

# Papers I have read

宋明辉

June 1, 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[23] . . . . .	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 [26] . . . . .	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 常用数据集 . . . . .	70
5.20.1 KITTI . . . . .	70
5.20.2 官网资源介绍 . . . . .	70
5.20.3 详述 . . . . .	71
5.20.4 Cityscape . . . . .	72
5.20.5 TUM . . . . .	72
<b>6 Open Set Recognition</b> . . . . .	<b>73</b>
6.1 GAN . . . . .	73
6.1.1 GAN 原理笔记 . . . . .	73
6.2 从头开始 GAN . . . . .	75
6.2.1 定义 . . . . .	76
6.2.2 DCGAN: Deep Convolution GAN . . . . .	76
6.2.3 CGAN: Conditional Generative Adversarial Nets . . . . .	77
6.2.4 InfoGAN . . . . .	77
6.3 Generative Adversarial Nets . . . . .	78
6.4 Towards Open Set Deep Networks . . . . .	78
6.4.1 Introduction & Related Works . . . . .	78
6.5 Probability Models for Open Set Recognition . . . . .	78
6.5.1 背景及相关工作 . . . . .	79
<b>7 MXNet</b> . . . . .	<b>81</b>
7.1 Optimizing Memory Consumption in DL . . . . .	81
7.1.1 Computation Graph . . . . .	81
7.1.2 What Can be Optimized? . . . . .	83
7.1.3 Memory Allocation Algorithm . . . . .	83
7.1.4 Static vs. Dynamic Allocation . . . . .	85
7.1.5 Memory Allocation for Parallel Operations . . . . .	85
7.1.6 How Much Can we Save ? . . . . .	86
7.1.7 References . . . . .	87
7.2 Deep Learning Programming Style . . . . .	87
7.2.1 Symbolic vs. Imperative Program . . . . .	87
7.2.2 Imperative Programs Tend to be More Flexible . . . . .	87
7.2.3 Symbolic Programs Tend to be More Efficient . . . . .	87
7.2.4 Case Study: Backprop and AutoDiff . . . . .	88
7.2.5 Model Checkpoint . . . . .	89
7.2.6 Big vs. Small Operations . . . . .	89
7.2.7 Mix The Approaches . . . . .	90
7.3 Dependency Engine for Deep Learning . . . . .	90
7.3.1 Problems in Dependency Scheduling . . . . .	90
7.3.2 Implementing the Generic Dependency Engine . . . . .	92
7.3.3 Discussion . . . . .	92

## CONTENTS

---

7.4	Designing Efficient Data Loaders for DL . . . . .	92
7.4.1	Design Insight . . . . .	93
7.4.2	Data Format . . . . .	93
7.4.3	Data Loading and Preprocessing . . . . .	93
7.4.4	MXNet IO Python Interface . . . . .	95
7.5	Except Handling in MXNet . . . . .	96
7.6	MXNet-Gluon 创建模型 . . . . .	97
7.6.1	模型构造 . . . . .	97
7.6.2	自定义层 . . . . .	98
7.6.3	实际例子 . . . . .	99
7.7	MXNet 中的 Deconvolution . . . . .	100
7.8	MXNet 读取训练数据 . . . . .	101
7.8.1	使用 mx.io 读取数据 . . . . .	101
7.8.2	常用的系统提供的接口函数 . . . . .	103
7.8.3	使用 mx.image 读取数据 . . . . .	104
7.8.4	使用 Gluon 接口读取数据 . . . . .	104
7.9	Gluon 中的数据结构 . . . . .	106
<b>8</b>	<b>Tips in DL</b> . . . . .	<b>107</b>
8.1	Enlarge the FOV . . . . .	107
8.2	Upsampling . . . . .	107
8.3	Multiscale Ability . . . . .	107
8.4	Dilated Convolution . . . . .	108
8.5	Deconvolutional Network . . . . .	109
8.5.1	Convolutional Spare Coding . . . . .	109
8.5.2	CNN 可视化 . . . . .	109
8.5.3	Upsampling . . . . .	109
8.5.4	补充 . . . . .	109
8.5.5	Deconvolution 与 Upsample 的区别 . . . . .	110
8.6	Dilated Network 与 Deconv Network 之间的区别 . . . . .	112
8.7	目标检测中的 mAP 的含义 . . . . .	112
8.8	统计学习方法 . . . . .	113
8.9	Distillation Module . . . . .	113
8.9.1	Knowledge Distillation . . . . .	114
8.9.2	Recurrent Knowledge Distillation [18] . . . . .	114
8.10	光流估计中的 Average end-point error . . . . .	114
8.11	CNN 中的卷及方式汇总 . . . . .	114
8.11.1	Inception . . . . .	114
8.11.2	空洞卷积, Dilation . . . . .	115
8.11.3	深度可分离卷积, Depthwise Separable Convolution . . . . .	116
8.11.4	可变性卷积 . . . . .	117
8.11.5	特征重标定卷积 . . . . .	119
8.11.6	小结 - 比较 . . . . .	120
8.12	全连接与卷积的异同 . . . . .	120
8.13	Pooling . . . . .	121
8.14	Local Response Normalization . . . . .	122

---

8.15 CNN 中感受野的计算 . . . . .	122
8.16 神经网络中的初始化 . . . . .	126
8.16.1 Xavier . . . . .	127
8.17 计算图的后向传播计算 . . . . .	128
8.18 神经网络中的 Attention 机制 . . . . .	129
8.18.1 Recurrent Models of Visual Attention . . . . .	129
8.19 小型网络 . . . . .	131
8.20 Deep Reinforcement Learning . . . . .	131
8.21 为什么 Python 中要继承 object 类 . . . . .	131
8.22 1*1 卷积 . . . . .	132
8.23 待续 . . . . .	133
<b>9 Image Processing . . . . .</b>	<b>135</b>
9.1 Feature Extraction . . . . .	135
9.1.1 SIFT . . . . .	135
<b>10 Feature Extraction . . . . .</b>	<b>137</b>
10.1 Selective Search . . . . .	137
10.1.1 Efficient Graph-Based Image Segmentation . . . . .	137
10.1.2 Selective Search . . . . .	138
10.2 Region CNN . . . . .	138
10.2.1 概述 . . . . .	138
10.3 SPP Net . . . . .	138
10.3.1 背景与相关工作 . . . . .	138
10.3.2 网络结构 . . . . .	139
10.3.3 Object Detection Experiments . . . . .	141
10.3.4 Conclusion . . . . .	141
10.4 Fast RCNN . . . . .	142
10.5 Faster RCNN . . . . .	142
10.6 R FCN . . . . .	142
10.7 FPN . . . . .	142
10.8 Mask RCNN . . . . .	142
10.9 YOLO . . . . .	142
10.10 YOLO v2 . . . . .	142
10.11 YOLO v3 . . . . .	142
10.12 SSD . . . . .	142
10.13 DSSD . . . . .	142
10.14 Retina Net (Focal Loss) . . . . .	142

# 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	KITTI 官网截图 . . . . .	70
5.22	早期的数据文件组织格式 . . . . .	71

6.1 GAN 训练过程 . . . . .	74
6.2 CGAN 示意图, 在 G、N 网络中新增了数据 y . . . . .	77
7.1 The implicitly & explicitly back-propagation on Graph . . . . .	81
7.2 Dependencies can be found quickly. . . . .	82
7.3 Different backward path from forward path. . . . .	82
7.4 Standard Memory sharing between B & the result of E. . . . .	83
7.5 Standard Memory sharing between B & the result of E. . . . .	84
7.6 Standard Memory sharing between B & the result of E. . . . .	84
7.7 Standard Memory sharing between B & the result of E. . . . .	85
7.8 Color the longest paths in the Graph. . . . .	86
7.9 Operation Folding 示意图。 . . . . .	88
7.10 第一步, 给变量分配 tag . . . . .	91
7.11 把相应的 Function Closure push 进依赖分析引擎 . . . . .	91
7.12 一个具体的例子 . . . . .	92
7.13 Binary recordIO 数据结构 . . . . .	94
7.14 Binary recordIO 的一个例子 . . . . .	94
7.15 并行预处理例子 . . . . .	95
7.16 数据预取的示意图, 借助 Buffer 来实现 . . . . .	95
8.1 Dilated Convolution 示意图 . . . . .	108
8.2 Dilated Convolution 在 WaveNet 中的应用示意图 . . . . .	108
8.3 Transpose Convolution 过程示意图 . . . . .	110
8.4 一个例子, 用于卷积操作说明 . . . . .	111
8.5 卷积操作时的矩阵形式 . . . . .	112
8.6 三种不同的 Distillation Module . . . . .	114
8.7 Inception 卷积结构 . . . . .	115
8.8 Inception 结构与 Depthwise Separable Convolution 结构对比 . . . . .	116
8.9 Channel 分组的极限版本 . . . . .	117
8.10 Deformable Convolution 示意图 . . . . .	118
8.11 Deformable Convolution 的实现示意图 . . . . .	118
8.12 Squeeze-and-Excitation Block 示意图 . . . . .	119
8.13 SENet 与 Inception 以及 ResNet 结合的示意图 . . . . .	120
8.14 不同卷积策略的比较 . . . . .	121
8.15 Unpooling 示意图 . . . . .	121
8.16 文章的总结与说明 . . . . .	125
8.17 二维图像像素编号示意图 . . . . .	125
8.18 BP 计算过程示意图 . . . . .	129
8.19 Attention 模型示意图 . . . . .	130
8.20 1*1 Convolutoin 的作用示意图 . . . . .	133
10.1 RCNN 整体思想 . . . . .	138
10.2 Pyramid Pooling 在网络结构中的示意图, 与传统 CNN 结构的比较 . . . . .	139
10.3 Pyramid Pooling 的计算过程示意图 . . . . .	140

**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

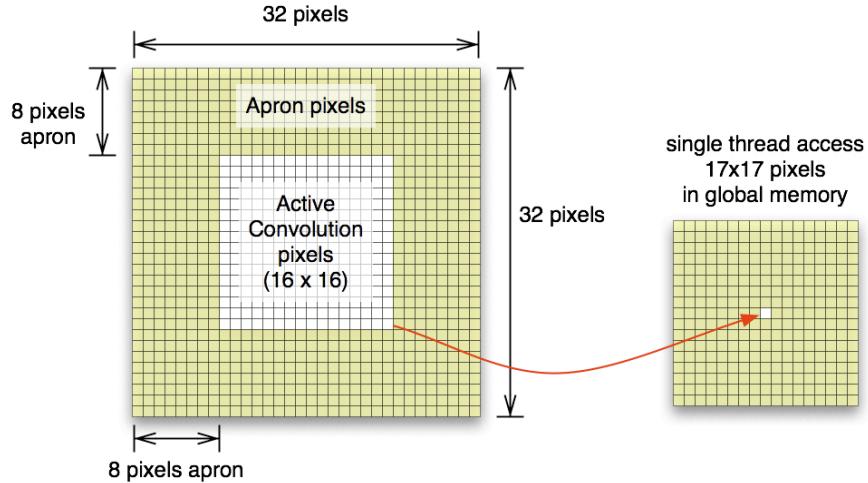
From the idea of convolution filter itself, the most naive approach is to use global memory to send data to device and each thread accesses this to compute convolution kernel. Our convolution kernel size is radius 8 (total  $17 \times 17$  multiplication for single pixel value). In image border area, 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;
```

}

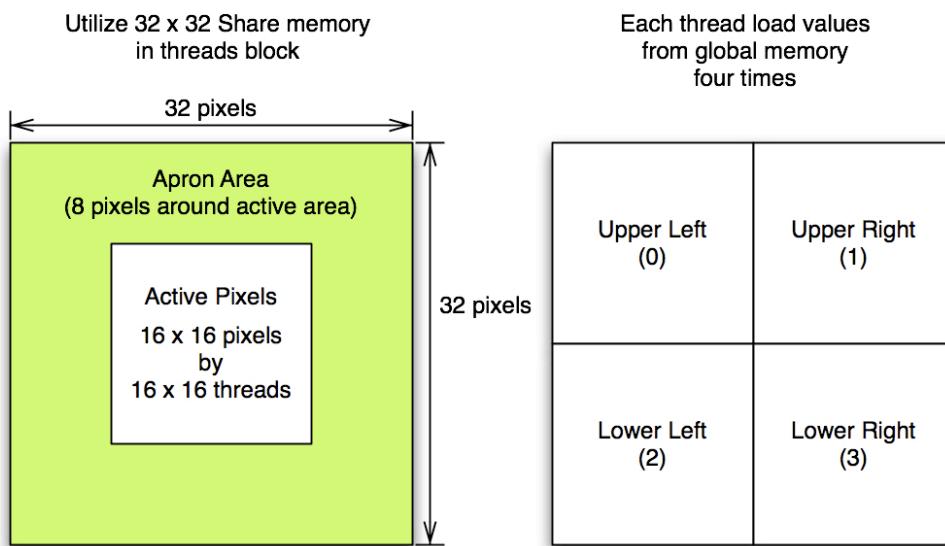
For  $396 \times 396$  input image, the time is 1.6ms. When the input filter is stored in constant memory or specified by 'restrict', the time is 1.7 or 1.8 ms.

### 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.

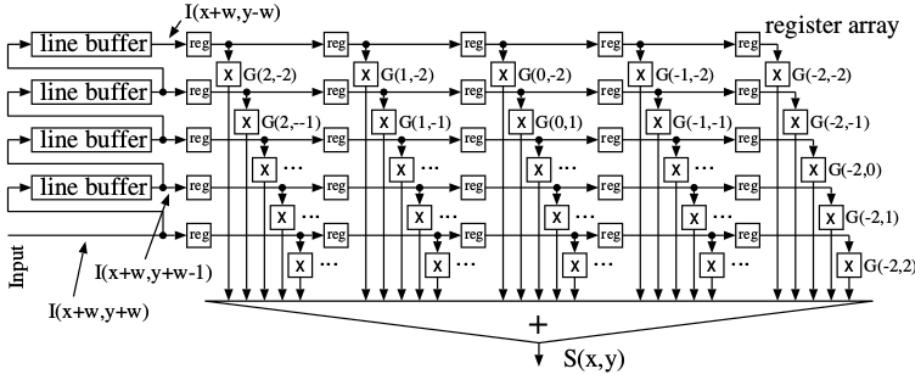
#### GPU Analysis

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.

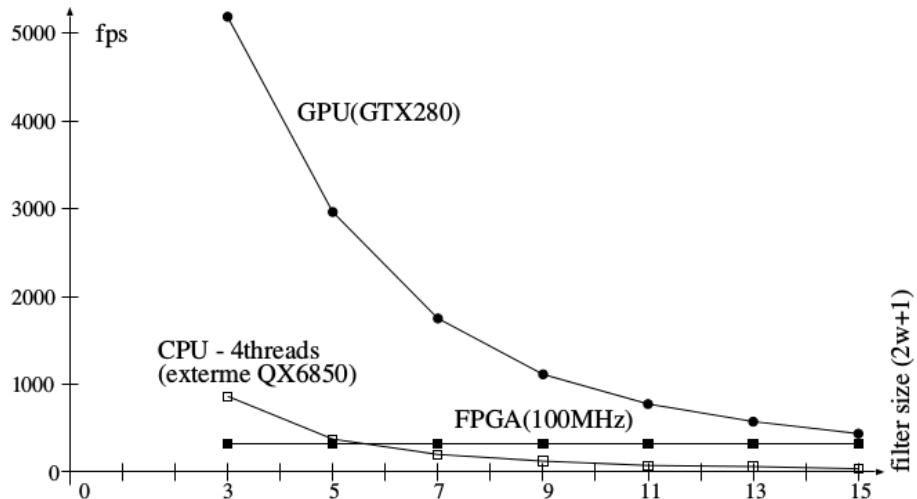
The time to download images from main memory is not included. CPU has four cores, FPGA is fixed to 100MHz. Fig is the performance of two-dimension filters.

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

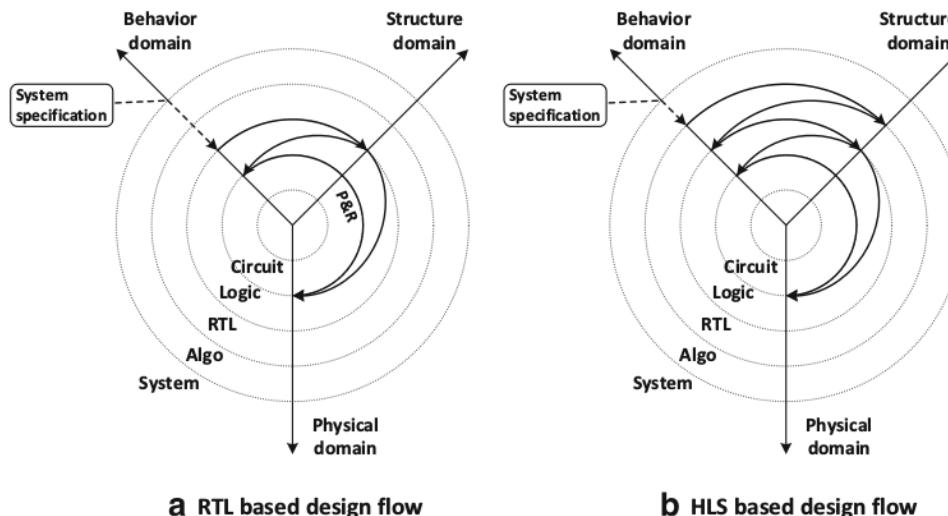
very high-level synthesis method which allows fast prototyping and verifying the FPGA-based image processing designs in the MATLAB environment. We build a heterogeneous development flow by using currently available tool kits for verifying the proposed approach and evaluated it within two real-life applications. 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[20].

### 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.

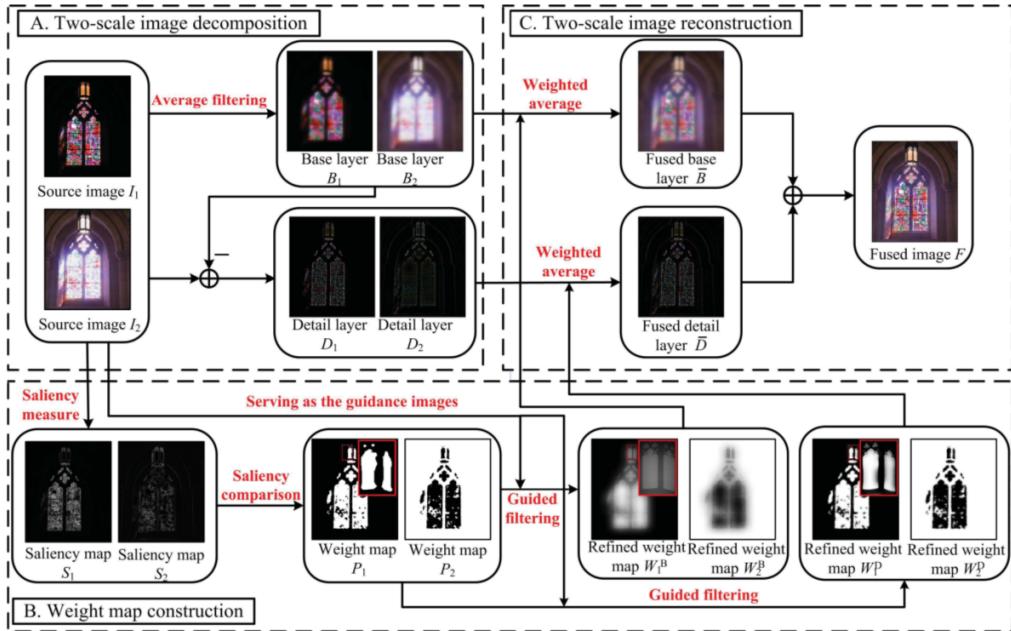
$$O_i = \bar{a}_i I_i + \bar{b}_i$$

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 [9].

### 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[23]

#### 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。

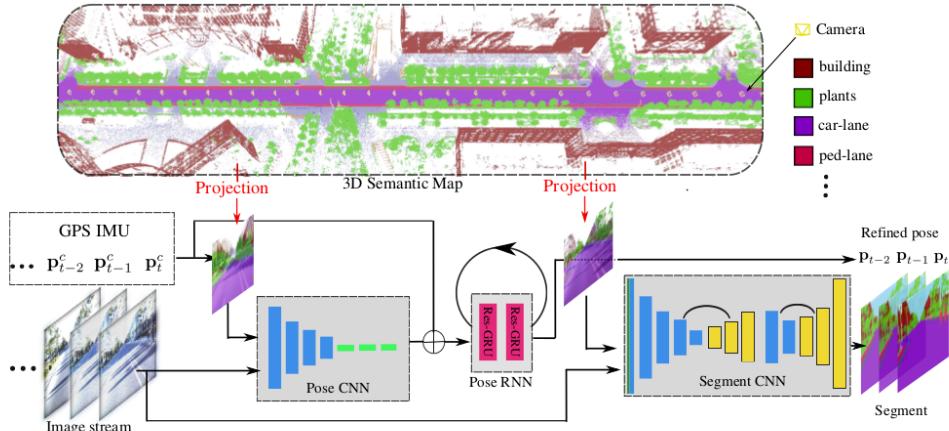


Figure 1: System overview. The black arrows show the testing process, and red arrows indicate the rendering (projection) operation in training and inference. The yellow frustum shows the location of cameras inside the 3D map. The input of our system contains a sequence of images and corresponding GPS/IMU 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[23]

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次的 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。

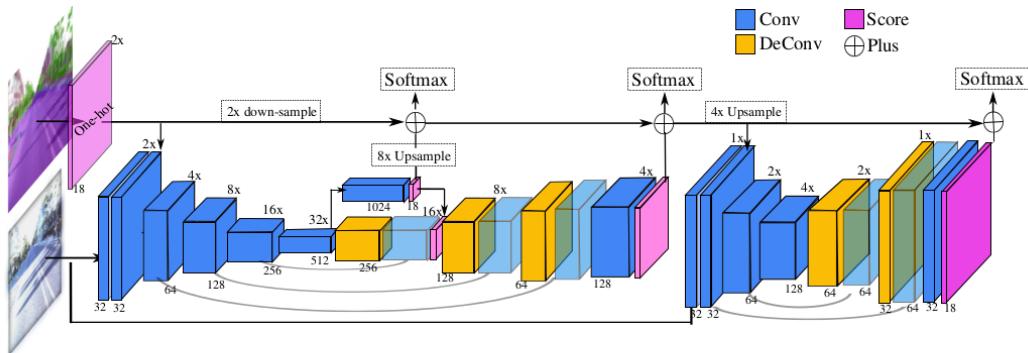


Figure 4: Architecture of the segment CNN with rendered label map as a segmentation priori. At bottom of each convolutional block, we show the filter size, and at top we mark the downsample rates of each block w.r.t. the input image size. The softmax' text box indicates the places a loss is calculated. 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 [26]**

---

- 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 [26]**

### **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: [24]

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:

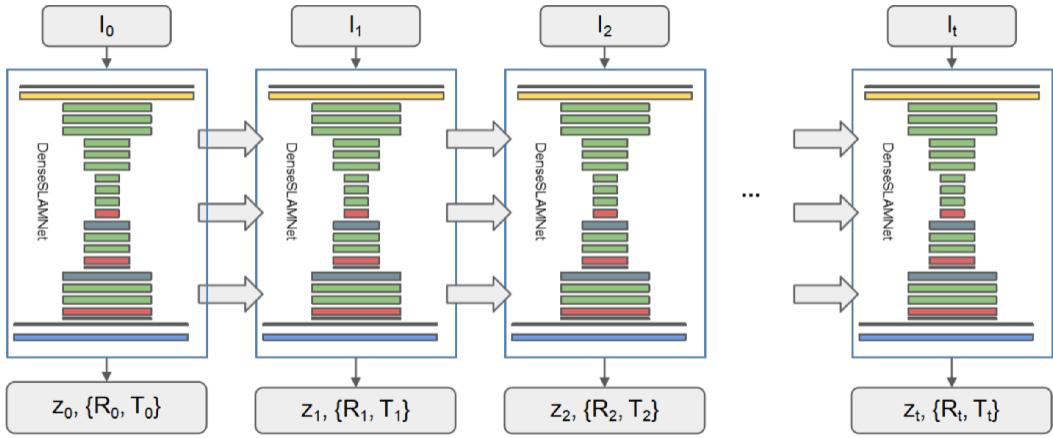
- Depth maps
- point clouds and voxels

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：

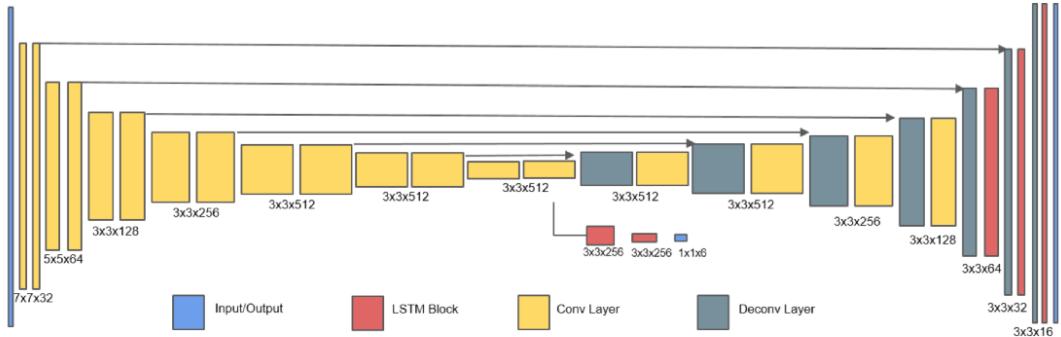


Fig. 4: (Best viewed in color) Our network architecture at a single time step. We use the DispNet architecture. The width and height of each rectangular block indicates the size and the number of the feature map at that layer. Each increase and decrease of size represents a change factor of 2. The first 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。

## 5.4 DA-RNN

---

**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: [25]

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 [26] 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 [26] 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：[12]

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。

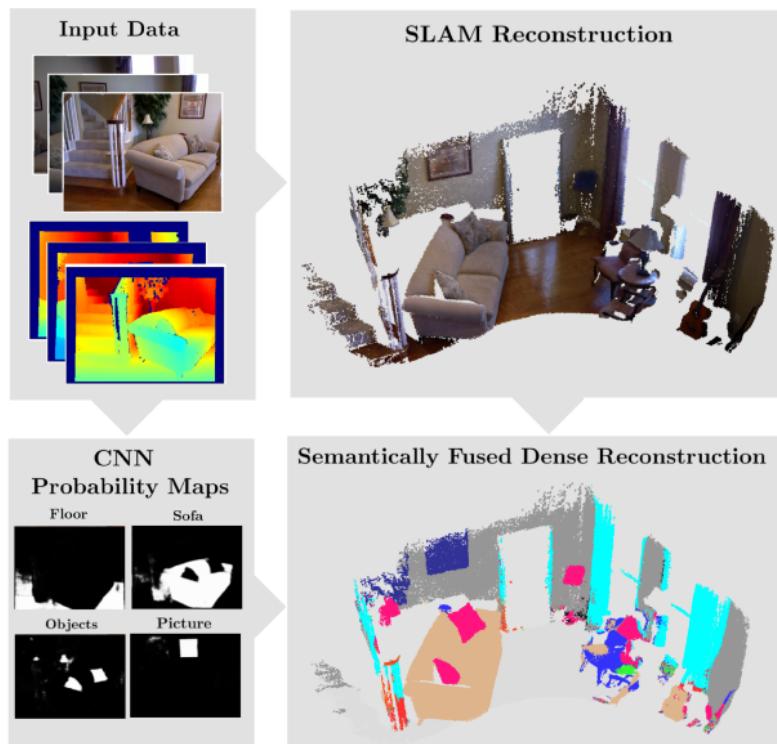
### 5.5.2 Method

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: [19]

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。

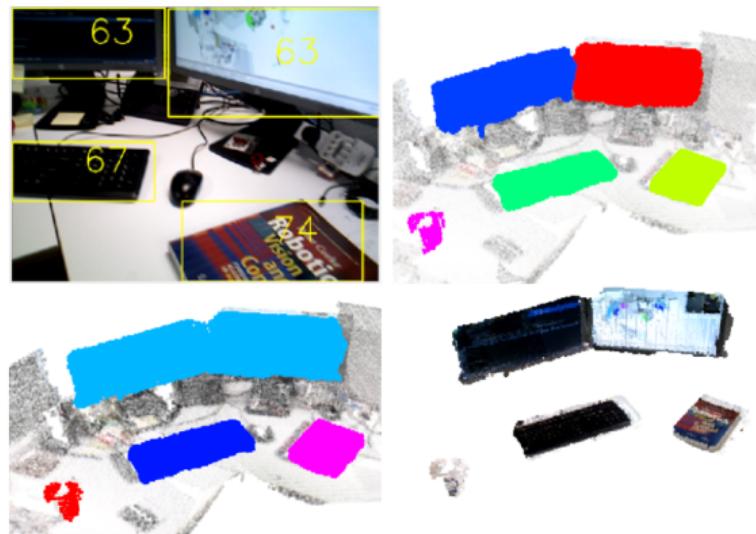
### 3D Segmentation

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 [3][15] 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。

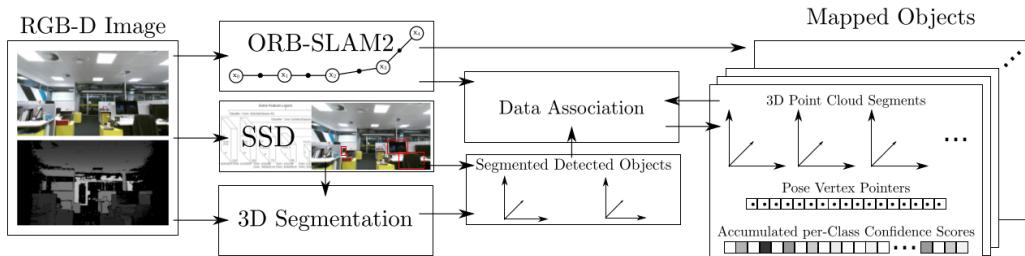
在第二步中，如果多余 50 % 的 Points 的都距离小于 2cm 的话，就说明这个检测到的 Object 已经存在了。

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: [7], 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 [7]。

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:

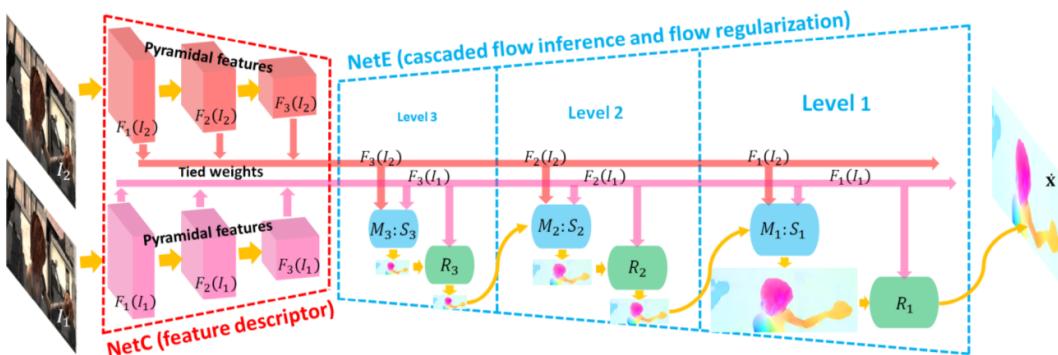


Figure 2: The network structure of LiteFlowNet. For the ease of representation, only a 3-level design is shown. Given an image pair ( $I_1$  and  $I_2$ ), NetC generates two pyramids of high-level features ( $\mathcal{F}_k(I_1)$  in pink and  $\mathcal{F}_k(I_2)$  in red,  $k \in [1, 3]$ ). NetE yields 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。

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

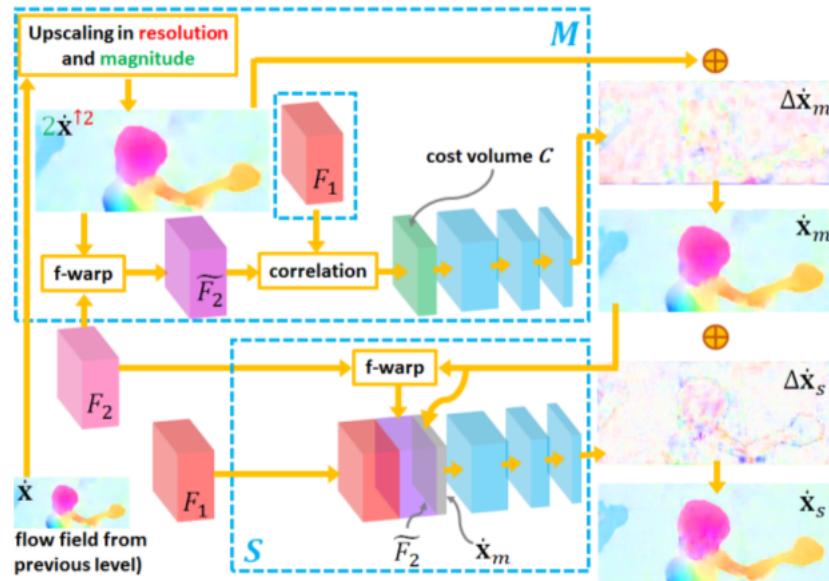


Figure 3: A cascaded flow inference module  $M:S$  in NetE. It consists of a descriptor matching unit  $M$  and a sub-pixel refinement unit  $S$ . In  $M$ , f-warp transforms high-level feature  $\mathcal{F}_2$  to  $\widetilde{\mathcal{F}}_2$  via upsampled flow field  $2\dot{x}^{t2}$  estimated at 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。

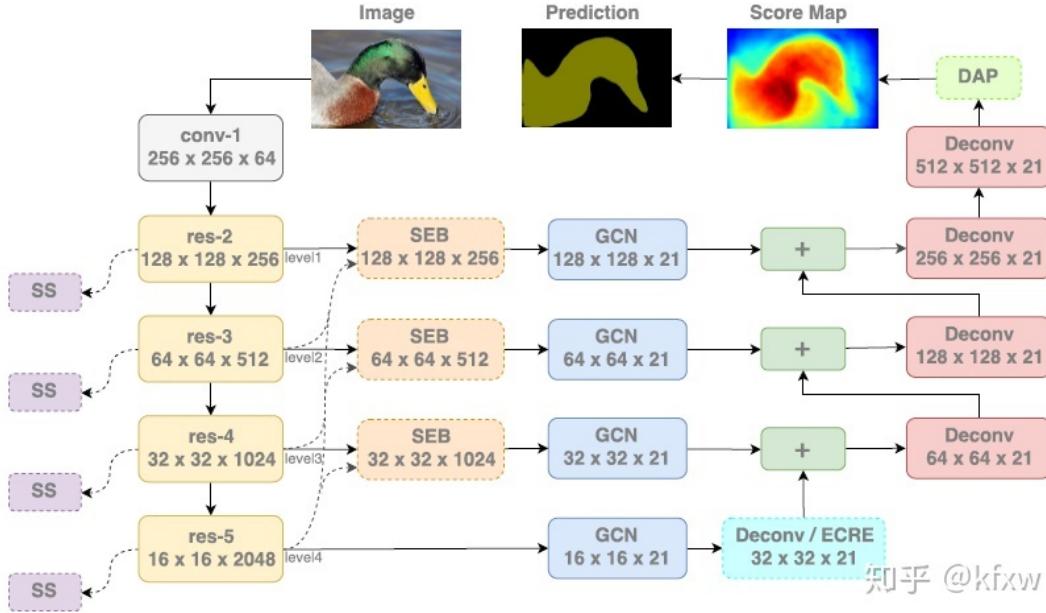


Fig 5.10. ExFusion 的实现框图

### 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)

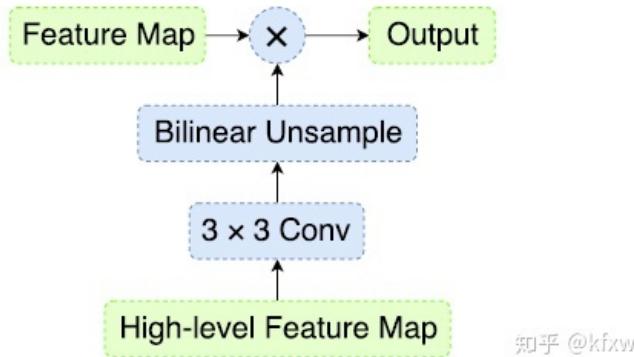
其做法是将高层特征上采样后，与低层特征逐像素相乘，用在 GCN 之前。该部分能带来大约 0.7 个点的提升。

### 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 [26] 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。

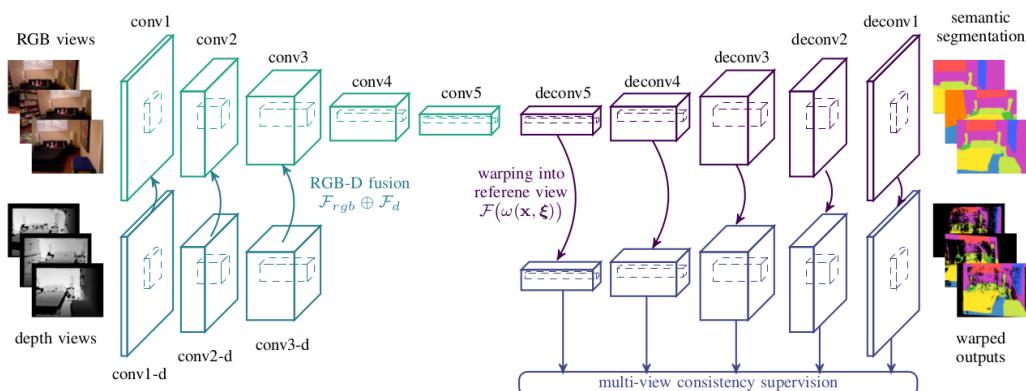
存在用 3D CNN 来处理系数的 Octree 数据结构，得到对 voxel 的语义分割。

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

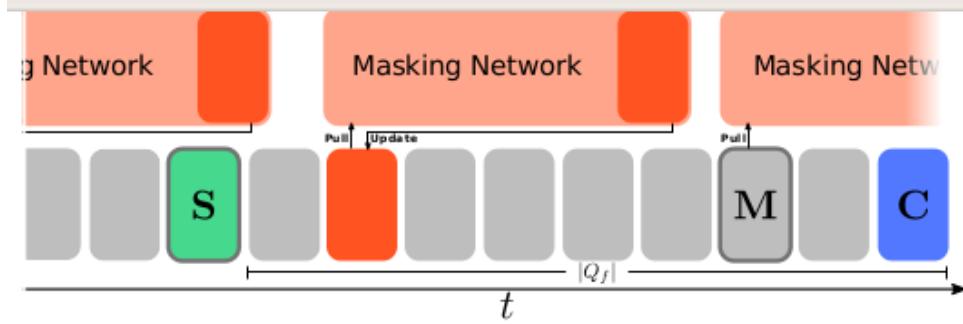
本文采用 ElasticFusion 类似的 3D Model. 其中，Model 用 Surfel 来表示，Surfel 具体包含以下内容： $\mathcal{M}_m^s \in \{\mathbf{p} \in \mathbb{R}^3, \mathbf{n} \in \mathbb{R}^3, \mathbf{c} \in \mathbb{N}^3, \mathbf{w} \in \mathbb{R}, \mathbf{r} \in \mathbb{R}, \mathbf{t} \in \mathbb{R}^2\}, \forall s < |\mathcal{M}_m|$ ，分别表示：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：

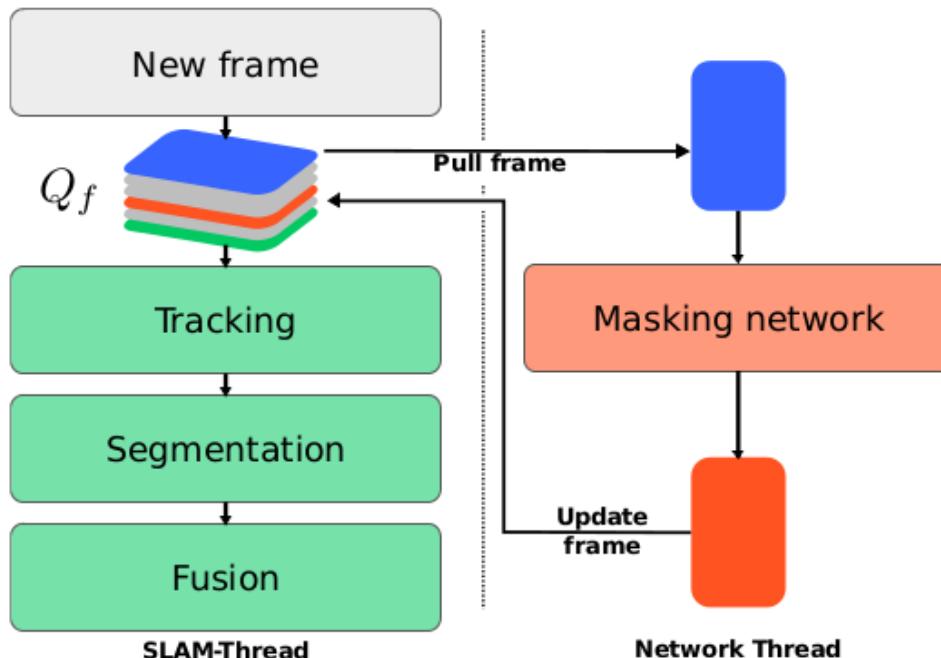
- Geometry Error

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

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?

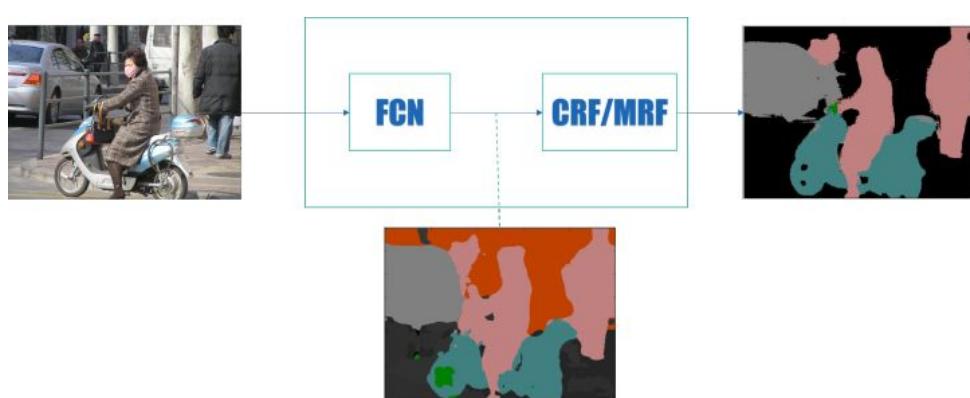
## 5.13 CNN-SLAM:

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：

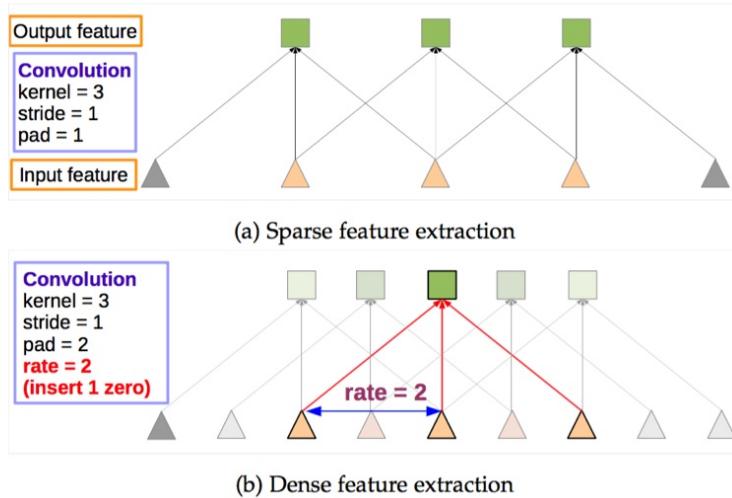


Fig 5.15. Atrous Convolution 示意图

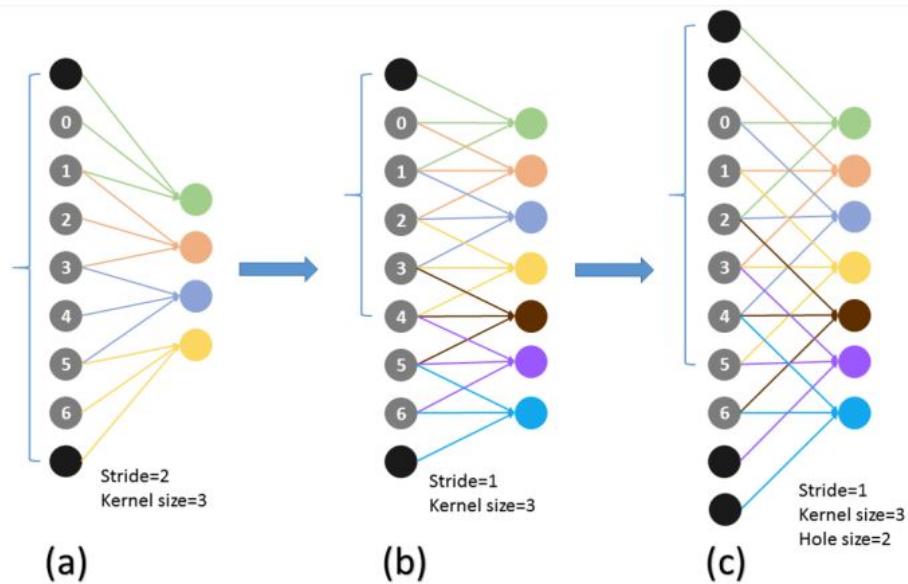


Fig 5.16. Atrous Convolution 中的感受野变化示意图

## 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

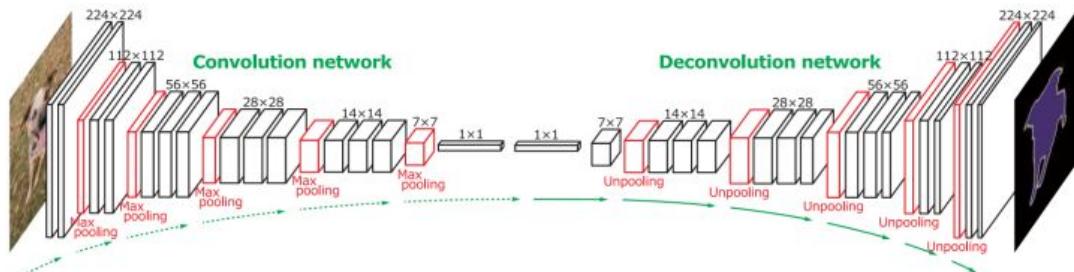


Figure 2. Overall architecture of the proposed network. On top of the convolution network based on VGG 16-layer net, we put a multi-layer deconvolution network to generate the accurate segmentation map of an input proposal. Given a feature representation obtained from the convolution network, dense pixel-wise 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: [22]

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：

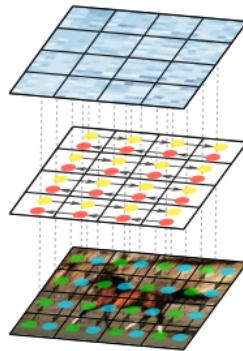


Figure 1. A ReNet layer. The blue and green dots on the input image/feature map represent the steps of  $f^\downarrow$  and  $f^\uparrow$  respectively. On the concatenation of the resulting feature maps,  $f^\rightarrow$  (yellow dots) and  $f^\leftarrow$  (red dots) are subsequently swept. Their feature 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：[16]

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。

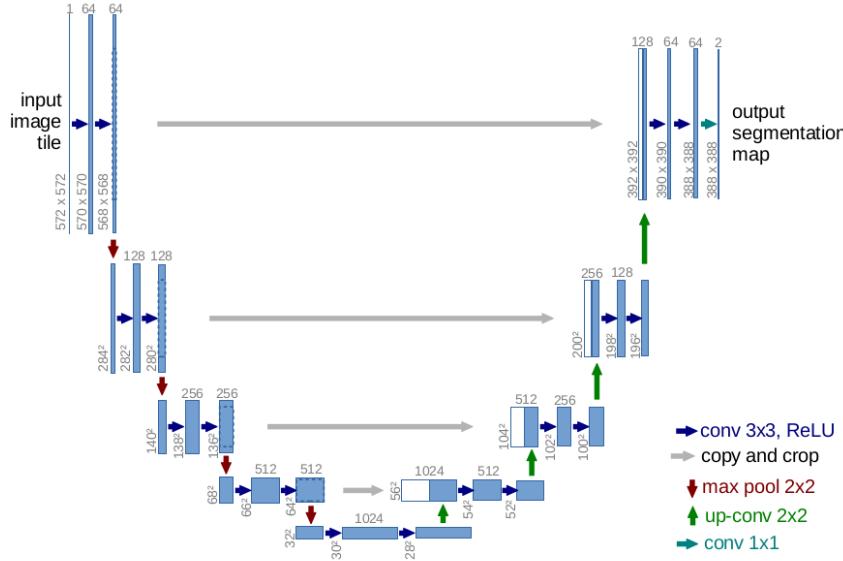
网络结构如图5.19所示。

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[10] 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：[17] 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：

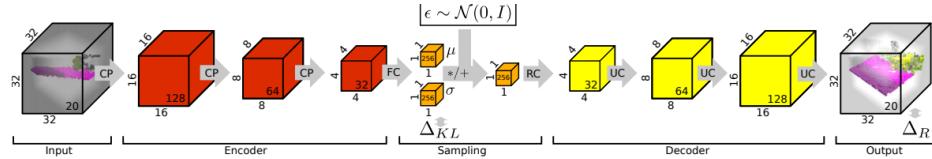


Figure 2: **Variational Encoder-Decoder Architecture.** Legend: CP = Convolution + Pooling, FC = Fully Connected, RC = Reshape + Convolution, UC = Upsampling + Convolution,  $\Delta_{KL}$  = KL Divergence wrt.  $\mathcal{N}(0, I)$ ,  $\Delta_R$  = Reconstruction Loss. The numbers at the bottom right 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 常用数据集

### 5.20.1 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。

### 5.20.2 官网资源介绍

最官网页面顶部，一共分了 13 栏，如图 5.21 所示。



Fig 5.21. KITTI 官网截图

## 5.20 常用数据集

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, raw 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。

### 5.20.3 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。

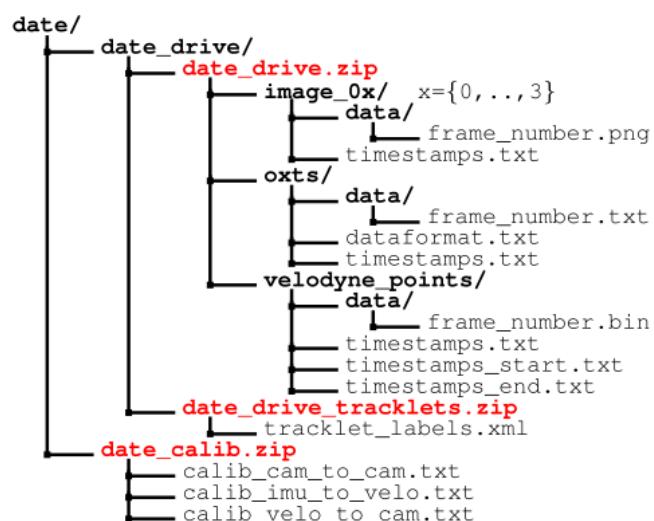


Fig. 4. **Structure of the provided Zip-Files** and their location within a global file structure that stores all KITTI sequences. Here, 'date' and 'drive' are placeholders, and 'image\_0x' refers to the 4 video camera streams.

**Fig 5.22.** 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。

### 5.20.4 Cityscape

数据集官网: [Cityscapes Datasets](#)

### 5.20.5 TUM

数据集官网: [TUM Datasets](#)

# 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。

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

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

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

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

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

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

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  $G$ 。

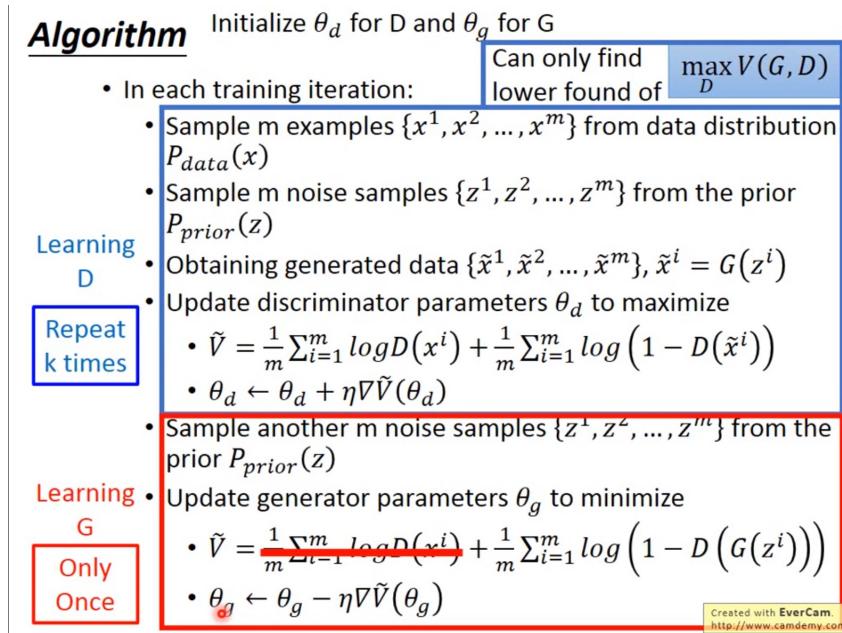


Fig 6.1. GAN 训练过程

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，表示两个分布之间的差异。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[5]:

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

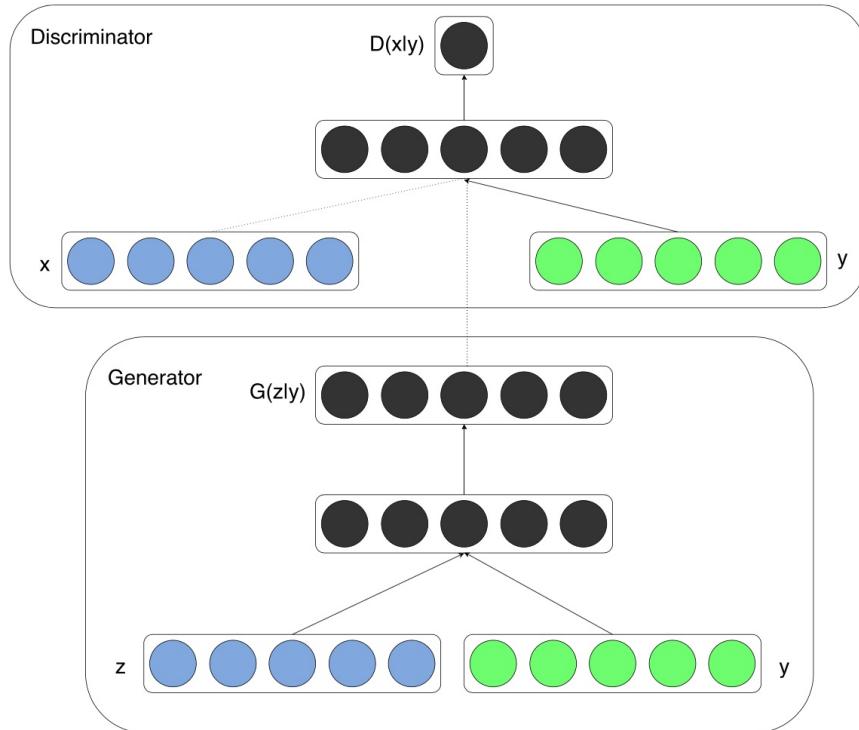
DCGAN 的几点改造：

- 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: [5]

**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。

## 6.5 Probability Models for Open Set 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



# Chapter 7

## MXNet

参考文献: [MXNet Architecture](#)

Time: 2018.05.18

### 7.1 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.

#### 7.1.1 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.

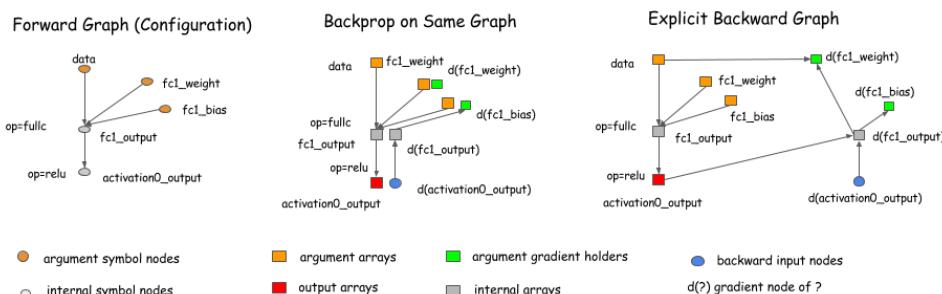


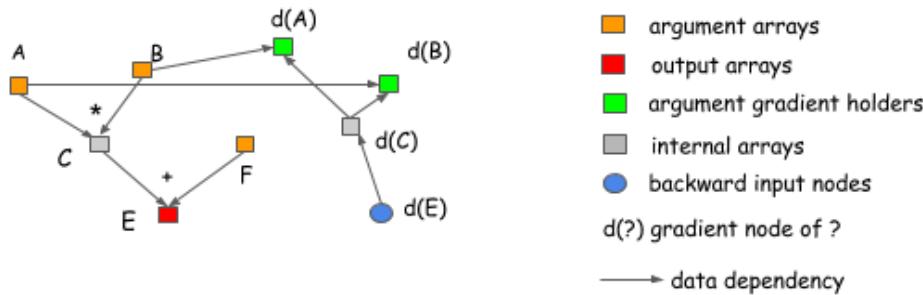
Fig 7.1. The implicitly & explicitly back-propagation on Graph

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, explicitly 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:

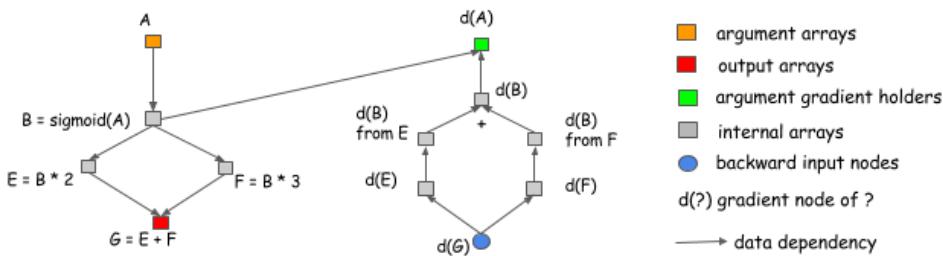
- The explicit backward path clearly describes the dependency between computations. Like the following case, where we want to get the gradient of **A** and **B**. As we can see clearly from the graph, the computation of the  $d(C)$  gradient doesn't depend on **F**. This means that we can free the memory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



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

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.

## 7.1 Optimizing Memory Consumption in DL

### 7.1.2 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**.

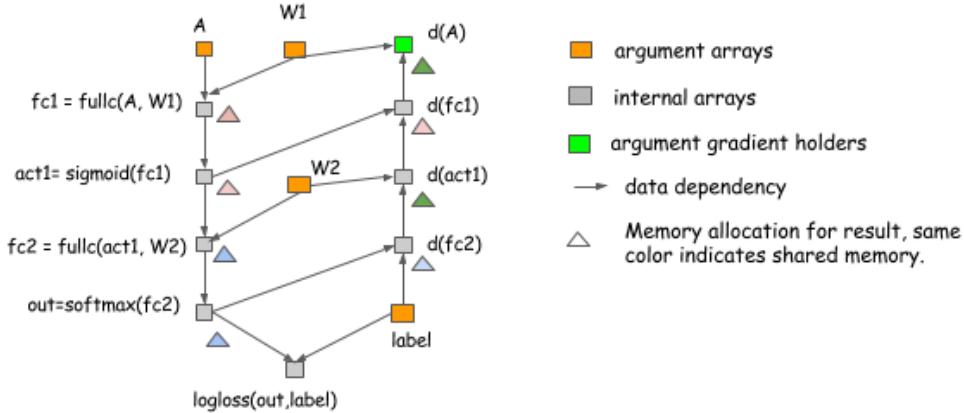
Memory sharing doesn't necessarily require the same data shape. Note that in the preceding example, the shapes of **B** and **E** can differ. To handle such a situation, we can allocate a memory region of size equal to the maximum of that required by **B** and **E** and share it between them.

### 7.1.3 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

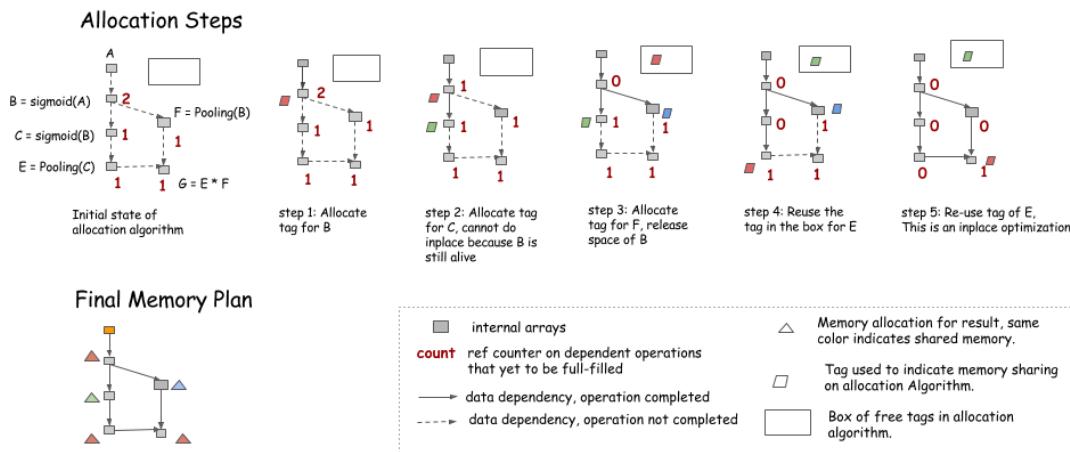
until the last time it is used. In the case of the multi-layer perceptron, the life time of  $fcl$  ends after  $act1$  get computed. See below figure:



**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.



**Fig 7.6.** Standard Memory sharing between **B** & the result of **E**.

- An in-place optimization can be performed when only the current operation depends on the source (i.e.  $\text{count} == 1$ ).
- Memory can be recycled into the box on the upper right corner when the  $\text{count}$  goes to 0.
- When we need new memory, we can either get it from the box or allocate a new one.

## 7.1 Optimizing Memory Consumption in DL

**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.

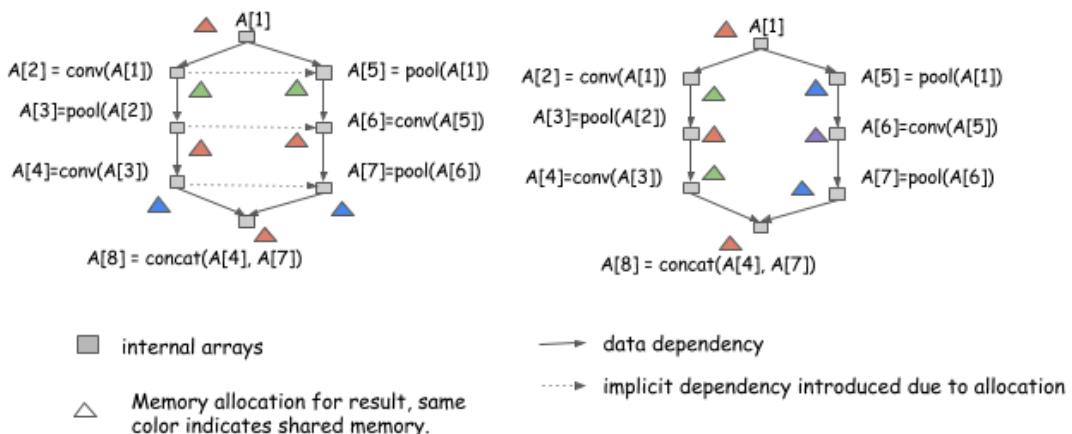
### 7.1.4 Static vs. Dynamic Allocation

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.

### 7.1.5 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:



**Fig 7.7.** Standard Memory sharing between **B** & the result of **E**.

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

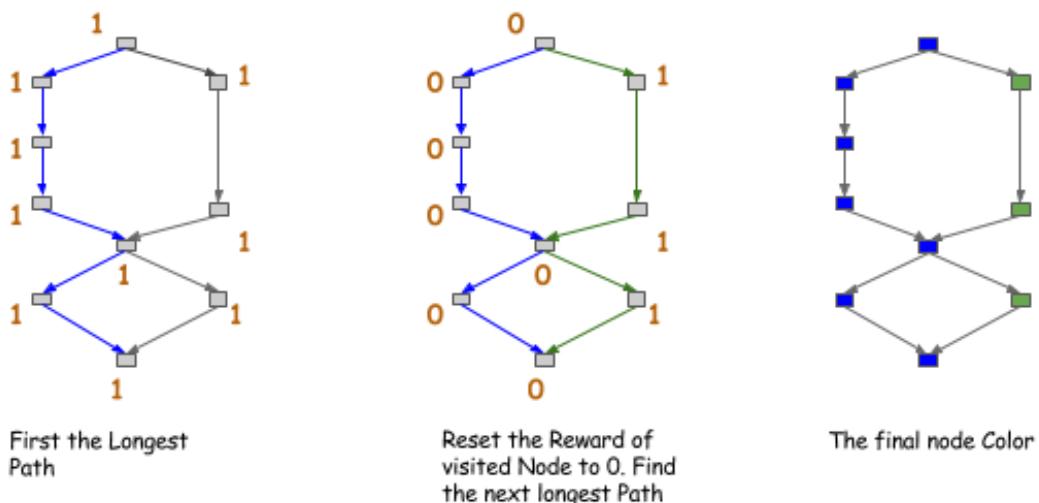
Being correct is our first principle. This means to execute in a way that takes implicit dependency memory sharing into consideration. You can do this by adding the implicit

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.

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 it costs only  $O(n)$  of time if you search for only the first  $k$  path.

### 7.1.6 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 Deep Learning Programming Style

---

### 7.1.7 References

More details can be found in: [Optimizing the Memory Consumption in DL\(MXNet\)](#).

## 7.2 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

### 7.2.1 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.

### 7.2.2 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.

### 7.2.3 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.

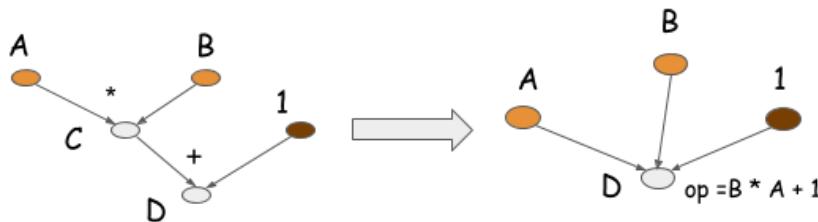


Fig 7.9. Operation Folding 示意图。

- 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.

#### 7.2.4 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：

## 7.2 Deep Learning Programming Style

---

```
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.

### 7.2.5 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。

### 7.2.6 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。

- 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。

### 7.2.7 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

## 7.3 Dependency Engine for Deep Learning

### 7.3.1 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.

## 7.3 Dependency Engine for Deep Learning

### 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.

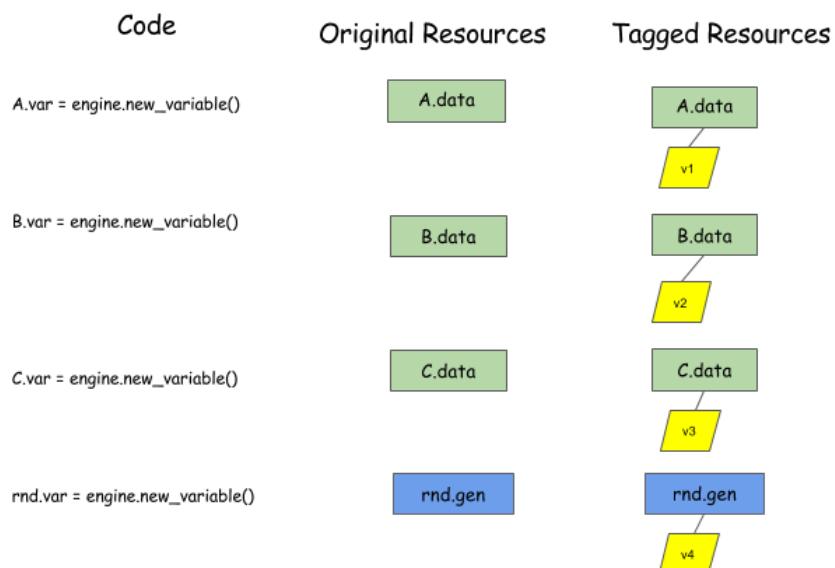


Fig 7.10. 第一步，给变量分配 tag

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，这样的好处是不涉及 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。

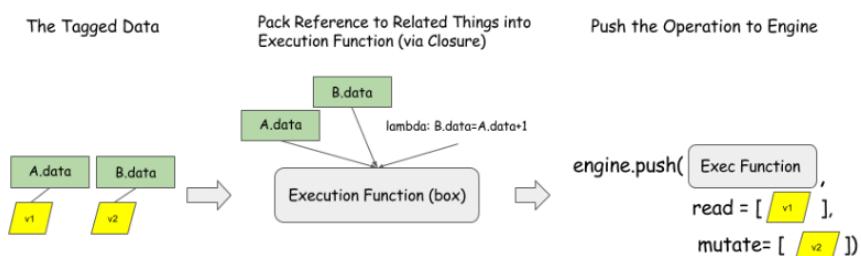


Fig 7.11. 把相应的 Function Closure push 进依赖分析引擎

### 7.3.2 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
<code>A=2 is pushed, because all its dependencies are satisfied, it is executed directly.</code>	<code>A</code> <code>B</code> <code>rnd</code>		
<code>B = A + rnd.uniform(-1,1) (3)</code>	<code>A</code> <code>B</code> <code>rnd</code>	<code>A = 2</code>	
<code>This represents in the intermediate stage, where the previous pushed op waits on the variable queue.</code>	<code>A</code> <code>B</code> <code>rnd</code>	<code>B = A + rnd.uniform(-1,1) (1)</code>	<code>A = 2</code>
<code>A=2 finishes, and the dependent operations are triggered. Another new operation is pushed.</code>	<code>A</code> <code>B</code> <code>rnd</code>	<code>B = A + rnd.uniform(-1,1)</code>	<code>A = 2</code>
<code>The newly pushed operation is added to the two dependency queues it waits on.</code>	<code>A</code> <code>B</code> <code>rnd</code>	<code>B = A + rnd.uniform(-1,1)</code>	
<code>The previous operation on B finishes, as a result, all dependencies of A = rnd.uniform is satisfied and it is able to run.</code>	<code>A</code> <code>B</code> <code>rnd</code>	<code>A = rnd.uniform(-1,1)</code>	<code>B = A + rnd.uniform(-1,1)</code>
<code>All pushed operations finished running.</code>	<code>A</code> <code>B</code> <code>rnd</code>		<code>A = rnd.uniform(-1,1)</code>
<ul style="list-style-type: none"> <li><span style="border: 1px solid black; padding: 2px;">operation (wait counter)</span> operation and the number of pending dependencies it need to wait for</li> <li><span style="background-color: green; border: 1px solid black; color: white; padding: 2px;">var</span> Variable queue, ready to read and mutate</li> <li><span style="background-color: yellow; border: 1px solid black; color: black; padding: 2px;">var</span> Variable queue, ready to read, but still have uncompleted reads. Cannot mutate</li> <li><span style="background-color: pink; border: 1px solid black; color: black; padding: 2px;">var</span> Variable queue, still have uncompleted mutations. Cannot read/write</li> <li><span style="border: 1px solid black; padding: 2px;">Execution Cycle(step)</span> Separator Line</li> </ul>			

Fig 7.12. 一个具体的例子

### 7.3.3 Discussion

- Dynamic vs. Static
- Mutation vs. Immutable

## 7.4 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

## 7.4 Designing Efficient Data Loaders for DL

---

### 7.4.1 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

### 7.4.2 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。

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.

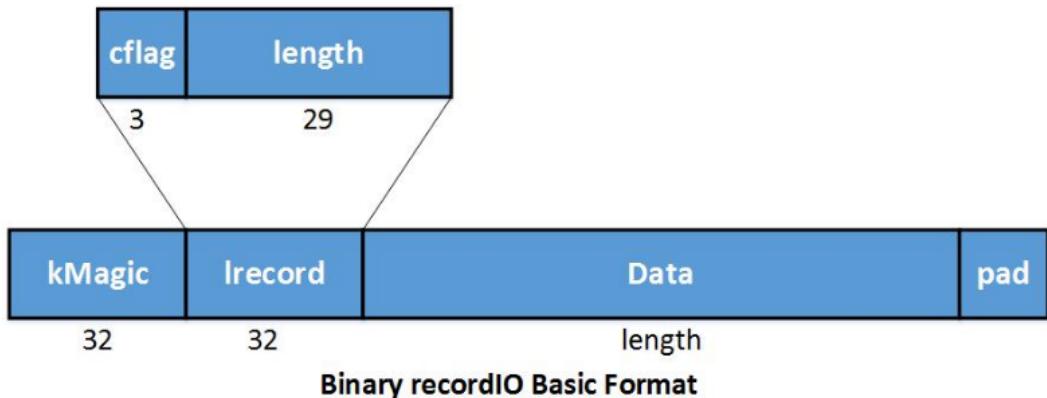
在 recordIO 中借助 Magic Number 来实现上述目的，具体是使用 dmlc-core 中的 *InputSplit* 函数来实现。这个函数极大的帮助了并行实现，因为每一个节点只处理一个 *Part*.

### 7.4.3 Data Loading and Preprocessing

#### Loading and Preprocessing on the Fly

In service of efficiency, we also address multi-threading techniques.

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。

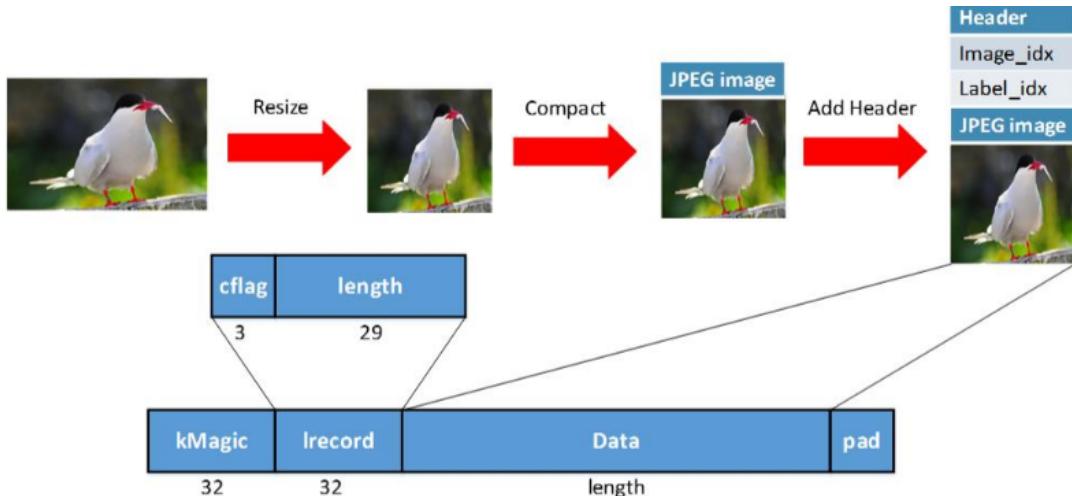


In MXNet's binary RecordIO, we store each data instance as a record. **kMagic** is a *magic number* indicating the start of a record. **Lrecord** encodes length and a continue flag. In lrecord,

- 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



**Fig 7.14.** Binary recordIO 的一个例子

## 7.4 Designing Efficient Data Loaders for DL

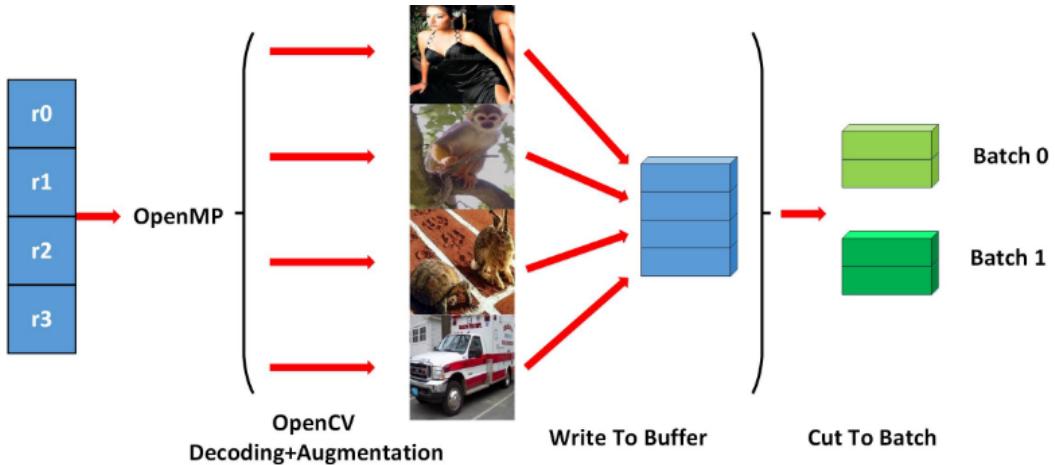


Fig 7.15. 并行预处理例子

### 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.

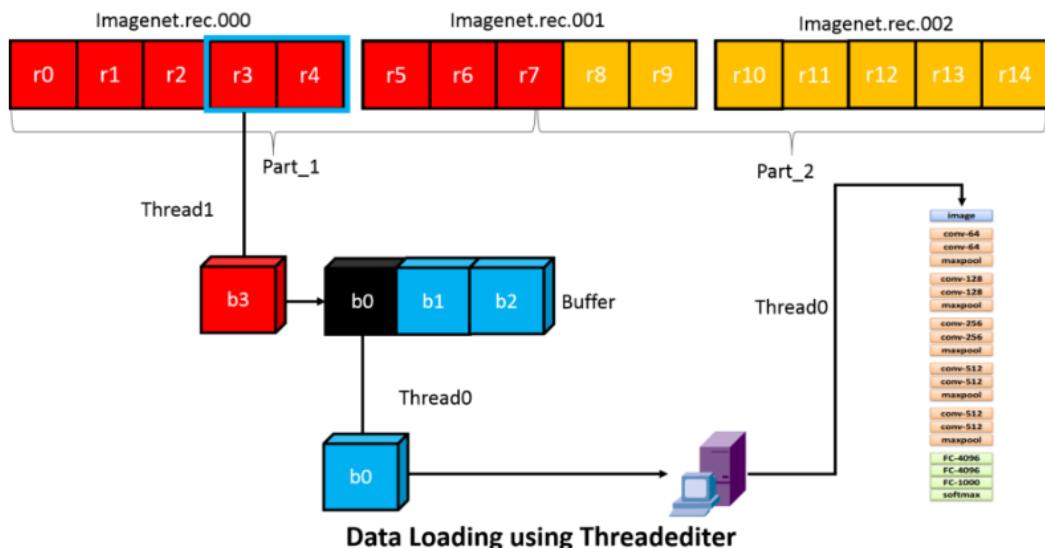


Fig 7.16. 数据预取的示意图，借助 Buffer 来实现

### 7.4.4 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
- 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

## 7.5 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
- Spawns 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。

## 7.6 MXNet-Gluon 创建模型

参考文献: [Gluon](#)

### 7.6.1 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 7.2: 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')
        # 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 7.3: 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 7.4: 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)

```

## 7.6.2 自定义层

也需要继承gluon.nn.Block 以及重载forward 函数。

Listing 7.5: 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 7.6: 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)
```

### 7.6.3 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：

Listing 7.7: 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。

## 7.7 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:

$$\begin{aligned} Out\_height &= (height - 1) * strides[0] - 2 * padding[0] + kernel\_size[0] + output\_padding[0] \\ Out\_width &= (width - 1) * strides[1] - 2 * padding[1] + kernel\_size[1] + output\_padding[1] \end{aligned}$$

↑正确性有待验证。

### 7.8 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。

#### 7.8.1 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 7.8: `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)
    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 7.9: 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。

## 7.8.2 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。

### 7.8.3 使用 mx.image 读取数据

从 imagelist 中读取可迭代的数据。

Listing 7.10: 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

### 7.8.4 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：

## 7.8 MXNet 读取训练数据

---

- 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：

```
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。

## 7.9 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。

# Chapter 8

## Tips in DL

### 8.1 Enlarge the FOV

增加网络的感受野。目前看到的主要方法如下：

- CRFs[11]
- Global Graph-reasoning module[2]
- Pooling
- Dilated conv[27]
- 

### 8.2 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
- 

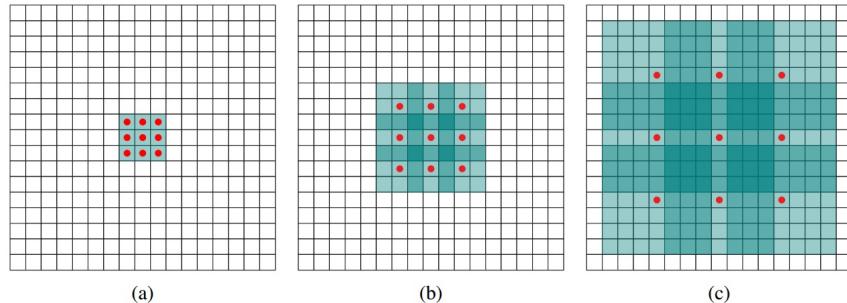
### 8.3 Multiscale Ability

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 ?
-

## 8.4 Dilated Convolution

主要原理如下。



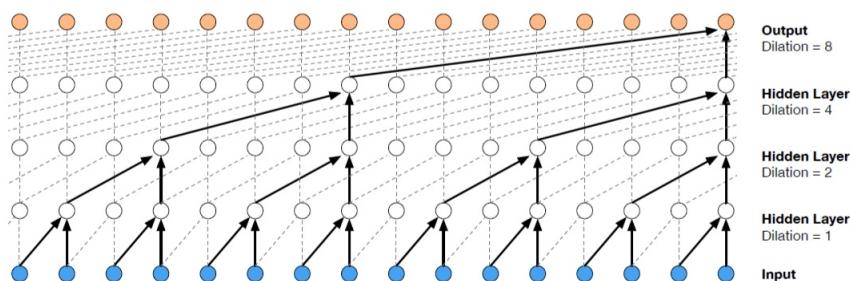
**Fig 8.1.** Dilated Convolution 示意图

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8.1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。



**Fig 8.2.** Dilated Convolution 在 WaveNet 中的应用示意图

参考文献：Dilated Conv 知乎

# 8.5 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。

## 8.5.1 Convolutional Spare 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 Unspervised Feature Learning。更多内容参考本小节的参考文献。

## 8.5.2 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。

## 8.5.3 Upsampling

用于 FCN[10] 和 DCGAN。

## 8.5.4 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？

### 8.5.5 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

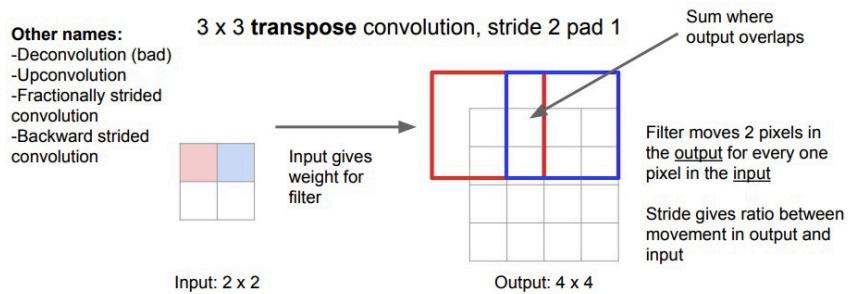


Fig 8.3. Transpose Convolution 过程示意图

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：“Deconvolution-Layer 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

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，一直作为 bilinear resize 的作用。

Listing 8.1: 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):
```

## 8.5 Deconvolutional Network

```
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

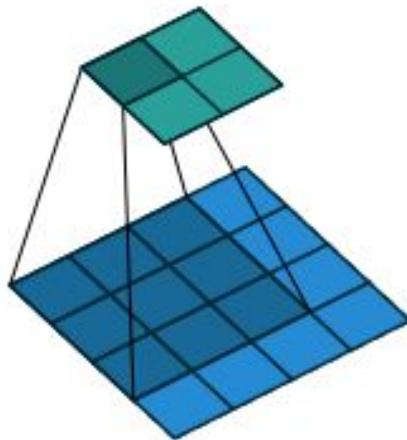
- **Upsampling**

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.



**Fig 8.4.** 一个例子, 用于卷积操作说明

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

$$\begin{pmatrix} w_{0,0} & w_{0,1} & w_{0,2} & 0 & w_{1,0} & w_{1,1} & w_{1,2} & 0 & w_{2,0} & w_{2,1} & w_{2,2} & 0 & 0 & 0 & 0 \\ 0 & w_{0,0} & w_{0,1} & w_{0,2} & 0 & w_{1,0} & w_{1,1} & w_{1,2} & 0 & w_{2,0} & w_{2,1} & w_{2,2} & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & w_{0,0} & w_{0,1} & w_{0,2} & 0 & w_{1,0} & w_{1,1} & w_{1,2} & 0 & w_{2,0} & w_{2,1} & w_{2,2} & 0 \\ 0 & 0 & 0 & 0 & 0 & w_{0,0} & w_{0,1} & w_{0,2} & 0 & w_{1,0} & w_{1,1} & w_{1,2} & 0 & w_{2,0} & w_{2,1} & w_{2,2} \end{pmatrix}$$

**Fig 8.5.** 卷积操作时的矩阵形式

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  $\frac{\partial Loss}{\partial y}$ .

$$\frac{\partial Loss}{\partial x_j} = \sum_i \frac{\partial Loss}{\partial y_i} \frac{\partial y_i}{\partial x_j} = \sum_i \frac{\partial Loss}{\partial y_i} C_{i,j} = \frac{\partial Loss}{\partial y} \cdot C_{*,j} = C_{*,j}^T \frac{\partial Loss}{\partial y}$$

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.

## 8.6 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。

## 8.7 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。

$$Precision_C = \frac{N(TP)_C}{N(Total)_C}$$

<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  $\delta$

## 8.8 统计学习方法

---

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,  $T(Total)_C$  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。

$$AveragePrecision_C = \frac{\sum Precision_C}{N(TotalImage)_C}$$

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, 是对于类别总数的平均值。

参考文献: [知乎文章](#)

## 8.8 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](#)

## 8.9 Distillation Module

文献来源: [26][13]

在 [26] 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[26]. 高效的利用中间任务的信息互补。文章 [26] 提出的三种不同的实现方式如图8.6所示。
- Distillation loss function[13]

利用 Distillation 帮助将 Teacher Network(精度更高) 的知识迁移到的 Student Network.

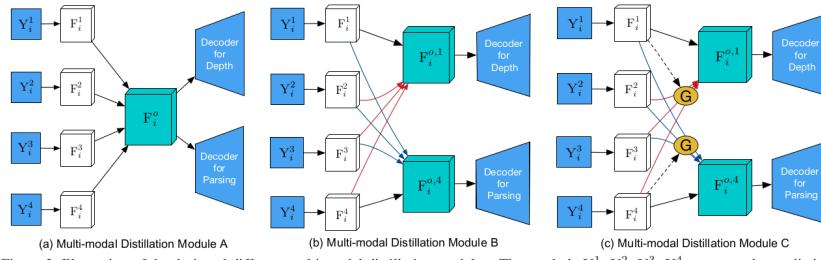


Figure 3. Illustration of the designed different multi-modal distillation modules. The symbols  $Y_i^1, Y_i^2, Y_i^3, Y_i^4$  represent the predictions corresponding to multiple intermediate tasks. The distillation module A is a naive combination of the multiple predictions; the module B proposes a mechanism 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 8.6. 三种不同的 Distillation Module**

### 8.9.1 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

### 8.9.2 Recurrent Knowledge Distillation [18]

## 8.10 光流估计中的 Average end-point error

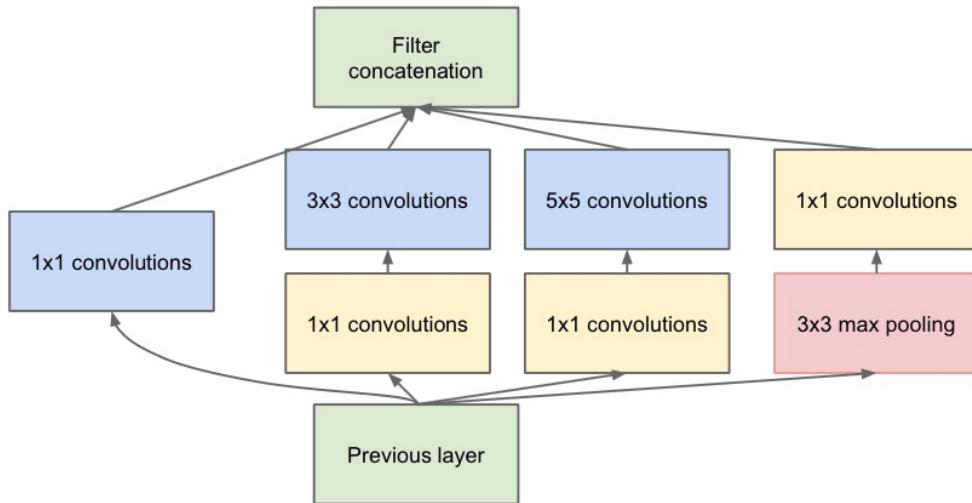
貌似就是类似于均方误差类似。具体定义还没查到。

## 8.11 CNN 中的卷及方式汇总

参考文献：CNN 中的卷积方式 2017.9-知乎

### 8.11.1 Inception

如图8.7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



**Fig 8.7. Inception 卷积结构**

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8.7中的结构进行 Stack, GoogLeNet?
- **Inception-V2**  
加入了 Batch Normalization 正则，去除了  $5 \times 5$  卷积，用两个  $3 \times 3$  卷积代替
- **Inception-V3**  
 $7 \times 7$  卷积被拆分成  $7 * 1 + 1 * 7$ , 可分离卷积?
- **Inception-V4**  
加入了残差结构。

### 8.11.2 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。

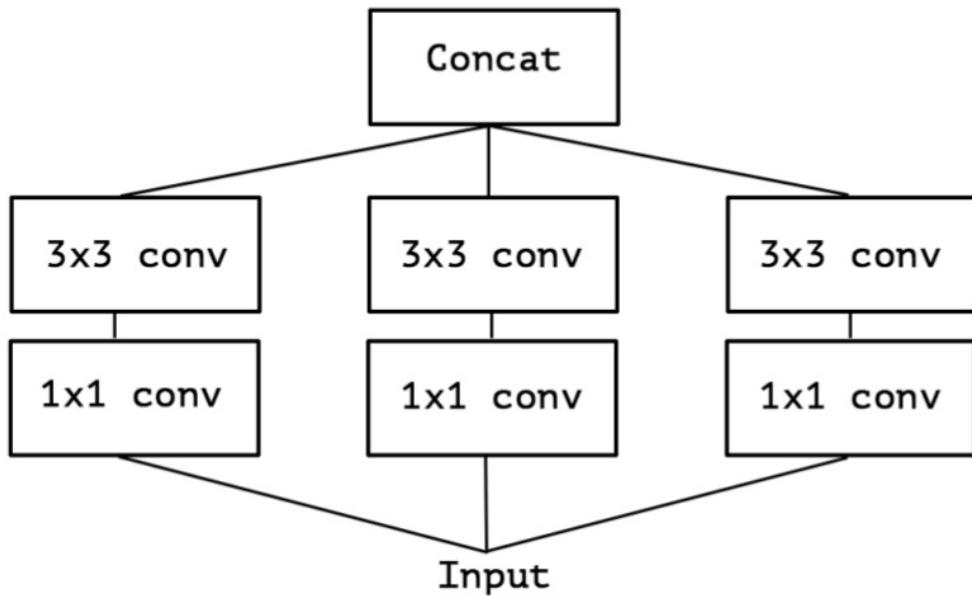
应用：

- FCN

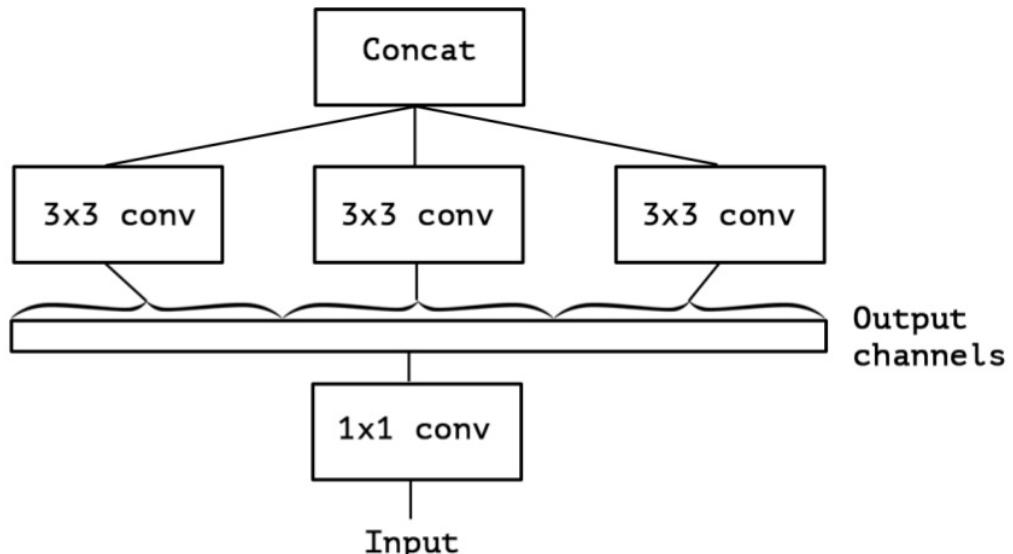
- WaveNet

### 8.11.3 深度可分离卷积, Depthwise Separable Convolution

是 Inception 的延续。



(a) 简化的 Inception



(b) Depthwise Separable Convolution 的示意图，可以看出来，与 Inception 很像。

**Fig 8.8.** Inception 结构与 Depthwise Separable Convolution 结构对比

如图8.8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

## 8.11 CNN 中的卷积方式汇总

别  $3 \times 3$  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  $3 \times 3$  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8.9）：

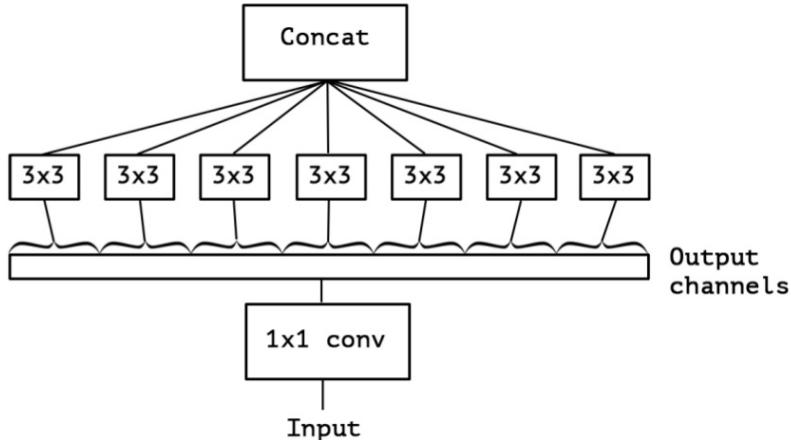


Fig 8.9. Channel 分组的极限版本

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  $1 \times 1$  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

### 8.11.4 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  $3 \times 3$  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8.10：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。

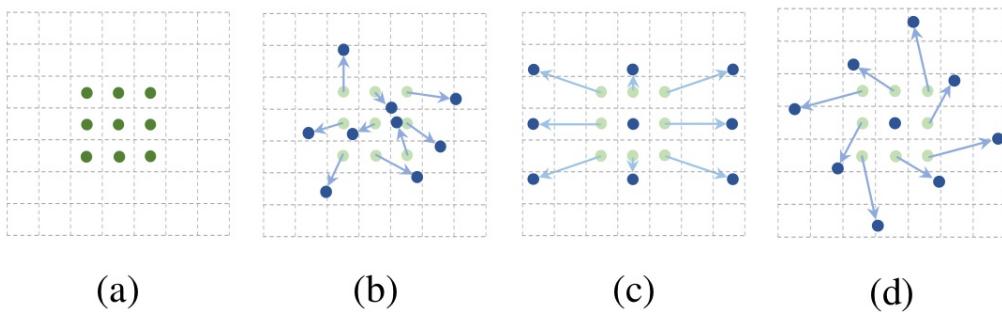
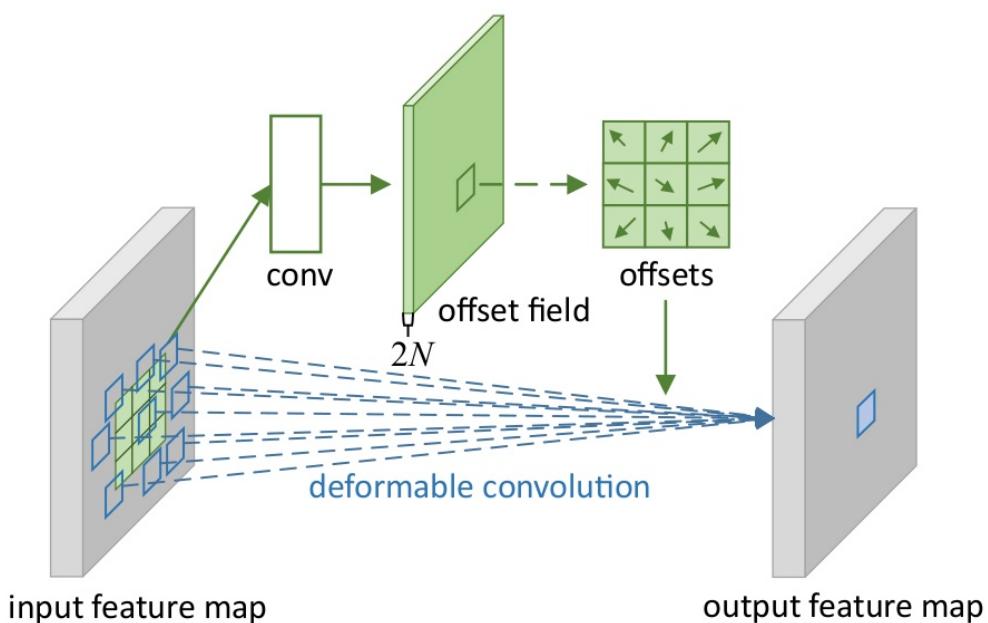
**Fig 8.10.** Deformable Convolution 示意图

图8.11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  $Batch, height, width, 2 * channels$ ，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 8.11.** 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。

### 8.11.5 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。

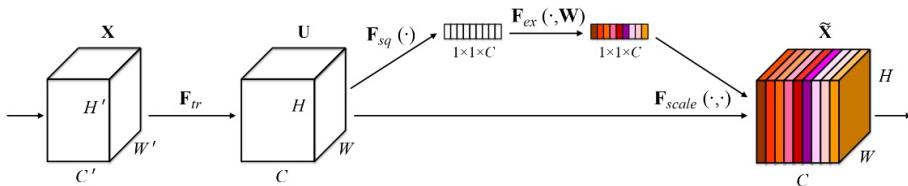


Figure 1. A Squeeze-and-Excitation block.

**Fig 8.12.** Squeeze-and-Excitation Block 示意图

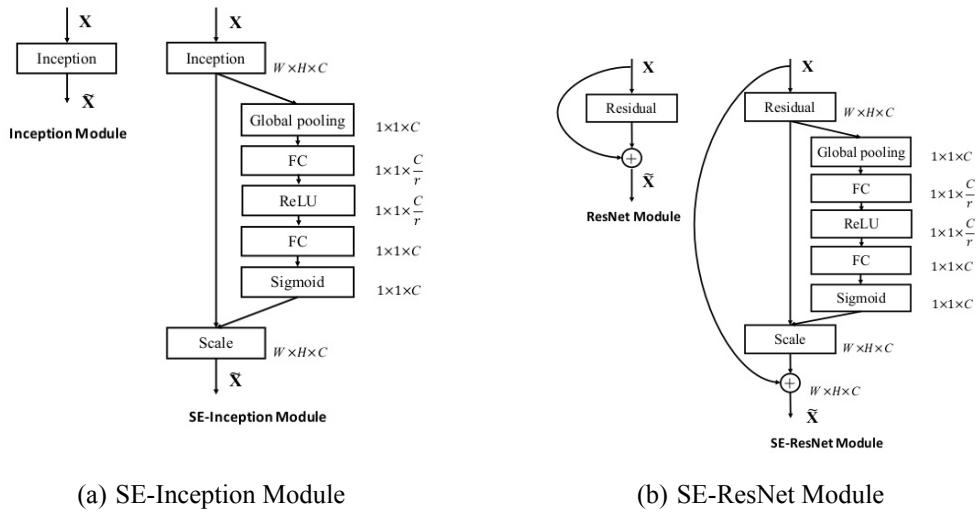
虽然听上去很复杂，但实现起来却比较简单？步骤如下：

- 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8.12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8.13所示。

<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。



**Fig 8.13.** SENet 与 Inception 以及 ResNet 结合的示意图

### 8.11.6 小结 - 比较

图8.14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。

## 8.12 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。

## 8.13 Pooling

	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 8.14. 不同卷积策略的比较

## 8.13 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）

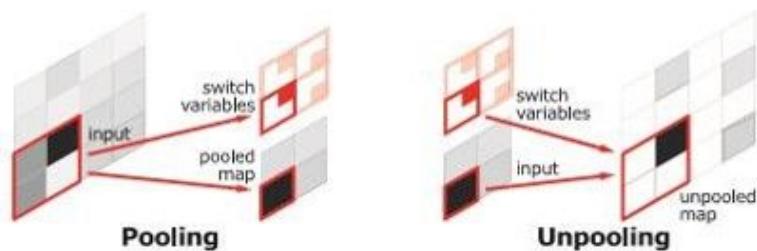


Fig 8.15. Unpooling 示意图

图8.15可以认为额外的switch variables 实现了保存 Pooling 过程中的位置。

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](#)

## 8.14 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。

## 8.15 CNN 中感受野的计算

参考文献:

[1] [CNN 中感受野的计算 -CSDN](#)

## 8.15 CNN 中感受野的计算

---

### [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

对应 conv3 的输入为  $6 \times 6$

对应 pool2 的输入为  $12 \times 12$

对应 conv2 的输入为  $14 \times 14$

对应 pool1 的输入为  $28 \times 28$

对应 conv1 的输入为  $30 \times 30$

所以 pool3 的感受野大小就是  $30 \times 30$

## 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

$$\text{strids}(i) = \text{stride}(1) * \text{stride}(2) * \text{stride}(3) * \dots * \text{stride}(i - 1)$$

## 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

$$\text{start}_{out} = \text{start}_{in} + (\frac{k - 1}{2} - p) * j_{in}$$

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8.16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8.17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。

## 8.15 CNN 中感受野的计算

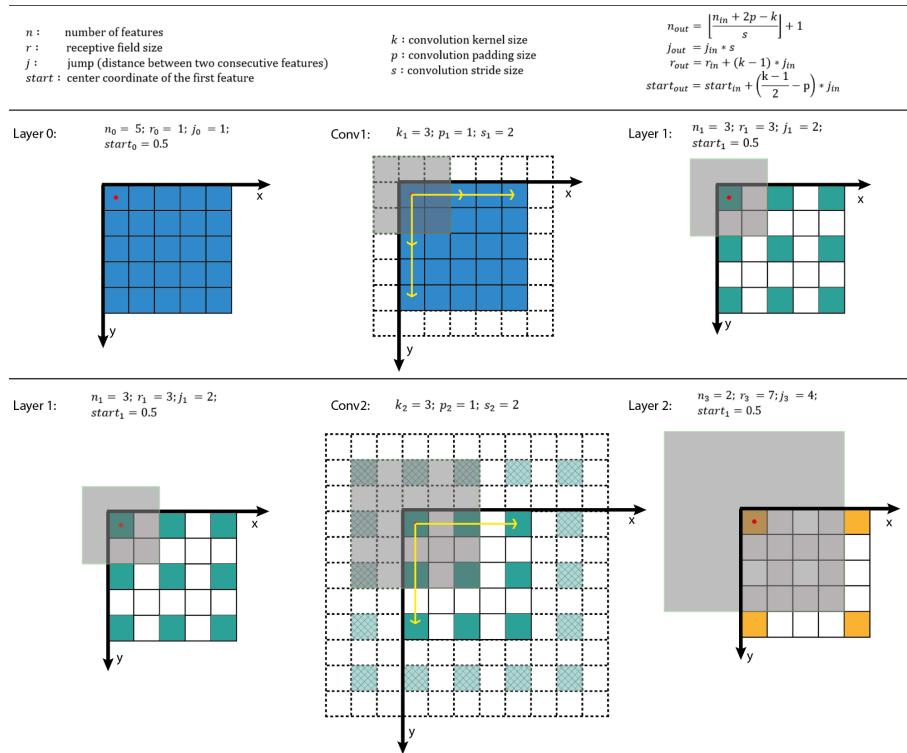


Fig 8.16. 文章的总结与说明

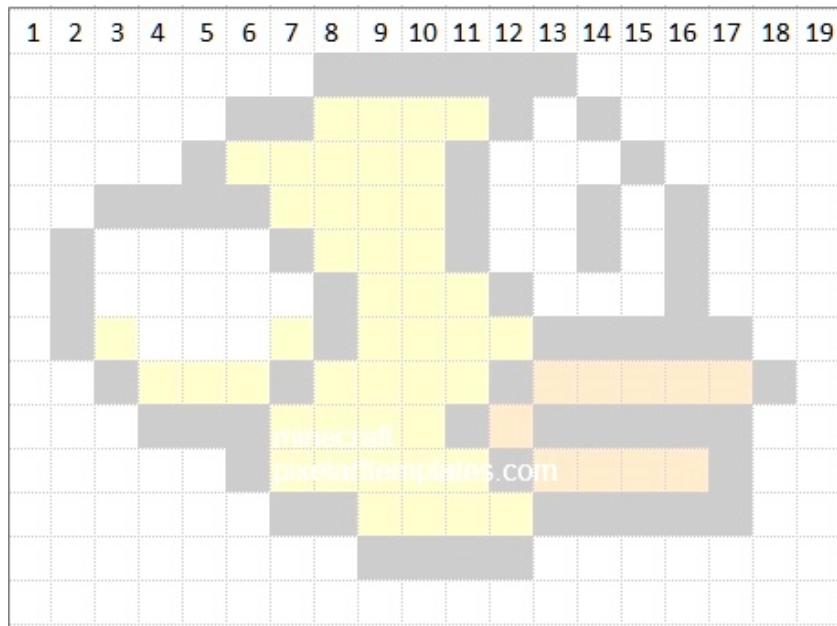


Fig 8.17. 二维图像像素编号示意图

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，因此要减去 (kernel\_size - 1) 个重叠部分，而重叠部分的计算方式则为感受野减去前面所说的偏移量，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。

## 8.16 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：

## 8.16 神经网络中的初始化

---

- 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。

### 8.16.1 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：

$$\text{Var}(z) = n \cdot \text{Var}(x) \text{Var}(\omega)$$

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，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。

## 8.17 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：

$$\frac{\partial a}{\partial x} = \sum_i \frac{\partial a}{\partial z_i} \cdot \frac{\partial z_i}{\partial x}$$

## 8.18 神经网络中的 Attention 机制

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):

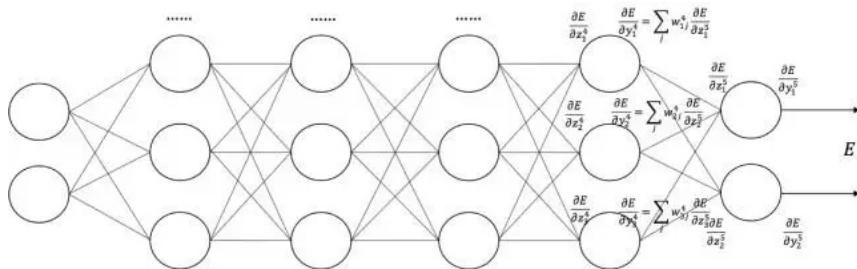


Fig 8.18. BP 计算过程示意图

## 8.18 神经网络中的 Attention 机制

参考文献：浅谈 Attention 机制 -知乎

### 8.18.1 Recurrent Models of Visual Attention

参考文献：[14]

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 Interate 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
- 

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。

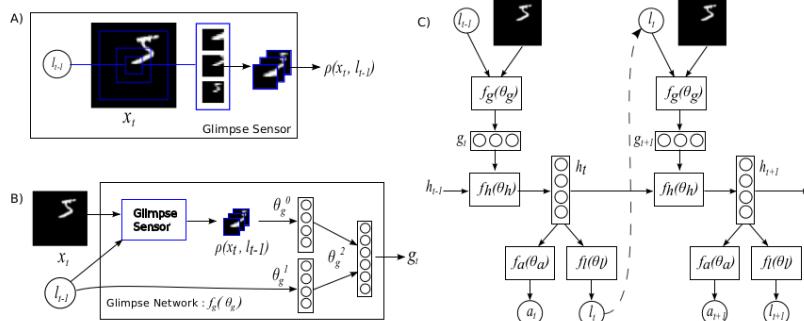


Figure 1: **A) Glimpse Sensor:** Given the coordinates of the glimpse and an input image, the sensor extracts a *retina-like* representation  $\rho(x_t, l_{t-1})$  centered at  $l_{t-1}$  that contains multiple resolution patches. **B) Glimpse Network:** Given the location ( $l_{t-1}$ ) and input image ( $x_t$ ), 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 8.19.** Attention 模型示意图

图8.19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：

## 8.19 小型网络

---

- Location actions

are chosen stochastically from a distribution parameterized by the location network  $f_l(h_t; \theta_t)$  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。

## 8.19 小型网络

参考文章: [CVPR2018 高效小网络 -知乎](#)

## 8.20 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.

## 8.21 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.

## 8.22 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:

## 8.23 待续

- 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。

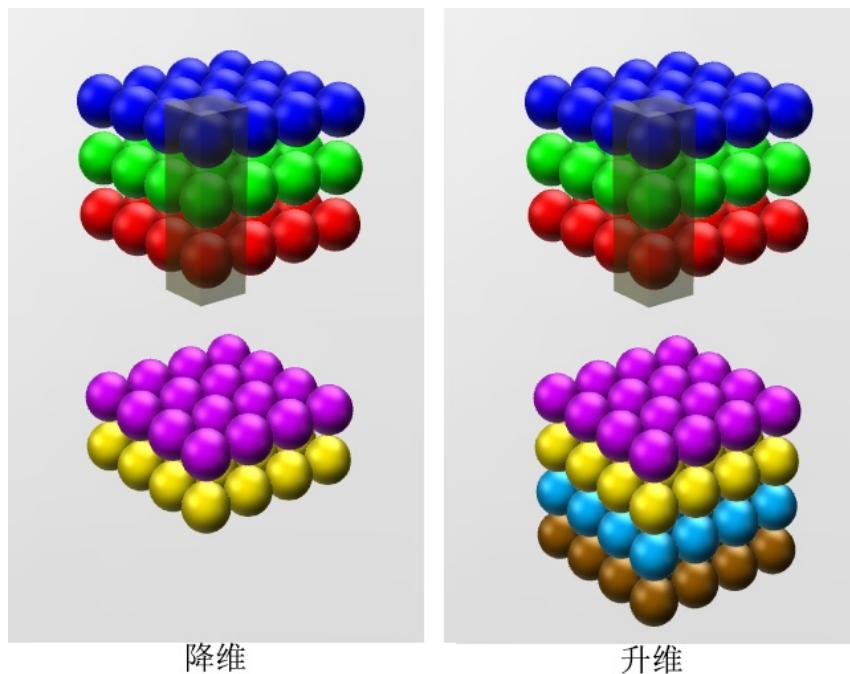


Fig 8.20.  $1*1$  Convolution 的作用示意图

## 8.23 待续



# Chapter 9

## Image Processing

### 9.1 Feature Extraction

#### 9.1.1 SIFT

详细信息可以参考: [SIFT CSND 博客](#)  
我的未验证实现: [SIFT Triloo Github](#)



# Chapter 10

## Feature Extraction

总结文章: [Object Detection 总结 Two Stage](#)

总结文章: [Object Detection 总结 One Stage](#)

### 10.1 Selective Search

#### 10.1.1 Efficient Graph-Based Image Segmentation

Selective Search 基于文章 [3] 中基于图分割的图像分割技术。

论文 [3] 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 [3] 的 **Algorithm 1**。

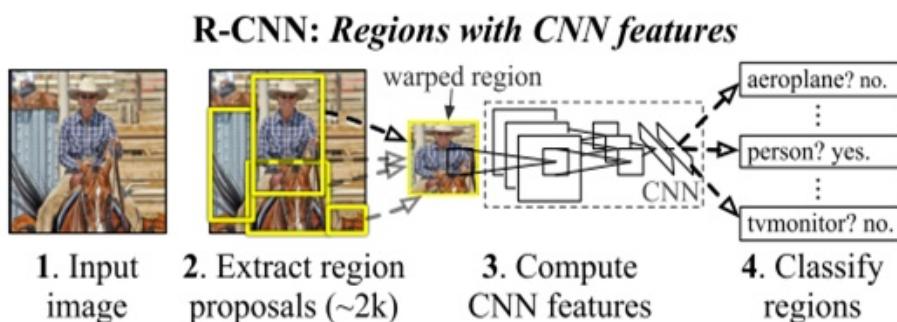
### 10.1.2 Selective Search

主要参考文献: [21]

接下来介绍 Selective Search 算法, 该算法利用 [3] 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。

## 10.2 Region CNN

### 10.2.1 概述



**Fig 10.1.** 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**

## 10.3 SPP Net

参考文献: [6]

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。

### 10.3.1 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：

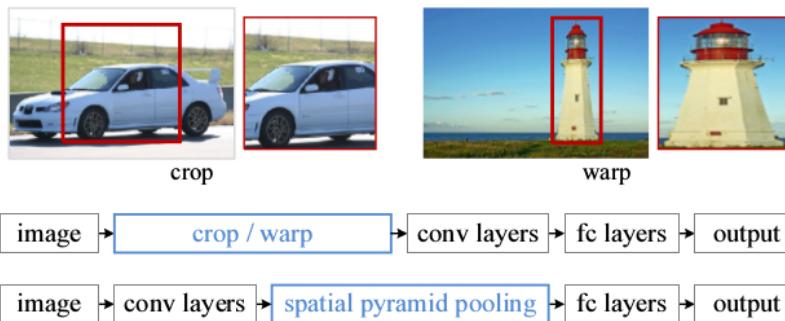


Figure 1: Top: cropping or warping to fit a fixed size. Middle: a conventional CNN. Bottom: our spatial pyramid pooling network structure.

**Fig 10.2.** 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。

### 10.3.2 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。

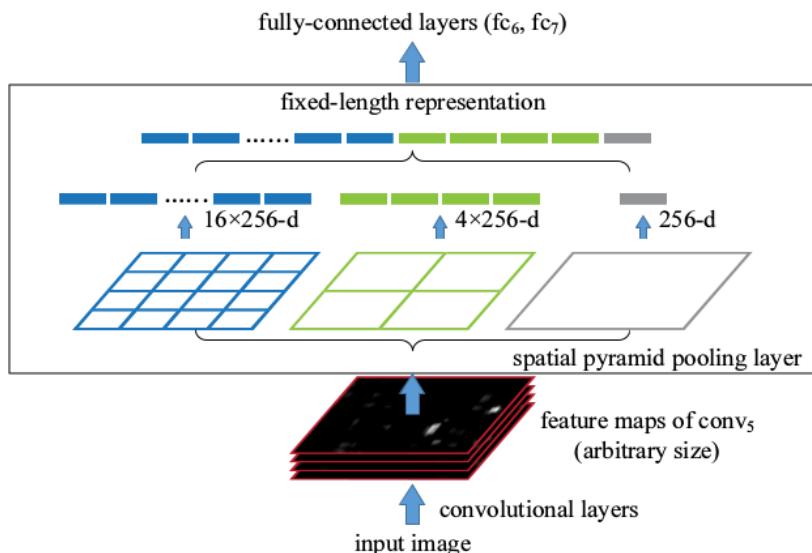


Figure 3: A network structure with a **spatial pyramid pooling layer**. Here 256 is the filter number of the  $\text{conv}_5$  layer, and  $\text{conv}_5$  is the last convolutional layer.

**Fig 10.3.** Pyramid Pooling 的计算过程示意图

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。

### 10.3.3 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。

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。

### 10.3.4 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！

**10.4 Fast RCNN**

**10.5 Faster RCNN**

**10.6 R FCN**

**10.7 FPN**

**10.8 Mask RCNN**

**10.9 YOLO**

**10.10 YOLO v2**

**10.11 YOLO v3**

**10.12 SSD**

**10.13 DSSD**

**10.14 Retina Net (Focal Loss)**

# 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] Pedro F Felzenszwalb and Daniel P Huttenlocher. Efficient Graph-Based Image Segmentation. *International Journal of Computer Vision*, 59(2):167--181, 2004.
- [4] Pedro F. Felzenszwalb and Daniel P. Huttenlocher. Efficient graph-based image segmentation. *International Journal of Computer Vision*, 59(2):167--181, 2004.
- [5] 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.
- [6] 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.
- [7] T.-W. Hui, X. Tang, and C. Change Loy. LiteFlowNet: A Lightweight Convolutional Neural Network for Optical Flow Estimation. *ArXiv e-prints*, May 2018.
- [8] 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.
- [9] Shutao Li, Xudong Kang, and Jianwen Hu. Image fusion with guided filtering. *IEEE Transactions on Image Processing*, 22(7):2864--2875, 2013.
- [10] J. Long, E. Shelhamer, and T. Darrell. Fully Convolutional Networks for Semantic Segmentation. *ArXiv e-prints*, November 2014.
- [11] Roberto Cipolla Marvin T.T. Teichmann. Convolutional crfs for semantic segmentation. *arXiv:https://arxiv.org/pdf/1805.04777.pdf*, 2018.

- 
- [12] J. McCormac, A. Handa, A. Davison, and S. Leutenegger. SemanticFusion: Dense 3D Semantic Mapping with Convolutional Neural Networks. *ArXiv e-prints*, September 2016.
  - [13] R. Mehta and C. Ozturk. Object detection at 200 Frames Per Second. *ArXiv e-prints*, May 2018.
  - [14] V. Mnih, N. Heess, A. Graves, and K. Kavukcuoglu. Recurrent Models of Visual Attention. *ArXiv e-prints*, June 2014.
  - [15] 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.
  - [16] O. Ronneberger, P. Fischer, and T. Brox. U-Net: Convolutional Networks for Biomedical Image Segmentation. *ArXiv e-prints*, May 2015.
  - [17] J. L. Schönberger, M. Pollefeys, A. Geiger, and T. Sattler. Semantic Visual Localization. *ArXiv e-prints*, December 2017.
  - [18] Jan C. van Gemert Silvia L. Pintea, Yue Liu. Recurrent knowledge distillation. *ArXiv e-prints*, May 2018.
  - [19] N. Sünderhauf, T. T. Pham, Y. Latif, M. Milford, and I. Reid. Meaningful Maps With Object-Oriented Semantic Mapping. *ArXiv e-prints*, September 2016.
  - [20] 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.
  - [21] 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.
  - [22] 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.
  - [23] 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.
  - [24] 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.
  - [25] Y. Xiang and D. Fox. DA-RNN: Semantic Mapping with Data Associated Recurrent Neural Networks. *ArXiv e-prints*, March 2017.
  - [26] 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.

## REFERENCES

---

- [27] Fisher Yu and Vladlen Koltun. Multi-scale context aggregation by dilated convolutions. In *ICLR*, 2016.

# **Index**

SAD, 8