:

$$\dot{x}_r = R(\dot{x}_s; G)$$

输出的是正则化后的光流估计  $\dot{x}_r$

下面的关键是如何生成这个用于正则化的卷积核。为此, 作者定义了一个 feature-driven 的距离度量  $\mathcal{D}$ , 总的来说, 该度量由一个 CNN unit  $R_D$  来计算:

$$\mathcal{D} = R_D(\mathcal{F}_1, \dot{x}_s, O)$$

基于这个度量, 可以计算得到卷积核:

$$g(x, y, c) = \frac{\exp(-\mathcal{D}(x, y, c)^2)}{\sum_{(x_i, y_i) \in N(x, y)} \exp(-\mathcal{D}(x_i, y_i, c)^2)}$$

其中  $N(x)$  表示一个  $\omega * \omega$  的近邻。

### 5.7.4 Ablation Study

算法的结果是优于 FlowNet2, SPyNet 等。

### Feature Warping

没有 Warping, 光流更 Vague. 通过计算 residual flow (M:S 两个模块的功能) 可以提高估计效果。

M: Matching

S: Sub-pixel refinement

R: Regularization units in NetE

### Descriptor Matching

### Sub-Pixel Refinement

The flow field generated from WMS is more crisp and contains more fine details than that generated from WM with sub-pixel refinement disabled.

更小的 flow artifacts.

### 5.7.5 Regularization

In comparison WMS with regularization disabled to ALL, undesired artifacts exist in homogeneous regions

Flow bleeding and vague flow boundaries are observed.

表明, Feature-driven local convolution 对于光滑光流场、保持 crisp flow boundaries 非常重要!

### 5.7.6 Conclusion

Pyramidal feature extraction and feature warping (f-warp) help us to break the de facto rule of accurate flow network requiring large model size. To address large-displacement and detail-preserving flows, LiteFlowNet exploits short-range matching to generate pixel-level flow field and further improves the estimate to sub-pixel accuracy in the cascaded flow inference. To result crisp flow boundaries, LiteFlowNet regularizes flow field through feature-driven local convolution (f-lcon). With its lightweight, accurate, and fast flow computation, we expect that LiteFlowNet can be deployed to many applications such as motion segmentation, action recognition, SLAM, 3D reconstruction and more.

## 5.8 小结

2018.05.23 小结

在生成 Semantic Map 的时候, 看样子现在的趋势, 是利用 RNN 保证时间一致性; 利用多模态数据提高精度, 但如何融合多模态数据的 Feature 有待研究, 现有的有一些是直接 Concatenate、Knowledge Distillation、Attntion(?) 等机制。

## 5.9 ExFuse: Enhancing Feature Fusion for Semantic Segmentation

参考文献: [ExFuse 简介 -知乎](#)

### 5.9.1 要解决的问题

在语义分割领域中, 经常需要融合多层的 Feature。然而, 底层的 Feature 含有的语义信息较少, 但分辨率较高, 噪声也比较少, 这是由于卷积层比较浅; 而高层的 Feature 语义信息多, 但空间分辨率很小。

所以本文就提出了: 1) 增加底层特征的语义; 2) 在高层中增加空间信息

### 5.9.2 Method

使用了 ResNet、GCN(Global Convolution Net) 的思想。

其中, SS, SEB, ECRE, DAP 是文章作者提出的算法。

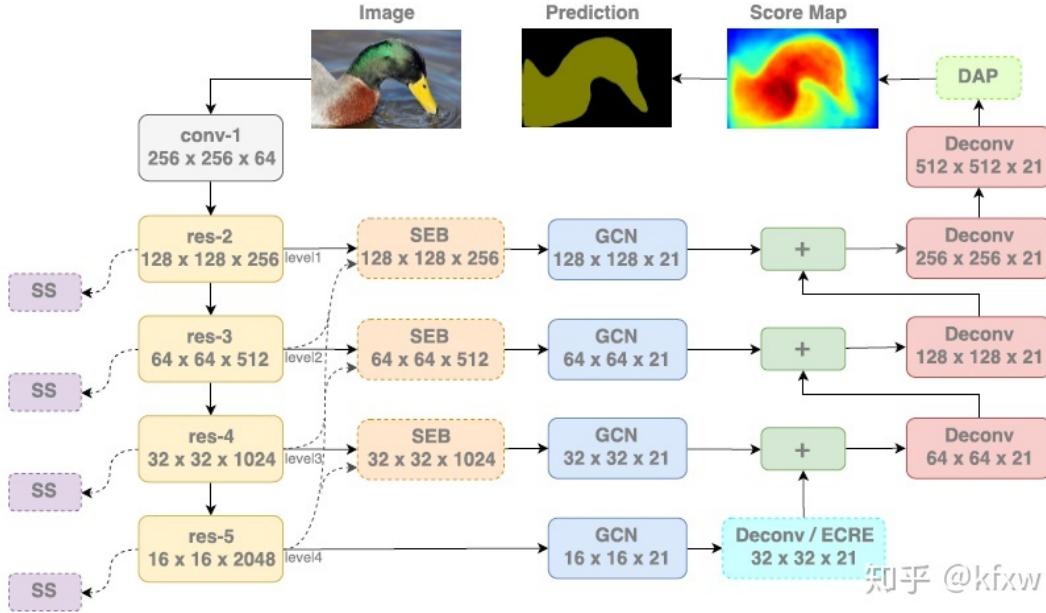


Fig 5.10. ExFusion 的实现框图

### 在底层中加入更多的语义信息

具体是三个方面的改进：

- Layer Rearrangement

ResNeXt 网络结构中，各级的网络包含的残差单元个数为 3,4,23,3。为了提高底层特征的语义性，一个想法便是让低层的两级网络拥有的层数更多。因此作者将残差单元个数重排为 8,8,9,8，并重新在 ImageNet 上预训练模型。重排后网络的分类性能没有明显变化，但是分割模型可以提高约 0.8 个点 (mean intersection over union) 的性能。

- Semantic Supervision(SS)

深度语义监督其实在其他的一些工作里 (如 GoogLeNet, 边缘检测的 HED 等等) 已经使用到了。这里的使用方法基本上没有太大变化，能够带来大约 1 个点的提升。

- Semantic Embedding branch(SEB)

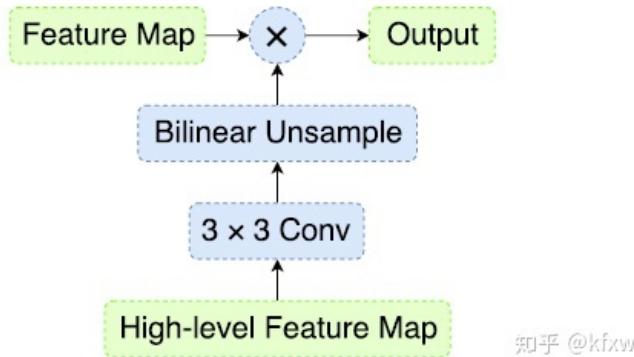
其做法是将高层特征上采样后，与低层特征逐像素相乘，用在 GCN 之前。该部分能带来大约 0.7 个点的提升。

### 在高层中加入更多的空间信息

用两种方法来把更多的空间信息融入到高层特征中：

- 通道分辨率嵌入 (Explicit Channel resolution embedding, ECRE)

其思路是在上采样支路中使用 [2,3,4] 工作中都使用到的子像素上采样模块 (sub-pixel upsample)。作者的出发点并不是前人工作中强调的如速度快、消除反卷积的棋盘效应等等，而是通过这个结构能够让和空间信息相关的监督



**Fig 5.11.** 语义嵌入分支的结构图

信息回传到各个通道中，从而让不同通道包含不同空间信息。该模块和原有的反卷积一起使用才能显示出更好的性能。同单独使用反卷积相比，性能可以提高约 0.6 个点。

- 稠密邻域预测 (Densely adjacent prediction, DAP)

DAP 模块只使用在输出预测结果的时候。其想法也是通过扩展通道数来增加空间信息。举一个例子来描述其功能，假设 DAP 的作用区域为  $3 \times 3$ ，输出结果的通道数为 21，则扩展后的输出通道数为  $21 \times 3 \times 3$ 。每  $3 \times 3$  个通道融合成一个通道。如在最终结果中，第 5 通道（共 21 通道）的 (12,13) 坐标上的像素，是通过 DAP 之前的第 5+0 通道 (11,12)、5+1 通道的 (11,13)、5+2 通道的 (11,14)、5+3 通道的 (12,12)、5+4 通道的 (12,13)、5+5 通道的 (12,14)…平均得到的。DAP 能带来约 0.6 个点的提升。

## 5.10 Multi-View Deep Learning for Consistent Semantic Mapping with RGB-D Cameras

时间：2018.05.24

主要的贡献是：以自监督的方式预测多视角一致的语义分割，在与关键帧的语义地图融合时，更加一致。

### 5.10.1 Introduction & Related Works

现阶段，RGB-D 数据，Appearance 以及 Shape Modalities 对于语义分割都会有帮助，可以结合起来。但结合多视觉信息的做法，还没有人尝试。

那么如何把多视觉 Frames 与 SLAM 结合起来呢，作者的做法是，根据 SLAM 提供的 Pose，其中一个帧为目标帧，目标帧存在 Ground Truth 标签数据，然后，同一时刻其它视角的 Frame 可以通过 SLAM 计算得到的位姿被 Warp 到目标帧中。这样，网络可以学习视角不变的特性，且不需要额外的标签数据。其它视角的输出被 Warp 到目标帧 (Reference) 中，然后融合，融合过程是以概率的形式完成的。

## 5.10 Multi-View Deep Learning for Consistent Semantic Mapping with RGB-D Cameras

### Image based Semantic Segmentation

有一些是单独利用 RGB 输入，有一些输入是 RGB-D 数据。

前者，主要的思想有 FCN、Encoder-decoder 结构通过可学习的 Unpooling, Deconvlution 等完成语义分割。

后者，有一些基于 encoder-decoder 的可以融合 RGB 数据与 Depth 数据，也可以吧 Depth 数据转换成 HHA，其它的还有 Mult-scale Refinement, dilated Convolutions 以及 Residual units，此外还有 LSTM 来融合 RGB 与 Depth 数据，可生成更平滑的结果。最后，还有一些用到了 CRF 的算法。

**评语：**如果输入数据是 RGB-D 数据的话，那么就不用像 [39] 那样需要额外的单独预测 Depth 的网络结构了，而直接使用 Attention 或 Knowledge Distillation 等机制处理 Depth 与 RGB 数据不就可以了么！？

### Semantic SLAM

SLAM++ 可以实现 Object Instance level 的跟踪、地图构建等。

其它的一些算法包括概率的形式融合语义信息。还有是处理点云的算法。

### Multi-View Semantic Segmentation

CNN 用于多目 3D 的语义分割研究很少。

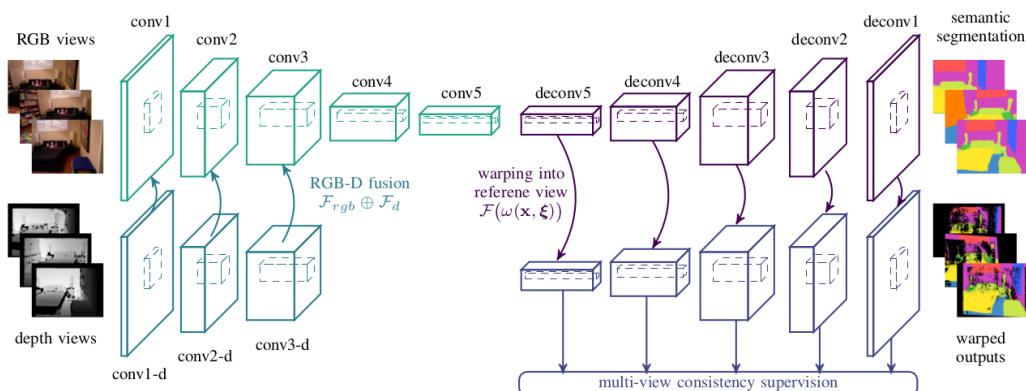
存在用 3D CNN 来处理系数的 Octree 数据结构，得到对 voxel 的语义分割。

有算法提出使用视频中的 Superpixel 和 Optical Flow 来融合视频中的多目视觉的语义信息。

跟本文相近的一个算法是利用多目来实现 Shape Recognition 中。

### 5.10.2 CNN Architecture For Semantic Segmentation

本文在 FuseNet 上的改进是，加入了 Multi-scale 的 Loss Function Minimization。



**Fig. 2:** The CNN encoder-decoder architecture used in our approach. Input to the network are RGB-D sequences with corresponding poses from SLAM trajectory. The encoder contains two branches to learn features from RGB-D data as inspired by FuseNet [1]. The obtained low-resolution high-dimension feature maps are successively refined through deconvolutions in the decoder. We warp feature maps into a common reference view and enforce multi-view consistency with various constraints. The network is trained in a deeply-supervised manner where loss is computed at all scales of the decoder.

**Fig 5.12.** 基于 Encoder-Decoder CNN 的语义分割示意图

需要注意以下几点：

- 使用 Unpooling、Deconvolution 等实现 Decoder
- 注意，RGB 与 Depth 经过两个分离的分支提取特征，然后在多个尺度 (Multi-scale!) 进行融合！
- 使用 KL 散度作为 Loss Function：

$$L(\mathcal{W}) = -\frac{1}{N} \sum_i^N \sum_j^K [j = l_{gt}] \log p_j(x_i, \mathcal{W}|\mathcal{I})$$

- 使用 Deeply Supervised Learning method 计算 all Upsample Scales，注意，Ground Truth 使用 **Stochastic Pooling** 这种正则化技术生成各个尺度下对应的 Ground Truth

### 5.10.3 Multi-View Consistent Learning and Prediction

本文的主要贡献是，Explore the use of temporal multi-view consistency within RGB-D sequences for CNN training and prediction. 这是通过把多帧数据 Warp 到一个 Common Reference View 来实现。

#### Multi-view Data Associate Through Warping

为了保证时间一致性，引入了 Warping Layers。

该 Layer 的操作原理是：

$$X^\omega = w(x, \xi) = \pi(T(\xi)\pi^{-1}(x, Z_i(x)))$$

其中， $\omega$  为 Warping function， $T(\xi)$  为 SLAM 得到的位姿  $\xi$  下的单应变换； $\pi$  实现 3D 坐标与图像坐标之间的转换。 $Z_i(x)$  为图像  $x$  处的深度信息。

#### Consistency Through Warp Augmentation

把 Keyframe Warp 到 Neighboring Frames。

#### Consistency Through Bayesian Fusion

$$\begin{aligned} p(y|z^i) &= \frac{p(z_i|y, z^{i-1})p(y|z^{i-1})}{z_i|z^{i-1}} \\ &= \beta_i p(z_i|y, z^{i-1})p(y|z^{i-1}) \end{aligned}$$

其中， $z^i$  为到达第  $i$  帧的所有 Measurements。

上式也就是 Recursive Bayesian Update。

#### Consistency Through Multi-View Max-Pooling

Bayesian Fusion 是在 Probability 域内进行的，而这里作者还尝试了直接在 Feature 域内进行 Fuse。

使用 Multi-View max-pooling 对 Warp 的 Feature Map 进行处理。

### 5.10.4 Experiments

使用 NYUDv2 数据集、DVO-SLAM 框架。  
使用 Cubic Interpolation 来对 RGB 图像进行降采样，利用 Nearest-neighbor Interpolation 来对 Depth 和 Labelling 数据进行降采样  
使用 SGD Moment 训练，动量为 0.9，同时有 0.0005 的 Weight Decay。Learning Rate 被设置为 0.001 然后每 30K 次迭代就 Decay by a factor of 0.9。  
使用 IOU 来判断 Segment 的效果。

### 5.10.5 总结

We base our CNN design on FuseNet, a recently proposed CNN architecture in an encoder-decoder scheme for semantic segmentation of RGB-D images.

All multi-view consistency training approaches outperform single-view trained baselines. They are key to boosting segmentation performance when fusing network predictions from multiple view points during testing.

In future work, we want to further investigate integration of our approach in a semantic SLAM system, for example, through coupling of pose tracking and SLAM with our semantic predictions.

所以，作者的意思是说现在还没有达到 Semantic SLAM 的水平，而实现的也就是只有 Semantic Segmentation 吗？SLAM 只是提供了必要的位姿估计，来帮助相邻帧之间的 Warping？

## 5.11 MaskFusion: Real-Time Recognition, Tracking, and Reconstruction of Multiple Moving Objects

MaskFusion, 看样子挺厉害的样子。  
A real-time, object-aware, semantic And dynamic RGB-D SLAM.(为什么这几篇基于 RGB-D 数据的算法这么多呢！？所以，要发文章的话，可以选面向不同输入数据的一种方法！比如，双目的文章我还没有看到，虽然上篇是关于 Multi-View 的语义分割，但不涉及 SLAM 啊。看样子生成语义地图的结果应该会挺炫的。)

可以实现更有优势的 Instance-level 的 Semantic Segmentation。利用 VR 应用来表明 MaskFusion 的独特优势：Instance-aware, Semantic and Dynamic。

### 5.11.1 背景及相关工作

把 SLAM 应用到 AR(Augmented Reality) 时，还有两个问题解决的不太理想：

- 现在的 SLAM 系统均假设环境是静态的，会主动忽略 Moving Objects
- 现在的 SLAM 系统得到的均是一些几何的地图，缺少语义信息。有一些尝试，如 SLAM++, CNN-SLAM

本文提出的 MaskFusion 算法可以解决这两个问题，首先，可以从 Object-level 理解环境，在准确分割运动目标的同时，可以识别、检测、跟踪以及重建目标。

分割算法由两部分组成：

- Mask RCNN  
提供多达 80 类的目标识别等。
- A geometry-based segmentation algorithm  
利用 Depth 以及 Surface Normal 等信息向 Mask RCNN 提供更精确的目标边缘分割。

上述算法的结果输入到本文的 Dynamic SLAM 框架中。

使用 Instance-aware semantic segmentation 比使用 pixel-level semantic segmentation 更好。目标 Mask 更精确，并且可以把不同的 object instance 分配到同一 object category。

本文的作者又提到了现在 SLAM 所面临的另一个大问题：Dynamic 的问题。作者提到，本文提出的算法在两个方面具有优势：

- 传统的 Semantic SLAM 尝试  
相比于这些算法，本文的算法可以解决 Dynamic Scene 的问题。
- 解决 Dynamic SLAM 的尝试  
本文提出的算法具有 Object-level Semantic 的能力。

**评语：**所以总的来说，作者就是与那些 Semantic Mapping 的方法比 Dynamic Scene 的处理能力，与那些 Dynamic Scene SLAM 的方法比 Semantic 能力，在或者就是比速度。确实，前面的作者都只关注 Static Scene，现在看来，实际的 SLAM 中还需要解决 Dynamic Scene(Moving Objects 存在) 的问题。

## Dense RGB-D SLAM

KinectFusion 证明使用 Truncated Signed Distance Function(TSDF) 可以表示室内地图构建，此外，其它的工作也证明使用合适的数据结构也可以完成室外环境的地图构建。

Surfels: Surface Elements, 在 Computer Graph 中应用比较早。

Surfel 类似于 Point Cloud，不同的是，Surfel 不仅包含 Location 信息，还包含 Radius 和 Normal 等，此外，不像 Point Cloud 下在切换 Mapping 与 Tracking 之间的 Overhead. 且是高效存储的。

## Scene Segmentation & Semantic Scene Segmentation

单纯的 Scene Segmentation 可以提供精确的目标边界，但缺乏 Semantic Information.

Semantic Scene Segmentation 作者只提到了基于 MRFs 的方法。

## Semantic SLAM & Dynamic SLAM

Semantic SLAM 中提到了 CNN-SLAM, Semantic Fusion

**评语：**但作者的意思是这些算法没有考虑 Object instance level 的语义分割，而只考虑点云层面的分割，如果真是这样的话，那么不就没多少意义了，当然对人来说可能很清楚，但对机器而言，必须考虑 Object Instance level. 好像，这两个算法中确实是这样。

## 5.11 MaskFusion

---

存在两种方法可以处理 Dynamic SLAM 的问题，而不把 Moving Object 当做 Outlier 来处理：

- Deformable world is assumed and as-rigid-as-possible registration is performed
- Object instances are identified and potentially tracked rigidly

上面两种方法都涉及要么 Template- 要么是 descriptor-based methods

### 5.11.2 System Design

#### System Overview

每新来一帧数据，整个算法包括以下几个流程：

##### 1. Tracking

每一个 Object 的 6 DoF 通过最小化一个能量函数来确定，这个能量函数由两部分组成：几何的 ICP Error, Photometric cost?。此外，作者仅对那些 Non-static Model 进行 Track。最后，作者比较了两种确定 Object 是否运动的方法：

- Based on Motion Inconsistency
- Treating objects which are being touched by a person as dynamic

##### 2. Segmentation

使用了 Mask RCNN 和一个基于 Depth Discontinuities and surface normals 的分割算法。前者有两个缺点：物体边界不精确、运行不实时。后者可以弥补这两个缺点，但可能会 Oversegment objects。

##### 3. Fusion

就是把 Object 的几何结构与 labels 结合起来。

#### Multi-Object SLAM

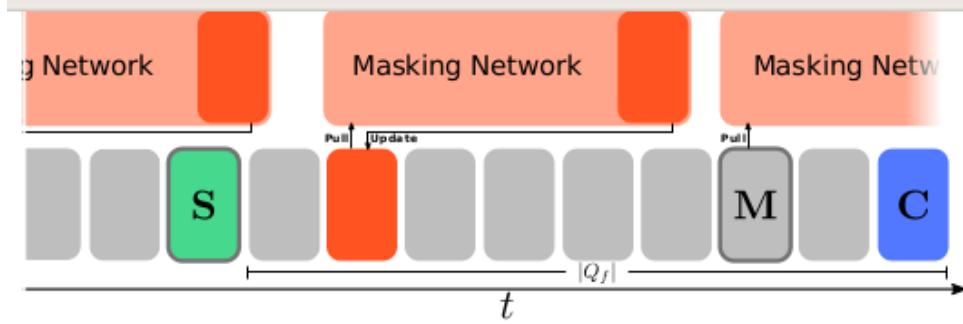
本文采用 ElasticFusion 类似的 3D Model. 其中，Model 用 Surfel 来表示，Surfel 具体包含以下内容： $\mathcal{M}_m^s \in \{\mathbf{p} \in \mathbb{R}^3, \mathbf{n} \in \mathbb{R}^3, \mathbf{c} \in \mathbb{N}^3, \mathbf{w} \in \mathbb{R}, \mathbf{r} \in \mathbb{R}, \mathbf{t} \in \mathbb{R}^2\}, \forall s < |\mathcal{M}_m|$ ，分别表示：Position, normal, color, weight, radius, and two timestamps. 此外，系统还包含每一个目标的 Label  $l_m = m \forall m \in 0 \dots N$ ,  $N$  是场景中包含的 Object 的数量。

这一步 SLAM 主要解决的问题是，对 Object 的 Pose 的跟踪，由于输入时 RGB-D 数据，且像前面说的那样，作者的目标函数包含两个部分：

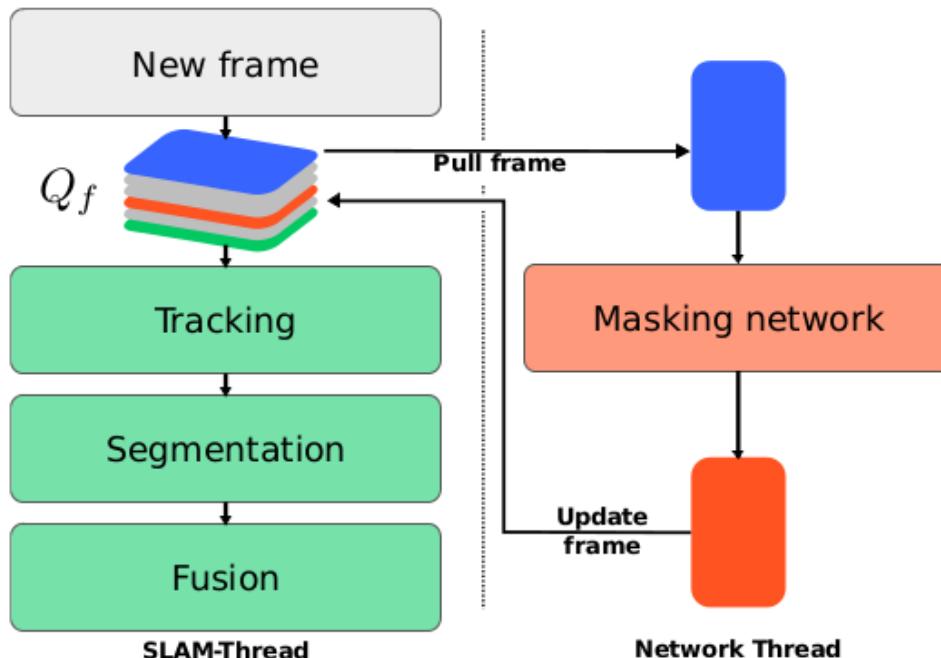
- Geometry Error

$$E_m^{icp} = \sum_i ((v^i - \exp(\xi_m)v_t^i) \cdot n^i)^2$$

其中， $\xi_m$  表示待求解的用 6DoF 李代数表示的相机位姿。 $v^i, v_t^i$  分别表示第  $i$  个 vertex in surfels，以及领一帧图像在这上面的投影。



(a) Timing of asynchronous components: In this timeline, frame **S** and **M** are highlighted with thick borders, as the SLAM and masking threads are working on them respectively. While **C**, the current frame (tail of queue  $Q_f$ ) is shown in blue, the head of the queue is shaded in green and frames with available object masks are marked orange.



(b) Dataflow in MaskFusion: Camera frames are added to a fixed length queue  $Q_f$ . The SLAM system (green) operates on its head. The semantic masking DNN pulls input frames from the tail and updates frames back to the queue, as soon as results (semantic masks) are available.

Figure 2: High-level overview of the SLAM back-end and masking network and their interaction.

**Fig 5.13.** Overview of MaskFusion

## 5.11 MaskFusion

---

- Photoconsistency residuals

把输入的 RGB 图像转化成灰度图:  $r, g, b \rightarrow 0.114r + 0.299g + 0.587b$ , 然后这一部分的误差  $E_m^{rgb}$  参考论文。

总的目标函数就是两者之和:

$$E_m = \min_{\xi_m} (E_m^{icp} + E_m^{rgb})$$

在完成 Pose 的跟踪后, 然后跟下面的分割结果进行 Associate。  
这一步并不是每一帧都这样处理。

### Segmentation

本部分主要包括三个小模块:

- Semantic Instance Segmentation

这部分就是 Mask RCNN 来做。

- Geometric Segmentation

这一部分借鉴了别人的做法, 在 Depth 图中引入两个概念:  $\Phi_d$  Depth Discontinuity term, Concavity term  $\Phi_c$ , 然后如果  $\Phi_d + \Phi_c > \tau$  那么该点就是位于一个边缘上, 是分割点。 $\tau$  是一个阈值。 $\Phi_c, \Phi_d$  在邻域上计算。

更多细节, 看论文。

这一步是每一帧都会这样处理。

- 如何把上面两个分割结果整合到一起, 然后与 Map 整合到一起

第一步与第二步并不是完全一致的, 如果 match 了, 才会整合到一起, 平时用第二步比较多。

- Mapping geometric segmentation to Mask

根据 Maximal Overlap 进行匹配。

- Mapping Mask to Model

### 5.11.3 Evaluation

使用了 ATE、RPE、RMSE 等评价指标。

### 5.11.4 总结

总的来说, 本文算是一个大杂烩, 亮点不是很明显啊, 虽然在开始的地方, 作者强调了几个本文解决的重要问题, 但实际上, 文中并没有看到清晰的对这些问题的解决。可能是我没看明白。

不过, 在 Geometry Segment 这一块, 作者借鉴了另一个工作, 这与其它算法有点不同, 其它的都是基于 CNN 的实现? 输入是 Depth 数据, 而作者这里没有用到 CNN 网络, 而是通过一种邻域阈值的方法实现的。

另外一点就是, Tracking 的时候, 可能通过引入上面提到的两种不同的误差项就可以实现多目标跟踪吧, 结果存疑, 然后公式写的还那么非主流。

总体而言, 并不好, 前面的讨论倒是挺开拓视野的。

## 5.12 小结 2

现在看来语义 SLAM 主要的研究点：

- 输入数据：

单目、多目、RGB-D 数据等, 如果是单目等, 可能就需要单独的 SLAM 框架来提供相机的 Pose 以及 Depth, 如果提供了 Depth 可以考虑用 CNN 进行处理两种模式的数据。

- 实时性

- Semantic SLAM

现在要实现 Object Instance-level, 不能再是 Point-level 了, 不然对于系统而言, 可做的事情减少很多, 因为只有认识 Object Instance 才能更好的交互, 反而如果只有一堆 Point 的分类, 对于机器系统用处不大吧, 可能。

- Dynamic SLAM

如何处理好 Moving Object 的问题。

- 输出数据:

稠密还是稀疏?

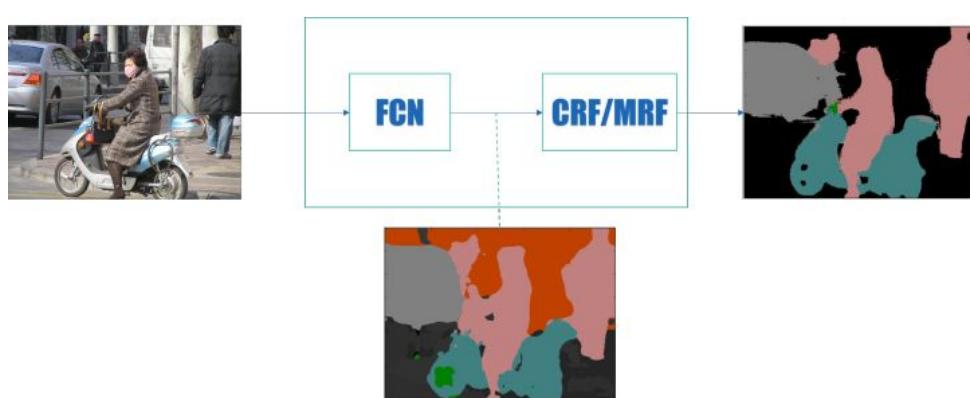
## 5.13 CNN-SLAM:

时间：2018.05.25

## 5.14 图像语义分割之 FCN 和 CRF

参考文章：[图像分割 2017.7-知乎](#)

知乎上专栏总结的一个总体分割框架：



**Fig 5.14.** 图像分割的算法框架

前端使用 FCN 进行特征粗提取, 后端使用 CRF/MRF 优化前端的输出, 最后得到分割图。

### 5.14.1 前端 FCN

#### FCN

主要用到了三种技术：

##### 1. 卷积化

卷积化即是将普通的分类网络，比如 VGG16, ResNet50/101 等网络丢弃全连接层，换上对应的卷积层即可。

##### 2. 上采样

此处的上采样即是反卷积 (Deconvolution)。当然关于这个名字不同框架不同，Caffe 和 Kera 里叫 Deconvolution，而 tensorflow 里叫 conv\_transpose。CS231n 这门课中说，叫 conv\_transpose 更为合适。

众所皆知，普通的池化（为什么这儿是普通的池化请看后文）会缩小图片的尺寸，比如 VGG16 五次池化后图片被缩小了 32 倍。为了得到和原图等大的分割图，我们需要上采样/反卷积。

反卷积和卷积类似，都是相乘相加的运算。只不过后者是多对一，前者是一对多。而反卷积的前向和后向传播，只用颠倒卷积的前后向传播即可。所以无论优化还是后向传播算法都是没有问题。具体的原理，需要参考 DLTips 一章。

##### 3. 跳跃结构

这个结构的作用就在于优化结果，因为如果将全卷积之后的结果直接上采样得到的结果是很粗糙的，所以作者将不同池化层的结果进行上采样之后来优化输出。

#### SegNet/DecovNet

这两种结构可以直接参考原文，因为就只有结构的图示，没有详细解释。

### 5.14.2 DeepLab

首先这里我们将指出一个第一个结构 FCN 的粗糙之处：为了保证之后输出的尺寸不至于太小，FCN 的作者在第一层直接对原图加了 100 的 padding，可想而知，这会引入噪声。

而怎样才能保证输出的尺寸不会太小而又不会产生加 100 padding 这样的做法呢？可能有人会说减少池化层不就行了，这样理论上是可以的，但是这样直接就改变了原先可用的结构了，而且最重要的一点是就不能用以前的结构参数进行 fine-tune 了。所以，Deeplab 这里使用了一个非常优雅的做法：将 pooling 的 stride 改为 1，再加上 1 padding。这样池化后的图片尺寸并未减小，并且依然保留了池化整合特征的特性。

但是，事情还没完。因为池化层变了，后面的卷积的感受野也对应的改变了，这样也不能进行 fine-tune 了。所以，Deeplab 提出了一种新的卷积，带孔的卷积：Atrous Convolution. 即图 5.15 所示。

具体的感受野变化：

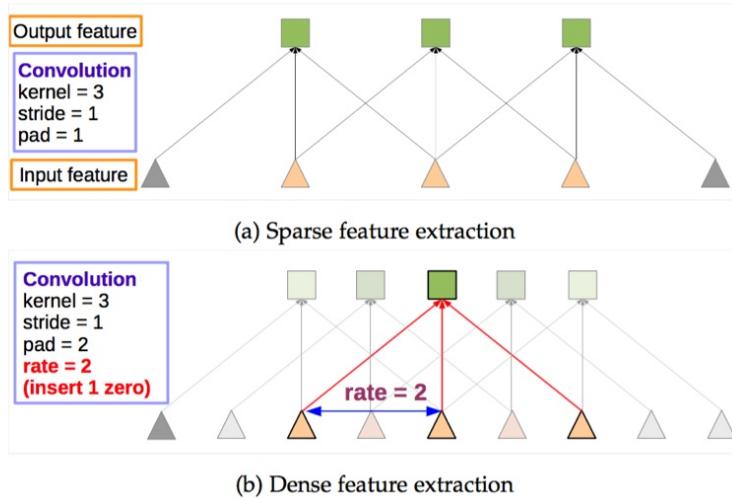


Fig 5.15. Atrous Convolution 示意图

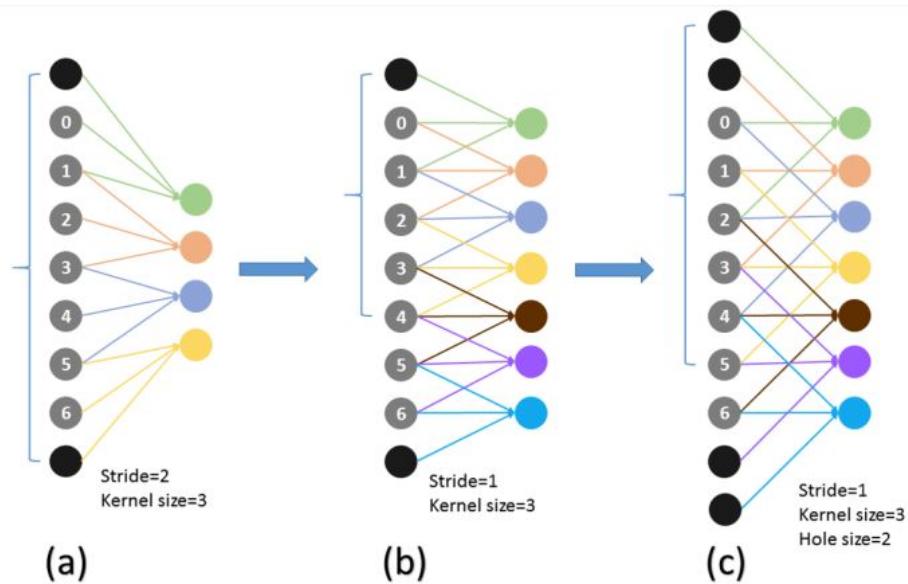


Fig 5.16. Atrous Convolution 中的感受野变化示意图

## 5.14 图像语义分割之 FCN 和 CRF

---

图5.16中， $a$ 为普通的池化的结果， $b$ 为“优雅”池化的结果。我们设想在 $a$ 上进行卷积核尺寸为3的普通卷积，则对应的感受野大小为7。而在 $b$ 上进行同样的操作，对应的感受野变为了5(与 $a$ 中颜色相同的输出所对应的输入只有0, 1, 2, 3, 4。所以感受野变为5了。). 感受野减小了。但是如果使用hole为1的Atrous Convolution则感受野依然为7. 判断感受野的过程是，在输出数量一致的情况下(即和 $a$ 中的输出的4种颜色相同的输出)，所覆盖的输入的个数。

所以，Atrous Convolution能够保证这样的池化后的感受野不变，从而可以fine tune，同时也能保证输出的结果更加精细。

其实就是Dilated Convolution的思想。

### 5.14.3 后端优化 CRF/MRF

#### 全连接CRF, DenseCRF

对于每个像素 $i$ 具有类别标签 $x_i$ 还有对应的观测值 $y_i$ ，这样每个像素点作为节点，像素与像素间的关系作为边，即构成了一个条件随机场。而且我们通过观测变量 $y_i$ 来推测像素 $i$ 对应的类别标签 $x_i$ 。也就是说，这里的观察值 $y_i$ 为对应第 $i$ 位置的像素。

二元势函数就是描述像素点与像素点之间的关系，鼓励相似像素分配相同的标签，而相差较大的像素分配不同标签，而这个“距离”的定义与颜色值和实际相对距离有关。所以这样CRF能够使图片尽量在边界处分割。

在这里，势函数的输入来自FCN的输出。

而全连接条件随机场的不同就在于，二元势函数描述的是每一个像素与其他所有像素的关系，所以叫“全连接”。

#### CRFasRNN

最开始使用DenseCRF是直接加在FCN的输出后面，可想这样是比较粗糙的。而且在深度学习中，我们都追求end-to-end的系统，所以CRFasRNN这篇文章将DenseCRF真正结合进了FCN中。

这篇文章也使用了平均场近似的方法，因为分解的每一步都是一些相乘相加的计算，和普通的加减（具体公式还是看论文吧），所以可以方便的把每一步描述成一层类似卷积的计算。这样即可结合进神经网络中，并且前后向传播也不存在问题。

当然，这里作者还将它进行了迭代，不同次数的迭代得到的结果优化程度也不同（一般取10以内的迭代次数），所以文章才说是as RNN。

#### 马尔科夫随机场，MRF

没看懂，但还是把原文贴过来了。论文是：Deep Parsing Network  
算了，还是去看论文吧，虽然我还没看。

#### 高斯条件随机场，G-CRF

### 5.14.4 小结

2017.07.06

- FCN 更像一种技巧。随着基本网络（如 VGG, ResNet）性能的提升而不断进步。
- 深度学习 + 概率图模型（PGM）是一种趋势。其实 DL 说白了就是进行特征提取，而 PGM 能够从数学理论很好的解释事物本质间的联系。
- 概率图模型的网络化。因为 PGM 通常不太方便加入 DL 的模型中，将 PGM 网络化后能够是 PGM 参数自学习，同时构成 end-to-end 的系统。

## 5.15 Learning Deconvolution Network for Semantic

参考文献：[Learning Deconvolution Network for Semantic 1-知乎](#)

[Learning Deconvoluton Network for Semantic 2-知乎](#)

背景：FCN 具有以下限制：

- 固定尺寸的感受野。对于大尺度目标，只能获得该目标的局部信息，该目标的一部分将被错误分类；对于小尺度目标，很容易被忽略或当成背景处理
- 目标的细节结构容易丢失，边缘信息不够好。FCNs 得到的 label map 过于粗糙，而用于上采样的反卷积操作过于简单

本文的主要贡献：

- 提出了可学习的反卷积网络，并将其首次应用于语义分割
- 分割结果是 Instance-wise 的分割，所以摆脱了原来 FCNs 中的尺度问题，这也是对应 FCNs 中的第一个问题
- 在 Pascal Voc12 测试集上得到一个很好的效果，与 FCN8s 模型融合，得到当时最好的准确率

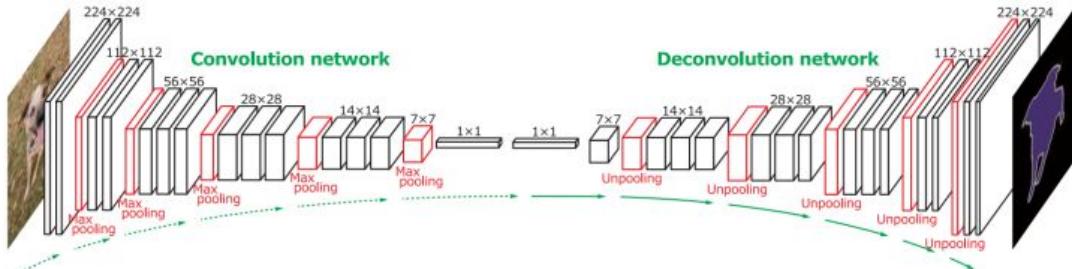


Figure 2. Overall architecture of the proposed network. On top of the convolution network based on VGG 16-layer net, we put a multi-layer deconvolution network to generate the accurate segmentation map of an input proposal. Given a feature representation obtained from the convolution network, dense pixel-wise class prediction map is constructed through multiple series of unpooling, deconvolution and rectification operations.

**Fig 5.17.** 论文提出的神经网络结构

反卷积网络主要由 unpooling 层、deconv 层、relu 层、BN 层组成。

### Instance-wise Segmentation

每张图片被裁剪为很多子图片，每张子图片包含一个目标实例。每张子图片作为输入，得到分割结果，再把这些分割结果聚合起来，得到整张图片的分割图。

## 5.16 ReSeg: A RNN-based Model for Semantic Segmentation

参考文献: [35]

基于 ReNet 实现，每一层 ReNet 基于四个 RNN 实现，这四个 RNN 分别对应水平、垂直方向的双向信息。此外，ReNet is stacked 在 CNN 之上，可以利用 Generic local features。在 ReNet 之后，是提高分辨率的网络。

### 5.16.1 背景 & 相关工作

普通的 CNN 会极大降低分辨率。FCN 类的方法不能很好的利用 local 和 global contextual dependencies，而这些已被证明对分割具有较大帮助。因此，这些模型也经常利用 CRF 等作为后端处理，来 locally smooth the model predictions，但对于 long-range contextual dependencies 还是没有被很好的研究过。

而 RNN 可已被用于 retrieve global spatial dependencies，但是计算量较大。In this paper, we aim at the efficient application of Recurrent Neural Networks RNN to retrieve contextual information from images.

ReNet 的每一层可以通过先水平扫描，然后垂直扫描来提高获取 global information 的能力。

在相关工作中，RNN 可以实现分析长距离像素之间的关系，因此被用于语义分割中。ReNet 具有很高的并行性，因为每一行之间的计算是相互独立的，每一列的处理也是独立并可以并行计算。

### 5.16.2 Model Description

首先，输入图像被送入在 ImageNet 上预训练好的 VGG16 网络。

然后，输出的 Feature Maps 被送进 ReNet 层，该结构可以 sweep over 输入的 feature maps。

最后，多个 Upsampling layer 被用于把最后的 Feature map 的分辨率恢复到输入图像的分辨率的大小，接着，用 Softmax 进行分类。

使用 GRU 来很好的平衡存储量与计算能力的开销。

#### Recurrent layer

结构如图5.18所示。

其中， $f^\downarrow$  与  $f^\uparrow$  分别表示垂直方向两个方向的 RNN，具有 U Recurrent unit 结构？

#### Upsampling layer

作者说有三种提高分辨率的方法：

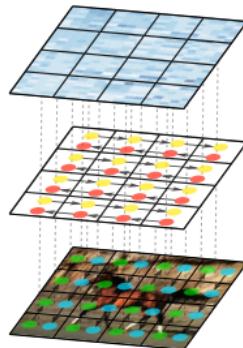


Figure 1. A ReNet layer. The blue and green dots on the input image/feature map represent the steps of  $f^\downarrow$  and  $f^\uparrow$  respectively. On the concatenation of the resulting feature maps,  $f^\rightarrow$  (yellow dots) and  $f^\leftarrow$  (red dots) are subsequently swept. Their feature maps are finally concatenated to form the output of the ReNet layer, depicted as a blue heatmap in the figure.

**Fig 5.18.** ReNet layer 结构

- Fully Connected layers

这种方法不适合，没有考虑输入的拓扑结构

- Full convolutions

这种方法需要大的 Kernel 与 Stride size 等

- Transposed Convolutions

这个方法好，既 Memory efficient 又 Computation Efficient。

Transposed Convolution, 又被称为 Fractionally strided convolutions. 下面是比较有意思的解释：

Convolution is based on the observation that direct convolutions can be expressed as a dot product between the flattened input and a sparse matrix, whose non-zero elements are elements of the convolutional kernel. The equivalence with the convolution is granted by the connectivity pattern defined by the matrix.

Transposed convolutions apply the transpose (转置) of this transformation matrix to the input, resulting in an operation whose input and output shapes are inverted with respect to the original direct convolution.

上一段的引用，详情可以从参考第八章中关于 Deconvolution 的说明，是一致的，且还有例子！

关于 Deconvolution 的更多说明可以参考：A guide to convolution arithmetic for deep learning.

### 5.16.3 实验结果

数据集：

Weizmann Horse, Oxford Flowers 17, CamVid Dataset.

使用 IoU。

一个重要的现象：当数据集是 Highly imbalanced, 分割效果会极大的变差，因为此时网络会最大化那些经常出现的类别的分数，而忽略不经常出现的类别？为了解决这个问题，作者在 Cross-entropy loss 中加入了一个新的项，来 bias the prediction towards the low-occurrence classes.

具体的就是：**Median Frequency Balancing**。

which re-weights the class predictions by the ratio between the median of the frequencies of the classes (computed on the training set) and the frequency of each class. This increases the score of the low frequency classes (see Table 4) at the price of a more noisy segmentation mask, as the probability of the underrepresented classes is overestimated and can lead to an increase in misclassified pixels in the output segmentation mask.

### 5.16.4 总结

The proposed architecture shows state-of-the-art performances on CamVid, a widely used dataset for urban scene semantic segmentation, as well as on the much smaller Oxford Flowers dataset. We also report state-of-the-art performances on the Weizmann Horses.

## 5.17 U-Net: CNN for Biomedical Image Segmentation

参考文献：[26]

本文的一大贡献：证明通过更好的使用数据增广来提高现有训练数据的利用率。

另一大贡献，是提出一种结构，该结构由 Contracting 和 Expanding 两个部分组成，这一点与 FlowNet 类似啊。

结果表明，仅需要非常少的图像数据就可以实现不错的效果。

### 5.17.1 Network Architecture

本文提出的网络结构实在 FCN 基础上进行的改进。

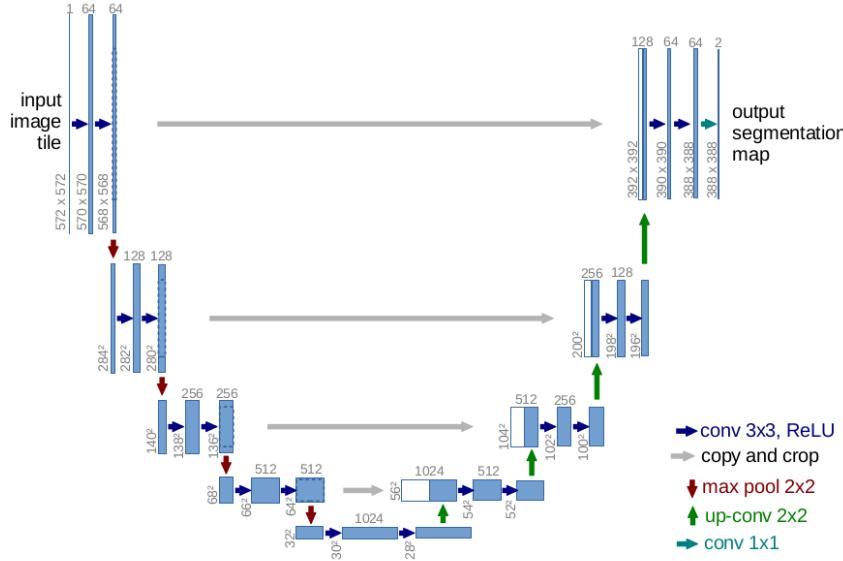
网络结构如图5.19所示。

与 FCN 不同的是，U-Net 把结构分成 Contracting 与 Expanding 两个 Path，Contracting 部分与普通的卷积网络相同。It consists of the repeated application of two  $3 \times 3$  convolutions (unpadded convolutions), each followed by a rectified linear unit (ReLU) and a  $2 \times 2$  max pooling operation with stride 2 for downsampling. At each downsampling step we double the number of feature channels.

而 Expanding 部分与 FCN 中不同的在于，其 Feature Channel 特别多，与 Contracting 中基本对称的一样多。每一层包含一个'Up-Convolution'，同时把 Feature Channel 数量减半。

在最后一层，一个  $1 \times 1$  的 Convolution 被用于把 64-Component feature vector 映射到目标的 number of classes。

Contracting 部分的 Feature Map 与 Expanding Path 部分的 Feature Map 会进行 **Concatenation**，这一点与 FCN 也不同。U-Net 是在 Channel 维度进行拼接，而 FCN



**Fig. 1.** U-net architecture (example for 32x32 pixels in the lowest resolution). Each blue box corresponds to a multi-channel feature map. The number of channels is denoted on top of the box. The x-y-size is provided at the lower left edge of the box. White boxes represent copied feature maps. The arrows denote the different operations.

**Fig 5.19.** U-Net 结构示意图

是逐点相加！参考文章：[FCN/U-Net 结构解析 - 知乎](#)，这一点属于特征融合步骤的不同！

在训练过程中，用到了 Cross Entropy 作为目标函数，同时进行数据增广：Shift, Rotation, Deformation, Gray value variantions.

貌似 Random elastic deformation 对于训练少量 Label 的数据来说非常重要。We generate smooth deformations using random displacement vectors on a coarse 3 by 3 grid. The displacements are sampled from a Gaussian distribution with 10 pixels standard deviation. Per-pixel displacements are then computed using bicubic interpolation. Drop-out layers at the end of the contracting path perform further implicit data augmentation.

### 5.17.2 Conclusion

Thanks to data augmentation with elastic deformations (弹性变形)，it only needs very few annotated images and has a very reasonable training time of only 10 hours on a NVidia Titan GPU (6 GB).

### 5.17.3 补充

- U-Net 中的'Up-Convolution' 是什么意思？

知乎上有文章说，就是用的 Deconv 来实现的。

- 在最后的几层，如何把 Feature map 映射成语义分割图像呢？这在 FCN[18] 中也是之前被忽略的问题。

其实在最后的输出层也是一个 Feature map，它的 Channel 数量就是目标类别的数量，比如，在在 Pacal 中类别数为 21(包括 Background)，所以输出的最后的 Feature Map 具有 21 个 Channel，这个可以参考 FCN 论文中的图 1。另外，在图 5.19 中也可以看出，最后的输出通道数为 2，即每一类别代表一个 channel，然后利用 SoftMax 进行在 Channel 维度上进行计算，得到对应像素的类别结果。这一步一般通过一个  $1 \times 1$  的 Convolution 来实现。这一点也可以参考下面 FCN 论文中的原话：

We append a  $1 \times 1$  convolution with channel dimension 21 to predict scores for each of the PASCAL classes (including background) at each of the coarse output locations, followed by a deconvolution layer to bilinearly upsample the coarse outputs to pixel-dense outputs as described in Section 3.3.

这么说来，原来 Fully Convolution 是这个意思，即相当于在原来 Fully Connect 的层在空间维度上进行扩充扩充，最终的结果是，现在每一个像素位置都对应于一个原来 Fully Connect 的维度的 Vector！

总结一下，CNN 图像分割基本上套路是：

- 下采样 + 上采样：Convolution + Deconvolution /Resize
- 多尺度特征特征融合：如 Concatenate 或者逐点相加
- 获得像素级的 Segment Map，一般通过  $n * 1 \times 1$  Convolution 得到，其中  $n$  为类别数目。

## 5.18 Semantic Visual Localization

参考文献：[28] 妹的，21 页。

看样子，本文的主要关注点在于建立：Robust visual localization under a wide range of viewing condition.

本文提出利用联合 **3D Geometry** 和 **Semantic understanding of the world** 来帮助解决这个问题。

提出的算法，利用一个新提出的生成模型来完成 descriptor learning，该过程 trained on semantic scene completion as an auxiliary task.

最终，得到的 3D descriptors 可以很鲁棒，可以借助高层的 3D geometric 以及语义信息来处理丢失 Observe 的情况。

看样子，本文算法比较新颖，效果也很好，一个比较大的优势是采用了生成模型，而不是常用的判别模型。

### 5.18.1 背景 & 相关工作

其实这篇文章主要的关注点是 Localization，也就是新的传感器数据来了后，怎么更加准确的与地图中的信息进行匹配，难点在于：The main challenge in this setting is successful data association between the query and the database.

为什么难呢，因为实际的场景是经常变化的，所以很容易就匹配错误了。一般的做法是基于描述子之间的匹配来定位，且通过提高描述子的质量来提高定位的质量。这种方法还是不好，比如尺度性不好、需要大量的匹配计算等。

本文提出了一种高效的描述子学习算法，是基于生成模型而不是判别模型实现的。本文证明了，通过引入 Semantics 提供了 strong cues 对于场景补全。而且不需要人为操作，就可以泛化到其它数据集以及其它传感器类型，并且都不需要重新训练！（生成模型这么强么还是本文的算法很强？）

生成的是 3D 描述子。需要解决的一个重要问题是物体遮挡的场景 (Occlusion)，本文提出的算法通过借助 Semantic Completion 这一辅助任务来实现遮挡问题的解决。

本文的算法本质上还是提高 Descriptor 的生成质量以及匹配速度，在 Euclidean space 内完成。效果来看，可以在不同光照、季节变换等场景下实现定位。

优点没看明白。在相关工作中，作者分析了大量的已有的算法的缺点，我记得的主要的缺点就是：生成的 Descriptor 不鲁棒或不满足要求、匹配过程复杂、对多变场景处理不好等等。

### 5.18.2 Semantic Visual Localization

从上文可以看出，作者的主要贡献是：在 Euclidean 生成包含语义的 3D Descriptor，可以满足场景的多变、快速匹配、很高的精确度等需求。

模型的输入数据：1) A set of color images with associated depth maps, 2) database image, including their respective camera poses.

算法过程：

1. 对于 Database 的一个子 set，预先生成其语义地图： $M_D$
2. 对于 Query 的一张或多张图像生成其语义地图  $M_Q$
3. 建立  $M_D$  与  $M_Q$  之间的匹配，得到相机的位姿估计  $\mathcal{P}_Q$

上述算法过程，得到的结果，作者认为就可以实现对 Extreme viewpoint 和光照的鲁棒了。Semantic information is comparatively invariant to these types of transformations through higher-level scene abstraction.

额，貌似走着不是对输入的 RGB 图像进行提取 Descriptor，而是对 Semantic Maps 提取 Descriptors！这也行么？

按照上面三个步骤，具体内容如下：

- Semantic Segmentation and Fusion

首先生成所有输入图像的 Dense pixelwise semantic segmentation，然后把这些分割结果融合到 3D Voxel maps  $M_D$  与  $M_Q$ 。

任务的目的是计算把 Query 和 database map 之间最佳匹配的位姿  $P \in SE(3)$ 。

下面给出如何在缺少观察的情况下，通过对场景的语义理解来建立鲁棒的 Semantic Maps。

- Generative Descriptor Learning

又强调了一次：Semantic scene understanding is key to learning such an invariant function.

## 5.18 Semantic Visual Localization

传统方案中，有两种方式提高这种 Invariant，一种是学习更好的 Matching function，一种是学一种更好的 Embedding。前者效果较好，但计算量很大，因为要比较每一对的 Descriptor，后者计算量小，本文的算法只需要每一个 Descriptor 计算一次就好。

具体使用一个 Encoder 来对 Scene Semantics 和 Geometry 进行编码。为了可以推理出没观察到的部分，算法还涉及一个 Semantic scene completion 的辅助任务，同时推理 Scenem Semantic 和 Geometry。整个结构如图5.20所示：

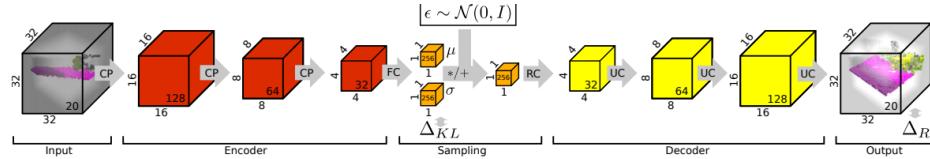


Figure 2: **Variational Encoder-Decoder Architecture.** Legend: CP = Convolution + Pooling, FC = Fully Connected, RC = Reshape + Convolution, UC = Upsampling + Convolution,  $\Delta_{KL}$  = KL Divergence wrt.  $\mathcal{N}(0, I)$ ,  $\Delta_R$  = Reconstruction Loss. The numbers at the bottom right of each block denote the number of feature channels. The network takes incomplete semantic observations as input (left) and predicts completed semantic subvolumes (right). The latent code  $\mu$  forms our descriptor.

**Fig 5.20.** Generative Descriptor Learning 的示意图

- **Bags of Semantic Words**

Note that given an incomplete map, the bag of semantic words is a description of its complete semantic scene layout. Our localization pipeline uses this representation to robustly match the query map to the database map, as detailed in the following.

这个 bag of semantic words 是通过计算属于  $M$  的所有 subvolume 来得到集合。也就是 Encoder 部分的输出构成的集合，具体得看论文吧。

- **Semantic Vocabulary for Indexing and Search**

基于 Euclidean metric 来计算 Query 和 Database 之间的相似性，具体是利用 Encoder 部分的输出来计算距离。

- **Semantic Alignment and Verification**

计算一组 Query 在不同 Rotation 下与 Database 之间的相似性，这些计算结果作为准确定位的基础，这就是这一小节要说明的内容。

通过类似于一种重投影误差的方式来判断计算的位姿的效果的好坏。

### 5.18.3 Experiments

用到了 KITTI, NCLT 数据集。

重复的场景结构容易引入 Local Ambiguities，会导致错误的定位。Global Ambiguities 发生的较少，但会导致上百米的定位错误。通过考虑多帧的数据可以有效降低上述两种 Ambiguities。此外，可以增加训练数据的多样性对于提高精度或许也有帮助。

### 5.18.4 Conclusions

In this paper, we proposed a novel method for localization using a joint semantic and geometric understanding of the 3D world.

At its core lies a novel approach to learning robust 3D semantic descriptors.

## 5.19 小结 3

在前面两个小结的基础上，这里在强调一下几个重要的改进点！：

首先，实验目的是什么，有一些是实现 Semantic SLAM，也有提高定位信息的！然后，输入数据是什么，现在趋势是借助多模态的数据，最经常用的是 Depth 和 RGB 图像，此外，还有前面涉及到的 Contour 以及 Semantic Information 和 3D Geometry 信息，还有就是多帧数据实现非监督什么的。所以，一个可选的工作是，怎么把这些数据最高效的整合起来，可以借助 RNN 中的 Gate 思想，也可以看一下 Knowledge 蒸馏法，以及 Attention 机制什么的!!!

## 5.20 StaticFusion: Background reconstruction for dense RGBD SLAM Dynamic Environments

参考文献：[29]

主要目的：提出一个鲁棒的、稠密的、基于 RDBD 的、可处理 Dynamic Environment 的 SLAM 系统。

可以实现对 Static 和 Dynamic Environment 的分割，以及只对 Static 部分建模，来提高相机位姿估计的精度，降低 overall drift 的影响。

### 5.20.1 背景

现有的 SLAM 系统都假设场景是静态的，所以通过图像对之间的匹配可以得到相机的变换关系。但是现在场景中存在动态目标，那么图像之间的匹配就不单单是由于相机变换引起的了，所以需要区分静态场景和动态场景，并且动态场景对地图的构建也会产生不可逆的影响。

主要做了两件事：

- 同时估计相机的位姿以及对 static 和 Dynamic 场景的分割

其中，RGBD 相机估计，通过把新来的图像帧与一个稠密的 surfel-based SLAM model 一起实现，如 ElasticFusion。

- 使用 Static 场景对环境进行了更具有一致性的建模

作者有公开源码。

### 5.20.2 动态场景下 SLAM 系统的相关工作

ElasticFusion 是首个基于 RGBD 的 SLAM 系统，现在在 Scalable estimation、loop closure capabilities、consider run-time limitations 等方面改进，这一小节主要总结在 Dynamic Environment 场景下的一些改进方法。主要有三个方面：

## 5.20 StaticFusion

- Implicit Handing of Dynamic Elements

这一类算法通过对 High-residual points 施加惩罚项来提高位姿估计的鲁棒性。其他的如 ElasticFusion 等算法在构建地图之前，需要目标在连续的多帧图像中都可以观察到，才会被认为是 Static 部分，加入到地图构建中。但这些算法并不会明确的对 Dynamic Object 进行处理，只能适用于运动物体在图像中范围比较小或者相机的运动幅度很小。

- Outlier Rejection Strategies

一种常用的方法是把 Dynamic moving object 当做噪声来处理，被检测出来然后被滤波去除。

这一类的方法包括：DTAM、ORB-SLAM 等。通过非常保守的关键帧选取策略来提高系统的鲁棒性。但这些算法并没有考虑在连续帧中检测 Dynamic points 是使用 Spatial 和 temporal 上面的 Coherence。

**评语：**所以在 Temporal 上的 Coherence 可以使用 RNN 来帮助解决，而在 Spatial 上面的 Coherence 可以通过 CRF 之类的算法实现，最近不是也有 ConvCRF 的算法出现么！可以考虑考虑。

- Methods Enforcing Spatial on temporal Coherence

此外，还可以借助 IMU 以及 Robot's odometry 来提高 Dynamic 下的 SLAM 的鲁棒性，但本文主要基于单目 RGBD 相机来做。

### 5.20.3 Framework 和 Notation

主要的流程：

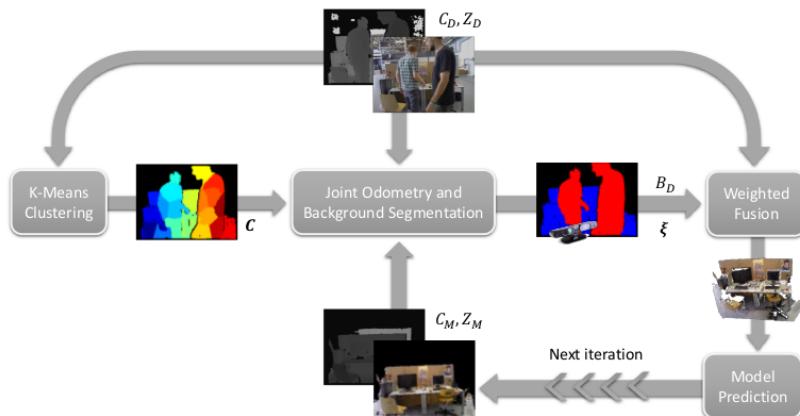


Fig. 1: System architecture: the process starts by receiving a new RGB-D image ( $C_D, Z_D$ ) and grouping its pixels into geometric clusters  $C$ . A prediction ( $C_M, Z_M$ ) is rendered from the model and the last pose estimate  $T(\xi)$  and used for joint alignment and background segmentation. Both results are then exploited for weighted fusion of the static clusters of  $C_D, Z_D$  with the map.

**Fig 5.21.** StaticFusion 算法的三个主要流程

三个步骤：

1. From incoming pair  $I_D, Z_D$  得到 3D Geometry Clusters  $\mathbf{C}$

其中,  $C_D$  是输入的彩色图像,  $Z_D$  是输入的 Depth 图像,  $I_D$  是基于  $C_D$  得到的 Intensity image。这个分割算法来自于: Fast odometry and scene flow from RGB-D cameras based on geometric clustering. 作者假设分割后的每一个 cluster 是 rigid body.

2. 根据上一帧的相机位姿预测这一时刻的场景图片  $C_M, D_M$ , 然后这个预测结果与这一时刻实际的数据的分割结果来联合估计相机的运动  $\xi \in \mathfrak{se}(3)$ , 与此同时, 这一步还会得到一个基于运动的场景分割, 对于每一 Cluster, 我们分配一个 Score,  $b_i \in [0, 1]$ , 来表示这一个 Cluster 属于 Dynamic 还是 Static, 本文中,  $b_i = 1$  是 Static,  $b_i = 0$  是 Dynamic, 在之间则表明不同程度的不确定性。

具体还会用到上一时刻截止的 Map 信息。

3. 最后, 利用上一步计算得到的  $\xi$  和运动分割结果, 得到一个 pixel-wise 的分割图 5.21 中的  $B_D$ , 这一步只处理输入上一步计算的 Static 部分来计算。然后这个计算结果与输入  $C_D, Z_D$  进行加权 3D Fusion。

每一步的具体实现, 还需要下面的讲解。

## 第二步的计算

### 5.21 Convolutional CRFs for Semantic Segmentation

参考文献: [19]

之前的研究人员会用 CRF 作为语义分割的后处理步骤, 但 CRF 有两个缺点: 训练困难, 推理计算速度慢。

为了解决这两个问题, 作者在本文中提出在全连接 CRF 中引入条件独立性的假设。这样的后果就是, 我们可以以卷积的形式重新组织 CRF 的推理计算过程, 而且可以通过后向传播的方式在训练过程中优化模型参数。最终可以借助 GPU 进行加速训练, 效果是速度可以提高两个数量级。

#### 5.21.1 背景 & 相关工作

传统用于语义分割的 CNN 的缺点是: 只能提取局部特征, 不能利用全局的 Contexture。所以有人认为简单的前向 CNN 不能满足 Structured Predictions 类任务。

在 FCN 的基础上, Transposed convolution layer 用于提高 Feature Map 的分辨率, Dilated convolutions 用于提取空间信息 (Spatial information)。很多网络结构都是基于这种思路来做的! 而这一类做法严重依赖于 CNN 提取特征的能力, 进行 pixel-wise 的预测。structured knowledge and background context 被忽略。

一种在 CNN 中加入 sturctured predictions into CNN 的方法是在 CNN 的输出上加入 fully-connected CRF 进行后续处理。Pyramid pooling 是另一种与 CRF 类似作用的方法, 用于提高 CNN 的感受野, 但并不会提供真正的 structured reasoning。

还是那句话, 主要是两个问题, 那么现在有哪些方法来解决这两个问题呢?

- Parameter Learning in CRFs

## 5.21 Convolutional CRFs for Semantic Segmentation

---

一些方法包括：EM 和 Grid-search, Gradient Decent, Quadratic Optimization 等。但都不太适合 CNN 的训练流程。

- Inference speed of CRFs

一些方法输出降维的数据，但会影响预测能力。据作者所知，目前还没有一种可以有效提高 FullCRFs 推理速度的方法。

### 5.21.2 Fully Connected CRFs

CRF 嘛，基于随机场的原理来实现的。首先说明下符号： $I$  是输入图像，具有  $n$  个像素； $\mathbf{X} \in \{X_1, \dots, X_n\}$  表示每一个像素的标签，即分割结果中的类别；假设共有  $k$  个类别，即每一个  $X_i$  都有  $k$  个可能的取值。所以得到的条件随机场模型，以吉布斯分布的形式(基于成本函数表示的吧应该)：

$$P(X = \hat{x} | \tilde{I} = I) = \frac{1}{Z(I)} \exp(-E(\hat{x}|I))$$

其中， $E(\hat{x}|I)$  表示能量函数，由下式给出：

$$E(\hat{x}|I) = \sum_{i \leq N} \psi_u(\hat{x}_i|I) + \sum_{i \neq j \leq N} \psi(\hat{x}_i, \hat{x}_j|I)$$

上式中的第一项是 Unary Potential，传统的来自于普通的 CNN 的语义分割结果可以认为就是这一项的具体取值了。那么还有后一项，后一项包含两个像素的分割结果，表示的就是利用 Spatial Information 了。重点说一下这一项。

后一项叫做 Pairwise Potential，表示  $\hat{x}_i, \hat{x}_j$  的联合分布，这里的  $x_i, x_j$  其实也是可以是来自于 CNN 的语义分割预测结果，与 Unary potential 中的输入是一样的！可以用于明确表示这两个像素之间的联系，如近似的颜色等。在 FullCRFs 中，这一项是基于 weighted sum of M 个 gaussian kernels 来实现的。

$$\psi(x_i, x_j|I) = \mu(x_i, x_j) \sum_{m=1}^M \omega^{(m)} k_G^{(m)}(f_i^I, f_j^I)$$

其中， $\omega^{(m)}$  是可以学习的参数，特征向量  $f_i^I, f_j^I$  可以任意选择，并可以依赖于输入图像  $I$ 。但是最开始的那个函数  $\mu(x_i, x_j)$  是 the compatibility transformation，只与  $x_i, x_j$  有关，而与输入无关。接下来重点说一下这个函数的选择。

一种普遍的选择是用 Patts Model 来做  $\mu(x_i, x_j) = |x_i - x_j|$ ，关于这个模型，可以参考 Wikipedia，我记得笔记中也有，但记不清了。而在 CRFasRNN 中，提出利用  $1*1$  的卷积来做 Compatibility transformation。这样一种函数可以让模型学习更多的 structured interactions between predictions。

说完了  $\mu$  函数，那么来看一下在 FullCRF 中，是怎么实现后面对 Gaussian kernel 的加权的吧。在 FullCRF 中，有两个 Gaussian Kernelz 作为 hand-crafted features。Appearance kernel  $k_\alpha$  考虑了颜色的取值  $I_i, I_j$  作为特征；smoothness kernel 基于 spatial coordinates  $p_i, p_j$  来实现。整个的 Pairwise potential 由下式给出：

$$k(f_i^I, f_j^I) = \omega^{(1)} \exp\left(-\frac{|p_i - p_j|^2}{2\theta_\alpha^2} - \frac{|I_i - I_j|^2}{2\theta_\beta^2}\right) + \omega^{(2)} \exp\left(-\frac{|p_i - p_j|^2}{2\theta_\gamma^2}\right)$$

其中， $\omega, \theta$  都是可以学习的参数。需要指出的是，这是一种 Hand-crafted feature，有助于提高 CRF 的优化效率。

## Full CRF 的 Mean field inference

FullCRF 的推理主要基于 Mean Field inference 来实现的。具体步骤如图5.22所示。

**Algorithm 1** Mean field approximation in convolutional connected CRFs

---

```

1: Initialize:  $\tilde{Q}_i \leftarrow \frac{1}{Z_i} \exp(-\psi_u(x_i|I))$  "softmax"
2: while not converged do
3:    $\tilde{Q}_i(l) \leftarrow \sum_{i \neq j} w^{(m)} k_G^{(m)}(f_i^I, f_j^I) \tilde{Q}_i(l)$   $\triangleright$  Message Passing
4:    $\tilde{Q}_i(x_i) \leftarrow \sum_{l' \in L} \mu(x_i, l') \tilde{Q}_i(l)$   $\triangleright$  Compatibility Transformation
5:    $\tilde{Q}_i(x_i) \leftarrow \psi_u(x_i|I) + \tilde{Q}_i(x_i)$   $\triangleright$  Adding Unary Potentials
6:    $\tilde{Q}_i(x_i) \leftarrow \text{normalize}(\tilde{Q}_i(x_i))$   $\triangleright$  e.g. softmax
7: end while

```

---

**Fig 5.22.** Mean field inference 算法步骤

对这个算法，简单说两句，除了 message passing 这一步之外，其它的几个步骤都是可以高度并行的。可以用 GPU 进行加速。Message Passing 也是 CRF 计算的瓶颈，计算量与输入图像的像素数的平方成正比，有研究人员提出利用 Permutohedral lattice 模型来近似，从而提高计算效率，这个近似模型是一种高维的滤波算法。但这种模型是基于一种很复杂的数据结构来实现的，存在基于 CPU 的复杂但快速的实现，但不适合 GPU 的实现。实际上，Permutohedral lattice 模型的梯度求解也是一件很麻烦的事情，这也是为什么 FullCRF 之类的算法基于 Hand-crafted features 来做。

### 5.21.3 Convolutional CRFs

Convolution CRF 其实就是在 Full CRF 的基础上引入了条件独立假设，即如果两个特征之间的 Manhattan 距离超过一个阈值  $k$  即  $d(i, j) > k$  那么就认为这两个点之间是独立的，这个  $k$  作者称为 filter-size。

在 Pairwise Potential 上面的体现就是，若大于这个阈值， $\psi_p$  为 0。

#### Efficient Message Passing in ConvCRFs

上面也提到了，Message Passing 是 CRF 的计算瓶颈，本文提出的方法可以有效解决这个瓶颈，这是本文的主要贡献之一，就不需要再使用 Permutohedral 模型近似了，也可以借助 GPU 加速计算了。为了实现这个目标，作者提出利用截断 Gaussian Kernel 实现的卷积操作来完成 Message Passing 这一步，这样就可以利用普通的 CNN 来完成了。

假设输入  $P$  是的尺寸是  $[bs, c, h, w]$ ，分别对应 Batch size、通道数、高度、宽度。最终可以得到下面的公式，具体的构成看论文吧，那个公式有点复杂。

$$Q[b, c, x, y] = \sum_{dx, dy \leq k} K[b, dx, dy, x, y] \cdot P[b, c, x + dx, y + dy]$$

可以看出滤波器的取值还与 Spatial dimension 有关，需要注意的地方是，卷及操作实在通道为度上进行的。这里进行的信号处理中的卷及操作，而不是 NN 中的卷及操作，后者其实是相关操作。

这一步可以基于 GPU 来加速，但会涉及 GPU 中数据的重新组织，而实际上 90% 的 GPU 运算时间都花在了数据重新组织上了。有一段关于卷积运算的阐述看论文。

### Additional implementation details

ConvCRF 中的设计策略与 FullCRF 中相同，即使用 Softmax 进行归一化、使用 Potts model 以及 Hand-crafted gaussian feature 等。同样的，对 Pairwise kernels 做 Gaussian blur，这样可以增加滤波器的有效大小变成 4 倍，这是为什么！

作者也实现了基于  $1 \times 1$  卷积的实现，作为实验结果进行分析，或者在 Pairwise 中的输入换成可以学习的 Feature 数值。

使用 VOC2012。使用 ResNet101 作为 Unary potentials。

训练的时候，采用 Decoupled Training 的策略。即首先训练 CNN 模型，以语义分割为目标；然后固定 CNN 的输出，进行 CRF 的训练。

**评语：可以看得出来，作者是用 CNN 输出的完整分辨率的 CRF 进行训练的，这得多耗时，所以，下一步就是在 Feature 领域进行 CRF 操作！**

### 5.21.4 总结

通过增加条件独立的假设，我们可以去除 Permutohedral lattice approximation，从而可以借助 GPU 完成高效的消息传递的计算，并是以卷积的方式进行。

下面，作者说还是要如何更好的捕获全局信息。也会把 ConvCRF 应用于 Instance Segmentation 以及 Landmark recognition 等应用中看一下效果怎么样！

## 5.22 Co-Fusion: Real time segmentation tracking and fusion of multiple objects

参考文献：[27]

输入的数据是 RGBD，主要通过对运动物体进行建模，完成场景的描述，可以作用在动态场景下。

表现在：把场景分割成多个 Objects，并实时的对他们进行跟踪和 Shape 建模。

## 5.23 Squeeze-and-Excitation Networks

参考文献：[10]

本文主要的关注点是如何更好的处理 Channel-wise 的信息。普通的卷积操作，会同时考虑局部的区域特征以及不同特征图之间的组合，而本文主要考虑 Channel wise 信息的提取，作者认为这样可以提高模型的表示能力！

具体结构如下图所示：

图 5.23 需要注意以下几点：

- $F_{sq}(\cdot)$  表示文中所说的 Squeeze，这是基于 Global Max Pooling (GAP) 实现的，所以这一步并不会引入额外的参数，只又很少量的计算代价。这一步的结果就是输入的 Feature Map 编程  $1 \times 1 \times C$  的向量，作者把这个 vector 称为 Channel Descriptor。

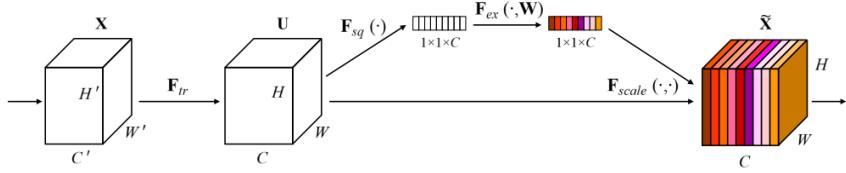


Figure 1: A Squeeze-and-Excitation block.

**Fig 5.23.** SENet 结构示意图

- $F_{ex}(\cdot, \omega)$  表示文中所说的 Excitation，这是由 Sigmoid 和 Bottleneck 结构实现的。Bottleneck 结构是两层具有不同输出维度的全连接层，可以表示成：

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

其中  $\sigma$  为 ReLU， $\mathbf{W}_1 \in R^{\frac{C}{r} \times C}$ ， $\mathbf{W}_2 \in R^{C \times \frac{C}{r}}$ ，其中  $C$  表示输入 Feature Map 的通道数，也就是  $\mathbf{U}$  的通道数，这一结构也称为 Bottleneck，是为了降低参数数量， $r$  是 ratio，作者实验证明了可以选择 16。

- $F_{scale}(\cdot, \cdot)$  表示 Channel-wise 的相乘
- 作者的实验表明，这个 S-and-E Block 放在网络结构的中部、前部比较好，放在后面效果不是特别很好，但会增加比较多的参数，因为后面的通道数  $C$  比较大，所以可以去掉。这是基于 ResNet 验证的。

如何把 SE Block 加入到 ResNet 等网络中，可以看文章。

## 5.24 Deep Multi-scale Architectures For Monocular Depth Estimation

参考文献：

本文主要回顾使用 Multi-scale 的网络结构进行深度估计，表明相比于只使用一种 Scale 的 Feature，可以提高精度、定性上更好的深度图。在 NYU Depth 上实现了 SOTA。

### 5.24.1 背景与相关工作

有学者指出，受人类视觉的激发，发展了一些算法，人类视觉会利用 Texture, Perspective, defocus 等信息。

最早的两篇基于深度学习来做的工作是：

- Deep Deconvolutional Networks for Scene Parsing
- Depth map prediction from a single image using a multi-scale deep network

## 5.25 Depth Map Prediction from a single image usign a multi-Scale Deep Network

---

基于这两个工作，后面提出了非常多的改进方法，但比较少关注多尺度问题。本文的实验表明，如果在小的数据集上，多尺度网络可以比精心设计的单尺度网络更好一点的话，那么在打的数据集上，就会有 SOTA 结果。

相关工作有：

- Depth map prediction from a single image using a multi-scale deep network, 2014  
    网络结构包含两个部分，由两个网络 Stack 而成。第一个网络输出 Coarse depth map，第二个网络对输出的这个粗糙的深度图进行 Refines，后一个网络通过 3 successive 卷积实现，用 Coarse depth map 和 RGB image 作为输入。
- Predicting depth, surface normals and semantic labels with a common multi-scale convolutional architecture. 2015  
    增加了一个新的 CNN 模块，同时多任务预测。
- Towards unified depth and semantic prediction from a single image, 2015  
    使用语义信息帮助深度估计。
- Learning depth from single monocular image using deep convolutional neural fields, 2016
- Deep convolution neural fields for depth estimation from a single image, 2015
- Depth and surface normal estimation from monocular images using regression on deep features and hierarchical crfs. 2015
- Monocular depth estimation using neural regression forest. 2015
- Depth from a single image by harmonizing overcomplete local network predictions, 2016
- Deeper depth prediction with fully convolutional residual network, 2016  
    Encoder-decoder network。实验表明使用 BerHu loss 可以提高精度。是下了 SOTA 的实验结果。

## 5.25 Depth Map Prediction from a single image usign a multi-Scale Deep Network

参考文献：

基于单目图像预测深度需要来自于多个 Cues 的 both global and local 信息。本文的两个主要贡献：

- 提出利用 two network stack 完成深度预测，一个完成 coarse global prediction based on entire image, 另一个完成 refines this prediction locally.
- 提出一个 scale-invariant error，帮助测量深度

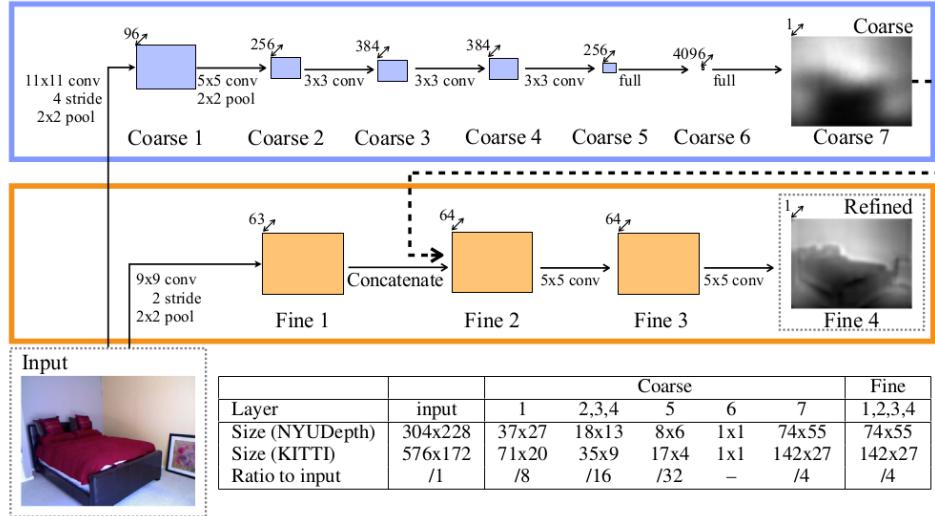


Figure 1: Model architecture.

**Fig 5.24.** 网络结构示意图

整体结构如图5.24所示：

在第一个部分，也就是得到 Coarse Depth Map 的部分，通过全连接层实现输入图像的全局视野，此外，在中间几层使用 Max Pooling 增加局部的感受野。但有一地方是如何根据 Coarse6 的输出得到 Coarse7。作者也没说明白！全连接中还应用了 Dropout。

在第二部分，首先对输入图像进行一次卷积运算，然后结果与上面的 Coarse7 的输出进行 Concatenate，后面的卷积运算都会有 Zero padding 保持输出的大小一致。

第二个贡献是提出了一个 Scale-Invariant Error (Metric)，因为单目预测深度的时候会有尺度问题，所以作者用下面的公式：

$$D(y, y^*) = \frac{1}{2n} \sum_{i=1}^n (\log y_i - \log y_i^* \alpha(y, y^*))^2$$

其中， $\alpha(y, y^*) = \frac{i}{n} \sum_i (\log y_i - \log y_i^*) \cdot n$  为像素的个数， $i$  为像素的索引。

最后作者的目标函数就是基于这个 Scale-Invariant Error 来做的。

*Seethepaper.*

由于网络最后的输出是下采样了 4 倍，所以，在训练的时候需要把 depth 和 mask 都直接下采样比例为 4。

## 5.26 Mask R-CNN

参考文献：[8]

这篇文章的重要贡献是实现了 Instance-Level 的语义分割，具体是实现实在 Faster-RCNN 的基础上，新增了一个 Branch 用于预测 Object Mask。它与物体检测 (Object Detection) 的区别是，提供 Mask 的预测结果而不是 Boundingbox；它与语

## 5.26 Mask R-CNN

义分割的区别是，它是 Instance-level 的分割，而不是像素级的、不区分 Instance 的分割。

示意图如图所示：

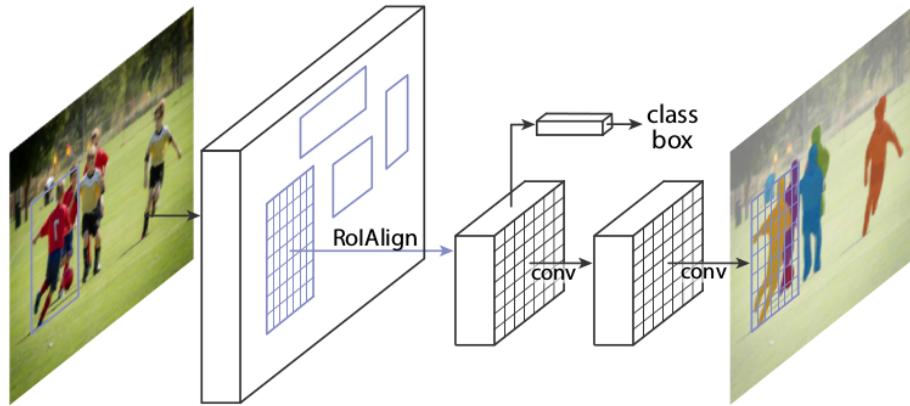


Figure 1. The Mask R-CNN framework for instance segmentation.

**Fig 5.25.** Mask RCNN 示意图，在 class box 的基础上增加了预测 mask 的分支。

### 5.26.1 研究现状与相关背景

在 Faster-RCNN 的基础上实现的，但是由于 Faster-RCNN 中的 ROI Pooling 会导致网络的输出与输入并不是 Pixel-by-pixel 的 Alignment，所以，在得到 Mask 的时候，作者提出利用 ROIAlign 的结构，可以 faithfully preserves exact spatial locations.

非常重要的一点是：mask 和 class 的预测是分离的 (decouple) 的两个过程，而 FCN 之类的算法是既需要 perform per-pixel multi-class categorization, which couples segmentation and classification，后一种方式对 Instance segmentation 的效果较差。

### 5.26.2 Mask RCNN

主要有以下几点需要注意的地方：

- 使用 ROIAlign 层代替原来的 ROI Pooling 层
- 分离 Region Proposal 的分类 +BBox 回归和像素级的语义分割
- 第三条 Branch 输出的二值的图像，然后与 Class Prediction 的结果一起计算误差

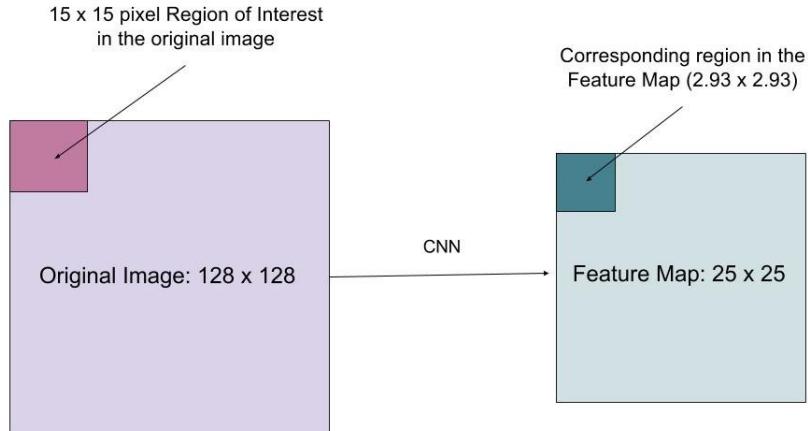
主要想明白这一点：就是如何对齐的问题！

为什么在 SPP Net 和 Fast RCNN 中的 ROI Pooling 会对不齐呢，有两个方面的原因：

- 因方面由于存在 Pooling 的降维计算，会求除法一次

- 在 Spatial Pyramid Pooling 产生固定大小的输入的时候 (在 Mask RCNN 论文里面叫做 bin)，还会求除法一次

那么具体是怎么操作的呢：  
参看文献：CNN 图像分割简史 -知乎



**Fig 5.26.** ROI Align 的示意图, 如何将 region of interest 的原始图像和特征图精确对齐?

假设有一幅 128\*128 大小的图像，对应的特征图大小为 25\*25，如果要将原始图像左上角 15\*15 大小的像素映射到特征图中（如图 16 所示），如何从特征图中选择像素呢？显然原始图像中每个像素对应特征图中的  $25/128$  个像素，要选择原始图像中的 15 个像素，需要选择  $15*25/128=2.93$  个像素。

在 RoIPool 中，我们会四舍五入选择 3 个像素，从而会导致微小的偏移。在 ROIAlign 中，我们要避免这样的近似。取而代之的是采用 bilinear interpolation 获取精确的对应值，即选择像素 2.93，从而能够避免 RoIPool 导致的偏移。

### 5.26.3 实验

使用 Averaged over IoU thresholds 来计算。AP 是 Average Precision 的意思，只不过这里只有当 IoU 与 Ground Truth 重叠到一定程度后才认为是分类正确的，作者用到的： $AP_{50}, AP_{75}$  等。

### 5.26.4 总结

基于 Faster RCNN 实现的 Mask RCNN 可以出人意料简单的解决 Instance Segmentation 问题。

## 5.27 Learning to segment every thing

参考文献：[11]  
本文非常值得再读。  
需要注意的几个亮点：

## 5.27 Learning to segment every thing

- 提出了基于 Transfer Learning 的 Partially supervised instance segmentation 功能，可以实现 3000 个种类的 Instance 的分割。
- 与以往不同的是，作者是在 Mask RCNN 中 Detection Branch Head 部分的权重的基础上进行迁移学习的，学习 Segmentation Head 的权重参数！以往的做法可以是利用 Word2Vec 或 GloVe 等算法把 Feature Map 映射到 Embedding 空间中进行迁移学习，这也是 Zero Shot Learning 中的做法。
- 那么具体是怎么实现从 Detection Head 的圈中参数迁移到 Segmentation Head 部分的权重参数的呢。作者比较了两种思路：利用一个仿射变换实现、利用两三层的神经网络 (MLP)+LeakyReLU 来做。Ablation 测试发现后者更好
- 前面说到了，作者测试了集中迁移结构的输入，包括 Detection Head 的 Feature 输出、NLP 算法输出的 Embedding、DetectionHead 的权重参数，Ablation 实验发现后者更好。
- 需要值得注意的是，作者认为 MLP 的 Head 更适合 Category Agnostic，FCN 更适合 Class-specific 的分割，所以他们两个算法可以互补！
- 训练数据分成了两组，一组具有精确地 Segmentation Mask，一组只有 Bounding box。
- 比较了两种不同的训练策略。Stage Wise，即先训练 Faster RCNN 部分，然后固定参数再训练 Segementation 任务；End-to-End 训练，即整个模型一起训练，但是在训练 Segmentation Head 的样本时，不能对 Detection Head 部分的权重进行更新！
- 存在扩展到 Open Set Recognition 的论文：Bayesian Semantic Instance Segmentation in Open set World.

暂时就只有以上几点。

论文思路的整体框架如图所示：

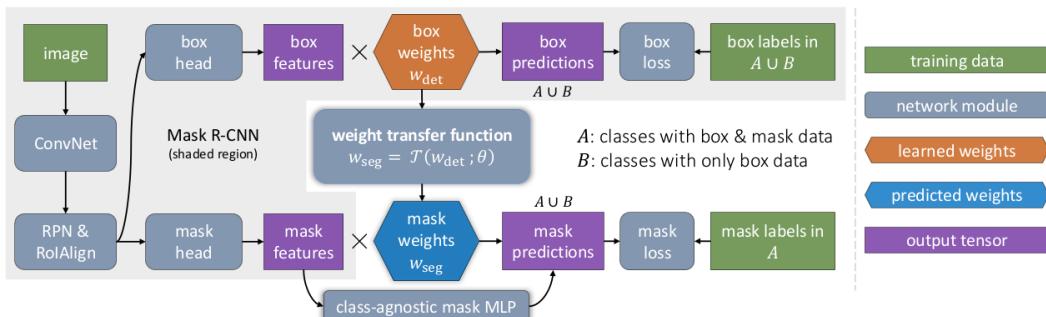


Figure 2. Detailed illustration of our Mask<sup>X</sup> R-CNN method. Instead of directly learning the mask prediction parameters  $w_{seg}$ , Mask<sup>X</sup> R-CNN predicts a category's segmentation parameters  $w_{seg}$  from its corresponding box detection parameters  $w_{det}$ , using a learned weight transfer function  $T$ . For training,  $T$  only needs mask data for the classes in set  $A$ , yet it can be applied to all classes in set  $A \cup B$  at test time. We also augment the mask head with a complementary fully connected multi-layer perceptron (MLP).

**Fig 5.27.** 论文的整体结构，包括 Weight Transfer 结构

其中 Weight Transfer 结构的数学表达式是：

$$\omega_{seg}^c = T(\omega_{det}^c; \theta)$$

其中,  $\omega$  是权重,  $\omega_{det}$  来自于 Mask RCNN 中的 Detection Head, 论文作者通过把用于分类的权重  $\omega_{cls}$  和用于 BBox 回归的权重  $\omega_{box}$  进行 Concatenate 构成  $\omega_{dec}$ 。

## 5.28 Path aggregation network for instance segmentation

参考文献: [17]

根据论文题目也可以看出, 作者主要的改进的是 Path Aggregation. 也就是增加了 Bottom->top 的 Feature Pyramid, 实在 FPN 的基础上实现的, 也就是说本身就包含一条 Top->Bottom 的 Path。

此外, 作者针对 Mask RCNN 中的三个问题, 提出了三个改进的办法!

有时间把论文细节补上。还是有很多值得学习的地方的。

**自从 ResNet 火起来后, 这个 Path Aggregation 的方式貌似用的挺多的, 关键还是让不同 Level 的 feature 更方便的信息交流! 比如在本文中, 低层次的 Feature 也可以支持 Large Object, 高层次的 Feature 也可以支持 Small Object。**

## 5.29 End-to-End Instance Segmentation with Recurrent Attention

参考文献: [25]

比较有意思的几点:

在图像预处理过程中, 作者用 FCN 输出两类数据, 一类是 1 个 Channel 的前景分割; 另一类是每一个 Object 的 Angle Map, 作者把这个角度分成八个方向。为什么也会输出 Angle Map 呢? 因为作者认为: Predicting the angle map forces the model to encode more detailed information about object boundaries。

**所以通过多任务学习、改进 Loss 函数, 可以有效提高多任务中的单个任务的性能。我猜测, 这个多任务肯定得是具有某种同性的作用, 通过多个任务之间的共同的约束来提高单一任务的性能, 或者强迫模型学习得到我们想要的性质!**

现在这个智商啊, 下降的非常厉害。

## 5.30 Fully Convolutional Instance-aware Semantic Segmentation

参考文献: [16]

同时检测、分割 Instances。引入了 Position-sensitive inside/outside score maps, 两个任务的底层卷积表示是完全 Shared 的。

### 5.30.1 背景与相关现状

Extends the translation invariant score maps in conventional FCNs to *position-sensitive* score maps, which are somewhat translation-variant.

也就是说抛弃了以往的 Translation invariant, 而是加上了部分的 Translation Variant 性质(在分割的时候这样做)。

### 5.30.2 Our Approach

为了在 Mask Prediction 中引入 Translation-variant 性质，作者用了一个 Fully Convolution solution 来实现 Instance Mask Proposal。

作者总结到，高性能的原因来自于高度集中和高效的网络结构，尤其是联合训练 Classification 和 Mask Segmentation 的思想。

那么是怎么整合到一起的呢？作者说是通过 Fuses the two answers into two scores. 此外，提高 Position-sensitive score map 的想法来联合的和同时地完成 Object segmentation 和 Detection 两个任务。

所以，在深度学习中，重要的点还包括：网络结构 + 适合自己任务的 Loss Function。对于后者，应该想明白怎样才是适合自己的 Loss Function.

## 5.31 TernausNetV2: Fully Convolutional Network for Instance Segmentation

参考文献：[13]

作者利用 Encoder-Decoder 结构实现了对遥感数据的 Instance-level Semantic Segmentation。输入除了普通的 RGB 数据，还包括其它通道的数据。

### 5.31.1 背景及相关工作

首先提出了 FCN 结构用于语义分割，后来通过引入 Skip Connections 来允许综合 low-level 的 Feature maps 和 higher-leve ones，可以保证精确地 Pixel-level localization。后者的典型是 U-Net。

作者的方案是，在 RGB 上进行迁移学习，输出 Semantic Mask 后在用额外的 Output Channel 用于 Predict areas where objects are touched or close to each other. 最后输出的是 Separate Instances.

用到的数据集是 SpaceNet。

### 5.31.2 Model

采用 Encoder-Decoder 的 U-Net 网络结构，并且作者把 Encoder 部分替换成 WideResNet-38 Network，后面这个结构具有 In-place activated batch normalization 的特点。In-palce activated Batch Normalization 把 Batch Normalization layer 和 Activation Layer 合并到一起了，可以节省 50% 的 GPU 存储空间。

此外，相比于 ResNet, WideResnet 每次卷积具有更多的输出 Channel，同时具有更浅的卷积层数。

而在 Decoder 部分，采用 Nearest Neighbor Upsampling 来实现分辨率的翻倍。

### 5.31.3 Training

再输入中，比 3 个通道多出来的几个通道的网络权重被初始化为 0。

在第一个 Epoch，固定 Encoder 部分的权重，然后只训练 Decoder 部分的权重。在第二个 Epoch 开始，训练所有的权重。作者认为，这样网络可以轻松的从 3 个通

道学习扩展到多个通道的学习，并且学习的过程是: Delicate, careful manner, slowly increasing weights of the multi-spectral part of the input.

作为 Loss function，作者采用了下面的损失函数:

$$L = \alpha H - (1 - \alpha)(1 - J)$$

其中 J 是 Jaccard loss, H 是普通的交叉熵损失函数。

这个 Jaccard loss 具体长下面这样:

$$J = \frac{1}{n} \sum_{c=1}^C \omega_c \sum_{i=1}^n \left( \frac{y_i^c \hat{y}_i^c}{y_i^c + \hat{y}_i^c - y_i^c \hat{y}_i^c} \right)$$

其中， $\hat{y}$  是预测的 Mask， $y$  是真实的 Mask,  $n$  是每一个通道的像素数， $C$  是总的通道数，也就是类别数。

### 5.31.4 常用数据集

### 5.31.5 KITTI

数据集官网: [KITTI Dataset](#)

参考文章:

[1] [KITTI 数据集简介和使用 -CSDN](#)

KITTI 数据集由德国卡尔斯鲁厄理工学院和丰田美国技术研究院联合创办，是目前国际上最大的自动驾驶场景下的计算机视觉算法评测数据集。该数据集用于评测立体图像(stereo)，光流(optical flow)，视觉测距(visual odometry)，3D 物体检测(object detection) 和 3D 跟踪(tracking) 等计算机视觉技术在车载环境下的性能。KITTI 包含市区、乡村和高速公路等场景采集的真实图像数据，每张图像中最多达 15 辆车和 30 个行人，还有各种程度的遮挡与截断。整个数据集由 389 对立体图像和光流图，39.2 km 视觉测距序列以及超过 200k 3D 标注物体的图像组成 [1]，以 10Hz 的频率采样及同步。总体上看，原始数据集被分类为'Road'，'City'，'Residential'，'Campus' 和 'Person'。对于 3D 物体检测，label 细分为 car, van, truck, pedestrian, pedestrian(sitting), cyclist, tram 以及 misc 组成。

### 5.31.6 官网资源介绍

最官网页面顶部，一共分了 13 栏，如图5.28所示。



**Fig 5.28.** KITTI 官网截图

其中，这些栏分别对应不同的应用目的提供单独的数据集，如 Stereo, flow, sceneflow, depth, odometry, object, tracking, road, semantics, row data 等。其中前三个

貌似是对应同一个数据集。在 depth 中会有深度预测和深度补全等功能，semantics 等包含一些标注好的图像数据等。Odometry 等数据集比较大！

此外，作者还提供了一个 Submit results 栏，可以上传自己的测试结果等。但不能造假，即训练过程中，需要自己负责把训练数据分成 train data 和 validate data，然后用 test data 对模型进行测试。

### 5.31.7 详述

主要参考本小节的参考文章。

原始数据采集于 2011 年的 5 天，共有 180GB 数据。

#### 组织形式

图所示为早期的数据组织形式，与目前 KITTI 数据集官网公布的形式不同。在这早期的版本中，一个视频序列的所有传感器数据都存储于 data\_drive 文件夹下，其中 data 和 driver 是占位符，表示采集数据的日期和视频编号。时间戳记录在 Timestamps.txt 文件中。

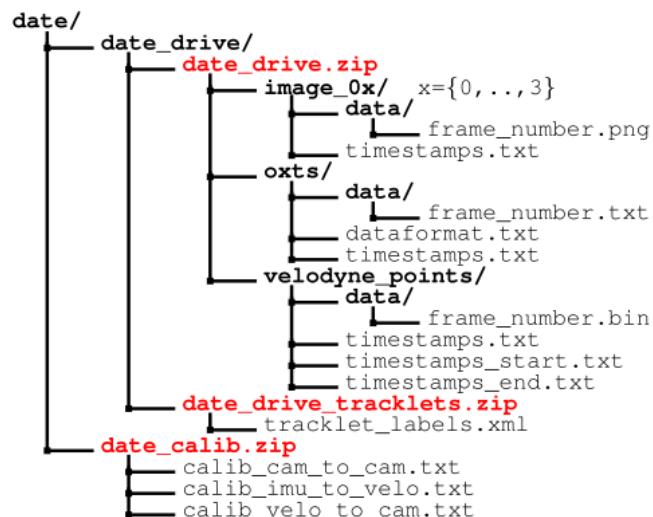


Fig. 4. **Structure of the provided Zip-Files** and their location within a global file structure that stores all KITTI sequences. Here, 'date' and 'drive' are placeholders, and 'image\_0x' refers to the 4 video camera streams.

**Fig 5.29.** 早期的数据文件组织格式

而现在，对于从官网下载的各个分任务的数据集，其文件组织形式比较简单，以 Object Detection 为例，下面是 Object detection Evaluation 2012 标准数据集中 left color images 文件的目录结构，样本分别存储在 testing 和 training 数据集中。

```

data_object_image_2/
| -- testing/
|   | -- image_2/
| -- training/
|   | -- image_2/
|   | -- label_2/
  
```

## Development Kit

KITTI 各个子数据集都提供开发工具 development kit, 主要由 cpp 文件夹, matlab 文件夹, mapping 文件夹和 readme.txt 组成。

cpp 文件夹主要包含评估模型的源代码 evaluate\_object.cpp。Mapping 文件夹中的文件记录训练集到原始数据集的映射, 从而开发者能够同时使用激光雷达点云, gps 数据, 右边彩色摄像机数据以及灰度摄像机图像等多模态数据。Matlab 文件夹中的工具包含读写标签, 绘制 2D/3D 标注框, 运行 demo 等工具。Readme.txt 文件非常重要, 详述介绍了某个子数据集的数据格式, benchmark 介绍, 结果评估方法等详细内容。

### 5.31.8 Cityscape

数据集官网: [Cityscapes Datasets](#)

### 5.31.9 TUM

数据集官网: [TUM Datasets](#)

## 5.32 实例分割 - 图像分割 2018.06.11 博客总结

### 5.32.1 Mask R-CNN 狙击目标实例分割

参考文献: [Mask R-CNN 狙击目标实例分割 - 知乎](#)

#### 什么是实例分割

举例子的方式: 简单讲, 一群人在图片里面, 我希望把每个人都给我分割出来。分类只能做到识别这个图片是人; 目标检测只能检测到这个图片里有人, 把人的地方框出来, 对每一个人这个个体不一样是没有判断的, 统一认为是人; 而图像分割主要是将人和背景分割出来, 而实例分割就是要把每个人清晰的分割出来。

总的来说, Mask R-CNN 是基于 Faster R-CNN 的基础上演进改良而来, FasterR-CNN 并不是为了输入输出之间进行像素对齐的目标而设计的, 为了弥补这个不足, 我们提出了一个简洁非量化的层, 名叫 RoIAlign, RoIAlign 可以保留大致的空间位置, 除了这个改进之外, RoIAlign 还有一个重大的影响: 那就是它能够相对提高 10% 到 50% 的掩码精确度 (Mask Accuracy), 这种改进可以在更严格的定位度量指标下得到更好的度量结果。第二, 我们发现分割掩码和类别预测很重要: 为此, 我们为每个类别分别预测了一个二元掩码。

#### Mask RCNN 介绍

Mask R-CNN 拥有简洁明了的思想: 对于 Faster R-CNN 来说, 对于每个目标对象, 它有两个输出, 一个是类标签 (classlabel), 一个是边界框的 Offset (bounding-box offset), 在此基础上, Mask R-CNN 方法增加了第三个分支的输出: 目标掩码。目标掩码与已有的 class 和 box 输出的不同在于它需要对目标的空间布局有一个更精细的提取。

## 5.33 无需 Proposal 的实例分割论文

### Mask RCNN 的工作机理

在 Faster RCNN 中，相比于 Fast RCNN，多了一个 Stage，那就是 Region Proposal Network。所以在 Faster RCNN 中，整个算法由两个 Stage 构成，第一个 Stage，就是 RPN，这一步产生 ROI，前面也提到了，在 Fast RCNN 中，输入既包括 Image 也包括一个 ROI，所以这里采用 RPN 来生成 ROI。在第二 Stage 中，其实和 Fast RCNN 是一样的，都是借助 ROI Pool 来完成目标检测（目标分类 + Bounding box offsets）。

那么在 Mask RCNN 中，在第二步骤中，与预测目标类别与 Offset 平行的是还会对每一个 ROI 输出一个 Binary mask。所以 Mask 的 branch，会输出一个  $K*m*m$  大小的张量，其中  $K$  表示类别的个数。在实现过程中，

相应的，在训练的时候，也会采用一个多任务 Loss Function：

$$L = L_{cls} + L_{box} + L_{mask}$$

前两个 Loss 和 Fast RCNN 中是一样的，后面的 mask 的 loss。有很多不懂的地方啊。

### Tensorflow + Keras 实现实例分割

参考文献：[Tensorflow+Keras 实现实例分割 -微信公众号](#)

分割掩码是输入目标的空间布局编码，不像类别标签或边界框偏移量那样将特征经过全连接层“坍缩”到短的输出向量，而是自然地采用卷积实现“像素到像素”的变换，抽取掩码空间结构信息。使用 FCN 网络为每个 RoI 预测一个  $m \times m$  掩码，这样允许它们保持  $m \times m$  目标空间布局信息。

Mask RCNN 中使用 RoIAlign 替换了 RoIPool，RoIPool 没有考虑保留尽可能多的空间布局信息，难以实现输入图片/输出掩码像素一一对应。

## 5.33 无需 Proposal 的实例分割论文

参考文献：[一文介绍 3 篇无需 Proposal 的实例分割论文 -知乎](#)

基于 Proposal(如：Mask R-CNN, MaskLab, PANet) 的实例分割有三个根本缺陷：

- 两个物体可能共享同一个或者非常相似的边界框，此时，Mask Head 无法区分要从边界框中拾取的对象。这对于其所在的边界框中具有低填充率的线状物体（如自行车和椅子）而言是非常严重的问题。
- 架构中没有任何能够阻止两个共享像素的东西的存在
- 实例的数量通常首先与网络能够处理的 Proposal 数量

前两点是什么意思？

文章列出了三篇通过距离学习的 Instance Segmentation，貌似文章的作者称之为实例嵌入。

- 基于判断损失函数的语义实例分割，使用了非成对的损失函数。使用图像中所有像素产生了特别丰富的梯度。

Semantic Instance Segmentation with a Discriminative Loss Function

- 基于深度度量学习的实例语义分割，引入了种子模型，同时帮助我们分类并拾取最佳种子，做了速度优化。

Semantic Instance Segmentation via Deep Metric Learning

- 用于实例分组的递归像素嵌入，GBMS 是均值漂移的一种变体，在网络内部用于训练和解析。创建了非常密集的聚类。  
(用于实例分组的递归像素嵌入)

他们比 Proposal 方法更简单，也可能更快，同时避免了 Proposal 的实例分割架构存在的三个根本缺陷。但实际效果并不如 Proposald 的方法。

此外，作者还给出了一些其它的方法的论文：

- 基于循环注意力机制的端到端实例分割（End-to-End Instance Segmentation with Recurrent Attention）
- 用于实例分割的深分水岭变换（Deep Watershed Transform for Instance Segmentation）
- 联合嵌入：用于联合检测和分组的端到端学习（Associative Embedding: End-to-End Learning for Joint Detection and Grouping）
- 用于实例分割的序列分组网络（SGN: Sequential Grouping Networks for Instance Segmentation）

## 5.34 Learning Rich Features from RGB-D Images for Object Detection and Segmentation

参考文献：[7]

主要的几点：

- 使用预训练的网络对提取 RGB 图像、Depth 图像的特征都是非常重要的
- 由于 Depth 是单通道的，作者把这个单通道的扩展成：Horizontal Disparity, Height above ground, 和 Angle with Gravity 三个通道
- 使用 R-CNN 框架完成目标识别，使用决策树完成实例分割

## 5.35 Multimodal Deep Learning for Robust RGB-D Object Recognition

参考文献：[3]

值得注意的几点：

- 提出了一种基于 NiN 和 GoogLeNet 结构的 RGBD 处理框架以及一种有效的初始化方法。

### 5.36 3D Graph Neural Networks for RGBD Semantic Segmentation

---

- 具体的，输入的 Depth 为单通道，同时 depth 的 NiN 结构的 Channel 数都减少为原来的 1/3 并被随机初始化。Depth 分支学习到的 Filter 与 RGB 分支学习到的 Filter 作者认为具有很大的不同。
- 对于 RGB 通路，采用 GoogLeNet 结构，Depth 的 Feature 通过 Concatenate 来引入到 RGB 分支中
- 对比 Early fusion, Mid fusion, late fusion，最后发现 a mid-level fusion after 2NiN modules leads to the best results
- 对比了几种对 Depth 分支的初始化策略：随机初始化、使用在 RGB 上的训练权重、使用 CityScape 预训练。结果表明：降低 channel 数为原来的 1/3，并采用 CityScape 与训练效果最好！
- 使用 CityScape 训练集

## 5.36 3D Graph Neural Networks for RGBD Semantic Segmentation

## 5.37 Multi-Path Refinement Networks for High-resolution semantic segmentation

时间：2018.07.09

需要注意的几点：

- 提出了一种利用多层中间层信息的网络结构
- 引入了一种新的结构，叫做：Chained residual pooling，一个比较重要的点是在进行具体 Pooling 之前，引入一个 RELU 层，作者说这样可以让后面的 Pooling 更高效，同时学习过程对学习率更鲁棒
- 包含 Short-range 和 long-range residual connections.
- 在 Pascal Voc 2012, SUN RGBD, Cityscape 等七个数据集上结果很好。用到的评估包括：mIoU, Pixel Accuracy, mAP 等。在测试的时候，还测试了不同 Scale 的输入情况下的平均表现。

### 5.37.1 引言中的重要几点

- 在 CNN 结构中，在底层部分，分辨率较高，但都是一些低层次的特征，如边缘等；在高层部分，具有叫高层的语义信息，但分辨率较低。一种不降低分辨率但提高感受野的技术是 Dilated Convolution。
- Dilated Convolution: 缺点，占用 GPU 显存较大

5.37.2 算法思想

5.37.3 实验部分

5.37.4 小结

**5.38 RedNet: Residual Encoder-Decoder Network for indoor RGB-D Semantic Segmentation**

**5.39 RDFNet: RGB-D Multi-level Residual Feature fusion for Indoor Sematnic Segmentation**

**5.40 Progressively Complementarity-aware Fusion Network for RGB-D Salient Object Detection**

**5.41 Semantic-Guided Multi-level RGB-D Feature Fusion for Indoor Semantic segmentation**

**5.42 Learning Common and Specific Features for RGB-D Semantic Segmentation with Deconvolutional Networks**

# Chapter 6

## Open Set Recognition

Open set recognition with GAN & OpenMax.

### 6.1 GAN

#### 6.1.1 GAN 原理笔记

参考文献: [GAN 原理学习笔记 -知乎](#)

传统的生成模型,如自编码机(Auto-Encoder),通常采取MSE作为Loss Function,这样的弊端是学习得到的Decoder模块性能不太令人满意。

#### GAN 原理

首先,真实数据的分布已知, $P_{data}(x)$ ,我们需要做的就是生成一些也在这个分布内的图片,但无法直接利用这个分布。

刚才讨论过了,MSE的损失函数可能效果较差,改进办法是用交叉熵(Cross Entropy)来计算损失,从下面的推导可以看出,交叉熵的最大化等价于KL散度的最小化。KL散度衡量的是模型分布与真实数据分布之间的差异。

模型分布用 $P_G$ 表示,数据分布用 $P_{data}$ 表示。

$$\hat{\theta} = \arg \max_{\theta} \prod_{i=0}^m P_G(y_i|x_i, \theta) \quad (6.1)$$

$$= \arg \max_{\theta} \sum_{i=0}^m \log P_G(y_i|x_i, \theta)$$

$$= \arg \max_{\theta} E_{x \sim P_{data}} \log \log P_G(y_i|x_i, \theta) \quad (6.2)$$

$$= \arg \max_{\theta} \int_x P_{data}(x) \log P_G(y_i|x_i, \theta) dx - \int_x P_{data}(x) \log P_G(y_i|x_i, \theta) dx$$

$$= \arg \min_{\theta} \int_x P_{data}(x) \log \frac{P_{data}(x)}{P_G(x; \theta)} dx \quad (6.3)$$

$$= \arg \min_{\theta} KL(P_{data}(x), P_G(x; \theta)) \quad (6.4)$$

其中,  $m$  表示从训练数据中采样的样本数。主要难理解的地方在于公式 (6.2) 中的后一项, 由于这一项与  $\theta$  无关, 所以加上之后也不会影响  $\arg \max_{\theta}$  运算的取值。

## GAN 公式

$$V(G, D) = E_{x \sim P_{data}} [\log D(x)] + E_{x \sim P_G} [1 - \log D(x)] \quad (6.5)$$

优化目标是:

$$G^* = \arg \min_G \max_D V(G, D) \quad (6.6)$$

下面解释上面的两个式子。

$D$  的作用是让这个式子尽可能的大。对于第一项, 在输入  $x$  来自于真实数据时为了使  $V$  最大,  $D(x)$  应该接近于 1; 对于第二项, 在输入  $x$  来自于  $G$  的生成时, 则应该使  $D(x)$  尽可能的接近于 0。

## 训练过程

根据公式 6.5 与 6.6, GAN 的训练过程是  $G$  与  $D$  相互迭代更新的过程。具体如下: 注意, 可能更新多次  $D$  次之后才更新一次  $G$ 。

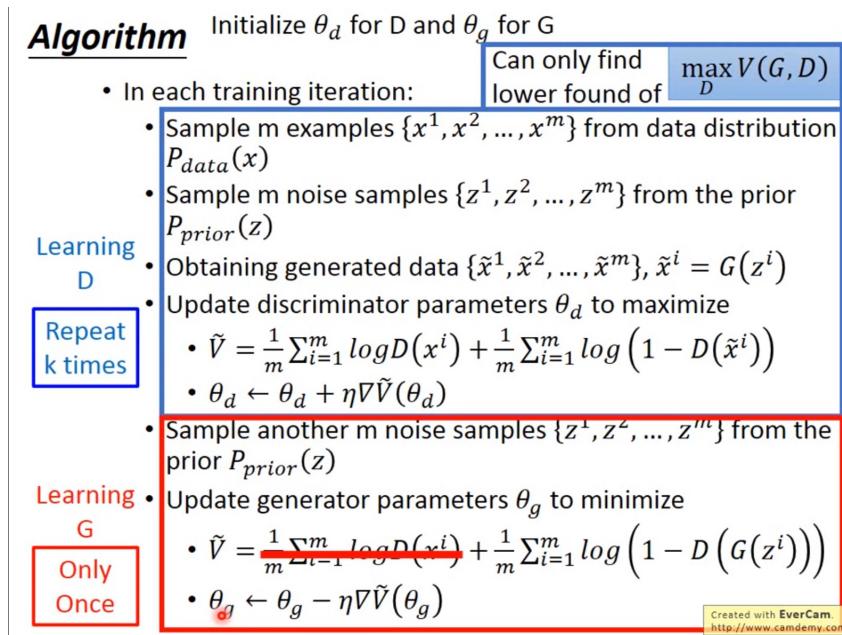


Fig 6.1. GAN 训练过程

给定  $G$ , 首先推导  $D$  的最优解。

$$\begin{aligned} V(G, D) &= E_{x \sim P_{data}} [\log D(x)] + E_{x \sim P_G} [1 - \log D(x)] \\ &= \int_x P_{data}(x) \log D(x) dx + \int_x P_G \log(1 - D(x)) dx \\ &= \int_x P_{data}(x) \log D(x) + P_G(x) \log(1 - D(x)) dx \end{aligned} \quad (6.7)$$

## 6.2 从头开始 GAN

---

在给定的  $x$  时，对上式中  $D$  的最大化，等价于：

$$\arg \max_D \left[ \underset{a}{P_{data}(x)} \log \underset{D}{D}(x) + \underset{b}{P_G(x)} (1 - \log \underset{D}{D}(x)) \right]$$

另

$$f(D) = a \log D + b \log(1 - D) \quad (6.8)$$

对式6.8进行求导并另其为 0 得到最优  $D$  的表达式：

$$\begin{aligned} D^*(x) &= \frac{a}{a+b} \\ &= \frac{P_{data}(x)}{P_{data}(x) + P_G(x)} \end{aligned}$$

把上式关于  $D$  的最优解代入6.5可以得到以下公式：

$$\begin{aligned} \max_D V(G, D) &= V(G, D^*) \\ &= \int_x P_{data}(x) \log \frac{P_{data}(x)}{P_{data}(x) + P_G(x)} dx + \int_x P_G(x) \log \frac{P_G(x)}{P_{data}(x) + P_G(x)} dx \\ &= -2\log 2 + KL(P_{data}(x) \parallel \frac{P_{data}(x) + P_G(x)}{2}) + KL(P_G(x) \parallel \frac{P_{data}(x) + P_G(x)}{2}) \\ &= -2 \log 2 + 2JSD(P_{data}(x) \parallel P_G(x)) \end{aligned}$$

其中， $JS$  散度是  $KL$  散度的平滑版本，表示两个分部之间的差异。所以固定  $G$  时， $\max_D V(G, D)$  表示两个分布之间的差异，最小值是  $-2 \log 2$ ，最大值是 0。当  $P_G(x) \equiv P_{data}(x)$ ， $G$  是最优的。

### Loss Function 中的两个小问题

- 修改  $G$  的 Loss Function

现有的  $G$  的 Loss Function 中的  $\log(1 - D(x))$  在  $D(x)$  趋近于 0 时，梯度非常小。所以在开始训练时，十分缓慢，一种改进办法是将其更改为： $\min_V = -\frac{1}{m} \sum_{i=1}^m \log(D(x^i))$

- Mode Collapse

这是由于  $KL$  散度的不对称引起的，一种办法是将  $KL$  求解的顺序取反。即： $KL(P_G \parallel P_{data})$  更改为  $KL(P_{data} \parallel P_G)$

$$KL(P_{data} \parallel P_G) = E_{x \sim P_{data}} \log P_G(x)$$

Time: 2018.05.21

## 6.2 从头开始 GAN

参考文献：从头开始 GAN 知乎

### 6.2.1 定义

Ian Goodfellow 自己的话说 GAN[6]:

The adversarial modeling framework is most straightforward to apply when the models are both multilayer perceptrons. To learn the generator's distribution  $p_g$  over data  $x$ , we define a prior on input noise variables  $p_z(z)$ , then represent a mapping to data space as  $G(z; g)$ , where  $G$  is a differentiable function represented by a multilayer perceptron with parameters  $g$ . We also define a second multilayer perceptron  $D(x; d)$  that outputs a single scalar.  $D(x)$  represents the probability that  $x$  came from the data rather than  $p_g$ . We train  $D$  to maximize the probability of assigning the correct label to both training examples and samples from  $G$ . We simultaneously train  $G$  to minimize  $\log(1 - D(G(z)))$ .

简单的说，GAN 包含以下三个主要元素：

- 两个网络：一个生成网络  $G$ ，一个判别网络  $D$
- 训练误差函数：
  - G Net:  $\log(1 - D(G(z)))$   
希望  $D(G(z))$  趋近于 1。
  - D Net:  $-(\log D(x) + \log(1 - D(G(z))))$   
 $D$  网络是一个二分类，希望真实数据的输出趋近于 1，而生成数据的输出即  $D(G(z))$  趋近于 0。
- 数据输入：G Net 的输入是随机噪声，D Net 的输入是混合 G 的输出与真实数据样本的数据

值得注意的地方在于，训练 D 时，输入数据同时来自于真实数据  $x$  以及生成数据  $G(z)$ 。且不用 Cross Entropy 的原因是，如果使用 CE，会使  $D(G(z))$  变为 0，导致没有梯度，而 GAN 这里的做法是让  $D(G(z))$  收敛到 0.5。

实际训练中，G Net 使用了 ReLU 和 Sigmoid，而 D Net 中使用了 MaxOut 和 DropOut，并且修改了 G Net 的 Loss Function，后一点可参考上一节。

但作者指出，此时的 GAN 不容易训练。

### 6.2.2 DCGAN: Deep Convolution GAN

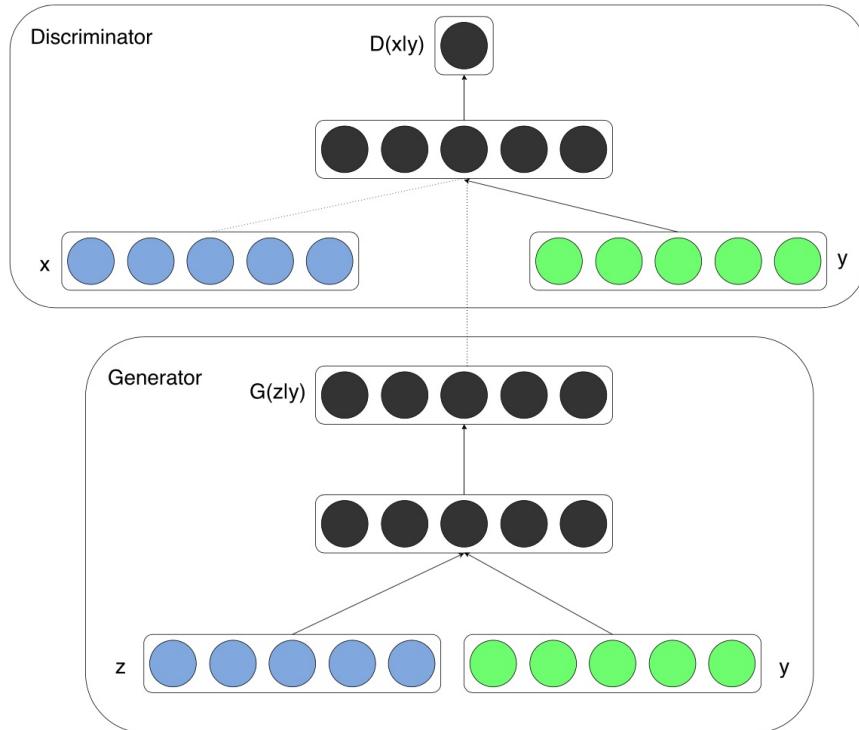
DCGAN 的几点改造：

- 去掉了 G 网络和 D 网络中的 Pooling Layer
- 在 G 网络和 D 网络中都是用 BN
- 去掉全连接的隐藏层
- 在 G 网络中去除最后一层 ReLU，改用 Tanh
- 在 D 网络中每一层使用 LeakyReLU

G 网络使用了 4 层反卷积，而 D 网络使用了 4 层卷积。基本上，G 网络和 D 网络的结构正好反过来的。在使用 DCGAN 生成图像的研究线上，最新到了 BEGAN(十个月以前)，达到了以假乱真的效果。

### 6.2.3 CGAN: Conditional Generative Adversarial Nets

此时，输入不再仅是随机的噪声。就是在 G 网络的输入在 z 的基础上连接一个输入 y，然后在 D 网络的输入在 x 的基础上也连接一个 y：



**Fig 6.2.** CGAN 示意图，在 G、N 网络中新增了数据 y

相应的目标函数变为：

$$\arg \min_G \max_D V(D, G) = \mathbb{E}_{x \sim P_{data}} [\log D(x|y)] + \mathbb{E}_{x \sim P_G} [\log(1 - D(G(z|y)))]$$

训练方式几乎就是不变的，但是从 GAN 的无监督变成了有监督。只是大家可以看到，这里和传统的图像分类这样的任务正好反过来了，图像分类是输入图片，然后对图像进行分类，而这里是输入分类，要反过来输出图像。显然后者要比前者难。

### 6.2.4 InfoGAN

在 CGAN 的基础上，将其变为无监督学习过程。要实现无监督的 CGAN，意味着需要让神经网络不但通过学习提取了特征，还需要把特征表达出来。

怎么做呢？作者引入了信息论的知识，也就是 mutual information 互信息。作者的思路就是 G 网络的输入除了 z 之外同样类似 CGAN 输入一个 c 变量，这个变量一开始神经网络并不知道是什么含义，但是没关系，我们希望 c 与 G 网络输出的 x 之间的互信息最大化，也就是让神经网络自己去训练 c 与输出之间的关系。

Mutual Information 在文章中的定义如下:

$$I(c, G(z, c)) = \mathbb{E}_{c \sim P(c), x \sim G(z, c)} [\log Q(c|X) + H(c)]$$

其中,  $H$  为熵运算。 $Q$  网络则是反过来基于  $X$  输出  $c$ 。基于上式定义的  $I$ , 则整个 GAN 的训练目标变为:

$$\min_G \max_D V(D, G) - \lambda I(c, G(z, c))$$

相比于 CGAN, InfoGAN 又做了如下改变:

- D 网络的输入只有  $x$ , 不加  $c$
- Q 网络和 D 网络共享同一个网络, 只是到最后一层独立输出

## 6.3 Generative Adversarial Nets

参考文献: [6]

**评语:** 需要重点关注, 提出一个新的网络后, 如何在数学上证明是可行的呢?

## 6.4 Towards Open Set Deep Networks

引入了一种新型的新型的神经网络层结构, OpenMax, 可以评估输入来自于一个未知类的概率。其中一个重要的组成部分是采用 Meta-Recognition 来对网络的 Activation patterns 进行处理。

比对 SoftMax 的结果进行 Thresholding 要好很多。(为什么会好?)

这是因为, 实际使用中, 即使输入是非常奇怪的, 也就是说"fooling" or "Rubbish" images 时, 网络也可能会在某一类上产生很高的输出概率, 所以这时候通过设置 Threshod 的方式是不合适的。

They strongly suggests that thresholding on uncertainty is not sufficient to determine what is unknown.

### 6.4.1 Introduction & Related Works

In Sec. 3, we show that extending deep networks to threshold SoftMax probability improves open set recognition somewhat, but does not resolve the issue of fooling images.

Thresholding 可能会在某些时候会帮助 Open Set Recognition, 但对于 Fooling images, 结果可能比较差。

## 6.5 Probability Models for Open Set Recognition

本文的主要目的是, 在限制 Open Space Risk 的分类算法中, 使其能完成非线性多分类的功能。

此外, 提出了 Compact Abating Probability (CAP), 也就是说, 随着距离 Known Data 越来越远, 即距离 Open Space 越来越近的时候, 样本分类的概率会降低。

## 6.6 Meta Recognition

---

提出了具体的 Weibull-calibrated SVM 算法，该算法基于 EVT 原理进行 Score Calibration，以及线性 SVM 来实现。

本文的重点：

- 提出了 Compact abating probability (CAP)
- 提出了 Weibull-calibrate SVM (W-SVM):
  - CAP
  - Statistic Extreme Value Theory (EVT)
- 在 Caltech 256 & ImageNet 上也有对比试验

### 6.5.1 背景及相关工作

## 6.6 Meta Recognition

主要的思想：作者把网络的输出称为 scores，并用 Extreme Value Theory 来对每个类别的 scores 进行分析，发现每一类的 scores 都符合 Weibull 分布!!!

所以这里的 Meta Recognition 的思想就比较明显了，就是判断当前样本的 score 是否符合其中某一类的 Weibull 的分布，如果都不符合，那么就可以认为这个样本就是未知的了，从这个角度来看，Meta Recognition 的数学基础应该是度量分布之间的距离!!!

### 6.6.1 该死的 Fisher-Tippet Theorem

也被称为 Fisher-Tippett-Gnedenko theorem 或者 Extreme Value Theorem。

与中心极限定理的关系，Fisher-Tippet theorem 分析的是最大值的极限分布，而中心极限定理 (centric limit theorem) 分析的均值的极限分布，并收敛到近似高斯分布。

Fisher-Tippet Theorem(Wikipedia):

The maximum of a sample of iid(独立同分布) random variables after proper renormalization can only converge in distribution to one of 3 possible distributions: the Gumbel distribution, the Frechet distribution, or the Weibull distribution.

这句话的意思是：对于来自独立同分布随机变量的采样，所有采样的最大值在经过合适的正则化后，只会收敛到 3 中可能的概率分布。

定理内容如下：

有独立同分布的随机变量： $X_1, X_2, X_3, \dots$ ，他们的最大值为： $M_n = \max X_1, \dots, X_n$ 。如果存在一个序列对： $(a_n, b_n)$ ，满足  $a_n > 0$  且

$$\lim_{n \rightarrow \infty} P\left(\frac{M_n - b_n}{a_n} \leq x\right) = F(x)$$

其中， $F$  是不退化的分布函数，那么这个分布  $F$  属于上述三种分布之一。可以被归为 Generalized extreme value distribution 一类。

来自文章的一句话：EVT is analogous to a central limit theorem, but tells us what the distribution of extreme values should look like as we approach the limit.

## Extreme Value Theory

[Extreme Value Theory - Wikipedia](#)

Extreme Value Theory 是处理远离概率分布中值的极端情况下的数据理论。

对于给定的有序的随机变量的样本，EVT 用于评估出现相对于已有 Observation 的非常极端的情况的概率。如发生百年一遇的洪灾的概率。

Wikipedia 的后面就提到 Fisher-Tippet Theorem 了。

### 6.6.2 Weibull Distribution

参考文献： Wikipedia

Weibull distribution 公式如下：

$$f(x; \lambda, k) = \begin{cases} \frac{k}{\lambda} \left(\frac{x}{\lambda}\right)^{-(x/\lambda)^k} & \text{if } x \geq 0, \\ 0 & \text{if } x < 0 \end{cases}$$

其中， $k > 0$  是 shape parameter,  $\lambda > 0$  是 scalar parameter。

Weibull Distribution 的 CDF(Cumulative distribution function):

当  $x \geq 0$  时：

$$F(x; k, \lambda) = 1 - e^{-(x/\lambda)^k}$$

对于  $x = 0$  时， $F(x; k, \lambda) = 0$

# Chapter 7

## MXNet

参考文献: MXNet Architecture

Time: 2018.05.18

### 7.1 MXNet System Overview

今天把这个 Overview 补上。

#### 7.1.1 MXNet System Architecture

MXNet 系统主要的 Modules 包含:

- Runtime Dependency Engines: 根据程序的读写依赖对它们进行调度和执行
- Storage Allocator: 用于高效的分配和回收 CPU 和 GPU 上的存储空间
- Resource Manager: 管理 Global 资源, 比如随机数发生器等
- NDArray: 动态的、异步的 n 维 array, 可以为 MXNet 提供方便的命令式编程
- Symbolic Execution: 静态的符号图执行器, 可以提供搞笑的符号计算图执行和优化
- Operator: 定义了前向和反向传播计算的计算子
- SimpleOp: 以同意的风格扩展 NDarray 计算子和符号计算子
- Symbol Construction: 提供一种定义计算图的方法
- KVStore: Key-Value 存储接口, 用于高效的参数同步
- Data Loading (IO): 高效的分布式数据加载和增广工具

## 7.1.2 MXNet System Components

### Execution Engine

我们不仅可以使用 mxnet 的 engine 用于深度学习，而且还可以用于其他领域特定的问题。它主要用于执行一些具有依赖关系的 functions。任何有依赖关系的两个函数只能顺序执行，而那些没有依赖关系的计算可以并行执行。

```
using Fn = std::function<void(RunContext)>
```

所以，函数的类型是void(RunContext)。另外一个需要注意的地方是，这里的RunContext与Context的区别，前者包含了只有在运行时才能确定的一些信息，比如函数执行的stream；后者包含了device type与device看文档，ImageFolderDataset的说明文档。其中

另外一个变量是VarHandle，这个主要用于指定function的依赖。在const\_vars用于存储那些只读变量，而mutate\_vars用于指定那些可以修改的变量。

### Operators in MXNet

在 MXNet 中，每一个操作都是一个类，包含了实际运算逻辑和一些辅助信息，这些辅助信息可以帮助确定可以优化的地方。建议让自己熟悉mshadow库，因为所有的计算都是mshadow::Tblob这一tensor-like的结构进行的。MXNet里面的运算接口可以允许：

- 指定 inplace 更新来降低 memory allocation cost
- 隐藏 Python 中的一些临时变量，让其更干净
- 定义输入与输出张量的关系，可以让系统帮助 shape 检查
- 索取更多的临时变量用于计算，比如 cudnn 等

### Operator Interface

Forward 是核心算子接口，此外还有 Backward 接口。

Listing 7.1: Image colvotion based on Shared memory and Apron

```
virtual void Backward(const OpContext &ctx,
                      const std::vector<TBlob> &out_grad,
                      const std::vector<TBlob> &in_data,
                      const std::vector<TBlob> &out_data,
                      const std::vector<OpReqType> &req,
                      const std::vector<TBlob> &in_grad,
                      const std::vector<TBlob> &aux_states);
```

在 Backward 中，输入数据时 out\_grad, in\_data, 和 out\_data, 同时计算 in\_grad 作为函数的结果。有些操作可能不是同时需要上述三个参数，此时可以通过OperatorProperty里面的DeclareBackwardDependency 接口来设置实际的依赖关系。

### Operator Property

上面说完了两个最重要的核心 Operator Interface，在这里，就开始说一下 Operator Property 了。

单独设置 Operator Property 的目的：有时候卷积操作可能有多种不同的实现方式，并且想要通过切换它们来实现最高的性能。因此，我们把 Operator 的 semantic interfaces 和 Implementation interface (Operator 类) 分开，并把前者在 OperatorProperty 类中。OperatorProperty 主要由以下几个部分构成：

- InferShape

这个函数有两个目的：

- 告诉系统输入、输出 tensor 的 size，因此系统可以在实际调用 Forward 和 Backward 之前就可以分配空间了；
- 在实际运行前，先进行 shape 的检查，以避免一些明显的错误。同时如果没有足够的信息用于推断出 shape 时会返回 false，如果 shape 不一致，那么就会 throws an error

- RequestResources

有一些操作如 cudnnConvolutionForward 在计算过程中需要额外的一些空间。MXNet 定义了两个函数分别用于 Forward 和 Backward 过程的资源请求，资源请求的结果会保存在 ctx 中，然后传入 Operator 中的 Forward 和 Backward 函数。具体做法如下：

Listing 7.2: Image colvotion based on Shared memory and Apron

```
auto tmp_space_res = ctx.requested[kTempSpace].get_space(  
    some_shape, some_stream);  
auto rand_res = ctx.requested[kRandom].get_random(some_stream);
```

- Backward dependency

用于声明在反向传播过程中所依赖的变量，这样的话，如果发现不需要的变量，那么就可以及早把资源释放掉。

- In place Option

为了进一步节省存储的消耗，可以使用 in-place update。这个策略适合 element-wise 类的操作，并且输入、输出有相同的 shape。通过 ForwardInplaceOption 和 BackwardInplaceOption 来实现，这两个函数的返回值说明是可以 inplace 的两个变量。

需要注意的地方是，及时你用到了这些设置，但也不能保证结果是确实是 inplace 的。实际上，这只是对系统的一个建议，而不是强制性的，系统还会考虑一些其它因素，但这些因素对开发者是不可见的，因此对于 Forward 和 Backward 的实现可以不用考虑这些隐含的策略。

- Expose Operator to Python

由于 C++ 的限制，必须实现以下接口：

Listing 7.3: Image colvotion based on Shared memory and Apron

```
// initial the property class from a list of key-value string
// pairs
    virtual void Init(const vector<pair<string, string>> &
                      kargs) = 0;
    // return the parameters in a key-value string map
    virtual map<string, string> GetParams() const = 0;
    // return the name of arguments (for generating
    // signature in python)
    virtual vector<string> ListArguments() const;
    // return the name of output values
    virtual vector<string> ListOutputs() const;
    // return the name of auxiliary states
    virtual vector<string> ListAuxiliaryStates() const;
    // return the number of output values
    virtual int NumOutputs() const;
    // return the number of visible outputs
    virtual int NumVisibleOutputs() const;
```

## Create an Operator from the Operator Property

Operator Property 不仅提供一下 Semantic 属性，而且还负责创建 Operator 指针，用于实际计算。

待续...

## 7.2 Optimizing Memory Consumption in DL

Over the last ten years, a constant trend in deep learning is towards deeper and larger networks. Despite rapid advances in hardware performance, cutting-edge deep learning models continue to push the limits of GPU RAM. So even today, it's always desirable to find ways to train larger models while consuming less memory. Doing so enables us to train faster, using larger batch sizes, and consequently achieving a higher GPU utilization rate.

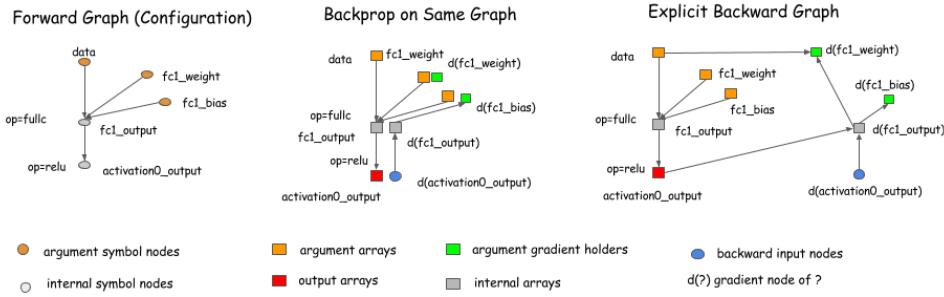
### 7.2.1 Computation Graph

A computation graph describes the (data flow) dependencies between the operations in the deep network. The operations performed in the graph can be either fine-grained or coarse-grained.

The concept of a computation graph is explicitly encoded in packages like Theano and CGT. In other libraries, computation graphs appear implicitly as network configuration files. The major difference in these libraries comes down to how they calculate gradients. There are mainly two ways: performing back-propagation on the same graph or explicitly representing a backwards path to calculate the required gradients.

Libraries like Caffe, CXXNet, and Torch take the former approach, performing back-prop on the original graph. Libraries like Theano and CGT take the latter approach, ex-

## 7.2 Optimizing Memory Consumption in DL



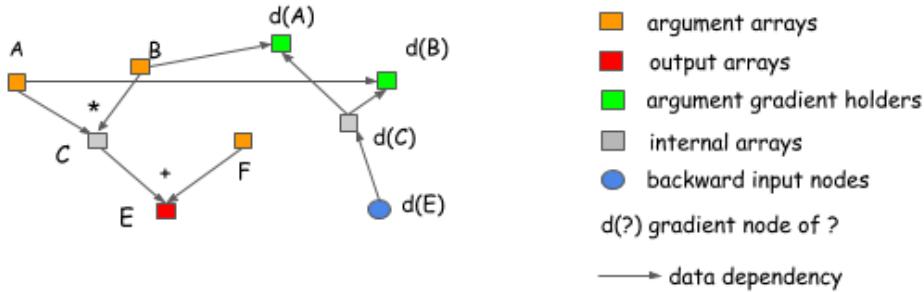
**Fig 7.1.** The implicitly & explicitly back-propagation on Graph

implicitly representing the backward path. In this discussion, we adopt the explicit backward path approach because it has several advantages for optimization.

We adopt the explicit backward path approach because it has several advantages for optimization.

Why is explicit backward path better? Two reasons:

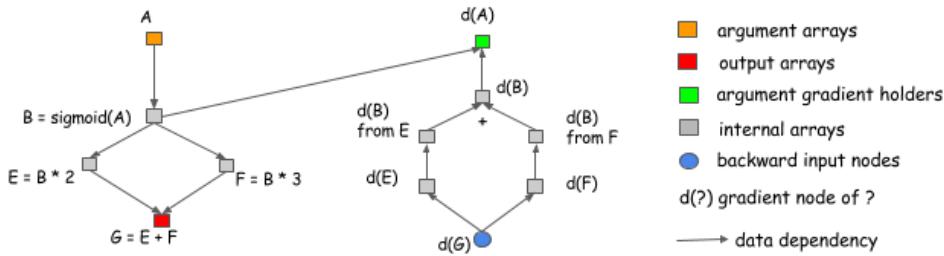
- The explicit backward path clearly describes the dependency between computations. Like the following case, where we want to get the gradient of **A** and **B**. As we can see clearly from the graph, the computation of the  $d(C)$  gradient doesn't depend on **F**. This means that we can free the memory of **F** right after the forward computation is done. Similarly, the memory of **F** can be recycled.



**Fig 7.2.** Dependencies can be found quickly.

- Another advantage of the explicit backward path is the ability to have a different backward path, instead of a mirror of forward one.

A common example is the split connection case, as shown in the following figure. In this example, the output of **B** is referenced by two operations. If we want to do the gradient calculation in the same network, we need to introduce an explicit split layer. This means we need to do the split for the forward pass, too. In this figure, the forward pass doesn't contain a split layer, but the graph will automatically insert a gradient aggregation node before passing the gradient back to **B**. This helps us to save the memory cost of allocating the output of the split layer, and the operation cost of replicating the data in the forward pass.



**Fig 7.3.** Different backward path from forward path.

### 7.2.2 What Can be Optimized?

As you can see, the computation graph is a useful way to discuss memory allocation optimization techniques. Already, we've shown how you can save some memory by using the explicit backward graph. Now let's explore further optimizations, and see how we might determine reasonable baselines for benchmarking.

Assume that we want to build a neural network with  $n$  layers. Typically, when implementing a neural network, we need to allocate node space for both the output of each layer and the gradient values used during back-propagation. This means we need roughly  $2n$  memory cells. We face the same requirement when using the explicit backward graph approach because the number of nodes in a backward pass is roughly the same as in a forward pass.

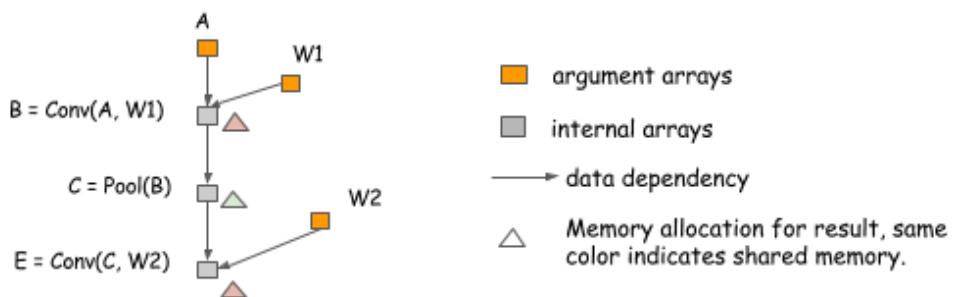
#### In-place Operations

One of the simplest techniques we can employ is *In-place memory sharing* across operations. For neural networks, we can usually apply this technique for the operations corresponding to activation functions.

"In-place" means using same memory for input and output. But you should be careful about that the result is used by more than one operation!

#### Standard Memory Sharing

In-place operations are not the only places where we can share memory. In the following example, because the value of  $B$  is no longer needed after we compute  $E$ , we can reuse  $B$ 's memory to hold the result of  $E$ .



**Fig 7.4.** Standard Memory sharing between  $B$  & the result of  $E$ .

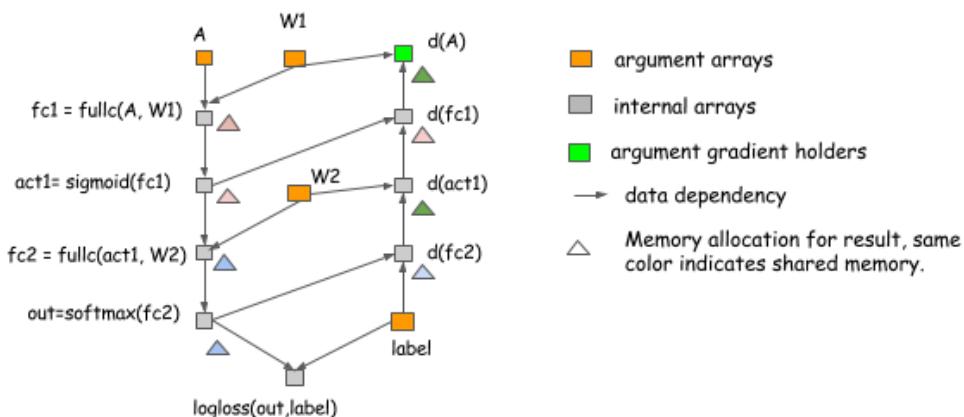
## 7.2 Optimizing Memory Consumption in DL

Memory sharing doesn't necessarily require the same data shape. Note that in the preceding example, the shapes of **B** and **E** can differ. To handle such a situation, we can allocate a memory region of size equal to the maximum of that required by **B** and **E** and share it between them.

### 7.2.3 Memory Allocation Algorithm

Based on the "In-Place Operations", how can we allocate memory correctly?

The key problem is that we need to place resources so that they don't conflict with each other. More specifically, each variable has a **life time** between the time it gets computed until the last time it is used. In the case of the multi-layer perceptron, the life time of *fcl* ends after *act1* get computed. See below figure:



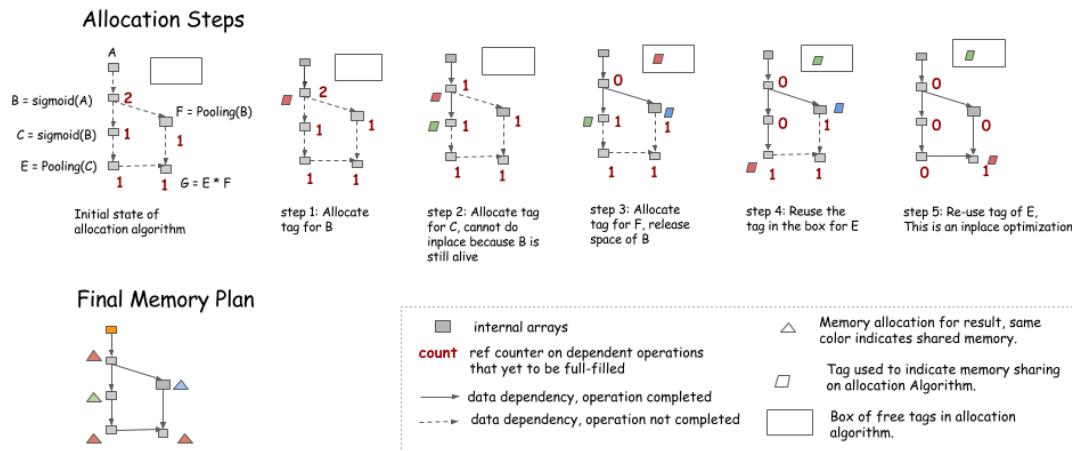
**Fig 7.5.** Standard Memory sharing between **B** & the result of **E**.

The principle is to allow memory sharing only between variables whose lifetimes don't overlap. There are multiple ways to do this. You can construct the conflicting graph with each variable as a node and link the edge between variables with overlapping lifespans, and then run a graph-coloring algorithm. This likely has  $O(n^2)$  complexity, where  $n$  is the number of nodes in the graph. This might be too costly.

Let's consider another simple heuristic. The idea is to simulate the procedure of traversing the graph, and keep a count of future operations that depends on the node.

- An in-place optimization can be performed when only the current operation depends on the source (i.e. count == 1).
- Memory can be recycled into the box on the upper right corner when the *count* goes to 0.
- When we need new memory, we can either get it from the box or allocate a new one.

**Noet:** During the simulation, no memory is allocated. Instead, we keep a record of how much memory each node needs, and allocate the maximum of the shared parts in the final memory plan.



**Fig 7.6.** Standard Memory sharing between **B** & the result of **E**.

### 7.2.4 Static vs. Dynamic Allocation

The major difference is that static allocation is only done once, so we can afford to use more complicated algorithms. For example, we can search for memory sizes that are similar to the required memory block. The Allocation can also be made graph aware. We'll talk about that in the next section. Dynamic allocation puts more pressure on fast memory allocation and garbage collection.

There is also one takeaway for users who want to rely on dynamic memory allocations: do not unnecessarily reference objects. For example, if we organize all of the nodes in a list and store them in a Net object, these nodes will never get dereferenced, and we gain no space. Unfortunately, this is a common way to organize code.

### 7.2.5 Memory Allocation for Parallel Operations

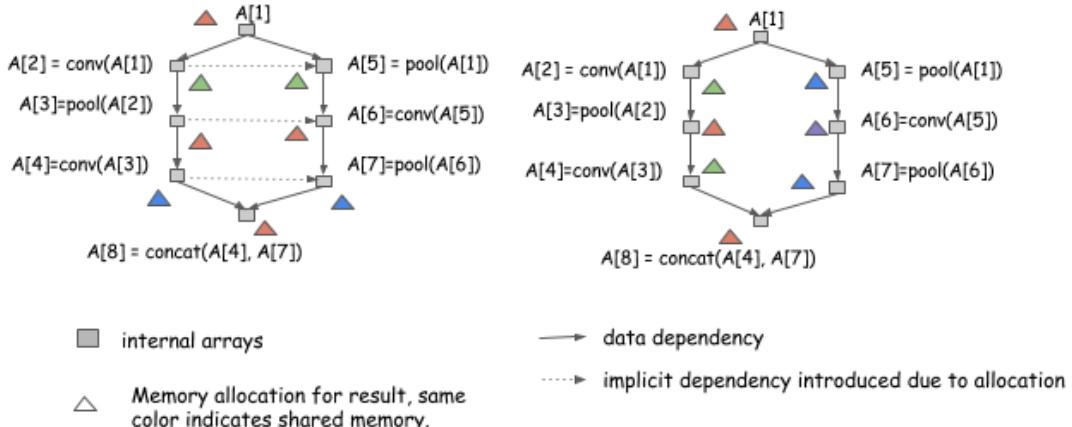
In the previous section, we discussed how we can simulate running the procedure for a computation graph to get a static allocation plan. However, optimizing for parallel computation presents other challenges because resource sharing and parallelization are on the two ends of a balance. Let's look at the following two allocation plans for the same graph:

Both allocation plans are valid if we run the computation serially, **from  $A[1]$  to  $A[8]$** . However, the allocation plan on the left introduces additional dependencies, which means we can't run computation of  $A[2]$  and  $A[5]$  in parallel. The plan on the right can. To parallelize computation, we need to take greater care.

#### Be Correct and Safe First

Being correct is our first principle. This means to execute in a way that takes implicit dependency memory sharing into consideration. You can do this by adding the implicit dependency edge to the execution graph. Or, even simpler, if the execution engine is mutation aware, as described in [our discussion of dependency engine design](#), push the operation in sequence and write to the same variable tag that represents the same memory region.

## 7.2 Optimizing Memory Consumption in DL

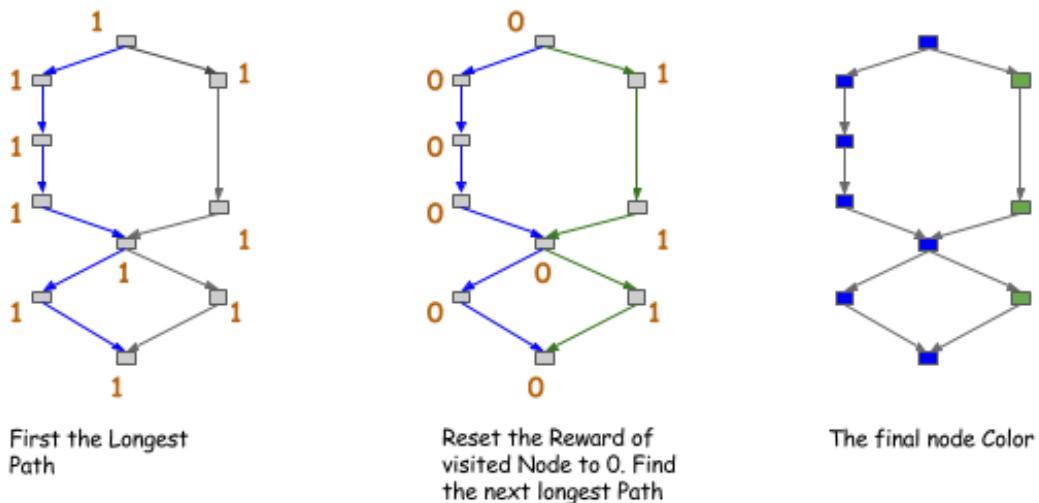


**Fig 7.7.** Standard Memory sharing between **B** & the result of **E**.

Always produce a safe memory allocation plan. This means never allocate the same memory to nodes that can be parallelized. This might not be ideal when memory reduction is more desirable, and we don't gain too much when we can get benefit from multiple computing streams simultaneously executing on the same GPU.

### Try to Allow More Parallelization

Now we can safely perform some optimizations. The general idea is to try and encourage memory sharing between nodes that can't be parallelized. You can do this by creating an ancestor relationship graph and querying it during allocation, which costs approximately  $O(n^2)$  in time to construct. We can also use a heuristic here, for example, color the path in the graph. As shown in the following figure, when you try to find the longest paths in the graph, color them the same color and continue.



**Fig 7.8.** Color the longest paths in the Graph.

After you get the color of the node, you allow sharing (or encourage sharing) only between nodes of the same color. This is a stricter version of the ancestor relationship, but

is costs only  $O(n)$  of time if you search for only the first  $k$  path.

### 7.2.6 How Much Can we Save ?

On coarse-grained operation graphs that are already optimized for big operations, you can reduce memory consumption roughly by half. You can reduce memory usage even more if you are optimizing a fine-grained computation network used by symbolic libraries, such as Theano.

### 7.2.7 References

More details can be found in: [Opimizing the Memory Consumption in DL\(MXNet\)](#).

## 7.3 Deep Learning Programming Style

Two of the most important high-level design decisions

- Whether to embrace the symbolic or imperative paradigm for mathematical computation
- Whether to build networks with bigger or more atomic operations

### 7.3.1 Symbolic vs. Imperative Program

即：符号式编程 vs. 命令式编程

Symbolic programs are a bit different. With symbolic-style programs, we first define a (potentially complex) function abstractly. When defining the function, no actual numerical computation takes place. We define the abstract function in terms of **placeholder values**(占位符). Then we can compile the function, and evaluate it given real inputs.

This operation generates a computation graph (also called a symbolic graph) that represents the computation.

Most symbolic-style programs contain, either explicitly or implicitly, a compile step.  
真正的计算只发生在传入数值之时，在这之前，都没有任何计算发生。

The defining characteristic of symbolic programs is their clear separation between building the computation graph and executing it. For neural networks, we typically define the entire model as a single compute graph.

### 7.3.2 Imperative Programs Tend to be More Flexible

使用命令式编程，那么任何 Python 语法都可以使用 (Nearly anything)，但使用符号式编程时，一些 Python 特性可能无法使用，如迭代。

当使用 Python 的符号式编程时，实际实在一个 Domain-Specific-Language(DSL) 定义的空间中进行编程。

Intuitively, you might say that imperative programs are more native than symbolic programs. It's easier to use native language features.

## 7.3 Deep Learning Programming Style

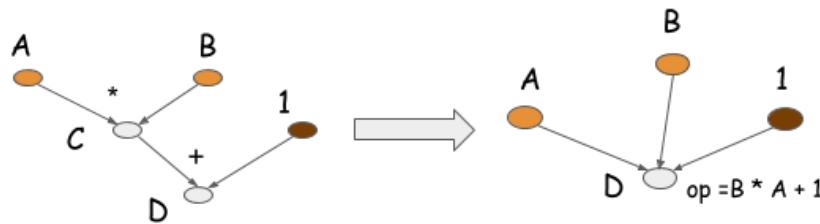


Fig 7.9. Operation Folding 示意图。

### 7.3.3 Symbolic Programs Tend to be More Efficient

命令式编程与原生 Python 的相差不大，所以灵活性很高。但符号式编程更有利于速度、存储优化。

Symbolic programs are more restricted. When we call *Compile* on d, we tell the system that only the value of d is needed. The intermediate values of the computation, is then invisible to us.

- We benefit because the symbolic programs can then safely reuse the memory for in-place computation.
- Symbolic programs can also perform another kind of optimization, called operation folding(图7.9). In fact, this is one way we hand-craft operations in optimized libraries, such as CXXNet and Caffe. Operation folding improves computation efficiency.

Note, you can't perform operation folding in imperative programs, because the intermediate values might be referenced in the future. Operation folding is possible in symbolic programs because you get the entire computation graph, and a clear specification of which values will be needed and which are not.

### 7.3.4 Case Study: Backprop and AutoDiff

toy model

```
import numpy as np
a = np.ones(10)
b = np.ones(10) * 2
c = b * a
d = c + 1
...
```

#### 基于命令式编程的自动求导

循环调用求导函数，直至最开始的输入变量。

利用 grad 闭包 (Closure) 来隐含的保存后向计算图。

但一个坏处是，必须保存所有中间变脸的 Grad 闭包。

## 基于符号式编程的自动求导

可以实现的优化力度更大。

### Analysis

可以实现的优化的程度，依赖于可以允许的操作 (Restrictions on what you can do)。使用符号式编程时，必须明确提供这些约束，所以可进行优化也就更多。

对于命令式编程，可以通过其它的一些方式添加明确的约束。比如一种方法是 Context Variable。如：

```
with context.NoGradient()
    ...
```

可以关闭梯度的计算。但这样也不能利用 In-Place Calculation 来对存储空间进行复用。

其实是一种 trade-off between restriction and flexibility.

### 7.3.5 Model Checkpoint

保存和加载模型。保存模型的时候，需要保存两类变量：网络的结构配置、网络的权重系数。

符号式编程有利于配置的检查。而对于命令式编程，需要保存所有的代码，或者利用符号式指令进行能够顶层封装。

- Parameter Updates

大部分符号式编程，都是基于数据流图 (Data Flow Graphs, DFG) 实现的。DFG 描述的是计算。但这种方式下，参数的更新不是很方便。一些做法是将更新过程转化为基于命令式的方式实现，而梯度的计算是基于计算图的方式计算。

- There is No Strict Boundary

这两种风格的框架其实没有很明显的区分。如在命令式编程时，可以借助 Just-in-Time(JIT) Compiler 来实现一些符号式编程里面的全局优化等好处。

### 7.3.6 Big vs. Small Operations

- Big Operations

主要是一些经典的神经网络层在用，如 Fully Connected and Batch Normalize.

- Small Operations

一些数学上的计算，如矩阵乘、Element-wise Addition.

CXXNet, Caffe 支持 Layer 一级的计算，而 Theano, Minerva 支持更精细的计算。

- Smaller Operations Can Be More Flexible (小运算更灵活)

可实现的东西多，且建立新的 Layer 比较简单，直接添加部件就行。

## 7.3 Deep Learning Programming Style

---

- Big Operations Are More Efficient (大运算更高效)

可能引起计算、存储上的开支。

- Compilation and Optimization

对于小运算的优化，计算图支持一下两种优化：

- Memory Allocation Optimization

重用中间结果的存储空间。也可用于大运算。

- Operator Fusion

Fuse several small operations into big one.

这些优化对小操作十分重要。小操作对编译器也增加了负担。

- Expression Template and Statically Typed Language

借助 Expression Template 来产生具体的 Kernels. [Template Expression](#), 其实底层是基于 C++ Template 实现的。

### 7.3.7 Mix The Approaches

Amdahl's Law:

If you are optimizing a non-performance-critical part of your problem, you won't get much of a performance gain

实际考虑编程 Style 时，需要综合考虑：性能、灵活性、工程复杂度等。实践表明，混合使用多个 Style 可以得到更好的性能。

- Symbolic and Imperative Programs

有两种方式可以实现这种混用：

- Use imperative programs within symbolic programs as callbacks
  - Use symbolic programs as part of imperative programs

如在参数更新中的讨论。如果代码中，混合了 Symbolic 和 Imperative，那么结果是 Imperative. 但更好的选择是，用支持 GPU 计算、参数更新的符号式编程框架来开发。

- Small and Big Operations

- Choose Your Own Approach

## 7.4 Dependency Engine for Deep Learning

### 7.4.1 Problems in Dependency Scheduling

- Data Flow Dependency

Data flow dependency describes how the outcome of one computation can be used in other computations.

- Memory Recycling
- Random Number Generation

A pseudo-random number generator (PRNG) is not thread-safe because it might cause some internal state to mutate when generating a new number. Even if the PRNG is thread-safe, it is preferable to serialize number generation, so we can get reproducible random numbers.

### Design a Generic Dependency Engine

目标是建立一个轻量级、普用的依赖引擎。目标如下：

- 识别有效的操作
- 可以调度 GPU、CPU 存储的依赖，以及处理随机发生器的依赖
- 引擎不应分配资源，而仅处理依赖

步骤如下：

1. At the beginning, the user can allocate the variable tag, and attach it to each of the objects that we want to schedule.

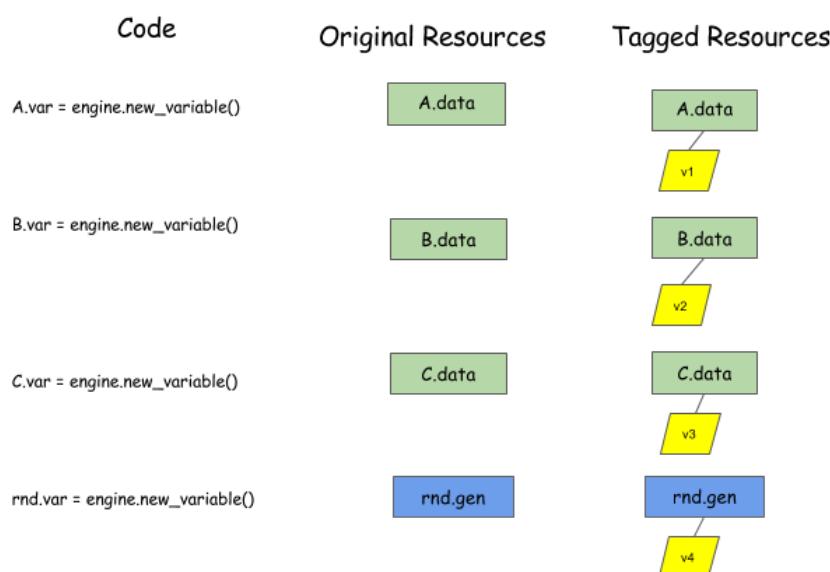
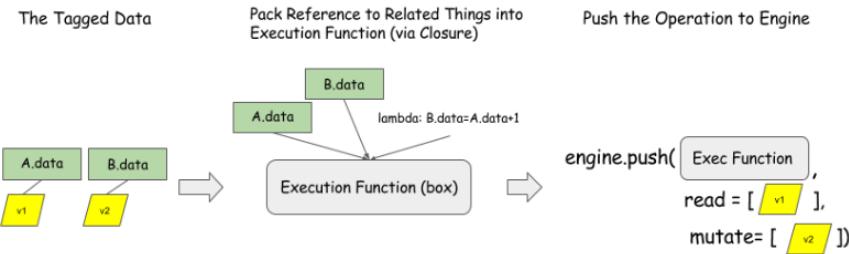


Fig 7.10. 第一步，给变量分配 tag

## 7.4 Dependency Engine for Deep Learning

2. 然后调用 *push* 来告知引擎需要运行的函数、参数等，需要区分读取、写入的参数，分别输入。引擎通过识别上一步的 tag 来识别变量，这样好处是不涉及 tag 具体指向什么，所以引擎可以处理包括变量、函数之类的 Everything。



**Fig 7.11.** 把相应的 Function Closure push 进依赖分析引擎

### 7.4.2 Implementing the Generic Dependency Engine

基本思想

- Use a queue to track all of the pending dependencies on each variable tag
- Use a counter on each operation to track how many dependencies are yet to be fulfilled
- When operations are completed, update the state of the queue and dependency counters to schedule new operations

一个例子如图所示：

Push Sequence	Variable Pending Queue	Running Operations	Completed Operations in Current Cycle
A=2 is pushed, because all its dependencies are satisfied, it is executed directly.	A B rnd		
Another operation is pushed, the random number dependency is satisfied, but still need to wait for A	A B rnd	A = 2	
This represents in the intermediate stage, where is the previous pushed up waits on the variable queue.	A B rnd	B = A + rnd.uniform(-1,1) {1}	A = 2
A=2 finishes, and the dependent operations are triggered. Another new operation is pushed.	A B rnd	B = A + rnd.uniform(-1,1)	A = 2
The newly pushed operation is added to the two dependency queues it waits on.	A B rnd	B = A + rnd.uniform(-1,1)	A = 2
The previous operation on B finishes, as a result, all dependencies of A = rnd.uniform is satisfied and it is able to run,	A B rnd	A = rnd.uniform(-1,1)	B = A + rnd.uniform(-1,1)
All pushed operations finished running.	A B rnd		A = rnd.uniform(-1,1)
operation (wait counter) operation and the number of pending dependencies it need to wait for			
var Variable queue, ready to read and mutate			
var Variable queue, ready to read, but still have uncompleted reads. Cannot mutate			
var Variable queue, still have uncompleted mutations. Cannot read/write			
Execution Cycle(step) Separator Line			

**Fig 7.12.** 一个具体的例子

### 7.4.3 Discussion

- Dynamic vs. Static
- Mutation vs. Immutable

## 7.5 Designing Efficient Data Loaders for DL

几点重要的考虑:

- Small File Size
- Parallel (Distributed) packing of data
- Fast data loading and online augmentation
- Quick reads from arbitrary parts of the dataset in the distributed setting

### 7.5.1 Design Insight

为了设计好的 IO 系统，需要解决两类任务：Data Preparation, Data Loading.

#### Data Preparation

Data preparation describes the process of packing data into a desired format for later processing.

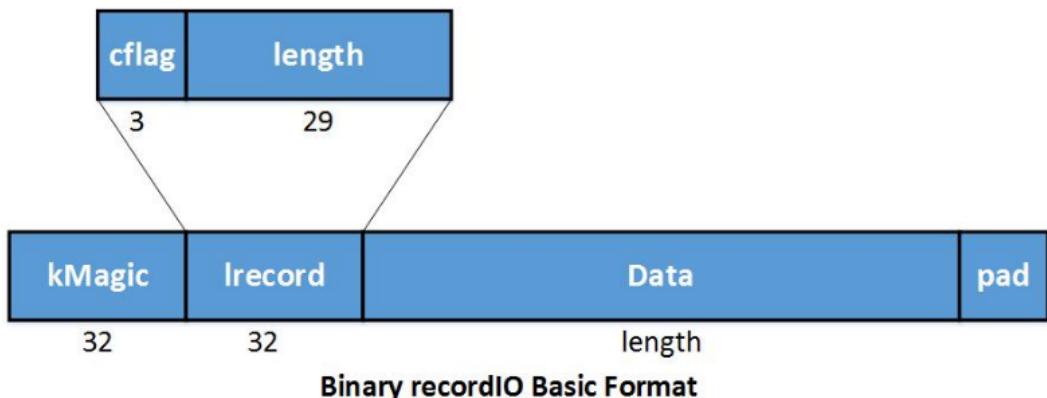
- Pack the dataset into small numbers of files
- Do the packing once
- Process the packing in parallel to save time
- Be able to access arbitrary parts of the data easily

#### Data Loading

The next step to consider is how to load the packed data into RAM.

- Read Continuously
- Reduce the bytes to be loaded, 可以借助于数据压缩等
- Load and train in different threads
- Save RAM

## 7.5 Designing Efficient Data Loaders for DL



In MXNet's binary RecordIO, we store each data instance as a record. **kMagic** is a *magic number* indicating the start of a record. **Lrecord** encodes length and a continue flag. In lrecord,

- cflag == 0: this is a complete record
- cflag == 1: start of a multiple-records
- cflag == 2: middle of multiple-records
- cflag == 3: end of multiple-records

**Data** is the space to save data content. **Pad** is simply a padding space to make record align to 4 bytes.

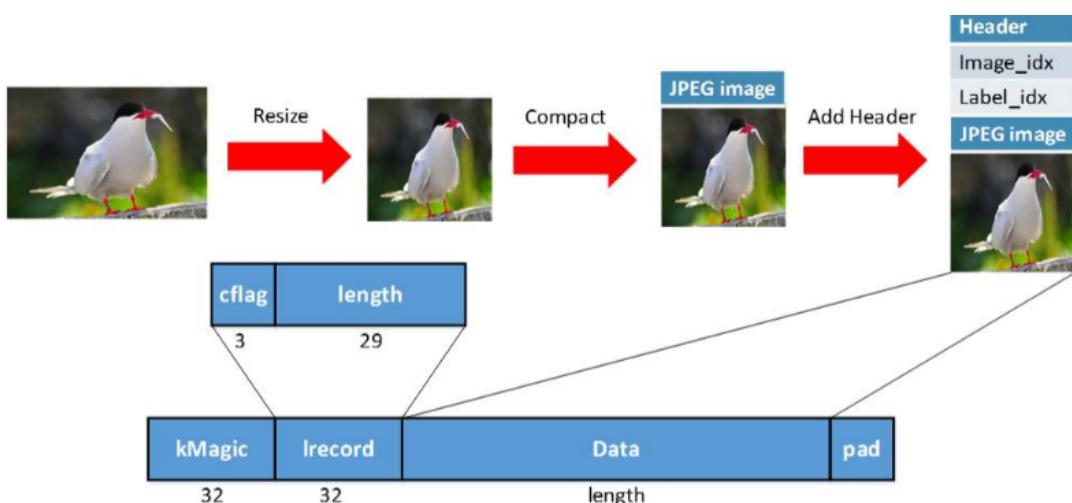
**Fig 7.13.** Binary recordIO 数据结构

### 7.5.2 Data Format

需要选择既高效又方便的数据结构。为了实现这个目标，把二进制数据包装成可以分离的结构。具体，MXNet 采用 DMLC-Core 里面的 recordIO 类型。

通过连续读，来避免随机读引入的延时。

这种数据结构的一个好处是，每一个 record 的长度可以改变。从而支持更好的数据压缩。



**Fig 7.14.** Binary recordIO 的一个例子

其中，*resize* 把输入图像变为 256 \* 256 大小。

## Access Arbitrary Parts of Data

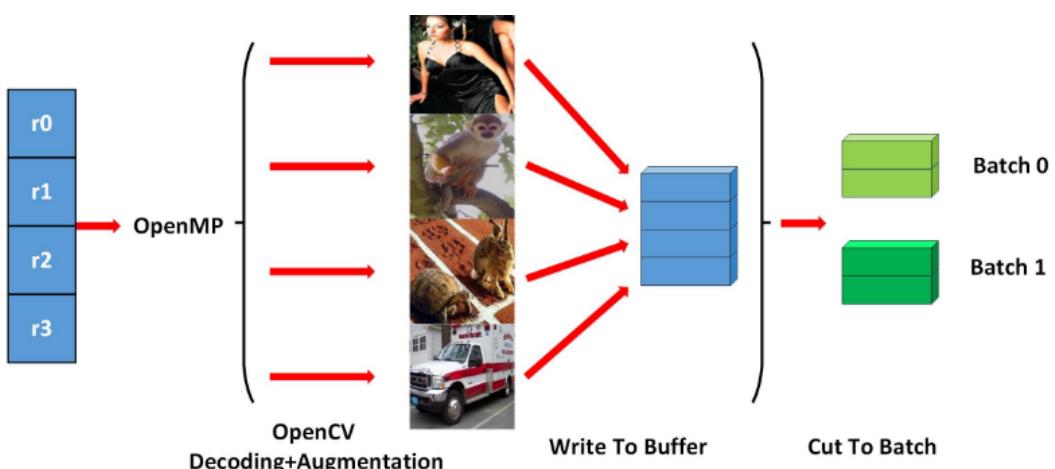
The packed data can be logically sliced into an arbitrary number of partitions, no matter how many physical packed data files there are.

在 recordIO 中借助 Magic Number 来实现上述目的，具体是使用 dmlc-core 中的 *InputSplit* 函数来实现。这个函数极大的帮助了并行实现，因为每一个节点只处理一个 *Part*.

### 7.5.3 Data Loading and Preprocessing

#### Loading and Preprocessing on the Fly

In service of efficiency, we also address multi-threading techniques.



**Fig 7.15.** 并行预处理例子

在加载了大量图像数据后，利用多线程工具 (OpenMP) 进行并行处理。

#### Hide IO Cost Using Threadediter

一种降低 IO 影响的办法是：数据预取。具体使用 dmlc-core 提供的 *threadediter* 来处理 IO. The key of *threadediter* is to start a stand-alone thread that acts as a data provider, while the main thread acts as a data consumer as illustrated below.

The *threadediter* maintains a buffer of a certain size and automatically fills the buffer when it's not full. And after the consumer finishes consuming part of the data in the buffer, *threadediter* will reuse the space to save the next part of data.

### 7.5.4 MXNet IO Python Interface

We make the IO object as an iterator in numpy. By achieving that, the user can easily access the data using a for-loop or calling *next()* function. Defining a data iterator is very similar to defining a symbolic operator in MXNet.

为了创建一个数据迭代器，需要提供五种类型的参数：

- Dataset Param: 如路径、输入的尺寸等

## 7.6 Except Handling in MXNet

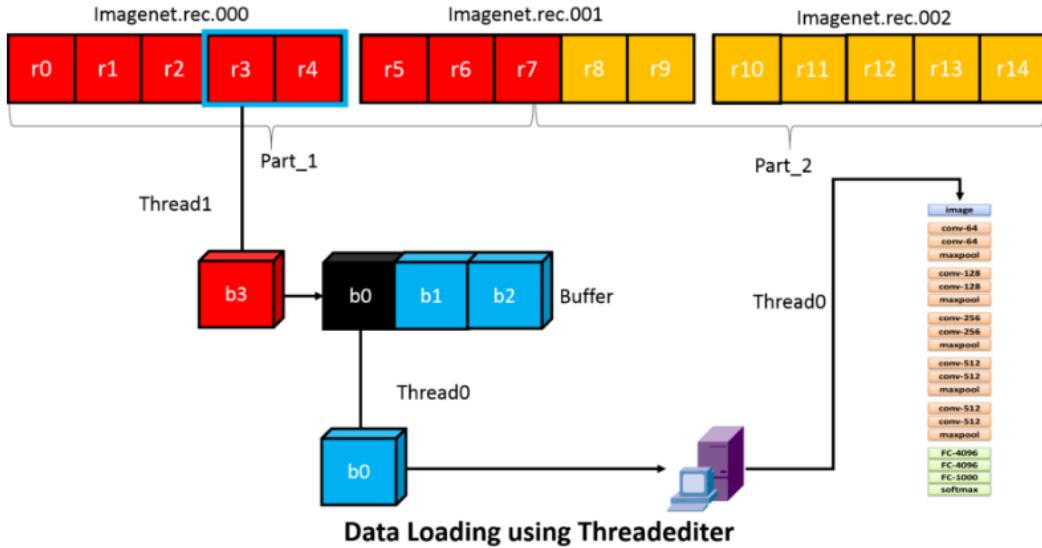


Fig 7.16. 数据预取的示意图，借助 Buffer 来实现

- Batch Param: Batch Size
- Augmentation Param: 确定数据增广的类型，如翻转等
- Backedn Param: 控制后台线程，来隐藏数据读取延时
- Auxiliary Param: 帮助 Debug 的信息等。

通常必须确定 **Dataset Param** 和 **Batch Param**。MX Data IO 也支持模块化，如以下两种：

- 自己的高效数据预取。allows the user to write a data loader that reads their customized binary format that automatically gets multi-threaded prefetcher support.
- 数据转换。image random cropping, mirroring, etc. Allows the users to use those tools, or plug in their own customized transformers

## 7.6 Except Handling in MXNet

MXNet 在两类情况下可以抛出异常：

- MXNet main thread. For e.g. InferShape and InferType。在主线程中处理
- Spawed threads (生成线程？)
  - By dependency engine for operator execution in parallel。会被 rethrown 到主线程，进行处理
  - By the iterators, during the data loading, text parsing phase etc.

## Exception Handling for Iterators

CVIter 使用 PrefetcherIter 来加载和解析数据。PrefetcherIter 会生成一个 Producer 线程并后台运行，在主线程(Consumer)中使用这些数据。在 Producer 线程中可能抛出异常，该异常被发送到主线程。

可能引起线程之间的竞争(Race). To avoid this situation, you should try and iterate through your full dataset if you think it can throw exceptions which need to be handled.

## Except Handling for Operators

For the operator case, the dependency engine spawns a number of threads if it is running in the **ThreadedEnginePool** or **ThreadedEnginePerDevice** mode. The final operator is executed in one of the spawned threads.

If an operator throws an exception during execution, this exception is propagated down the dependency chain. Once there is a synchronizing call i.e. **WaitToRead** for a variable in the dependency chain, the propagated exception is rethrown.

**注意:** Please avoid waitalls in your code unless you are confident about your code not throwing exception in any scenario. 因为 **mx.nd.waitall** 不支持 rethrowing 异常。

## 7.7 MXNet-Gluon 创建模型

参考文献: [Gluon](#)

### 7.7.1 模型构造

继承`gluon.nn.Block`来实现，必须重载`forward`函数。在`__init__`函数里面定义模型所需的变量等。

**需要值得注意的地方是**，在基于自定义的模型构建大的模型时，需要调用的是`__init__`函数的参数；而一旦模型定义好了，开始向里面传入数据时，数据的格式需要根据`forward`函数来确定！

#### 两种稍微不同的实现

方式一，是直接利用 Gluon 预定义的层构建类型变量。

Listing 7.5: 自定义模型，方式一，label

```
from mxnet import nd
from mxnet.gluon import nn

class MLP(nn.Block):
    # 声明带有模型参数的层，这里我们声明了两个全链接层。
    def __init__(self, **kwargs):
        # 调用 MLP 父类 Block 的构造函数来进行必要的初始化。
        # 这样在构造实例时还可以指定
        # 其他函数参数，例如下一节将介绍的模型参数 params。
        super(MLP, self).__init__(**kwargs)
        # 隐藏层。
        self.hidden = nn.Dense(256, activation='relu')
```

## 7.7 MXNet-Gluon 创建模型

---

```
# 输出层。  
    self.output = nn.Dense(10)  
# 定义模型的前向计算，即如何根据输出计算输出。  
def forward(self, x):  
    return self.output(self.hidden(x))
```

方式二，即先定义一个 Sequential 的类型变量 Net，然后在向里面增加 gluon.nn.Layer 等具体的层，或者自己定义的层。

Listing 7.6: 自定义模型，方式二，label

```
class NestMLP(nn.Block):  
    def __init__(self, **kwargs):  
        super(NestMLP, self).__init__(**kwargs)  
        self.net = nn.Sequential()  
        self.net.add(nn.Dense(64, activation='relu'),  
                    nn.Dense(32, activation='relu'))  
        self.dense = nn.Dense(16, activation='relu')  
  
    def forward(self, x):  
        return self.dense(self.net(x)) # 调用要加 self  
  
net = nn.Sequential()  
net.add(NestMLP(), nn.Dense(20), FancyMLP())  
  
net.initialize()  
net(x)
```

由于 FancyMLP 和 Sequential 都是 Block 的子类，我们可以嵌套调用他们。实际使用中，貌似还存在第三种构建方式，直接以函数的形式使用，如 VGG：

Listing 7.7: 自定义模型 VGG11，方式三，label

```
import sys  
sys.path.append('..')  
import gluonbook as gb  
from mxnet import nd, init, gluon  
from mxnet.gluon import nn  
  
def vgg_block(num_convs, num_channels):  
    blk = nn.Sequential()  
    for _ in range(num_convs):  
        blk.add(nn.Conv2D(  
            num_channels, kernel_size=3, padding=1, activation='relu'))  
    blk.add(nn.MaxPool2D(pool_size=2, strides=2))  
    return blk  
conv_arch = ((1, 64), (1, 128), (2, 256), (2, 512), (2, 512))  
  
def vgg(conv_arch):  
    net = nn.Sequential()  
    # 卷积层部分  
    for (num_convs, num_channels) in conv_arch:  
        net.add(vgg_block(num_convs, num_channels))  
    # 全连接接层部分
```

```

        net.add(nn.Dense(4096, activation="relu"), nn.Dropout(.5),
            nn.Dense(4096, activation="relu"), nn.Dropout(.5),
            nn.Dense(10))
    return net

net = vgg(conv_arch)

```

## 7.7.2 自定义层

也需要继承gluon.nn.Block 以及重载forward 函数。

Listing 7.8: 自定义层，方式一，label

```

class CenteredLayer(nn.Block):
    def __init__(self, **kwargs):
        super(CenteredLayer, self).__init__(**kwargs)

    def forward(self, x):
        return x - x.mean()

```

## 带参数的自定义层

我们还可以自定义含模型参数的自定义层。这样，自定义层里的模型参数就可以通过训练学出来了。我们在“模型参数”一节里介绍了 Parameter 类。其实，在自定义层的时候我们还可以使用 Block 自带的 ParameterDict 类型的成员变量 params。顾名思义，这是一个由字符串类型的参数名字映射到 Parameter 类型的模型参数的字典。我们可以通过 get 函数从 ParameterDict 创建 Parameter，注意是创建，也就是加入！

Listing 7.9: 带参数的自定义层，方式一，label

```

class MyDense(nn.Block):
    def __init__(self, units, in_units, **kwargs):
        super(MyDense, self).__init__(**kwargs)
        self.weight = self.params.get('weight', shape=(in_units,
                                                       units))
        self.bias = self.params.get('bias', shape=(units,))

    def forward(self, x):
        linear = nd.dot(x, self.weight.data()) + self.bias.data()
        return nd.relu(linear)

```

## 7.7.3 实际例子

### ResNet

使用的是自定义层方式一中的方式构建残差块。然后定义更高一级的结构(自定义模型方式二中的方法)，最后再在更高一级进行构架整个模型。层次结构如下：

## 7.7 MXNet-Gluon 创建模型

---

Listing 7.10: ResNet using Gluon

```
import sys
sys.path.append('..')
import gluonbook as gb
from mxnet import nd, gluon, init
from mxnet.gluon import nn

# 构建底层
class Residual(nn.Block):
    def __init__(self, num_channels, use_1x1conv=False, strides=1,
                 **kwargs):
        super(Residual, self).__init__(**kwargs)
        self.conv1 = nn.Conv2D(num_channels, kernel_size=3, padding
                           =1,
                           strides=strides)
        self.conv2 = nn.Conv2D(num_channels, kernel_size=3, padding
                           =1)
        if use_1x1conv:
            self.conv3 = nn.Conv2D(num_channels, kernel_size=1,
                               strides=strides)
        else:
            self.conv3 = None
        self.bn1 = nn.BatchNorm()
        self.bn2 = nn.BatchNorm()

    def forward(self, X):
        Y = nd.relu(self.bn1(self.conv1(X)))
        Y = self.bn2(self.conv2(Y))
        if self.conv3:
            X = self.conv3(X)
        return nd.relu(Y + X)

# 构建高一层
def resnet_block(num_channels, num_residuals, first_block=False):
    blk = nn.Sequential()
    for i in range(num_residuals):
        if i == 0 and not first_block:
            blk.add(Residual(num_channels, use_1x1conv=True, strides
                           =2))
        else:
            blk.add(Residual(num_channels))
    return blk

# 构建整个模型
net.add(resnet_block(64, 2, first_block=True),
        resnet_block(128, 2),
        resnet_block(256, 2),
        resnet_block(512, 2))
```

需要注意的是,即使在class里面也可以调用底层的自定义模块,如在resnet\_block中调用Residual, 具体调用方式是调用底层类的构造函数, 即`__init__`函数, 参数要跟这个函数的参数一致。

## DenseNet

发现与上面的 ResNet 类似，所以这里省略。

## 7.8 MXNet 中的 Deconvolution

具体实现是，Conv2DTranspose 等。

由于 Conv2DTranspose 这个函数会引起棋盘状 Artifacts，所以在 MXNet 里面，还有一个上采样卷积的函数：mxnet.ndarray.UpSampling()，但是，听说目前还不好用啊。

参考 [FCN 讨论区 -chen0510566](#) 回答，gluon.nn.Conv2DTranspose 的文档，Conv2DTranspose 输出的 size 是这样计算的：

$$\begin{aligned} \text{Out\_height} &= (\text{height} - 1) * \text{strides}[0] - 2 * \text{padding}[0] + \text{kernel\_size}[0] + \text{output\_padding}[0] \\ \text{Out\_width} &= (\text{width} - 1) * \text{strides}[1] - 2 * \text{padding}[1] + \text{kernel\_size}[1] + \text{output\_padding}[1] \end{aligned}$$

↑ 正确性有待验证。

## 7.9 MXNet 读取训练数据

重要，我发现，对于我这样的新手来说，如何处理数据也是个问题，尤其现在，我感觉一头雾水。

参考文章：

- [1] [MXNet 常见加载数据的方式 -简书](#)
- [2] [MXNet API](#)

有必要对 MXNet 的数据结构进行说明一下！尤其对于 axes 的概念。在 MXNet 中，我认为 axes 可以取四个值，分别对应以下概念：

- axes = 0, Batches，比如 batchsize=8，那么 axes 就对应这 8 个样本
- axes = 1, Channels，比如输入彩色图像，那么 axes 对应的就是 RGB 三个通道
- axes = 2, Width or Height，还不太确定，与 axes=3 共同确定每一个 Channel 的尺寸

接下来就是本小节的重点内容，也就是如果处理训练数据，这是开始的一步，但我感觉也是容易被各种教程忽视的一步，如果这步不清楚，那么会影响整个后面的学习与实践过程。

正如参考文献中所总结的那样，一共可以分为三类。分别如下！

- 基于 mxnet.io 读取数据
- 使用 mxnet.image 读取数据
- 使用 gluon 接口读取数据，这个部分是重点

此外，作者还提到了几种支持的数据结构，貌似就这几种么？

- .rec 文件
- raw image
- .lst 文件，使用 mx.image.ImageIter 接口读取，这个接口可以读.rec 的文件

大概流程：传 data iter 对象给 base\_module，把数据对象变为 iterator。下面分别对三种数据读取方式进行示例。

### 7.9.1 使用 mx.io 读取数据

#### 从内存中读取

调用 mx.io.NDArrayIter 接口，以 MXNet 官方的 train\_mnist.py 为例。

Listing 7.11: train\_mnist.py 例子

```
def read_data(label, image):  
    """  
    download and read data into numpy  
    """  
    base_url = 'http://yann.lecun.com/exdb/mnist/'  
    with gzip.open(download_file(base_url+label, os.path.join('data',  
        label))) as flbl:  
        magic, num = struct.unpack(">II", flbl.read(8))  
        label = np.fromstring(flbl.read(), dtype=np.int8)  
    with gzip.open(download_file(base_url+image, os.path.join('data',  
        image)), 'rb') as fimg:  
        magic, num, rows, cols = struct.unpack(">IIII", fimg.read(16))  
        image = np.fromstring(fimg.read(), dtype=np.uint8).reshape(  
            len(label), rows, cols)  
    return (label, image)  
  
def to4d(img):  
    """  
    reshape to 4D arrays  
    """  
    return img.reshape(img.shape[0], 1, 28, 28).astype(np.float32)  
    /255  
  
def get_mnist_iter(args, kv):  
    """  
    create data iterator with NDArrayIter  
    """  
    (train_lbl, train_img) = read_data(  
        'train-labels-idx1-ubyte.gz', 'train-images-idx3-ubyte.gz'  
    )  
    (val_lbl, val_img) = read_data(  
        't10k-labels-idx1-ubyte.gz', 't10k-images-idx3-ubyte.gz')  
    train = mx.io.NDArrayIter(  
        to4d(train_img), train_lbl, args.batch_size, shuffle=True)  
    val = mx.io.NDArrayIter(  
        to4d(val_img), val_lbl, args.batch_size, shuffle=False)
```

---

```

    to4d(val_img), val_lbl, args.batch_size)
    return (train, val)
#参考 train_mnist.py

```

详解如下：

- 顶层函数为get\_mnist\_iter 函数

该函数的参数是：包含 batch size 的类 arg。kv 参数，但貌似没用到。

函数get\_mnist\_iter 调用read\_data 函数，返回图像数据及其对应的 label。该函数的

将read\_data 函数的返回值传入mx.io.NDArrayIter 函数，该函数返回名称为train 和val 的 data iterator. 分别作用与训练图像与训练标签。NDArrayIter 函数的使用参数解释在下文单独总结。

返回 data iterator 的训练图像与训练标签。

- 被调用函数read\_data

该函数在这里主要负责下载数据文件 (download\_file())、解压缩 (struct.unpack) 等。各个函数的输入输出可参考源代码。

- 被调用函数to4d

负责把输入图片 reshape 成四维数据结构，维度分解见上文中的 axes 的解释。

所以总结一下的话，就是这样的，首先下载文件并解压缩，然后读取数据，送入mx.io.NDArrayIter 产生 data iterator.

## 从.csv 文件中读取数据

Listing 7.12: Read .csv 数据

```

#lets save `data` into a csv file first and try reading it back
np.savetxt('data.csv', data, delimiter=',')
data_iter = mx.io.CSVIter(data_csv='data.csv', data_shape=(3,), 
    batch_size=30)
for batch in data_iter:
    print([batch.data, batch.pad])

```

详解：

- 调用mx.io.CSVIter 读取.csv 格式文件，同时输入数据 shape 以及 batch size。返回的就是一个 data iterator。

## 使用 mx.io.ImageRecordIter 接口，需要定制 KVStore

这一部分还没看明白，可以看一下原文。

### 7.9.2 常用的系统提供的接口函数

分析以上实现方式，主要涉及以下几个系统接口函数。

- `mx.io.NDArrayIter`

原型: `class mxnet.io.NDArrayIter(data, label=None, batch_size=1, shuffle=False, last_batch_handle='pad', data_name='data', label_name='softmax_label')`

其中, `data`, `label` 为 array 或 list of array or dict of string to array。

`last_batch_handle` 为最后一个 batch 的处理, 即训练数据的数目不能被 batch size 整除时的处理方式, 可以选取'pad', 'discard' or 'roll\_over' 等。

更多细节见文档吧。

Returns an iterator for `mx.nd.NDArray`, `numpy.ndarray`, `h5py.Dataset` `mx.nd.sparse.CSRNDArray` or `scipy.sparse.csr_matrix`.

- `mx.io.CSVIter`

return the CSV file iterator, 详情见文档吧。

- `struct.unpack`

原型: `struct.unpack(fmt, buffer)`

Unpack from the buffer `buffer` (presumably packed by `pack(fmt, ...)`) according to the format string `fmt`. The result is a tuple even if it contains exactly one item. The buffer's size in bytes must match the size required by the format, as reflected by `calcsize()`. 关于 `fmt` 的具体格式, 参考[struct library python](#)

- `gzip.read()`

原型: `read(size=-1)`

Read and return up to `size` bytes. If the argument is omitted, `None`, or negative, data is read and returned until EOF is reached. An empty bytes object is returned if the stream is already at EOF.

If the argument is positive, and the underlying raw stream is not interactive, multiple raw reads may be issued to satisfy the byte count (unless EOF is reached first). But for interactive raw streams, at most one raw read will be issued, and a short result does not imply that EOF is imminent.

所以参数是要读取的 byte 的数目。

### 7.9.3 使用 `mx.image` 读取数据

从 `imagerlist` 中读取可迭代的数据。

Listing 7.13: `mx.image` 使用示例

```
data_iter = mx.image.ImageIter(batch_size=4, data_shape=(3, 224,
    224), label_width=1,
data_iter.reset()
for data in data_iter:
```

```

d = data.data[0]
print(d.shape)
# we can apply lots of augmentations as well
data_iter = mx.image.ImageIter(4, (3, 224, 224), path_imglist='data/
    custom.lst',
data = data_iter.next()
# specify augmenters manually is also supported
data_iter = mx.image.ImageIter(32, (3, 224, 224), path_rec='data/
    caltech.rec',

```

详解：

- 调用 `mx.image.ImageIter` 实现

#### 7.9.4 使用 Gluon 接口读取数据

由于用的比较多，所以这部分是重点，起码目前来看是重点。

使用 Gluon 接口时，我觉着主要分为两步：产生 Dataset，基于 Dataset 产生 Mini-batch iterator！

下面分别对这两个步骤进行简单说明，后续有时间在详细说一下吧。通过本小节的简单解释，然后再看这个文档，应该就不那么难了吧，就知道各个部分是什么意思了，怎么组织在一起的了。

具体的参考文档：[Gluon API 文档](#)

#### 产生 Dataset

Dataset 的生成主要分为几种方式，这里主要讨论三种形式：

- Gluon 自带的数据库，包括：MNIST, FashionMNIST, CIFAR10, CIFAR100 等。

产生 Dataset 的方式为：

```
data_set = gluon.data.vision.Dataset
```

其中，Dataset 需要替换成提到的提供的自带数据库。

- 来自目录生成 Dataset，调用：`ImageFolderDataset`，目录中保存的 raw image。

产生 Dataset 的方式：

```
data_set = gluon.data.vision.ImageFolderDataset(root, flag=1, transform=None)
```

A dataset for loading image files stored in a folder structure like: 看文档，`ImageFolderDataset` 的说明文档。其中, `flag=1` 说明把读入的图像转换成三通道彩色的，若 `flag=0` 那么读入的是灰度图像。

- 自己生成 Dataset 类！

产生自定义 Dataset 的方式：

## 7.10 Gluon 中的数据结构

---

```
class myDatasetName(gluon.data.Dataset)
    ... other functions, to process image as your need
    def __getitem__(..):
        ...
    def __len__(..):
        ...
```

从上面的代码可以看出,生成自定义的 Dataset 时,需要继承 `gluon.data.Dataset` 类, 同时, 必须 override `__getitem__` 和 `__len__` 两个函数! 实际使用中, 该类可以按照自己数据库的格式, 在 `myDatasetName` 类中定义其它帮助处理数据的函数, 一个更具体点的例子是[FCN-Gluon](#)这个文档的代码!

### 产生 Mini-batch iterator

在上一步, 无论用哪一种方式产生了 Dataset 后, 就可以送入 `DataLoader` 产生 data iterator 了!

这一步主要通过借助 `gluon.data.DataLoader` 这一 Iterator 类来实现。此外还有一些基于 Sample 的函数, 具体看文档。

产生 Mini-batch Iterator 的方式:

```
data_iter = gluon.data.DataLoader(dataset, batch_size=Noen,
                                   shuffle=False, ....)
```

后面还有很多参数, 可以去文档查看。

这里的第一个参数: `dataset` 也就是上文提到的生成 Dataset 的结果了。

## 7.10 Gluon 中的数据结构

这一部分, 我主要想总结一下 Gluon 中的各种部件的数据结构, 如 `Net = gluon.nn.Sequential()` 后, `net` 的数据结构到底是什么样的, 还有就是网络的参数又是什么结构的, 比如 `Parameter` 类到底长什么样, 又该如何使用等。有时间慢慢写吧。

填下坑。

我们可以通过 `[]` 来访问 `Sequential` 类构造出来的网络的特定层。对于带有模型参数的层, 我们可以通过 `Block` 类的 `params` 属性来得到它包含的所有参数, **注意这里的 Block 类是所有 neural network layer 和 model 的基类!** 如: `net[0].params` 会返回 parameter dictionary。

既然 `params` 会返回 parameter dictionary, 那么就可以通过名字来访问字典里面的元素, 也可以直接使用它的变量名。下面的两种方法是等价的, 但通常后者的代码可读性更好。补充一下这个字典的 Key-Value 的类型, Key 是 `net` 的层名字, 如 `dense0_weight` 等; Value 是类型是 `gluon.nn.Parameter` 类型。

```
net[0].params['dense0_weight']
```

```
net[0].weight
```

后者貌似不是 Block 的成员了，通过看 Conv2D 为例，`weight` 是 Conv2D 的一个类成员，其它 layer 类型应该也有吧，还包括 `bias` 等。

上面两种方式，我们都得到了一个 Parameter 类型的变量了。该类型支持 `lr_mult` 等类型成员了，而且，支持通过 `data()` 和 `grad()` 等函数来获取具体的数据。

最后，基类 Block 提供了 `collect_params` 函数来获取这个实例包含的所有参数，他返回的同样是一个 Parameter Dictionary，该字典类型还支持 `initialize` 和 `reset_ctx` 等等等等。

## 7.11 MXNet 中的 Confusing 参数

### 7.11.1 mx.symbol.reshape

```
reshape(data=None, shape=_Null, reverse = _Null, target_shape=_Null, target_shape=_Null, keep_highest=_Null, name=None, attr=None, out=None, **kwargs)
```

需要注意的点是，`shape` 参数可以取一些特殊的值  $\in \{0, -1, -2, -3, -4\}$ 。它们的含义如下：

- 0: Copy this dimension from the input to the output shape
- -1: refers the dimension of the output shape by using the remainder of the input dimensions keeping the size of the new array same as that of the input array. At most one dimension of shape can be -1
- -2: Copy all/remainder of the input dimensions to the output shape
- -3: use the product of two consecutive dimensions of the input shape as the output dimension
- -4: split one dimension of the input into two dimensions passed subsequent to -4 in shape

此外，还需要注意一下 `reverse` 参数的设置问题，具体的更多讲解、例子参考 [API 文档](#)。

### 7.11.2 mx.symbol.transpose

```
transpose(data=None, axis=_Null, name=None, attr=None, out=None)
```

其中 `axis` 表明输出数据的 `axis` 顺序，默认是取反。

另一个很重要的点是 `axis` 的具体取值，如  $axis = (a, b, c)$ ，这句话的具体操作时：输出数据中的第 0 维，对应的是输入数据中的第  $a$  维，输出数据中的第 1 维是对应输入数据中的第  $b$  维，输出数据中的第 2 维对应的是输入中的第  $c$  维。

例如：输入是高 \* 宽 \* 通道数，那么要转换成通道数 \* 高 \* 宽，那么 `axis` 应该取  $(2, 0, 1)$ 。

## 7.12 UpSample In MXNet

这一部分主要总结今天看到的所有关于 MXNet 中 UpSample 使用的网上帖子。

参考 [1]: [UpSampling-MXNet API](#)

```
mxnet.symbol.UpSampling(*data, **kwargs)
```

上式用于对输入实现最近邻/双线性插值。其中，\*data 可以选择：data, scale, num\_filter, sample\_type, multi\_input\_mode, workspace, name, 以及num\_args。

参考 [2]: [使用 Deconvolution 作为 UpSampling 来使用](#)

原文题目：[How to use mx.sym.UpSampling for bilinear upsampling](#), 提问者不知道如何设置 UpSampling 的参数，下面有答案说可以用 Deconvolution 来实现分辨率的提高，具体的跟 DLTips 一章里面的用法是一样的，也就是如何设置参数来成倍的提高分辨率。

另一个讨论者通过对 `src/operator/upsampling.cc` 代码分析可知，当创建 Bilinear Upsampling operator 时候，其实会返回 Deconvolution Opeartioin。所以可以确定的是，Deconvolution 可以用于双线性插值，而要完成这个，只要保证 `upsample_weight` 被初始化为 Bilinear Kernel。但讨论者有另外一个疑问：就是在后向传播的时候，`upsampling_weight` 也会更新，但这可以通过设置网络层的学习率来决定更新或者不更新。

在这个问题下面，有回答者又给出了另一个链接：[BilinearResize2D-PR](#)，貌似是 5 月份刚提交的，所以可能需要更新下 MXNet。

参考 [3]: [Deconvolution layer 的参数配置](#)

下面 piiswrong 的回答同样是 DLTips 那一章节的提升 Scale 倍分辨率时的参数配置。

参考 [4]: [不理解 UpSampling 是如何工作的](#)

还是 piiswrong 的回答：Blinear Upsampling 是基于 Convolution 实现的，因此需要权重，如果不想在训练过程中更新这些参数，需要设置 `lr_scale` 为 0！

提问者的回答：只要不更新权重系数，就可以当做 Bilinear Interpolation 来用，但如果跟新参数的话，就会被当做普通的 Deconvolution 来用，piiswrong 认为这样更好一点。

那么具体怎么实现呢？

piiswrong 的回答是需要一个 weight matrix. 可以被 `mx.init` 中的 bilinear initializer 进行初始化。

一个例子：

Listing 7.14: UpSampling 的一个例子

```
#upsample filter
def upsample_filt(size):
    factor = (size + 1) // 2
    if size % 2 == 1:
        center = factor - 1
    else:
        center = factor - 0.5
    og = np.ogrid[:size, :size]
    return (1 - abs(og[0] - center) / factor) * \
           (1 - abs(og[1] - center) / factor)

# network
net = mx.symbol.Variable('data')
```

```

net = mx.symbol.UpSampling(net, num_filter=3, scale=2,
                           sample_type='bilinear', num_args = 2, name="up")

#upsampling weights
filt = upsample_filt(4)
initw = np.zeros((3,1,4,4), dtype=np.float32)
initw[range(3), 0, :, :] = filt
initw = mx.nd.array(initw)
#forward
c_exec = net.bind(ctx=mx.cpu(), args={'data': im_data, 'up_weight':
    initw})
c_exec.forward()

```

最后，给出一个底层一点的解释：

Each upsample layer consists of (1) Upsample (bilinear, zero padding, nearest-neighbor) and (2) convolution with a transposed filter. That is why you need the convolution operation in these layers. Convolution always reduce the input matrix to the output, to go to a bigger output, you have to upsample the input first and potentially do zero - padding to the output later on in order to fix the dimensions. Also, the filters are called learned filters, because the un-transposed version (filter) that we used to calculate the transposed filter learned the features during training.

在代码里面,需要注意的是,当使用Bilinear 算法的时候,要设置参数为num\_args=2。另一个非常重要的是：在权重参数初始化的时候，用的字典的 key 是up\_weight!!!

参考 [5]: 不能让 `mxnet.ndarray.UpSampling` 实现 Bilinear 插值

主要有两点值得关注的：

- From zhanghang1989: [PR](#) about BilinearResize2D 和 AdaptiveAvgPooling2d
- From [Stack Overflow Answers](#)

`mxnet.ndarray.UpSampling` seems to expect 2 inputs (1 input and 1 weight) for bilinear sample\_type.

Also, I think documentation for num\_args parameter is missing which you can check here: [Github Souce code](#)

This should work:

Listing 7.15: UpSample 例子 2

```

import mxnet as mx
import mxnet.ndarray as nd
xx = nd.random_normal(shape=[1,1,256,256], ctx=mx.cpu())
xx1 = nd.random_normal(shape=[1,1,4,4], ctx=mx.cpu())
temp = nd.UpSampling(xx,xx1, num_filter=1, scale=2, sample_type='bilinear', num_args=2)

```

参考 [6]: [nd.UpSampling 到底怎么使用 Bilinear](#)

UpSampling 的源码可以在 Github 上找到，里面说：

## 7.13 MXNet 中设置层的学习率

---

For bilinear upsampling this must be 2; 1 input and 1 weight.

后来改成这种就可以了：

Listing 7.16: UpSampling 的一个例子

```
scale = 2
seq.add(nn.Conv2DTranspose(128, kernel_size=(2*scale, 2*scale),
                          strides=(scale, scale), padding=(scale/2, scale/2),
                          weight_initializer=mx.init.Bilinear()))
```

参考 [7]: [UpSampling-Gluon 论坛](#)

重点。

这个问题下面的 Chinakook 回答的比较好。

严格意义的 Upsample 应该用 Conv2DTranspose, upsampling 到指定大小至少到现在我还没看到有任何提升。

```
upsampler = nn.Conv2DTranspose(channels=channels, kernel_size=4, strides=2,
                             padding=1, use_bias=False, groups=channels, weight_initializer=mx.init.Bilinear())
upsampler.collect_params().setattr('lr_mult', 0.)
```

### 7.12.1 总结

回顾上面的几个帖子，总结一下可以用以下三个方法：

- 使用 Conv2DTranspose + mx.init.Bilinear() + net.collect\_params().setattr('lr\_mult', 0) 实现, 可以见参考 [2, 6, 7]
- 使用 UpSample 实现, 可以见参考 [4, 5]
- 使用最新的BilinearResize2D

## 7.13 MXNet 中设置层的学习率

参考: [MXNet 如何在命令行设置层的学习率](#)

Gluon 的话可以用layer.weight.set\_lr\_mult 来改。

## 7.14 SoftmaxCrossEntropyLoss 释疑

API:

```
SoftmaxCrossEntropyLoss(axis=-1, sparse_label=True, from_logits=False,
                        weight=None, batch_axis=0, **kwargs)
```

主要说一下sparse\_label 和axis 两个参数。前者表明输入的 label 是整数 array 还是一个概率分布！

如果是一个整数, 那么损失的计算方式是:

$$L = - \sum_i \log p_{i,label_i}$$

此时输入的 label 是一个去掉 axis 哪一维同时保持其它维度的数据尺寸。而如果输入的是一个概率分布，那么交叉熵的计算方式是：

$$L = - \sum_i \sum_j^{classnumber} label_j \log p_{ij}$$

此时输入的 label 要与输入的 pred 的尺寸是完全一致的。

也就是说，如果输入是 sparse=True，则每一个 label 是一个整数标量；而如果输入的是 sparse=False，那么输入的每一个 label 是一个尺寸是 class number 的 array，表示该点的概率分布，这里的概率分布是 Multinoulli。

再说一下 axis，表示在该维度上进行计算。

## 7.15 MXNet 中的 3D CNN

### 7.15.1 应该怎么理解 3D CNN

参考：[卷积神经网络中的二维卷积核和三维卷积核有什么区别 - 知乎](#)

首先需要明确一点的是：在讨论卷积核的维度时，是不把 channel 维加进来的（或者说卷积核的维度指的是进行划窗操作的维度，而划窗操作是不在 channel 维度上进行的，因为每个 channel 共享同一个划窗位置，但每个 channel 上的卷积核权重是独立的）。所以，二维 conv 的卷积核其实是 (c, h, w)，3D 的卷积核就是 (c, d, h, w)，其中多出来的 d 就是第三维，根据具体应用，在视频中对应的就是时间维，在 CT 图像中对应就是层数维。

## 7.16 向 MXNet 添加新的操作

### 7.16.1 添加新的 Operator

参考 [1]: [mxnet 学习笔记 - 自定义 Op-CSDN](#)

需要注意以下几点：

#### 自定义 Op

自定义 Op 都需要去继承 operator.py 中的类，其中提供以下几类：

- CustomOp(object)
- CustomOpProp(object)
- NDArrayOp(PythonOp)
- NumpyOp(PythonOp)
- PythonOp(object)

## 7.16 向 MXNet 添加新的操作

这里可以看出，operator 分成两条路线，一条路线为 CustomOp，另一条路线为继承 PythonOp，下面分别介绍下。

### CustomOp 类

这条路线由三步组成。

- 第一步继承 CustomOp，重写方法forward() 和backward()
- 然后继承CustomOpProp 重写成员方法，并在方法create\_operator 中调用之前写好的 Op
- 第三步调用operator.register 对操作进行注册

下面就结合具体的代码来说一下：

第一步是继承 CustomOp 定义自己的 Operator 类，并重载forward() 和backward() 两个函数。

Listing 7.17: 基于 CustomOp 的 Softmax

```
class Softmax(mx.operator.CustomOp):
    # 这里的的 in_data 就是 Prop 里面的 list_augments
    # 返回的数据所包含的内容
    def forward(self, is_train, req, in_data, out_data, aux): #
        length of in_data: 2, 也就是 list_augments() 里面返回数据的
        length
        x = in_data[0].asnumpy() # x 的 shape 就是 Prop 里面的 data 的
        # shape,
        # 而 data 的 shape 就是 (batch_size, softmax 输入 data 的 shape! )
        # in_data[1] 的 shape 就是 Prop 里面 label 的 shape。在这里，具体的
        # shape 是 (batch_size, ), 即每一个样本对应一个 label!
        y = np.exp(x - x.max(axis=1).reshape((x.shape[0], 1))) # y 的 shape 变成 x 的 shape 也就是输入 data 的 shape
        # 下面这一步会涉及 Python 里面的 Broadcast 机制！
        y /= y.sum(axis=1).reshape((x.shape[0], 1))
        self.assign(out_data[0], req[0], mx.nd.array(y)) #
        # 输出的值就是 mx.nd.array(y), shape=x.shape
    def backward(self, req, out_grad, in_data, out_data, in_grad,
                aux):
        l = in_data[1].asnumpy().ravel().astype(np.int) # in_data
        # [1] 对应着 labels
        y = out_data[0].asnumpy() # 对应
        # forward() 的输出，也就是 out_augments()
        # 的返回值的第一个元素
        # 知乎上有推导，貌似 softmax 的求导只需要在响应位置上 -1 就可，
        # 并且其它地方的值是不变的！ !
        y[np.arange(l.shape[0])), 1] -= 1.0 # l.shape
        # [0] 对应着 batch_size, l 对应每个 sample 的 label 的值
        self.assign(in_grad[0], req[0], mx.ndarray(y)) # 返回的是
        # y, 对应的 shape 还是 forward() 里面 x 的 shape
```

其中，req 是 list of { 'null', 'write', 'inplace', 'add' }，确定如何 assign to out\_data. null means skip assignment etc. assign 函数定义在基类 CustomOp 中，定义如下：

```

def assign(self, dst, req, src):
    if req == 'null' # 什么也不做
        return
    elif req == 'write' or req == 'inplace':
        dst[:] = src
    elif req == 'add':
        dst[:] += src

```

第二步是继承 mx.operator.CustomOpProp，定义 SoftmaxProp 类，该类共包含六个函数(包含 `__init__`)，代码如下：

Listing 7.18: 第二步，基于 CustomOpProp 的 SoftmaxProp 类

```

@mx.operator.register("softmax")
class SoftmaxProp(mx.operator.CustomOpProp):
    def __init__(self):
        super(SoftmaxProp, self).__init__(need_top_grad=False)
    def list_arguments(self):
        return ['data', 'label']
    def list_outputs(self):
        return ['output']
    def infer_shape(self, in_shape): # 第一次调用时,
        会运行这个进行 shape 的推理
        data_shape = in_shape[0]
        label_shape = in_shape[0][0]
        output_shape = in_shape[0]
        # 返回的的三个 [] 表示 aux_data 的 shape
        return [data_shape, label_shape], [output_shape], []
    def infer_type(self, in_type): # 同样也是第一次调用时运行
        return in_type, [in_type[0]], []
    def create_operator(self, ctx, shapes, dtypes):
        return Softmax() # 返回 Softmax 类

```

上述代码就完成了对 Softmax 的自定义，在类 Softmax 中重写 `forward()` 和 `backward()`，这里与 caffe 中定义层操作类似，`forward` 中定义层的前向操作，`backward` 中定义反向传播的梯度计算。在完成定义之后，在类 SoftmaxProp 中的 `create_operator()` 调用并返回 `Softmax()` 实例。那么第三步 `register` 如何实现呢？可以看到 `SoftmaxProp` 中带有装饰器 `mx.operator.register()`，等价于操作 `register("custom_op")`(`CustomOpProp`)，这里即在代码运行前即完成了该 Op 的实例化，与 Optimizer 的装饰器类似，关于这个 `register` 的解释见单独介绍。

完成上述操作符的定义后，就可以使用新的操作符了，具体是使用 `mx.symbol.Custom()` 函数，具体的调用形式如下：

Listing 7.19: 第二步，基于 CustomOpProp 的 SoftmaxProp 类

```

data_sym = mx.symbol.Custom(data=fc3, name='softmax', op_type='softmax')

```

需要注意的地方在于：1) `fc3` 是 softmax 上一层网络的输出数据的名字，2)`name` 是本层 softmax 的名字，3) 是 `op_type`，这里指定的是自己上面新定义的操作类的名字，或者说是注册时用到的名字。

最后调用 `Module` 进行测试：

## 7.16 向 MXNet 添加新的操作

---

Listing 7.20: 第二步，基于 CustomOpProp 的 SoftmaxProp 类

```
mod = mx.mod.Module(data_sym, context=mx.cpu()) # 等  
mod.fit(...)
```

其中，`data_sym` 是 symbol 模型的最后的输出的变量的名字。

需要注意的地方是，例子程序里面的注释，以及 `register` 和两个类的创建过程!!

### PythonOp 类

在这条路线中，`PythonOp` 为基类，而我们大多数定义 `Op` 时不会去继承它，而是使用它的 subclass: `NDarrayOp`、`NumpyOp`。这条路线不需要像 `CustomOp` 那样需要三步。

具体的官网例子可以查看: [NDarray Softmax-github](#)

在这个例子中，类的定义为 `class NDarraySoftmax(mx.operator.NDArrayOp)`。类内部主要包含六个函数（一个为 `init` 函数）。

纵观整个实现代码，`NDarraySoftmax` 类继承自 `NDArrayOp`，后者其实与 `NumpyOp` 类似，不同之处在于 `forward()` 和 `backward()` 重写方式使用函数不停，`NDArrayOp` 中使用 `mx.nd` 中的操作，但 `NumpyOp` 使用 `numy` 中的操作。总之重点在 `forward()` 和 `backward()`。当然，如此的自定义方法在使用时需要预先定义类对象才可以使用，基于 `CustomOp` 中的定义时机不同。

成员方法 `list_augments()` 和 `list_outputs` 和 `infer_shape`。第一个函数主要对该 `Op` 定义是形参的命名，如代码中的 `['data', 'label']`，那么该 `Op` 在使用时形参必须为 `data` 和 `label`！这里也可以看出 `mxnet` 是使用名字寻找变量的，`DataIter` 和 `Optimizer` 也是如此。第二个函数定义了 `Op` 的输出变量的名字，一般命名规则是：`op_name + '_output'`。最后一个函数是用于在给定输入时，获取该 `Op` 输出的 `Shape`。当然，在自定义 `Op` 时，需要自己设计 `Op` 的输入和输出的 `Op`。在这个具体的例子中，输出包含两个 `List`，第一个 `List` 是输入 `data` 和 `label` 的 `Shape`，第二个 `List` 是输出 `output` 的 `Shape`。输入的 `Shape` 也就是包含 `data` 和 `label`。其中，`data` 的 `shape` 就是输入 `data` 的 `shape` 的第一个维度，也就是样本的个数，而 `label` 的 `shape` 是输入 `in_shape` 的第二个维度，也就是每个样本的 `Channel` 的个数；然后就是输出的 `shape`，这个数值根据 `Op` 的操作来确定，这里的具体数值等价于输入 `in_shape` 的第一个维度的取值，也就是样本的个数。

在定义好 `Op` 后，我们需要通过 `mx.mod.Module()` 将 `Op` 进行整合，并通过 `bind()` 来申请内存，在此之后，我们可以通过以下两种方法训练它：

- 分别调用 `init_params()` 初始化参数，调用 `init_optimizer()` 初始化 `Optimizer`，接下来就可以通过 `forward()` 和 `backward()` 进行前向、反向传播训练模块。
- 或者直接调用 `fit()` 方法进行训练，因为 `fit()` 中包含所需的初始化操作。

### 7.16.2 添加新的 Layer

参考 [1]: [MXNet 创建新的操作\(层\)-CSDN](#)

如果需要创建自定义层，有 3 个选择：

- 使用 Python 原生的语言和它的矩阵库 (`numpy` in `python`)。这不需要过多的能力和对 `MXNet` 的了解，但会影响性能

- 使用 mxnet 原生的语言和它的矩阵库 mxnet.rtc 和 mxnet.ndarray。这会得到更好的性能，但是相应的需要了解更多的 MXNet 的知识，可以通过 Python 来写 CUDA 的 Kernel 函数，并且在运行时编译
- 使用 C++/mshadow (CUDA)，这需要对 mxnet、mshadow 和 CUDA 都比较熟悉

参考里面给出了基于 Python/Numpy 的一个例子，步骤如下：

- 定义新的类，继承自 mx.operator.NumpyOp
- 定义函数，返回输入、输出的数据列表，名称分别为：list\_arguments(self) 和 list\_outputs(self)  
推荐的返回数据的顺序为：['input1', 'input2', ..., 'weight1', 'weight2', ...]
- 提供infer\_shape 来声明我们的 output/weight 并且检测输入的形状的一致性  
第一个维度总是 batch size.infer\_size 需要返回两列，即使他们是空的。
- 定义forward()、backward() 函数  
forward 函数含有两个参数：in\_data 和 out\_data，backward() 函数含有四个参数：out\_grad, in\_data, out\_data 和 in\_grad

## 如何创建新的网络层 -CSDN

参考 [2]: [如何创建新的网络层 -CSDN](#)

如果需要自定义一个网络层，比如新的损失函数，有两个选择：

- 借助前端语言(比如 Python)，使用 CustomOp 来写新的操作符。既可以运行在 CPU 上，也可以运行在 GPU 上。根据实际情况，性能可能很高，也可能很低。
- 使用 C++/mshadow(CUDA)。但前提是需要熟悉 MXNet、mshadow 和 CUDA，能获得最佳的性能。

这里还是以上面的具体的 Softmax 代码为例子。需要补充说明的一点是，assign 函数会根据 req 的取值来决定对 out\_data[0] 的赋值方式，req 取值包括：write, add 或 null 等。

Softmax 类定义了新的自定义操作符的计算，但仍然需要通过定义 mx.operator.CustomOpProp 的子类的方式，来定义它的 I/O 格式！！

在infer\_shape 里面，声明输出和权重的形状，并检查输入形状的一致性。输入/输出张量的第一维是数据的批量大小 (Batch size)。标签是一组整数，每个整数对应一个样本，并且输出、输入的形状相同。函数infer\_shape 的输出是三个列表，及时某一项为空，顺序也必须是：输入、输出和辅助状态。

重要的一点：如果想使用自定义操作符，需要创建 mx.sym.Custom 符号。其中，使用新操作符的注册名字来设置参数op\_type，具体代码如下：

```
mlp = mx.symbol.Custom(data=fc3, name='softmax', op_type='softmax')
```

## 7.16 向 MXNet 添加新的操作

---

### MXNet 添加新层 -CSDN

参考 [3]: [MXNet 添加新层 -CSDN](#)

具体的是，本文将开始讲一下如何基于 C++、CUDA 来添加新的操作，而不是上面的那些 Python 了！具体添加的操作时 Softplus 函数，然后就可以使用`mx.sym.Softplus`进行调用了。

首先额外说明一下 Softplus 的计算公式：

$$f(x) = \log(1 + \exp(x))$$

具体的步骤如下：

1. 在 `src/operator` 下面写好三个文件：`-inl.h`, `.cc`, `.cu` 等
2. 对于激活层，在 `src/operator/mshadow_op.h` 加入对`mshadow_op::softplus`、`mshadow_op::softplus_grad` 结构体的描述。

`struct softplus { ... }` 只需要修改 return 语句即可， $y=f(x)$ ，计算  $y$

`struct softplus_grad { ... }` 只需要修改 return 语句即可，计算  $f$  的导数  $f'(x)$   
具体代码如下：

Listing 7.21: 在 `mshadow_op.h` 里面添加结构体

```
/*! \brief Softplus */
struct softplus{
    template<typename DType>
    MSHADOW_XINLINE static DType Map(DType a)
    {
        return DType(log1pf(a))
    }
};

struct softplus_grad
{
    template<typename DType>
    MSHADOW_XINLINE static DType Map(DType a)
    {
        return DType(1.0f) - exp(-a)
    }
};
```

重新编译 mxnet 即可，将 `python/mxnet/lib` 库文件，重新拷贝到 `python2.7` 下的 `site-packages` 或 `Python3` 的 `site-packages`(或 `dist-packages`) 下。在 `mxnet` 文件夹外验证：

```
>>> import mxnet as mx
>>> help(mx.sym.Softplus)
```

源码分析。

MXNet 输入数据格式是 `TBlob`，输出数据格式是 `Tensor`，两者都是高维的数组类型。`mshadow` 用于数据存储结构。代码较多，内容待续。

## 7.17 关于 MXNet 里面的 register 修饰符

### 7.17.1 调用顺序 -CSDN 博客

在添加自己的 softmax 的例子中，一个需要值得注意的地方就是 register 装饰器的理解问题。这里还是参考一个 CSDN 博客！

参考 [1]: [MXNet-Python 源码中的装饰器理解 -CSDN](#)

这里主要注意的地方是调用顺序。简单来说，顺序如下：首先执行类方法 register，然后执行构建基类，在构建基类的派生类。

稍具体来说：调用类静态方法 (@staticmethod) 是不需要构建对象的，即调用基类的静态方法时，还不会创建基类的对象，而且在 debug 里面可以发现 register 是在调用这个基类的静态函数之前执行的。然后创建基类的对象，最后调用 create 函数什么的创建派生类。

总而言之一句话：装饰器运行在代码运行之前，即定义阶段就已经完成了。

### 7.17.2 Python 中的装饰器之 register

# **Chapter 8**

## **GPU Optimization in DL**

在存储优化上面，近三年的文章如下。In-place ABN 里面提到，一方面是提出高效的内存管理算法；另一个方向是通过 Training with reduced precision.

### **8.1 In-Place Activated BatchNorm for Memory-Optimized Training of DNNs**

作者提到了几种常见的优化方法。一个是本文所属的利用 In-place 和 Sharing Memory 来提高存储效率；另一个研究方向是用低精度或混合精度训练。其实还有一种方法是改进神经网络架构，如 The revesible residual network。对于部署，作者提到了可以使用 TensorRT。

### **8.2 Training deep nets with sublinear memory cost**

又名 Checkpointing。

### **8.3 Memory-efficient backpropagation through time**

### **8.4 The Reversible Residual Network: Backpropagation Without Storing Activations**

### **8.5**



# Chapter 9

## Tips in DL

重要的一点说明: CNN 中的卷积操作其实是信号处理中的相关操作 (Correlation Operation)。

### 9.1 Enlarge the FOV

增加网络的感受野。目前看到的主要方法如下:

- CRFs[19]
- Global Graph-reasoning module[2]
- Pooling
- Dilated conv[40]
- 

### 9.2 Upsampling

在卷积以及 Pooling 之后保持分辨率。目前看到的主要方法如下:

- Padding

- Deconvolution

- Uppooling

- Bilinear

-

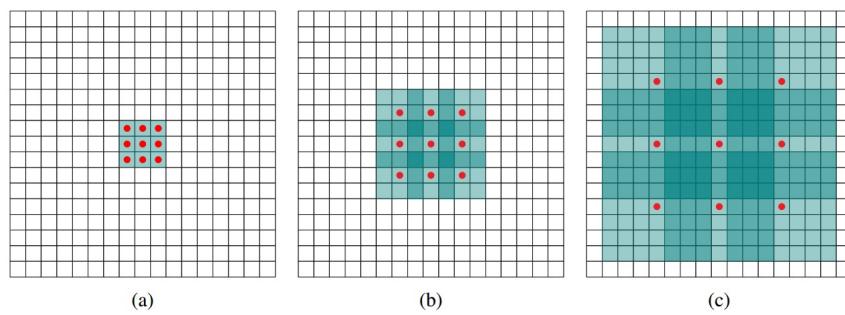
## 9.3 Multiscale Ability

在目标检测 (Object Detection) 中加入多尺度信息。记得的有以下几个方法：

- Pyramid Network
- Stacked CNN ?
- 

## 9.4 Dilated Convolution

主要原理如下。



**Fig 9.1.** Dilated Convolution 示意图

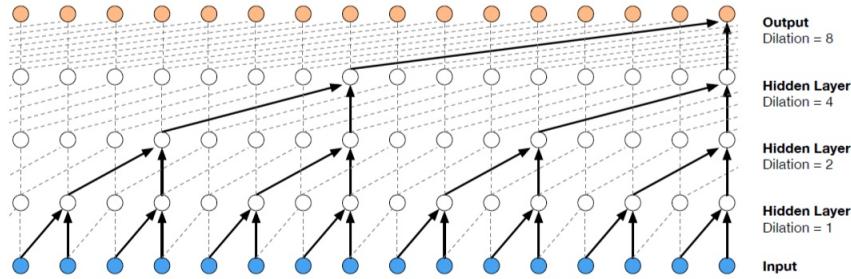
注意，下文提到的 **N-Dilated Conv** 中的  $N = 1, 2, 3, \dots$  是指图中相邻红点之间的间隔。

图9.1中，(a) 图对应  $3 \times 3$  的 1-dilated conv，和普通的卷积操作一样，(b) 图对应  $3 \times 3$  的 2-dilated conv，实际的卷积 kernel size 还是  $3 \times 3$ ，但是空洞为 1，也就是对于一个  $7 \times 7$  的图像 patch，只有 9 个红色的点和  $3 \times 3$  的 kernel 发生卷积操作，其余的点略过。也可以理解为 kernel 的 size 为  $7 \times 7$ ，但是只有图中的 9 个点的权重不为 0，其余都为 0。可以看到虽然 kernel size 只有  $3 \times 3$ ，但是这个卷积的感受野已经增大到了  $7 \times 7$ （如果考虑到这个 2-dilated conv 的前一层是一个 1-dilated conv 的话，那么每个红点就是 1-dilated 的卷积输出，所以感受野为  $3 \times 3$ ，所以 1-dilated 和 2-dilated 合起来就能达到  $7 \times 7$  的 conv），(c) 图是 4-dilated conv 操作，同理跟在两个 1-dilated 和 2-dilated conv 的后面，能达到  $15 \times 15$  的感受野。对比传统的 conv 操作，3 层  $3 \times 3$  的卷积加起来，stride 为 1 的话，只能达到  $(\text{kernel}-1) * \text{layer} + 1 = 7$  的感受野，也就是和层数 layer 成线性关系，而 dilated conv 的感受野是指数级的增长。

Dilated 的好处是不做 Pooling 算是信息的情况下，加大了感受野，让每个卷积核输出都包含较大范围的信息。在图像需要全局信息或者语音文本需要较长的 Sequence 信息依赖的问题中，都能很好的应用 Dilated Convolution，比如图像分割、语音合成 WaveNet、机器翻译 ByteNet。

WaveNet 的例子。

参考文献： [Dilated Conv 知乎](#)



**Fig 9.2.** Dilated Convolution 在 WaveNet 中的应用示意图

## 9.5 Deconvolutional Network

务必看补充部分的内容！

参考文献: [Deconvolution Networks](#)

可能应用的领域: Visualization, Pixel-wise Prediction, Unsupervised Learning, Image Generation.

大致可分为以下几个方面:

- Unsupervised Learning

其实是 Covolutional Sparse Coding. 这里的 Deconv 只是观念上和传统的 Conv 反向, 传统的 conv 是从图片生成 feature map, 而 deconv 是用 unsupervised 的方法找到一组 kernel 和 feature map, 让它们重建图片。

- CNN Visualization

通过 deconv 将 CNN 中 conv 得到的 feature map 还原到像素空间, 以观察特定的 feature map 对哪些 pattern 的图片敏感, 这里的 deconv 其实不是 conv 的可逆运算, 只是 conv 的 transpose, 所以 tensorflow 里一般取名叫 transpose\_conv。

- Upsampling

在 pixel-wise prediction 比如 image segmentation[4] 以及 image generation[5] 中, 由于需要做原始图片尺寸空间的预测, 而卷积由于 stride 往往会降低图片 size, 所以往往需要通过 upsampling 的方法来还原到原始图片尺寸, deconv 就充当了一个 upsampling 的角色。

下面主要介绍这三个方面的论文。

### 9.5.1 Convolutional Sparse Coding

#### 第一篇: Deconvolutional Networks

主要用于学习图片的中低层级的特征表示, 属于 Unsupervised Feature Learning。更多内容参考本小节的参考文献。

### 9.5.2 CNN 可视化

ZF-Net 中利用 Deconv 来做可视化, 它是将 CNN 学习到的 Feature Map 的卷积核, 取转置, 将图片特征从 Feature Map 空间转化到 Pixel 空间, 用于发现哪些 Pixel 激活了特定的 Feature Map, 达到分析理解 CNN 的目的。

### 9.5.3 Upsampling

用于 FCN[18] 和 DCGAN。

### 9.5.4 补充

一个非常好的可以看到多种卷积操作动作图的资源: [Convolution Arithmetic Github](#)

上面的东西还是没有说明白 Deconvolution 到底是怎么回事啊。

实际使用中, 反卷积会引起棋盘状的 Artifacts。所以要采用上采样卷积层。

上面这句话是什么意思?

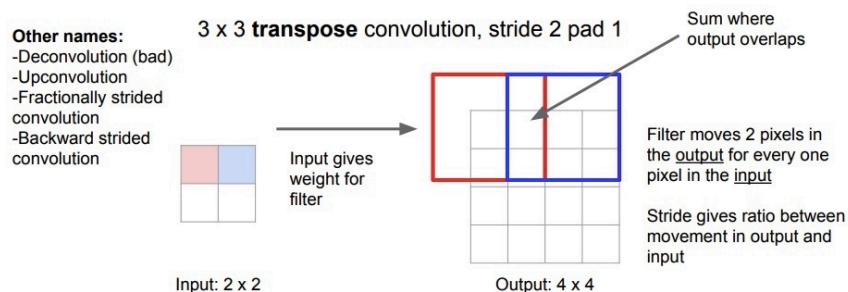
### 9.5.5 Deconvolution 与 Upsample 的区别

参考文献: [Caffe 中的 Deconvolution 和 Upsample 区别 -知乎](#)

高票回答:

Deconvolution

Learnable Upsampling: Transpose Convolution



**Fig 9.3.** Transpose Convolution 过程示意图

Input pixel \* filter = output window, 不同 output window 重合的部分使用 sum 叠加处理。

这一解释和 caffe 的定义保持一致, caffe 中定义解释过来就是: “DeconvolutionLayer 逐像素地将输入值乘上一个 filter, 并将结果输出 windows 叠加起来”

Convolve the input with a bank of learned filters, and (optionally) add biases, treating filters and convolution parameters in the opposite sense as ConvolutionLayer. ConvolutionLayer computes each output value by dotting an input window with a filter; DeconvolutionLayer multiplies each input value by a filter elementwise, and sums over the resulting output windows. In other words, DeconvolutionLayer is ConvolutionLayer with the forward and backward passes reversed. DeconvolutionLayer reuses ConvolutionParameter for its parameters, but they take the opposite sense as in ConvolutionLayer (so padding is removed from the output rather than added to the input, and stride results in upsampling rather than downsampling).

Upsampling

## 9.5 Deconvolutional Network

---

该层权重通过 BilinearFiller 初始化，因此当学习率为 0 时，权重在训练过程中保持初始值不变，一一直作为 bilinear resize 的作用。

Listing 9.1: Bilinear filter initializer in MXNet

```
class Bilinear(Initializer):
    """Initialize weight for upsampling layers."""
    def __init__(self):
        super(Bilinear, self).__init__()
    def _init_weight(self, _, arr):
        weight = np.zeros(np.prod(arr.shape), dtype='float32')
        shape = arr.shape
        f = np.ceil(shape[3] / 2.)
        c = (2 * f - 1 - f % 2) / (2. * f)
        for i in range(np.prod(shape)):
            x = i % shape[3]
            y = (i / shape[3]) % shape[2]
            weight[i] = (1 - abs(x / f - c)) * (1 - abs(y / f - c))
        arr[:] = weight.reshape(shape)
```

James Liu 的回答：

- **Deconvolution**

上采样就是把  $[W, H]$  大小的 feature map  $F_{W,H}$  扩大为  $[nW, nH]$  尺寸大小的  $\hat{F}_{nW,nH}$ ，其中  $n$  为上采样倍数。那么可以很容易的想到我们可以在扩大的 feature map  $\hat{F}$  上每隔  $n$  个位置填补原 F 中对应位置的值。

但是剩余的那些位置怎么办呢？deconv 操作是把剩余位置填 0，然后这个大 feature map 过一个 conv。

所以：Deconv = 扩大 + 填 0 + Convolution

- **Upsampling**

插值上采样 = 扩大 + 插值

### 理解深度学习中的 Deconvolution Networks

参考文献：[理解深度学习中的 Deconvolution Networks - 知乎](#)

逆卷积 (Deconvolution) 比较容易引起误会，转置卷积 (Transposed Convolution) 是一个更为合适的叫法。

输入矩阵可展开为 16 维向量，记作  $x$

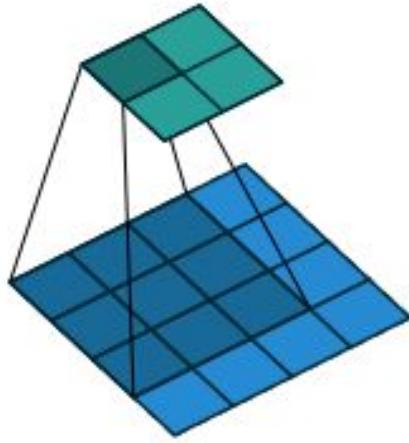
输出矩阵可展开为 4 维向量，记作  $y$

卷积运算可表示为  $y = Cx$

不难想象  $C$  其实就是如下的稀疏阵

那么当反向传播时又会如何呢？首先我们已经有从更深层的网络中得到的  $\frac{\partial Loss}{\partial y}$ 。

$$\frac{\partial Loss}{\partial x_j} = \sum_i \frac{\partial Loss}{\partial y_i} \frac{\partial y_i}{\partial x_j} = \sum_i \frac{\partial Loss}{\partial y_i} C_{i,j} = \frac{\partial Loss}{\partial y} \cdot C_{*,j} = C_{*,j}^T \frac{\partial Loss}{\partial y}$$



**Fig 9.4.** 一个例子，用于卷积操作说明

$$\begin{pmatrix} w_{0,0} & w_{0,1} & w_{0,2} & 0 & w_{1,0} & w_{1,1} & w_{1,2} & 0 & w_{2,0} & w_{2,1} & w_{2,2} & 0 & 0 & 0 & 0 \\ 0 & w_{0,0} & w_{0,1} & w_{0,2} & 0 & w_{1,0} & w_{1,1} & w_{1,2} & 0 & w_{2,0} & w_{2,1} & w_{2,2} & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & w_{0,0} & w_{0,1} & w_{0,2} & 0 & w_{1,0} & w_{1,1} & w_{1,2} & 0 & w_{2,0} & w_{2,1} & w_{2,2} & 0 \\ 0 & 0 & 0 & 0 & 0 & w_{0,0} & w_{0,1} & w_{0,2} & 0 & w_{1,0} & w_{1,1} & w_{1,2} & 0 & w_{2,0} & w_{2,1} & w_{2,2} \end{pmatrix}$$

**Fig 9.5.** 卷积操作时的矩阵形式

回想第一句话，你猜的没错，所谓逆卷积其实就是正向时左乘  $C^T$ ，而反向时左乘  $(C^T)^T$ ，即 C 的运算。

为什么 CS321n 里面说要使用 Convolution Transpose 而不是 Deconvolution 呢：  
反卷积的数学含义，通过反卷积可以将通过卷积的输出信号，完全还原输入信号。

而事实是，转置卷积只能还原 shape 大小，不能还原 value.

### 9.5.6 Deconvolution 输出的大小计算

好吧，我初步知道是怎么做的了：

首先看正向卷积时，假设输入时  $1 * 1 * N_{in} * N_{in}$  得到的 Feature Map 是  $(1 * 1 * N_{out} * N_{out})$ ，该过程中，假设步长为  $S$ ，卷积核为  $K$ , Padding 为  $P$ ，那么输入输出大小的关系是：

$$N_{out} = \frac{N_{in} - K + 2 * P}{S} + 1$$

那么在转置卷积时，要得到与输入同等大小的 Feature Map，需要确定 Transpose Convolution 的步长、卷积核大小、Padding 的大小，那么应该怎么求呢，只需要把上式取反函数就行：

$$N_{in} = (N_{out} - 1) * S + K - 2 * P$$

一般来说， $S$  取正向卷积计算中的 Strides 大小， $K$  也可以去相同的，也可以取不同的，这时候只需要确定 K 与 P 的合适关系就行。

起码 MXNet 中的 Conv2Transpose 可以这样设置参数。

### 9.5.7 Conv2DTranspose 用于成倍的提高分辨率的时候

参考文献：UpSampling 中使用 Bilinear 到底怎么用 -Gluon

## 9.6 Dilated Network 与 Deconv Network 之间的区别

---

一种简单的做法，假设需要提升的分辨率的倍数是  $scale$  那么设置 Transpose Convolution 的参数如下：

```
gluon.nn.Conv2DTranspose(output_channels, kernel_size=(2*scale, 2*scale),
strides=(scale, scale), padding=(scale/2, scale/2),
weight_initializer=mx.init.Bilinear())
```

比如，要把 Feature Map 放大 2 倍，则  $scale = 2$  带入上面的代码就行。

## 9.6 Dilated Network 与 Deconv Network 之间的区别

Dilated Convolution 主要用于增加感受野，而不是 Upsampling；Deconv Network 主要用于 Upsample，即增加图像分辨率。

对于标准的  $k \times k$  的卷积操作，stride 为  $s$ ，分为一下几种情况：

- $s > 1$

即卷积的同时做了降采样，输入 Feature Map 的分辨率<sup>1</sup>下降。但这一般也会增加感受野。

- $s = 1$

普通的步长为 1 的卷积，输入与输出分辨率相同。

- $0 < s < 1$

Fractionally strided convolution. 相当于图像做 upsampling。比如  $s = 0.5$  时，意味着图像像素之间 padding 一个空白的像素（像素值为 0）后，stride 改为 1 进行卷积，达到一次卷积看到的空间范围变大的目的。

## 9.7 目标检测中的 mAP 的含义

- 对于类别 C，在一张图像上

首先计算 C 在一张图像上的精度。

$$Precision_C = \frac{N(TP)_C}{N(Total)_C}$$

其中， $Precision_C$  为类别 C 在一张图像上的精度。 $N(TP)_C$  为算法检测正确 (True Positive) 的 C 的个数，检测是否正确按照  $IoU > 0.5$  算，同理， $N(Total)_C$  为这一张图像所有 C 类的个数。所以则一步，仅涉及一个类别 C 以及一张图像。

- 对于类别 C，在多张图像上

这一步计算的是类别 C 的 AP 指数。

$$AveragePrecision_C = \frac{\sum Precision_C}{N(TotalImage)_C}$$

---

<sup>1</sup>分辨率是指像素的多少，而尺度是指模糊程度的大小，即 Gaussian Filter 中的方差  $\delta$

其中,  $AveragePrecision_C$  是类别  $C$  的  $AP$  指数,  $Precision_C$  为上文计算得到的类别  $C$  的在一张图像上的精度, 然后对所有包含类别  $C$  的图像上的  $C$  的精度  $Precision_C$  求和;  $N(TotalImage)_C$  为包含类别  $C$  的图像的数量, 也对应于分子中求和所涉及的图像。

- 在整个数据集上, 多个类别

$mAP$  在上一步的计算结果的基础上, 计算所有类别的  $AP$  和 / 总的类别数。

$$meanAveragePrecision = \frac{\sum_C AveragePrecision_C}{N(Class)}$$

也就是相当于计算所有类别的  $AP$  的平均值, 是对应于类别总数的平均值。

参考文献: [知乎文章](#)

## 9.8 统计学习方法

一个比较好的总结: [机器学习常见算法个人总结](#)

## 9.9 Distillation Module

文献来源: [39][21]

在 [39] 中, 同时完成深度估计以及场景解析两个任务。

Distillation Module 的目的:

- Deep-model distillation modules fuses information from the intermediate predictions for each specific final task[39]. 高效的利用中间任务的信息互补。文章 [39] 提出的三种不同的实现方式如图9.6所示。

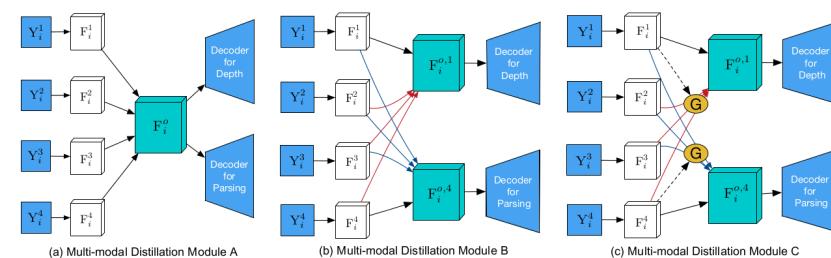


Figure 3. Illustration of the designed different multi-modal distillation modules. The symbols  $Y_i^1, Y_i^2, Y_i^3, Y_i^4$  represent the predictions corresponding to multiple intermediate tasks. The distillation module A is a naive combination of the multiple predictions; the module B proposes a mechanism of passing message between different predictions; the module C shows an attention-guided message passing mechanism for distillation. The symbol G denotes a generated attention map which is used as guidance in the distillation.

**Fig 9.6.** 三种不同的 Distillation Module

- Distillation loss function[21]

利用 Distillation 帮助将 Teacher Network(精度更高) 的知识迁移到的 Student Network。

## 9.10 光流估计中的 Average end-point error

---

### 9.9.1 Knowledge Distillation

#### 什么是 Distilling the knowledge

一句话总结，就是用 teacher network 的输出作为 soft label 来训练一个 student network.

#### Disilling the knowledge in a Neural Network

$$q_i = \frac{\exp(z_i/T)}{\sum_j \exp(z_j/T)}$$

其实就是一个 Softmax，值得注意的是  $T$  是 Temperature， $T$  越大，概率分布就越 Soft。在训练 Student Network 时，该概率分布就是 Student Network 的 soft label.

Student Network 的训练策略：

- 先用 Teacher Network 的概率分布训练
- 再用 Real Label 训练

### 9.9.2 Recurrent Knowledge Distillation [30]

## 9.10 光流估计中的 Average end-point error

貌似就是类似于均方误差类似。具体定义还没查到。

## 9.11 CNN 中的卷及方式汇总

参考文献：CNN 中的卷积方式 2017.9-知乎

### 9.11.1 Inception

如图9.7所示。主要思想是，融合 Network in Network 的思想来增加隐层提升非线性表达的思想。先用  $1*1$  的卷积映射到隐空间，再在隐空间做卷积操作。同时考虑多尺度，在单层 Inception 中，用多个大小不同的卷积核做卷积，然后把结果 Concat 起来。

代表模型：

- Inception-V1

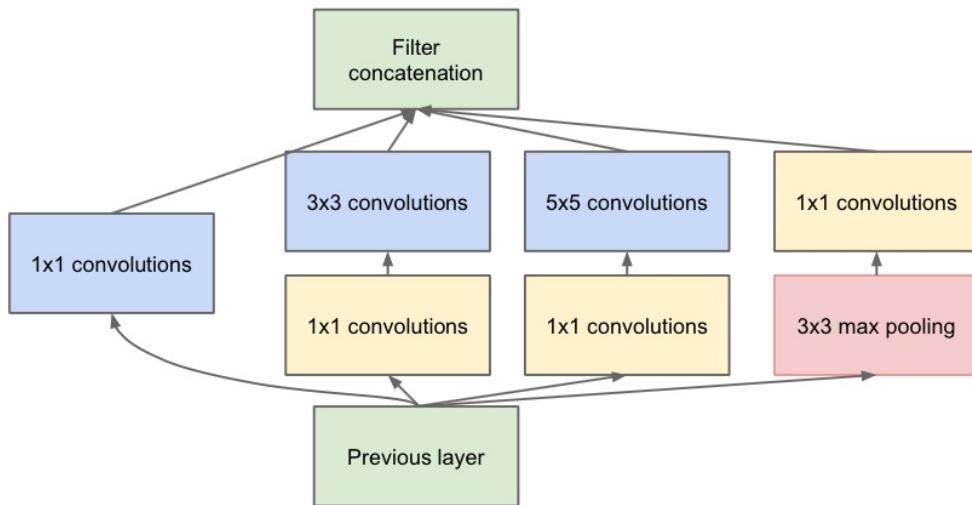
就是把图9.7中的结构进行 Stack,GoogLeNet?

- Inception-V2

加入了 Batch Normalization 正则，去除了  $5*5$  卷积，用两个  $3*3$  卷积代替

- Inception-V3

$7 \times 7$  卷积被拆分成  $7 * 1 + 1 * 7$ , 可分离卷积?



**Fig 9.7.** Inception 卷积结构

- Inception-V4  
加入了残差结构。

### 9.11.2 空洞卷积, Dilation

Dilation 卷积，通常译作空洞卷积或者卷积核膨胀操作，它是解决 pixel-wise 输出模型的一种常用的卷积方式。一种普遍的认识是，pooling 下采样操作导致的信息丢失是不可逆的，通常的分类识别模型，只需要预测每一类的概率，所以我们不需要考虑 pooling 会导致损失图像细节信息的问题，但是做像素级的预测时（譬如语义分割），就要考虑到这个问题了。

所以就要有一种卷积代替 pooling 的作用（成倍的增加感受野），而空洞卷积就是为了做这个的。通过卷积核插“0”的方式，它可以比普通的卷积获得更大的感受野。

所以，本意是为了增加感受野。

应用：

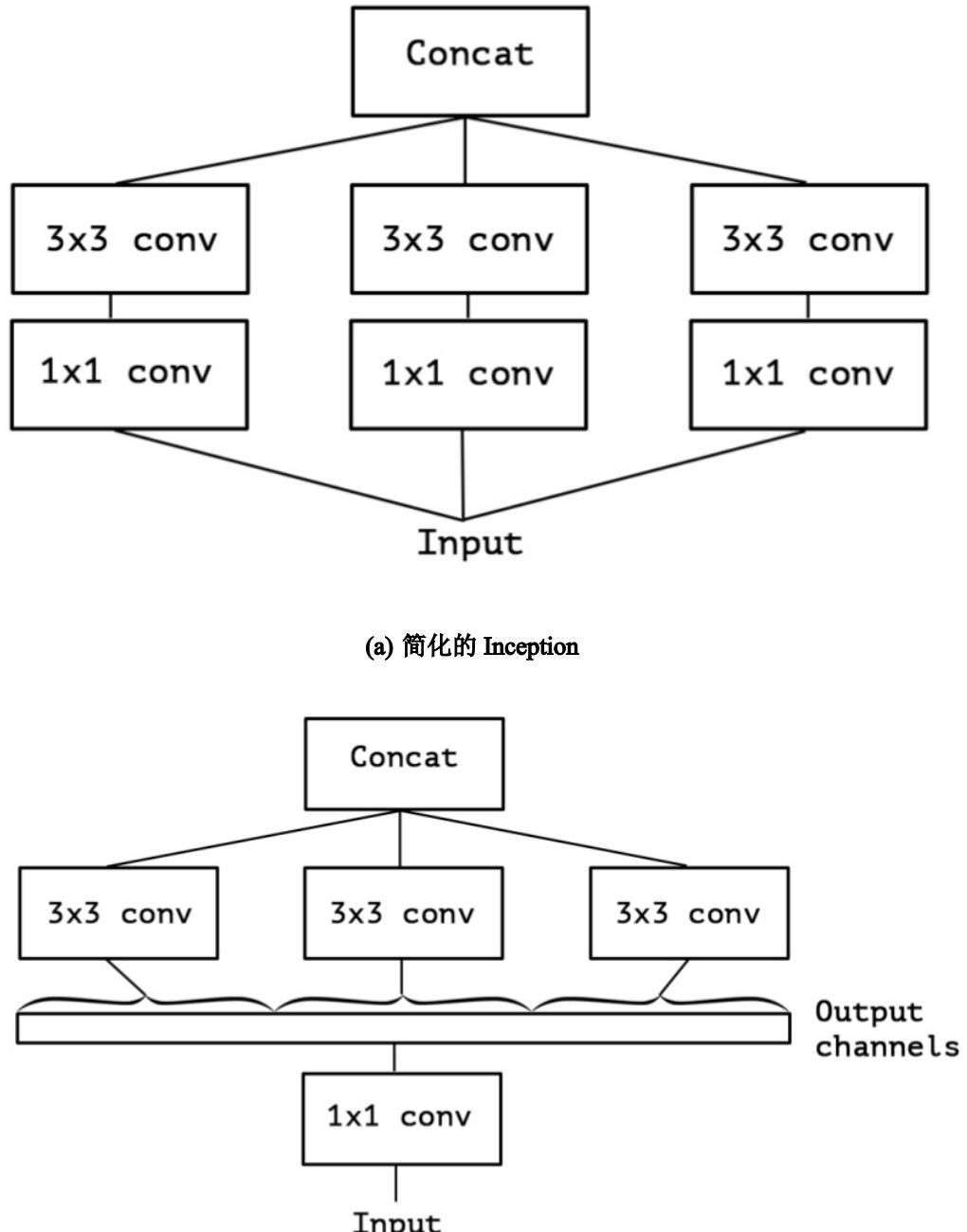
- FCN
- WaveNet

### 9.11.3 深度可分离卷积, Depthwise Separable Convolution

是 Inception 的延续。

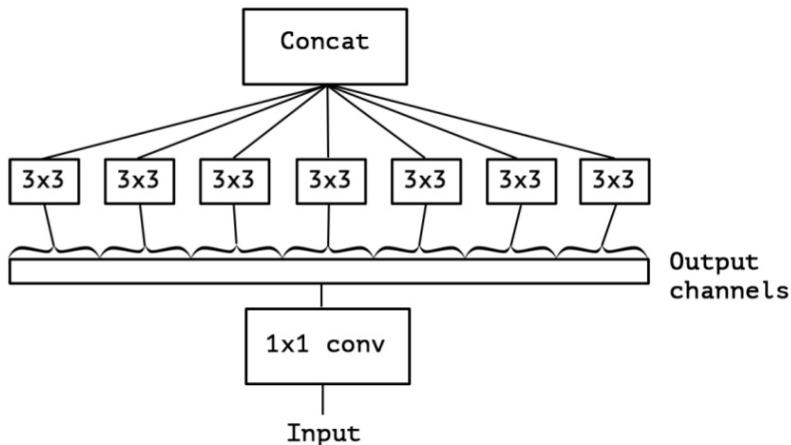
如图9.8所示，我们又可以看做，把一整个输入做  $1*1$  卷积，然后切成三段，分别  $3*3$  卷积后相连。注意的是，在三个不同的卷积时，是对 Channel 进行分组进行  $3*3$  卷积。

OK，现在我们想，如果不是分成三段，而是分成 5 段或者更多，那模型的表达能力是不是更强呢？于是我们就切更多段，切到不能再切了，正好是 Output channels 的数量（极限版本，图9.9）：



(b) Depthwise Separable Convolution 的示意图，可以看出来，与 Inception 很像。

**Fig 9.8.** Inception 结构与 Depthwise Separable Convolution 结构对比



**Fig 9.9.** Channel 分组的极限版本

于是，就有了深度卷积（depthwise convolution），深度卷积是对输入的每一个 channel 独立的用对应 channel 的所有卷积核去卷积，假设卷积核的 shape 是 [filter\_height, filter\_width, in\_channels, channel\_multiplier]，那么每个 in\_channel 会输出 channel\_multiplier 那么多个通道，最后的 feature map 就会有 in\_channels \* channel\_multiplier 个通道了。反观普通的卷积，输出的 feature map 一般就只有 channel\_multiplier 那么多个通道。

也就是说，对于每一个 Channel，都用不同的多个卷积核进行卷积，具体的是 Channel\_multiplier 个不同的卷积核。

既然叫深度可分离卷积，光做 depthwise convolution 肯定是不够的，原文在深度卷积后面又加了 pointwise convolution，这个 pointwise convolution 就是 1\*1 的卷积，可以看做是对那么多分离的通道做了个融合。

这两个过程合起来，就称为 Depthwise Separable Convolution 了。

应用：

- Xception

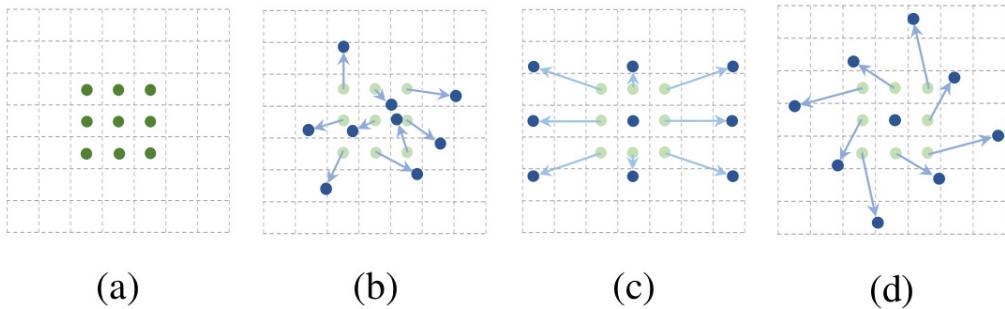
#### 9.11.4 可变性卷积

可形变卷积的思想很巧妙：它认为规则形状的卷积核（比如一般用的正方形 3\*3 卷积）可能会限制特征的提取，如果赋予卷积核形变的特性，让网络根据 label 反传下来的误差自动的调整卷积核的形状，适应网络重点关注的感兴趣的区域，就可以提取更好的特征。

如图9.10：网络会根据原位置（a），学习一个 offset 偏移量，得到新的卷积核（b）（c）（d），那么一些特殊情况就会成为这个更泛化的模型的特例，例如图（c）表示从不同尺度物体的识别，图（d）表示旋转物体的识别。

具体实现如下。

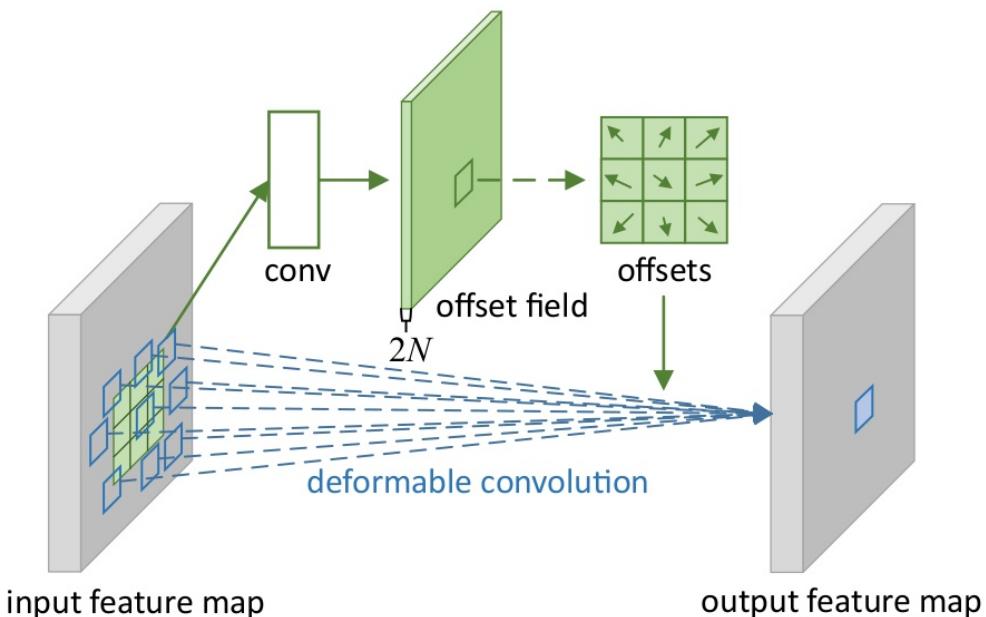
图9.11中包含两处卷积。第一处是绿色的获取 Offsets 的卷积，即图中上方部分，这一处的输入是 Input feature map，得到一个输出 (Offset Field)，然后再在这个输出上取一个卷积核大小的切片，作为卷积核对应位置的 Offset。若 Input Feature 的尺寸为：Batch, height, width, channels，其中，Batch 表示批数，Channel 表示输入的通道数，然后输出通道变为两倍，卷积得到的 Offset Field 就是 Batch, height, width, 2 \*



**Fig 9.10. Deformable Convolution 示意图**

*channels*, 那么为什么会翻倍呢, 也就是图中上半部分中的  $2N$  是什么意思呢? 首先要确定一个卷积核的 Offset, 需要在不同的方向上单独确定, 比如 X, Y 方向, 所以就变成二倍了。在实际进行 Deformable 时, 首先从 Offset Field 中对应位置上取一个对应卷积核的切片, 如果卷积核是  $3 * 3$ , 那么我们在两个方向都取一个  $3 * 3$  的切片, 代表来年各个方向的 Offset, 然后就完成卷积核形变了。

第二处的卷积, 是图中下方的表示, 就是一个常规的卷积运算, 只不过卷积核是经过上述 Offset 之后的变形卷积核。这里还有一个用双线性插值的方法来获取某一卷积形变后位置的输入的过程。



**Fig 9.11. Deformable Convolution 的实现示意图**

应用:

- Deformable Convolutional Networks

可能会跟目标检测、跟踪相结合。

### 9.11.5 特征重标定卷积

在 ImageNet2017 比赛中，冠军模型 SENet 的核心模块，被称为" Squeeze-and-Excitation"，知乎的作者把它就先成为特征重标定卷积了。

和前面的不同，本文提出的算法的出发点在于改进特征为度，包括卷积核的数量等。现在一个卷积层中，还不是整个神经网络，有数以千计的卷积核，而且我们知道每一种卷积核对应提取一种特征，但得到这么多的特征，肯定有一些是更重要的，有一些是不那么重要的。所以本文的方法是通过学习方式来自动获取每个特征通道的重要程度，然后按照计算出来的重要程度去提升有用的特征并抑制对当前任务用处不大的特征。

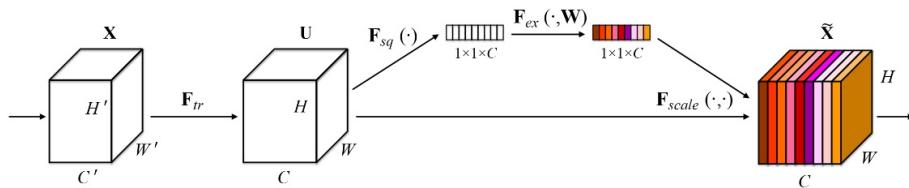


Figure 1. A Squeeze-and-Excitation block.

**Fig 9.12.** Squeeze-and-Excitation Block 示意图

虽然听上去很复杂，但实现起来却比较简单？步骤如下：

- 首先对输入  $X$  做常规的卷积  $F_{tr}$ ，得到一个 Output Feature Map ( $U$ )，它的大小为  $C, H, W$ ，文章的作者认为，得到的这个 Output 是非常混乱的。
- 为了得到这些 Feature Maps(共  $C$  个)的重要程度，直接对这些 Feature Map 做一个 Global Average Pooling<sup>2</sup>，然后就得到一个长度为  $C$  的向量，注意是向量了！在图9.12中表现就是由  $U$  经操作  $F_{sq}(\cdot)$  得到  $1 \times 1 \times C$  的过程。
- 然后我们对这个向量加两个 FC 层，做非线性映射，这俩 FC 层的参数，也就是网络需要额外学习的参数。对应图中  $F_{ex}(\cdot, W)$  操作。
- 最后输出的向量，我们可以看做特征的重要性程度，然后与 feature map 对应 channel 相乘就得到特征有序的 feature map 了。对应图中  $F_{scale}(\cdot, \cdot)$  操作。

应用：

- Squeeze-and-Excitation Networks
- 另外它还可以和几个主流网络结构结合起来一起用，比如 Inception 和 Res。图9.13所示。

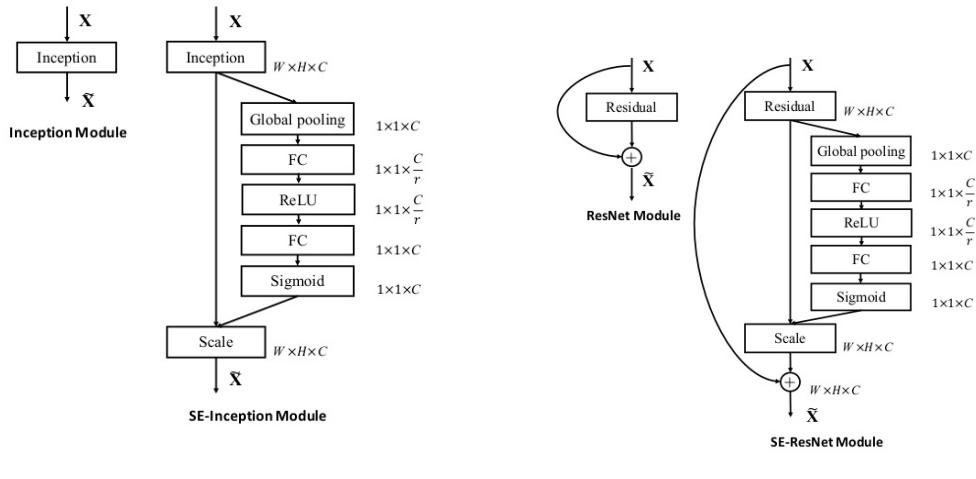
### 9.11.6 小结 -比较

图9.14是对上面提到的不同的卷及类型的一个比较与总结。

我们把图像 (height, width) 作为空间维度，把 channels 做为特征维度。

<sup>2</sup>这里还涉及到一个额外的东西，如果你了解卷积，你就会发现一旦某一特征经常被激活，那么 Global Average Pooling 计算出来的值会比较大，说明它对结果的影响也比较大，反之越小的值，对结果的影响就越小。

## 9.11 CNN 中的卷积方式汇总



**Fig 9.13. SENet 与 Inception 以及 ResNet 结合的示意图**

	参数数量 (输出大小不变的情况下)	切入点	层数
多尺度非线性卷积	增加很多	多尺度 (空间维度)	$\geq 2$
空洞卷积	不变	感受野 (空间维度)	1
深度可分离卷积	减少	空间维度 / 特征维度	$\geq 2$
可变形卷积	增加	空间维度	$\geq 2$
特征重标定卷积	增加	特征维度	$\geq 2$

**Fig 9.14. 不同卷积策略的比较**

## 9.12 全连接与卷积的异同

全连接：Fully Connected

注意这里与 FCN 中的 FC 不同，FCN 里面的 FC 是 Fully Convolution，所以还是卷积操作，所以才有用 Fully Convolution 代替 Fully Connected 之说。

Fully Connected 的作用是什么？其本质上还是一个卷积运算，只不过输入卷积核的大小跟最后一层 Feature Map 的大小一致，所以得到的结果是一个标量。其实就是对前面 CNN 网络提取的特征进行变换，对这些 Feature maps 进行组合，得到目标类的一个代表值，输入 Sigmoid 函数进行计算，得到分类结果。但这种方式参数非常多，因为在实现 Fully Connected 时需要根据最后一层 Feature Map 的大小来确定计算的卷积核，这也导致了它需要解决不同输入大小时候的数据问题。

一个简单的例子，输入是  $228 * 228 * 3$ ，然后最后一层 Feature Map 的维度是  $7 * 7 * 512$ ，即，共包含 512 个 Feature Maps，每一个 Feature Map 的大小是  $7 * 7$ ，那么全连接层需要这样设计，假设我们想要输出 1024 个分类，那么全连接的参数的数量就是： $7 * 7 * 512 * 1024$ ，所以参数非常多！另一方面，如果输入不是  $228 * 228 * 3$  而是  $456 * 456 * 3$  那么，得到的最后一层 Feature map 的大小也不是  $7 * 7 * 512$ ，现在而是  $14 * 14 * 512$ ，那么全连接也需要改变，变成  $14 * 14 * 512 * 1024$ ，这样非常不灵活。

因为参数实在太多，所以现在在最后为了得到 Sigmoid 函数的输入（标量），都选择使用 Global Average Pooling 来代替全连接。Global Average Pooling 也就是说用一个 Feature Map 的平均值作为 Sigmoid 输入，输出为类别信息。

不过，也有研究人员表明，全连接层有助于在微调（Fine-tune）过程中进行知识迁移，尤其源领域与目标领域很不一样的时候，更是如此。

## 9.13 Pooling

### Global Average Pooling

就是对一个 Feature Map 进行加和求平均值。具体的应用可参考上一小节。

### Unpooling

而上池化的实现主要在于池化时记住输出值的位置，在上池化时再将这个值填回原来的位置，其他位置填 0 即 OK。（参考：SegNet, DeconvNet）

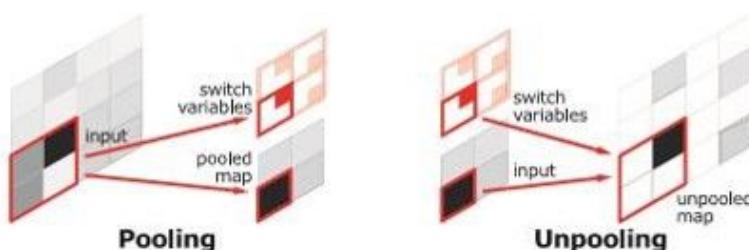


Fig 9.15. Unpooling 示意图

图9.15可以认为额外的switch variables实现了保存 Pooling 过程中的位置。

## 9.14 Local Response Normalization

---

In the convnet, the max pooling operation is non-invertible, however we can obtain an approximate inverse by recording the locations of the maxima within each pooling region in a set of switch variables. In the deconvnet, the unpooling operation uses these switches to place the reconstructions from the layer above into appropriate locations, preserving the structure of the stimulus.

也就是说用一组开关变量保存最大值在 Pooling Region 中的位置。

参考文献: [Quora Answer](#)

## 9.14 Local Response Normalization

Reference: [What is LRN](#)

为什么需要设置 Normalization Layers? Anyway, the reason we may want to have normalization layers in our CNN is that we want to have some kind of inhibition scheme. 这个 Inhibition Scheme 是什么意思。

侧边抑制: lateral inhibition。一个激活的神经会抑制旁边神经的激活。

### 到底什么是 LRN

Local Response Normalization (LRN) layer implements the lateral inhibition we were talking about in the previous section. This layer is useful when we are dealing with ReLU neurons. Why is that? Because ReLU neurons have unbounded activations and we need LRN to normalize that. We want to detect high frequency features with a large response. If we normalize around the local neighborhood of the excited neuron, it becomes even more sensitive as compared to its neighbors.

At the same time, it will dampen(抑制) the responses that are uniformly large in any given local neighborhood(值普遍很大的局部). If all the values are large, then normalizing those values will diminish all of them. So basically we want to encourage some kind of inhibition and boost the neurons with relatively larger activations.

### 如何实现 LRN

There are two types of normalizations available in Caffe. You can either normalize within the same channel or you can normalize across channels. Both these methods tend to amplify the excited neuron while dampening the surrounding neurons. When you are normalizing within the same channel, it's just like considering a 2D neighborhood of dimension  $N \times N$ , where  $N$  is the size of the normalization window. You normalize this window using the values in this neighborhood. If you are normalizing across channels, you will consider a neighborhood along the third dimension but at a single location. You need to consider an area of shape  $N \times 1 \times 1$ . Here  $1 \times 1$  refers to a single value in a 2D matrix and  $N$  refers to the normalization size.

在 AlexNet 中 Normalized 的计算公式如下:

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

其中,  $a_{x,y}^i, b_{x,y}^i$  分别是输入的激活值, 输出的 Normalized 的值。

## 9.15 CNN 中感受野的计算

参考文献：

[1] CNN 中感受野的计算 -CSDN

[2] CNn 中感受野的计算 -博客园

感受野（receptive field）是怎样一个东西呢，从 CNN 可视化的角度来讲，就是输出 featuremap 某个节点的响应对应的输入图像的区域就是感受野。

比如我们第一层是一个  $3 \times 3$  的卷积核，那么我们经过这个卷积核得到的 featuremap 中的每个节点都源自这个  $3 \times 3$  的卷积核与原图像中  $3 \times 3$  的区域做卷积，那么我们就称这个 featuremap 的节点感受野大小为  $3 \times 3$

如果再经过 pooling 层，假定卷积层的 stride 是 1，pooling 层大小  $2 \times 2$ ，stride 是 2，那么 pooling 层节点的感受野就是  $4 \times 4$

有几点需要注意的是，padding 并不影响感受野，stride 只影响下一层 featuremap 的感受野，size 影响的是该层的感受野。

具体计算时，需要：

- 第一层卷积层的输出特征像素的感受野的大小就是滤波器的大小
- 深层卷积层的感受野大小和它之前的所有层的滤波器大小和步长有关系
- 计算感受野大小时，忽略图像边缘的影响，即不考虑 Padding 的大小。

关于感受野的计算，多采用 Top to Down 的方式，即最先计算最深层的前一层的感受野，然后逐渐传递到第一层，使用的公式如下：

1.  $RF = 1 //$  待计算的 Feature Map 的感受野大小

2. For layer in (top layer to down layer):

3.  $RF = ((RF - 1) * stride) + fsize$

stride 表示卷积的步长；fsize 表示卷积层滤波器的大小。

具体的例子

type	size	stride
conv1	3	2
pool1	2	2
conv2	3	1
pool2	2	2
conv3	3	1
conv4	3	1
pool3	2	2

pool3 的一个输出对应 pool3 的输入大小为  $2 \times 2$

感受野计算如下：

依次类推，对应 conv4 的输入为  $4 \times 4$ ，因为  $2 \times 2$  的每个角加一个  $3 \times 3$  的卷积核，就成了  $4 \times 4$ ，当然这是在 stride=1 的情况下才成立的，但是一般都是 stride=1，不然也不合理

## 9.15 CNN 中感受野的计算

---

对应 conv3 的输入为 6\*6  
对应 pool2 的输入为 12\*12  
对应 conv2 的输入为 14\*14  
对应 pool1 的输入为 28\*28  
对应 conv1 的输入为 30\*30  
所以 pool3 的感受野大小就是 30\*30

### Stride 的计算

每一个卷积层有一个 Strides 的概念，这个 Strides 就是之前所有层 Stride 的乘积。即

$$strids(i) = stride(1) * stride(2) * stride(3) * \dots * stride(i - 1)$$

### 专业的计算

参考文献：[卷积神经网络中的感受野计算 \(译\)-知乎](#)，原文 - 英语

定义：The receptive field is defined as the region in the input space that a particular CNN's feature is looking at (i.e. be affected by).

这篇英语文章，主要是四个公式：

- Calculate the number of output features

$$n_{out} = \lfloor \frac{n_{in} + 2p - k}{s} \rfloor + 1$$

其中， $p$  表示单侧 Padding 大小， $k$  表示 filter kernel 的大小。

- Calculate the Jump (Strides) in the output feature map

$$j_{out} = j_{in} * s$$

其中， $j$  表示 Strides，注意 stride 是不断积累的。如上下文中提到的。

- Calculate the receptive field size

$$r_{out} = r_{in} + (k - 1) * j_{in}$$

$k$  表示 kernel filter 的大小。

- Calculate the center position of the receptive field of the first output feature

$$start_{out} = start_{in} + \left( \frac{k - 1}{2} - p \right) * j_{in}$$

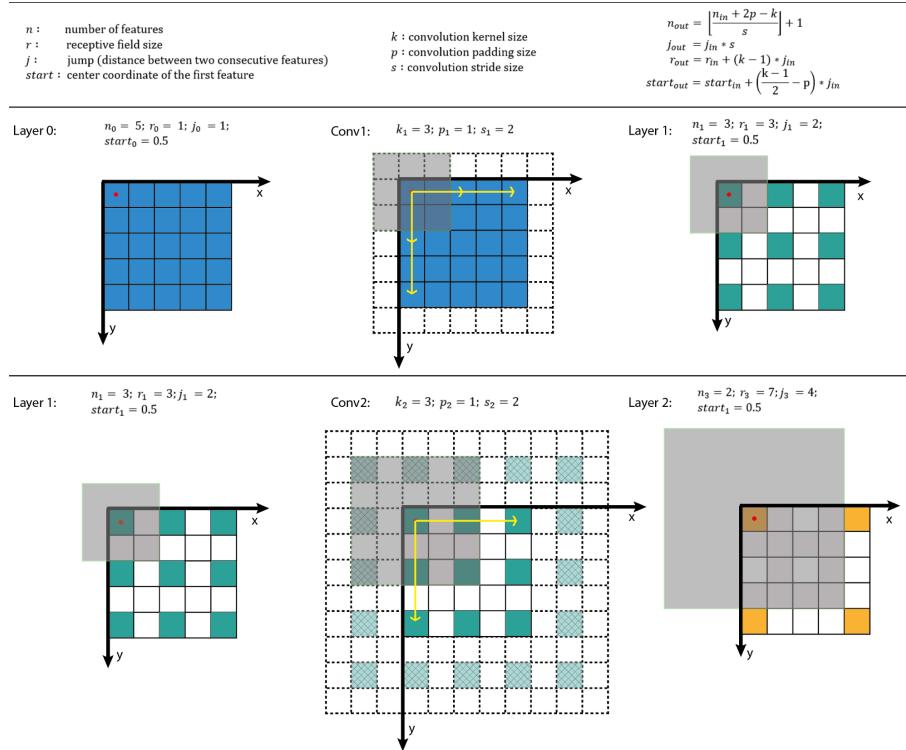


Fig 9.16. 文章的总结与说明

The first layer is the input layer, which always has  $n = \text{image size}$ ,  $r = 1$ ,  $j = 1$ , and  $\text{start} = 0.5$ .

小结如图9.16所示。

### 神经网络中的感受野 - 知乎

由于图像是二维的，具有空间信息，因此感受野的实质其实也是一个二维区域。但业界通常将感受野定义为一个正方形区域，因此也就使用边长来描述其大小了。在接下来的讨论中，本文也只考虑宽度一个方向。我们先按照图9.17所示对输入图像的像素进行编号。

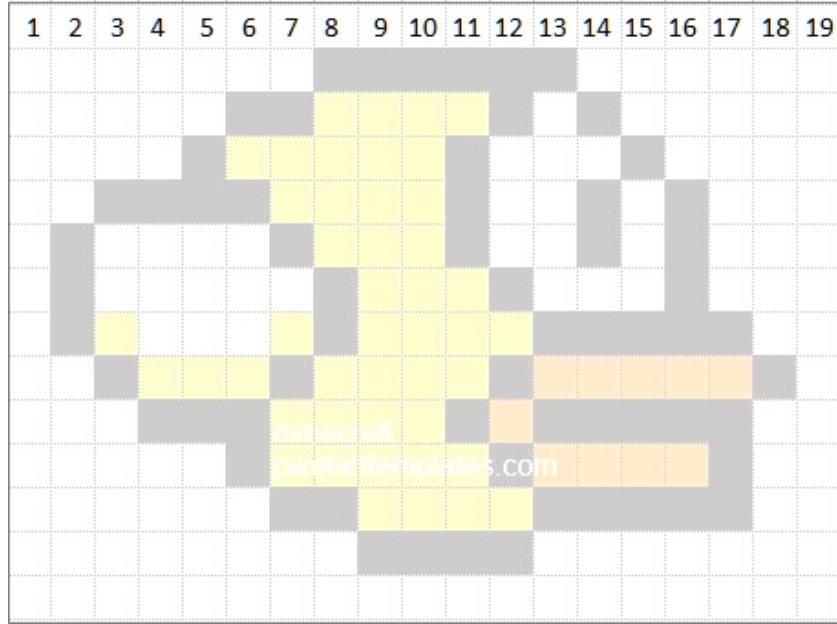
接下来我们使用一种并不常见的方式来展示 CNN 的层与层之间的关系（如下图，请将脑袋向左倒  $45^\circ$  观看  $>_<$ ），并且配上我们对原图像的编号。

图中黑色的数字所构成的层为原图像或者是卷积层，数字表示某单元能够看到的原始图像像素。我们用  $r_n$  来表示第  $n$  个卷积层中，每个单元的感受野（即数字序列的长度）；蓝色的部分表示卷积操作，用  $k_n$  和  $s_n$  分别表示第  $n$  个卷积层的 `kernel_size` 和 `stride`。

对 Raw Image 进行 `kernel_size=3, stride 2` 的卷积操作所得到的 `fmap1` (`fmap` 为 `feature map` 的简称，为每一个 `conv` 层所产生的输出) 的结果是显而易见的。序列 [1 2 3] 表示 `fmap1` 的第一个单元能看见原图像中的 1, 2, 3 这三个像素，而第二个单元则能看见 3, 4, 5。这两个单元随后又被 `kernel_size=2, stride 1` 的 Filter 2 进行卷积，因而得到的 `fmap2` 的第一个单元能够看见原图像中的 1, 2, 3, 4, 5 共 5 个像素（即取 [1 2 3] 和 [3 4 5] 的并集）。

接下来我们尝试一下如何用公式来表述上述过程。可以看到，[1 2 3] 和 [3 4 5] 之间因为 Filter 1 的 `stride 2` 而错开（偏移）了两位，而 3 是重叠的。对于卷积两个感受野为 3 的上层单元，下一层最大能获得的感受野为  $3 \times 2 = 6$ ，但因为有重叠，

## 9.15 CNN 中感受野的计算



**Fig 9.17.** 二维图像像素编号示意图

因此要减去  $(\text{kernel\_size} - 1)$  个重叠部分，Kernel 的 size 是本层的 Kernel 的尺寸，而重叠部分的计算方式则为上一层的感受野减去前面所说的本层的偏移量，这里是 2。因此我们就得到  $r_2 = r_1 \times k_2 - (r_1 - s_1) \times (k_2 - 1) = 3 \times 2 - (3 - 2) \times (2 - 1) = 5$ 。

继续往下一层看，我们会发现 [1 2 3 4 5] 和 [3 4 5 6 7] 的偏移量仍为 2，并不简单地等于上一层的  $s_2$ ，这是因为之前的 stride 对后续层的影响是永久性的，而且是累积相乘的关系（例如，在 fmap3 中，偏移量已经累积到 4 了），也就是说  $r_3$  应该这样求

$$r_3 = r_2 \times k_3 - (r_2 - s_1 \times s_2) \times (k_3 - 1) = 5 \times 3 - (5 - 2) \times (3 - 1) = 9$$

以此类推，

$$r_4 = r_3 \times k_4 - (r_3 - s_1 \times s_2 \times s_3) \times (k_4 - 1) = 9 \times 2 - (9 - 4) \times (2 - 1) = 13$$

于是我们就可以得到关于计算感受野的抽象公式了：

$$r_n = r_{n-1} \times k_n - (r_{n-1} - \prod_{i=1}^{n-1} s_i) \times (k_n - 1)$$

经过简单的代数变换之后，最终形式为：

$$r_n = r_{n-1} + (k_n - 1) \prod_{i=1}^{n-1} s_i$$

这个公式也体现了上文提到的 stride 影响的下一层。

## 9.16 神经网络中的初始化

### 回答一

参考文献: [为什么要进行初始化 -知乎](#)

参数初始化的目的是为了让神经网络在训练过程中学习到有用的信息, 这意味着参数梯度不应该为 0。所以参数初始化应该满足两个条件:

- 各个激活层不会出现饱和现象, 比如对于 Sigmoid 激活函数, 初始化值不能太大也不能太小, 导致进入饱和区
- 各个激活值不为 0, 如果激活层输出为 0, 也就是下一层的输入为 0, 所以这个卷积层对权重的偏导数为 0, 那么导致梯度也为 0
- 另一个非常重要的原因, 就是 Xavier, MSRA 等这些初始化不至于一开始就把网络发散, 或者梯度消失

所以, 最主要要注意两个问题, 首先是梯度不能消失, 然后网络不能发散。

### 9.16.1 Xavier

参考文章: [深度前馈网络与 Xavier 初始化原理 -知乎](#)

而为什么把模型的参数初始化成全 0 就不行了呢? 这个不用讲啦, 全 0 的时候每个神经元的输入和输出没有任何的差异, 换句话说, 根据前面 BP 算法的讲解, 这样会导致误差根本无法从后一层往前传 (乘以全 0 的  $\omega$  后误差就没了), 这样的 model 当然没有任何意义。

**那么我们不把参数初始化成全 0, 那我们该初始化成什么呢? 换句话说, 如何既保证输入输出的差异性, 又能让 model 稳定而快速的收敛呢?**

要描述“差异性”, 首先就能想到概率统计中的方差这个基本统计量。对于每个神经元的输入  $z$  这个随机变量, 根据前面讲 BP 时的公式, 它是由线性映射函数得到的, 也就是:

$$z = \sum_{i=1}^n \omega_i x_i$$

其中  $n$  是上一层神经元的数量。因此, 根据概率统计里的两个随机变量乘积的方差展开式

$$\text{Var}(\omega_i x_i) = E[\omega_i]^2 \text{Var}(x_i) + E[x_i]^2 \text{Var}(\omega_i) + \text{Var}(\omega_i) \text{Var}(x_i)$$

所以, 如果  $E(x_i) = E(\omega_i) = 0$ (可以通过批量归一化 Batch Normalization 来满足, 其它大部分情况也不会差太多), 那么就有:

$$\text{Var}(z) = \sum_{i=1}^n \text{Var}(x_i) \text{Var}(\omega_i)$$

如果变量  $x_i$  与  $\omega_i$  满足独立同分布的话:

$$\text{Var}(z) = n \cdot \text{Var}(x) \text{Var}(\omega)$$

## 9.17 计算图的后向传播计算

---

好了，这时重点来了。试想一下，根据文章《激活函数》，整个大型前馈神经网络无非就是一个超级大映射，将原始样本稳定的映射成它的类别。也就是将样本空间映射到类别空间。试想，如果样本空间与类别空间的分布差异很大，比如说类别空间特别稠密，样本空间特别稀疏辽阔，那么在类别空间得到的用于反向传播的误差丢给样本空间后简直变得微不足道，也就是会导致模型的训练非常缓慢。同样，如果类别空间特别稀疏，样本空间特别稠密，那么在类别空间算出来的误差丢给样本空间后简直是爆炸般的存在，即导致模型发散震荡，无法收敛。因此，我们要让样本空间与类别空间的分布差异（密度差别）不要太大的，也就是要让它们的方差尽可能相等。这里的样本空间与类别空间可以理解成是输入  $x$  空间与输出  $z$  的空间，以及其所对应的方差。

如果需要两个空间的方差差异不大，即满足  $\text{Var}(z) = \text{Var}(x)$ ，那么需要  $n \cdot \omega = 1$ ，也就是说  $\text{Var}(\omega) = 1/n$ 。其中  $n$  表示神经元输出端对应的个数。在前向传播时，对于特定一层神经网络， $n = n_{in}$ ，在后向传播时， $n = n_{out}$  而一般这两者都不相同，因此可以取它们的中间值：

$$\text{Var}(\omega) = \frac{1}{\frac{n_{in}+n_{out}}{2}}$$

假设  $\omega$  均匀分布时，由  $\omega$  在区间  $[a, b]$  内均匀分布的方差为：

$$\text{Var} = \frac{(b-a)^2}{12}$$

联立上面两个公式，可以得出  $\omega$  的分布区间（假设  $b = -a$ ）：

$$\omega \sim U \left[ -\frac{\sqrt{6}}{\sqrt{n_{in} + n_{out}}}, \frac{\sqrt{6}}{\sqrt{n_{in} + n_{out}}} \right]$$

（让  $w$  在这个区间里均匀采样就好啦）

得到的这个结论就是 Xavier 初始化方法。这就是为什么使用 Xavier 初始化这个 trick 经常可以让 model 的训练速度和分类性能取得大幅提高啦。所以在使用前馈网络时，除非你的网络设计的明显不满足 xavier 的假设，否则使用 xavier 往往不会出错。当然，另一方面来说，也很少有场合可以完全迎合 xavier 假设，因此时间充裕的话，改改分子，甚至去掉  $n_{out}$  都有可能带来意想不到的效果。

## 9.17 计算图的后向传播计算

参考文章：

- [1] 深度前馈网络与 Xavier 初始化原理 -知乎
- [2] BP 算法一夕小瑶

### 简单的说明

误差反向传播过程，其本质就是基于链式求导 + 梯度下降。

首先往上翻一翻，记住之前说过的前馈网络无非就是反复的线性与非线性映射。

首先，假设某个神经元的输入为  $z$ ，经过激活函数  $f_1(\cdot)$  得到输出  $a$ 。即函数值  $a = f_1(z)$ 。如果这里的输入  $z$  又是另一个函数  $f_2$  的输出的话（当然啦，这里的  $f_2$

就是线性映射函数，也就是连接某两层的权重矩阵），即  $z = f_2(x)$ ，那么如果基于  $a$  来对  $z$  中的变量  $x$  求导的时候，由于

$$\frac{\partial a}{\partial x} = \frac{\partial a}{\partial z} \cdot \frac{\partial z}{\partial x} = f'_1(z) \frac{\partial z}{\partial x}$$

上式也就是链式求导。更一般的链式求导法则为：

$$\frac{\partial a}{\partial x} = \sum_i \frac{\partial a}{\partial z_i} \cdot \frac{\partial z_i}{\partial x}$$

显然只要乘以激活函数  $f_1$  的导数，就不用操心激活函数的输出以及更后面的事儿了（这里的“后面”指的是神经网络的输出端方向），只需要将精力集中在前面的东西，即只需要关注  $z$  以及  $z$  之前的那些变量和函数就可以了。因此，误差反向传播到某非线性映射层的输出时，只需要乘上该非线性映射函数在  $z$  点的导数就算跨过这一层啦。

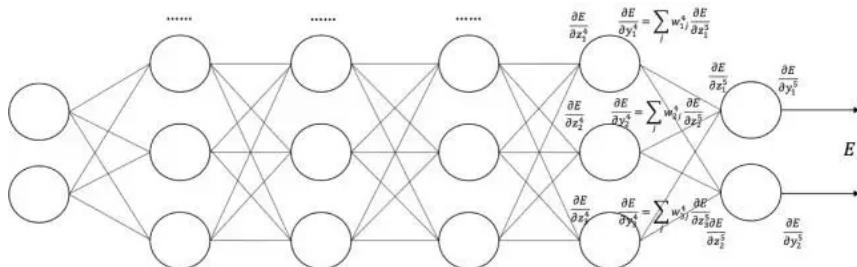
而由于  $f_2(\cdot)$  是个线性映射函数，即  $f_2(x) = \omega \cdot x + b$ ，因此

$$\frac{\partial z}{\partial x} = \omega$$

因此，当误差反向传播到线性映射层的输出时，若想跨过该层，只需要乘以线性映射函数的参数就可以啦 即乘上  $\omega$ 。

而这里的  $x$ ，又是更前面的非线性映射层的输出，因此误差在深度前馈网络中反向传播时，无非就是反复的跨过非线性层和线性层，也就是反复的乘以非线性函数的导数（即激活函数的导数）和线性函数的导数（即神经网络的参数/权重/连接边）。

也就是下面这张图啦（从右往左看）：



**Fig 9.18.** BP 计算过程示意图

### 9.17.1 Notes on CNNs

参考文献：CNN 反向求导理解 -知乎

#### CNN 的目标函数，包含 Weight Decay 的形式

假设共有  $m$  个样本  $(x_i, y_i)$ ，CNN 网络共有  $L$  层，中间包含若干卷积层和 Pooling 层，最后一层的输出为  $f(x_i)$ ，那么目标函数是（交叉熵的形式）：

$$Loss = -\frac{1}{m} \sum_{i=0}^{m-1} y'_i \log f(x_i) + \lambda \sum_{k=1}^L \| W_k \|^2$$

## 9.17 计算图的后向传播计算

其中， $W_k$  为每一层卷积层的权重。

### 求解输出层的误差敏感项

现在只考虑一个输入样本  $(x, y)$  的情形，loss 函数和上面的公式类似，使用交叉熵来表示的，暂时不考虑权值规则项，样本标签采用 One-Hot 编码，CNN 网络的最后一层采用 SoftMax 输出，样本  $(x, y)$  经过 CNN 网络后，输出用  $f(x)$  表示，则对应该样本的 loss 是：

$$l(f(x, y)) = - \sum_c 1_{(y=c)} \log f(x)_c = - \log f(x)_y$$

其中， $f(x)_c = p(y = c|x)$ 。因为  $x$  通过 CNN 后得到的输出  $f(x)$  是一个向量，该向量的元素值都是概率值，分别代表着  $x$  被分到各个类中的概率，而  $f(x)$  中下标  $c$  的意思就是输出向量中取对应  $c$  那个类的概率值。

采用上面的符号，可以求得此时 loss 值对输出层的误差敏感性表达式为：

$$\nabla_{\mathbf{a}^{(L+1)(x)}} - \log f(x)_y = -(\mathbf{e}(y) - f(x))$$

其中， $\mathbf{a}^{(L+1)(x)}$  是最后一层卷积的输出，有  $f(x) = \text{softmax}(\mathbf{a}^{(L+1)(x)})$ ， $e(y)$  为样本  $x$  标签值的 one-hot 表示，其中只有一个元素是 1，其它的都为 0。

推到如下：

$$\begin{aligned}
 & \frac{\partial}{\partial a^{(L+1)(x)}_c} - \log f(x)_y \\
 &= \frac{-1}{f(x)_y} \frac{\partial}{\partial a^{(L+1)(x)}_c} f(x)_y \\
 &= \frac{-1}{f(x)_y} \frac{\partial}{\partial a^{(L+1)(x)}_c} \text{softmax}(\mathbf{a}^{(L+1)(x)})_y \\
 &= \frac{-1}{f(x)_y} \frac{\partial}{\partial a^{(L+1)(x)}_c} \frac{\exp(a^{(L+1)(x)}_y)}{\sum_{c'} \exp(a^{(L+1)(x)}_{c'})} \\
 &= \frac{-1}{f(x)_y} \left( \frac{\frac{\partial}{\partial a^{(L+1)(x)}_c} \exp(a^{(L+1)(x)}_y)}{\sum_{c'} \exp(a^{(L+1)(x)}_{c'})} - \frac{\exp(a^{(L+1)(x)}_y) \left( \frac{\partial}{\partial a^{(L+1)(x)}_c} \sum_{c'} \exp(a^{(L+1)(x)}_{c'}) \right)}{\left( \sum_{c'} \exp(a^{(L+1)(x)}_{c'}) \right)^2} \right) \\
 &= \frac{-1}{f(x)_y} \left( \frac{1_{(y=c)} \exp(a^{(L+1)(x)}_y)}{\sum_{c'} \exp(a^{(L+1)(x)}_{c'})} - \frac{\exp(a^{(L+1)(x)}_y)}{\sum_{c'} \exp(a^{(L+1)(x)}_{c'})} \frac{\exp(a^{(L+1)(x)}_c)}{\sum_{c'} \exp(a^{(L+1)(x)}_{c'})} \right) \\
 &= \frac{-1}{f(x)_y} \left( 1_{(y=c)} \text{softmax}(\mathbf{a}^{(L+1)(x)})_y - \text{softmax}(\mathbf{a}^{(L+1)(x)})_y \text{softmax}(\mathbf{a}^{(L+1)(x)})_c \right) \\
 &= \frac{-1}{f(x)_y} (1_{(y=c)} f(x)_y - f(x)_y f(x)_c) \\
 &= -(1_{(y=c)} - f(x)_c)
 \end{aligned}$$

**Fig 9.19.** 目标函数对最后一层卷积层的输出，即最后一层 Softmax 的输入的求导

注意的是，是对任意一个  $f(x)_c$  求到的过程，但目标函数是根据正确的标签来确定的，也就是  $-\log f(x)_y$  中的  $y$  与分母中的  $c$  是不同的！前者是标签，后者是 Softmax 的第  $c$  个输入。

由上面公式可知，如果输出层采用 softmax，且 loss 用交叉熵形式，则最后一层的误差敏感值就等于 CNN 网络输出值  $f(x)$  减样本标签值  $e(y)$ ，即  $f(x) - e(y)$ ，其形式非常简单，这个公式是不是很眼熟？很多情况下如果 model 采用 MSE 的 loss，即

$loss = 1/2 * (e(y) - f(x))^2$ , 那么 loss 对最终的输出  $f(x)$  求导时其结果就是  $f(x)-e(y)$ , 虽然和上面的结果一样, 但是大家不要搞混淆了, 这 2 个含义是不同的, 一个是对输出层节点输入值的导数 (softmax 激发函数), 一个是对输出层节点输出值的导数 (任意激发函数)。而在使用 MSE 的 loss 表达式时, 输出层的误差敏感项为  $(f(x) - e(y)) \cdot f(x)'$ , 两者只相差一个因子。

这样就可以求出第 L 层的权值 W 的偏导数:

$$\frac{\partial Loss}{\partial W_L} = \frac{\partial Loss}{\partial (-\log f(x)_y)} \cdot \frac{f(x)}{\partial W} + \lambda W_L$$

对输出偏置层的偏导数是:

$$\frac{\partial Loss}{\partial b_L} = -\frac{1}{m} * (e(y) - f(x))$$

### 当卷积层的下一层是 Pooling 层时, 求解卷积层的误差敏感项

若第 1 层为卷积层, 第  $l+1$  层为 Pooling 层, 且 Pooling 层的误差敏感项是:

$$\delta_j^{l+1}$$

卷积层的误差敏感项为:

$$\delta_j^l$$

那么两者的关系是:

$$\delta_j^l = upsample(\delta_j^{l+1}) \cdot h(\delta_j^l)'$$

其中,  $\cdot$  是矩阵的点积, 即对应元素的乘积。卷积层与 Upsample 后的 Pooling 的输出层的节点是一一对应的, 所以可以都用下标 j 表示。后面的符号:  $h(\delta_j^l)'$  表示的是第 1 层第 j 个节点处激活函数的导数, 是对节点输入的导数? 也就是第 1 层卷积层的第 j 个节点。

那么问题来了, 重点就是这个 Upsample 怎么实现了! 而具体的形式是与采用什么 Pooling 方法有关的。但 unsample 的大概思想为: pooling 层的每个节点是由卷积层中多个节点(一般为一个矩形区域)共同计算得到, 所以 pooling 层每个节点的误差敏感值也是由卷积层中多个节点的误差敏感值共同产生的, 只需满足两层见各自的误差敏感值相等, 下面以 mean-pooling 和 max-pooling 为例来说明。

假设卷积层的矩形大小为  $4 \times 4$ , pooling 区域大小为  $2 \times 2$ , 很容易知道 pooling 后得到的矩形大小也为  $2 \times 2$  (本文默认 pooling 过程是没有重叠的, 卷积过程是每次移动一个像素, 即是有重叠的, 后续不再声明), 如果此时 pooling 后的矩形误差敏感值如下:

因为得满足反向传播时各层间误差敏感总和不变, 所以卷积层对应每个值需要平摊(除以 pooling 区域大小即可, 这里 pooling 层大小为  $2 \times 2 = 4$ ), 最后的卷积层值

分布为:

mean-pooling 时的 unsample 操作可以使用 matlab 中的函数 `kron()` 来实现, 因为是采用的矩阵 Kronecker 乘积。 $C=kron(A, B)$  表示的是矩阵 B 分别与矩阵 A 中每个元素相乘, 然后将相乘的结果放在 C 中对应的位置。

上面说的是 Avg Pooling 的计算, 下面说一下 MaxPooling 的计算:

需要记录前向传播过程中 pooling 区域中最大值的位置, 这里假设 pooling 层值 1,3,2,4 对应的 pooling 区域位置分别为右下、右上、左上、左下。则 Upsample 时, 在相应的位置放置误差敏感项, 其余的位置填 0。

1	3
2	4

(a) 输出的 2\*2Pooling 结果

1	1	3	3
1	1	3	3
2	2	4	4
2	2	4	4

(b) 输入的 4\*4 卷积层输出

**Fig 9.20.** 则按照 mean-pooling, 首先得到的卷积层应该是  $4 \times 4$  大小, 其值分布为(等值复制).

0.25	0.25	0.75	0.75
0.25	0.25	0.75	0.75
0.5	0.5	1	1
0.5	0.5	1	1

**Fig 9.21.** 反向传播时的误差

## 当 Pooling 的下一层为卷积层时，求该 Pooling 层的误差敏感项

假设第 l 层 (pooling 层) 有 N 个通道，即有 N 张特征图，第 l+1 层 (卷积层) 有 M 个特征，l 层中每个通道图都对应有自己的误差敏感值，其计算依据为第 l+1 层所有特征核的贡献之和。

下面是第 l+1 层中第 j 个核对第 l 层中第 i 个通道的误差敏感值计算方法：

$$\delta_i^l = \sum_{j=1}^M \delta_j^{l+1} * K_{ij}$$

其中，\* 是矩阵的卷积操作。

如果不明白，或者更多的内容，还是看原文吧。

### 9.17.2 Pooling 层的反向传播

前面一直关注了正向卷积的求导问题了，现在应该关注下 Pooling 层的求导问题！

## 9.18 神经网络中的 Attention 机制

参考文献：浅谈 Attention 机制 -知乎

### 9.18.1 Recurrent Models of Visual Attention

参考文献：[22]

Our model considers attention-based processing of a visual scene as a control problem and is general enough to be applied to static images, videos, or as a perceptual module of an agent that interacts with a dynamic visual environment (e.g. robots, computer game playing agents).

Instead of processing an entire image or even bounding box at once, at each step, the model selects the next location to attend to based on past information and the demands of the task.

We describe an end-to-end optimization procedure that allows the model to be trained directly with respect to a given task and to maximize a performance measure which may depend on the entire sequence of decisions made by the model. This procedure uses back-propagation to train the neural-network components and policy gradient to address the non-differentiabilities due to the control problem

目标检测的几种方法：

- 基于窗口的分类器，包括 Proposal 等，计算量大
- 基于 Saliency Detection 的，不能 Integrate information across fixations, 仅利用底层图像特征，忽略了 Semantic Content of a scene and task demands.
- 一些工作把视觉问题当做 Sequential decision task，本文也是
-

## 9.18 神经网络中的 Attention 机制

Our formulation which employs an RNN to integrate visual information over time and to decide how to act is, however, more general, and our learning procedure allows for end-to-end optimization of the sequential decision process instead of relying on greedy action selection.

我们的模型，既可以实现性质图像的 Object Recognition，而且还适用于动态环境，以一种 Task-driven 的方式。

### The Recurrent Attention Model - RAM

In this paper we consider the attention problem as the sequential decision process of a goal-directed agent interacting with a visual environment.

这篇文章里面，感觉这意思，Attention 机制不就是 Reinforcement Learning 么？

该公式包含了各种任务，如静态图像中的对象检测，控制问题，即根据屏幕上的图像流来学习打游戏等。

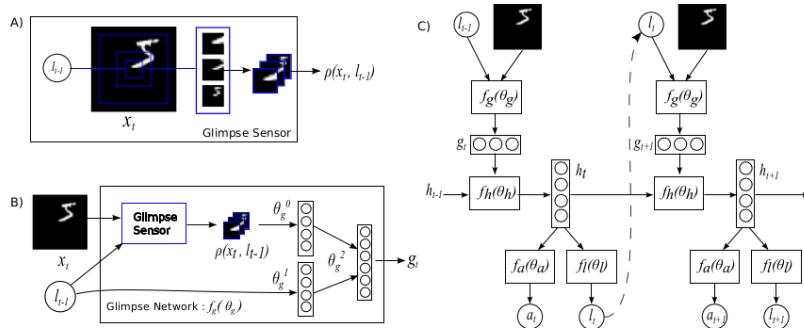


Figure 1: **A) Glimpse Sensor:** Given the coordinates of the glimpse and an input image, the sensor extracts a *retina-like* representation  $\rho(x_t, l_{t-1})$  centered at  $l_{t-1}$  that contains multiple resolution patches. **B) Glimpse Network:** Given the location ( $l_{t-1}$ ) and input image ( $x_t$ ), uses the glimpse sensor to extract retina representation  $\rho(x_t, l_{t-1})$ . The retina representation and glimpse location is then mapped into a hidden space using independent linear layers parameterized by  $\theta_g^0$  and  $\theta_g^1$  respectively using rectified units followed by another linear layer  $\theta_g^2$  to combine the information from both components. The glimpse network  $f_g(\cdot; \{\theta_g^0, \theta_g^1, \theta_g^2\})$  defines a trainable bandwidth limited sensor for the attention network producing the glimpse representation  $g_t$ . **C) Model Architecture:** Overall, the model is an RNN. The core network of the model  $f_h(\cdot; \theta_h)$  takes the glimpse representation  $g_t$  as input and combining with the internal representation at previous time step  $h_{t-1}$ , produces the new internal state of the model  $h_t$ . The location network  $f_l(\cdot; \theta_l)$  and the action network  $f_a(\cdot; \theta_a)$  use the internal state  $h_t$  of the model to produce the next location to attend to  $l_t$  and the action/classification  $a_t$  respectively. This basic RNN iteration is repeated for a variable number of steps.

**Fig 9.22.** Attention 模型示意图

图9.22中， $x_t$  表示第  $t$  时刻的输入图像， $l_{t-1}$  表示位置， $\rho(x_t, l_{t-1})$  表示这个位置的 Retina-like representation.  $g_t$  为 glimpse representation。

We will refer to this low-resolution representation as a *glimpse*.

Action 分为两个部分：

- Location actions

are chosen stochastically from a distribution parameterized by the location network  $f_l(h_t; \theta_l)$  at time  $t$ 。

- Environment action

The environment action  $a_t$  is similarly drawn from a distribution conditioned on a second network output  $a_t \sim p(\cdot | f_a(h_t; \theta_a))$

与 RL 里面的 Partially Observable Decision Process (POMDP) 的一个特例。

## 总结

感觉本文中的 Attention 与 RL 关系密切，而且与 RNN 网络结合起来了。那么到底什么是 Attention 机制，还需要阅读其它文章，现在我还不太明白。

## 9.19 小型网络

参考文章：[CVPR2018 高效小网络 -知乎](#)

### 9.19.1 理论分析

先说结论：卷积层主要引入了计算量，全连接层主要引入了参数数量。  
参数数量用  $\text{params}$  表示，关系到模型大小，单位通常为 M，通常参数用 float32 表示，所以模型大小是参数数量的 4 倍。

理论计算量用 FLOPs 或者 M-Adds 表示，这里用 FLOPs 写起来简单，关系到算法速度，大模型的单位通常为 G，小模型通道为 M。

需要注意的两点：

- 理论计算量通常只考虑乘加操作 (Multi-Adds) 的数量，而且只考虑 CONV 和 FC 等参数层的计算量，忽略 BatchNorm 和 PReLU 等等。一般情况，CONV 和 FC 层也会忽略仅纯加操作的计算量，如 bias 偏置加和 shortcut 残差加等，**目前技术有 BN 的 CNN 可以不加 bias**。
- 理论计算量通常和实际 ARM 实测速度会有不一致，主要是理论计算量太过理论化，没有考虑不同硬件 IO 速度和计算力差异，最重要的是 inference framework 部署框架优化水平和程度差异，不同框架的优化的关注点不一样，同一框架对不同层的优化程度也不一样。Inference Framework 以我用过的 ncnn 为代表。

假设，卷积核大小为  $k_h * k_w$ ，输入的特征的通道为  $C_{in}$ ，输出的 Feature Map 的数量为  $C_{out}$ ，Feature Map 的尺寸是  $H * W$ 。那么，**卷积层**对应的参数量  $\text{params}$  和计算量  $FLOPs$  分别是：

$$\begin{aligned} \#Params &: (k_h * k_w * c_{in} + 1) * c_{out} \\ \#FLOPs &: k_w * k_h * c_{in} * c_{out} * W * H \end{aligned}$$

相对比的，**全连接层**对应的上述两个参数分别是：

$$\begin{aligned} \#Params &: (c_{in} + 1) * C_{out} \\ \#FLOPs &: c_{in} * C_{out} \end{aligned}$$

### 9.19.2 GCONV & DWCONV

有时间在整理吧，细节见原文。

结论就是，通过输入通道的分组，可以降低参数量以及计算量  $g$  倍，其中  $g$  是输入通道的分组数，并且在 DWCONV 中这个分组数  $g$  就是输入的通道数  $C_{in}$ 。

其它的一点说明：

## 9.20 Deep Reinforcement Learning

---

设计 CNN 目前都采用堆 block 的方式，后面对每个模型只集中分析其 block 的设计。堆 Block 简化了网络设计问题：只要设计好 block 结构，对于典型  $224 \times 224$  输入，设计 CNN 只需要考虑  $112 \times 112$ 、 $56 \times 56$ 、 $28 \times 28$ 、 $14 \times 14$ 、 $7 \times 7$  这 5 个分辨率阶段 (stage) 的 block 数量和通道数量就可以了。

CNN 加深度一般都选择  $14 \times 14$  这个分辨率阶段，多堆几个 block，这一层的参数数量和计算量占比最大，所以我们选这一层作为特例分析每种 block 的参数数量和计算量，量化说明选择  $14 \times 14 \times 512$  的输入和输出情况。

### 9.19.3 NiN 及相关

## 9.20 Deep Reinforcement Learning

参考文献：[A Beginner's Guide to Deep Reinforcement Learning](#)  
几个重要的摘抄。

- Algorithms can start from a blank slate, and under the right conditions they achieve superhuman performance. Like a child incentivized by spankings and candy, these algorithms are penalized when they make the wrong decisions and rewarded when they make the right ones – this is reinforcement.
- Reinforcement learning solves the difficult problem of correlating immediate actions with the delayed returns they produce.

## 9.21 为什么 Python 中要继承 object 类

参考文章：[stackoverflow answer](#)

在 Python3 中，除了因为要兼容于 Python2，没有其它原因。而在 Python2 中，则有些复杂。

### Python 2.x story

Python2(从 python2.2 开始) 中，有两种类型的类 (two styles of classes)，他们根据是否继承自 **object** 来区分。

- Classic style classes, they don't have object as a base class
- New style classes: they have, directly or indirectly (e.g. inherit from a built-in type), **object** as a base class

那么为什么选择用新式类呢 (New style classes)? 几个原因如下：

- Support for descriptors. Specifically, the following constructs are made possible with descriptors:
  - classmethod
  - staticmethod
  - properties with property

- slots

- the `__new__` static method
- method resolution order (MRO)
- Related to MRO

### Python 3.x story

In Python 3, things are simplified. Only new-style classes exist (referred to plainly as classes) so, the only difference in adding object is requiring you to type in 8 more characters.

### Choose which one

- In Python 2.x, always inherit from `object` explicitly
- In Python 3.x, inherit `object` if you are writing code that tries to be Python agnostic (不可知论者), that is, it needs to work both in python 2 and in python 3. Otherwise, don't, it really makes no difference since Python inserts it for you behind the scenes.

## 9.22 1\*1 卷积

参考文章: [1\\*1 卷积的作用与好处 - 知乎](#)

1\*1 卷积和正常的卷积一样, 唯一不同的是它的大小是 1\*1, 没有考虑在前一层局部信息之间的关系, 所以这里跟普通说卷积的时候一样, 都是自动忽略了在 Channel 上的维度, 而只是空间域的维度是 1\*1 或 3\*3 或 5\*5 等。最早出现在 Network In Network 的论文中, 使用 1\*1 卷积是想加深加宽网络结构, 在 Inception 网络 (Going Deeper with Convolutions) 中用来降维,

由于 3\*3 卷积或者 5\*5 卷积在几百个 filter 的卷积层上做卷积操作时相当耗时, 所以 1\*1 卷积在 3\*3 卷积或者 5\*5 卷积计算之前先降低维度。(说的是 Channel 的维度。)

所以 1\*1 卷积的主要作用有以下几点:

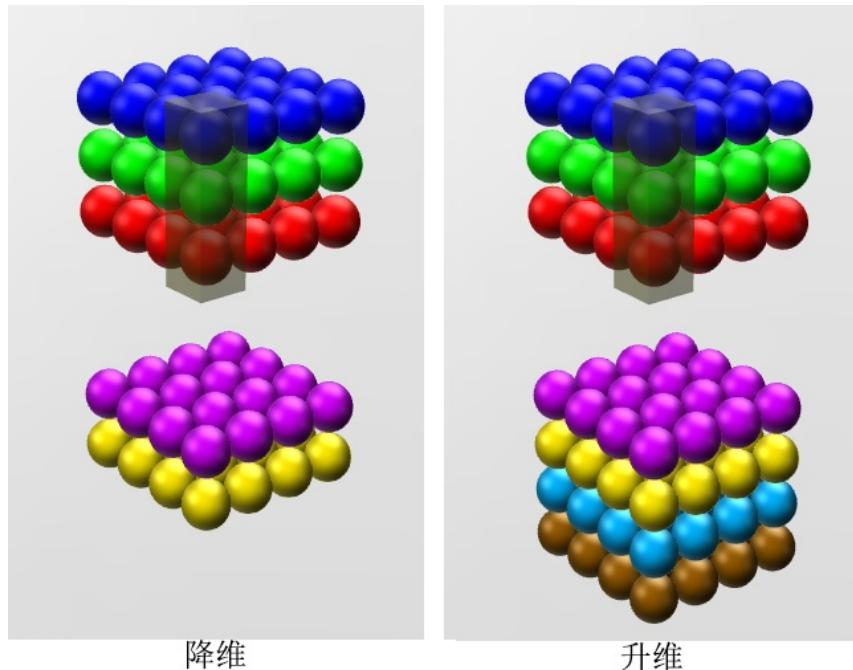
- 降维

比如, 一张 500 \* 500 且厚度 depth 为 100 的图片在 20 个 filter 上做 1\*1 的卷积, 那么结果的大小为 500\*500\*20。

- 加入非线性

卷积层之后经过激励层, 1\*1 的卷积在前一层的学习表示上添加了非线性激励 (non-linear activation), 提升网络的表达能力。类似于在一个像素位置, 在所有 Channel 上进行非线性变换, 而不考虑附近的信息。

所以 1\*1 卷积实际上是对每个像素点, 在不同的 Channel 上进行线性组合, 即信息整合, 且保留图像的原有平面结构, 调控 Depth, 自动完成升维、降维的作用。如下图所示, 如果选择 2 个 filter 的 1\*1, 那么数据就从原来的 depth 为 3 变为 depth 为 2. 若用 4 个 filter, 则起到了升维的作用。



**Fig 9.23. 1\*1 Convolutoin 的作用示意图**

## 9.23 目标检测中完整流程

由于几个经典的目标检测模型的实现不是很复杂，所以这里主要记录数据的处理过程。首先输入是什么数据、用到了哪些增广技术、输入图像的尺寸、输出是什么形式、目标函数怎么实现怎么优化、怎么评估结果等等。

在目标检测中，主要用到的评估指标是：mAP、IOU。但是怎么实现呢？

目标函数是什么，如何实现 BBox 的确定等等，好麻烦啊。

## 9.24 Batch Size 的影响

打的批量通常能够让训练收敛更快、性能更强，但会增加内存的开销。

为什么批量更大，能提升性能的两个重要参数：

- 较大的批量，'可能'会提高优化步骤的有效性，从而导致模型参数更快的收敛
- 较大的批量也可以减少把训练数据移动到 GPU 导致的通信开销来提高性能

## 9.25 非极大值抑制 NMS

Non-Maximum Suppression, NMS

对于相似的预测边界框，非最大抑制（Non-Maximum Suppression, NMS）只保留置信度最高的那个来去除冗余。它的工作机制如下：对于每个类别，我们首先拿到每个预测边界框被判断包含这个类别物体的概率。然后我们找到概率最大的那个边界框并保留它到输出，接下来移除掉（抑制）其它所有的跟这个边界框的

IoU 大于某个阈值的边界框。在剩下的属于该类别的边界框里我们再找出预测概率最大的边界框，重复前面的移除过程。到这里我们还只是对其中的一个类别做的事情；完成之后，还需要对其它类别进行同样的事情，也就是去除那些与当前这个类别的重叠较大的边界框。直到我们要么保留或者移除了每个边界框。

## 9.26 Bilinear filler 初始化 ConvTranspose 层的权重

参考文献： [Bilinear Filler 初始化 -知乎](#)

当 ConvTranspose 层的权重的学习率被初始化为 0 时，该层的权重在训练过程中保持不变，一直作为 Bilinear Resize 的作用。

MXNet 中，Bilinear Filter Initializer 实现代码：

Listing 9.2: Bilinear Filter Initializer 实现代码 -MXNet

```
class Bilinear(Initializer):

    """Initialize weight for upsampling layers."""
    def __init__(self):
        super(Bilinear, self).__init__()
    def _init_weight(self, _, arr):
        weight = np.zeros(np.prod(arr.shape), dtype='float32')
        shape = arr.shape
        f = np.ceil(shape[3] / 2.)
        c = (2 * f - 1 - f % 2) / (2. * f)
        for i in range(np.prod(shape)):
            x = i % shape[3]
            y = (i / shape[3]) % shape[2]
            weight[i] = (1 - abs(x / f - c)) * (1 - abs(y / f - c))
        arr[:] = weight.reshape(shape)
```

上述的完成以下公式：

$$weight[i] = (1 - abs(x/f - c)) * (1 - abs(y/f - c))$$

虽然具体的公式来源没找到，但可以借鉴下面的 Bilinear Interpolation 公式：

$$\begin{aligned} f(x, y) &\approx \frac{y_2 - y}{y_2 - y_1} f(x, y_1) + \frac{y - y_1}{y_2 - y_1} f(x, y_2) \\ &= \frac{y_2 - y}{y_2 - y_1} \left( \frac{x_2 - x}{x_2 - x_1} f(Q_{11}) + \frac{x - x_1}{x_2 - x_1} f(Q_{21}) \right) + \frac{y - y_1}{y_2 - y_1} \left( \frac{x_2 - x}{x_2 - x_1} f(Q_{12}) + \frac{x - x_1}{x_2 - x_1} f(Q_{22}) \right) \\ &= \frac{1}{(x_2 - x_1)(y_2 - y_1)} (f(Q_{11})(x_2 - x)(y_2 - y) + f(Q_{21})(x - x_1)(y_2 - y) \\ &\quad + f(Q_{12})(x_2 - x)(y - y_1) + f(Q_{22})(x - x_1)(y - y_1)) \end{aligned}$$

参考的知乎答案中，给出了一个插值的例子，可以参考下。

## 9.27 卷积层输出的尺寸

假设输入尺寸是:  $(B, C, W, H)$

且卷积核的大小是  $(K, K)$ , Stride 大小为  $(S, S)$ , Padding 的大小是  $Padding$ , 那么输出的大小是:

$$W_{out} = \frac{W - K + 2 * Padding}{S} + 1$$

需要注意的是, 如果公式中的除法除不尽, 那么需要舍去小数部分, 也就是, 公式中的分式是整除!

## 9.28 机器学习面试博客总结 2018.06.11

### 9.28.1 大疆

参考文献: [大疆 2018 机器学习笔记](#)

#### 机器学习中的范数问题

L0 范数指向量中非 0 元素的个数, L1 范数是指各个元素的绝对值之和, 也叫稀疏规则算法; L2 范数就是先平方在求和开方。L2 范数有助于处理 Condition number 不好的情况下矩阵求逆困难的问题, 有助于有噪声的情况。

更多的内容看花书部分的总结。

#### 类别中不平衡

1. 过抽样抽样处理不平衡数据的最常用方法, 基本思想就是通过改变训练数据的分布来消除或减小数据的不平衡。

过抽样方法通过增加少数类样本来提高少数类的分类性能, 最简单的办法是简单复制少数类样本, 缺点是可能导致过拟合, 没有给少数类增加任何新的信息。改进的过抽样方法通过在少数类中加入随机高斯噪声或产生新的合成样本等方法。

2. 引入代价敏感因子, 设计出代价敏感的分类算法。通常对小样本赋予较高的代价, 大样本赋予较小的代价, 期望以此来平衡样本之间的数目差异。如 Focal Loss

3. 特征选择

样本数量分布很不平衡时, 特征的分布同样会不平衡。尤其在文本分类问题中, 在大类中经常出现的特征, 也许在稀有类中根本不出现。因此, 根据不平衡分类问题的特点, 选取最具有区分能力的特征, 有利于提高稀有类的识别率。

4. 直接基于原始训练集进行学习, 但是在用训练好的分类器进行预测的时候, 将上面的第一种方法嵌入到决策过程中, 这种方法称为“阈值移动 (threshold-moving)”。

5. Hard Negative Mining,

也就是仅利用负类(很多)的少量样本进行训练, 也就是改变数据的分布的方式。

#### Confusion Matrix

如图9.24所示。

		True condition			
		Condition positive	Condition negative	Prevalence = $\frac{\sum \text{Condition positive}}{\sum \text{Total population}}$	Accuracy (ACC) = $\frac{\sum \text{True positive} + \sum \text{True negative}}{\sum \text{Total population}}$
Predicted condition	Predicted condition positive	True positive, Power	False positive, Type I error	Positive predictive value (PPV), Precision = $\frac{\sum \text{True positive}}{\sum \text{Predicted condition positive}}$	False discovery rate (FDR) = $\frac{\sum \text{False positive}}{\sum \text{Predicted condition positive}}$
	Predicted condition negative	False negative, Type II error	True negative	False omission rate (FOR) = $\frac{\sum \text{False negative}}{\sum \text{Predicted condition negative}}$	Negative predictive value (NPV) = $\frac{\sum \text{True negative}}{\sum \text{Predicted condition negative}}$
	True positive rate (TPR), Recall, Sensitivity, probability of detection = $\frac{\sum \text{True positive}}{\sum \text{Condition positive}}$	False positive rate (FPR), Fall-out, probability of false alarm = $\frac{\sum \text{False positive}}{\sum \text{Condition negative}}$	Positive likelihood ratio (LR+) = $\frac{\text{TPR}}{\text{FPR}}$	Diagnostic odds ratio (DOR) = $\frac{\text{LR}^+}{\text{LR}^-}$	F <sub>1</sub> score = $\frac{2}{\frac{1}{\text{Recall}} + \frac{1}{\text{Precision}}}$
	False negative rate (FNR), Miss rate = $\frac{\sum \text{False negative}}{\sum \text{Condition positive}}$	True negative rate (TNR), Specificity (SPC) = $\frac{\sum \text{True negative}}{\sum \text{Condition negative}}$	Negative likelihood ratio (LR-) = $\frac{\text{FNR}}{\text{TNR}}$		

Fig 9.24. Confusion Matrix 示意图

## ROC & AUC

ROC 曲线，恒周假阳性概率 (FP)，真阳性 (TP) 为纵轴组成的坐标图。  
真阳性又称为灵敏度。

ROC 曲线越靠近左上角，实验的准确性就越高。

AUC 为 ROC 曲线下的面积，哪一种实验的 AUC 最大，则哪一种实验的诊断判断性最佳。

AUC 越接近于 1，说明效果越好。

## 9.28.2 机器学习面试总结

参考文献：机器学习面试总结 - 博客园

## 9.29 花书前两部分总结

### 9.29.1 第三章

#### 概率论的知识

Sigmoid 函数：

$$\sigma(x) = \frac{1}{1 + \exp(-x)}$$

其导数：

$$\frac{d}{dx} \delta(x) = \sigma(x) (1 - \sigma(x))$$

并且有：

$$1 - \sigma(x) = \sigma(-x)$$

#### 信息论

一个事件  $x$  的自信息为：

$$I(x) = -\log(Pr(x))$$

自信息只处理单个输出。我们可以用香农熵来对整个概率分布中的不确定性总量进行量化：

$$H(x) = \mathbb{E}_{x \sim P} [\log Pr(x)]$$

对于同一个随机变量  $x$ , 有两个单独的概率分布  $P(x)$  和  $Q(x)$ , 可以使用 **KL 散度**来衡量两个分布的差异!

$$\begin{aligned} D_{KL}(P \parallel Q) &= \mathbb{E}_{x \sim P} \left[ \log \frac{P(x)}{Q(x)} \right] \\ &= \mathbb{E}_{x \sim P} [\log P(x) - \log Q(x)] \end{aligned}$$

从上面的公式可以看出, 假设输入的  $x$  服从  $P$  的分布, 计算此时的  $P$  与  $Q$  之间的差异。

**KL 散度**是非负的, 并且衡量的是两个分部之间的差异, 它经常被用作分布之间的某种距离。然而他并不是真正的距离, 因为它不是对称的。通过选择  $D_{KL}(P \parallel Q)$  和  $D_{KL}(Q \parallel P)$  对结果影响较大。

一种与 **KL 散度**联系密切的量是交叉熵。即  $H(P, Q) = D_{KL}(P \parallel Q) + H(P)$ , 可以得到下面的公式:

$$H(P, Q) = -\mathbb{E}_{x \sim P} [\log Q(x)]$$

针对分布  $Q$  最小化交叉熵等价于最小化 **KL 散度**, 因为分布  $Q$  并不参与被省略的那一项。一种特殊的情况:  $\lim_{x \leftarrow 0} x \log x = 0$ 。

个人补充:

后面我们可以看到, 这个交叉熵的优化等价于最大似然的优化! 如果但看上面交叉熵的计算公式的话, 可以这么认为: 分布  $P$  是数据的真实分布,  $Q$  是模型的分布, 我们优化的目的就是让这两种分布之间的差距最小; 然而一般情况下我们并不知道真是的数据分布, 一种直观的方法就是用训练数据的经验分布代替真实的数据分布。

### 结构化概率模型

这部分的有向、无向图模型更详细的内容可以参考<<计算机视觉 - 模型、学习和推理>>或者<<统计学习方法>>。

## 9.29.2 第四章

数值计算

### 上溢和下溢

必须对上溢和下溢进行数值稳定的一个例子是 **Softmax 函数**, 该函数后面可以看到, 经常用于预测和 Multinoulli 分部相关联的概率, 定义如下:

$$\text{softmax}(x)_i = \frac{\exp(x_i)}{\sum_{j=1}^n \exp(x_j)}$$

若所有的  $x_j = c$ , 那么 softmax 结果为  $\frac{1}{n}$ 。从数值上来说, 当  $c$  量级很大时, 这可能不会发生什么。如果  $c$  很小的负数, 那么  $\exp(c)$  就会下溢, 这就意味着 softmax

的分母变为 0，所以结果是未定义的；另一方面，如果  $c$  很大的正数，那么  $\exp(x)$  本身就会上溢，同样导致未定义的结果。

解决办法：

构建新的变量： $\mathbf{z} = \mathbf{x} - \max_i x_i$ ，然后用  $\mathbf{z}$  代替  $\mathbf{x}$ ，即：

$$\text{softmax}(\mathbf{z})$$

从公式中可以看出，softmax 解析上的函数值不会因为从输入向量减去或加上标量而改变。减去  $\max_i x_i$  导致  $\exp$  的最大参数为 0，这排除了  $\exp$  函数的上溢的可能性；同样的，分母中至少有一个值为 1 的项，这就排除了因分母下溢而导致的零除的可能性。

## 病态条件

首先引入条件数的概念，条件数指的是函数相对于输入的微小变化而变化的快慢程度。也就是说，输入微小扰动，输出也可能迅速变化。

考虑函数  $f(\mathbf{x}) = \mathbf{A}^{-1}\mathbf{x}$ ，当  $A \in R^{n*n}$  具有特征分解时，其条件数为：

$$\max_{i,j} \left| \frac{\lambda_i}{\lambda_j} \right|$$

这里与通常的条件数定义有所不同，即取最大和最小特征值的模之比。当该数很大时，矩阵求逆对输入的误差特别敏感。后面可以看到，L2 正则项对病态有一些帮助。

## 基于梯度的优化方法

函数在  $\mathbf{u}$  方向的方向导数，是函数  $f$  在  $\mathbf{u}$  方向的斜率。换句话说，方向导数是函数  $f(\mathbf{x} + \alpha\mathbf{u})$  关于  $\alpha$  的导数。使用链式法则，我们可以看到当  $\alpha = 0$  时，有：

$$\frac{\delta}{\delta\alpha} f(\mathbf{x} + \alpha\mathbf{u}) = \mathbf{u}^T \nabla_x f(\mathbf{x})$$

为了最小化  $f$ ，我们希望找到使  $f$  下降的最快的方向。通过上面公式可以看出，当  $\mathbf{u}$  和  $\nabla_x f(\mathbf{x})$  的方向相反时，它们的夹角为 0， $\cos$  值为 1，得到最大。所以这又被称为最速下降法或梯度下降。

梯度下降决定了下降最快的方向，我们还需要设定在该方向的步长，也就是学习率了，一种选择学习率的方法是线搜索。

## 梯度之上

Jacobian 和 Hessian 矩阵。

Hessian 是实对称矩阵，可以分解成  $H = Q\Lambda Q^T$ 。在特定方向  $\mathbf{d}$  上的二阶导数可以写成  $d^T H d$ 。当  $\mathbf{d}$  是  $H$  的一个特征向量时，这个方向的二阶导数就是对应的特征值。对于其他方向的  $d$ ，方向的二阶导数是所有特征值的加权平均。最大特征值确定最大二阶导数，最小特征值确定最小二阶导数。

将函数  $f(x)$  进行二级泰勒级数近似，则可以计算求得在进行  $\epsilon g$  步长更新后，其函数值为：

$$f(x^{(0)} - \epsilon g) = f(x^{(0)}) - \epsilon g^T g + \frac{1}{2} g^T H g$$

其中有三项，函数的原始值、函数斜率导致的预期改善、函数曲率导致的矫正。为了使  $f(x^{(0)} - \epsilon g)$ ，那么上式的右边应该使  $-\epsilon g^T g + \frac{1}{2}g^T H g$  最小，为了确定最优的学习率，那么把后面这一项看做是  $\epsilon$  的二阶方程，求解其最小值，应该使得一阶导数为 0。对后一项进行对  $\epsilon$  求导，则得到最优的学习率为：

$$\epsilon^* = \frac{g^T g}{g^T H g}$$

在最坏的情况下， $g$  与  $H$  的最大特征值  $\lambda_{max}$  对应的特征向量对齐，此时最优步长为  $\frac{1}{\lambda_{max}}$ 。Hessian 的特征值决定了学习率的量级。

Hessian 的另一个作用就是：二阶导数测试。在临界点（一阶导数为 0）时，若 Hessian 是正定的，则该临界点是**全局最小点**，因为方向二阶导数在任意方向都是正的  $d^T H d > 0$ ，所以该临界点的导数导数都是正的，也就是该点是全局最小点了。当 Hessian 矩阵是负定的时候，该点是**局部极大点**。如果 Hessian 的特征值有大于 0 的值，也有小于 0 的值，则为鞍点。

当 Hessian 的条件数很差时，梯度下降法也会表现的很差。这是因为一个方向上的导数增加的很快，而在另一个方向上增加的很慢。梯度下降不知道导数的这种变化，所以它不知道应该优先搜索导数长期为负的方向，而不是曲率最大的方向。病态条件也导致很难选择合适的补偿。

我们可以使用 Hessian 矩阵的信息来指导搜索。比如牛顿法。

### 约束优化

包含著名的 KKT 条件 + 广义拉格朗日了。

### 9.29.3 第五章

#### 机器学习基础

#### 设计矩阵

设计矩阵的每一行包含一个不同的样本，每一列对应不同的特征。

#### 容量、过拟合和欠拟合

**模型的容量：**通俗来讲，模型的容量是指其拟合各种函数的能力。容量高的模型可能会过拟合，因为记住了不适用于测试集的训练集性质，所以从这个角度来看，增加正则项来防止过拟合本质上是限制模型可以拟合的函数的空间的大小。

**VC 维度：**定义为该分类器能够分类的训练样本的最大数目。

**正则化：**是指修改学习算法、使其降低泛化误差而非训练误差。

正则化是机器学习领域的中心问题之一，只有优化能够与其重要性相提并论。这一点从花书的第七章、第八章可以看出来，第七章全面的讲了正则化、第八章全面的讲了优化算法。

## 估计、偏差和方差

偏差和方差度量着估计量的两个不同误差来源。偏差度量着偏离真实函数或参数的误差期望，而方差度量着数据上任意特定采样可能导致的估计期望的偏差。

均方误差 (MSE):

$$\begin{aligned} MSE &= \mathbb{E} [(\hat{\theta}_m - \theta)^2] \\ &= Bias(\hat{\theta}_m)^2 + Var(\hat{\theta}_m) \end{aligned}$$

MSE 度量着估计与真实参数  $\theta$  之间的平方误差的总体期望偏差。如上式所示，MSE 包含了偏差和方差。理想的估计具有较小的 MSE 或是在检查中会稍微约束他们的偏差和方差。

偏差和方差的关系与机器学习容量、欠拟合和过拟合的概念紧密相连。用 MSE 度量泛化误差(偏差和方差对于泛化误差都是有意义的)时，增加容量会增加方差，降低偏差。

补充一点：

### 偏差和方差的区别

偏差：描述的是预测值的期望与真实值之间的差距。

方差：描述的是预测值的变化范围，离散程度，也就是距离其期望值的距离。

当偏差较大时，一般发生欠拟合；当方差较大时，泛化能力较差，一般发生过拟合。随着模型容量的增加，偏差下降，方差呈 U 型。

训练程度与偏差、方差的关系：

1. 训练不足时，学习器的拟合能力不够强，训练数据的扰动不足以使学习器发生显著变化，偏差将主导泛化错误率。
2. 当训练程度加深，学习器的能力逐渐增强，训练数据产生的扰动逐渐能够被学习器学到，方差将主导泛化错误率。
3. 训练程度充足后，学习器的拟合能力已经非常强，训练数据产生的轻微扰动都会导致学习器发生显著变化。将发生过拟合。

## 最大似然估计

这一部分主要推导下前面提到的，最大似然估计与交叉熵之间的关系。

最大似然估计的目标是：

$$\begin{aligned} \theta_{ML} &= \arg \max_{\theta} p_{model}(\mathcal{X}; \theta) \\ &= \arg \max_{\theta} \prod_{i=1}^m p_{model}(x^{(i)}; \theta) \end{aligned}$$

其中， $m$  是样本数量。

推导如下，将上面的最大似然转换成对数形式，则乘积编程加法：

$$\begin{aligned} \theta_{ML} &= \arg \max_{\theta} \sum_{i=1}^m \log p_{model}(x^{(i)}; \theta) \\ &= \arg \max_{\theta} \mathbb{E}_{x \sim \hat{p}_{data}} \log p_{model}(x; \theta) \end{aligned}$$

上面的推导成立的原因：因为当重新缩放代价函数时， $\text{argmax}$  的结果不会改变，我们可以除以  $m$  得到和训练数据经验分布  $\hat{p}_{data}$  相关的期望作为准则，为啥会是  $\hat{p}_{data}$  而不是  $p_{model}$  呢，因为输入的数据  $x$  是来自训练数据而不是模型。

一种，解释最大似然估计的观点是将它看做最小化训练集上的经验分布和模型分布之间的差异，两者之间的差异程度可以通过 KL 散度度量：

$$D_{KL}(\hat{p}_{data} \parallel p_{model}) = \mathbb{E}_{x \sim \hat{p}_{data}} [\log \hat{p}_{data}(x) - \log p_{model}(x)]$$

在上式中，由于左边一项仅设计数据的生成过程，和模型无关。这意味着最小化 KL 散度时，可以省略第一项，得到：

$$-\mathbb{E}_{x \sim \hat{p}_{data}} [\log p_{model}(x)]$$

对比上式与前面的最大似然的式子，可以发现，最小化 KL 散度就是最小化分布之间的交叉熵，也就等价于最小化负对数似然，也就是最大似然了。

因为最大似然具有：一致性、统计效率，使其成为机器学习中的首选估计方法。

### 贝叶斯估计

这里主要说明最大后验 (MAP) 估计与后面正则项的关系！

MAP 的公式为：

$$\begin{aligned}\theta_{MAP} &= \arg \max_{\theta} p(\theta | \mathbf{x}) \\ &= \arg \max_{\theta} [\log p(\mathbf{x} | \theta) + \log p(\theta)]\end{aligned}$$

公式默认省略了数据的生成分布  $p(\mathbf{x})$ 。从上面可以看出来，相比于 ML，MAP 多了一项： $p(\theta)$ ，这一项就是参数  $\theta$  的先验概率分布了，该先验信息有助于减少最大后验点估计的方差，但缺点是增加了偏差。更重要的，看上面的公式：是不是跟带有正则项的目标函数比较相似。事实上，许多正则化估计方法，例如权重衰减正则化的最大似然学习，可以解释为贝叶斯推断的 MAP 近似。这个是英语正则化时加到目标函数的附加项对应着  $\log p(\theta)$ 。但并非所有的正则化惩罚对应着 MAP 贝叶斯推断。

MAP 贝叶斯推断提供了一个直观的方法来设计复杂但可解释的正则化。

### 随机梯度下降

随机梯度下降的核心是，梯度是期望（这一点可以从训练数据集上的梯度计算可以看出来，是一个求平均的过程）。既然是期望，就可以用小规模的样本近似估计。一个非常重要的优势：使用随机梯度下降，当总的样本数  $m$  不断增加时，我们都可以只使用小批量  $m'$  来计算梯度估计，而且该模型最终会在随机梯度下降抽样完训练集上的所有样本之前收敛到可能的最优测试误差。继续增加  $m$  不会延长达到模型可能的最优测试误差的时间，从这个角度来看，SGD 训练模型的渐进代价是关于  $m$  的函数的  $O(1)$  级别的!!!

## 促使深度学习发展的挑战

- 维数灾难
  - 局部不变性和平滑正则项
  - 流行学习
- 细节就不说了。

### 9.29.4 第六章

#### 深度前馈网络

##### 基于梯度的学习

代价函数：

1. 使用最大似然学习条件分布，正如前面说过的，基于最大似然的优化等价于基于 KL 散度和交叉熵的优化，准确的说，与训练数据和模型分布间的交叉熵等价。等价函数表示为：

$$J(\theta) = -\mathbb{E}_{\mathbf{x}, y \sim \hat{y}_{data}} \log p_{model}(y|\mathbf{x})$$

代价函数的具体形式与模型相关，取决于  $\log p_{model}$  的形式，比如当  $p_{model}$  具有高斯分布的形式时，最大后验概率具有均方误差的形式。具体的推导只需要把高斯分布的公式带入到上式就行。

使用最大似然来导出代价函数的方法的一个优势是，它减轻了为每个模型设计代价函数的负担。

贯穿神经网络设计的一个反复出现的主题就是代价函数的梯度必须足够大和具有足够的预测性。很多输出单元都会包含一个指数函数，这在它的变量取绝对值非常大的赋值时会造成饱和，所以，**通过负对数的形式可以在一定程度上抵消输出层的指数效果**。

##### 2. 学习条件概率统计量

这是另一种代价函数，可以把代价函数看做一个**泛函**。而不仅仅是一个函数，泛函的意思是函数到实数的映射。常用到变分法，这个方法可以在花书第 19 章找到。

输出单元：

任何可用作输出的神经网络单元也可以用于隐藏单元。通过下面的问题，可以看出，输出单元的形式(输出函数)与模型的分布形式相关。

##### 1. 用于高斯输出分布的线性单元

给定特征  $\mathbf{h}$ ，线性输出单元产生一个分量， $\hat{y} = \mathbf{W}^T \mathbf{h} + b$ ，由于输出还具有高斯函数的形式，因此等价函数就是前文提到的等价于均方误差了。

多说一句，为啥要用线性单元呢？因为对于高斯输出分布，我们只需要知道两个连续的变量就可以，一个是期望，一个是方差；这两个值基本上没什么限制，最大的特点就是它们是连续的值，所以用线性单元就可以了。

##### 2. 用于 Bernoulli 输出分布的 sigmoid 单元

为啥要引入 sigmoid 单元呢，因为该函数可以将所有的输入压缩到  $(0, 1)$  区间。对于 Bernoulli 输出，我们需要输出的是一个用于二分类的概率的值，但概率需要满足在 0 到 1 之间，因此 Sigmoid 就刚好合适。Sigmoid 输出单元的形式如下：

$$\hat{y} = \sigma(z), z = \omega^T \mathbf{h} + b$$

下面简单推导一下输出单元是 sigmoid 时的代价函数形式，还是基于最大似然实现代价函数。需要强调一下，若要使用最大似然，我们需要知道  $p_{model}(y|x)$  的形式，在 Bernoulli 分布中，首先假定费归一化的对数概率对 y 和 z 是线性的（不明白为什么可以），并且先不考虑 x 的条件，那么  $\hat{y}_{model}(y)$  为：

$$\begin{aligned}\log \tilde{P}(y) &= yz \\ \tilde{P}(y) &= \exp(yz) \\ P(y) &= \frac{\exp(yz)}{\sum_{y'=0}^1 \exp(y'z)} \\ P(y) &= \sigma((2y - 1)z)\end{aligned}$$

需要说明的一点：基于指数和归一化的概率分布在统计建模的文献中很常见。用于定义这种二值型变量分布的变量 z 被称为分对数。

所以损失函数就是：

$$\begin{aligned}J(\theta) &= -\log P(y|x) \\ &= -\log \sigma((2y - 1)z) \\ &= \zeta((1 - 2y)z)\end{aligned}$$

其中， $\zeta(x) = \log(1 + \exp(x))$  是 Softplus 函数。

几点重要的性质：

通过分析上面的损失函数，可以看到，只有在  $(1 - 2y)z$  取绝对值很大的负值时才会饱和。因此饱和只会出现在模型应得到正确答案的时候。反正对于极限情况下极度不正确的 z，softplus 函数完全不会收缩梯度。这个性质可以让基于梯度的学习可以很快的改正错误。

最大似然几乎总是训练 sigmoid 输出单元的优选方法。

3. 用于 Multinoulli 输出分布的 softmax 单元

主要看一下基于 softmax 输出单元的代价函数长什么样。

softmax 函数：

$$\text{softmax}(z)_i = \frac{\exp(z)_i}{\sum_j \exp(z_j)}$$

则负对数似然的形式是：

$$\log \text{softmax}(z)_i = z_i - \log \sum_j \exp(z_j)$$

上式中，第一项表示输入  $z_i$  总是对代价函数有直接的贡献。因为这一项不会饱和，因此即使  $z_i$  对第二项的贡献很小，学习依然可以进行！当最大化对数似然时，第一项会一直鼓励  $z_i$  被推高，而第二项则鼓励所有的 z 被压低。注意，可以把第二项大致近似为  $\max_j z_j$ 。这种近似对任何明显小于  $\max_j z_j$  的  $z_k, \exp(z_k)$  都是不重要的。所以，可以认为负对数似然总是强烈的惩罚最活跃的不正确的预测。也就

是说，如果当前的  $z_i$  就是所有的  $z$  中的最大值，那么这两项大致抵消，也就是没有多少损失；但如果不是最大的值，那么就会产生一个差值，而且对于负对数形式（上面公式乘以 -1）而言，第一项负数的绝对值小于第二项的正数，因此会贡献一个正的损失。

额外的一点是，softmax 正如前文提到的，可能会遇到饱和的情况，这种背景下的解决办法就是前文提到的方法了。

最后说一点，softmax 跟神经元之间的侧抑制类似，极端情况下会出现赢者通吃的形式。

#### 4. 其它输出类型

### 隐藏单元

#### 1. 整流线性单元及其扩展

- ReLU

$$\text{ReLU}(x) = \max(0, x)$$

优点是：已与优化，因为他们和线性单元非常相似。由于，在激活状态时，一阶导数处处为 1，这意味着相比于引入二阶效应的激活函数来说，它的梯度方向对于学习来说更加有用。

缺陷是：它们不能通过基于梯度的学习方法学习那些使它们激活为 0 的样本。也就是在非激活状态下，梯度为 0 了。

针对上面的缺陷，主要的改进下式中的  $\alpha$  的不同得到集中不同的改进方法：

$$\max(0, x_i) + \alpha_i \min(0, x_i)$$

- 绝对值整流

此时： $\alpha_i = -1$

- 渗漏整流线性单元 Leaky ReLU

此时， $\alpha_i$  被固定成一个类似 0.01 的小值。

- 参数化整流线性单元 PReLU

此时，将  $\alpha_i$  是可以学习的参数。

- maxout

进一步扩展了整流线性单元，就是将  $z$  划分成具有  $k$  个值的组，即每一组具有  $k$  个值。每个 maxout 单元则输出每组中的最大元素。

maxout 可以学习具有多达  $k$  段的分段线性的凸函数。maxout 单元因此可以视为学习激活函数本身，而不仅仅是单元之间的关系。使用足够大的  $k$ ，maxout 可以以任意精度近似任何凸函数。maxout 具有一些冗余来帮助抵抗灾难遗忘的现象，即神经网络忘记了如何执行它们过去训练的任务。

## 9.29 花书前两部分总结

---

整流线性单元及其扩展都是基于一个原则，就是如果它们的行为更接近线性，那么模型更容易优化。

### 2. Logistic sigmoid

$$g(z) = \sigma(z)$$

sigmoid 函数在大多定义域内都是饱和的，仅在 0 附近梯度比较大。这使得基于梯度的学习变得非常困难。现在不鼓励将其作为隐藏单元了。但在循环网络、许多概率模型以及一些自编码器中还是比较有吸引力的。

### 3. 双曲正切函数

$$g(z) = \tanh(z) = 2\sigma(2z) - 1$$

如果必须使用 sigmoid 激活函数时，双曲正切激活函数更好。并且  $\tanh(0) = 0$ ，而  $\sigma(0) = 1/2$ ，使后者看上去更像单位函数。因为  $\tanh$  函数在 0 附近跟单位函数更向。

### 4. 其它

包括：径向基函数，此函数大部分输入容易饱和，很难优化；softplus，是整流线性单元的平滑版本，不鼓励使用；硬双曲正切函数。

### 5. 个人补充

比较好的总结：[激活函数面面观 - 知乎](#)

激活函数需要的一些性质：

- 非线性

当激活函数是非线性（按照后面的万能近似性质，这里原文可能有误，所以改成非线性）的时候，一个两层的神经网络就可以逼近基本上所有的函数了。

- 可微性

用于基于梯度的学习。

- 单调性

当激活函数是单调的时候，单层网络能够保证是凸函数。

- 输出值的范围

当激活函数输出值是有限的时候，基于梯度的优化方法会更加稳定，因为特征的表示受有限权值的影响更显著；当激活函数的输出是无限的时候，模型的训练会更加高效，不过在这种情况下，一般需要更小的 learning rate。（不太理解！）

对于每一个激活函数，简单补充如下：

- Sigmoid

它能够把输入的连续实值“压缩”到 0 和 1 之间。

缺点：

- Sigmoids saturate and kill gradients. sigmoid 有一个非常致命的缺点，当输入非常大或者非常小的时候，这些神经元的梯度是接近于 0 的，从图中可以看出梯度的趋势。所以，你需要尤其注意参数的初始值来尽量避免 saturation 的情况。如果你的初始值很大的话，大部分神经元可能都会处在 saturation 的状态而把 gradient kill 掉，这会导致网络变的很难学习。
- Sigmoid 的 output 不是 0 均值。这是不可取的，因为这会导致后一层的神经元将得到上一层输出的非 0 均值的信号作为输入。产生的一个结果就是：如果数据进入神经元的时候是正的 (e.g.  $x > 0$ ) elementwise  $inf = w^T x + b$ ，那么针对  $w$  计算出的梯度 (简单的可以认为就是  $x$ ，见本章的后向传播的解释) 也会始终都是正的。
- 当然了，如果你是按 batch 去训练，那么那个 batch 可能得到不同的信号，所以这个问题还是可以缓解一下的。因此，非 0 均值这个问题虽然会产生一些不好的影响，不过跟上面提到的 kill gradients 问题相比还是要好很多的。

- tanh

与 sigmoid 不同的是，tanh 是 0 均值的。因此，实际应用中，tanh 会比 sigmoid 更好（毕竟去粗取精了嘛）。

- ReLU

梯度消失和梯度爆炸在 relu 下都存在，sigmoid 下没有梯度爆炸。

随着网络层数变深，activations 倾向于越大和越小的方向前进，往大走梯度爆炸（回想一下你在求梯度时，每反向传播一层，都要乘以这一层的 activations），往小走进入死区，梯度消失。

这两个问题最大的影响是，深层网络难于 converge。

BN 和 xavier 初始化（经指正，这里最好应该用 msra 初始化，这是 he kaiming 大神他们对 xavier 的修正，其实就是 xavier 多除以 2）很大程度上解决了该问题。

### 怎么理解梯度弥散和梯度爆炸？ - 知乎

个人认为梯度不稳定的根本原因在于神经网络的高度非线性，这使得优化过程中既存在很小的梯度，同时也存在很大的梯度

relu 只是缓解了梯度消失的问题，并不是解决了梯度消失问题，前向神经网络由于每层的权重相关性较小，bp 过程相乘之后相对来说不易产生消失的梯度，再加上 relu 本身不具饱和性，所以才会取得很好的效果。但是并不是所有的神经网络都用 relu，比如循环神经网络，当然也可以通过加入一些线性通路来缓解。

ReLU 的负半轴梯度为 0，所以有时候（比较少见）也还是会梯度消失，这时可以使用 PReLU 替代，如果用了 PReLU 还会梯度弥散和爆炸，请调整初始化参数，对自己调参没信心或者就是懒的，请直接上 BN。

至于 sigmoid 为什么会有梯度消失现象，是因为  $sigmoid(x)$  在不同尺度的  $x$  下的梯度变化太大了，而且一旦  $x$  的尺度变大，梯度消失得特别快，网络得不到更新，就再也拉不回来了。我想 sigmoid 函数的情况，可以通过 sigmoid 函数的求导看出来了，因为求导后的结果同样包含 sigmoid 函数！

## 9.29 花书前两部分总结

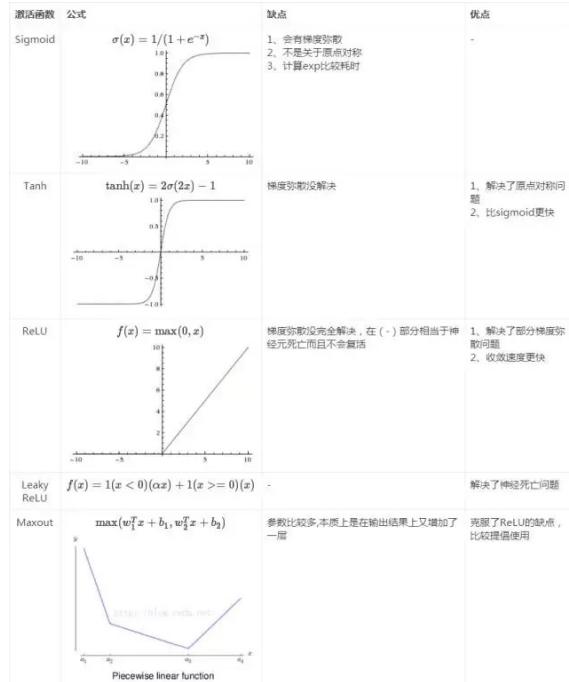


Fig 9.25. 激活函数对比

梯度弥散就是梯度消失。

## 架构设计

主要几点：

- 万能近似性质

万能近似定理：一个前馈神经网络如果具有线性输出层和至少一层具有任何一种挤压性质的激活函数，例如 sigmoid，的隐藏层，只要给予足够数量的隐藏单元，它可以以任意的精度来近似任何一个从有限维空间到另一个空间有限维空间的 Borel 可测函数。

- 深度

Montufar 等人指出，一些用深度深度整流网络表示的函数可能需要浅层网络（一个隐藏层）指数级的隐藏单元才能表示！一个公式：具有  $d$  个输入、深度为 1、每个隐藏层具有  $n$  个单元的深度整流网络可以描述的线性区域的数量是：

意味着，这是深度 1 的指数级。

- 其它架构上的考虑

比如跳跃连接等。

## 反向传播

这部分单独写。感觉重要的一个公式是：微积分中的链式法则。

## 9.29.5 第七章

深度学习中的正则化

根据第五章的定义，正则化是对学习算法的修改 ----旨在减少泛化误差而不是训练误差。

目前有许多正则化策略，有些策略向机器学习模型添加限制参数值的额外约束、有些策略向目标函数添加额外项对参数值进行软约束。

估计的正则项以偏差的增加换取方差的减少。

按照花书的意思，正则化的目标是使模型从过拟合状态变为正常状态。

### 参数范数惩罚

其实，主要是 L1、L2 范数正则项。

- L2 参数范数正则化

通常被称为权重衰减 (weight decay)，又叫岭回归或 Tikhonov 正则。

前面提到了，L2 范数有助于病态条件下的优化。这是因为加入权重衰减的效果是沿着 Hessian 矩阵  $H$  的特征向量所定义的轴缩放未加约束时的最优值  $\omega^*$ 。具体来说，我们会根据  $\frac{\lambda_i}{\lambda_i + \alpha}$  因子缩放与  $H$  的第  $i$  个特征向量对齐的  $\omega^*$  分量。若该方向特征值较大，则正则化的影响较小，若该方向的特征值较小，则该分量将会收缩到几乎为 0.

- L1 参数范数正则化

最终的特性是，会产生更稀疏的解。会起到特征选择的作用。

补充：

### L1 正则化与 L2 正则化

再这篇文章中，作者提到了本笔记前面提到的 MAP 中的重点说明。也就是说，L1 范数等价于先验让权重  $\omega$  服从标准拉普拉斯分布，即概率密度函数为  $1/2 \exp(-|x|)$ ，则 MAP 多出来的一项就是 L1 的形式了。而通过拉普拉斯分布可知，当  $\omega$  取到 0 的概率特别大。也就是说我们提前先假设了  $\omega$  的解更容易取到 0。

而对于 L2 范数，其等价于先验让权重  $\omega$  服从标准正态分布，即概率密度为  $1/\sqrt{2\pi} \exp(-(x)^2/2)$ ，带入这个权重的先验分布，就可以得到 L2 范数的形式了。根据正态分布可以知道，我们预先假设了  $\omega$  的最终值可能取到 0 附近的概率特别大。

**结构风险最小化：**在经验风险最小化的基础之上（也就是训练误差最小化），尽可能采用简单的模型，以此提高泛化预测精度。

## 数据集增强

### 噪声鲁棒性

对于某些模型，向输入添加方差极小的噪声等价于对权重施加范数惩罚。一般情况下，注入噪声远比简单的收缩参数强大，特别是噪声被添加到隐藏单元时，会更加强大，比如 Dropout 就是这种做法的主要发展方向。

## 9.29 花书前两部分总结

---

另一种正则化模型的噪声使用方式是将其添加到权重上。这项技术主要用于循环神经网络。

### 半监督学习

### 多任务学习

当模型的一部分被多个额外的任务共享时，这部分将被约束为良好的值。

### 提前终止

通过分析，可以发现，提前终止与权重衰减有非常大的联系，尤其是权重衰减系数的倒数。

### Bagging 等集成方法

模型平均奏效的原因是不同的模型通常不会在测试集上产生完全相同的误差。

Bagging 是一种允许重复多次使用同一种模型，训练算法和目标函数的方法。

### Dropout

Dropout 提供了一种廉价的 Bagging 集成近似，能够训练和评估指数级数量的神经网络。

Dropout 训练与 Bagging 训练不太一样，在 Bagging 的情况下，所有模型都是独立的。在 Dropout 的情况下，所有模型共享参数，其中每个模型继承父神经网络参数的不同子集。

下面一段话来自 Gluon 教程：对一个随机采样的一个批量的训练数据分别训练一个元神经网络子集的分类器，且每个神经网络子集使用同一套参数。

### 对抗训练

## 9.29.6 第八章

### 深度模型中的优化

### 神经网络优化中的挑战

#### 1. 病态

病态体现在随机梯度下降会卡在某些情况，此时即使很小的更新步长也会增加代价函数。数学原理如下：

将跟新后的代价函数的新值进行泰勒二阶近似：

$$f(x_0 - \epsilon g) = f(x_0) - \epsilon g^T g + \frac{1}{2} \epsilon^2 g^T H g$$

我们主要关注上式的后两项。

$$-\epsilon g^T g + \frac{1}{2} \epsilon^2 g^T H g$$

很明显，当  $\frac{1}{2}\epsilon^2 g^T Hg$  超过  $-\epsilon g^T g$  时，梯度的病态就会成为问题。虽然牛顿法在解决图优化问题中的病态条件比较有效果，但对神经网络效果不太好。

2. 局部极小值
3. 高原、鞍点和其它平坦区域
4. 悬崖和梯度爆炸
5. 长期依赖

当计算图变得极深时，神经网络优化算法面临的另一个难题就是长期依赖问题。深层的计算图不仅存在于前馈网络中，还存在于循环网络中。其数学原理如下：

假设某个计算图中包含一条反复与矩阵  $W$  相乘的路径，那么  $t$  步后，相当于乘以  $W^t$ 。假设  $W$  有特征分解  $W = V \text{diag}(\lambda) V^{-1}$ 。在这种简单的情况下，很容易看出：

$$W^t = (V \text{diag}(\lambda) V^{-1})^t = V \text{diag}(\lambda)^t V^{-1}$$

所以，当特征值  $\lambda_i$  不在 1 附近时，若在量级大于 1 则会爆炸；若小于 1 时则会消失。梯度消失和爆炸问题，是指该计算图上的梯度也会因为  $\text{diag}(\lambda)$  大幅变化。

## 基本算法

- SGD

---

### Algorithm 1 SGD

---

**Require:** 学习率  $\epsilon$ , 初始参数  $\theta$

**Ensure:** 更新后的参数  $\theta$

1: **repeat**

从训练样本中随机采样  $m$  个样本，构成一个小批量  
计算小批量的梯度：

$$g \leftarrow \frac{1}{m} \sum_i \nabla_{\theta} L(f(x^i; \theta), y^i)$$

更新参数： $\theta \leftarrow \theta - \epsilon g$

2: **until** 条件满足

---

虽然随机梯度下降仍然是非常受欢迎的优化方法，但其学习过程有时会很慢。特别是高曲率，小但一致的梯度，或是带噪声的梯度。

- 动量

动量法旨在加速学习过程，特别是处理高曲率，小但一致的梯度，或是带噪声的梯度。动量算法积累之前梯度指数级衰减的移动平均，并且继续沿该方向移动。算法如下：

- Nesterov 动量

Nesterov 动量与标准动量之间的区别体现在梯度计算上，Nesterov 动量中，梯度计算在施加当前速度之后。

在凸批量梯度的情况下，Nesterov 动量将额外误差收敛率从  $O(1/k)$  改进到  $O(1/k^2)$ ，其中  $k$  是迭代次数。但在随机梯度的情况下，Nesterov 动量没有改进收敛率。

---

### Algorithm 2 使用动量的 SGD

---

**Require:** 学习率  $\epsilon$ , 初始参数  $\theta$ , 动量参数  $\alpha$ , 初始速度  $v$

**Ensure:** 更新后的参数

**repeat** 随机抽样  $m$  个样本构成一个小批量

计算小批量上的梯度:

$$g \leftarrow \frac{1}{m} \sum_i \nabla_{\theta} L(f(x^i; \theta), y^i)$$

计算动量:  $v \leftarrow \alpha v - \epsilon g$

更新:  $\theta \leftarrow \theta + v$

**until** 满足停止条件

---

---

### Algorithm 3 使用 Nesterov 动量的 SGD

---

**Require:** 学习率  $\epsilon$ , 初始参数  $\theta$ , 动量参数  $\alpha$ , 初始速度  $v$

**Ensure:** 更新后的参数

**repeat** 随机抽样  $m$  个样本构成一个小批量

首先更新一次参数:  $\tilde{\theta} \leftarrow \theta + v$

计算小批量上的梯度:

$$g \leftarrow \frac{1}{m} \sum_i \nabla_{\tilde{\theta}} L(f(x^i; \theta), y^i)$$

计算动量:  $v \leftarrow \alpha v - \epsilon g$

更新:  $\theta \leftarrow \theta + v$

**until** 满足停止条件

---

## 参数初始化策略

参数的初始点对算法的性能影响很大。也许完全可知的唯一特性是初始参数需要在不同单元间“破坏对称性”。这或许有助于确保没有输入模式丢失在前向传播的零空间中，没有梯度模式丢失在反向传播的零空间中。

现在存在 Normalized initialization 或者 Xaive 初始化方法。在该模型下，这个初始化方案确保了达到收敛所需的训练迭代总次数独立于深度。

## 自适应学习率算法

- AdaGrad

AdaGrad 算法，独立地适应所有模型参数的学习率，缩放每个参数反比其所有梯度历史平方值总和的平方根。具有损失最大偏导的参数相应地有一个快速下降的学习率，而具有小偏导的参数在学习率上具有较小的下降，净效果就是在参数空间中，更为平缓的倾斜方向会取得更大的进步。

但有一个缺点：从训练开始时积累平方会导致有效学习率过早或过量的减少。在一些深度学习模型上效果不错，但不是全部。算法如下：

---

### Algorithm 4 AdaGrad 算法

---

**Require:** 全局学习率  $\epsilon$ , 初始参数  $\theta$ , 小常数  $\delta$ , 为了数值稳定, 一般设为  $10^{-7}$

**Ensure:** 自适应学习率

初始梯度累计变量  $r = 0$

**repeat** 从训练集中随机抽样  $m$  个样本构成一个小批量

计算小批量上的梯度:

$$g \leftarrow \frac{1}{m} \sum_i \nabla_{\tilde{\theta}} L(f(x^i; \theta), y^i)$$

计算梯度累计量 (梯度历史平方值总和):  $r \leftarrow r + g \odot g$

计算更新:

$$\Delta\theta \leftarrow -\frac{\epsilon}{\sqrt{r} + \delta} \odot g$$

应用更新:  $\theta \leftarrow \theta + \Delta\theta$

**until** 达到更新准则

---

- RMSProp

RMSProp 算法修改 AdaGrad 以在非凸设定下效果更好，改变梯度积累为指数加权的移动平均。AdaGrad 旨在应用于凸问题时快速收敛。当应用于非凸函数训练时神经网络时，学习轨迹可能穿过很多不同的结构，最终到达一个局部是凸碗的区域。AdaGrad 根据平方梯度的整个历史收敛学习率，可能使得学习率在达到这样的凸结构之前就变得太小了。RMSProp 使用指数衰减平均以丢弃遥远过去的历史，使其能够再找到凸碗状结构后快速收敛，它像一个初始化于该碗状结构的 AdaGrad 算法实例。算法如下：

具有 Nesterov 动量的 RMSProp，思想大体一致，先暂时更新参数  $\theta$ ，然后用这个更新后的参数计算梯度，就如 Nesterov 动量 SGD 一样。

### Algorithm 5 RMSProp 算法

**Require:** 全局学习率  $\epsilon$ , 初始化参数  $\theta$ , 小常数  $\delta$ , 为了数值稳定, 一般设为  $10^{-7}$ , 衰减速度  $\rho$

**Ensure:** 自适应学习率

初始梯度累计变量  $r = 0$

**repeat** 从训练集中随机抽样  $m$  个样本构成一个小批量

计算小批量上的梯度:

$$g \leftarrow \frac{1}{m} \sum_i \nabla_{\theta} L(f(x^i; \theta), y^i)$$

计算梯度累计量 (指数加权的移动平均):  $r \leftarrow \rho r + (1 - \rho)g \odot g$

计算更新:

$$\Delta\theta \leftarrow -\frac{\epsilon}{\sqrt{r + \delta}} \odot g$$

应用更新:  $\theta \leftarrow \theta + \Delta\theta$

**until** 达到更新准则

---

RMSProp 是现阶段深度学习从业者经常采用的优化方法。

- Adam

Adam 包含了偏置修正, 修正从原点初始化的一阶矩和二阶矩的估计。Adam 通常被认为对超参数的选择相当鲁棒, 尽管学习率有时需要跟从建议的默认修改。

- AdaDelta

RMSProp 针对 Adagrad 在迭代后期可能较难找到有用解的问题, 对小批量随机梯度按元素平方项做指数加权移动平均而不是累加。另一种应对该问题的优化算法叫做 Adadelta。有意思的是, 它没有学习率超参数。

Adadelta 算法也像 RMSProp 一样, 使用了小批量随机梯度按元素平方的指数加权移动平均变量  $s$ , 并将其中每个元素初始化为 0。给定超参数  $\rho$  且  $0 \leq \rho < 1$ , 在每次迭代中, RMSProp 首先计算小批量随机梯度  $g$ , 然后对该梯度按元素平方项  $g \odot g$  做指数加权移动平均, 记为  $s$ :

$$s \leftarrow \rho s + (1 - \rho)g \odot g$$

然后, 计算当前需要迭代的目标函数自变量的变化量  $g'$ :

$$g' \leftarrow \frac{\sqrt{\Delta x + \delta}}{\sqrt{s + \delta}} \odot g$$

其中  $\delta$  为数值稳定而添加的常数。和 Adagrad 与 RMSProp 一样, 目标函数自变量中每个元素都分别拥有自己的学习率。上式中的  $\Delta x$  初始化为零张量, 并记录  $g'$  暗元素平方的指数加权移动平均:

$$\Delta x \leftarrow \Delta x + (1 - \rho)g' \odot g'$$

---

**Algorithm 6** Adam 算法

---

**Require:** 全局学习率  $\epsilon$ , 建议为 0.001, 初始参数  $\theta$ , 小常数  $\delta$ , 为了数值稳定, 一般设为  $10^{-7}$

**Require:** 矩估计的指数衰减速率  $\rho_1$  和  $\rho_2$ , 在区间  $[0, 1]$ , 建议分别为 0.9 和 0.999

**Ensure:** 自适应学习率

初始梯度累计变量  $s = 0$  和  $r = 0$

**repeat** 从训练集中随机抽样  $m$  个样本构成一个小批量

计算小批量上的梯度:

$$g \leftarrow \frac{1}{m} \sum_i \nabla_{\tilde{\theta}} L(f(x^i; \theta), y^i)$$

$$t \leftarrow t + 1$$

$$\text{更新有偏一阶矩估计: } s \leftarrow \rho_1 s + (1 - \rho_1) \odot g$$

$$\text{更新有偏二阶矩估计: } r \leftarrow \rho_2 r + (1 - \rho_2) g \odot g$$

$$\text{修正一阶矩的偏差: } \hat{s} \leftarrow \frac{s}{1 - \rho_1^t}$$

$$\text{修正二阶矩的偏差: } \hat{r} \leftarrow \frac{r}{1 - \rho_2^t}$$

计算更新:

$$\Delta\theta \leftarrow -\epsilon \frac{\hat{s}}{\sqrt{\hat{r}} + \delta} \odot g$$

$$\text{应用更新: } \theta \leftarrow \theta + \Delta\theta$$

**until** 达到更新准则

---

最后, 自变量迭代步骤和小批量随机梯度下降类似:

$$x \leftarrow x - g'$$

一些资料: [Adam 优化算法 -知乎](#)

### 选择正确的优化算法

花书说, 目前最流行的并且使用很高的优化算法包括 SGD, 包含动量的 SGD, RMSProp, 具有栋梁的 RMSProp, AdaDelta 和 Adam。选择哪一个算法似乎主要取决于使用者对算法的熟悉程度, 以便调节超参数。

### 优化策略和元算法

1. 批标准化
2. 坐标下降
3. 监督与训练  
等

## 9.30 梯度消失 - 梯度爆炸专题

参考文献 [1]: [神经网络训练中的梯度消失和梯度爆炸 -知乎](#)

## 9.31 ROI Pooling

---

本质上是因为梯度反向传播中的连乘效应。

参考文献 [2]: [BP 算法中为什么产生梯度消失 - 知乎](#)

此外，关于梯度爆炸与梯度消失，我觉着花书上说的就先这样认为是对的吧，虽然非常简单。

### 9.30.1 自圆其说的解释

在反向传播过程中，需要两个大步骤的迭代向前，首先是是对当前层输出的激活值求导；其次是对当前层的权重求导。前者在求导过程中需要不停地乘以更早层的权重矩阵，而借鉴在 RNN 中（网络展开后表示的权重是共享的）分析长期依赖时的想法，也就是说权重矩阵进行特征值分解，如果其特征值远大于 1，那么反向传播中链式法则的连乘效应会让梯度在那个方向爆炸，反之如果某方向的特征值小于一，就会发生梯度消失！

## 9.31 ROI Pooling

参考 [1]: [Region of interest pooling explained](#)

The purpose is to perform max pooling on inputs of nonuniform size to obtain fixed-size feature maps (e.g. 7\*7).

在 Fast R-CNN 中最先提出，ROI Pooling 层需要两个输入，一个是 ROI 区域，一个是来自 CNN 的 Feature Map。而一般输入的 ROI 区域的个数是固定的，比如在 Faster RCNN 中是 300 个。所以 ROI Pooling 的结果是，共含输入的 ROI 个数的固定大小的 Feature Map。

原文：

What does the ROI pooling actually do? For every region of interest from the input list, it takes a section of the input feature map that corresponds to it and scales it to some pre-defined size (e.g. 7\*7). This scaling is done by: 1) Dividing the region proposal into equal-size sections, 2) finding the largest value in each section, 3) copying these max values to the output buffer.

**ROI Pooling: The result is that from a list of rectangles with different sizes (ROI) we can quickly get a list of corresponding feature maps with a fixed size.**

What's the benefit of ROI pooling? One of them is processing speed. If there are multiple object proposals on the frame (and usually there'll be a lot of them), we can still use the same input feature map for all of them. Since computing the convolutions at early stages of processing is very expensive, this approach can save us a lot of time.

## 9.32 深度学习面试 100 题

### 9.32.1 Tensorflow 计算图

Tensorflow 计算图是一种有向图表示的计算过程。其中每一个节点代表一种计算，节点之间的边代表张量的依赖关系以及流通。反向传播中，向计算图增加新的节点。

### 9.32.2 DL 调参

- 参数初始化

比如常用的 Xavier(), 适用于普通计划函数, 如 (tanh, sigmoid) 等  
He 初始化, 适用于 ReLU

normal 高斯分布初始化

svd 初始化, 对 RNN 具有较好的结果

- 数据预处理方式

比如下面提到的数据归一化方法等, 如白化。

- 训练技巧

要做梯度归一化, 即计算出来的梯度除以 minibatch size

梯度裁剪, 限制最大梯度

dropout: 对小数据防止过拟合具有很好的效果, 一般设置概率为 0.5。小数据上 dropout + sgd 效果提升都比较明显。Dropout 的位置比较讲究, 对于 RNN, 建议放在 “输入 ->RNN” 与 “RNN -> 输出”的位置。

adam, adadelta 等, 在小数据上, 实验效果的不如 SGD, 尽管 SGD 的收敛速度会慢一些, 但是最终收敛后的结果一般比较好。如果选择 SGD, 学习率可以从 1.0 或 0.1 的学习率开始, 隔一段时间在验证集上进行测试如果 Loss 没有下降, 就对学习率减半。据说, adadelta 一般在分类问题上效果比较好, adam 在生成问题上效果比较好。

除了 gate 之类的地方, 尽量不要使用 sigmoid, 可以使用 tanh 或者 relu 之类的激活函数。sigmoid 函数在 -4 到 4 之间时才有较大的梯度, 之外的区间, 梯度接近 0, 很容易造成梯度消失。另外, 输入 0 均值, sigmoid 函数的输出不是 0 均值的。

rnn 的 dim 和 embedding size 一般从 128 上下开始调整。batch size 一般从 128 左右开始调整, 并不是越大越好。

- 尽量对数据进行 shuffle

- Ensemble

模型平均, 凑效的原因在于不同的模型通常不会在测试集上产生完全相同的误差。在误差完全相关的时候, 模型平均没有帮助, 错误完全不相关时, 集成后的平方误差的期望可以降低为原来的  $\frac{1}{k}$ , 其中  $k$  是模型的个数。

### 9.32.3 为什么 CNN 在 CV、NLP、Speech、AlphaGo 里面都有应用

题目中的这几个不相关的问题的相似性在哪里? CNN 通过什么手段来抓住了这个共性?

回答: 以上几个不相关的问题的相关性在于, 都存在局部和整体的关系, 由低层次的特征经过组合, 组成高层次的特征, 并且得到不同特征之间的空间相关性。

CNN 抓住此共性的手段主要有四个: 局部连接、权值共享、池化操作、多层次结构。

局部连接是网络可以提取数据的局部特征；权值共享可以降低网络的训练难度，一个 filter 只提取一种特征，在整个图片（或语音、文本）中进行卷积；赤化操作和多层次结构一起，实现了数据的降维，将低层次的局部特征组合成较高层次的特征，从而对整幅图像进行表示。

### 9.32.4 为什么要引入非线性激励函数

首先，对于神经网络来说，如果没有非线性激励函数，那么整个网络就也是线性的。

其次，非线性变换是深度学习有效的原因之一。非线性变换相当于对空间进行变换，变换完成后相当于对问题空间进行简化，原来线性不可解的问题变得可以解了。

最重要的是，万能近似性质说明：一个前馈神经网络如果具有线性输出层和至少一层具有“挤压”性质的激活函数的隐藏层，只要给予网络足够数量的隐藏单元，它可以以任意精度来近似任何一个有限维空间到另一个有限维空间的 Borel 可测函数。

### 9.32.5 为什么 ReLU 要好于 tanh 和 sigmoid function

首先，采用 sigmoid 函数，计算激活函数时，计算量大，反向传播求误差梯度时，求导涉及除法和指数计算，计算量相对大，采用 ReLU 激活函数，整个过程的计算量节省很多。

其次，对于深层网络，sigmoid 函数反向传播时，很容易就会出现梯度消失的情况，这种现象称为饱和，从而无法完成深层网络的训练。而 ReLU 就不会有饱和倾向，不会有特别小的梯度出现。

然后，ReLU 会使一些神经元的输出为 0，这样就造成了网络的稀疏性，并且减少了参数相互依存关系，缓解了过拟合问题的发生。当然现在也有一些对 ReLU 的改进，如 PReLU, LeakyReLU 等，在不同的数据集上会有一些训练速度上或者准确率上的改进。

### 9.32.6 为啥 LSTM 模型中既存在 sigmoid 又存在 tanh 两种激活函数

为什么 LSTM 模型中既存在 sigmoid 函数又存在 tanh 激活函数呢，而不是只存在一种？

首先这两种激活函数在 LSTM 模型中的作用是不同的，sigmoid 产生 gate 信号，数值在 0 到 1 之间，所以这个一般只有 sigmoid 最直接了；tanh 用在了状态和输出上，是对数据的处理，这个用其它激活函数或许也可以。

### 9.32.7 如何解决 RNN 梯度爆炸和弥散问题

首先是梯度爆炸问题，花书上也提到了，Thomas Mikolov 首先提出了一种简单的启发性的解决方案，就是当梯度大于一定的阈值的时候，将它截断为一个较小的数。

对于梯度弥散问题，为了解决梯度弥散问题，第一种方法是将随机初始化权重  $\omega$  更改为一个有关联的矩阵初始化。第二种方法是使用 ReLU 代替 Sigmoid 函数。

ReLU 函数的梯度不是 0 就是 1，因此，神经元的梯度将始终为 1，而不会当梯度传播了一定时间之后变小。

### 9.32.8 什么样的数据集不适合深度学习

一、数据集太小，数据样本不足时，深度学习相对其它机器学习算法，而没有明显优势

二、数据集没有局部相关性，目前深度学习表现比较好的领域主要是图像、语音、自然语言处理等领域，这些领域的一个共性是具有局部相关性。对于没有这样局部相关性的数据集，不适于使用深度学习进行处理。

### 9.32.9 如何解决梯度消失和梯度膨胀

- 梯度消失

根据链式法则，如果每一层神经元对上一层的输出的偏导数乘上权重结果都小于 1 的话，那么即使这个结果是 0.99，在足够多层传播后误差对输入层的偏导会趋近于 0.

可以采用 ReLU 激活函数有效的解决梯度消失的情况，也可以用 BN 解决这个问题。此外，在 Identity Mapping of Resnet 那篇文章中，提到通过 Identity mapping 可以减轻这个问题。

- 梯度膨胀

根据链式法则，如果每一层的神经元对上一层的输出的偏导数乘上权重的结果都大于 1 的话，在经过足够多层传播之后，误差对输入层的偏导数会趋于无穷大。

可以通过激活函数来解决、或用 BN 解决。

### 9.32.10 激活函数的真正意义

激活函数的真正意义？一个激活函数需要具有那些必要的属性？还有那些属性是好的但不必要？

- 非线性

及倒数不是常熟。

- 几乎处处可微

可微保证了在优化中梯度的可计算性。

- 计算简单

- 非饱和性

饱和指的是梯度在某些区间接近于零（即梯度消失），使得参数无法继续更新的问题。

- 单调性

即导数符号不变。

## 9.33 数据的归一化

---

- 输出范围有限

有限的输出范围使得网络对于一些较大的输入也比较稳定。

### 9.32.11 梯度下降训练神经网络容易收敛到局部最优，为什么还应用广泛

深度神经网络“容易收敛到局部最优”很可能是一种假象，实际情况是，我们可能从来没有找到过局部最优，更别说全局最优了。

现在看来，神经网络训练的困难主要是鞍点问题。另一个比较好的消息是，即使具有局部最小值，具有较差的 loss 的局部极值的吸引域也是很小的。

## 9.33 数据的归一化

### 9.33.1 为什么需要归一化

数据归一化后，最优解的寻找过程明显变得平缓，更容易正确的收敛到最优解。归一化除了能提高求解速度，还可能提高计算精度。另外，归一化后的数据的量级一致，利于计算。

### 9.33.2 归一化的方法

常用归一化方法：

- 线性归一化函数 (Min-Max Scaling)

$$x' = (x - \min(x)) / (\max(x) - \min(x))$$

通过上式，把输入数据转换到 0 到 1 之间。实际使用时，不同样本集得到的 min 和 max 可能不同，导致归一化结果不稳定，从而使模型后续使用也不稳定。

- 0 均值标准化 (Z-standardization)

$$x' = (x - u) / \theta$$

其中， $u$  为样本均值， $\theta$  为样本方差。转换后的数值服从均值为 0，方差为 1 的高斯正态分布。

应用场景：原始数据（近似）高斯分布，否则归一化后的效果很差。

- 非线性归一化

包括对数，指数，正切等。

应用场景：数据分化较大，有些很大，有些很小，可能用此方法将数值映射到一个比较小的范围进行处理。

### 9.33.3 小结

对于要求距离的分类、聚类、相似度、协方差等，数据符合或者近似符合高斯正态分布时，PCA 降维时，常用 0 均值归一化，可以得到较好的效果。

对于其它情况，如果数据分化不是很大，可以采用线性归一化处理。

如果数据分化很大，可以用非线性处理。

### 9.34 待续

# Chapter 10

## Image Processing

### 10.1 Feature Extraction

#### 10.1.1 SIFT

详细信息可以参考: [SIFT CSND 博客](#)  
我的未验证实现: [SIFT Triloo Github](#)



# Chapter 11

## Feature Extraction

总结文章: [Object Detection 总结 Two Stage](#)

总结文章: [Object Detection 总结 One Stage](#)

### 11.1 Selective Search

#### 11.1.1 Efficient Graph-Based Image Segmentation

Selective Search 基于文章 [4] 中基于图分割的图像分割技术。

论文 [4] 提出的是一种基于贪心选择的图像分割算法，论文中把图像中的每个像素表示图上的一个节点，每一条连接节点的无向边都具有一个权重 (weights)，以衡量其连接的两个节点之间的不相似度，这篇论文的创新点在于该算法能够根据相邻区域在特征值上变化速度的大小动态调整分割阈值。这个特征值就是类间距离和类内距离，如果类间距离大于类内距离就认为是两个区域。定义类内距离为对应区域的最小生成树（因为把图像看做一个连接图，所以每个区域可以用最小生成树来表示）中权重最大的边的权重值，类间距离定义为两个区域内相邻点的最小权重边，如果两个区域没有相邻边则取无穷大。但是这样其实还是有问题，比如一个类中只有一个点的时候，它的类内距离为 0，这样就没法搞了（每个点都变成了一类，此时类内距离变为 0），所以作者又引入了一个阈值函数，用来表示两个区域的区域的类间距离至少要比类内距离大多少才能认为是两个区域。

判断两个点  $C_1$  与  $C_2$  是否属于同一类的判定：

$$D(C_1, C_2) = \begin{cases} \text{true} & \text{if } Dif(C_1, C_2) > MInt(C_1, C_2) \\ \text{false} & \text{otherwise} \end{cases}$$

其中， $MInt(C_1, C_2)$  为判断类间间距  $Dif$  的阈值，由类内间距计算得到：

$$MInt(C_1, C_2) = \min(Int(C_1) + \tau(C_1), Int(C_2) + \tau(C_2))$$

其中， $\tau$  即为控制当类间的最短距离。一般取为  $C$  的负相关函数， $k$  为常数：

$$\tau(C) = k/|C|$$

当  $k$  的取值大时，类间距要求大，所以分割后的区域面积也会较大，否则区域面积较小。

具体的分割过程，可以参考文章 [4] 的 **Algorithm 1**。

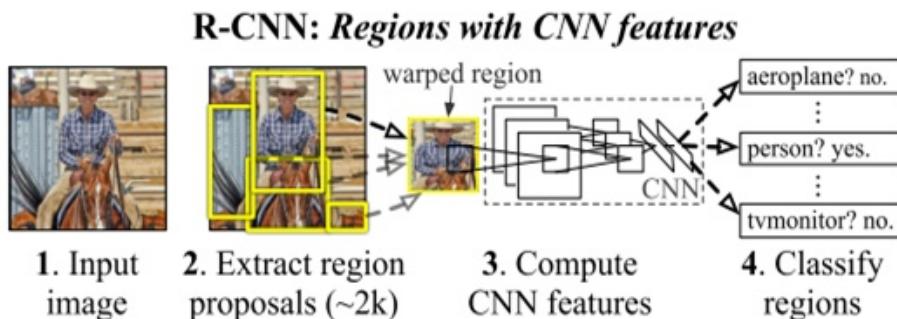
### 11.1.2 Selective Search

主要参考文献: [34]

接下来介绍 Selective Search 算法, 该算法利用 [4] 的图分割算法获得的分割区域结果, 再一次根据一些搜索策略 (相似度) 做了一个聚类。也是和上面的思路一致, 首先根据获得的区域, 算出每个区域和其他区域的相似度, 不相邻的和自身与自身的相似度都设置为 0, 得到一个  $N \times N$  的矩阵, 然后将相似度最大的合并, 再次计算相似度矩阵 (这里是增量更新, 只需要计算新生成的区域和其他区域的相似度就可以了), 这样合并一次较少一个区域, 对于  $N$  个区域需要执行  $N-1$  次合并, 最终得到一个区域。对于相似度的度量, 作者主要选取了颜色和区域两大块, 颜色作者比较了 8 种模型, 最终选择了 HSV, 区域主要考量大小、纹理和吻合度 (相交包含关系) 这三个因素。

## 11.2 Region CNN

### 11.2.1 概述



**Fig 11.1. RCNN 整体思想**

这里的 Extract region proposals 是基于 Selective Search 实现的, 然后将这些候选框输入到 CNN 进行特征提取, 最后利用 SVM 对提取的特征进行分类。

**Boundingbox 回归**

## 11.3 SPP Net

参考文献: [9]

在这之前的网络都要求输入图像数据的大小是固定的, 而本文提出了 Spatial pyramid pooling, 可以允许输入图像是任意大小, 并且都会生成一个固定大小的表示。

此外, Pyramid pooling 用于目标检测时, 可以提高 RCNN 的速度, 因为只需要对整幅图像提取一次特征, 然后在 Pooling 层进行获取 sub-images。

### 11.3.1 背景与相关工作

固定输入图像的大小, 不利于识别在不同 Scale 下的目标。

那么为什么会有这种限制呢？传统的 CNN 结构可以认为由 Convolution layers 和 Fully connected layers 两部分组成，前者对输入尺寸没有要求，且可以输出任意的尺寸；但是后者的输入对尺寸有要求，所以限制了输入图像的尺寸。训练的时候也可以使用不同尺寸的图像进行训练。

改进的 Spatial Pyramid Pooling 在网络结构中的位置：

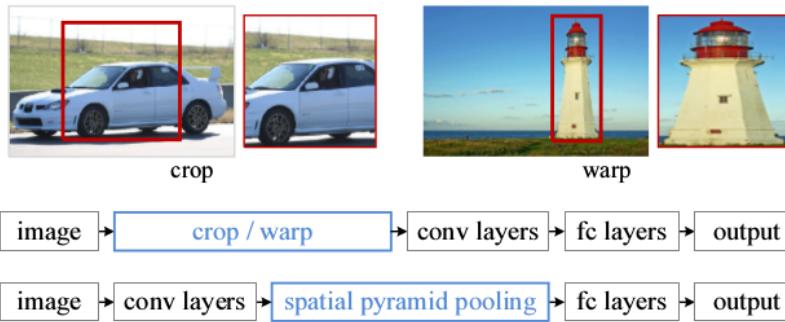


Figure 1: Top: cropping or warping to fit a fixed size. Middle: a conventional CNN. Bottom: our spatial pyramid pooling network structure.

**Fig 11.2.** Pyramid Pooling 在网络结构中的示意图，与传统 CNN 结构的比较

即在 Convolution 部分的最后一层之后，加入了 Pyramid Layer，可以认为是一种 Information Aggregation。

本文在 Feature Map 上对识别器进行训练与测试。

基于最近新提出的一个 Fast proposal method of EdgeBoxes。

### 11.3.2 网络结构

#### Fisher Kernel

参考文献：[Fisher Kernel Wikipedia](#)

目的是，通过对两个 object 进行一系列的 Measurements 和一个 Statistical model，得到两个 Object 的相似性。

在分类中，可以通过最小化 New object 与 each known number of the given class 之间的 Fisher Kernel distance 来实现分类。

Fisher Kernel 借鉴了生成模型和判别模型：

- 生成模型可以处理任意长的数据
- 判别模型可以 have flexible criteria 和得到更好的结果

Fisher Kernel 基于 Fisher Score 来实现：

$$U_X = \nabla_{\theta} \log P(X|\theta)$$

其中， $\theta$  是模型参数， $\log P(X|\theta)$  是对数似然。

然后 Fisher Kernel 如下：

$$K(X_i, X_j) = U_{X_i}^T I^{-1} U_{X_j}$$

其中,  $I$  是 Fisher Information matrix, 比较复杂, 查 Wikipedia 吧。

一种应用是: 用在图像分类和 Retrieval 中, 由于 Bag of visual words 模型 suffer from sparsity 和 high dimensionality。而 Fisher Kernel 可以产生稠密、Compact 的 Representation。

### Convolutional Layers and Feature Maps

在强调一下, 对输入图像固定尺寸这一限制来自于全连接层的输入要求。基于传统的特征表示的, 会再利用其他技术进行表示的处理, 如上文提到的 Fisher Kernel 等。

### The Spatial Pyramid Pooling Layer

相比于传统的基于 BoW 的模型, Spatial pyramid pooling 还可以保留空间信息 by pooling in local spatial bins, These spatial bins have sizes proportional to the image size, so the number of bins is fixed regardless of the image size.(这句话有错误吧, 先说这些 Spatial bins 与输入图像大小成正比, 但又说 the number of bins 是固定的, 与图像尺寸无关。???)

图表示了提出的 Pyramid Pooling Layer 的示意图。在每一个 Spatial bin, 作者对每一个 filter 的 response 进行 Pool(MaxPooling)。而输出是  $kM$ , 其中,  $k$  是 filter 的数目,  $M$  是 the number of bins? 这里的 bin 是指 Pooling 后的结果么? 然后结果被输入到 Fully connected 层。

补充: 首先从图11.3可以出以及从名字上可以看出, 这个 Pooling 层是 Pyrmid 的, 也就是说有几个输出大小不同但是固定的 Pooling 的操作同时对 Conv5 的结果进行 Pooling, 这里的 M 就是所有不同输出大小的 Pooling 的结果的 Flatten 起来的, 然后输入到全连接层。所以才有了 Fast RCNN 中说的只用了一种 Scale 的 Pooling 的特殊的 Spatial Pyramid Pooling Layer。这里的 Spatial 是指在 Pooling 的过程中也保持了 Feature 的空间信息, 这本来就是 Pooling 操作的本质特征之一! 也就是既降维、又会保持空间位置信息!

作者证明, 在深度网络中, 多尺度对提高精度同样重要。

作者认为: The global pooling operation corresponds to the traditional BoW method.?

### Training

在这部分, 作者貌似提到了 Pyramid Pooling 是怎么进行的。得到最后一层的  $a \times a$  大小的 Feature Map 进行 Pooling 操作, 操作时, Sliding window Pooling 的 window size 是  $win = \lceil a/n \rceil$ , 然后 Pooling 的 stride 是  $\lfloor a/n \rfloor$ 。

基于 cuda-convnet 和 caffe 实现。

### 11.3.3 Object Detection Experiments

在分类的试验中, 实验结果表明, Multi-level Pooling 提高精度, Multi-size 也提高精度, Fully-image representation 提高精度。

Model Combination 是提高 CNN-based 分类器精度的一种重要手段。实验表明, 这种提升貌似来自于 Convolutional layer。如果组合两个具有一样结构的 Conv, 那么效果貌似并没有提升。

这个细节还是有必要说一下。

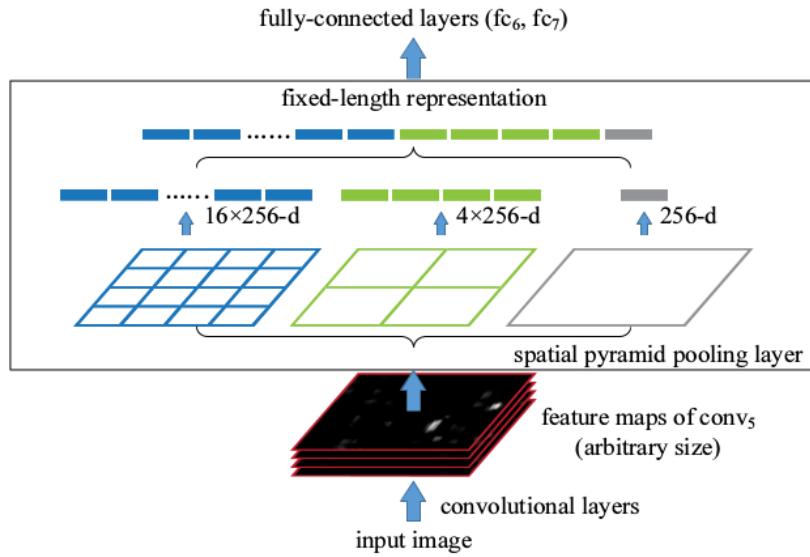


Figure 3: A network structure with a **spatial pyramid pooling layer**. Here 256 is the filter number of the conv<sub>5</sub> layer, and conv<sub>5</sub> is the last convolutional layer.

**Fig 11.3.** Pyramid Pooling 的计算过程示意图

RCNN 的工作流程是：首先用 Selective search 在输入图像上得到候选框，然后进行在每一个候选框输入到 CNN 进行特征提取，提取的特征送入 SVM 进行分类。

注意，因为与之前的方法相比，在目标识别中，本分的方法只对整幅图像进行一次卷积，然后在 Feature Map 进行选框操作 (Candidate window of feature maps)。那么这个在 Feature Map 中的候选框是怎么来的呢？

看文章附件的意思：

首先在输入图像中生成候选框，可以用相同的 Selective 方法等，在 Image Domain 生成一系列的候选框，然后把这个候选框的位置映射到 Feature Map 中，得到其在 Feature map 中的 Corner point。由于在得到 Feature Map 的过程中涉及很多的下采样过程，所以，在映射的过程中需要注意映射的准确性。本文用到的映射公式如下：

$$\begin{aligned} x' &= \lfloor x/S \rfloor + 1 && (\text{lefttopcorner}) \\ x' &= \lfloor x/S \rfloor - 1 && (\text{rightbottomcorner}) \end{aligned}$$

其中  $S$  是当前层的 Stride。上述过程的前提是，Padding 为  $\lfloor p/2 \rfloor$ 。 $p$  为 Pooling 的 size。

### 11.3.4 Conclusion

SPP is a flexible solution for handling different scales, sizes, and aspect ratios. The resulting SPP-net shows outstanding accuracy in classification/detection tasks and greatly accelerates DNN-based detection.

还有就是，一些经典的视觉算法在深度学习中，同样可以发挥重要的作用！

## 11.4 Fast RCNN

参考文献:

[2] 详解 ROI-Pooling 层的实现 -知乎

有几点需要注意的是:

- 引入了 ROI Pooling 的概念，同时每一个 Minibatch 的多个 ROI 来自同一幅图像，因此可以合并 Feature Extract 的过程，提高 Training 的速度！
- 多任务的 Loss Function，同时预测分类错误、Boundingbox 回归：

$$Loss = L_{cls}(p, u) + \lambda[u \geq 1]L_{loc}(t^u, v)$$

其中，p 为预测，u 为类标签。这里的  $L_{loc}$  采用的是 Smooth L1 loss。就是 YOLOv2 代码中的那样。

实验表明，多任务训练可以提高分类的精度。

- ROI Pooling Layer 的后向传播，这个可以看一下 DLTips 那一章节的对 Pooling 层的后向传播的简单说明，是 Max Pooling 的情况。
- 由于 ROI 很多，所以全连接层耗时非常大，所以作者用 SVD 对一层全连接层进行了分解，这样，本来一层的全连接层被分解成两个全连接层实现，且两个全连接层之间没有非线性激活函数。个人感觉类似于 Bottleneck 的结构。
- 使用 Average Recall 来判断 Region Proposal 的效果

### 11.4.1 ROI-Pooling 层的实现 -知乎

Fast-RCNN 的大体步骤：

- 输入：images 的 shape 是 (N, W, H, C) 这是 Tensorflow 的输入。Rois 的 shape 是 (nums\_rois, 4) 这里注意了，ROI 也是输入的参数，那么这些 ROI 怎么来的呢，可能是 Selective Search 的结果。
- VGG 网络对输入的 Image 部分提取特征，走到最后一层卷积层
- 最后一个卷积层的输出作为 ROI Pooling 层的输入
- ROI Pooling 的输出在做分类和回归

### ROI Pooling 层

输入 Shape: (N, W/16, H/16, channels)，因为是使用 VGG16 实现，经过 4 次 2\*2 的 Max Pooling，所以降维 16 倍，这里的 Pooling 默认 Strides 为 2。

输出 Shape: (num\_rois, expected\_H, expected\_W, channels)，论文中提到如果使用 VGG16，则输出是 W=H=7。

由于输入的图像下采样了 16 倍，所以输入的 ROI 也会缩小 16 倍，包括索引、长宽等。

### 11.5 Faster RCNN

### 11.6 R FCN

### 11.7 Mask RCNN

见 Semantic SLAM 那一章。

### 11.8 YOLO

### 11.9 YOLO v2

### 11.10 YOLO v3

### 11.11 SSD

### 11.12 DSSD

### 11.13 Retina Net (Focal Loss)

### 11.14 Feature Pyramid Network

在处理高层但分辨率小的特征图与底层但分辨率高的特征图的时候，使用 Nearest neighbor interpolation 的方法进行 Upsample。

#### 11.14.1 Nearest Neighbor Interpolation

参考：最近邻插值

这是一种简单的插值算法：不需要计算，在待求象素的四邻象素中，将距离待求象素最近的邻象素灰度赋给待求象素

设  $i+u, j+v$ ( $i, j$  为正整数， $u, v$  为大于零小于 1 的小数，下同) 为待求象素坐标，则待求象素灰度的值  $f(i+u, j+v)$

如果  $(i+u, j+v)$  落在 A 区，即  $u < 0.5, v < 0.5$ ，则将左上角象素的灰度值赋给待求象素，同理，落在 B 区则赋予右上角的象素灰度值，落在 C 区则赋予左下角象素的灰度值，落在 D 区则赋予右下角象素的灰度值。最邻近元法计算量较小，但可能会造成插值生成的图像灰度上的不连续，在灰度变化的地方可能出现明显的锯齿状。

### 11.15 ResNet

最需要明确的是：ResNet 现在基本上作为各种视觉算法的 Backbone 了!!! 要知道 Backbone 的牛逼。

然后需要注意以下两点：再由 ResNet18 到 ResNet50 的时候是把基本 block 扩展为 bottleneck，这个 bottleneck 就是输出 Channel 先变小在变大！第二点是：Identity 和 Non-Identity 两条路径的结果是通过相加实现的，分辨率不同的时候会先通过 Maxpooling 降维！

# References

- [1] Shuichi Asano, Tsutomu Maruyama, and Yoshiki Yamaguchi. Performance comparison of fpga, gpu and cpu in image processing. In *Field Programmable Logic and Applications, 2009. FPL 2009. International Conference on*, pages 126--131. IEEE, 2009.
- [2] Xinlei Chen, Li-Jia Li, Li Fei-Fei, and Abhinav Gupta. Iterative visual reasoning beyond convolutions. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2018.
- [3] Andreas Eitel, Jost Tobias Springenberg, Luciano Spinello, Martin Riedmiller, and Wolfram Burgard. Multimodal deep learning for robust rgb-d object recognition. In *Intelligent Robots and Systems (IROS), 2015 IEEE/RSJ International Conference on*, pages 681--687. IEEE, 2015.
- [4] Pedro F Felzenszwalb and Daniel P Huttenlocher. Efficient Graph-Based Image Segmentation. *International Journal of Computer Vision*, 59(2):167--181, 2004.
- [5] Pedro F. Felzenszwalb and Daniel P. Huttenlocher. Efficient graph-based image segmentation. *International Journal of Computer Vision*, 59(2):167--181, 2004.
- [6] Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial nets. In Z. Ghahramani, M. Welling, C. Cortes, N. D. Lawrence, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 27*, pages 2672--2680. Curran Associates, Inc., 2014.
- [7] Saurabh Gupta, Ross Girshick, Pablo Arbeláez, and Jitendra Malik. Learning rich features from rgb-d images for object detection and segmentation. In *European Conference on Computer Vision*, pages 345--360. Springer, 2014.
- [8] K. He, G. Gkioxari, P. Dollár, and R. Girshick. Mask R-CNN. *ArXiv e-prints*, March 2017.
- [9] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Spatial pyramid pooling in deep convolutional networks for visual recognition. In *European conference on computer vision*, pages 346--361. Springer, 2014.
- [10] J. Hu, L. Shen, and G. Sun. Squeeze-and-Excitation Networks. *ArXiv e-prints*, September 2017.

- 
- [11] Ronghang Hu, Piotr Dollár, Kaiming He, Trevor Darrell, and Ross Girshick. Learning to segment every thing. *arXiv preprint arXiv:1711.10370*, 2017.
  - [12] T.-W. Hui, X. Tang, and C. Change Loy. LiteFlowNet: A Lightweight Convolutional Neural Network for Optical Flow Estimation. *ArXiv e-prints*, May 2018.
  - [13] Vladimir I Iglovikov, Selim Seferbekov, Alexander V Buslaev, and Alexey Shvets. Ternausnetv2: Fully convolutional network for instance segmentation. *arXiv preprint arXiv:1806.00844*, 2018.
  - [14] Chao Li, Yanjing Bi, Franck Marzani, and Fan Yang. Fast fpga prototyping for real-time image processing with very high-level synthesis. *Journal of Real-Time Image Processing*, pages 1–18, 2017.
  - [15] Shutao Li, Xudong Kang, and Jianwen Hu. Image fusion with guided filtering. *IEEE Transactions on Image Processing*, 22(7):2864–2875, 2013.
  - [16] Yi Li, Haozhi Qi, Jifeng Dai, Xiangyang Ji, and Yichen Wei. Fully convolutional instance-aware semantic segmentation. *arXiv preprint arXiv:1611.07709*, 2016.
  - [17] Shu Liu, Lu Qi, Haifang Qin, Jianping Shi, and Jiaya Jia. Path aggregation network for instance segmentation. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 8759–8768, 2018.
  - [18] J. Long, E. Shelhamer, and T. Darrell. Fully Convolutional Networks for Semantic Segmentation. *ArXiv e-prints*, November 2014.
  - [19] Roberto Cipolla Marvin T.T. Teichmann. Convolutional crfs for semantic segmentation. *arXiv:https://arxiv.org/pdf/1805.04777.pdf*, 2018.
  - [20] J. McCormac, A. Handa, A. Davison, and S. Leutenegger. SemanticFusion: Dense 3D Semantic Mapping with Convolutional Neural Networks. *ArXiv e-prints*, September 2016.
  - [21] R. Mehta and C. Ozturk. Object detection at 200 Frames Per Second. *ArXiv e-prints*, May 2018.
  - [22] V. Mnih, N. Heess, A. Graves, and K. Kavukcuoglu. Recurrent Models of Visual Attention. *ArXiv e-prints*, June 2014.
  - [23] Trung T. Pham, Markus Eich, Ian Reid, and Gordon Wyeth. Geometrically consistent plane extraction for dense indoor 3d maps segmentation. In *Ieee/rsj International Conference on Intelligent Robots and Systems*, pages 4199–4204, 2016.
  - [24] Xiaojuan Qi, Renjie Liao, Jiaya Jia, Sanja Fidler, and Raquel Urtasun. 3d graph neural networks for rgbd semantic segmentation. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 5199–5208, 2017.
  - [25] Mengye Ren and Richard S Zemel. End-to-end instance segmentation with recurrent attention. *arXiv preprint arXiv:1605.09410*, 2017.

## REFERENCES

---

- [26] O. Ronneberger, P. Fischer, and T. Brox. U-Net: Convolutional Networks for Biomedical Image Segmentation. *ArXiv e-prints*, May 2015.
- [27] Martin Rünz and Lourdes Agapito. Co-fusion: Real-time segmentation, tracking and fusion of multiple objects. In *Robotics and Automation (ICRA), 2017 IEEE International Conference on*, pages 4471--4478. IEEE, 2017.
- [28] J. L. Schönberger, M. Pollefeys, A. Geiger, and T. Sattler. Semantic Visual Localization. *ArXiv e-prints*, December 2017.
- [29] Raluca Scona, Mariano Jaimez, Yvan R Petillot, Maurice Fallon, and Daniel Cremers. Staticfusion: Background reconstruction for dense rgb-d slam in dynamic environments. Institute of Electrical and Electronics Engineers, 2018.
- [30] Jan C. van Gemert Silvia L. Pintea, Yue Liu. Recurrent knowledge distillation. *ArXiv e-prints*, May 2018.
- [31] N. Sünderhauf, T. T. Pham, Y. Latif, M. Milford, and I. Reid. Meaningful Maps With Object-Oriented Semantic Mapping. *ArXiv e-prints*, September 2016.
- [32] Marvin TT Teichmann and Roberto Cipolla. Convolutional crfs for semantic segmentation. *arXiv preprint arXiv:1805.04777*, 2018.
- [33] Alexander Toet and Maarten A Hogervorst. Multiscale image fusion through guided filtering. In *SPIE Security+ Defence*, pages 99970J--99970J. International Society for Optics and Photonics, 2016.
- [34] J. R. R. Uijlings, K. E. A. van de Sande, T. Gevers, and A. W. M. Smeulders. Selective search for object recognition. *International Journal of Computer Vision*, 104(2):154--171, Sep 2013.
- [35] F. Visin, M. Ciccone, A. Romero, K. Kastner, K. Cho, Y. Bengio, M. Matteucci, and A. Courville. ReSeg: A Recurrent Neural Network-based Model for Semantic Segmentation. *ArXiv e-prints*, November 2015.
- [36] P. Wang, R. Yang, B. Cao, W. Xu, and Y. Lin. DeLS-3D: Deep Localization and Segmentation with a 3D Semantic Map. *ArXiv e-prints*, May 2018.
- [37] R. Wang, J.-M. Frahm, and S. M. Pizer. Recurrent Neural Network for Learning DenseDepth and Ego-Motion from Video. *ArXiv e-prints*, May 2018.
- [38] Y. Xiang and D. Fox. DA-RNN: Semantic Mapping with Data Associated Recurrent Neural Networks. *ArXiv e-prints*, March 2017.
- [39] D. Xu, W. Ouyang, X. Wang, and N. Sebe. PAD-Net: Multi-Tasks Guided Prediction-and-Distillation Network for Simultaneous Depth Estimation and Scene Parsing. *ArXiv e-prints*, May 2018.
- [40] Fisher Yu and Vladlen Koltun. Multi-scale context aggregation by dilated convolutions. In *ICLR*, 2016.

# **Index**

SAD, 8