

关于神经网络的一切（追真）

All About Neural Network (AANN)

2021.03.18

Coldwind

jimzeus@gmail.com

前言

本文是作者最近一段时间学习**神经网络**的笔记，部分内容原创，部分内容来自网上，部分则是对他人文章和论文的总结，有参考到的内容尽量给出了连接，如果忘了给出欢迎指出来。

它不是什么？

- 它不是一篇《从入门到精通》那一类一步步带领你学习深度学习的书，这一类书已经不少，并且都是专业人士所著。
- 它不是针对某一个知识点进行详细解说的文章，如果想了解其中某个具体的细节，可以去网上自行了解。

它是什么？

本文的目录结构可以让你知道 NN：

- 包括哪些知识领域
- 各个知识领域包括哪些知识点
- 知识点之间是什么关系

总之，这是个类似关于神经网络的知识点结构介绍的文章，有点像字典，并不建议以本文按顺序学习神经网络。

为什么要写这个？

作者记性不好，学过了解的过的东西经常会忘，下次碰到了又得重新学习一遍，为了减少回忆的时间，于是会将自己的理解进行文档化（俗称写笔记）。

为什么要 po 出来？

作者工作期间有过几次大规模系统学习的文档化，其中第一次是在将近 20 年前，当时作者还在做系统移植、驱动开发等 Linux 内核相关的工作（当时经常上 China Linux Forum 的人应该知道我），彼时的文章和书籍很少，主要就是 ULK、LDD、深入理解 Linux 内核几本。为了能全面理解，作者通读了当时的 Linux 某个版本（2.4.xx），并且形成了系统的笔记。这些笔记现在都找不到了，除了其中部分放到网上的（因为论坛上有人问才 po 出来解答，其中还有一篇被人收录在书里，好像是叫 Linux 什么宝典）。现在想来有点后悔，如果放出来能帮到其它人岂不是比消失不见好？

为什么没有做成网页版本？

精力有限。

为什么感觉没有完成？

迄今为止下面各章都是作者业余时间独力完成，限于作者个人精力，因此：

- 部分章节尚处于未完成状态
- 有些章节并未细化、深入、展开
- 由于作者并非 NN 方面的专业人士，很可能多有错误
- NN 发展日新月异，有些知识和经验过时得很快

自然也不可能做到标题中的“一切”这么厉害。

如果你想补充、扩展、修正文中的内容，欢迎通过以下方式联系作者：

邮箱地址：jimzeus@gmail.com

知乎帐号：<https://www.zhihu.com/people/jimzeus>

简介

介绍各章大致的内容，这里没有注明第几章，因为未来章节可能会有调整。

- “**概念&定义**” 介绍了与 NN 相关的一些基本概念，作为基础。
- “**Python 相关**” 介绍了一些 Python 相关的易混淆的概念，这章和 NN 关系不大，只是作者在工作过程中遇到的相关问题，需要搞清楚这些概念。
- “**框架&接口**” 介绍了和机器学习相关的库和框架，基本都是基于 Python 的。
- “**图像视频处理-OpenCV**” 介绍了传统的图像/视频处理的概念和算法，及 OpenCV 中的实现，这章本身和 NN 关系也不大，但是 CV 是 NN 的应用中的一个大头，了解这些有助于理解和解决 CV 相关的问题。
- “**网络构成**” 介绍了 NN 的各种结构、微结构、激活函数、损失函数、优化方法、标准化方法等等组成部件。
- “**研究方向：XX**” 介绍了 NN 的各个主要研究方向，目前 NN 最大的研究方向还是 CV（计算机视觉）相关，其次就是 NLP（自然语言处理）。对于每个子方向，我希望能给出“数据集”、“衡量标准”、“传统方法”及各个神经网络的实现这些内容，而实际上在很多方向上差得还很远。
- “**实践应用**” NN 的某些具体的应用，是上述某些研究方向的工程化/实用化。
- “**元学习**” 是和具体的研究方向无关的研究，比如迁移学习、集成学习、模型压缩、AutoML 等。
- “**硬件支持**”，和硬件相关的杂项放在这里，虽然作者是底层（操作系统/驱动）出身，但对这方面并未有深入研究
- “**NVidia GPU**” 本来是硬件支持中的单独一部分，后被单独拿出来作为一章，介绍了英伟达 GPU 的构成和型号
- “**边缘计算**” 介绍了移动设备上的部分硬件及各个框架，以及（并不那么）相关的 ARM 指令集/芯片/架构。
- “**其他**”：其他与 NN 相关的东西，大部分未完成或者有待调整。

目录

一、 概念&定义.....	20
(一) 希腊字母列表.....	20
(二) 微积分.....	21
1 · 导数和积分.....	21
2 · 链式法则.....	22
3 · 偏导数.....	22
4 · 梯度 (Gradient)	23
(三) 线性代数/几何学.....	23
1 · 张量/标量/向量/矩阵.....	23
2 · 特征向量/特征值.....	24
3 · 逐点操作.....	24
4 · 欧式距离.....	25
5 · 余弦距离 (余弦相似度)	25
6 · 曼哈顿距离.....	26
7 · 闵可夫斯基距离.....	26
8 · 范数.....	26
9 · 齐次坐标.....	27
10 · 仿射变换 (Affine Transformation)	29
(四) 概率论.....	29
1 · 概率.....	30
2 · 条件概率.....	30
3 · 贝叶斯定理.....	30
4 · 期望值 (EV)	31
5 · 概率密度函数 (PDF).....	31
6 · 累积分布函数 (CDF).....	31
7 · 概率质量函数.....	32
8 · 连续分级概率评分 (CRPS)	32
9 · 分位数 (Quantile).....	32
10 · 独立.....	33
11 · 独立同分布.....	33
12 · 联合分布.....	33
13 · 伯努利分布 (0-1 分布)	34
14 · 二项分布.....	34
15 · 泊松分布.....	34
16 · 指数分布.....	35
17 · 伽玛分布.....	36
18 · 正态分布 (高斯分布)	37
19 · 学生 t 分布.....	37
20 · 多元正态分布.....	39
(五) 信息论.....	39
1 · 信息熵 (Entropy)	39
2 · 联合熵 (Joint entropy)	40

3 · 条件熵 (Conditional entropy)	40
4 · 相对熵 (Relative entropy)	41
5 · 交叉熵 (Cross entropy)	41
(六) 统计学.....	42
1 · 方差 (Variance)	42
2 · 标准差 (Standard Deviation)	42
3 · 协方差 (Covariance)	42
4 · 自相关 (Auto correlation)	43
5 · 互相关 (Correlation)	44
6 · 蒙特卡洛法 / 拉斯维加斯法.....	44
7 · 马尔可夫链 (Markov Chain)	44
8 · 回归分析.....	44
9 · 线性回归 (Linear Regression)	45
10 · 逻辑回归 (Logistic Regression)	45
11 · softmax 回归.....	45
12 · 最大似然估计 (MLE)	45
(七) 数字信号处理.....	46
1 · 傅立叶变换.....	46
2 · 离散傅立叶变换.....	46
3 · 快速傅立叶变换.....	46
4 · 基波和谐波.....	46
5 · 滤波器.....	47
6 · FIR 滤波器.....	47
7 · IIR 滤波器.....	47
8 · 小波变换.....	47
9 · ACF (自相关函数)	47
10 · XCF (互相关函数)	48
(八) 机器学习.....	48
1 · 监督学习.....	49
2 · 统计分类.....	49
3 · 回归分析.....	50
4 · 无监督学习.....	50
5 · 聚类分析.....	50
6 · 强化学习.....	51
7 · SVM.....	51
8 · k-means.....	52
9 · 感知器.....	52
10 · 朴素贝叶斯方法.....	53
11 · 决策树.....	53
12 · 随机森林.....	54
13 · kNN 算法.....	54
14 · 隐马尔可夫模型.....	55
15 · 神经网络.....	56
(九) NN 相关.....	56

1 · 数据集.....	56
2 · 层.....	57
3 · 参数和 FLOPs.....	58
4 · Ground Truth.....	58
5 · 置信度.....	58
6 · One-hot 标签.....	58
7 · 降维/升维.....	59
8 · 上采样/下采样.....	59
9 · 编码器/解码器.....	59
10 · 开集/闭集.....	59
11 · 类内/类间.....	59
12 · 二分类任务衡量标准.....	60
13 · 感受野.....	61
14 · 表现力.....	61
15 · 嵌入.....	61
16 · 二分类/多分类/多标签分类.....	62
17 · 骨干网络 (Backbone)	62
18 · 分布 (Distribution)	62
(十) 超参数.....	64
1 · 学习率.....	64
2 · Dropout Ratio.....	64
3 · Batch size.....	64
(十一) NN 训练相关.....	64
1 · Epoch / Batch.....	64
2 · 过拟合 (Overfit)	65
3 · 学习率衰减.....	65
4 · 权值衰减.....	65
5 · 梯度消失.....	65
6 · 超参数.....	66
7 · 标准化 (Normalization)	66
8 · 泛化 (Generalization)	66
(十二) CNN 相关.....	66
1 · 全连接层.....	66
2 · 卷积层.....	66
3 · 池化层.....	66
4 · 卷积核.....	66
5 · 特征图.....	67
6 · 填充 (Padding)	67
7 · 步幅 (Stride)	67
8 · 通道 (Channel)	67
二、 Python 相关.....	68
(一) 包、模块、属性.....	68
(二) 类和对象.....	70
1 · 对象.....	70

2 · 类=对象.....	70
3 · metaclass.....	71
4 · 实例化的过程.....	72
(三) 类.....	73
1 · Final 和 Virtual.....	73
2 · 抽象基类.....	75
3 · 内嵌抽象类 (Collections ABC)	76
4 · 内嵌类型 (built-in types)	78
5 · 类的层次.....	78
(四) 泛型别名.....	79
(五) 方法.....	80
1 · 标准库抽象类、内嵌函数、类特殊方法.....	80
2 · 内嵌函数 (Built-in Functions)	81
3 · 类特殊方法 (Special method)	83
(1) 基本方法.....	83
(2) 属性访问.....	84
(3) 其他.....	84
(4) with 语句上下文.....	85
(5) 协程相关.....	85
4 · 修饰符.....	85
(六) 迭代器.....	86
1 · 定义.....	87
(1) iterable.....	87
(2) iterator.....	87
(3) generator.....	87
(4) generator iterator.....	88
2 · 内嵌抽象基类.....	88
(1) Iterable.....	88
(2) Iterator.....	88
(3) Reversible.....	89
(4) Generator.....	89
3 · 类特殊方法.....	89
三、 框架&接口.....	89
(一) 框架简介.....	89
1 · 传统库.....	90
2 · 通用框架.....	91
3 · 专用框架.....	91
4 · 边缘计算.....	92
5 · 快捷接口.....	93
(二) 不同格式间的转换.....	94
1 · 文件格式.....	94
2 · 格式转换.....	95
3 · MMdnn 转换.....	98
4 · ncnn 转换.....	99

(三) 传统库.....	99
1 · wave.....	99
2 · scipy.....	99
3 · numpy.....	100
4 · pandas.....	101
5 · matplotlib.....	104
四、 NN 框架.....	105
(一) Google.....	105
1 · Tensorflow (基础框架)	105
2 · Keras (高级接口)	106
(二) Facebook.....	110
1 · Pytorch (基础框架)	110
2 · torchvision (图像处理)	114
3 · Detectron (已废弃)	115
4 · Maskrcnn-benchmark (已废弃)	115
5 · Detectron2 (计算机视觉)	115
6 · PySlowFast (视频理解)	117
7 · Prophet (时间序列分析)	118
(三) Amazon.....	118
1 · MXNet (基础框架)	118
2 · Gluon (高级接口)	119
3 · gluoncv (图像处理)	119
4 · gluonts (时间序列)	120
5 · gluonnlp (自然语言处理)	142
(四) CUHK.....	143
1 · mmdetection (图像处理)	143
2 · mmsegmentation (图像分割)	143
3 · mmaction (视频理解)	144
(五) 图森未来.....	146
1 · SimpleDet (图像处理)	146
(六) Hugging Face.....	146
1 · Transformers.....	146
(七) 阿里巴巴.....	146
1 · MNN.....	146
(八) 腾讯.....	147
1 · ncnn.....	147
(九) Redmon.....	147
1 · Darknet.....	147
(十) 其它.....	150
1 · PyVideoResearch.....	150
2 · Theano.....	150
3 · sklearn.....	150
4 · DNN (OpenCV).....	150
五、 传统图像视频处理 (OpenCV)	152

(一) 基本绘图	152
(二) 色彩空间 (Colorspace)	153
(三) 几何变换 (Geometry Transformation)	154
1. 刚体变换 (Rigid Transformation)	155
2. 仿射变换 (Affine Transformation)	156
3. 投影变换 (Projective Transformation)	156
4. OpenCV Python	163
(四) 阈值处理 (Threshold)	164
1. 大津算法 (Otsu's Method)	164
(五) 过滤器-模糊	165
1. 2D 卷积 (2D Convolution)	165
2. 均值模糊 (Averaging Blur)	165
3. 中值模糊 (Median Blur)	166
4. 高斯模糊 (Gaussian Blur)	166
5. 双边滤波 (Bilateral Filter)	167
(六) 形态变换 (Morphological Transformation)	168
(七) 图像导数 (Image Gradient)	169
1. 索伯算子 (Sobel Operator)	169
2. Scharr 算子 (Scharr Operator)	170
3. 拉普拉斯算子 (Laplace Operator)	171
(八) Canny 边缘检测算法	172
1. Douglas-Peucker 算法 (TODO)	173
(九) 图像金字塔 (Image Pyramids)	173
(十) 轮廓 (Contour)	174
1. 矩 (moment)	174
(十一) 直方图 (Histogram)	176
(十二) 模板匹配 (Matching Template)	176
(十三) 霍夫变换 (Hough Transform)	177
(十四) Harris 角检测器 (TODO)	178
(十五) SIFT 算法 (TODO)	178
(十六) SURF 算法 (TODO)	178
(十七) 图像分割 (Image Segmentation)	178
(十八) 视频处理 (Video Process)	178
1. 均值漂移 (mean-shift)	178
2. 光流	179
(十九) 相机标定 (Camera Calibration)	179
1. 光心、焦距、焦点	179
2. 坐标系	179
3. 畸变 (distortion)	182
4. 相机标定	184
5. OpenCV Python	186
六、网络构成	188
(一) DNN 及 CNN 微结构	188
1. 全连接层 (Dense)	188

2 · 池化层 (Pooling)	189
3 · 全局池化层.....	189
4 · 反池化层.....	189
5 · 常规卷积层.....	190
6 · 本地卷积层.....	191
7 · Dropout 层.....	191
8 · Deconv (ZFNet)	191
9 · Group Conv (AlexNet)	192
10 · Depthwise Separable 卷积 (Xception)	192
11 · 3*3 卷积核 (VGGNet)	194
12 · 1*1 卷积核.....	195
13 · Spatial Separable 卷积.....	195
14 · 带孔卷积.....	196
15 · Bottleneck 结构.....	196
16 · Residual Block (ResNet)	197
17 · Inverted Residual Block (MobileNet V2)	198
18 · Linear bottleneck (MobileNet V2)	199
19 · Dense Block (DenseNet)	199
20 · Inception 结构 (GoogleNet)	200
21 · ResNeXt 结构 (ResNeXt)	202
22 · Fire Module (SqueezeNet)	202
23 · SE 结构 (SENet)	203
24 · NASNet 单元 (TODO)	204
25 · AmoebaNet 单元 (TODO)	204
26 · SPP 层 (SPP-net)	204
27 · RoI Pooling 层 (Fast R-CNN)	205
(二) CNN 结构.....	205
(三) RNN 结构.....	206
1 · 通用结构.....	206
2 · 变种结构.....	208
3 · 双向 RNN.....	212
(四) RNN 微结构.....	212
1 · 基本 RNN 单元.....	213
2 · LSTM 单元.....	214
3 · GRU 单元.....	216
(五) 注意力机制 (Attention)	216
1 · CNN 中的 Attention.....	219
2 · RNN 中的 Attention (201409).....	219
3 · Soft & Hard attention (201502).....	220
4 · Global & Local Attention (201508).....	221
5 · Transformer & Self-Attention (201706).....	222
6 · Non-Local Network (201711)	222
(六) GAN 结构.....	224
(七) 激活函数 (Activation Function)	224

1 · sigmoid 函数.....	227
2 · tanh 函数.....	228
3 · ReLU 函数系列.....	228
4 · softmax 函数.....	230
5 · softsign 函数.....	230
(八) 损失函数 (Loss Function)	231
1 · MSE (均方误差)	231
2 · RMSE.....	231
3 · MAE (平均绝对误差)	231
4 · MAPE.....	232
5 · sMAPE.....	232
6 · MASE.....	232
7 · MSIS.....	233
8 · ND.....	233
9 · NRMSE.....	233
10 · OWA.....	233
11 · Cross-entropy Loss (交叉熵损失)	233
12 · Softmax Loss.....	234
13 · Triplet Loss (FaceNet)	234
14 · Contrastive Loss (TODO).....	235
15 · Center Loss (2016)	235
16 · A-Softmax Loss (SphereFace)	237
17 · Focal Loss (RetinaNet).....	238
(九) 优化方法 (Optimizer)	239
1 · 梯度下降法 (GD)	240
2 · 动量法 (Momentum)	241
3 · NAG.....	242
4 · AdaGrad.....	243
5 · RMSprop.....	243
6 · Adam.....	243
(十) 标准化方法 (Normalization)	244
1 · LRN (AlexNet).....	245
2 · Min-max Normalization.....	245
3 · Z-score Normalization.....	245
4 · L1 Normalization.....	246
5 · L2 Normalization.....	246
6 · Batch Normalization (201502).....	246
7 · Layer Normalization (201607).....	246
8 · Instance Normalization (201607).....	247
9 · Group Normalization (201803).....	248
10 · Switchable Normalization (201806).....	248
七、研究方向：图像.....	250
(一) 图像分类 (Image Classification)	251
1 · 衡量标准.....	252

2 · 数据集.....	253
3 · 代码.....	255
4 · LeNet (1998)	255
5 · AlexNet (2012)	256
6 · ZFNet (201311)	257
7 · VGGNet (201409)	258
8 · GoogLeNet (201409)	259
9 · ResNet (201509)	261
10 · SqueezeNet (201602)	261
11 · DenseNet (201608)	262
12 · Xception (201610)	263
13 · ResNeXt (201611).....	264
14 · MobileNet V1 (201704)	264
15 · NasNet (201707).....	266
16 · ShuffleNet V1 (201707).....	266
17 · SENet (201709)	267
18 · MobileNet V2 (201801)	268
19 · MNasNet (201807)(TODO).....	269
20 · ShuffleNet V2 (201807).....	269
21 · EfficientNet (201905)(TODO).....	270
22 · RegNet (TODO).....	271
(二) 目标检测 (Object Detection)	271
1 · 衡量标准.....	274
2 · 数据集.....	276
3 · 代码.....	279
4 · R-CNN 系列.....	279
5 · YOLO 系列.....	284
6 · SSD (201512)	293
7 · FPN (201612)(TODO).....	295
8 · RetinaNet (201708)	295
9 · MaskX R-CNN (201711)(TODO).....	296
10 · CenterNet (201904)(TODO).....	296
11 · EfficientDet (201911)(TODO).....	296
(三) 图像分割 (Image Segmentation)	296
1 · FCN (201411)	297
2 · SegNet (201511)	298
3 · DeepLab 系列(TODO).....	299
4 · Mask R-CNN (201703)	300
5 · PointRend (201912) (TODO).....	300
(四) 人脸识别和建模 (Face Recognition)	300
1 · 数据集.....	302
2 · 代码.....	305
3 · DeepFace (201406)	305
4 · DeepID 系列 (2014)	306

5 · FaceNet (201503)	308
6 · MTCNN (201604)	308
7 · CenterLoss (2016)	309
8 · SphereFace (201704)	309
9 · FacePoseNet (201708)	309
10 · CosFace (201801)	310
11 · ArcFace/InsightFace (201801)	310
12 · SeqFace (201803)	310
13 · MobileFaceNets (201804)	310
(五) 人体姿态估计 (Pose Estimation)	311
1 · 数据集.....	312
2 · CPM (201602).....	312
3 · HourGlass (201603).....	313
4 · OpenPose (201611).....	313
5 · CPN (201711).....	313
6 · MSPN (201901).....	314
7 · HRNet (201902).....	314
八、研究方向：视频.....	315
(一) 行为识别 (Video Action Classification)	315
1 · 数据集.....	317
2 · 衡量标准.....	319
3 · 传统方法.....	319
4 · Two-Stream.....	319
5 · 3D Conv.....	324
6 · Skeleton-based.....	330
7 · LSTM-based.....	331
(二) 时序行为识别 (Temporal Action Recognition)	331
1 · 数据集.....	332
2 · 衡量标准.....	332
3 · SSN.....	332
(三) 时空行为识别	333
1 · 数据集.....	333
(四) 视频字幕 (Video Captioning)	333
1 · 数据集.....	333
(五) 视频问答 (Video QA)	333
1 · Zhou (201804).....	334
2 · Video-BERT (201904).....	334
(六) 视频目标跟踪 (Video Object Tracking)	334
1 · 衡量标准.....	335
2 · 数据集.....	336
(七) 多目标跟踪 (Multiple Object Tracking)	336
1 · 衡量标准.....	337
2 · 数据集.....	338
3 · 传统算法.....	339

4 · SORT (201602).....	340
5 · DeepSORT (201703).....	341
6 · MOTDT (201809).....	341
7 · JDE(201909).....	341
8 · FairMOT (202004).....	341
(八) 行人识别 (Person Recognition)	342
1 · 数据集.....	342
2 · MLCNN (2015ICB)	343
3 · OIM 损失函数 (201604)	344
4 · PCB 和 RPP (201711)	345
5 · Height, Color, Gender (201810)	346
6 · st-ReID (201812)	347
7 · DG-net (201904)	347
九、研究方向：自然语言处理 (NLP)	348
(一) 概念.....	348
1 · 语言学相关.....	348
2 · Tokenize (分词)	349
3 · TF-IDF.....	349
4 · 语言模型 (LM)	349
5 · n-grams (n 元语法)	350
(二) 任务.....	350
1 · 语言学 NLP 任务分类.....	351
2 · 神经网络 NLP 任务分类.....	353
(三) 衡量标准.....	354
1 · Accuracy.....	354
2 · F1-Score.....	354
3 · BLEU.....	354
(四) 数据集.....	354
1 · ChnSentiCorp.....	354
2 · SQuAD.....	354
3 · GLUE.....	355
4 · SuperGLUE.....	357
5 · CLUE.....	357
(五) 框架.....	359
1 · SpaCy.....	359
2 · NLTK.....	359
3 · Stanford CoreNLP (2014)	359
4 · Gensim.....	360
5 · OpenNMT.....	360
6 · Par1AI.....	361
7 · DeepPavlov.....	361
8 · SnowNLP.....	361
9 · Senta.....	362
10 · HuggingFace Transformers.....	362

11 · HanLP.....	364
12 · A11enNLP.....	365
(六) NN : 词的表征.....	365
1 · NNLM (2003)	366
2 · Word2Vec (201301)	367
3 · GloVe(2014).....	371
4 · fastText(201607).....	371
5 · ELMo(201802).....	372
(七) NN : 基于 RNN.....	373
1 · Encoder-Decoder (201406)	373
2 · Seq2Seq (201409)	374
3 · Attention 机制 (201409)	375
(八) NN : 基于 Transformer.....	375
1 · Transformer(201706).....	376
2 · GPT 系列.....	386
3 · BERT 系列.....	389
4 · T5 (201910)	401
十、研究方向：时间序列 (TS)	401
1 · 数据集.....	404
2 · 衡量标准.....	404
(二) 传统方法及概念.....	404
1 · AR 模型.....	404
2 · VAR 模型.....	405
3 · MA 模型.....	406
4 · 白噪声	406
5 · ARMA 模型.....	406
6 · ARIMA 模型.....	407
7 · ARFIMA 模型.....	408
8 · ARCH 模型.....	408
9 · DTW.....	408
10 · COTE.....	409
11 · HIVE-COTE (2016ICDM).....	409
(三) 神经网络方法.....	409
1 · WaveNet (201609) (TODO).....	409
2 · DeepAR (201704).....	409
3 · Deep state(201800) (TODO).....	410
4 · Deep factor (201905) (TODO).....	410
十一、实践应用.....	412
(一) 车牌识别.....	412
1 · 数据集.....	412
2 · 代码：EasyPR.....	413
3 · 代码：HyperLPR.....	413
(二) 无人机识别.....	413
1 · 数据集.....	413

(三) 视频监控异常检测.....	413
1 · Real-World Anomaly Detection in Surveillance Videos (201801) (TODO)	414
(四) 红绿灯识别.....	414
1 · 数据集.....	414
2 · 基于 OpenCV 的红绿灯识别.....	415
3 · 用深度学习识别交通灯.....	415
十二、元学习.....	415
(一) 迁移学习	415
1 · Fine-tuning.....	416
(二) 集成学习	416
1 · Boosting.....	416
(三) 模型压缩.....	417
1 · 剪枝.....	417
2 · 量化.....	419
(四) AutoML.....	420
1 · AutoKeras(开源).....	421
2 · NNI (Microsoft 开源)	423
3 · Cloud AutoML (Google)	426
4 · AutoGluon (Amazon)	427
5 · 算法：NAS.....	428
6 · 算法：特征工程.....	432
(五) 多示例学习	433
(六) 半监督学习	433
十三、硬件支持.....	434
(一) GPGPU.....	434
(二) OpenCL.....	434
(三) CUDA (NVIDIA)	435
(四) cuDNN (NVIDIA)	435
(五) TPU (Google)	435
(六) Linux 下 Nvidia/CUDA/cuDNN 的安装.....	435
1. Nvidia 驱动.....	436
2. CUDA.....	436
3. CuDNN.....	437
(七) Keras 中 GPU/CPU 的切换.....	438
十四、Nvidia GPU.....	439
(一) 图形渲染流水线.....	439
1 · 应用程序阶段.....	440
2 · 几何阶段.....	440
3 · 光栅化阶段 & 像素处理.....	443
(二) GPU 的构成.....	445
1 · 着色器.....	445
2 · 统一着色器/流处理器.....	446
3 · CUDA 核心.....	446

4 · 纹理单元.....	447
5 · 光栅单元.....	447
6 · 光线追踪核心.....	447
7 · 张量核心.....	447
8 · GPU 大核.....	447
(三) NVidia 微架构 (Microarchitecture).....	450
1 · Pascal 架构.....	450
2 · Volta 架构.....	450
3 · Turing 架构.....	450
4 · Ampere 架构.....	451
(四) NVidia 产品系列及型号.....	451
1 · GeForce.....	451
2 · Quadro.....	456
3 · Tesla.....	456
(五) API.....	457
1 · DirectX.....	457
2 · OpenGL.....	458
3 · OpenGL ES.....	459
4 · Mesa.....	459
5 · Vulkan.....	459
十五、边缘计算.....	460
(一) 达芬奇 NPU (华为) (TODO).....	460
(二) 寒武纪 (寒武纪)	460
(三) TensorRT (NVIDIA)	460
(四) ncnn (腾讯)	460
(五) TNN (腾讯)	461
(六) MNN (阿里)	461
(七) mace (小米)	461
(八) Paddle-Lite (百度)	461
(九) Google.....	461
1 · TensorFlow Lite (Google)	462
2 · NNAPI (Google)	467
(十) Facebook.....	468
1 · Pytorch Mobile (Facebook).....	468
2 · QNNPACK (Facebook).....	468
(十一) ARM.....	469
1 · 指令集/架构/核心.....	469
2 · big.LITTLE (ARM).....	474
3 · ARM NN (ARM)	474
4 · Neon(ARM).....	476
(十二) Vulkan.....	476
(十三) NNIE(TODO).....	476
(十四) AidLearning.....	476
十六、其它.....	478

(一) NN 上的工作.....	478
(二) 神经网络类型.....	479
1 · 多层感知机 (MLP)	479
2 · 卷积神经网络 (CNN)	479
3 · 递归神经网络 (RNN)	479
4 · 生成对抗网络 (GAN)	480
5 · 受限玻尔兹曼机 (RBM)	480
6 · 深度置信网络 (DBN)	481
(三) 神经网络可视化.....	481
1 · 模型可视化.....	482
2 · 训练可视化.....	483
3 · 卷积核/特征图可视化.....	483
(四) 相关学习资料.....	484
1 · 视频.....	484
2 · 课程.....	484
3 · 电子书.....	484
(五) 著名人士 (TODO).....	485
1 · Michael Jordan.....	485
2 · Bengio.....	485
3 · Hinton.....	485
4 · Yann LeCun.....	485
5 · 李飞飞.....	485
6 · 吴恩达.....	485
7 · Ian Goodfellow.....	485
8 · 汤晓鸥.....	485

一、概念&定义

这一章介绍了神经网络涉及到的数学及工程方面的（包括所用到的数学的各个分支，及神经网络的各个研究方向）的一些基本概念、定义、算法等等。

混淆注意！

首先来搞清楚几个基本的概念，这些概念之间的区别让很多人都挺迷惑。

NN（神经网络）、ANN（人工神经网络）、DNN（深度神经网络）、DL（深度学习）、ML（机器学习）、AI（人工智能）、MLP（多层感知机）、CNN（卷积神经网络）、RNN（循环神经网络）这些词之间是什么关系？

- $AI > ML > DL$

这些定义里，**AI** 是最宽泛的定义，而 **ML** 是 **AI** 的一种实现方式，其特点是不明确编码，而让计算机自行学习参数。**DL** 则是 **ML** 的子集，**DL** 指的是使用 **DNN** 方式实现的 **ML**，相对于普通 **ML** 的特征需要自己选择（即特征工程），**DL** 的特征无须手动选择（所谓端到端）。

- $DL = NN$

DL 和 **NN** 基本是同一个东西，区别在于强调不同方面，**DL** 是跟 **ML** 等概念在一个范围内的，而 **NN** 则主要强调是其实现的方式。

- $NN = ANN = DNN$

一般来说，**NN=ANN=DNN**，指的都是人工神经网络。

- $DNN > MLP/CNN/RNN$

MLP、**CNN** 和 **RNN** 则算是 **DNN** 的子类（严格来说应该算其要素特点），主要指代其网络中的全连接、卷积和循环特点。

（一）希腊字母列表

下表用于输入公式时复制：

字母大小写	名称	拉丁转写
A α	Alpha	a
B β	Beta	v
Γ γ	Gamma	g
Δ δ	Delta	d

E	ε	Epsilon	e
Z	ξ	Zeta	z
H	η	Eta	i
Θ	θ	Theta	th
I	ι	Iota	i
K	κ	Kappa	k
Λ	λ	Lambda	l
M	μ	Mu	m
N	ν	Nu	n
Ξ	ξ	Xi	x
O	\circ	Omicron	o
Π	π	Pi	p
P	ρ	Rho	r
Σ	σ	Sigma	s
T	τ	Tau	t
Υ	υ	Upsilon	i
Φ	ϕ	Phi	f
X	χ	Chi	ch
Ψ	ψ	Psi	ps
Ω	ω	Omega	o

(二) 微积分

1 · 导数和积分

假设函数 $f: \mathbb{R} \rightarrow \mathbb{R}$ 的输入和输出都是标量。函数 f 的导数为：

$$f'(x) = \lim_{h \rightarrow 0} \frac{f(x + h) - f(x)}{h},$$

且假定该极限存在。给定 $y = f(x)$ ，其中 x 和 y 分别是函数 f 的自变量和因变量。
以下有关导数和微分的表达式等价：

$$f'(x) = y' = \frac{dy}{dx} = \frac{df}{dx} = \frac{d}{dx}f(x) = Df(x) = D_x f(x),$$

其中符号 D 和 d/dx 也叫微分运算符。常见的微分演算有：

- $Dc=0$ (C 为常数)
- $Dx^n=nx^{n-1}$ (n 为常数)
- $De^x=e^x$
- $D\ln(x)=1/x$ 等。

如果函数 f 和 g 都可导，设 C 为常数，那么有以下法则：

$$\begin{aligned}\frac{d}{dx}[Cf(x)] &= C\frac{d}{dx}f(x), \\ \frac{d}{dx}[f(x) + g(x)] &= \frac{d}{dx}f(x) + \frac{d}{dx}g(x), \\ \frac{d}{dx}[f(x)g(x)] &= f(x)\frac{d}{dx}[g(x)] + g(x)\frac{d}{dx}[f(x)], \\ \frac{d}{dx}\left[\frac{f(x)}{g(x)}\right] &= \frac{g(x)\frac{d}{dx}[f(x)] - f(x)\frac{d}{dx}[g(x)]}{[g(x)]^2}.\end{aligned}$$

2 · 链式法则

如果 $y=f(u)$ 和 $u=g(x)$ 都是可导函数，那么有链式法则：

$$\frac{dy}{dx} = \frac{dy}{du} \cdot \frac{du}{dx}$$

3 · 偏导数

设 u 为一个有 n 个自变量的函数， $u=f(x_1, x_2, \dots, x_n)$ ，它有关第 i 个变量 x_i 的偏导数为：

$$\frac{\partial u}{\partial x_i} = \lim_{h \rightarrow 0} \frac{f(x_1, \dots, x_{i-1}, x_i + h, x_{i+1}, \dots, x_n) - f(x_1, \dots, x_i, \dots, x_n)}{h}.$$

以下有关偏导数的表达式等价：

$$\frac{\partial u}{\partial x_i} = \frac{\partial f}{\partial x_i} = f_{x_i} = f_i = D_i f = D_{x_i} f.$$

为了计算 $\frac{\partial u}{\partial x_i}$ ，只需将 $x_1, \dots, x_{i-1}, x_{i+1}, \dots, x_n$ 视为常数并求 u 有关 x_i 的导数。

4 · 梯度 (Gradient)

假设函数 $f: \mathbb{R}^n \rightarrow \mathbb{R}$ 的输入是一个 n 维向量 $x = [x_1, x_2, \dots, x_n]^\top$ ，输出是标量。函数 $f(x)$ 有关 x 的梯度 (Gradient) 是一个由 n 个偏导数组成的向量：

$$\nabla_x f(x) = \left[\frac{\partial f(x)}{\partial x_1}, \frac{\partial f(x)}{\partial x_2}, \dots, \frac{\partial f(x)}{\partial x_n} \right]^\top$$

为表示简洁，我们有时用 $\nabla f(x)$ 代替 $\nabla_x f(x)$ 。假设 x 是一个向量，常见的梯度演算包括：

$$\begin{aligned}\nabla_x A^\top x &= A, \\ \nabla_x x^\top A &= A, \\ \nabla_x x^\top A x &= (A + A^\top)x, \\ \nabla_x \|x\|^2 &= \nabla_x x^\top x = 2x.\end{aligned}$$

类似地，假设 X 是一个矩阵，那么：

$$\nabla_X \|X\|_F^2 = 2X.$$

(三) 线性代数/几何学

1 · 张量/标量/向量/矩阵

张量指的是 N 轴 (axis) 的数据。轴的个数也被称为阶 (order)。

标量 (Scalar) : 0 阶的张量，只有一个数字
向量 (Vector) : 1 阶的张量，也叫矢量，一个数组
矩阵 (Matrix) : 2 阶的张量，2 个轴通常称为行和列。

混淆注意！

维 (dimension) 通常用来描述向量，比如 N 维向量，比如“128 维特征值”。这里的维和张量的轴定义不一样，这种 N 维向量描述的仍然是向量（即 1 阶张量），其中的 N 描述的是数组的元素个数。

但是由于 2 阶张量（即矩阵）形似 2 维平面，3 阶张量形似 3 维空间，而 N 阶张量又可以用 N 维数组来表示，导致维经常被拿来形容张量的阶。为了避免混淆，尽量不要用维来描述张量的阶。

我们以 numpy.ndarray 类型的变量 t 的形状 (t.shape) 来说明：

- (1,) : 标量，0 阶张量，1 维向量，只有 1 个数值
- (3,) : 向量，1 阶张量，3 维向量，有 3 个数值
- (5,) : 向量，1 阶张量，5 维向量，有 5 个数值
- (2,3) : 矩阵，2 阶张量，有 6 个数值， $t[0]$ 为 3 维向量
- (3,1) : 矩阵，2 阶张量，有 3 个值， $t[0]$ 为 1 维向量
- (3,4,5) : 3 阶张量，有 60 个数值， $t[0,0]$ 为 5 维向量， $t[0]$ 为矩阵

2 · 特征向量/特征值

对于一个 n 行 n 列的矩阵 A，假设有标量 λ 和非零的 n 维向量 v 使 $Av = \lambda v$ 。
那么 v 是矩阵 A 的一个特征向量，标量 λ 是 v 对应的特征值。

3 · 逐点操作

Pointwise(elementwise) operation of matrix/vector，指的是形状一致的矩阵/向量对应的每个元素进行操作（加、乘等）。

比如逐点乘法：

```
a = [[ 1.  2.  3.]
      [ 10. 20. 30.]]
b = [[ 2.  2.  2.]
      [ 3.  3.  3.]]
c = a*b

c == [[ 20. 40. 60.]
      [ 30. 60. 90.]]
```

4 · 欧式距离

欧式距离 (Euclidean distance) 指的是欧式空间中的两个点之间的距离：

在二维空间 (平面) 中为： $\rho = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$

在三维空间中的距离为： $\rho = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2 + (z_2 - z_1)^2}$,

在 N 维空间中的距离为： $d(x, y) := \sqrt{(x_1 - y_1)^2 + (x_2 - y_2)^2 + \dots + (x_n - y_n)^2} = \sqrt{\sum_{i=1}^n (x_i - y_i)^2}$.

在神经网络中，两个 N 维特征值之间的差距可以用欧式距离来衡量。

欧式边缘 (Euclidean Margin) 指的是两个不同类别间的在欧式空间中的间隔区，在分类问题中，欧式边缘越大，被认为特征判别度越高，在面对 **unseen** 数据（未见过的数据）时候的鲁棒性越好。

5 · 余弦距离 (余弦相似度)

余弦距离 (Cosine distance)，是用向量空间中两个向量夹角的余弦值作为衡量两个个体间差异的大小的度量。

余弦距离来自余弦相似度，余弦相似度取值范围为 $[-1, 1]$ ，向量夹角为 0 的情况下取 1，夹角为 180 度时取 -1。余弦相似度定义如下，其中 a, b 分别为两个向量：

$$\cos \Theta = \frac{a * b}{|a| * |b|}$$

在二维空间中，展开为：

$$\cos \Theta = \frac{x_1 * x_2 + y_1 * y_2}{\sqrt{x_1^2 + y_1^2} * \sqrt{x_2^2 + y_2^2}}$$

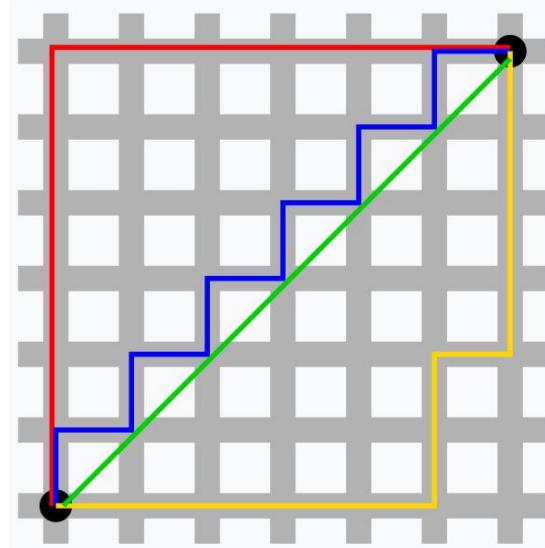
余弦距离的定义则为： $dist(A, B) = 1 - \cos \theta$ ，取值范围为 $[0, 2]$

6 · 曼哈顿距离

Manhattan distance，表示两个点在标准坐标系上的绝对轴距之总和。公式为：

$$dis = |x_1 - x_2| + |y_1 - y_2|$$

下图中绿色为欧式距离（长度约为 8.48），红色、蓝色和绿色表示的曼哈顿距离长度一样，均为 12。



7 · 闵可夫斯基距离

闵可夫斯基距离，Minkowski distance，两个向量（点）的 p 阶距离。

公式为： $d_{Minkowski} = (\|\mathbf{x} - \mathbf{y}\|^p)^{1/p}$

当 $p=1$ 时就是曼哈顿距离， $p=2$ 时是欧式距离。

8 · 范数

范数 (norm)，对于向量来说，其公式为：

$$\|\mathbf{x}\|_p = \sqrt[p]{\sum_i |x_i|^p}$$

- 当 p 为 0 时，被称为 L0 范数，表示向量 \mathbf{x} 中非 0 元素的个数。
- 当 p 为 1 时，被称为 L1 范数，即为各个元素绝对值之和。对应到原点的曼哈顿距离。

- 当 p 为 2 时，被称为 L2 范数或者欧式范数，若 x 为 N 维向量，其 L2 范数为 x 到其所在 N 维空间原点的距离。对应到原点的欧式距离，通常 $\|x\|$ 指代 L2 范数

对于矩阵来说，设 X 是一个 m 行 n 列矩阵。矩阵 X 的 Frobenius 范数为该矩阵元素平方和的平方根：

$$\|X\|_F = \sqrt{\sum_{i=1}^m \sum_{j=1}^n x_{ij}^2},$$

其中 x_{ij} 为矩阵 X 在第 i 行第 j 列的元素。

9 · 齐次坐标

齐次坐标（homogeneous coordinates），或投影坐标（projective coordinates）是指一个用于投影几何里的坐标系统，如同用于欧氏几何里的笛卡儿坐标一般。齐次坐标可让包括无穷远点的点坐标以有限坐标表示。使用齐次坐标的公式通常会比用笛卡儿坐标表示更为简单，且更为对称。齐次坐标有着广泛的应用，包括电脑图形及 3D 电脑视觉。使用齐次坐标可让电脑进行仿射变换，并通常，其投影变换能简单地使用矩阵来表示。

如一个点的齐次坐标乘上一个非零标量，则所得之坐标会表示同一个点。因为齐次坐标也用来表示无穷远点，为这一扩展而需用来标示坐标之数值就比投影空间之维度多一。例如，在齐次坐标里，需要两个值来表示在投影线上的一点，需要三个值来表示投影平面上的一点。

比如在欧式空间中表示一个 2D 的点需要 2 个值，以下为矩阵形式：

$$p = \begin{bmatrix} x \\ y \end{bmatrix}$$

而对于点的表示，齐次坐标系加上了一个维度，值为 1，并且乘以一个系数 k_p ，如下：

$$p = k_p \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} k_p x \\ k_p y \\ k_p \end{bmatrix}$$

对于直线的表示，欧式空间中的笛卡尔坐标表示为：

$$y = ax + b$$

或者

$$ax + by + c = 0$$

而用齐次坐标系来表示，则为

$$1 = \begin{bmatrix} a \\ b \\ c \end{bmatrix} = k_1 \begin{bmatrix} m \\ -1 \\ g \end{bmatrix} = \begin{bmatrix} k_1 m \\ -k_1 \\ k_1 g \end{bmatrix}$$

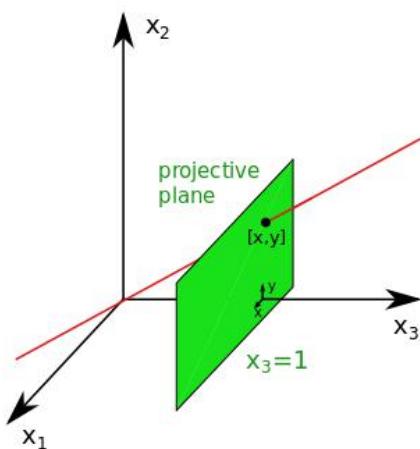
实投影平面可以看作是一个具有额外点的欧氏平面，这些点称之为无穷远点，并被认为是在一条新的线上（该线称之为无穷远线）。每一个无穷远点对应至一个方向（由一条线之斜率给出），可非正式地定义为一个点自原点朝该方向移动之极限。在欧氏平面里的平行线可看成会在对应其共同方向之无穷远点上相交。给定欧氏平面上的一点 (x, y) ，对任意非零实数 Z ，三元组 (xZ, yZ, Z) 即称之为该点的齐次坐标。依据定义，将齐次坐标内的数值乘上同一个非零实数，可得到同一点的另一组齐次坐标。例如，笛卡儿坐标上的点 $(1, 2)$ 在齐次坐标中即可标示成 $(1, 2, 1)$ 或 $(2, 4, 2)$ 。原来的笛卡儿坐标可透过将前两个数值除以第三个数值取回。因此，与笛卡儿坐标不同，一个点可以有无限多个齐次坐标表示法。

一条通过原点 $(0, 0)$ 的线之方程可写作 $nx + my = 0$ ，其中 n 及 m 不能同时为 0。以参数表示，则能写成 $x = mt$, $y = -nt$ 。令 $Z=1/t$ ，则线上的点之笛卡儿坐标可写作 $(m/Z, -n/Z)$ 。在齐次坐标下，则写成 $(m, -n, Z)$ 。当 t 趋向无限大，亦即点远离原点时， Z 会趋近于 0，而该点的齐次坐标则会变成 $(m, -n, 0)$ 。因此，可定义 $(m, -n, 0)$ 为对应 $nx + my = 0$ 这条线之方向的无穷远点之齐次坐标。因为欧氏平面上的每条线都会与透过原点的某一条线平行，且因为平行线会有相同的无穷远点，欧氏平面每条线上的无穷远点都有其齐次坐标。

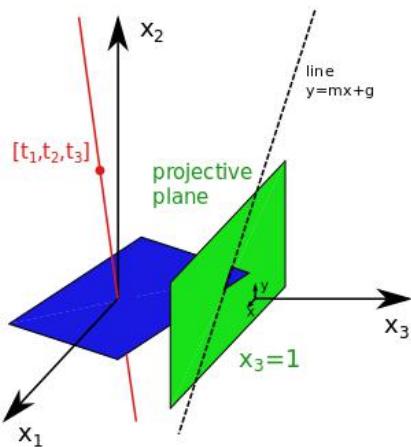
概括来说：

- 投影平面上的任何点都可以表示成一三元组 (X, Y, Z) ，称之为该点的齐次坐标或投影坐标，其中 X 、 Y 及 Z 不全为 0。
- 以齐次坐标表示的点，若该坐标内的数值全乘上一相同非零实数，仍会表示该点。
- 相反地，两个齐次坐标表示同一点，当且仅当其中一个齐次坐标可由另一个齐次坐标乘上一相同非零常数得取得。
- 当 Z 不为 0，则该点表示欧氏平面上的该 $(X/Z, Y/Z)$ 。
- 当 Z 为 0，则该点表示一无穷远点。

注意，三元组 $(0, 0, 0)$ 不表示任何点。原点表示为 $(0, 0, 1)$



齐次坐标系中的点（红色线）在投影平面（绿色， $x_3=1$ ）上的投影（交点）



齐次坐标系中的直线（红线 (t_1, t_2, t_3) 及其对应的蓝色平面 $t_1x_1+t_2x_2+t_3x_3=0$ ）在投影平面（绿色， $x_3=1$ ）上的投影（交叉线）

参考：

[维基百科](#)

<https://mc.ai/part-i-projective-geometry-in-2d/>

10 · 仿射变换 (Affine Transformation)

仿射变换，又称仿射映射，是指在几何中，对一个向量空间进行一次线性变换并接上一个平移，变换为另一个向量空间。

一个对向量 \vec{x} 平移 \vec{b} ，与旋转放大缩小 A 的仿射映射为

$$\vec{y} = A\vec{x} + \vec{b}$$

上式在齐次坐标上，等价于下面的式子

$$\begin{bmatrix} \vec{y} \\ 1 \end{bmatrix} = \begin{bmatrix} A & \vec{b} \\ 0, \dots, 0 & 1 \end{bmatrix} \begin{bmatrix} \vec{x} \\ 1 \end{bmatrix}$$

（四）概率论

1 · 概率

概率（Probability）是概率论的基本概念，是一个在 0 到 1 之间的实数，是对随机发生事件之可能性的度量。。

事件 A 发生的概率通常用 $P(A)$ 表示。

2 · 条件概率

假设事件 A 和事件 B 的概率分别为 $P(A)$ 和 $P(B)$ ，两个事件同时发生的概率记作 $P(A \cap B)$ 或 $P(A, B)$ 。给定事件 B，事件 A 的条件概率（后验概率）：

$$P(A | B) = \frac{P(A \cap B)}{P(B)}.$$

也就是说，

$$P(A \cap B) = P(B) \cdot P(A|B) = P(A) \cdot P(B|A)$$

当满足

$$P(A \cap B) = P(A) \cdot P(B)$$

时，事件 A 和事件 B 相互独立。

3 · 贝叶斯定理

贝叶斯定理（Bayes' theorem）是概率论中的一个定理，描述在已知一些条件下，某事件的发生概率。

$$P(A | B) = \frac{P(B | A)P(A)}{P(B)}$$

其中 A 以及 B 为随机事件，且 $P(B)$ 不为零。 $P(A|B)$ 是指在事件 B 发生的情况下事件 A 发生的概率。

在贝叶斯定理中，每个名词都有约定俗成的名称：

- $P(A|B)$ 是已知 B 发生后，A 的条件概率。由于得自 B 的取值也叫 A 的后验概率。
- $P(A)$ 是 A 的先验概率（或边缘概率）。之所以称为“先验”是因为它不考虑任何 B 方面的因素。
- $P(B|A)$ 是已知 A 发生后，B 的条件概率。由于得自 A 的取值也叫 B 的后验概率。

- $P(B)$ 是 B 的先验概率。

按这些术语，贝叶斯定理可表述为：

$$\text{后验概率} = (\text{似然性} * \text{先验概率}) / \text{标准化常量}$$

也就是说，后验概率与先验概率和相似度的乘积成正比。

另外，比例 $P(B|A)/P(B)$ 也有时被称作标准似然度（standardised likelihood），贝叶斯定理可表述为：

$$\text{后验概率} = \text{标准似然度} * \text{先验概率}$$

参考：

<https://zh.wikipedia.org/wiki/%E8%B4%9D%E5%8F%B6%E6%96%AF%E5%AE%9A%E7%90%86>

4 · 期望值 (EV)

离散的随机变量 X 的期望（Expected Value，EV，数学期望、期望值、平均值）为

$$E(X) = \sum_x x P(X = x).$$

5 · 概率密度函数 (PDF)

PDF，Probability Density Function，用于描述一个连续随机变量的输出值，在某个确定的取值点附近的可能性的函数。

概率密度函数是对连续随机变量定义的，本身不是概率，只有对连续随机变量的概率密度函数在某区间内进行积分后才是概率。

6 · 累积分布函数 (CDF)

累积分布函数 (CDF，Cumulative Distribution Function) 是概率密度函数的积分：

$$F_X(x) = P(X \leq x)$$

累积分布函数有界（介于 0 和 1 之间），单调增加，且右连续。

维基百科：

<https://zh.wikipedia.org/wiki/%E7%B4%AF%E7%A7%AF%E5%88%86%E5%B8%83%E5%87%BD%E6%95%BD>

7 · 概率质量函数

概率质量函数（Probability Mass Function），类似概率密度函数，用于描述一个离散随机变量在各特定取值上的概率。

8 · 连续分级概率评分（CRPS）

Continuous Rank Probability Score，可以量化一个连续概率分布（理论值）和观测样本（真实值）之间的差异。可以作为概率模型的损失函数和评价函数。

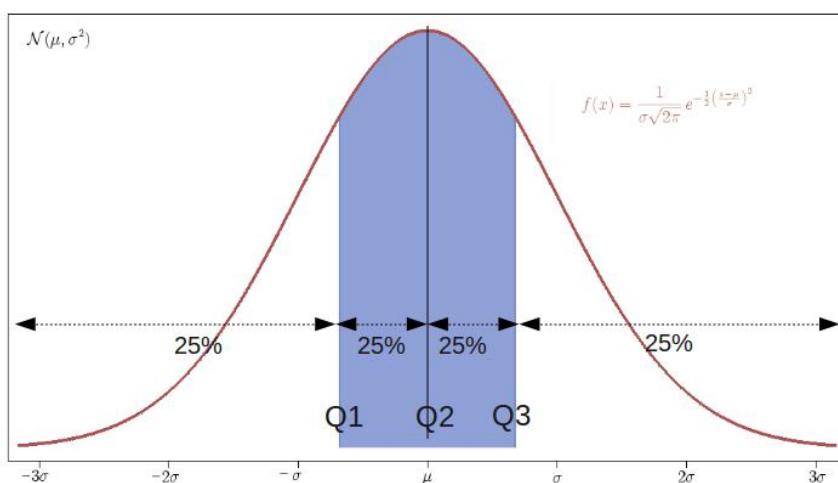
CRPS 在数学形式上是累积分布函数 CDF 与阶跃函数（Heaviside step function）之差的平方在实数域的积分，因此可视为平均绝对误差（Mean Absolute Error, MAE）在连续概率分布上的推广。

参考：

<https://www.lokad.com/continuous-ranked-probability-score>

9 · 分位数（Quantile）

分位数指的是在概率分布中（概率密度函数）的分割点，这些分割点使得相邻分割点形成的区间的概率分布一致，下图中正态分布的三个 Quantile (Q1, Q2, Q3) 使得整个正态分布被分为 4 个概率相同的部分（均为 25%）



参考：

<https://en.wikipedia.org/wiki/Quantile>

10 · 独立

在概率论里，说两个事件是独立的，直觉上是指一次实验中一事件的发生不会影响到另一事件发生的概率。例如，在一般情况下可以认为连续两次掷骰子得到的点数结果是相互独立的。类似地，两个随机变量是独立的，若其在一事件给定观测量的条件概率分布和另一事件没有被观测的概率分布是一样的。

标准的定义为：

两个事件 A 和 B 是独立的，当且仅当 $P(A \cap B) = P(A) \cdot P(B)$ 。

参考：

[维基百科](#)

11 · 独立同分布

在概率论与统计学中，**独立同分布**（Independent and identically distributed，缩写为 **IID**）是指一组随机变量中每个变量的概率分布都相同，且这些随机变量互相独立。

一组随机变量独立同分布并不意味着它们的样本空间中每个事件发生概率都相同。例如，投掷非均匀骰子得到的结果序列是独立同分布的，但掷出每个面朝上的概率并不相同。

12 · 联合分布

在概率论中，对两个随机变量 X 和 Y，其**联合分布**是同时对于 X 和 Y 的概率分布。

对离散随机变量而言，联合分布概率质量函数为 $\Pr(X = x \& Y = y)$ ，即：

$$P(X = x \text{ and } Y = y) = P(Y = y | X = x)P(X = x) = P(X = x | Y = y)P(Y = y).$$

因为是概率分布函数，所以必须有

$$\sum_x \sum_y P(X = x \text{ and } Y = y) = 1.$$

类似地，对连续随机变量而言，联合分布概率密度函数为 $f_{X,Y}(x, y)$ ，其中 $f_{Y|X}(y|x)$ 和 $f_{X|Y}(x|y)$ 分别代表 $X = x$ 时 Y 的条件分布以及 $Y = y$ 时 X 的条件分布； $f_X(x)$ 和 $f_Y(y)$ 分别代表 X 和 Y 的边缘分布。

同样地，因为是概率分布函数，所以必须有

$$\int_x \int_y f_{X,Y}(x, y) dy dx = 1.$$

13 · 伯努利分布 (0-1 分布)

伯努利分布 (Bernoulli distribution, 又名两点分布或者 0-1 分布) , 是一个离散型概率分布。若伯努利试验成功, 则伯努利随机变量取值为 1。若伯努利试验失败, 则伯努利随机变量取值为 0。记其成功概率为 $p(0 \leq p \leq 1)$, 失败概率为 $q = 1 - p$ 。

其概率质量函数为 :

$$f_X(x) = p^x(1-p)^{1-x} = \begin{cases} p & \text{if } x = 1, \\ q & \text{if } x = 0. \end{cases}$$

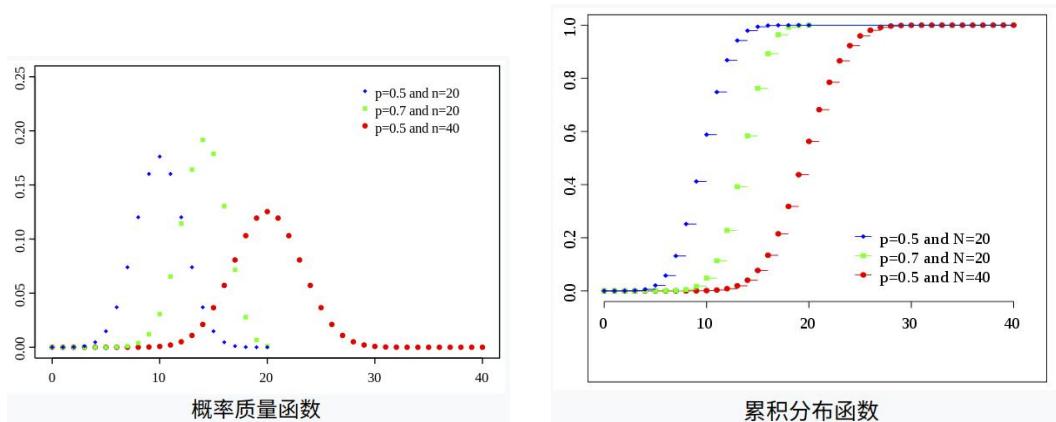
14 · 二项分布

n 次独立重复的伯努利实验中, 设每次试验中事件 A 发生的概率为 p 。用 X 表示 n 重伯努利试验中事件 A 发生的次数, 则 X 的可能取值为 $0, 1, \dots, n$, 且对每一个 k ($0 \leq k \leq n$), 事件 $\{X=k\}$ 即为 “ n 次试验中事件 A 恰好发生 k 次”, 随机变量 X 的离散概率分布即为二项分布 (Binomial Distribution)。

二项分布的概率质量函数为 :

$$P(k) = C_n^k \cdot p^k (1-p)^{n-k}$$

二项分布的期望 $E(X)=np$, 方差 $D(X)= np(1-p)$



15 · 泊松分布

Poisson Distribution 是一种统计与概率学里常见的离散概率分布。

泊松分布适合于描述单位时间内随机事件发生的次数的概率分布。如某一服务设施在一定时间内受到的服务请求的次数，汽车站台的候客人数、机器出现的故障数、自然灾害发生的次数、DNA 序列的变异数、放射性原子核的衰变数、激光的光子数分布等等。

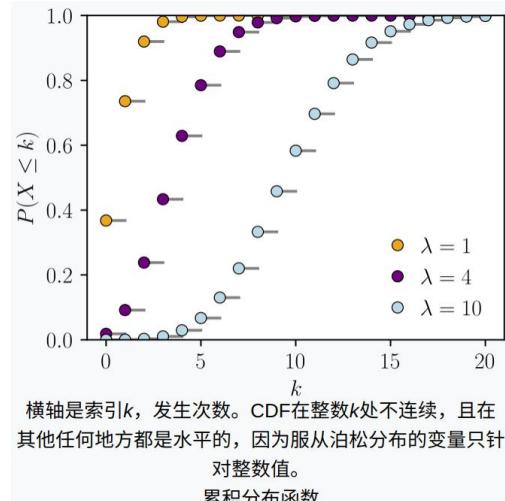
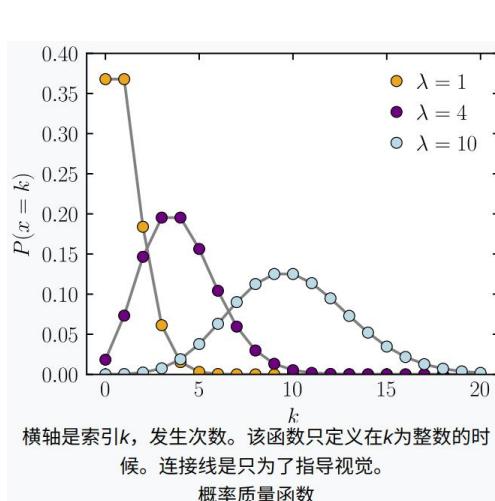
泊松分布的概率质量函数为：

$$P(X = k) = \frac{e^{-\lambda} \lambda^k}{k!}$$

泊松分布的参数 λ 是单位时间（或单位面积）内随机事件的平均发生率。

泊松分布具有如下特性：

- 服从泊松分布的随机变量，其数学期望与方差相等，同为参数 $E(X) = V(X) = \lambda$
- 两个独立且服从泊松分布的随机变量，其和仍然服从泊松分布。更精确地说，若 $X \sim \text{Poisson}(\lambda_1)$ ，且 $Y \sim \text{Poisson}(\lambda_2)$ ，则 $X + Y \sim \text{Poisson}(\lambda_1 + \lambda_2)$



16 · 指数分布

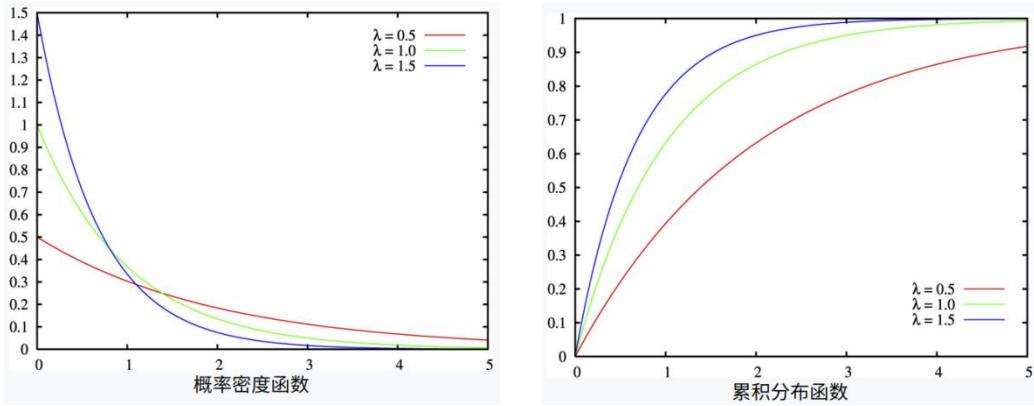
指数分布 (Exponential distribution) 是一种连续概率分布。指数分布可以用来表示独立随机事件发生的时间间隔，比如旅客进入机场的时间间隔、打进客服中心电话的时间间隔、中文维基百科新条目出现的时间间隔等等。

其 PDF (概率密度函数) 为：

$$f(x; \lambda) = \begin{cases} \lambda e^{-\lambda x}, & x \geq 0, \\ 0, & x < 0. \end{cases}$$

CDF (累积分布函数) 为：

$$F(x; \lambda) = \begin{cases} 1 - e^{-\lambda x}, & x \geq 0, \\ 0, & x < 0. \end{cases}$$



维基百科

17 · 伽玛分布

Gamma Distribution，是一个连续概率函数，有两个参数：

- α ：形状参数
- β ：尺度参数。

假设 X_1, X_2, \dots, X_n 为连续发生事件的等候时间，且这 n 次等候时间为独立的，那么这 n 次等候时间之和 Y ($Y=X_1+X_2+\dots+X_n$) 服从 **伽玛分布**，即 $Y \sim \text{Gamma}(\alpha, \beta)$ ，其中 $\alpha = n$, $\beta = \lambda$ 。这里的 λ 是连续发生事件的平均发生频率。**指数分布**是**伽玛分布** $\alpha = 1$ 的特殊情况。

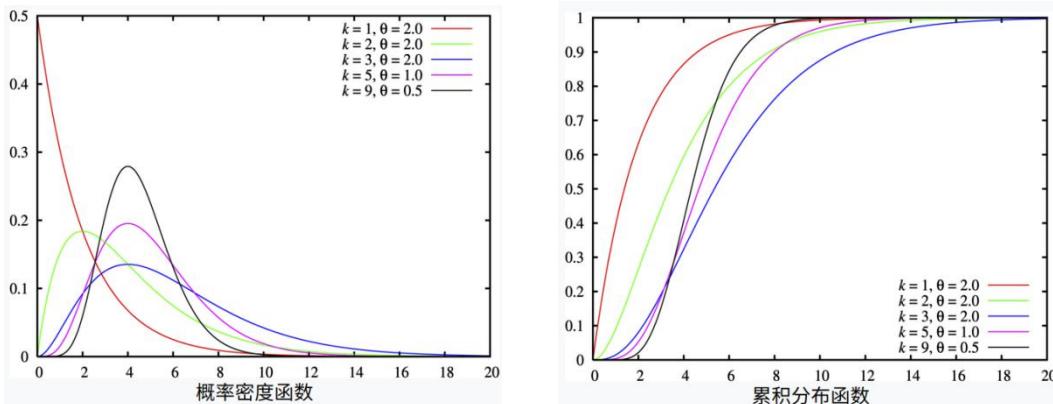
其概率密度函数为：

令 $X \sim \Gamma(\alpha, \beta)$, 且令 $\lambda = \beta$ (即 $X \sim \Gamma(\alpha, \lambda)$) , 则:

$$f(x) = \frac{x^{(\alpha-1)} \lambda^\alpha e^{(-\lambda x)}}{\Gamma(\alpha)}, \quad x > 0$$

其中**Gamma函数**之特征为：

$$\begin{cases} \Gamma(\alpha) = (\alpha - 1)! & \text{if } \alpha \text{ is } \mathbb{Z}^+ \\ \Gamma(\alpha) = (\alpha - 1)\Gamma(\alpha - 1) & \text{if } \alpha \text{ is } \mathbb{R}^+ \\ \Gamma(\frac{1}{2}) = \sqrt{\pi} & \end{cases}$$



18 · 正态分布（高斯分布）

正态分布（normal distribution）又名高斯分布（Gaussian distribution），是一个非常常见的连续概率分布。正态分布在统计学上十分重要，经常用在自然和社会科学来代表一个不明的随机变量。

若随机变量 X 服从一个位置参数为 μ 、尺度参数为 σ 的正态分布，记为：

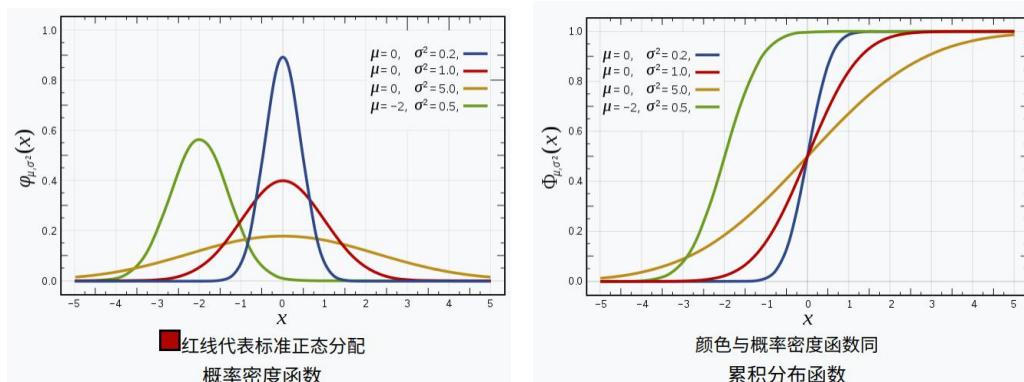
$$X \sim N(\mu, \sigma^2),$$

则其概率密度函数为：

$$f(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$

正态分布的期望值（EV）或者平均值（mean）等于位置参数 μ ，决定了分布的位置；其方差 σ^2 的开平方或标准差等于尺度参数 σ ，决定了分布的幅度。

正态分布的概率密度函数曲线呈钟形，因此人们又经常称之为钟形曲线（类似于寺庙里的大钟，因此得名）。我们通常所说的标准正态分布是位置参数 $\mu=0$ ，尺度参数 $\sigma^2=1$ 的正态分布。



19 · 学生 t 分布

Student's t-distribution，可简称为 t 分布，常被用于根据小样本来估计呈正态分布且方差未知的总体的平均值。如果总体方差已知（例如在样本数量足够多时），则应该用正态分布来估计总体均值。

t 分布并不是仅仅用于小样本（虽然小样本中用的风生水起）中，大样本依旧可以使用。t 分布与正态分布相比多了自由度参数，在小样本中，能够更好的剔除异常值对于小样本的影响，从而能够准确的抓住数据的集中趋势和离散趋势。

t 分布是如何产生的：

假设 X_1, \dots, X_n 为是正态分布（期望值为 μ 、方差为 σ^2 ）的独立的随机采样，令：

$$\bar{X} = \frac{1}{n} \sum_{i=1}^n X_i$$

为样本均值，而令

$$S^2 = \frac{1}{n-1} \sum_{i=1}^n (X_i - \bar{X})^2$$

为贝塞尔校正（Bessel's Correction）后的样本方差。（[为什么其分母是 n-1？](#)）

那么随机变量

$$\frac{\bar{X} - \mu}{\sigma / \sqrt{n}}$$

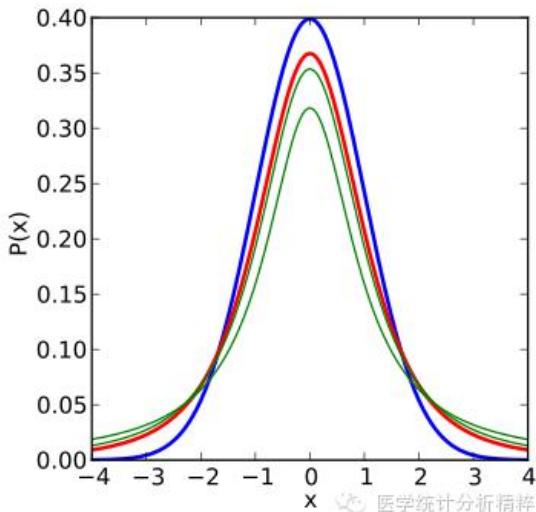
服从标准正太分布。并且随机变量

$$\frac{\bar{X} - \mu}{S / \sqrt{n}},$$

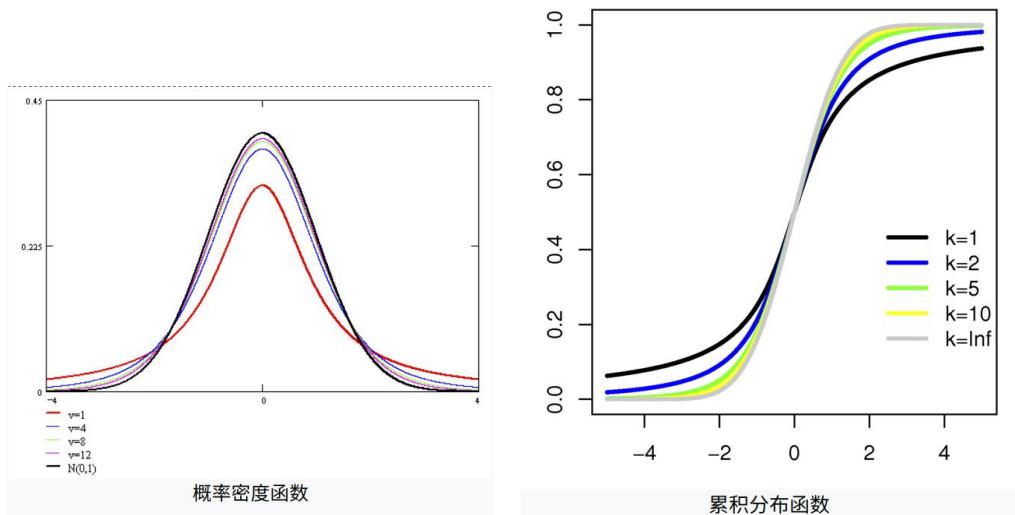
服从自由度为 $n-1$ 的学生 T 分布，其概率密度函数为：

$$f(t) = \frac{\Gamma(\frac{\nu+1}{2})}{\sqrt{\nu\pi} \Gamma(\frac{\nu}{2})} \left(1 + \frac{t^2}{\nu}\right)^{-\frac{\nu+1}{2}}$$

其中 $\nu = n-1$ ，表示自由度，T 分布相较于正态分布有“厚尾（heavy tail）”的特点，表现在 PDF 函数曲线中即峰低尾高，自由度越高，越接近正态分布。当为无穷大时，等价于正态分布，当 $\nu=1$ 的时候，就退化成了柯西分布，下图中蓝色为正态分布，红色为 $\nu=3$ 的 T 分布，绿色为 $\nu=1$ 和 2 的 T 分布。



下图为 T 分布的 PDF 和 CDF



[维基百科](#)

参考：

<https://zhuanlan.zhihu.com/p/42136925>

<https://www.zhihu.com/question/20099757/answer/658048814>

<https://www.cnblogs.com/think-and-do/p/6509239.html>

20 · 多元正态分布

Multivariate Normal Distribution，也叫多变量高斯分布（Multivariate Gaussian Distribution），指的是 N 维随机变量 $X = [X_1, \dots, X_N]^T$ 满足下列等价条件（即任意一个）：

- 任何线性组合 $Y = a_1 X_1 + \dots + a_N X_N$ 服从正态分布
- 存在随机变量 $Z = [Z_1, \dots, Z_M]^T$ （它的每个元素服从独立标准正态分布），向量 $u = [u_1, \dots, u_N]^T$ 及 $N \times M$ 矩阵 A ，满足 $X = AZ + u$

(五) 信息论

1 · 信息熵 (Entropy)

即信息论中的熵（entropy），是接收的每条消息中包含的信息的平均量，又被称为信息熵、香农熵、信源熵、平均自信息量。这里，“消息”代表来自分布或数据流中的事件、样本或特征。其定义为：

$$S = - \sum_i P_i \log P_i$$

其中 P_i 为每种情况发生的概率，各种情况的概率之和为 1，比如，一件事情发生的三种情况的概率分别为 $1/2, 1/4, 1/4$ ，则其信息熵为（以 2 为底）， $1/2 * \log_2 + 1/4 * \log_4 + 1/4 * \log_4 = 0.5 + 0.5 + 0.5 = 1.5$ 。

从定义中可以看出，越小概率的事情，其信息熵越高，如果某事件只有一种情况（此时其概率为 100%，即 1），则信息熵为 $1 * \log_1 = 0$ 。举个例子，如果有人告诉你“太阳今天从东方升起”那么你没有获得任何信息，因为这必然发生，这话没有任何信息量，等于没说。

熵最好理解为不确定性的量度而不是确定性的量度，因为越随机的信源的熵越大。来自信源的另一个特征是样本的概率分布。这里的想法是，比较不可能发生的事情，当它发生了，会提供更多的信息。由于一些其他的原因，把信息（熵）定义为概率分布的对数的相反数是有道理的。事件的概率分布和每个事件的信息量构成了一个随机变量，这个随机变量的均值（即期望）就是这个分布产生的信息量的平均值（即熵）。熵的单位通常为比特，但也用 Sh、nat、Hart 计量，取决于定义用到对数的底。

信息熵公式的来源：

<https://zhuanlan.zhihu.com/p/26486223>

2 · 联合熵 (Joint entropy)

联合熵用于衡量若干变量的不确定性，两个变量 X 和 Y 的联合信息熵定义为：

$$H(X, Y) = - \sum_x \sum_y P(x, y) \log_2 [P(x, y)]$$

$P(x, y)$ 为 X 和 Y 分别取值 x, y 的概率。

- 变量的联合熵大于等于变量中任何一个的独立熵
- 变量的联合熵小于等于变量的独立熵之和

3 · 条件熵 (Conditional entropy)

条件熵描述了在已知第二个随机变量 X 的值的前提下，随机变量 Y 的信息熵还有多少。同其它的信息熵一样，条件熵也用 Sh、nat、Hart 等信息单位表示。

基于 X 条件的 Y 的信息熵，用 $H(Y|X)$ 表示。

- 条件熵的链式法则：

条件熵 $H(Y|X)$ 等于联合熵 $H(Y, X)$ 减去独立熵 $H(X)$ ：

$$H(Y|X) = H(X, Y) - H(X)$$

- 条件熵的贝叶斯规则：

$$H(Y|X) = H(X|Y) - H(X) + H(Y)$$

$$H(Y|X) = H(X, Y) - H(X)$$

$$H(X|Y) = H(X, Y) - H(Y)$$

上式减下式，可得

$$H(Y|X) - H(X|Y) = H(Y) - H(X)$$

得：

$$H(Y|X) = H(X|Y) - H(X) + H(Y)$$

4 · 相对熵 (Relative entropy)

也叫 **KL 散度** (Kullback-Leibler Divergence)，是两个概率分布 P 和 Q 差别的非对称性的度量。KL 散度是用来度量使用基于 Q 的分布来编码服从 P 的分布的样本所需的额外的平均比特数。典型情况下，P 表示数据的真实分布，Q 表示数据的理论分布、估计的模型分布、或 P 的近似分布。

对于离散随机变量，其概率分布 P 和 Q 的 KL 散度可按下式定义为

$$D_{\text{KL}}(P\|Q) = - \sum_i P(i) \ln \frac{Q(i)}{P(i)}$$

5 · 交叉熵 (Cross entropy)

在信息论中，基于相同事件测度的两个概率分布 p 和 q 的交叉熵 (Cross entropy) 是指，当基于一个“非自然”（相对于“真实”分布 p 而言）的概率分布 q 进行编码时，在事件集合中唯一标识一个事件所需要的平均比特数 (bit)。基于概率分布 p 和 q 的交叉熵定义为：

$$H(p, q) = E_p[-\log q] = H(p) + D_{\text{KL}}(p\|q),$$

其中 $H(p)$ 为 p 的熵，也就是说交叉熵永远的大于等于真实分布的信息熵， $D_{\text{KL}}(p\|q)$ 被称为 p 到 q 的相对熵。

对于一个离散分布的 p 和 q，交叉熵为：

$$H(p, q) = - \sum_x p(x) \log q(x).$$

例如，一件事情发生的三种情况的真实分布分别为 $1/2, 1/4, 1/4$ ，预测分布则为 $1/4, 1/4, 1/2$ ，
则交叉熵为（以 2 为底）： $1/2 \log 4 + 1/4 \log 4 + 1/4 \log 2 = 1 + 0.5 + 0.25 = 1.75$ 。

(六) 统计学

1 · 方差 (Variance)

方差，**Variance**，在概率论和统计学中，一个随机变量的方差描述的是它的离散程度，也就是该变量离其期望值的距离。一个实随机变量的方差也称为它的二阶矩或二阶中心动差，恰巧也是它的二阶累积量。

简单来说，就是将各个误差将之平方（而非取绝对值，使之肯定为正数），相加之后再除以总数，透过这样的方式来算出各个数据分布、零散（相对中心点）的程度。继续延伸的话，方差的正平方根称为该随机变量的**标准差**（此为相对各个数据点间）。

$$\text{Var}(X) = \sigma^2 = \frac{1}{n} \sum_{i=1}^n (x_i - \mu)^2 = \frac{1}{n} \left(\sum_{i=1}^n x_i^2 - n\mu^2 \right) = \frac{\sum_{i=1}^n x_i^2}{n} - \mu^2$$

设 X 为服从分布 F 的随机变量，如果 $E[X]$ 是随机变数 X 的期望值（平均数 $\mu = E[X]$ ），随机变量 X 或者分布 F 的方差为： $\text{Var}(X) = E[(X - \mu)^2]$

2 · 标准差 (Standard Deviation)

标准差（**标准偏差、均方差**，**Standard Deviation, SD**），数学符号 σ （**sigma**），在概率统计中最常使用作为测量一组数值的离散程度之用。标准差定义：为方差开算术平方根，反映组内个体间的离散程度；标准差与期望值之比为标准离差率。

$$SD = \sqrt{\frac{1}{N} \sum_{i=1}^N (x_i - \mu)^2}$$

3 · 协方差 (Covariance)

协方差（**Covariance**）在概率论和统计学中用于衡量两个变量的总体误差。而方差是协方差的一种特殊情况，即当两个变量是相同的情况。

期望值分别为 $E(X)=\mu$ 与 $E(Y)=\nu$ 的两个具有有限二阶矩的实数随机变量 X 与 Y 之间的协方差定义为：

$$\text{cov}(X, Y) = E((X - \mu)(Y - \nu)) = E(X \cdot Y) - \mu \nu$$

其推导程为：

$$\begin{aligned} \text{Cov}(X, Y) &= E[(X - E[X])(Y - E[Y])] \\ &= E[XY] - 2E[Y]E[X] + E[X]E[Y] \\ &= E[XY] - E[X]E[Y] \end{aligned}$$

如果两个变量的变化趋势一致，也就是说如果其中一个大于自身的期望值时另外一个也大于自身的期望值，那么两个变量之间的协方差就是正值；如果两个变量的变化趋势相反，即其中一个变量大于自身的期望值时另外一个却小于自身的期望值，那么两个变量之间的协方差就是负值。

如果 X 与 Y 是统计独立的，那么二者之间的协方差就是 0，因为两个独立的随机变量满足 $E[XY]=E[X]E[Y]$ 。但是反过来并不成立，即如果 X 与 Y 的协方差为 0，二者并不一定是统计独立的。

4 · 自相关 (Auto correlation)

自相关 (Autocorrelation)，也叫序列相关，是一个信号于其自身在不同时间点的互相关。非正式地来说，它就是两次观察之间的相似度对它们之间的时间差的函数。它是找出重复模式（如被噪声掩盖的周期信号），或识别隐含在信号谐波频率中消失的基频的数学工具。它常用于信号处理中，用来分析函数或一系列值，如时域信号。

自相关函数在不同的领域，定义不完全等效。在某些领域，自相关函数等同于自协方差（信号与自身经过时间平移得到的信号之间的协方差）。比如在统计学中：

$$R(k) = \frac{E[(X_i - \mu_i)(X_{i+k} - \mu_{i+k})]}{\sigma^2}$$

\mathbf{E} ：期望值

X_i ：在 $t(i)$ 时的随机变量值。

μ_i ：在 $t(i)$ 时的预期值。

X_{i+k} ：在 $t(i+k)$ 时的随机变量值。

μ_{i+k} ：在 $t(i+k)$ 时的预期值。

σ^2 ：方差。

5 · 互相关 (Correlation)

在统计学中，互相关 (Correlation) 有时用来表示两个随机矢量 X 和 Y 之间的协方差 $\text{cov}(X, Y)$ ，以与矢量 X 的协方差概念相区分，矢量 X 的协方差是 X 的各标量成分之间的协方差矩阵。

6 · 蒙特卡洛法 / 拉斯维加斯法

蒙特卡罗算法 (Monte Carlo Method，统计模拟方法) 和拉斯维加斯算法 (Las Vegas Algorithm) 并不是一种算法的名称，而是对一类随机算法的特性的概括。具有如下特点：

- 蒙特卡罗算法：采样越多，越近似最优解；
- 拉斯维加斯算法：采样越多，越有机会找到最优解

参考：

<https://www.zhihu.com/question/20254139/answer/33572009>

7 · 马尔可夫链 (Markov Chain)

马尔科夫链 (MC, Markov Chain) 定义本身比较简单，它假设某一时刻状态转移的概率只依赖于它的前一个状态。举个形象的比喻，假如每天的天气是一个状态的话，那个今天是不是晴天只依赖于昨天的天气，而和前天的天气没有任何关系。

当然这么说可能有些武断，但是这样做可以大大简化模型的复杂度，因此马尔科夫链在很多时间序列模型中得到广泛的应用，比如循环神经网络 RNN，隐式马尔科夫模型 HMM 等，当然 MCMC (马尔科夫链-蒙特卡洛方法) 也需要它。

参考：

<https://www.cntofu.com/book/85/math/probability/markov-chain.md>

8 · 回归分析

回归分析 (Regression Analysis) 是一种统计学上分析数据的方法，目的在于了解两个或多个变量间是否相关、相关方向与强度，并建立数学模型以便观察特定变量来预测研究者感兴趣的变量。更具体的来说，回归分析可以帮助人们了解在只有一个自变量变化时因变量的变化量。一般来说，通过回归分析我们可以由给出的自变量估计因变量的条件期望

回归分析是建立因变量 Y (或称依变量，反因变量) 与自变量 X (或称独变量，解释变量) 之间关系的模型。简单线性回归使用一个自变量 X ，复回归使用超过一个自变量 (X_1 ,

$X_2 \dots X_i$) 。

参考：
[维基百科](#)

9 · 线性回归 (Linear Regression)

Linear Regression，线性回归输出是一个连续值，因此适用于回归问题。回归问题在实际中很常见，如预测房屋价格、气温、销售额等连续值的问题。

与回归问题不同，分类问题中模型的最终输出是一个离散值。我们所说的图像分类、垃圾邮件识别、疾病检测等输出为离散值的问题都属于分类问题的范畴。**softmax 回归**则适用于分类问题。

10 · 逻辑回归 (Logistic Regression)

统计学中，**逻辑回归 (Logistic Regression, LR)** 用于给分类问题的可能性建模，通常用于**二分类**，比如胜负、生死、健康/生病等等。但也可以用于**多分类**，在多分类的逻辑回归中，每个分类的可能性被赋予一个 0 到 1 之间的值，所有分类的值和为 1。

混淆注意：逻辑回归虽然名字叫回归，但却是一种分类模型

参考：
https://en.wikipedia.org/wiki/Logistic_regression

11 · softmax 回归

Softmax Regression，又叫**多项式回归 (Multinomial Regression, Multinomial Logistic Regression)**。用于分类问题，参考[softmax 激活函数](#)。

12 · 最大似然估计 (MLE)

最大似然估计 (MLE, Maximum Likelihood Estimation)，用于估计一个概率模型的参数的方法。

给定一个概率分布 D ，已知其概率密度函数（连续分布）或概率质量函数（离散分布）为 f_D ，以及一个分布参数 θ ，我们可以从这个分布中抽出一个具有 n 个值的采样 X_1, X_2, \dots, X_n ，利用 f_D 计算出其似然函数：

$$L(\theta | x_1, \dots, x_n) = f_{\theta}(x_1, \dots, x_n)$$

若 D 是离散分布， f_{θ} 即是在参数为 θ 时观测到这一采样的概率。若其是连续分布， f_{θ} 则为 X_1, X_2, \dots, X_n 联合分布的概率密度函数在观测值处的取值。一旦我们获得 X_1, X_2, \dots, X_n ，我们就能求得一个关于 θ 的估计。最大似然估计会寻找关于 θ 的最可能的值（即，在所有可能的 θ 取值中，寻找一个值使这个采样的“可能性”最大化）。从数学上来说，我们可以在 θ 的所有可能取值中寻找一个值使得似然函数取到最大值。这个使可能性最大的 θ 值即称为 θ 的最大似然估计。由定义，最大似然估计是样本的函数。

参考：

[维基百科](#)

(七) 数字信号处理

1 · 傅立叶变换

Fourier Transform，一种线性积分变换，用于信号在时域（或空域）和频域之间的变换。

2 · 离散傅立叶变换

DFT，Discrete Fourier Transform，是傅立叶变换在时域和频域上都呈离散的形式，将信号的时域采样变换为其 DTFT（离散时间傅立叶变换）的频域采样。

3 · 快速傅立叶变换

Fast Fourier Transform，FFT，是快速计算离散傅立叶变换（DFT）的方法

4 · 基波和谐波

在复杂的周期性震荡中，包含基波和谐波，和该振荡最长周期相等的正弦波分量称为基波。相应于这个周期的频率称为基波频率。频率等于基波频率的整倍数的正弦波分量称为谐波。

5 · 滤波器

filter，过滤特定频率信号的器件/装置/程序/函数，根据其作用大致分为

- **低通滤波器**：Low-Pass Filter，LPF，允许低频（低于某个频率）信号通过
- **高通滤波器**：High-Pass Filter，HPF 允许高频（高于某个频率）信号通过
- **带通滤波器**：Bandwidth-Pass Filter，BPF，允许某个频带（bandwidth，某两个频率之间）信号通过，通常是高通滤波器和低通滤波器的级联
- **带阻滤波器**：减弱某个频带（bandwidth，某两个频率之间）信号通过，通常是高通滤波器和低通滤波器的级联

6 · FIR 滤波器

Finite Impulse Response Filter，有限脉冲响应滤波器，数字滤波器的一种。这类滤波器对于脉冲输入信号的响应最终趋于 0，因此是有限的，从而得名。

FIR 滤波器是一线性系统，输入信号 $x(0), x(1), \dots, x(n)$ ，经过该系统后的输出信号 $y(n)$ 可表示为：

$$y(n) = h_0x(n) + h_1x(n-1) + \dots + h_Nx(n-N)$$

其中 h_0, h_1, \dots, h_N 是滤波器的脉冲响应，通常称为滤波器的系数， N 是滤波器的阶数。

7 · IIR 滤波器

Infinite Impulse Response Filter，无限脉冲响应滤波器，数字滤波器的一种。

8 · 小波变换

WT，**Wavelet Transform**，跟傅立叶变换一样分为连续小波变换（CWT）和离散小波变换（DWT）。

9 · ACF（自相关函数）

自相关函数（Auto-correlation function），用于衡量信号在不同时期之间的相关关系。

10 · XCF（互相关函数）

互相关函数（Cross-correlation function），是用来表示两个信号之间相似性的一个度量。

对于离散函数 f_i 和 g_i 来说，互相关定义为：

$$(f \star g)_i \equiv \sum_j f_j^* g_{i+j}$$

互相关实际上类似于两个函数的卷积。

(八) 机器学习

机器学习（Machine Learning，ML）是人工智能（AI）的一个分支，是深度学习的超集。

通过算法和统计数据的学习，来完成特定的任务。ML 无需明确的指令，而是依赖于模式和推论。ML 基于样本数据（训练数据集）建立一个数学模型，以便在不明确编码的情况下完成预测或者决定。

机器学习已广泛应用于数据挖掘、计算机视觉、自然语言处理、生物特征识别、搜索引擎、医学诊断、检测信用卡欺诈、证券市场分析、DNA 序列测序、语音和手写识别、战略游戏和机器人等领域。

机器学习有下面几种定义：

- 机器学习是一门人工智能的科学，该领域的研究对象是人工智能，特别是如何在经验学习中改善具体算法的性能。
- 机器学习是对能通过经验自动改进的计算机算法的研究。
- 机器学习是用数据或以往的经验，以此优化计算机程序的性能标准。

一种经常引用的英文定义是：A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P, if its performance at tasks in T, as measured by P, improves with experience E.

机器学习和数据挖掘的关系：

机器学习和数据挖掘经常使用同样的方法，且他们高度重合。

机器学习和深度学习的关系：

深度学习是机器学习的一种，是使用 DNN（深度神经网络）的机器学习。

本节包含了归属于机器学习（Machine Learning），但不属于深度学习（Deep Learning）的概念及算法。

1 · 监督学习

监督学习（Supervised learning），是机器学习的一种方法，可以由训练资料中学到或建立一个模式（函数 / learning model），并依此模式推測新的实例。训练资料是由输入（通常是向量）和预期输出所组成。函数的输出可以是一个连续的值（回归分析），或是预测一个分类标签（统计分类）。

一个监督式学习者的任务在观察完一些事先标记过的训练范例（输入和预期输出）后，去预测这个函数对任何可能出现的输入的输出。要达到此目的，学习者必须以"合理"的方式从现有的资料中一般化到没观察到的情况。

监督式学习有两种形态的模型。最一般的，监督式学习产生一个全域模型，会将输入物件对应到预期输出。而另一种，则是将这种对应实作在一个区域模型。

另外对于监督式学习所使用的辞汇则是分类。现著有著各式的分类器，各自都有强项或弱项。分类器的表现很大程度上地跟要被分类的资料特性有关。并没有某一单一分类器可以在所有给定的问题上都表现最好。各式的经验法则被用来比较分类器的表现及寻找会决定分类器表现的资料特性。决定适合某一问题的分类器仍旧是一项艺术，而非科学。

目前最广泛被使用的分类器有人工神经网络、支持向量机、最近邻居法、高斯混合模型、朴素贝叶斯方法、决策树和径向基函数分类。

主动式学习：一个情况是，有大量尚未标示的资料，但去标示资料则是很耗成本的。一种方法则是，学习算法会主动去向使用者或老师去询问标签。这种形态的监督式学习称为主动式学习。既然学习者可以选择例子，学习中要使用到的例子个数通常会比一般的监督式学习来得少。以这种策略则有一个风险是，算法可能会专注于一些不重要或不合法的例子。

参考：

[维基百科](#)

2 · 统计分类

统计分类（Statistical Classification）是机器学习非常重要的一个组成部分，它的目标是根据已知样本的某些特征，判断一个新的样本属于哪种已知的样本类。分类是监督学习的一个实例，根据已知训练集提供的样本，通过计算选择特征参数，创建判别函数以对样本进行的分类。与之相对的是无监督学习，例如聚类分析。

3 · 回归分析

回归分析（Regression Analysis）是一种统计学上分析数据的方法，目的在于了解两个或多个变量间是否相关、相关方向与强度，并建立数学模型以便观察特定变量来预测研究者感兴趣的变量。更具体的来说，回归分析可以帮助人们了解在只有一个自变量变化时因变量的变化量。一般来说，通过回归分析我们可以由给出的自变量估计因变量的条件期望。

回归分析是建立因变量 Y（或称依变量，反因变量）与自变量 X 或称独变量，解释变量之间关系的模型。简单线性回归使用一个自变量 X，复回归使用超过一个自变量（X₁, X₂…X_i）。

回归的最早形式是最小二乘法。

4 · 无监督学习

无监督学习（Unsupervised learning）是机器学习的一种类型，没有给定事先标记过的训练示例，自动对输入的数据进行分类或分群。无监督学习的主要运用包含：聚类分析（Cluster Analysis）、关系规则（Association Rule）、维度缩减（Dimensionality Reduce）。它是监督式学习和强化学习之外的一种选择。

一个常见的无监督学习是数据聚类。在人工神经网络中，生成对抗网络（GAN）、自组织映射（SOM）和适应性共振理论（ART）则是最常用的非监督式学习。

ART 模型允许集群的个数可随着问题的大小而变动，并让用户控制成员和同一个集群之间的相似度分数，其方式为透过一个由用户自定而被称为警觉参数的常量。ART 也用于模式识别，如自动目标识别和数字信号处理。第一个版本为 "ART1"，是由卡本特和葛罗斯柏格所发展的。

5 · 聚类分析

聚类分析（Cluster analysis）亦称为群集分析，是对于统计数据分析的一门技术，在许多领域受到广泛应用，包括机器学习，数据挖掘，模式识别，图像分析以及生物信息。聚类是把相似的对象通过静态分类的方法分成不同的组别或者更多的子集（subset），这样让在同一个子集中的成员对象都有相似的一些属性，常见的包括在坐标系中更加短的空间距离等。

6 · 强化学习

强化学习（Reinforcement learning，RL）是机器学习中的一个领域，强调如何基于环境而行动，以取得最大化的预期利益。其灵感来源于心理学中的行为主义理论，即有机体如何在环境给予的奖励或惩罚的刺激下，逐步形成对刺激的预期，产生能获得最大利益的习惯性行为。这个方法具有普适性，因此在其他许多领域都有研究，例如博弈论、控制论、运筹学、信息论、仿真优化、多主体系统学习、群体智能、统计学以及遗传算法。在运筹学和控制理论研究的语境下，强化学习被称作“近似动态规划”（approximate dynamic programming，ADP）。在最优控制理论中也有研究这个问题，虽然大部分的研究是关于最优解的存在和特性，并非是学习或者近似方面。在经济学和博弈论中，强化学习被用来解释在有限理性的条件下如何出现平衡。

在机器学习问题中，环境通常被规范为马可夫决策过程（MDP），所以许多强化学习算法在这种情况下使用动态规划技巧。传统的技术和强化学习算法的主要区别是，后者不需要关于MDP的知识，而且针对无法找到确切方法的大规模MDP。

强化学习和标准的监督学习之间的区别在于，它并不需要出现正确的输入/输出对，也不需要精确校正次优化的行为。强化学习更加专注于在线规划，需要在探索（在未知的领域）和遵从（现有知识）之间找到平衡。强化学习中的“探索-遵从”的交换，在多臂老虎机问题和有限MDP中研究得最多。

7 · SVM

在机器学习中，支持向量机（support vector machine，SVM，支持向量网络）是在分类与回归分析中分析数据的监督式学习模型与相关的学习算法。给定一组训练实例，每个训练实例被标记为属于两个类别中的一个或另一个，SVM训练算法创建一个将新的实例分配给两个类别之一的模型，使其成为非概率二元线性分类器。SVM模型是将实例表示为空间中的点，这样映射就使得单独类别的实例被尽可能宽的明显的间隔分开。然后，将新的实例映射到同一空间，并基于它们落在间隔的哪一侧来预测所属类别。

当数据未被标记时，不能进行监督式学习，需要用非监督式学习，它会尝试找出数据到簇的自然聚类，并将新数据映射到这些已形成的簇。将支持向量机改进的聚类算法被称为支持向量聚类，当数据未被标记或者仅一些数据被标记时，支持向量聚类经常在工业应用中用作分类步骤的预处理。

分类数据是机器学习中的一项常见任务。假设某些给定的数据点各自属于两个类之一，而

目标是确定新数据点将在哪个类中。对于支持向量机来说，数据点被视为 p 维向量，而我们想知道是否可以用 $(p-1)$ 维超平面来分开这些点。这就是所谓的线性分类器。可能有许多超平面可以把数据分类。最佳超平面的一个合理选择是以最大间隔把两个类分开的超平面。因此，我们要选择能够让到每边最近的数据点的距离最大化的超平面。如果存在这样的超平面，则称为最大间隔超平面，而其定义的线性分类器被称为最大间隔分类器，或者叫做最佳稳定性感知器。

更正式地来说，支持向量机在高维或无限维空间中构造超平面或超平面集合，其可以用于分类、回归或其他任务。直观来说，分类边界距离最近的训练数据点越远越好，因为这样可以缩小分类器的泛化误差。

参考：

[维基百科](#)

8 · k-means

k-means clustering，**k-means 聚类**，**k 均值聚类**，**k 平均算法**，一种迭代求解的聚类分析算法。

其步骤是：

- 将数据分为 K 组，随机选取 K 个对象作为初始的聚类中心。
- 然后计算每个对象与各个种子聚类中心之间的距离，把每个对象分配给距离它最近的聚类中心。聚类中心以及分配给它们的对象就代表一个聚类。
- 每分配一个样本，聚类中心会根据聚类中现有的对象被重新计算。这个过程将不断重复直到满足某个终止条件。

终止条件可以是：

- 没有（或最小数目）对象被重新分配给不同的聚类
- 没有（或最小数目）聚类中心再发生变化
- 误差平方和局部最小。

9 · 感知器

感知器（Perceptron） 是 Frank Rosenblatt 在 1957 年就职于康奈尔航空实验室（Cornell Aeronautical Laboratory）时所发明的一种人工神经网络。它可以被视为一种最简单形式的前馈神经网络，是一种二元线性分类器。

Frank Rosenblatt 给出了相应的感知机学习算法，常用的有感知机学习、最小二乘法和梯度下降法。譬如，感知机利用梯度下降法对损失函数进行极小化，求出可将训练数据进行线性划分的分离超平面，从而求得感知机模型。

在人工神经网络领域中，感知机也被指为单层的人工神经网络，以区别于较复杂的多层次感知机（MLP，Multi-Layer Perceptron）。作为一种线性分类器，（单层）感知机可说是最简单的前向人工神经网络形式。尽管结构简单，感知机能够学习并解决相当复杂的问题。感知机主要的本质缺陷是它不能处理线性不可分问题。

10 · 朴素贝叶斯方法

在机器学习中，**朴素贝叶斯分类器**（Naïve Bayes Classifier）是一系列以假设特征之间强（朴素）独立下运用贝叶斯定理为基础的简单概率分类器。

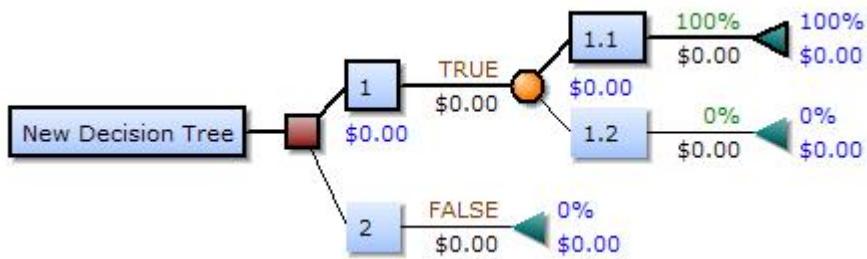
朴素贝叶斯分类器是高度可扩展的，因此需要数量与学习问题中的变量（特征/预测器）成线性关系的参数。最大似然训练可以通过评估一个封闭形式的表达式来完成，只需花费线性时间，而不需要其他很多类型的分类器所使用的费时的迭代逼近。

11 · 决策树

机器学习中，**决策树**（Decision tree）是一个预测模型；他代表的是对象属性与对象值之间的一种映射关系。树中每个节点表示某个对象，而每个分叉路径则代表某个可能的属性值，而每个叶节点则对应从根节点到该叶节点所经历的路径所表示的对象的值。决策树仅有单一输出，若欲有复数输出，可以建立独立的决策树以处理不同输出。数据挖掘中决策树是一种经常要用到的技术，可以用于分析数据，同样也可以用来作预测。

从数据产生决策树的机器学习技术叫做决策树学习，通俗说就是决策树。一个决策树包含三种类型的节点：

- 决策节点：通常用矩形框来表示
- 机会节点：通常用圆圈来表示
- 终结点：通常用三角形来表



参考：

[维基百科](#)

12 · 随机森林

在[机器学习](#)中，随机森林（Random Forrest）是一个包含多个[决策树](#)的分类器，并且其输出的类别是由个别树输出的类别的[众数](#)而定。

13 · kNN 算法

邻近算法，K 最近邻，kNN，k-NearestNeighbor，是一种分类算法。是数据挖掘分类技术中最简单的方法之一。所谓 K 最近邻，就是 k 个最近的邻居的意思，说的是每个样本都可以用它最接近的 k 个邻居来代表。

其步骤是：

1. 计算测试数据与各个训练数据之间的距离
2. 按照距离的递增关系进行排序；
3. 选取距离最小的 K 个点；
4. 确定前 K 个点所在类别的出现频率；
5. 返回前 K 个点中出现频率最高的类别作为测试数据的预测分类

K 值（临近数，即在预测目标点时取几个临近的点来预测）的选取非常重要，因为：

1. 如果当 K 的取值过小时，一旦有噪声得成分存在们将会对预测产生比较大影响，例如取 K 值为 1 时，一旦最近的一个点是噪声，那么就会出现偏差，K 值的减小就意味着整体模型变得复杂，容易发生过拟合；
2. 如果 K 的值取的过大时，就相当于用较大邻域中的训练实例进行预测，学习的近似误差会增大。这时与输入目标点较远实例也会对预测起作用，使预测发生错误。K 值的增大就意味着整体的模型变得简单；
3. 如果 $K=N$ 的时候，那么就是取全部的实例，即为取实例中某分类下最多的点，就对预测没有什么实际的意义了；

- K 的取值尽量要取奇数，以保证在计算结果最后会产生一个较多的类别，如果取偶数可能会产生相等的情况，不利于预测
- 无论是分类还是回归，衡量邻居的权重都非常有用，使较近邻居的权重比较远邻居的权重大。例如，一种常见的加权方案是给每个邻居权重赋值为 $1/d$ ，其中 d 是到邻居的距离。
- 邻居都取自一组已经正确分类（在回归的情况下，指属性值正确）的对象。虽然没要求明确的训练步骤，但这也可以当作是此算法的一个训练样本集。
- k-近邻算法的缺点是对数据的局部结构非常敏感。

混淆注意！

kNN 算法与 K-平均算法（k-means）没有任何关系，请勿与之混淆。

14 · 隐马尔可夫模型

隐马尔可夫模型 (Hidden Markov Model, HMM) 或称作隐性马尔可夫模型，是统计模型，它用来描述一个含有隐含未知参数的马尔可夫过程。其难点是从可观察的参数中确定该过程的隐含参数。然后利用这些参数来作进一步的分析。

任何一个 HMM 都可以通过下列五元组来描述：

- observations：观测序列
- states：隐状态
- start_prob：初始概率（隐状态）
- trans_prob：转移概率（隐状态）
- emit_prob：发射概率（隐状态表现为显状态的概率）

隐马尔可夫模型作了两个基本假设 (o_t 为 t 时刻的观测状态, i_t 为 t 时刻的隐状态) :

- **马尔可夫性假设**, 即假设隐藏的马尔可夫链在任意时刻 t 的状态只依赖于其前一时刻的状态, 与其它时刻的状态及观测无关, 也与时刻 t 无关:
$$p(i_t | i_{t-1}, o_{t-1}, \dots, i_1, o_1) = p(i_t | i_{t-1}), \quad t=1, 2, \dots, T$$
- **观测独立性假设**, 即假设任意时刻的观测只依赖于该时刻的马尔可夫链的状态, 与其他观测及状态无关:
$$p(o_t | i_T, o_T, i_{T-1}, o_{T-1}, \dots, i_{t+1}, o_{t+1}, i_t, o_t, \dots, i_1, o_1) = p(o_t | i_t)$$

举一个经典的例子：一个东京的朋友每天根据天气{下雨，天晴}决定当天的活动{公园散步，购物，清理房间}中的一种，我每天只能在 twitter 上看到她发的推“啊，我前天公园散步、昨天购物、今天清理房间了！”，那么我可以根据她发的推特推断东京这三天的天气。在这个例子里，显状态是活动，隐状态是天气。

```
states = ('Rainy', 'Sunny')
observations = ('walk', 'shop', 'clean')
start_prob = {'Rainy': 0.6, 'Sunny': 0.4}
trans_prob = { 'Rainy' : {'Rainy': 0.7, 'Sunny': 0.3}, 'Sunny' : {'Rainy': 0.4, 'Sunny': 0.6}, }
emission_prob = {'Rainy' : {'walk': 0.1, 'shop': 0.4, 'clean': 0.5}, 'Sunny' : {'walk': 0.6, 'shop': 0.3, 'clean': 0.1}, }
```

参考：

<https://www.zhihu.com/question/20962240>

<https://blog.csdn.net/hudashi/article/details/87867916>

15 · 神经网络

人工神经网络（Artificial Neural Network，ANN），也叫神经网络（Neural Network，NN）或类神经网络，在机器学习和认知科学领域，是一种模仿生物神经网络（动物的中枢神经系统，特别是大脑）的结构和功能的数学模型或计算模型，用于对函数进行估计或近似。神经网络由大量的人工神经元联结进行计算。大多数情况下人工神经网络能在外界信息的基础上改变内部结构，是一种自适应系统，通俗的讲就是具备学习功能。现代神经网络是一种非线性统计性数据建模工具。

神经网络的构筑理念是受到生物（人或其他动物）神经网络功能的运作启发而产生的。工神经网络通常是通过一个基于数学统计学类型的学习方法（Learning Method）得以优化，所以也是数学统计学方法的一种实际应用，通过统计学的标准数学方法我们能够得到大量的可以用函数来表达的局部结构空间，另一方面在人工智能学的人工感知领域，我们通过数学统计学的应用可以来做人工感知方面的决定问题（也就是说通过统计学的方法，人工神经网络能够类似人一样具有简单的决定能力和简单的判断能力），这种方法比起正式的逻辑学推理演算更具有优势。

和其他机器学习方法一样，神经网络已经被用于解决各种各样的问题，例如机器视觉和语音识别。这些问题都是很难被传统基于规则的编程所解决的。

支持向量机和其他更简单的方法（例如线性分类器）在机器学习领域的流行度层逐渐超过了神经网络，但是在 2000 年代后期出现的深度学习重新激发了人们对神经网络的兴趣。

参考：

[维基百科](#)

(九) NN 相关

神经网络相关的概念

1 · 数据集

数据集（Dataset），机器学习模型的输入数据，通常分为三部分：

- 训练集：Training Set，用于训练模型，调整权重参数
- 验证集：Validation Set，也叫开发集（Dev Set）用于模型的选择，超参数的选择
- 测试集：Test Set，为了最终测试选择并训练好的模型（的泛化能力）。

训练集是模型学习训练的数据集，不同的模型（神经网络结构、超参数）在这个数据集上调整权重参数，以求（在训练集上）达到更高的准确率。而验证集提供了一个统一的衡量

标准，以便调整模型（修改超参数、网络结构，或者直接放弃某个模型），某些训练过程不使用验证集。测试集则是在模型训练完成之后，用于测试模型的准确性（测试集之前从没喂给过模型，以测试泛化能力）。

打个比方，训练集就像平时作业，验证集像平时小考，测试集则是决定你升学方向的毕业考试。

数据集是神经网络的研究中非常重要的一个部分，甚至有独有的数据集就可以出论文的说法。

CV 数据集大全：

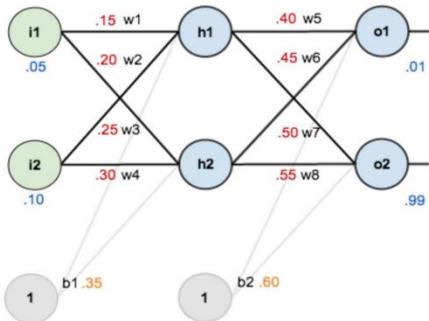
<http://homepages.inf.ed.ac.uk/rbf/CVonline/Imagedbase.htm>

数据集列表：

https://en.wikipedia.org/wiki/List_of_datasets_for_machine-learning_research

2 · 层

神经网络中的层通常可以分为输入层、输出层和隐藏层三大类。层是一个略微模糊的概念，举一个最简单的例子，下图是一个2个输入（ i_1 和 i_2 ），2个输出（ o_1 和 o_2 ），2个隐藏神经元（ h_1 和 h_2 ）的神经网络。



那么问题来了，这个网络有几层？

从结构图上看，可以认为节点（即张量）代表层，因此可以说它有三层（输入层 i 、输出层 o 和隐藏层 h ）。而从计算的角度来讲，连接线代表层，因此其实只有两层全连接层（输入 \rightarrow 隐藏，隐藏 \rightarrow 输出），所谓的输入层只是一个没有计算量的向量而已。

此外在某些环境里，只有卷积、全连接等这些层被视为层，而在某些环境里，BN、激活函数也被视为BN层、激活层。

3 · 参数和 FLOPs

有两个值可以用于描述神经网络的大小，层的结构直接决定了这两个值：

- **参数数量**

可学习参数（即权重 weight、偏置 bias 等）的数量，这个值的大小决定了神经网络在空间上的大小（即所使用的内存和硬盘的大小）。

- **FLOPs**

FLoat OPerations，描述一次计算的浮点操作数量（注意不要和 **FLOPS/FLoat Operations Per Second** 混淆，FLOPS 是用来描述计算机性能的），这个值的大小决定了计算量，也就是神经网络在时间上的大小。

FLOPs 通常是“加”和“乘”两种操作，有两种计算方法，一种是将加和乘分别算作一次，还有一种是将一次加和一次乘合起来算作一次 FLOP（因为加和乘操作是一一对应的，这也是通常的计算方式）。下面的计算中我们采用第二种。

参数和 **FLOPs** 正相关，但不成严格正比。比如同样 **FLOPs** 的情况下，全连接层的参数数量就要远大于卷积层，但可能是由于网络中大部分计算量都集中在卷积层的原因，很多文章中直接用参数数量来衡量计算量。

4 · Ground Truth

Ground Truth (GT) 表示**真实值**。

BTW：至于为什么叫 **Ground**，据说这个词的起源是**遥感领域**，相对于在天上的遥感卫星所测量到的数据来说，地面上的真实数据就是 **Ground Truth**。

5 · 置信度

Confidence，表示对某个预测的信心，通常取值范围从 0 到 1。

6 · One-hot 标签

One-hot 标签（中文通常翻译为**独热标签**，但很少用，一般直接用英文）用于描述分类任务的 **ground truth**，一个描述 N 个分类的 N 维向量，其中正确分类对应的标签值为 1，其余都为 0，没有其它取值。

7 · 降维/升维

降维指的是减少特征维度（比如 CNN 中特征图的通道数量），实现方法有 1×1 卷积等。

升维指的是增加特征维度，实现方法有 1×1 卷积等。

8 · 上采样/下采样

上采样（Upsampling），目的是放大图像，采用的具体方法有插值、反卷积、反池化等。

下采样又叫降采样（Subsampling，Downsampling），目的是缩小图像，具体采用的方法主要是池化，步长（stride）大于 1 的卷积也可以实现下采样。

9 · 编码器/解码器

编码器（encoder）和解码器（decoder），在深度学习中，编码器的概念是指一个将输入转化为特征图作为输出的网络（可以是 FC、CNN、RNN），特征图中包含了输入的信息。而解码器则是一个完全相反的网络，以特征图作为输入，而试图给出期望的输出。

通常编码是个降采样的过程，而解码是个上采样的过程。编码器和解码器成对出现（基础的 CNN 比如 VGG 等就可以看作一个编码器，但是在没有对应的解码器网络的时候它不会被叫做编码器），先编码，再解码。编码器/解码器在图像语义分割、机器翻译、GAN 等等中都有应用。

10 · 开集/闭集

闭集（close-set）指的是在分类问题中，所有的输入数据都属于某一类别，不会出现没有见过的 unknown 数据。而开集（open-set）正好相反。比如在人脸识别中，如果有不在数据库中的人脸作为输入，则为 open-set，否则为 close-set。

闭集对类别间的判别度（discriminative）要求更低，因为数据已知，而开集则因为运行中的数据未知，因此对判别度的要求更高，这样才好更清晰的区别于各种未知数据。

11 · 类内/类间

类内（intra-class）指的是分类问题中属于同一类别的不同样本，类间（inter-class）指的是属于不同类别的样本。

12 · 二分类任务衡量标准

在图像分类和目标检测等二分类任务中，检测结果可分为四类，即

- TP : True Positive，真阳性，阳性样本，检测为阳性
- FP : False Positive，假阳性，阴性样本，检测为阳性
- TN : True Negative，TN，真阴性，阴性样本，检测为阴性
- FN : False Negative，FN，假阴性，阳性样本，检测为阴性

其中 TP 和 FP 是检测结果为阳性，TN 和 FN 是检测结果为阴性；TP 和 FN 实际为阳性，TN 和 FP 实际为阴性。下面介绍由这四个值引申出几个“率”：

- Accuracy : 准确率，最直观的衡量的标准，等于 $(TP+TN)/(TP+TN+FP+FN)$ ，所有样本的检测准确性。
- TPR (True Positive Ratio)，也叫查全率，召回率 (Recall)，hit rate，或者 Sensitivity，等于 $TP/(TP+FN)$ ，用于衡量所有实际为阳性的样本的检测准确率。
- TNR (True Negative Ratio)，又叫 Specificity，等于 $TN/(TN+FP)$ ，用于衡量所有实际为阴性的样本的检测准确率，不太常用。
- FNR (False Negative Ratio)，等于 $FN/(TP+FN)$ ，等于 $1-TPR$ ，用于衡量所有实际为阳性样本的检测错误率，不太常用。
- FPR (False Positive Ratio)，等于 $FP/(FP+TN)$ ，等于 $1-TNR$ ，用于衡量所有实际为阴性样本的检测错误率，不太常用。
- 查准率 (准确率，Precision)，PPV (Positive Predictive Value)，等于 $TP/(TP+FP)$ ，表示所有被检测为阳性的样本中，检测正确的比例。
- NPV (Negative Predictive Value)，等于 $TN/(TN+FN)$ ，表示所有被检测为阴性的样本中，检测正确的比例，不太常用。

混淆注意！

Accuracy 和 Precision 这两个词中文都可翻译为准确率，但在这里是两个概念，因此建议使用查准率表示 Precision，或者直接用英文。

- F1-Score : 也叫 F1-measure，是 $\beta=1$ 时 F-score 的一个特例，F-score 同时考虑了查准率和查全率，是这两者的一个加权平均，F1-Score 的公式为：
$$\frac{2 \times \text{precision} \times \text{recall}}{\text{precision} + \text{recall}}$$

不同的衡量标准在不同的情况下有意义，要根据具体的工作场景来决定用什么衡量标准。

场景 A :

对医学影像的癌症检测，可能在所有样本中，实际为阳性的只有 0.1%，这时候如果算法将所有样本直接判定为阴性，则 Accuracy 也可以达到 99.9%，但这毫无意义。

场景 B :

PPV 和 NPV 这两个值，在二元图像分类任务中，仅在测试数据集中阳性/阴性样本的比例和真实环境差不多的时候有意义。举个例子，在鉴定枪械图片的任务中，测试数据集包括 100 张枪械图片和 100 张非枪械图片，枪械图片被检测为枪械图片有 95 张 (TP) ，5 张没检测出来 (FN) ；非枪械图片被检测为枪械图片有 10 张 (FP) ，被检测为非枪械图片为 90 张 (TN) 。

那么， TPR 为 95%， TNR 为 90%， $PPV=95/(95+10)$ 约为 90.5% ； $NPV=90/(90+5)$ 约为 94.7% 。仿佛挺美好，但是现实生活中的绝大部分图片都不是枪械图片，让我们假设每 10000 张非枪械图片里搭配 100 张枪械图片，假设 TPR 和 TNR 不变，则 TP 和 FN 数量不变， TN 为 9000 张， FP 为 1000 张，于是有： $PPV=95/(95+1000)$ ，约为 8.7%。或者说，每 100 张被识别为枪械的图片中，只有 9 张不到是真的枪械图片，另外 90 多张都是错误识别的结果。

13 · 感受野

感受野指的是在图像相关的任务中，卷积核能“看到”的范围，和卷积核的大小，以及卷积方式有关，比如 $7*7$ 卷积核的感受野大于 $5*5$ 卷积核，两次 $3*3$ 卷积核的感受野大于一次，带孔卷积的感受野大于普通卷积。

14 · 表现力

表现力指的是网络对输入数据描述的能力，表现力过弱会导致准确性不高，这就像你无法用 8 个 bit 来描述足够鲜艳的颜色一样。而表现力过强也可能会带来过拟合之类的问题（在数据集不够大的情况下），这种情况下网络表现力会过多的描述一些事实上和结果无关的特征。

15 · 嵌入

embedding，通常为一个 N 维的向量，其概念有点类似特征值，指的是某种高维空间的向量在低维空间的表示。

比如词嵌入（对应的原始高维度为所有词的个数，One-hot 格式），或者人脸识别中的脸部特征的嵌入（对应的原始高维度为图片长×宽×3，3 为 RGB）。

可以把嵌入视为一种描述，对于词来说，类似一个对该词的释义，对脸图来说，就是对脸的描述，比如眼睛大小用一个值表示，鼻子长短用一个值表示等，这些值组成了一个若干维的向量。只不过这种描述更加精确，且无法为人类所理解。

有一个例子可以阐述嵌入的意思，词嵌入中，比如我们用 $E_{国王}$ 表示“国王”的词嵌入，可以发现 $E_{国王}-E_{男人}+E_{女人}$ 得出的结果，和 $E_{女王}$ 很接近。

16 · 二分类/多分类/多标签分类

二分类 (binary-class classification) 指的是预测结果只有两种可能的类别，比如是或者否。

多分类 (multi-class classification) 指的是预测结果有若干种可能的类别，比如一个通常的图像分类问题，预测结果可能是人、车、树等等。

多标签分类 (multi-label classification) 指的是在多分类的基础上，一个实例预测的结果可能多个标签，而非仅属于一个类别，比如一张图片中既包含人，也包含车。图片多分类不是一个显学，因为类似的功能通常在目标检测任务 (Object Detection) 里实现。

17 · 骨干网络 (Backbone)

骨干网络 (Backbone)，也叫主干网络，基础网络，通常指的是图像/视频处理任务中，用于特征提取的核心部分，这种网络本身通常用于图像分类任务，而在更复杂的图像/视频处理中（目标检测、图像分割、行为识别等等），作为一个子部分出现。

18 · 分布 (Distribution)

Distribution，是用于描述**可能性分布 (Probability Distribution)**的数据结构，在 [Tensorflow](#)、[Pytorch](#)、[Gluonts](#) 中都有出现。

Distribution 相关的 shape (张量的维度) 分为三个级别，从小到大分别是：

- **Event Shape**：一个 Event shape 的张量描述了单个分布中的一个事件，如果这个分布是个单元分布 (Univariate Distribution)，则 event shape 为标量；如果该分布是一个 N 元的多元分布 (Multivariate distribution，比如多元正态分布)，则 event shape 是一个 N 维向量。
- **Batch Shape**：Batch 描述了一次采样中的若干分布 (样本空间) 的事件，batch shape 维度描述了这些相互独立的分布的数量。
- **Sample Shape**：描述了采样 (sample) 次数，这个 shape 中的每个元素都是同样的若干个 (单元或者多元) 分布的事件。

举个例子，让 5 个人每人掷 2 个骰子，两个骰子点数各为一个变量，两个骰子都为 3 是一个事件。让这 5 个人掷 4 次。则：

- event shape 为 2 维，两个元素都为 3，单个 event 的概率是 1/36
- batch shape 为 5 维
- sample shape 为 4 维

Distribution	Draws	Event Shape	Batch Shape	Sample Shape	Plain English
		()	()	()	one draw from one normal
		()	()	(2,)	two draws from one normal
		()	(2,)	()	one draw from two normals
		()	(2,)	(2,)	two draws from two normals
		(2,)	()	()	one draw from a bivariate normal
		(2,)	()	(2,)	two draws from a bivariate normal
		(2,)	(2,)	()	one draw from two bivariate normals
		(2,)	(2,)	(2,)	two draws from two bivariate normals

混淆注意！

`draw` 这个词在不同语境下会有歧义，可以把包含单个分布的一次抽签视为一个 `draw`（仅包括 event shape），也可以把所有分布的一次抽签视为一个 `draw`（包括 batch shape 和 event shape）。

由以上可知，对于单个 sample，变量的维度为 Batch shape * event shape，而其概率密度的维度为 Batch shape。

总结起来：

- Event 中的元素（变量）可能是相互依赖的（dependent），产生单个概率密度
- Batch 中的各个元素（Event，事件）互相独立（independent），但并不相同（identical），产生 batch shape 个概率密度。
- Sample 中的各元素（Sample）相互独立且相同（即独立同分布，Independent Identical Distribution）

参考：

https://www.tensorflow.org/probability/examples/Understanding_TensorFlow_Distributions_Shapes

<https://bochang.me/blog/posts/pytorch-distributions/>

<https://ericmjl.github.io/blog/2019/5/29/reasoning-about-shapes-and-probability-distributions/>

(十) 超参数

超参数是在网络构型确定的情况下，需要人为确定的，无法学习的参数。对于一个网络来说，构型+超参数的组合决定了这个网络（以及它是否优秀）。

1 · 学习率

学习率（Learning rate），学习的速度，通常被设置为 0.001。

2 · Dropout Ratio

Dropout 层每次随机关闭的神经元的比率。

3 · Batch size

单批次训练样本的数量。

(十一) NN 训练相关

1 · Epoch / Batch

在进行预测-反向传播-优化这个过程的时候，鉴于数据集的尺度，每次优化不可能对所有的数据都进行一遍计算，目前采用的方法是随机选择若干个数据元素，成为一个 batch，在此基础上进行一次训练。

而一个 epoch 指的是将数据集中所有的数据都训练一遍，一个 epoch 中包含的训练次数 (batch number) 为 n/b_s ， n 为数据集大小（比如图片集中总的图片数量）， b_s 为 batch size（每个 batch 中包含的图片数量）。

2 · 过拟合 (Overfit)

过拟合 (Overfitting) 指的是训练出来的网络过于精密的拟合训练数据，但不能很好的拟合不包含在训练数据中的其他数据 (unseen data)，其表现是在训练集上的成绩远好于在测试集上的成绩。

过拟合的原因是因为相较于有限的训练数据（如果训练数据足够多，多到可以覆盖所有可能的数据，也不会出现过拟合），网络的参数过多或者结构过于复杂，从而使得过多的参数从统计噪声中获取了信息表达。

防止过拟合的方法包括模型选择、正则化、dropout、权值衰减、剪枝等等。

过拟合本身是个来自于统计学的概念。

3 · 学习率衰减

学习率衰减 (Learning Rate Decay) 是神经网络训练中的一种技巧，随着学习的进行使学习率逐渐减小。在训练的初期快速的学习，使得模型更快速的接近最优解，而随着迭代的次数增加，学习率逐渐减小，使得模型在训练后期不会有太大波动。

4 · 权值衰减

权值衰减 (Weight Decay) 是神经网络训练中的一种技巧，以减小权重参数的值为目的进行学习的方法，通过减小权重参数的值来抑制过拟合的发生。

5 · 梯度消失

梯度消失 (Gradient Vanishing)，是在梯度下降法中会出现的问题，由于输出不断靠近 0 或者 1，使得其导数逐渐接近 0，因此在反向传播中的梯度不断变小，最后消失。由于梯度的消失，导致学习速度降低，权重无法得到有效的更新，甚至神经网络完全无法继续训练。

6 · 超参数

超参数（Hyper Parameter）：指的是各层的神经元数量、batch 大小、学习率、权值衰减等神经网络学习的过程中不会更新，而需要人为设定的参数。

7 · 标准化（Normalization）

标准化（Normalization）也叫正规化（Standardization），在统计学中通常指的是将不同衡量标准下的值调整到一个统一的标准下，最常见的是 0 到 1 之间。

8 · 泛化（Generalization）

泛化（Generalization）这个词在机器学习中指的是消除过拟合，将模型应用于非训练数据。

（十二）CNN 相关

1 · 全连接层

全连接层（Fully-connect Layer），也叫 Dense 层，相邻两层的所有神经元都有连接。

2 · 卷积层

卷积层（Convolution Layer），进行卷积运算的层，由卷积核组成。

3 · 池化层

池化层（Pooling Layer），进行缩小长和高方向上的空间的运算的层，没有可学习的参数，通常是 Max 池化，也有 Average 池化等。

4 · 卷积核

卷积核（Convolution Kernel），也叫过滤器（Filter），进行卷积运算的滤波器。

5 · 特征图

特征图（Feature Map），卷积层的输入/输出数据

6 · 填充（Padding）

填充（Padding），当需要输出特征图和输入特征图的大小（H*W）保持一致的时候，需要向输入特征图的周围填入固定的数据（比如 0）以让卷积核在边缘也可以正常工作

7 · 步幅（Stride）

步幅（Stride）：应用滤波器的位置间隔，步幅越大，输出特征图越小

8 · 通道（Channel）

通道（Channel），卷积运算输入数据的纵深维度。用于处理图像的卷积层是 2 维的，通道是除了长和高外的第三个维度，此外批量训练时还有第四个维度 N（即 batch size）

二、Python 相关

Python 是当前深度学习的主流语言，这里介绍了一些 Python 中难懂或易混淆的概念。而 Python 的基础的简单的语法，并不包含在这里。

完整的 Python 官方文档在这里：

<https://docs.python.org/3/tutorial/index.html>

如果英文吃力或者只想快速入门 Python：

<https://www.runoob.com/python/python-tutorial.html>

下文中：

- 蓝色表示 package (或者目录)
- 红色表示 module (或者文件)
- 绿色表示 class
- 黑色表示其他 (包括各种属性、函数、变量等)
- 青底黑字表示代码、命令行

(一) 包、模块、属性

关于 Python 的包、模块、属性的介绍。

- 包：python 中的包 (package) 就是同名目录，且需要在该目录下有 `__init__.py` 表示该目录是一个 package (其内容可以为空)。
- 模块：python 中的模块 (module) 就是同名的 py 文件。
- 属性：module 的属性 (attribute) 可以是函数、类、变量等。

package 的拓扑结构相当于目录的结构，即对应 `package1.package2` (`package2` 是 `package1` 的子 package)，目录 `package1` 有子目录 `package2`。在 `__init__.py` 中可以 import 其他的 package/module/attribute，从用户角度来看，类似于文件系统中的 link 或快捷方式。

package 也是 module，其内容是 package 对应目录中的 `__init__.py` 文件(你可以通过打印 `module.__file__` 来确认这点)。如果其中 import 了下属 module 的 attribute，则相当于该 package 对应的 module 也有了这个 attribute (就像快捷方式)。

模块文件 (`xxx.py` 或者 `__init__.py`) 中的 `__all__` 属性 (是一个 list 类型的变量) 用于显式定义 `from module import *` 时暴露出来的接口。如果没有 `__all__`，则：

- 对于 module 来说，其中所有非下划线开头的成员都会被暴露出来
 - 对于 package 来说，其所有非下划线开头的成员，和下级模块都会被暴露出来
- `from package.module import *` 受 `__all__` 的约束，只会导入被 `__all__` 暴露出来的

attribute 和子 module，但是在直接 `from package.module import xxx` 时，xxx 不受 `__all__` 约束。

import 和 from import :

- `from A.B import C` : A 必须是 package，B 可以是 package/module，C 可以是 package/module/attribute/*，用时直接用 C
- `import A.B.C` : A、B 必须是 package，C 可以是 package/module，用时需要写全 A.B.C (这个操作实际上相当于 `import A`，因为之后也可以使用 A 的其他 sub-package 或者 module)

导入 package:

- `import package` : 导入 package (作为 module)，之后可以直接用 package
- `import package1.package2` : 导入 package1.package2 进入命名空间，之后可以通过 package1.package2 使用 package2。该操作实际是导入整个 package1 进入命名空间，也可以通过 package1.xxx 来使用 package1 的其他成员。
- `from package1 import package2` : 直接导入 package2 进入命名空间，之后可直接使用 package2

导入 module :

- `import package.module` : 导入 package.module 进入命名空间 (其实是导入整个 package 进入命名空间)，之后可以通过 package.module 使用 module
- `from package import module` : 直接导入 module 进入命名空间，之后可直接使用 module

导入 attribute :

- `from package.module import attribute` : 导入 attribute，之后可以直接用
- `from package import attribute` : 同上，因为在 package 的 `__init__.py` 中导入 module，因此此处的 module 可以忽略

导入所有成员 :

- `from package1.package2 import *` : 导入 package2 下所有的 module 和 attribute，通过 package2 下 `__init__.py` 文件中的 `__all__` 变量指定，如果没有则导入所有非下划线开头的成员
- `from package.module import *` : 导入 module 下所有的 attribute

错误格式 :

- `import package.attribute` : 错误，import a.b.c 中 c 只能是 module/package
- `import package.module.attribute` : 错误，只有 package 之后才能跟 “.”
- `from package import module.attribute` : 错误，import 之后不能有 “.”

总结一下就是：

- 可以 `import package/module`
- 可以 `from package/module import package/module/attribute`
- “.” 只能在 package 之后

- from xxx import yyy 中，import 之后不能有 “.”

(二) 类和对象

Python 中，类 (Class) 也是一个对象 (Object)

1 · 对象

对象是 Python 组织数据的形式，所有的数据都是对象 (object)，即某个类 (Class) 的 instance。即便是整数，甚至整数常量这种简单的数据类型（其类为<class ‘int’ >）。每个对象都有 ID (identity)，类型 (type) 和值 (value)。这三者中，只有 value 是可以变化的，另外两个都是不可变的。

ID 可以被视为对象在内存中的位置，内嵌函数 id() 返回了对象的 ID，而 is 操作符则比较了两个对象的 ID 是否一致。

- type() 返回了对象的类型（即所属的类 Class）：type(3)==int，除了常量之外（变量、类、包等）也可以用 xxx.__class__ 得到同样结果
- 而类型本身也是一个对象，其类型是“type”：type(type(3))==type
- 而 type 的所属类型也是 type：type(type(type(3)))==type

某些对象包含了其他对象（的引用），它们被称为容器 (Container)，比如 list，tuple，dict 等，这些引用是容器的值的一部分。

下文中：

- 对象 (object)、实例 (instance) 基本同义
- 类型 (type)、类 (class) 基本同义，在描述元类的时候略有区别：类型可以用来形容元类，而类不行

2 · 类=对象

定义某个类的一个对象，可以用如下语句：

```
object = class(args...)
```

比如：

```
a = int(4)
```

或者

```
b = list([1,2,3])
```

由于 Python 中所有的类也都是对象，因此这里的 class() 相当于 class.__call__(), 即一

一个 object 的可调用函数：

```
a = int.__call__(4)
b = list.__call__([1,2,3])
```

Python 中所有的类也都是对象，而这些对象的类型是 ‘type’，或者说，所有类都是 ‘type’ 的实例（包括 ‘type’ 本身也是 ‘type’ 的实例），如此一来，定义一个新的类，相当于：

```
class = type(classname,  superclass, attributedict)
```

或者

```
class = type.__call__(classname,  superclass, attributedict)
```

例如：

```
class NewClass:
    data = 1
```

相当于：

```
NewClass = type( "NewClass" , () , { 'data' :1})
```

相当于：

```
NewClass = type.__call__( "NewClass" , () , { 'data' :1})
```

type 是一个 **metaclass**，即元类，元类是类的类型，元类和类的关系，一如类和对象的关系。

3 · metaclass

元类（metaclass）的对象是普通类，但是普通类的类可以不是 ‘type’，也就是说除了 ‘type’ 还可以定义其他的 metaclass（通常这些 metaclass 的类型也是 ‘type’）。

定义新的 metaclass，并将类的元类设置为这个新 metaclass 的目的，通常是为了修改创建类的对象时的过程。

在类的定义中，通过声明 `metaclass=MyMeta`，会使得：

- 在生成类的时候，如果有，MyMeta 的`__new__()`和`__init__()`会被调用
- 在生成类的对象时，如果有，MyMeta 类的`__call__()`替代 `type.__call__()`被调用：

例如：

```
class Foo(metaclass=MyMeta)
```

会依次调用以下函数来创建 Foo：

1. `type.__call__()`：MyMeta 的类型是 ‘type’
 - (1) `MyMeta.__new__()`
 - (2) `MyMeta.__init__()`

```
foo = Foo()
```

会依次调用如下函数来创建 Foo 的实例 foo：

1. `MyMeta.__call__()`

- (1) Foo.__new__()
- (2) Foo.__init__()

混淆注意：

声明某个类的元类在 Python3 的用法是：

```
class Foo(metaclass=MyMeta)
```

在 python2.7 的用法则是如下：

```
class Foo:  
    __metaclass__=MyMeta
```

混淆注意：

注意区分“父类-子类”和“类型-对象”的关系，父类-子类能构成若干层的继承结构，但是（在不指定 metaclass 的情况下）所有这些类的类型都是‘type’，而子类缺省和父类的元类一致，但可以通过声明 metaclass 来改变，简单来说，这是两个不同维度的概念。

参考：

<https://zhuanlan.zhihu.com/p/98440398>

<https://lotabout.me/2018/Understanding-Python-MetaClass/>

<https://www.cnblogs.com/chengege/p/11102802.html>

4 · 实例化的过程

我们知道，如果类 Foo 的定义中有__call__函数的实现，则 Foo 的对象 foo 是可以调用的，即：

```
foo()
```

相当于调用 foo 所属类 Foo 的__call__函数：

```
Foo.__call__()
```

而当一个类 Foo 的对象 foo 生成的时候，发生的函数调用是这样的：

```
foo = Foo()
```

1. 调用 Foo 的所属类型的__call__ : type.__call__()
 - (1) 调用 Foo 的__new__函数 : Foo.__new__()
 - (2) 调用 Foo 的__init__函数 : Foo.__init__()

当定义一个 metaclass MyMeta 的时候：

```
class MyMeta(type):
```

相当于：

```
MyMeta = type(“MyMeta”, ...)
```

1. 调用 type 所属类型（仍是 type）的__call__ : type.__call__()
 - (1) 调用 type 的__new__函数 : type.__new__()
 - (2) 调用 type 的__init__函数 : type.__init__()

当定义一个 metaclass 为 MyMeta 的类的时候：

```
class Foo(metaclass=MyMeta):
```

相当于：

```
Bar = MyMeta("Bar", ...)
```

1. 调用 MyMeta 所属类型 (type) 的 __call__ 函数：type.__call__()

(1) 调用 MyMeta 的 __new__ 函数：MyMeta.__new__()

(2) 调用 MyMeta 的 __init__ 函数：MyMeta.__init__()

当生成 Bar 的对象 bar 的时候：

```
bar = Bar()
```

1. 如果 Bar 的所属类型 MyMeta 有定义 __call__，则调用：MyMeta.__call__()

(1) 如果其中调用了 self.__new__ 函数，则调用：Bar.__new__()

(2) 如果其中调用了 self.__init__ 函数，则调用：Bar.__init__()

2. 否则，则调用 type.__call__()

(1) 调用 type 的 __new__ 函数：type.__new__()

(2) 调用 type 的 __init__ 函数：type.__init__()

(三) 类

1 · Final 和 Virtual

先来看看 C++ 和 Java 中的关于类成员函数的特点，并同时对 Python 中的情况做比较，下文混合使用两种语言中的名称，以便强调其特点（Final 函数，虚函数，纯虚函数）：

- **Final 函数**：指的是函数不会被子类的继承覆盖
 - C++ 的成员函数缺省就是这种函数，没有特别的名称
 - Java 中被称为 **final method**，需要加 **final** 前缀来定义。
 - Python 中没有 **Final** 函数，但可以通过 metaclass 实现（比较复杂）
- **虚函数**：函数会被子类的继承覆盖
 - C++ 需要加 **virtual** 前缀来定义，被称为 **virtual function**（虚函数）
 - Java 中的成员函数缺省就是这种函数，没有特别的名称。
 - Python 中类的成员函数缺省就是虚函数
- **纯虚函数**：函数只有声明没有实现，需要子类实现
 - C++ 中需要加 **virtual** 前缀和 =0 后缀来定义，被称为 **pure virtual function**（纯虚函数）
 - Java 中称为 **abstract method**（抽象方法），需要加 **abstract** 前缀来定义。
 - 纯虚函数在 Python 中被称为 **abstract method**（抽象方法），可以通过 **@abstractmethod** 修饰符来修饰函数，但仅仅如此无法阻止包含这种函数的类被直接实例化（需配合 ABCMeta 来使用）。

再来看看 C++ 和 Java 中关于类的特点，以及 Python 的比对：

- 普通类：可以被继承，可以被实例化
 - C++ 中的类缺省就是这种类
 - Java 中的类缺省就是这种类
 - Python 中的类缺省就是这种类
- Final 类：可以被实例化，但是不能被继承
 - C++ 中直到 C++11 标准之后，才有关键字 **final** 后缀来定义
 - Java 中通过 **final** 前缀来定义这种类，被称为 **final class (final 类)**
 - Python 中没有 **Final** 类，但可以通过 metaclass 实现
- 抽象类：可能包含纯虚函数，不能被实例化
 - C++ 中被称为 **abstract base class (抽象基类)**，没有专门的关键字描述，类中只要包含纯虚函数就成为抽象基类，子类只有实现了所有纯虚函数才可以实例化
 - Java 中被称为 **abstract class (抽象类)**，通过 **abstract** 前缀来定义。且任何包含纯虚函数（抽象方法）的类都自动成为抽象类。不包含纯虚函数但有 **abstract** 修饰的类也不可以被实例化。
 - Python 中被称为 **abstract base class (抽象基类)**，没有专门的关键字来声明，但是可以通过以下方法来实现抽象类，注意这两者要同时使用，方可阻止类的实例化：
 - ◆ 类的声明中设置元类：`metaclass=abc.ABCMeta`
 - ◆ 并且用 `@abstractmethod` 修饰纯虚函数（抽象方法）
- 接口：类似抽象类，但定义中仅包括纯虚函数，解决了单继承中的多继承问题。
 - C++ 中是多继承的，没有接口的概念，不过有个类似的东西被称为 **virtual base class (虚基类)**，本身的定义没有关键字描述，通过在子类继承的时候加上 **virtual** 关键字实现。虚基类可以解决基类共享的问题，这个问题在单继承的 Java 中不存在。
 - Java 中使用 **interface** 关键字定义接口，成员函数（可以不用显式指定）都是纯虚函数（抽象方法），成员变量只能是 **static** 和 **final** 的。Java 是单继承的，因此通过接口可以实现类似于 C++ 多继承的特性。
 - Python 中没有接口，不过 Python 是多继承的，接口意义不大

简化为下表：

名称	特点	C++	java	python
Final 函数	不会被子类覆盖	缺省	final method (前缀 final)	无
虚函数	会被子类覆盖	虚函数 (前缀 virtual 关键字)	缺省	缺省
纯虚函数	只有声明没有实现，必须被子类实现才可以实例化	纯虚函数 (前缀 virtual , 后缀 =0)	抽象函数 (前缀 abstract)	抽象方法 (用 <code>@abstractmethod</code> 修饰的函数，但仅仅如此无法阻止实例化)
普通类	可以被实例化 可以被继承	缺省	缺省	缺省

Final 类	不能被继承	后缀 final (C++11 之后)	前缀 final	无 (可以通过 metaclass 实现)
抽象类	带有纯虚函数的类，不能被实例化	抽象类 (包含纯虚函数，没有特别的关键字)	抽象类 (前缀 abstract)	抽象基类，实现如下： metaclass=abc.ABCMeta 给需要抽象的方法加上 @abstractmethod 修饰符
接口	只包含纯虚函数	无，但有类似的虚基类 (继承时加 virtual 前缀)	接口 (interface 关键字)	无

2 · 抽象基类

Python 中定义一个抽象基类 (Abstract Base Class) 没有关键字来实现，需要通过设置类的元类来实现：

- 设置 metaclass 为 abc.ABCMeta
- 函数要加上@abstractmethod 修饰符

注意这两个操作都需要，只采取其中任意一种，类都可以被实例化。

```
from abc import ABCMeta, abstractmethod
class C(metaclass=ABCMeta):
    @abstractmethod
    def func(self):
        pass
```

或者可以继承 ABC 作为其父类，这是个语法糖，ABC 是个元类已经设置为 ABCMeta 的类，因此继承 ABC 相当于设置 metaclass 为 ABCMeta：

```
from abc import ABC, abstractmethod
class C(ABC):
    @abstractmethod
    def func(self):
        pass
```

对应文件为 abc.py，给出了抽象基类 (ABC, abstract base classes) 的定义，collections.abc 中的各种具体的 ABC 均来源于此。这其中主要包含了：

- **abc.ABCMeta** : ABC 的元类 (metaclass)，定一个 ABC 需要通过如下格式实现：
`class MyABC(metaclass=ABCMeta)`
- **abc.ABC** : ABC 的助手类 (helper class)，定义一个 ABC 也可以通过继承该类实现：
`class MyABC(ABC)`
- **abc.abstractmethod()** : 函数修饰符，用于定义一个函数为抽象方法

参考：

<https://docs.python.org/3/library/abc.html>

<https://docs.python.org/3/glossary.html#term-abstract-base-class>

除了使用 **ABCMeta** 之外，还有两种方法可以近似模拟抽象基类——在基类创建的对象调用函数的时候报错，而不是在创建对象的时候。

1. 在基类的方法中使用断言 assert，使得基类的对象调用函数的时候报错

```
class BaseClass(object):
    def func(self):
        assert False, "Abstract class!"
```

2. 使用 NotImplementedError 异常，使得基类的对象调用函数的时候报错

```
class BaseClass(object):
    def func(self):
        raise NotImplementedError("Not implemented!")
```

3 · 内嵌抽象类 (Collections ABC)

内嵌抽象基类 (Collections ABC)，是系统中自带的一些抽象基类 (ABC, Abstract Base Class)。

对应文件为 `_collections_abc.py`，这个模块提供了一系列的抽象基类，这些抽象基类用于判断某个类是否实现了一个特定的抽象方法。比如 `Hashable` 类可用于判断某个类是否实现了 `__hash__()` 函数：

```
from typing import Hashable
isinstance(obj, Hashable)
```

换句话说，只要自定义的类实现了这些特定的抽象方法（比如 `__hash__()`），无须显式的继承这些类（比如 `Hashable`），`isinstance()` 函数就会返回 `True`，这些自定义的类被称为虚拟子类 (virtual subclass)。

参考：

<https://docs.python.org/3/glossary.html#term-abstract-base-class>

下面为内嵌 ABC 的列表：

ABC	Inherits from	Abstract Methods	Mixin Methods
Container		<code>__contains__</code>	
Hashable		<code>__hash__</code>	
Iterable		<code>__iter__</code>	
Iterator	Iterable	<code>__next__</code>	<code>__iter__</code>

Reversible	Iterable	<code>__reversed__</code>	
Generator	Iterator	<code>send</code> , <code>throw</code>	<code>close</code> , <code>__iter__</code> , <code>__next__</code>
Sized		<code>__len__</code>	
Callable		<code>__call__</code>	
Collection	Sized, Iterable, Container	<code>__contains__</code> , <code>__iter__</code> , <code>__len__</code>	
Sequence	Reversible, Collection	<code>__getitem__</code> , <code>__len__</code>	<code>__contains__</code> , <code>__iter__</code> , <code>__reversed__</code> , index, and count
MutableSequence	Sequence	<code>__getitem__</code> , <code>__setitem__</code> , <code>__delitem__</code> , <code>__len__</code> , <code>__insert__</code>	Inherited Sequence methods and append, reverse, extend, pop, remove, and <code>__iadd__</code>
ByteString	Sequence	<code>__getitem__</code> , <code>__len__</code>	Inherited Sequence methods
Set	Collection	<code>__contains__</code> , <code>__iter__</code> , <code>__len__</code>	<code>__le__</code> , <code>__lt__</code> , <code>__eq__</code> , <code>__ne__</code> , <code>__gt__</code> , <code>__ge__</code> , <code>__and__</code> , <code>__or__</code> , <code>__sub__</code> , <code>__xor__</code> , and <code>isdisjoint</code>
MutableSet	Set	<code>__contains__</code> , <code>__iter__</code> , <code>__len__</code> , add, discard	Inherited Set methods and clear, pop, remove, <code>__ior__</code> , <code>__iand__</code> , <code>__ixor__</code> , and <code>__isub__</code>
Mapping	Collection	<code>__getitem__</code> , <code>__iter__</code> , <code>__len__</code>	<code>__contains__</code> , keys, items, values, get, <code>__eq__</code> , and <code>__ne__</code>
MutableMapping	Mapping	<code>__getitem__</code> , <code>__setitem__</code> , <code>__delitem__</code> , <code>__iter__</code> , <code>__len__</code>	Inherited Mapping methods and pop, popitem, clear, update, and setdefault
MappingView	Sized		<code>__len__</code>
ItemsView	MappingView, Set		<code>__contains__</code> , <code>__iter__</code>
KeysView	MappingView, Set		<code>__contains__</code> , <code>__iter__</code>
ValuesView	MappingView, Collection		<code>__contains__</code> , <code>__iter__</code>
Awaitable		<code>__await__</code>	
Coroutine	Awaitable	<code>send</code> , <code>throw</code>	<code>close</code>
AsyncIterable		<code>__aiter__</code>	
AsyncIterator	AsyncIterable	<code>__anext__</code>	<code>__aiter__</code>
AsyncGenerator	AsyncIterator	<code>asend</code> , <code>athrow</code>	<code>aclose</code> , <code>__aiter__</code> , <code>__anext__</code>

参考：

<https://docs.python.org/3/glossary.html#term-abstract-base-class>

<https://docs.python.org/3/library/collections.abc.html>

混淆注意！

跟抽象基类相关的定义的包括：

- 抽象基类（即不能被实例化的抽象类）这个定义，又分为：
 - 用 ABCMeta 等手段实现，无法实例化
 - 虽然编程中仍然可以实例化，但在编码的架构设计中被定义为抽象基类
- 内嵌抽象类 (Collections ABC)
- abc 中的 ABC, ABCMeta 这些用于实现抽象基类的工具类/元类

4 · 内嵌类型 (built-in types)

<https://docs.python.org/3/library/stdtypes.html#>

5 · 类的层次

对象的类型 (`type`) 即其所属的类 (`class`)，这也决定了该对象所能支持的操作。内嵌函数 `type(object)` 可以得到对象所属的类。

类的层次关系：

- `NoneType`：只有一个对象，值唯一，即 `None`
- `NotImplementedType`：只有一个对象，值唯一，即 `NotImplemented`
- ellipsis :
- `numbers.Number` :
 - `numbers.Integral`：整数
 - ◆ `Integers`：即 `int` 类型
 - ◆ `Booleans`：即 `bool` 类型
 - `numbers.Real`：即 `float` 类型
 - `numbers.Complex`：即 `complex` 类型
- Sequences：有序的有限元素的集合，其元素可以用非负整数索引（即从 0 开始）
 - 不可变 Sequences：创建之后不可修改
 - ◆ `Strings`：即 `str` 类型，可通过 `str.encode()` 函数编码为 `bytes` 类型
 - ◆ `Tuples`：即 `tuple`，元组类型
 - ◆ `Bytes`：即 `bytes` 类型，可通过 `bytes.decode()` 函数变为 `str` 类型
 - 可变 Sequences：创建之后可以修改
 - ◆ `Lists`：即 `list` 类型
 - ◆ `Byte Arrays`：即 `bytearray` 类型，与 `bytes` 类型的区别在于可以修改
- Set 类型：无序的，有限元素的集合，可 `len()`，可 `iter()`
 - Sets：可变集合，通过 `set()` 创建
 - Frozen Sets：不可变的集合，通过 `frozenset()` 创建，可 `hash()`
- Mappings：映射，目前只有字典一个子类
 - 字典类：即 `dict` 类型
- Callable 类型：可调用的类型
 - 自定义函数：用户通过 `def` 定义
 - 实例方法：

- generator 函数：使用 `yield` 语句的函数
- Coroutine 函数：协程函数
- 异步 generator 函数：
- built-in function：内嵌函数
- built-in method：内嵌方法
- Class：就是类，所有的类都是可调用的，返回该类的对象
- class instance：类的实例，可以通过实现 `__call__()` 来变得可调用
- 模块：Python 中的模块都是 `module` 类的对象
- Custom classes：
- Class instance：
- I/O objects：
- Internal types：
 - Code objects：
 - Frame objects：
 - Traceback objects：
 - Slice objects：
 - Static method objects：
 - Class method objects：

参考：

<https://docs.python.org/3/reference/datamodel.html>

(四) 泛型别名

泛型别名（Generic Alias）通过给一个类（通常是容器）做下标创建，比如 `list[int]`。泛型别名没有强制性。

通常来说，给容器类的对象下标会调用类特殊方法 `__getitem__()` 函数，而给容器类做下标，则会调用类特殊方法 `__class_getitem__()`，这会返回一个泛型别名对象。

`T[X, Y, ...]`

表示了包含了 X, Y 等类型的容器 T，例如（List 和 Dict 为 list 和 dict 类型的别名）：

```
from typing import List, Dict
List[float]
Dict[int, str]
```

以上方式在 python3 中通用，下面不用类型别名直接用类型名的方法在 3.9 之后才合法：

```
list[float]
dict[int, str]
```

Python 并不做类型检查，因此下列语句是合法的：

```
>>> t = list[str]
>>> t([1, 2, 3])
[1, 2, 3]
```

而在生成实例的时候会消除类型参数的信息：

```
>>> t = list[str]
>>> type(t)
<class 'types.GenericAlias'>
>>> l = t()
>>> type(l)
<class 'list'>
```

泛型别名有如下一些属性：

- `generalalias.__origin__`：指代不包含参数的原泛型类型

```
>>> list[int].__origin__
<class 'list'>
```

- `generalalias.__args__`：元组类型，泛型类的参数类

```
>>> dict[str, list[int]].__args__
(<class 'str'>, list[int])
```

- `generalalias.__parameters__`：

参考：

<https://docs.python.org/3/library/stdtypes.html#generic-alias-type>

(五) 方法

1 · 标准库抽象类、内嵌函数、类特殊方法

- 标准库抽象类（standard library abstract class），指的是标准库中定义的一系列抽象基类。
- 内嵌函数（built-in function），指的是 python 的内嵌命名空间（built-in namespace）中自带的函数，不属于任何类
- 类特殊方法，指的是具有特殊名字的类，通常以“`__`”（两个下划线）开头和结尾，通常是标准库抽象类中的抽象方法。

这三者在 Python 中有相关性，某个类对内嵌函数的支持需要类实现某个内置虚函数，而某个标准库抽象类可以用于判断是否实现了这个虚函数。

举个例子，关于哈希，以下三者等价：

- 系统的内嵌函数 `hash(object)` 能运行成功
- `object` 所属的类实现了 `__hash__()` 这个内置虚函数（或者叫类特殊方法）
- 该类是 `Hashable` 这个标准库抽象类的子类（或者 `object` 是 `Hashable` 的实例）

2 · 内嵌函数 (Built-in Functions)

内嵌函数是内嵌命名空间 (`built-in namespace`) 中自带的函数，不属于任何类。

Python 有一系列内嵌函数，包括：

- `abs(x)`：返回一个数字的绝对值，如果 `x` 定义了 `__abs__()`，则返回 `x.__abs__()`
- `all(iterable)`：参数为迭代器，如果该迭代器产生的所有元素都为 `True` 则返回 `True`，否则返回 `False`
- `any(iterable)`：参数为迭代器，如果迭代器的任意元素为 `True` 则返回 `True`，否则返回 `False`
- `ascii()`:
- `bin()`
- `bool()`
- `breakpoint()`
- `bytearray()`
- `bytes()`
- `callable()`：返回布尔值，表示是否可以调用。所有类都是可以调用的，调用类返回该类的对象。对象是否可以被调用基于该类是否实现了 `__callab1e__()` 成员函数。
(也可以通过判断该类是否 `Callable` 的子类来判断)
- `chr()`
- `@classmethod`：函数修饰符，用于将类的成员函数声明为类函数，不跟对象绑定，隐性的第一个参数也不是对象 (`self`)，而是类 (`cls`)
- `compile()`
- `complex()`
- `delattr()`
- `dict()`
- `dir()`：不带参数则显示当前 `scope` 中的内容，带参数则返回给定 `object` 的属性
- `divmod()`
- `enumerate()`
- `eval()`
- `exec()`
- `filter()`
- `float()`
- `format()`
- `frozenset()`
- `getattr()`
- `globals()`
- `hasattr()`
- `hash()`
- `help()`
- `hex()`
- `id()`

- `input()`
- `int()`
- `isinstance(object, class)`: 判断 `object` 是否是 `class` 的实例，返回 `bool` 值
- `issubclass(class1, class2)`: 判断 `class1` 是否是 `class2` 的子类，返回 `bool` 值
- `iter(object)`: 返回该 `object` 的迭代器，`object` 需满足迭代协议（由 `__iter__()` 函数实现），或者满足序列协议（由 `__getitem__()` 实现）
- `len()`
- `list()`
- `locals()`
- `map()`
- `max()`
- `memoryview()`
- `min()`
- `next()`
- `object()`
- `oct()`
- `open()`
- `ord()`
- `pow()`
- `print()`
- `property()`
- `range()`
- `repr() :`
- `reversed()`
- `round()`
- `set()`
- `setattr()`
- `slice()`
- `sorted()`
- `staticmethod()`
- `str()`
- `sum()`
- `super()`
- `tuple()`
- `type()`
- `vars()`
- `zip()`
- `__import__()`

参考：

<https://docs.python.org/3/library/functions.html>

3 · 类特殊方法 (Special method)

类可以通过实现特别的类成员方法 (special method，这些方法的方法名前后有两个下划线，Python 中此类名称经常有特殊意义)，使得这些方法在某些特殊时候被回调，比如用于支持某些内嵌函数 (len(), dir() 等)，或者支持某些语句 (with, if 等)，这也是 Python 实现运算符重载的方式。

举个例子，如果对象 x 的类实现了 __getitem__() 方法，则 x[i] 相当于 x.__getitem__(i)，即 type(x).__getitem__(x, i)。（可以用 list 类型验证此方法）

参考：

<https://docs.python.org/3/reference/datamodel.html#special-method-names>

(1) 基本方法

- object.__new__(cls, ...): 静态方法，创建一个新的实例，第一个参数是类名，构造一个新的对象的时候，在__init__()之前被调用。其常用的一个用法是，子类的新建对象调用父类的该方法，即 super().__new__()。
- object.__init__(self, ...): 构造函数，被调用的时候对象已经创建（通过__new__() 函数），即第一个参数 self。常用的用法是在子类的构造函数中调用，即 super().__init__()。
- object.__del__(self): 对象被销毁之前调用，类似于解构函数
- object.__repr__(self): 调用内嵌函数 repr(object) 时被调用，用于返回该对象的“官方”字符串表示。
- object.__str__(self): 调用内嵌函数 str(object) 和 print(object) 的时候被调用，用于返回该对象的“非正式”或者“打印友好”的字符串表示。
- object.__bytes__(self): 字节序化，调用内嵌函数 bytes(object) 的时候被调用，返回对象的字节串 (byte-string) 表示 (bytes 类型)。
- object.__format__(self, format_spec): 调用内嵌函数 format(object) 的时候被调用，目前等同于 str(object)
- object.__lt__(self, other): 对“<”操作符的重载
object.__le__(self, other): 对“≤”操作符的重载
object.__eq__(self, other): 对“==”操作符的重载
object.__ne__(self, other): 对“!=”操作符的重载

`object.__gt__(self, other)`: 对“>”操作符的重载
`object.__ge__(self, other)`: 对“>=”操作符的重载

- `object.__hash__(self)`: 哈希值的计算，调用内嵌函数 `hash(object)` 的时候被调用，返回对象的哈希值
- `object.__bool__(self)`: 判断语句的时候（比如 `if`），以及调用内嵌函数 `bool(object)` 的时候被调用（如果 `__bool__()` 未被定义，则 `__len__()` 被调用）

(2) 属性访问

- `object.__getattr__(self, name)`: 当 `__getattribute__()` 或者 `__get__()` 调用出现 `AttributeError` 异常时被调用
- `object.__getattribute__(self, name)`: 返回对象的某个属性
- `object.__setattr__(self, name, value)`: 设置对象的某个属性
- `object.__delattr__(self, name)`: 当 `del object.name` 的时候被调用，删除对象的某个属性
- `object.__dir__(self)`: 调用内嵌函数 `dir(object)` 的时候被调用，返回对象的所有属性
- `object.__get__(self, instance)`:
- `object.__set__(self, instance, value)`:
- `object.__delete__(self, instance)`:
- `object.__set_name__(self, owner, name)`:

(3) 其他

- `object.__call__(self, args...)`: 实现此方法则对象可被调用，当实例被作为一个函数调用的时候，`__call__()` 被调用，即 `object(args...)`。
- `object.__len__(self, name)`: 调用内嵌函数 `len(object)` 的时候被调用，`bool` 判断的时候如果没有 `__bool__` 函数而有 `__len__` 函数，则会调用该函数判断 `bool` 值，0 返回 `False`。
- `object.__getitem__(self, key)`: 实现了 `object[key]` 的重载

- object.__setitem__(self, name, value)：实现了 object[key]=value 的重载
- object.__iter__(self)：参考迭代器
- object.__contains__(self, item)：实现了 item in object 的重载，返回布尔值

(4) with 语句上下文

- object.__enter__(self,)：进入 object 的 with 上下文环境时被调用
- object.__exit__(self, exc_type, exc_value, traceback)：退出 object 的 with 上下文环境时被调用

(5) 协程相关

- object.__await__(self) :
- object.__aiter__(self) :
- object.__anext__(self) :
- object.__aenter__(self) :
- object.__aexit__(self, exc_type, exc_value, traceback)

hash(), memoryview(), set(), a11(),

对于类 CLASS，可以实现如下虚函数，这些虚函数会用于对于 CLASS 类型的对象 object

_len__()：对象的长度，len(object)时候调用

_bool__()：判断 object 是否为 True 时候调用 (if object)，如果没有实现返回 True

4 · 修饰符

修饰符 (decorator) 其实是个语法糖，修饰符本质上也是个函数 de，这个函数 d 的参数是个函数 func (即被修饰的函数)，返回另一个函数 wrapper，这个函数会被用于替代原来的 func，使得每次调用 func 的时候都实际是在调用 wrapper，而在使用修饰符的时候都相当于调用了一次 de，即使用的时候：

```
@de
def func():
    ...

```

相当于：

```
def func():
    ...

```

```
func = de(func)
```

即用 `de` 函数返回的函数替代 `func`。

举个简单的例子：

```
def log(f): #定义修饰符函数
    def wrapper():
        print( "start" )
        return f() #当然你也可以完全不调用原函数 f，不过那就失去意义了
    return wrapper #wrapper 用于替换被修饰函数
```

```
@log
def func():
    print( "function" )

func()
```

运行后打印：

```
start
function
```

参考：

<https://docs.python.org/3/glossary.html#term-decorator>

(六) 迭代器

迭代器这玩意单独拿出来说，是因为有点绕。简单来说如下：

有 2 个和迭代相关的概念，分别是 `Iterator`（迭代器）和 `Iterable`（迭代器的容器）：

- `Iterable`：任何类实现了`__iter__()`方法即是内嵌抽象类 `Iterable` 的虚拟子类，该方法返回一个 `Iterator`，检查 `isinstance(obj, Iterable)` 可以判断 `obj` 是否有 `__iter__` 方法，但不能检查通过`__getitem__`方法迭代的类，唯一可靠判断类是否 iterable 的方式是通过 `iter()`
- `Iterator`：任何类实现了`__iter__()`和`__next__()`方法，即是内嵌抽象类 `Iterator` 的虚拟子类（从而必然是 `Iterable` 的虚拟子类）：
 - `__iter__()`：`Iterator` 的该方法返回自身
 - `__next__()`：返回当前元素，并将 `Iterator` 指向下一个元素

如果你仍然觉得不满足，可以继续往下看。

1 · 定义

python 中迭代器相关的一些定义。

(1) iterable

可以每次返回其一个成员的对象，比如所有的 sequence 类型（list, tuple, str），某些非 sequence 类型（比如 dict），或者任何定义了`__iter__()`方法或者`__getitem__()`方法的。

iterable 可以被用于 for 循环。当一个 iterable 对象被作为参数传递给内嵌函数`iter()`时，返回该对象的 iterator。当使用 iterable 的时候，通常无须调用`iter()`或者自己来处理 iterator，for 语句会自动处理这些：创建一个临时无名变量来表示 iterator，直到循环结束。

参考：

<https://docs.python.org/3/glossary.html#term-iterable>

(2) iterator

一个表示数据流的对象，持续的调用 iterator 的类特殊方法`__next__()`，或者将 iterator 传递给内嵌函数`next()`，会持续的返回流中的数据。当没有后续的数据时，会引起`StopIteration` 异常。

iterator 必须要实现`__iter__()`方法，以便返回自身，因此所有的 iterator 也是 iterable。可以被用在所有 iterable 的场景，但也有例外，当一个容器对象（比如 list）每次被传递给`iter()`的时候（或者在 for 循环中），都会返回一个新的 iterator，如果此时用该 iterator 试图继续，则会使得其看上去像一个空的容器。（FIXME：存疑，测试没有此问题）

参考：

<https://docs.python.org/3/glossary.html#term-iterator>

(3) generator

一个返回 genertor iterator 的函数。跟普通函数的区别在于包含了`yield`语句，`yield`语句产生一系列数值，可以用于 for 循环，或者通过内嵌函数`next()`获得。

通常指代一个 generator 函数，但是在某些情况下也可以指代 generator iterator。

参考：

<https://docs.python.org/3/glossary.html#term-generator>

(4) generator iterator

generator 函数产生的对象。每次执行到 yield 语句，会暂停执行续列，保留当前的上下文，直到下次执行的时候恢复（相较于此，函数的每次执行都会从头开始）。

参考：

<https://docs.python.org/3/glossary.html#term-generator-iterator>

2 · 内嵌抽象基类

(1) Iterable

描述 iterable 的内嵌抽象基类，调用 `isinstance(obj, Iterable)` 可以用于验证 obj 是否实现了类特殊方法 `__iter__()`，但不能检查通过 `__getitem__()` 迭代的类，唯一可靠判断类是否 iterable 的方式是调用 `iter()`。

`Iterable` 可以被视为是 `Iterator` 的容器，通过将 `Iterable` 作为参数传给 `iter()`，返回一个 `Iterator`。

参考：

<https://docs.python.org/3/library/collections.abc.html#collections.abc.Iterable>

(2) Iterator

描述 iterator 的内嵌抽象基类，调用 `isinstance(obj, Iterator)` 可以用于验证 obj 是否实现了类特殊方法 `__iter__()` 和 `__next__()`

参考：

<https://docs.python.org/3/library/collections.abc.html#collections.abc.Iterator>

(3) Reversible

`Reversible` 实现了 `__reversed__()` 的 `Iterable` 的 **内嵌抽象基类**，调用 `isinstance(obj, Reversible)` 可以用于验证 `obj` 是否实现了 **类特殊方法** `__iter__()` 和 `__reversed__()`。

参考：

<https://docs.python.org/3/library/collections.abc.html#collections.abc.Reversible>

(4) Generator

描述 `generator` 的 **内嵌抽象基类**，它实现了 PEP 342 中定义的协议：在 `Iterator` 的基础上实现了 `send()`, `throw()` 和 `close()` 方法。

参考：

<https://docs.python.org/3/library/collections.abc.html#collections.abc.Generator>

3 · 类特殊方法

- `__iter__()`：返回一个迭代器 `Iterator`，实现该方法的类是 `Iterable` 的虚拟子类，`Iterator` 的该方法返回自身。
- `__next__()`：返回当前元素，并将 `Iterator` 指向下一个元素

三、框架&接口

本章介绍了和机器学习相关的库和框架，基本都是基于 Python 的。

(一) 框架简介

之后的小节列出了一些传统库和 DL 框架的接口，这其中绝大部分（除了 `Darknet`）均为 Python 接口。

当前最热门的三个深度学习框架为 Facebook 的 `Pytorch` 和 Google 的 `Tensorflow/Keras`，以及 Amazon 的 `MXNet/Gluon`，从普及程度上来讲，此三者为第一集团，其中 `MXNet/Gluon`

目前略微落后，而 **Pytorch** 有反超 **Tensorflow** 的趋势。

此外还有 **CNTK**（微软），**sklearn**（并非是个纯 DL 框架，而是个 ML 框架），**Theano**（已经停止更新）等。

而 **Darknet** 作为一个 YOLO 作者的独立作品，在 YOLO 用户中被使用。

框架按照其作用，可以大致分为通用框架、专用框架、边缘计算库和快捷接口，这其中除了边缘计算库之外，其他三种之间都略有交叉。

出品方	通用框架	快捷接口	边缘计算	专用框架
Google	Tensorflow	Keras	tflite	
Facebook	Pytorch(torch) Caffe/Caffe2(并入 Pytorch)	Fast.ai?	Pytorch Mobile	torchvision (图像处理) Detectron (已废弃) Maskrcnn-benchmark (已废弃) Detectron2 (CV) PySlowFast (视频理解) Prophet (时间序列)
Amazon	MXNet	gluon		gluoncv (CV) gluonnlp (NLP) gluon-ts (时间序列)
腾讯			ncnn (C++) TNN	
阿里			MNN (C++)	
百度	PaddlePaddle(飞桨)		Paddle-lite	
华为	MindSpore			
港中文大学				mmdetection (CV, 基于 Pytorch) mmaction (视频理解)

1 · 传统库

传统库指的是一些并非 NN，但和 NN 有关的库，都有 Python 的实现。

- **NumPy**：多维数组和矩阵运算的基础库，其核心数据结构是 ndarray (n-dimensional array)。
- **Matplotlib**：NumPy 的可视化界面库
- **SciPy**：开源的算法库和数学工具包，包含最优化、线性代数、积分、插值、FFT、信号和图像处理、常微分方程等。
- **Pandas**：基于 NumPy 的开源 Python 库，用于快速分析数据、数据清洗等工作。
- **Scikit-learn**：Scikit = SciPy ToolKit，开源的机器学习库，支持多种分类、回归、聚类算法，比如 SVM、随机森林、k-means 等。可以和 NumPy 及 SciPy 交互。
- **Scikit-image**：开源的 Python 图像处理库，可以和 NumPy 及 SciPy 交互。
- **PIL/Pillow**：PIL (Python Imaging Library)，开源的 Python 图像处理库，于 2011

年停止开发候，后续项目 Pillow 从 PIL 的仓库 fork 出来。

- **Opencv**：开源的机器视觉库（CV：Computer Vision），可以与 Numpy 交互

2 · 通用框架

通用框架是 NN 中最基础的库，通常实现了以下内容：

- 神经网络的各种基本构成，包括基类及各种实现，比如各种卷积层、池化层、损失函数、激活函数、优化方法、标准化方法等等
- 数据集相关的工具，下载、预处理、数据增广等等
- 有些通用框架还包括了更底层的东西（比如 Tensorflow）

通用框架可以用于训练（反向传播）和前向推导，目前最火的两个分别是来自 Google 的 Tensorflow 和来自 Facebook 的 Pytorch，第三名则是 Amazon 的 MXNet。

- **Tensorflow**：这是当前最火的机器学习库之一，来自谷歌大脑团队，于 2015 年开源。
- **PyTorch**：基于 Torch 的开源 Python 机器学习库，由 Facebook 开源。2018 年 Caffe2 并入 PyTorch，近年来有超越 Tensorflow 的趋势。
- **Caffe/Caffe2**：Convolutional Architecture for Fast Feature Embedding，开源的机器学习库，来自 UB Berkley。2017 年，Facebook 发布 Caffe2，加入了 RNN 等新功能，2018 年，Caffe2 并入 PyTorch。
- **CNTK**：Microsoft Cognitive Toolkit，微软推出的机器学习库，编程语言是 C++。
- **MXNet**：Apache 的开源机器学习框架，支持多种语言。
- **Darknet**：是 YOLO 系列作者 Joseph Redmon 开发的一个框架，C 语言实现，也有一个同名的基础卷积网络模型。
- **Theano**：开源的 Python 数学库，用于定义、优化、求值数学表达式。
- **PaddlePaddle**：百度推出的 DL 框架
- **MindSpore**：华为的 DL 框架

3 · 专用框架

专用框架、库、工具集，专门针对某些类别任务的高层次框架，通常基于某通用框架。专用框架通常包括：

- 面向该领域的特殊的层或者微结构
- 已经实现的该领域任务的网络模型，及其相关工具，比如训练器、权重下载等
- 该领域数据集的相关工具，比如下载、格式转换等
- 该领域性能的衡量工具

具体的专用框架有：

- **torchvision**：Facebook 出品，Pytorch 的子集，机器视觉方面的框架，包括：
 - 图像分类
 - 语义分割
 - 少量目标检测、实例分割、人体关键点检测

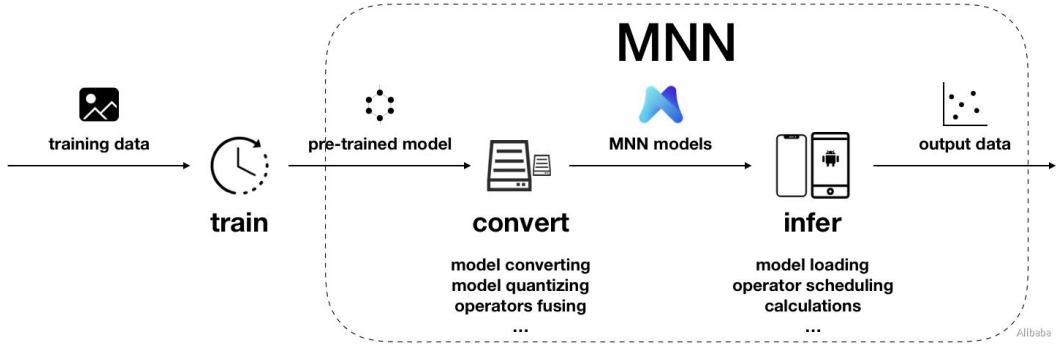
- 也包括少量视频理解方向的行为识别（R3D）。
- Detectron2 : Facebook 推出的图像处理的框架，基于 Pytorch，包括：
 - 图像分类
 - 目标检测
 - 图像分割：语义分割、实例分割、全景分割
 - 人体关键点检测
- PySlowFast : Facebook 推出的视频理解框架，基于 Pytorch，包括：
 - 行为识别：SlowFast、Slow、I3D、C2D、Non-local network
 - 时序行为检测：
- gluoncv : gluon (Amazon) 的一部分，机器视觉方面的框架，包括：
 - 图像分类
 - 目标检测
 - 图像分割
 - 姿态评估（人体关键点检测）
 - 行为识别（SlowFast、I3D）
- gluonts : gluon (Amazon) 的一部分，时间序列方面的框架
- gluonnlp : gluon (Amazon) 的一部分，NLP 方面的框架
- mmdetection : 香港中文大学推出的图像处理的工具集，基于 Pytorch，包括：
 - 图像分类
 - 目标检测
 - 图像分割：语义分割、实例分割、全景分割
- mmsegmentation : 香港中文大学推出的图像分割工具集，基于 Pytorch
- mmaction : 香港中文大学推出的视频理解的工具集，基于 Pytorch，包括：
 - 动作识别（Action Recognition）
 - 时序动作检测（Temporal Action Detection）
 - 时空动作检测（Spatio-temporal Action Detection）
- SimpleDet : 图森推出的图像处理的框架，基于 MXNet，包括：
 - 目标检测
 - 实例分割
- PyVideoResearch : 视频理解方面的框架
- SpaCy : 自然语言处理（NLP）开源库，偏重于产品。
- NLTK : Natural Language ToolKit，自然语言处理（NLP）开源库，偏重于研究教学

4 . 边缘计算

用于移动端推理的前向框架。用于部署在手机、平板、IoT 等嵌入式设备上，通常适用于对算力要求较小的网络，只能用于前向推理，没有训练能力。一般来说由以下两者组成：

- converter : 转换器，用于将其他格式的 model 转换为该框架的格式
- interpreter : 解释器，用于在嵌入式设备上运行该框架的格式的 model

下图以 MNN 为例：



具体的边缘计算框架包括：

- **Tensorflow lite** : Google 的移动端计算框架
- **ncnn** : 腾讯的移动端计算框架
- **TNN** : 腾讯的移动端计算框架
- **MNN** : 阿里巴巴的移动端计算框架
- **Paddle-lite** : 百度的移动端计算框架
- **QNNPACK** : Facebook 的移动端计算框架
- **CoreML** : 苹果的移动端计算框架

5 · 快捷接口

快捷接口用户友好，增加开发调试速度，但灵活性差，类似于高级语言。快捷接口基于某种通用框架。

具体的快捷接口包括：

- **Keras** : Google 推出的 Tensorflow 的前端接口，Python。专注于用户友好，后端的实现目前已经支持 TensorFlow、CNTK、Theano。
- **Gluon** : Amazon 推出的 MXNet 的前端接口。
- **Fast.ai** : Facebook 推出的 Pytorch 的前端接口。

其他：

- **ONNX** : Open Neural Network Exchange，是一种针对机器学习所设计的开放式的文件格式，用于存储训练好的模型。它使得不同的人工智能框架（如 Pytorch、MXNet）可以采用相同格式存储模型数据并交互。ONNX 的规范及代码主要由微软，亚马逊，Facebook 和 IBM 等公司共同开发，以开放源代码的方式托管在 Github 上。目前官方支持加载 ONNX 模型并进行推理的深度学习框架有： Caffe2, PyTorch, MXNet, ML.NET, TensorRT 和 Microsoft CNTK，并且 TensorFlow 也非官方的支持 ONNX。

(二) 不同格式间的转换

模型在不同的通用框架以不同的格式存储。从存储的位置上来说，从存储的内容来看，分为模型和权重；包括内存中的模型和磁盘上的文件。文件格式有以下：

1 · 文件格式

● Keras

.h5/.hdf5 : keras 的缺省存储格式，可以用于存储权重或者模型+权重
.json : JSON 格式，被 keras 用来存储模型

● TensorFlow

.pb : protocol buffer 格式，tensorflow 用来存储模型和权重
.meta : 保存计算结构图（即网络模型）的文件
.ckpt : checkpoint 文件，在 v0.11 之前用于保存权重，之后被.index 和.data 文件替代
.index : it is a string-string immutable table(tensorflow::table::Table). Each key is a name of a tensor and its value is a serialized BundleEntryProto. Each BundleEntryProto describes the metadata of a tensor: which of the "data" files contains the content of a tensor, the offset into that file, checksum, some auxiliary data, etc.
.data-00000-of-00001 : 在 v0.11 之后，用于保存参数名和权重的文件

对于 tensorflow 的模型来说，可以使用：

- 1.meta+ckpt (v0.11 之前)
 - 2.meta+index+data (v0.11 之后)
 - 3.一个 pb 文件
- 这三种方式来描述一个带有权重的模型。

● TensorFlow Lite

.tflite : tensorflow lite 解释器识别的格式，不能直接训练，而是通过其它（主要是 tensorflow 和 h5）转换

● MXNet

.json : JSON 格式，用于存放 MXNet 的模型
.params : 用于存放 MXNet 框架的权重

● Caffe

.prototxt : 描述网络模型的文件
.caffemodel : 描述权重（及模型）的文件

● Caffe2

.predict-net.pb : 描述权重的文件?
.init-net.pb : 描述网络模型的文件?

● Cntk

.model : 模型和权重文件

● Pytorch

.pth : 模型和权重的文件

.pk1 : 模型和权重的文件，pickle 格式

● ncnn

.bin : 权重文件

.param : 模型文件

● TNN

.tnnmodel : 权重文件

.tnnproto : 模型文件

● darknet

.cfg : 模型配置、超参数及训练参数

.weights : 权重文件

● CoreML

.mlmodel : 模型和权重的文件

2 · 格式转换

to \ from	tensorflow	pytorch	onnx	keras	darknet	caffe2
tensorflow		mmconvert	onnx-tf (cmdline tool, in onnx_tf py pacakge)	keras_to_tensorflow(from github, https://github.com/amir-abdi/keras_to_tensorflow)		
pytorch	mmconvert (in package mmdnn)			mmconvert		
onnx	mmconvert	mmconvert		keras2onnx (python package)		convert-caffe2-to-onnx (in package onnx-caffe2, now merged into caffe2)
keras	mmconvert	pytorch2keras (python package) use onnx2keras in backend	onnx2keras(python package)		convert.py (from github repo https://github.com/qqwweee/keras-yolo3)	
caffe	mmconvert	mmconvert				

caffe2	<code>mmconvert</code>	<code>mmconvert</code>	<code>convert-onnx-to-caff2</code> (in package onnx-caff2, now merged into caffe2)			
tflite	<code>tflite_convert</code> (cmdline tool, in tensorflow py package)			<code>tflite_convert</code> (cmdline tool, in tensorflow py package)		
ncnn			<code>onnx2ncnn</code>			<code>caffe2ncnn</code>
coreML	<code>mmconvert</code> (in package mmdnn, depend on coremltools)	<code>mmconvert</code>				

works fine (at least 1 model converted and tested)
seems to be fine (model converted, but not tested)
no model converted successfully
not test yet

(1) Tensorflow to ...

- tensorflow→pytorch(model+weights)

```
mmconvert -sf tensorflow -iw xxx.pb --inNodeName inputNode --inputShape  
224,224,3 --dstNodeName outputNode -df pytorch -om xxx.pth
```

- tensorflow→caffe(model+weights)

```
mmconvert -sf tensorflow -iw xxx.pb --inNodeName inputNode --inputShape  
224,224,3 --dstNodeName outputNode -df caffe -om xxx
```

生成 xxx.prototxt 和 xxx.caffemodel

(2) Keras to ...

- keras→h5(model+weights)

```
model1.save( "xxx.h5" )
```

- keras→h5(weights)

```
model1.save_weights( "xxx.h5" )
```

- keras→json(model)

```
model1.to_json( "xxx.json" )
```

- keras → onnx

安装 keras2onnx :

```
pip install keras2onnx
```

通过 python 转换 :

```
import keras
import onnx
import keras2onnx

from keras.models import load_model
km = load_model("test.h5")
om = keras2onnx.convert_keras(km, km.name)
onnx.save_model(om, "./test.onnx")
```

- keras→tflite(model+weights):

python 脚本 tflite_convert

```
tflite_convert --keras_model_file=xxx.h5 --output_file=xxx.tflite
```

- h5→keras(weights)

```
model1.load_weights("xxx.h5")
```

- h5→pb(model+weights)

https://github.com/amir-abdi/keras_to_tensorflow

```
keras_to_tensorflow.py --input_model=xxx.h5 --output_model=xxx.pb
```

(3) Darknet to ...

- darknet→h5(model+weights)

<https://github.com/qqwweee/keras-yolo3>

```
convert.py yolo.cfg yolo.weights yolo.h5
```

(4) Onnx to...

- onnx → keras

安装 onnx2keras :

```
pip install onnx2keras
```

通过 python 转换 :

```
import keras
import onnx
```

```
import onnx2keras
om = onnx.load("test.onnx")
km = onnx2keras.onnx_to_keras(om, ["input_1"])
keras.models.save_model(km, 'test.h5', overwrite=True, include_optimizer=True)
```

- onnx → tensorflow

安装 onnx-tf :

```
pip install onnx-tf
```

通过 python 转换 :

```
onnx-tf convert -i a.onnx -o a.pb
```

- onnx → CoreML

参考以下 github 仓库 :

<https://github.com/onnx/onnx-coreml>

(5) Pytorch to ...

- pytorch → keras

安装 pytorch2keras :

```
pip install pytorch2keras
```

通过 python 转换 :

```
import keras
```

3 · MMdnn 转换

MMdnn 是微软开源的 DNN 模型管理工具 (MM=Model Management) , 帮助用户在不同深度学习框架之间进行互操作, 例如模型转换和可视化。可在 Caffe , Keras , MXNet , Tensorflow , CNTK , PyTorch Onnx 和 CoreML 之间转换模型。

mmldnn 的转换在实际实现中使用了不同的工具 (包括不少上文提到的其他工具) , 因此实际效果会有重合。此外目前看来 mmconvert 的效果并不是非常好。

<https://github.com/Microsoft/MMdnn>

可以通过 pip install mmdnn 安装, 主要的转换程序是 mmconvert , 格式为 :

```
mmconvert -sf SourceFramework -in InputNetwork -iw InputWeights -df DestinationFramework -om OutputModel --inputShape dim1, dim2, dim3
```

4 · ncnn 转换

腾讯的手机端前向计算框架 ncnn 自带三种转换工具：`mxnet2ncnn`，`caffe2ncnn` 和 `onnx2ncnn`。可以将对应框架的模型转换为 ncnn 模型，然后通过 NDK 开发，在手机上使用这些模型。

(三) 传统库

1 · wave

`wave` 是对 `wave` 音频文件进行操作的 Python 库。

- `wave`
 - `open()`：打开 `wave` 文件，返回 `Wave_read` 或者 `Wave_write` 对象
 - `Wave_read`：读方式打开的 `wave` 文件对象
 - ◆ `getparams()`：返回 `nchannels`（通道数，单声道、多声道），`sampwidth`（数据宽度，8bit 或者 16bit），`framerate`（44.1K、48K），`nframes`（总数据帧数），`comptype`（压缩方式），`compname`。
 - `Wave_write`：写方式打开的 `wave` 文件对象

2 · scipy

- `scipy`
 - `signal`：
 - ◆ `lfilter()`：用 FIR 或者 IIR 过滤器滤波
 - `b`：FIR 或者 IIR 滤波器的系数（即 `firwin` 函数的返回值）
 - `a`：IIR 滤波器的系数
 - `x`：输入的信号序列
 - ◆ `firwin()`：根据参数生成 FIR 滤波器的系数（coefficients）
 - `numtaps`：FIR 滤波器系数的长度
 - `cutoff`：截止频率
 - `fs`：信号的采样频率
 - ◆ `remez()`：用 Remez 算法生成滤波器

- ◆ freqz() :
- fft : 参见 numpy.fft
- fftpack
 - ◆ helper.freqtfreq() : 返回离散的频率分布，等于 numpy.fft.helper.freqtfreq()
 - n : 返回的频率数组的长度
 - d : 采样间隔（单位为秒）
- io
 - ◆ wavfile
 - read() : 读取 wave 文件，参数：路径，返回值：采样频率和数据
 - write() : 写入 wave 文件

3 · numpy

Python 的张量运算包，所有的神经网络，以及大量的科学计算包都基于此。

- ceil() : 向上圆整
- floor() : 向下圆整
- isnan() : 判断张量中各个元素是否为 None，返回同形状的 bool 类型
- maximum() : 参数为两个同形张量，对每个元素取最大值，得到一个新的同形张量
- array() : 用数组初始化一个 ndarray
- matlab.empty() : 建立一个未初始化的数组，参数为张量形状，比如(2,3)
- matlab.zeros() : 建立全零张量，参数为张量的形状，比如(2,3)
- matlab.ones() : 建立全一张量，参数为张量的形状，比如(2,3)
- matlab.identity() : 建立一个正方形的单位矩阵，参数为矩阵边长
- matlab.eye() : 建立一个对角线为 1 其余元素为 0 的矩阵，参数为矩阵边长，与 identity 不同在于可为长宽可以不等
- core.function_base.linspace() : 返回一个等差数列
 - start : 数列的最小值
 - stop : 数列的最大值
 - num : 数列元素的数量，缺省为 50
- core.numeric.zeros_like() : 建立形似的全零张量
 - a : 建立的张量形似 a
 - dtype : 元素数据类型
- core.numeric.full() : 建立一个每个元素值都为 fill_value 的张量
 - shape : 张量的形状
 - fill_value : 每个元素的值
- lib.function_base.delete() : 删除元素
- lib.function_base.insert() : 在 ndarray 中插入值
- lib.function_base.average() : 计算张量的加权平均
- lib.shape_base.expand_dims() : 在原张量中插入一个新的维度

- a : 输入的张量
- axis : 新增维度位置，可以是整数（第几个维度），或 tuple（哪几个维度）
- `lib.shape_base.tile()` : 通过复制某些维度，来扩展张量的维度
 - A : 输入张量
 - reps : 各维度的重复次数，整数表示在最里一个维度重复，维度超出 A 的维度则表示扩展其维度
- `core.fromnumeric.reshape()` : 修改张量的形状（不改变其值）
- `core.fromnumeric.nonzero()` : 返回张量中所有的非零元素
 - 返回 : 一个 tuple，每个成员都是 numpy 的向量，为各元素在各个维度的坐标，即如果输入张量为 N 维，则输出的 tuple 有 N 个成员
- `core.fromnumeric.cumsum()` : 累加函数
- `core.multiarray.where()` : 返回张量中符合条件的元素
 - condition : 条件
 - 返回 : 一个 tuple，每个成员都是 numpy 的向量，为各元素在各个维度的坐标，即如果输入张量为 N 维，则输出的 tuple 有 N 个成员
- `core.multiarray.concatenate()` : 拼接若干张量
 - (a1, a2, ...) : 被拼接的张量，除了在拼接的维度外，这些张量必须形状相同
 - axis : 拼接的维度，缺省为 0，即在最外的维度拼接
- `fft` : 快速傅立叶变换相关包
 - `fftpack.fft()` : 快速傅立叶变换，返回频域的序列，长度与输入相同（即每个元素代表频率/总长度的
 - ◆ a : 输入的时域数据序列，其长度等于频率×时间
 - `fftpack.ifft()` :
 - `helper.freq()` :
- `random` :
 - `random()` : 返回随机张量
 - ◆ size : int 或者 tuple 类型，随机张量的大小，缺省为 1（返回单个随机数）
 - `normal()` : 返回符合正态分布的随机张量
 - ◆ loc : 位置参数，即正态分布的期望值
 - ◆ scale : 尺度参数，即正态分布的标准差
 - ◆ size : int 或者 tuple 类型，返回的随机张量大小

4 · pandas

pandas 是 Python 中常用的数据分析库，包含了以 `Series` 和 `DataFrame` 为主的数据结构。

User Guide :

https://pandas.pydata.org/docs/user_guide/index.html

API Reference :

<https://pandas.pydata.org/docs/reference/index.html>

参考：

<https://www.jianshu.com/p/8024ceef4fe2>

参考：

https://blog.csdn.net/weixin_38168620?t=1

以下为各种 pandas 数据类型

(1) Series

`Series` 是 Pandas 最基本的数据格式，类似一维数组。`Series` 的数据值（`values`）带有一对应的索引（`index`），`values` 和 `index` 都有 `name`。

因为有 `index` 和 `name`，因此一个长度为 L 的 `Series` 中其实包含了 $2L+2$ 个数据： L 个 `values`， L 个 `index`，以及 `index` 和 `values` 的 `name`。

- `core.indexes.series.Series` :

- `Series` 可以创建自己的索引（`index`），这点上更象是 `dict`，而与 `list` 和 `ndarray` 不同
- 既可以通过 `index` 访问，也可以通过位置下标访问，这点与 `dict` 不同
- `Series` 和 `ndarray` 一维数组一样，其数据只能是相同数据类型，而 `list` 中的数据可以是不同的类型
- `Series` 的 `index` 和 值（`values`）都有名称（`name`），而 `dict`、`ndarray` 和 `list` 都没有
- 支持 `numpy` 的张量运算

参考：

https://blog.csdn.net/weixin_38168620/article/details/79572544

(2) DataFrame

是 Pandas 中最常用的基本数据结构，表示二维表结构。`DataFrame` 除了数据值本身（`values`）之外，还包括了行索引（`index`）和列索引（`columns`），以及这两者的 `name`。但 `values` 本身没有 `name`。

因此一个 shape 为(y,x)的 `DataFrames`，实际包含了 $xy+x+y+2$ 个数据： $x*y$ 个 `values`， y 个 `index`， x 个 `columns`，以及 `index` 和 `column` 的 `name`。

- `DataFrame` 的特点包括（以下 `df` 为其实例）：

- 带有行索引（`index`）和列索引（`columns`）
- `index` 和 `columns` 都可以有 `name`
- 可变大小，且表中数据的类型可以不一致
- `df.values` 返回一个二维的 `ndarray`
- `df.XXX` 返回一个指定的列

- ◆ 返回值为 `Series` 格式
 - ◆ XXX 为某 `columns`，也是返回的 `Series` 的 `values` 的 name
 - ◆ 返回的 `Series` 的 index 的 name 同 `df.index.name`
- `df.loc(XXX)` 返回一个指定的行
 - ◆ 返回值为 `Series` 格式
 - ◆ XXX 为某 `index`，也是返回的 `Series` 的 `values` 的 name
 - ◆ 返回的 `Series` 的 index 的 name 同 `df.columns.name`
- `core.frame.DataFrame` 的成员变量及成员函数包括：
 - `index`：行的标签
 - `columns`：列的标签
 - `axes`：行和列的标签
 - `dtypes`：各列数据的类型
 - `head()`：返回数据的前若干行（缺省为 5）
 - `tail()`：返回数据的最后若干行（缺省为 5）
 - `to_numpy()`：转换成 numpy 格式的二维张量，行和列的标签会丢失
 - `describe()`：返回各列数据的各项统计值（均值，标准差，最大，最小等）
 - `loc`：通过行-列标签定位数据，如 `data.loc[“number”, [“name”, “sex”]]`
 - `iloc`：通过索引定位数据，如 `data.iloc[3,2:4]`
 - `shape`：数据的形状，因为是二维数据，形为(4,5)

参考：

https://blog.csdn.net/weixin_38168620/article/details/79572785

(3) Index

`Index` 类描述了 `Series` 和 `DataFrame` 的索引，可以被视为一个一维数组。

- `core.indexes.base.Index`：
 - `series.index` 返回值为 `Index` 类型
 - `dataframe.index` 和 `dataframe.columns` 返回值为 `Index` 类型
 - `Index` 对象创建后不可修改

(4) MultiIndex

`MultiIndex` 描述了一个多级标签。

参考：

https://blog.csdn.net/weixin_38168620/article/details/79580272

(5) Timestamp/Period/Timedelta

`Timestamp` 是继承自标准库 `datetime` 的类，表示了一个精确到秒的**时间戳**。

`Period` 表示一个标准的**时间段**。例如某年、某月、某日、某小时等。时间的长短由 `freq` 决定。

`Timedelta` 表示一个**时间间隔**。

参考：

https://blog.csdn.net/weixin_38168620/article/details/79596526

(6) DatetimeIndex/PeriodIndex/TimedeltaIndex

`Timestamp`、`Period` 和 `Timedelta` 对象都是单个值，这些值都可以放在索引或数据中。作为索引的时间序列有：`DatetimeIndex`、`PeriodIndex` 和 `TimedeltaIndex`，它们都可以作为 `Series` 和 `DataFrame` 的索引。

参考：

https://blog.csdn.net/weixin_38168620/article/details/79596564

5 · matplotlib

`matplotlib` 是 Python 的绘图库。它可与 `NumPy` 一起使用，提供了一种有效的 MatLab 开源替代方案。`matplotlib` 的语法类似 `matlab` 面向过程。

`matplotlib.pyplot` 是最常用的模块，通常通过如下语句引用：

```
import matplotlib.pyplot as plt
```

官网：

<https://matplotlib.org/>

常用的 `pyplot` 模块函数包括：

- `plt.subplots()`：创建一个图表及其子图表
 - 返回 `figure.Figure`：创建的图表
 - 返回 `axes.Axes` 或者 `np.ndarray`：子图（只有一个）或者子图的阵列（若干个）
- `plt.subplot()`：将子图表激活，参数为 2,3,5, 或者 235，表示 2 行 3 列，被激活的子图为第 5 个（第 2 行第 2 列），之后所有操作均是对该子图表操作
- `plt.title()`：设置图表的名称

- plt.xlabel()：设置 x 轴的名称
 - plt.ylabel()：设置 y 轴的名称
 - plt.axis()：控制坐标轴的显示
 - plt.grid()：网格显示的控制
 - plt.legend()：图例（说明）的显示
-
- plt.bar()：绘制柱状图
 - plt.plot()：绘制图表
 - plt.imshow()：在图表中显示图片
-
- plt.axvline()：画一条竖线
-
- plt.show()：显示，在以上各种操作完成后调用

其数据结构有：

- figure.Figure：表示图表的类
- axes._axes.Axes：

四、NN 框架

本章接续上一章，按照来源（公司/组织）+具体框架的组织方式，介绍了各个神经网络的框架/库。

(一) Google

Google 提供的各种深度学习框架

1 · Tensorflow (基础框架)

TensorFlow 是 Google 开源的神经网络框架，最流行的两个神经网络通用框架之一（另一个是 FB 的 Pytorch），相对于 Pytorch，TF 的原语更偏底层计算图。

中文教程：

<http://c.biancheng.net/tensorflow/>

Tensorflow 模型部署：

<https://blog.csdn.net/chongtong/article/details/90379347>

2 · Keras (高级接口)

Keras 是 Google 开源的神经网络前端接口。基于 Tensorflow，Keras 的出现是为了让用户更快速的开发并试用模型。其特点包括：

- 相比 Tensorflow，专注于用户友好
- 后端的多种实现，目前已经支持 TensorFlow、CNTK、Theano
- 包括了一个基本的图像处理库，模型在 keras.applications 中。

keras 包现已合并到 tensorflow 中，为 tensorflow.keras

- **keras.layers**：包含了各种层类型
 - **core**：各种核心层
 - ◆ `Dense`：全连接层
 - ◆ `Activation`：激活函数层，可单独加入 model，也可作为参数加入其他层
 - ◆ `Dropout`：在进行学习的时候使得某些神经元失效，以减少过拟合
 - ◆ `Flatten`：展平输入为向量（不影响 batch），`reshape` 的特例
 - ◆ `Input`：输入层，初始化 Keras 张量
 - ◆ `Reshape`：对张量进行尺寸 / 维度变换，比如将 `(None, 2, 3, 4)` 变为 `(None, 8, 3)`，`None` 为 batch
 - ◆ `Permute`：维度置换（比如将 2 维卷积的第 1 维和第 2 维对调），例如将 `[[1, 2, 3], [4, 5, 6]]` 变为 `[[1, 4], [2, 5], [3, 6]]`
 - ◆ `RepeatVector`：将输入的向量（2 维张量，包括 batch）重复 N 次，从 `(None, 32)` 变为 `(None, N, 32)`
 - ◆ `Lambda`：将指定函数包装为层
 - ◆ `ActivityRegularization`：
 - ◆ `Masking`：
 - ◆ `SpatialDropout1/2/3D`：类似 Dropout，但会丢弃整个（1 维 / 2 维 / 3 维）特征图，而非单个神经元。比如 `SpatialDropout2D`，输入为 `(None, X, Y, C)`，`dropout` 时会将某个通道的特征图 `(X, Y)` 全部丢弃。
 - **convolutional**：各种卷积相关层
 - ◆ `Conv1D/Conv2D/Conv3D`：一维二维三维的常规卷积层
 - ◆ `SeparableConv1/2D`：深度分离卷积，包括一个逐层卷积层（Depthwise）和一个逐点卷积层（Pointwise）
 - ◆ `DepthwiseConv2D`：单独的逐层卷积层
 - ◆ `Conv2/3DTranspose`：反卷积
 - ◆ `Cropping1/2/3D`：1 维 2 维 3 维的裁剪
 - ◆ `UpSampling1/2/3D`：1/2/3 维的上采样，对每个输入元素重复若干遍
 - ◆ `ZeroPadding1/2/3D`：值为 0 的 Padding
 - **pooling**：各种池化层
 - ◆ `MaxPooling1/2/3D`：Max 池化层
 - ◆ `AveragePooling1/2/3D`：Average 池化层
 - ◆ `GlobalMaxPooling1/2/3D`：全局 Max 池化层
 - ◆ `GlobalAveragePooling1/2/3D`：全局 Average 池化层

- **local** : 本地连接层，即不共享权重的卷积层
 - ◆ `LocallyConnected1/2D` : 1 维/2 维的本地卷积层
- **recurrent** : 循环层
 - ◆ `RNN` :
 - ◆ `SimpleRNN` :
 - ◆ `SimpleRNNCell` :
 - ◆ `GRU` :
 - ◆ `GRUCell` :
 - ◆ `LSTM` :
 - ◆ `LSTMCell` :
 - ◆ `ConvLSTM2D` :
 - ◆ `ConvLSTM2DCell` :
 - ◆ `StackedRNNCell` :
- **embeddings** : 嵌入层，只有一个 Embedding
 - ◆ `Embedding` :
- **merge** : 融合层，融合多个输入
 - ◆ `Add` : 将输入的张量逐元素相加 (所有输入张量必须有相同尺寸)
 - ◆ `Subtract` : 将输入的两个张量相减
 - ◆ `Multiply` : 将输入的张量逐元素相乘 (所有输入张量必须有相同尺寸)
 - ◆ `Average` : 将输入的张量逐元素取均值 (所有输入张量必须有相同尺寸)
 - ◆ `Maximum` : 将输入的张量逐元素取最大值 (输入张量必须有相同尺寸)
 - ◆ `Minimum` : 将输入的张量逐元素取最小值 (所有输入张量必须有相同尺寸)
 - ◆ `Concatenate` : 连接输入的张量 (输入张量除了连接轴之外，尺寸相同)
 - ◆ `Dot` : 计算输入张量的点积
- **advanced_activations** : 高级激活层，包括
 - ◆ `LeakyReLU` :
 - ◆ `PReLU` :
 - ◆ `ELU` :
 - ◆ `ThresholdedReLU` :
 - ◆ `Softmax` :
 - ◆ `ReLU` :
- **normalization** : 正规化层，只有一个 BN
 - ◆ `BatchNormalization` :
- **noise** : 噪声层，用于缓解过拟合，包括
 - ◆ `GaussianDropout` :
 - ◆ `GaussianNoise` :
 - ◆ `AlphaDropout` :
- **wrappers** : 层封装器
 - ◆ `Bidirectional` : 双向封装，比如双向 RNN，双向 LSTM 等
 - ◆ `TimeDistributed` :
- **convolutional_recurrent** :
 - ◆ `ConvLSTM2D` :
 - ◆ `ConvLSTM2DCell` :
 - ◆ `ConvRNN2D` :

- **cudnn_recurrent :**
 - ◆ CuDNNGRU :
 - ◆ CuDNNLSTM :
- **keras.backend** : 各种后端实现的支持 (Theano, CNTK, TensorFlow) , common 为三家共同支持的接口
- **keras.preprocessing** : 预处理, 包括:
 - **sequence** : 时序数据的预处理工具包, 包括 TimeseriesGenerator
 - **text** : 文本的预处理工具包
 - **image** : 图像的预处理工具包
 - ◆ **DirectoryIterator** : 遍历指定目录的迭代器
 - ◆ **ImageDataGenerator** : 图像数据产生器 (用于数据增广)
 - apply_transform() :
 - fit() :
 - flow() :
 - flow_from_dataframe() :
 - flow_from_directory() : 从指定目录产生增广数据, 指定目录中每个子目录被视为一个类别
 - get_random_transform() :
 - random_transform() :
 - standardize() :
 - ◆ **Iterator** :
 - ◆ **NumpyArrayIterator** :
 - ◆ array_to_img() : 将 3D numpy array 转为 PIL 图像
 - ◆ img_to_array() : 将 PIL 图像转为 3D numpy array
 - ◆ save_img() : 以 ndarray 形式保存图像到文件
- **keras.losses** : 内置的损失函数, 包括 MSE, MAE, KLD, MAPE, MSLE 等
- **keras.metrics** : 内置的评价函数, 评价函数不用于训练, 用于评估当前模型的性能。
- **keras.optimizers** : 内置的优化器, 包括 SGD, RMSprop, Adagrad, Adadelta , Adam, Adamax, Nadam
- **keras.activations** : 内置的激活函数, 包括 softmax, relu, tanh, sigmoid 等。
 - softmax() :
 - elu() :
 - selu() :
 - softplus() :
 - softsign() :
 - relu() :
 - tanh() :
 - sigmoid() :
 - exponential() :
 - linear() :
- **keras.callbacks** : 内置的回调函数, 回调函数在训练期间的特定时间点被调用。
 - BaseLogger() :
 - TerminateOnNaN() :
 - ProgbarLogger() :

- History() :
- ModelCheckpoint() : 在每个 epoch 之后保存模型
- EarlyStopping() : 当被监测的数据不再提升，则停止训练
- RemoteMonitor() :
- LearningRateScheduler() :
- TensorBoard() : 写 Tensor board 日志到指定目录
- ReduceLROnPlateau() : 在学习进入平台期之后降低学习率
- CSVLogger() :
- LambdaCallback() :
- **keras.datasets** : 内置的数据集
 - **cifar**:CIFAR 格式的文件的解析工具
 - ◆ load_batch():解析 CIFAR 格式的数据文件，返回格式为(data, labels)
 - **cifar10** : CIFAR10 数据集
 - ◆ load_data():返回格式为(x_train,y_train), (x_test,y_test)
 - **cifar100** : CIFAR100 数据集
 - ◆ load_data():返回格式为(x_train,y_train), (x_test,y_test)
 - **imdb** : IMDB 电影评论情感分类数据集
 - ◆ get_words_index():
 - ◆ load_data():返回格式为(x_train,y_train), (x_test,y_test)
 - **reuters** : 路透社新闻主题分类
 - ◆ get_words_index():
 - ◆ load_data():返回格式为(x_train,y_train), (x_test,y_test)
 - **mnist** : MNIST 数据集
 - ◆ load_data():返回格式为(x_train,y_train), (x_test,y_test)
 - **fashion_mnist** : 时尚物品数据集
 - ◆ load_data():返回格式为(x_train,y_train), (x_test,y_test)
 - **boston_housing** : 波士顿房价回归数据集
 - ◆ load_data():返回格式为(x_train,y_train), (x_test,y_test)
- **keras.applications** : 带有预训练权重的网络模型，可用来预测、特征提取和微调
 - **densenet** :
 - ◆ DenseNet121() : 返回一个 DenseNet121 模型，可带预训练权重
 - ◆ DenseNet169() : 返回一个 DenseNet169 模型，可带预训练权重
 - ◆ DenseNet201() : 返回一个 DenseNet201 模型，可带预训练权重
 - **inception_resnet_v2** :
 - ◆ InceptionResNetV2() : 返回一个 InceptionResNetV2 模型，可带预训练权重
 - **inception_v3** :
 - ◆ InceptionV3() : 返回一个 InceptionV3 模型，可带预训练权重
 - **mobilenet** :
 - ◆ MobileNet() : 返回一个 MobileNet 模型，可带预训练权重
 - **mobilenet_v2** :
 - ◆ MobileNetV2() : 返回一个 MobileNetV2 的模型，可带预训练权重
 - **nasnet**:
 - ◆ NASNetLarge() : 返回一个 NASNetLarge 的模型，可带预训练权重
 - ◆ NASNetMobile() : 返回一个 NASNetMobile 的模型，可带预训练权重

- **resnet50:**
 - ◆ ResNet50() : 返回一个 ResNet50 模型，可带预训练权重
- **vgg16 :**
 - ◆ VGG16() : 返回一个 VGG16 模型，可带预训练权重
- **vgg19 :**
 - ◆ VGG19() : 返回一个 VGG19 模型，可带预训练权重
- **xception :**
 - ◆ Xception() : 返回一个 Xception 模型，可带预训练权重
- **keras.initializers** : 内置的初始化器
- **keras.regularizers** : 内置的正则化器
- **keras.constraints** : 内置的约束项，包括 MaxNorm, MinMaxNorm, NonNeg, UnitNorm
- **keras.models** : 模型相关的
 - **Model :**
 - ◆ compile() : 编译模型
 - ◆ fit() : 训练模型
 - ◆ evaluate() : 评估模型的准确性
 - ◆ predict() : 使用模型预测
 - ◆ train_on_batch() :
 - ◆ test_on_batch() :
 - ◆ predict_on_batch() :
 - ◆ fit_generator() : 基于数据增广器训练模型
 - ◆ evaluate_generator() : 基于数据增广器进行评估
 - ◆ predict_generator() :
 - ◆ get_layer() :
 - **Sequential :**

(二) Facebook

Facebook 提供的各种深度学习框架，主要是以

1 · Pytorch (基础框架)

官方 API 文档：

<https://pytorch.org/docs/stable/torch.html>

中文版教程：

<https://www.pytorch123.com/>

(1) torch

torch 包本身包括了一些基本的张量操作，功能和 numpy 类似，可以支持 GPU。

- `torch`
 - `empty()`: 生成一个空矩阵
 - `rand()`: 生成一个随机矩阵
 - `zeros()`: 生成一个全零矩阵
 - `tensor()`: 生成一个矩阵，元素由 (list 格式的) 参数给定
 - `is_tensor()`: 如果参数是一个 pytorch 张量，返回 True
 - `is_storage()`: 如果参数是一个 pytorch storage，返回 True
 - `numel()`: 返回张量中的元素个数
 - `eye()`: 创建一个单位矩阵
 - `from_numpy()`: 将 numpy 的 ndarray 转换为 pytorch 张量
 - `linspace()`: 返回一个 1 维张量，包含在区间 start 和 end 上均匀间隔的 steps 个点。输出 1 维张量的长度为 steps。
 - `ones()`: 返回指定形状的全 1 张量
 - `serialization.save()`: 保存一个对象到文件中
 - `serialization.load()`: 从文件中读取一个 save() 保存的对象
 - `tensor.Tensor`: 表示 pytorch 张量的类
 - `jit`: TorchScript 格式相关，在非 Python 环境下运行（这个格式不能处理条件控制，比如 if。也不能处理第三方的函数，只能处理 pytorch 中的部分操作）
 - ◆ `ScriptModule`: TorchScript 格式的模块，类似 `torch.nn.Module`
 - ◆ `ScriptFunction`: TorchScript 格式的函数
 - ◆ `trace()`: 将一个函数转化为 ScriptFunction 格式并返回。
 - ◆ `trace_module()`: 将 `torch.nn.Module` 转化为 ScriptModule 格式并返回。
 - `quantization`:
 - ◆ `quantize.convert()`: 将一个浮点模型量化

(2) torch.nn

nn 是 torch 最大的子 pacakge，其中最主要是 modules 子 package，里面包含了各种网络模型相关的类，包括模型、各种卷积层、池化层、RNN 层、激活函数、dropout、损失函数。

- `torch.nn`: NN 包
 - `parameter.Parameters`:
 - `modules.container.Sequential`: 顺序模型，Module 的子类
 - `modules.module.Module`: 描述网络模型的类
 - ◆ `add_module()`: 添加子模块到当前模型
 - ◆ `children()`: 返回当前模型子模块的迭代器

- ◆ `cpu()`: 将所有参数和 buffers 复制到 CPU
- ◆ `cuda()`: 将所有参数和 buffers 复制到 GPU
- ◆ `eval()`: 模型转换为 evaluation (预测) 模式 (影响 Dropout 层和 BN 层)
- ◆ `double()`: 将参数和 buffers 的类型转为 double
- ◆ `float()`: 将参数和 buffers 的类型转为 float
- ◆ `half()`: 将参数和 buffers 的类型转为 half
- ◆ `modules()`: 返回当前模型所有模块 (各子孙模块) 的迭代器
- ◆ `named_children()`: 同 `children()`，但同时 yield 子模块名字
- ◆ `named_modules()`: 同 `modules()`，但同时 yield 子孙模块名字
- ◆ `parameters()`: 返回包含所有参数的迭代器
- ◆ `register_backward_hook()`
- ◆ `register_buffer()`
- ◆ `register_forward_hook()`
- ◆ `register_parameter()`
- ◆ `state_dict()`: 返回一个字典，包含模型的所有状态
- ◆ `load_state_dict()`:
- ◆ `train()`: 模型设置为 training 模式 (影响 Dropout 层和 BN 层)
- ◆ `zero_grad()`: 将模型所有参数梯度设置为 0
- `modules.conv.Conv(1|2|3)d`: 描述卷积层的模块
- `modules.conv.ConvTranspose(1|2|3)d`:
- `modules.pooling.AdaptiveAvgPool(1|2|3)d`: 自适应 (输出尺寸固定) 平均池化
- `modules.pooling.AdaptiveMaxPool(1|2|3)d`: 自适应 (输出尺寸固定) 最大池化
- `modules.pooling.AvgPool(1|2|3)d`: 平均池化
- `modules.pooling.FractionalMaxPool(2|3)d`:
- `modules.pooling.LPPool(1|2)d`:
- `modules.pooling.MaxPool(1|2|3)d`: 最大池化
- `modules.pooling.MaxUnpool(1|2|3)d`: 逆最大池化 (丢失的部分设置为 0)
- `modules.activation.ReLU, RReLU, ReLU6, ELU, CELU, SELU, GLU, LeakyReLU, PReLU`
- `modules.activation.Hardtanh, Tanh`
- `modules.batchnorm.BatchNorm(1|2|3)d`:
- `modules.batchnorm.SyncBatchNorm`:
- `modules.rnn`: 各种 RNN 单元 (微结构)，包括 `RNNBase`
- `modules.rnn.RNN`:
- `modules.rnn.LSTM`:
- `modules.rnn.GRU`:
- `modules.rnn.RNNCellBase`:
- `modules.rnn.RNNCell`:
- `modules.rnn.LSTMCell`:
- `modules.rnn.GRUCell`:
- `modules.dropout.Dropout(|2d|3d), AlphaDropout, FeatureAlphaDropout`:
- `modules.sparse.Embedding.EmbeddingBag`:
- `modules.loss.MSELoss`:

(3) torch.autograd

(4) torch.optim

Pytorch 中的各种优化方法

- `torch.optim`：
 - `adadelta.Adadelta` :
 - `adagrad.Adagrad` :
 - `adam.Adam` :
 - `adamw.AdamW` :
 - `sparse_adam.SparseAdam` :
 - `adamax.Adamax` :
 - `asgd.ASGD` :
 - `sgd.SGD` :
 - `rprop.Rprop` :
 - `rmsprop.RMSprop` :
 - `optimizer.Optimizer` :
 - `lbfgs.LBFGS` :

(5) torch.utils

相关工具

- `torch.utils.data`：
 - `dataset.Dataset` : 数据集的抽象基类，所有具体的数据集都是其子类，子类需实现`__getitem__()`函数
 - `dataset.IterableDataset` :
 - `dataset.TensorDataset` :
 - `dataset.ConcatDataset` :
 - `dataset.ChainDataset` :
 - `dataset.Subset` :
 - `dataloader.DataLoader` :
 - `sampler.Sampler` :
 - `sampler.SequentialSampler` :
 - `sampler.RandomSampler` :
 - `sampler.SubsetRandomSampler` :
 - `sampler.WeightedRandomSampler` :

- `sampler.BatchSampler` :
- `distributed.DistributedSampler` :
- `torch.utils.data` :

2 · torchvision (图像处理)

Pytorch 的机器视觉相关的框架，包括图像分类、语义分割、及少量目标检测、实例分割、人体关键点检测、行为识别（视频）。

官方文档：

<https://pytorch.org/docs/stable/torchvision/>

- `torchvision` : 包含了目前流行的数据集、网络模型和图片处理工具
 - `datasets`: 数据集
 - ◆ `cifar`
 - `CIFAR10`: 表示 CIFAR10 数据集的类
 - `CIFAR100`: 表示 CIFAR100 数据集的类
 - ◆ `mnist.MNIST`:
 - ◆ `coco.CocoCaptions`:
 - ◆ `coco.CocoDetections`:
 - ◆ ...
 - `models`: 模型卷积网络
 - ◆ `alexnet`:
 - `AlexNet`: 表示 Alexnet 的类
 - `alexnet()`: 返回 Alexnet 对象，pretrained 参数决定是否加载权重
 - ◆ `densenet`: densenet 模块
 - `DenseNet`: Densenet 的类
 - `densenet121()`: 返回 densenet121 结构的 DenseNet 对象
 - `densenet161()`
 - `densenet169()`
 - `densenet201()`
 - ◆ `inception`: inception 模块
 - `Inception3` : Inception V3 的类
 - `inception_v3()`: 返回 Inception3 的对象
 - ...
 - ◆ `resnet`:
 - `BasicBlock`: residual block 的类
 - `Bottleneck`: bottleneck 结构的类
 - `ResNet`: resnet 的类
 - `resnet50()`: 返回 resnet50 结构的 ResNet 对象
 - `resnet101()`: 返回 resnet101 结构的 ResNet 对象
 - `resnet152()`: 返回 resnet152 结构的 ResNet 对象
 - ◆ `mobilenet`: mobilenet 模块
 - `MobileNetV2` : mobilenet v2 的类

- `mobilenet_v2()`: 返回一个 MobileNetV2 的对象
- `ConvBNReLU`: 卷积+BN+ReLU 组成的模块
- `InvertedResidual`: Inverted Residual Block
- `transforms`: 对 PIL.Image 进行变换的模块
 - ◆ `transforms.Compose`:
 - ◆ `transforms.CenterCrop`: 对图片进行中心切割（中心位置同原图）
 - ◆ `transforms.RandomCrop`: 对图片进行随机切割（中心位置随机）
 - ◆ `transforms.RandomHorizontalFlip`: 对图片随机水平翻转，有一半几率水平翻转
 - ◆ `transforms.RandomSizeCrop`: 对图片进行随机大小的切割，然后 resize 到指定大小
 - ◆ `transforms.Pad`: 填充
 - ◆ `transforms.ToTensor`: 转换图片为 pytorch 张量
 - ◆ `transforms.ToPILImage`: 转换 pytorch 张量或者 numpy.ndarray 为 PIL 图片
 - ◆ `transforms.Lambda`: 使用指定转换器
- `utils`:
 - ◆ `make_grid()`:
 - ◆ `save_image()`: 将指定 pytorch 张量存为图片

3 · Detectron (已废弃)

原 Caffe2 的图像处理框架，Caffe2 合并入 Pytorch 后被废弃，由 Detectron2 继承。

4 · Maskrcnn-benchmark (已废弃)

一个 Facebook 的图像处理框架，已废弃，Detectron2 继承

5 · Detectron2 (计算机视觉)

Detectron2 也是 Facebook 推出的计算机视觉库，包括目标检测、语义分割、全景分割等

Detectron 基于 Caffe2，而 maskrcnn-benchmark 基于 Pytorch，Caffe2 和 Pytorch 合并之后，Facebook 在这两者基础上推出了 Detectron2。

github：
<https://github.com/facebookresearch/detectron2>

预训练模型介绍：
https://github.com/facebookresearch/detectron2/blob/master/MODEL_ZOO.md

参考：

<https://www.aiuui.cn/aifarm1288.html>

- `config`
 - `config`：网络模型的配置文件
 - ◆ `CfgNode`：描述网络模型的类，包括结构、权重、数据集等等
 - `merge_from_file()`：从 config 文件中读取网络模型的设置
 - `dump()`：
 - ◆ `get_cfg()`：返回当前的网络模型 cfg
 - ◆ `set_global_cfg()`：将当前的网络模型 cfg，设置为参数给定的网络模型 cfg
 - `model_zoo`：包含各个网络模型的 config 文件（config 目录），以及三个函数（model_zoo 模块中，参数均为 config 文件的相对路径）
 - `model_zoo.get()`：根据 config 文件的相对路径，得到对应的 model 对象
 - `model_zoo.get_config_file()`：将 config 文件的相对路径转换为绝对路径（相对于 detectron2/model_zoo/config）
 - `model_zoo.get_checkpoint_url()`：根据 config 文件名（相对路径），获得网络模型对应的预训练权重的 URL
 - `config`：保存各种网络模型的 config 文件（格式为 yaml）
 - `modeling`：
 - `meta_arch.build.build_model()`：根据 cfg 生成返回一个 torch.nn.Module
 - `meta_arch.panoptic_fpn.PanopticFPN`：全景分割 FPN
 - `meta_arch.rcnn.GeneralizedRCNN`：通用化的 RCNN，包括特征提取+Regional Proposal+每个候选区域预测
 - `meta_arch.retinanet.RetinaNet`：RetinaNet
 - `backbone.backbone.Backbone`：描述基础网络的基类
 - `backbone.resnet.ResNet`：基础网络 ResNet
 - `backbone.resnet.ResNetBlockBase, BottleneckBlock, DeformBottleneckBlock`：ResNet 的各种 block
 - `backbone.fpn.FPN`：基础网络 FPN
 - `backbone.fpn.build_resnet_fpn_backbone()`：生成一个以 ResNet 为基础网络的基础网络 FPN
 - `engine`
 - `defaults.DefaultPredictor`：根据 CfgNode 生成一个 predictor
 - `defaults.DefaultTrainer`：根据 CfgNode 生成一个 trainer
 - ◆ `resume_or_load()`：从 checkpoint 恢复，或者根据 CfgNode 生成模型
 - ◆ `train()`：训练
 - `checkpoint`：保存模型的 checkpoint
 - `detection_checkpoint.Checkpointer`：用于保存模型的 checkpoint，参数为 `torch.nn.modules.module.Module` 类型
 - `detection_checkpoint.DetectionCheckpointer`：同 Checkpointer，区别在于可以接受 detectron 和 detectron2 格式的 model 作为参数。
 - `detection_checkpoint.PeriodicCheckpointer`：
 - `data`：数据集相关

- `catalog.DatasetCatalog` : 数据集的目录
 - ◆ `register()` : 注册新的数据集
 - ◆ `get()` :
 - ◆ `list()` : 列出所有的注册的数据集
- `catalog.MetadataCatalog` : 描述一个数据集的元数据，比如分成哪些类，每个类用什么颜色表示等
 - ◆ `get()` : 返回对应某个数据集的 metadata (singleton 实例)
 - ◆ `list()` : 列出所有注册了 metadata 的数据集
- `utils` : 各种工具
 - `logger` : log 相关
 - ◆ `setup_logger()` : 初始化 log
 - `visualizer`
 - ◆ `Visualizer` : 图像的具像化工具，用原始图像初始化
 - `draw_instance_predictions()` : 画出图像的 instance-level 的预测结果
 - `draw_sem_seg()` : 画出图像的语义分割的预测结果
 - `draw_panoptic_seg_predictions()` : 画出图像的全景分割 (panoptic segmentation) 的预测结果
 - `draw_dataset_dict()` : 画出图像 (目标检测、图像分割等) 的 Ground Truth

6 · PySlowFast (视频理解)

PySlowFast 是 Facebook 的视频理解的库，包括 Slow、SlowFast、I3D、C2D、Non-local network 等几个网络模型。

PySlowFast 类似 Darknet，无须编写 python 等代码。训练、测试、推理是通过命令行进行，配置是通过配置文件。

github (也是 SlowFast 模型的仓库)：
<https://github.com/facebookresearch/SlowFast>

github 仓库结构：

- `configs` : 配置文件 (yaml 格式)，包括：
 - 对 AVA 数据集的配置 : SLOW_8x8_R50_SHORT.yaml 等
 - 对 Charades 配置 :
 - 对 Kinetics 的配置 :
 - 对 SSv2 的配置
- `demo` : 演示相关，包括 yaml 配置文件、label 文件
- `projects` : multigrid 相关介绍
- `slowfast` : SlowFast 源码
- `tools` :
 - `run_net.py` : 运行某个网络

- `train_net.py` : 训练某个网络
- `test_net.py` : 测试某个网络
- `benchmark.py` : 衡量某个网络
- `demo_net.py`
- `visualization.py`
- `setup.py` : build 脚本
- 各种 `md` : 安装、数据集下载、训练等的介绍文档

PySlowFast 的运行、训练、测试需要：

- 程序文件：即 tools 目录下的 `run_net.py` 等
- 配置文件：yaml 格式，在 configs 目录下有（配置也可以在命令行被覆盖及指定）
- 权重文件：pk1 格式的 checkpoints，在配置文件中指定

命令行格式形如：

```
./tools/run_net.py --cfg configs/Kinetics/C2D_8x8_R50.yaml NUM_GPUS 1
```

7 · Prophet (时间序列分析)

Facebook 推出的一个时间序列分析算法。

官网：

<https://facebook.github.io/prophet/>

参考：

<https://zhuanlan.zhihu.com/p/52330017>

github：

<https://github.com/facebook/prophet>

(三) Amazon

Amazon 的 DL 框架是 DL 生态的三强之一（另外两家是 Google 和 Facebook），其底层库是 `MXNet`，上层接口是 `gluon`，此外包括 `gluoncv`（机器视觉）、`gluonts`（时间序列）和 `gluonnlp`（NLP）等专用框架。

1 · MXNet (基础框架)

`MXNet` 是亚马逊的深度学习框架，算是当前三大深度学习框架之一，但相较于主流的 `Pytorch` (Facebook) 和 `Tensorflow/Keras` (Google)，流行程度要差一些。

官方 API 文档：<https://mxnet.apache.org/api/python/docs/api/index.html>

- `mxnet.nd` (`mxnet.ndarray`)
 - `NDArray` : mxnet 的张量类
 - ◆ `attach_grad()` : 申请计算梯度所需要的内存
 - `array()` :
 - `ones()` : 生成所有元素值为 1 的张量，参数为 shape
 - `zeros()` : 生成所有元素值为 0 的张量，参数为 shape
 - `random` : 随机数相关
 - ◆ `normal()` : 生成正态分布的随机数
- `mxnet.autograd` : 自动求导相关
 - `record()` : 返回一个上下文，要求 MXNet 记录与梯度相关的计算
 - `backward()` : 计算梯度
- `mxnet.image` : 图像相关
- `mxnet.gluon` : MXNet 的高层接口，类似 Keras 之于 Tensorflow
 - `data` : 数据操作工具
 - ◆ `DataLoader` : 从数据集中加载数据并返回 mini-batch 的迭代器
 - `loss` : 损失函数
 - ◆ `SoftmaxCrossEntropyLoss` : Softmax 及交叉熵损失
 - `nn` : 各神经网络层
 - `Trainer` :

2 · Gluon (高级接口)

gluon 是 Amazon 推出的高级接口，基于 MXNet，gluon 包括几个包：

- `gluon` : 类似于 Keras，已合并到 `mxnet` 下
- `gluon.models`

3 · gluoncv (图像处理)

gluoncv 是 Amazon 的图像处理相关的包，包括图像分类、目标检测、图像分割任务。

官网：

<https://gluon-cv.mxnet.io/>

- `loss` : 损失函数
- `nn` :
 - `bbox` : bounding box 相关操作
 - `block` :
 - `coder` :

- `data` : 数据集相关
- `utils` : 工具集
- `model_zoo` :
 - `model_zoo` : Model Zoo
 - ◆ `get_model()` : 获取指定的网络模型实例
 - ◆ `get_model_list()` : 获取支持的网络模型列表
 - `alexnet` :
 - ◆ `AlexNet` : AlexNet 的类
 - ◆ `alexnet` : 返回 AlexNet 的实例
 - ... : 各种网络模型的实现

4 · gluonts (时间序列)

gluonts 是亚马逊推出的时间序列 (Time Series) 线下建模工具包。其中包括：

- 用于创建模型的组件
- 数据加载、训练和处理工具
- 一些已有数据集
- 一些已有模型
- 图表绘制和评估

Gluonts 的主要数据结构包括：

- `Dataset` : 数据集，gluonts 中的 dataset 用于提供访问数据的统一接口
- `Estimator` : 可训练的模型
- `Predictor` : 不可训练的模型
- `Trainer` : 训练器，主要描述了一系列训练的参数，并提供了训练的核心函数，被传递给 Estimator
- `Evaluator` : 提供了（对模型的）多尺度评价
- `Distribution` : 描述一个具体的概率分布的类
- `DistributionOutput` : 产出 `Distribution` 的类
- `Forecast` : 描述预测的类（以采样、分位数等方式）

论文：

Gluonts: Probabilistic Time Series Models in Python

<https://arxiv.org/pdf/1906.05264.pdf>

官方文档：

<https://gluon-ts.mxnet.io/>

Tutorial :

https://gluon-ts.mxnet.io/examples/basic_forecasting_tutorial/tutorial.html

An Overview of GluonTS :

<https://github.com/aws-samples/amazon-sagemaker-time-series-prediction-using-gluonts>

github：
<https://github.com/awslabs/gluon-ts.git>

(1) 基本

- `block`：网络中用到的各种单元结构
 - `cnn`：
 - ◆ `CausalConv1D`：一维的因果卷积（指的是 `output[i]` 不依赖 `input[i+1]`）
 - ◆ `DilatedCausalGated`：膨胀的因果卷积
 - ◆ `ResidualSequential`：
- `core`：

(2) dataset

`dataset` 为数据集相关的包。

① 数据结构

```
class TrainDatasets(NamedTuple):  
    metadata: MetaData  
    train: Dataset  
    test: Optional[Dataset] = None
```

- 一个完整的数据集类型为 `TrainDatasets`
 - 父类为命名元组 (`NamedTuple`)
 - 包含了训练集和测试集，以及元数据 (`MetaData` 类型)

```
class BasicFeatureInfo(pydantic.BaseModel):  
    name: str  
class CategoricalFeatureInfo(pydantic.BaseModel):  
    name: str  
    cardinality: str  
class MetaData(pydantic.BaseModel):  
    freq : str
```

```
target : Optional[BasicFeatureInfo] = None
feat_static_cat: List[CategoricalFeatureInfo] = []
feat_static_real: List[BasicFeatureInfo] = []
feat_dynamic_real: List[BasicFeatureInfo] = []
feat_dynamic_cat: List[CategoricalFeatureInfo] = []
prediction_length: Optional[int] = None
```

类 `MetaData` 描述了数据集的元信息

- `feat_static_cat`：可以用于对记录所属的组进行编码的分类特征数组。分类要素必须编码为基于 0 的正整数序列。例如，分类域{R, G, B}可以编码为{0, 1, 2}。来自每个分类域的所有值都必须在训练数据集中表示。
- `feat_static_real`：
- `feat_dynamic_real`：代表自定义要素时间序列（动态要素）向量的浮点值或整数数组。如果设置此字段，则所有记录必须具有相同数量的内部数组（相同数量的特征时间序列）。此外，每个内部数组必须具有与关联 target 值相同的长度。例如，如果目标时间序列代表不同产品的需求，则 `feat_dynamic_real` 可能是布尔时间序列，它指示是否对特定产品应用了促销。
- `feat_dynamic_cat`：

```
Dataset = Iterable[DataEntry]
class FileDataset(Dataset):
class ListDataset(Dataset):
```

一个训练集或者测试集的类型为 `Dataset`

- 实际格式为 `DataEntry` (即字典，key 为字符串，value 为任意类型数据) 的 `Iterable` (迭代容器)。
- 有两种 dataset：`FileDataset` 和 `ListDataset`，均为 `DataSet` 的子类
- 如果想自己实现一个训练集或者测试集，需要：
 - 生成 `gluonts.dataset.common.Dataset` 的子类
 - 子类实现 `__iter__()`，返回 `DataEntry` 的迭代(iterator)，每次迭代给出一个数据样本 `DataEntry`
 - 子类实现 `__len__()`
 - 字典中要有“start”和“target”字段

```
DataEntry = Dict[str, Any]
```

数据集中的单个时间序列 (Time Series) 样本的类型为 `DataEntry`，这是一个泛型别名类型，实际是一个字典 dict，key 为字符串，value 为任意类型。

字典 key 对某些 str 有缺省的定义，在 `gluonts.dataset.field_names.FieldName` 中：

- 必须有的字典 key：
 - “start”：时间序列的开始时间
 - “target”：时间序列的值
- 可选的 key：
 - `feat_static_cat`：
 - `feat_static_real`：

- feat_dynamic_cat :
- feat_dynamic_real :
- 通过各种变换 (Transformation) 获得的 key :
 - time_feat :
 - feat_dynamic_const :
 - feat_dynamic_age : 通常给 `AddAgeFeature` 变换存放输出结果
 - observed_values : 通常给 `AddObservedValueIndicator` 变换存放输出的结果
 - is_pad :
 - forecast_start :

`DataEntry["target"]`

- 每个时间序列分为若干时间步骤 (Time Step) , 每个时间步骤通常为一个标量或者向量。是 `DataEntry["target"]` 字段的一个元素。

`DataBatch = Dict[str, Any]`

描述一个 batch 的数据的类型为 `DataBatch` , 同 `DataEntry` 一样也是一个泛型别名类型 , 为一个 key 为字符串 , value 为任意类型的字典。

```
class DataLoader(Iterable[DataEntry]):
    dataset: Dataset #关联的数据集
    transform: Transformation
    batch_size: int
    ctx: mxnet.Context #用于保存数据的 MXNet Context
    dtype:
    num_workers: int #用于数据预处理的并行线程数
    num_prefetch: int #预取的 batch 数, 更多表示
    cyclic: bool #数据是否循环 (训练集为 True)
    is_train: bool #是否用于训练 (InferenceDataLoader 为 False)

class TrainDataLoader(DataLoader)
class ValidationDataLoader(DataLoader)
class InferenceDataLoader(DataLoader)
```

`DataLoader` 为数据加载器的基类 , 用于指定的数据集中迭代数据 , 利用其成员变量 `parallel_data_loader` 迭代出 `DataBatch` 。

- 其中有核心成员为 `ParallelDataLoader` 类
- 有三个子类
 - `TrainDataLoader` : 训练集数据加载器
 - `ValidationDataLoader` : 验证数据集加载器
 - `InferenceDataLoader` : 推理数据集加载器

②包结构

- `dataset` : 数据集相关
 - `common` : 数据集的类
 - ◆ `DataEntry` : `Dict[str, Any]` , 描述单个时间序列（即一个样本）的数据帧
 - ◆ `DataBatch` : `Dict[str, Any]` ,
 - ◆ `Dataset` : `Iterable[DataEntry]` , `DataEntry` 的迭代，单个数据集
 - ◆ `ListDataset` : 数据格式为 dict 的数据集，`Dataset` 的子类
 - `data_iter` : `DataEntry` 的迭代
 - `freq` : 每个时间步骤的长度
 - `one_dim_target` :
 - ◆ `FileDataset` : 从 JSON 文件加载的数据集，`Dataset` 的子类
 - `path` : 数据集文件路径
 - `freq` : 每个时间步骤的长度
 - `one_dim_target` :
 - `cache` :
 - ◆ `MetaData` : 数据集的元信息，包括 `prediction_length` 等
 - ◆ `TrainDatasets` : 描述一个完整的、包括训练集和测试集的数据集。为 `NamedTuple` 的子类，包括三个元素：
 - `metadata` : 元信息，`MetaData` 类型
 - `train` : 训练集，`Dataset` 类型
 - `test` : 测试集，`Dataset` 类型
 - ◆ `load_datasets()` : 加载数据集，返回 `TrainDatasets`
 - `field_names` : `DataEntry` 中字典 key 的缺省定义
 - ◆ `FieldName` : 定义了 `DataEntry` 中各个有特殊含义的字段的字段名
 - `ITEM_ID` : 字段名 “item_id”
 - `START` : 时间序列的开始时间，`DataEntry` 中的字段名为 “start”
 - `TARGET` : 时间序列的值，字段名为 “target”
 - `FEAT_STATIC_CAT` : 字段名 “feat_static_cat”
 - `FEAT_STATIC_REAL` : 字段名 “feat_static_real”
 - `FEAT_DYNAMIC_CAT` : 字段名 “feat_dynamic_cat”
 - `FEAT_DYNAMIC_REAL` : 字段名 “feat_dynamic_real”
 - `FEAT_TIME` : 字段名 “time_feat”
 - `FEAT_CONST` : 字段名 “feat_dynamic_cost”
 - `FEAT_AGE` : 字段名 “feat_dynamic_age”
 - `OBSERVED_VALUES` : 字段名 “obsevered_values”
 - `IS_PAD` : 字段名 “is_pad”
 - `FORECAST_START` : 字段名 “forecast_start”
 - `TARGET_DIM_INDICATOR` : 字段名 “tareget_dim_indicator”
 - `loader` : 数据加载器，用于返回 `DataBatch` 的迭代
 - ◆ `DataLoader` : 所有加载器的基类
 - `parallel_data_loader` : 核心成员，实现了数据的迭代
 - `__init__()` : 构造函数

- `__iter__()`: 迭代函数
 - ◆ `TrainDataLoader`: 继承 `DataLoader`, 加载训练数据集
 - ◆ `ValidationDataLoader`: 继承 `DataLoader`, 加载验证数据集
 - ◆ `InferenceDataLoader`: 继承 `DataLoader`, 加载实际的推理数据集
- `utils`: 数据集工具
 - ◆ `to_pandas()`: 将 `dict` 转换为 `pandas.Series` 格式
- `repository`: 已经封装好的数据集, 其中的 `datasets.get_dataset()` 函数常用
 - ◆ `*`: 各种已有数据集的生成函数 (来自网络)
 - ◆ `datasets`:
 - `datasets_recipes`: 预设部分参数的数据集生成函数的字典
 - `get_datasets()`: 根据关键字, 生成某数据集, 返回 `TrainDataset`
- `artificial`: 人工生成的时间序列数据集
 - ◆ `_base.ComplexSeasonalTimeSeries`:
 - `num_series`: 时间序列的样本数量
 - `prediction_length`: 预测的时间步骤
 - `freq_str`: 频率, 缺省为 “H” (小时)
 - `length_low`: 最短的时间序列长度 (必须高于 `prediction_length`)
 - `length_high`: 最长的时间序列长度
 - `min_val`: 时间序列的最小值
 - `max_val`: 时间序列的最大值
 - `is_integer`:
 - `proportion_missing_values`:
 - `is_noise`: 是否添加噪音
 - `is_scale`: 是否
 - `percentage_unique_timestamps`:
 - `is_out_of_bounds_date`:

(3) distribution

`mx.distribution` (老版本中的 `distribution`) 目录中包含了分布相关的内容。虽然代码行数不多, 但与概率论及统计学结合比较紧密, 且代码比较晦涩难懂。与此相关的一些概念可以参考《[概念定义 > NN 相关 > 分布 Distribution](#)》和《[概念定义 > 概率论](#)》。

`gluonts` 中最常用的分布是 `StudentT` (学生 t 分布), 参考《[概念定义 > 概率论 > 学生 t 分布](#)》。

混淆注意!

`Distribution` 和 `DistributionOutput` 的区别在于:

- `DistributionOutput` 描述某类型的分布 (泊松分布、正态分布等), 它和模型绑定, 经过特定数据集的训练之后, 吐出一个 `Distribution`
- `Distribution` 描述了一个参数已定的具体分布, 它的参数经由训练确定

①数据结构

```
class Distribution:
```

distribution.py 中只包含一个类 `Distribution`，用于描述一个具体的（即参数已定的）分布：

- 是个基类，其子类则是不同类型的分布，比如泊松分布、高斯分布等
- 包含了各种决定该分布的参数，比如
 - `mean`：期望值（均值）
 - `stddev`：标准差
 - `variance`：方差，`stddev` 的平方
 - `batch_shape`：batch 的形状，抽象方法
 - `batch_dim`：batch 的维度，`len(batch_shape())`
 - `event_shape`：event 的形状
 - `event_dim`：event 的维度，标量则为 0
 - `all_dim`：`batch_dim + event_dim`
 -
- 提供了各种关于该分布的函数，比如
 - `sample(num_samples)`：从该分布中抽签，数量为 `num_samples`
 - `cdf(x)`：在 `x` 的 CDF（累积分布函数），抽象方法
 - `crps(x)`：在 `x` 的 CRPS（连续概率分级评分），抽象方法
 - `quantile()`：分位数
 - `log_prob(x)`：在点 `x` 的 log 密度，抽象方法
 - `loss(x)`：在点 `x` 的损失，缺省为负的 `log_prob()`
 - `prob(x)`：在点 `x` 的密度，缺省为 `log_prob().exp()`
 -

```
class StudentT(Distribution):
```

描述学生 t 分布的 `StudentT` 是 `Distribution` 的子类

- 有三个标量参数用于描述一个学生 t 分布：
 - `mu`（即 μ ）：期望值，形为 `batch_shape * event_shape`
 - `sigma`（即 σ ）：标准差，形为 `batch_shape * event_shape`
 - `nu`（即 ν ）：自由度，形为 `batch_shape * event_shape`
- 是标量，因此 `batch_shape` 为 `mu.shape`，`event_dim` 为 0

```
class TransformedDistribution(Distribution):
```

transformed_distribution.py 定义了 `TransformedDistribution`，描述基于一个基础分布，经过一系列变换之后得到的分布。

```
class ArgProj(gluon.HybridBlock):  
    class Output:  
        class DistributionOutput(Output):
```

distribution_output.py 定义了 `DistributionOutput`，以及另外两个相关的类，用于描述分布的输出：

- **ArgProj** : MXNet 神经网络模块，用于将网络输出转化为分布参数
 - 是 `gluon.HybridBlock` 的子类
 - 输入是全连接层（即神经网络的输出），输出是 `Distribution` 的参数。
 - 内部处理过程由两部分组成，此二者皆作为参数被传入构造函数：
 - ◆ 先是若干并行的全连接层，输入均是网络输出，输出为若干中间参数（参数的个数、名称和维度来自参数 `args_dim`）
 - ◆ 后接一个映射函数，来自参数 `domain_map`，其输入为中间参数，输出为分布参数
 - 在抽象基类 `Output` 的成员函数 `get_args_proj()` 中生成该类的实例并返回。`args_dim` 和 `domain_map` 都是 `Output` 的成员，但都在具体的子类中被赋值。
- **Output**
 - 是一个抽象基类，`DistributionOutput` 和 `BijectionOutput` 的父类
 - 成员 `domain_map` 和 `args_dim`，参考 `ArgProj`，由子类赋值
 - ◆ `args_dim` : `Distribution` 的参数，字典类型，参数个数为字典长度，key 为输出的参数名，value 为参数的维度
 - ◆ `domain_map()` : 映射函数，输入和输出都是分布参数，其中做了处理
 - 成员函数 `get_args_proj()` : 根据 `domain_map` 和 `args_dim`，返回一个 `ArgProj` 的实例
- **DistributionOutput** :
 - `Output` 的子类
 - 用于产生具体的（即带参数的）分布（即 `Distribution` 的实例）
 - 抽象基类，子类是具体的分布输出，比如：
 - ◆ 学生 T 分布 `StudentTOutput`
 - ◆ 泊松分布输出 `PoissonOutput`
 - ◆ 伽玛分布输出 `GammaOutput`
 - ◆ ...
 - 成员 `distr_cls` 为具体的分布类型，比如 `StudentT`，`Poisson`，`Gamm` 等
 - 成员函数 `distribution()` :
 - ◆ 输入为分布的参数 `distr_args`
 - ◆ 输出为带参数的 `Distribution`
 - ◆ 缺省返回 `distr_cls(distr_args)`，即根据分布参数直接构造一个分布
 - ◆ 可能被子类覆盖

```
class StudentTOutput(DistributionOutput):
    args_dim: Dict[str, int] = {"mu": 1, "sigma": 1, "nu": 1}
    distr_cls: type = StudentT
    def domain_map(cls, F, mu, sigma, nu):
```

`StudentTOutput` 是 `DistributionOutput` 的子类，也是 `gluonts` 中最常用的分布，用于产生学生 t 分布。

这几个关于分布输出的类有点绕，下面以学生 t 分布为例，解构一下：

- `ArgProj` 是神经网络模块，由若干平行 Dense 层 (`args_dim`) 和转换函数 (`domain_map`) 组成，输入为 `glutonts` 神经网络的输出，经过 Dense 层输出中间参数，在经过 `domain_map` 输出分布参数。`args_dim` 和 `domain_map` 是构造函数的参数。

- `Output` 类的大意是：神经网络输出->某种参数，主要向外提供了成员函数 `get_args_proj()` 用于返回 `ArgProj`，此外有成员 `domain_map` 和 `args_dim` 的声明作为 `ArgProj` 的参数，但是没有定义，需子类定义
- `DistributionOutput` 是 `Output` 的子类，定义了 `distribution()` 函数，该函数返回根据输入的分布参数返回一个分布，这里也没有实现 `domain_map` 和 `args_dim`
- `StudentTOutput` 作为 `DistributionOutput` 的子类，终于实现了 `domain_map` 和 `args_dim`，供基类中的函数回调。有的分布会覆盖 `distribution()` 的实现。

以上这些类总体向外提供了 `get_args_proj` 和 `distribution` 两个函数，使用者通过 `get_args_proj()` 得到分布参数（的函数），再将这些参数传入 `distribution()` 得到具体的分布。

②包结构

- `distribution`：分布相关的包
 - `distribution.Distribution`：抽象基类，描述一个具体的概率分布，相关概念可以参考《[概念定义 > NN 相关 > 分布 Distribution](#)》和《[概念定义 > 概率论](#)》
 - ◆ `batch_shape()`：batch shape，抽象方法，需子类实现
 - ◆ `batch_dim`：batch shape 的维度，`len(batch_shape)`
 - ◆ `event_shape()`：event shape，抽象方法，需子类实现
 - ◆ `event_dim`：`event_shape` 的维度，`len(event_shape)`
 - ◆ `all_dim`：所有张量的总维度，`batch_dim + event_dim`
 - ◆ `log_prob()`：根据 `x` 计算概率（以 e 为底的对数），抽象方法，需子类实现
 - `x`：描述事件，其形状为 `batch_shape * event_shape`
 - ◆ `loss()`：根据 `x` 计算 loss，直接返回负的 `log_prob()`
 - ◆ `prob()`：根据 `x` 计算概率，返回 `exp(log_prob())`
 - ◆ `crps()`：
 - ◆ `sample()`：返回若干次抽签，其形状为 `num_samples * batch_shape * event_shape`
 - `num_samples`：返回的抽签数
 - ◆ `mean`：平均值（即期望值）
 - ◆ `stddev`：标准差
 - ◆ `variance`：方差
 - `distribution_output.ArgProj`：是 `HybridBlock` 的子类，实例是 Callable 的，最终会调用到 `hybrid_forward()`，`ArgProj` 的实例用于将 Dense 层转换为 `Distribution` 的参数。在 `Output.get_args_proj()` 中返回其实例。
 - ◆ `__init__()`：构造函数，根据 `args_dim` 生成 `self.proj`（Dense 层的组合），保存 `domain_map`，这些成员在 `hybrid_forward()` 中被使用
 - `args_dim`：字典，key 为 str 类型的 Dense 层名，value 为该层单元数
 - `domain_map`：传入的映射函数
 - `dtype`：

- prefix : 表示层名前缀的字符串，缺省为 None
- ◆ proj : 若干层 Dense 层，构造函数中被赋值
- ◆ hybrid_forward() : 实际的转换过程。根据 self.proj 和 self.domain_map 生成并返回所需的 Distribution 参数（输入 $x > proj > domain_map$ ）
- distribution_output.Output : 抽象基类，给出分布
 - ◆ get_args_proj() : 返回一个 ArgProj 对象
 - ◆ args_dim : get_args_proj() 所返回的 ArgProj 的参数，具体由子类定义
 - ◆ domain_map() : 需子类实现的抽象方法，域的映射，将输入张量转换到合理的形状和值域，是 ArgProj 对象的最后一个步骤
- distribution_output.DistributionOutput : 抽象基类， Output 的子类，通过处理网络的输出结果，输出 Distribution。
 - ◆ distr_cls : 具体的 DistributionOutput 的子类类型
 - ◆ distribution() : 返回一次具体的分布 Distribution。
 - ◆ domain_map() : 父类 Output 的抽象方法，由具体的子类实现
- poisson.Poisson : Distribution 的子类，泊松分布
- poisson.PoissonOutput : DistributionOutput 的子类
 - ◆ args_dim : 覆盖 Output 的 args_dim，值为 { "rate" : 1 }
 - ◆ domain_map() : Soft Plus 函数，被父类 Output 传入 ArgProj 中使用
 - ◆ distribution() : 返回一个 Poisson
- student_t.StudentT : 学生 t 分布， Distribution 的子类
 - ◆ __init__() : 构造函数
 - mu : 期望值的张量，形状为 batch_shape * event_shape
 - sigma : 标准差的张量，形状为 batch_shape * event_shape
 - nu : 自由度的张量，形状为 batch_shape * event_shape
- student_t.StudentTOutput : 学生 t 分布的输出类
 - ◆ domain_map() :

(4) trainer

mx.trainer 目录中包含了训练相关的内容。

① 数据结构

```
class Trainer:
```

mx/trainer/_base.py (老版本是 trainer/_base.py) 中的类 Trainer 定义了一个网络应该如何被训练。这里的参数主要包括两个部分：

- 一部分定义了训练的样本的数量，包括：
 - epochs : 训练多少 epoch，缺省为 100
 - batch_size : 一个 batch 中包含多少个样本，缺省为 32
 - num_batches_per_epoch : 每个 epoch 中多少 batch，缺省为 50

- 另外一部分定义了梯度该如何更新：
 - learning_rate : 学习率，缺省为 $0.001 (10^{-3})$
 - learning_rate_decay_factor : 学习率衰减系数，缺省为 0.5
 - patience : 等待多少轮之后开始降低学习率
 - minimum_learning_rate : 最小学习率，缺省为 10^{-5}
 - clip_gradient :
 - weight_decay : 权值衰减系数
- 除了参数之外，`Trainer` 主要定义了`__call__()`函数（即 `Trainer` 对象可以被直接调用），这也是训练过程的核心函数，在 `Estimator` 的 `train()` 函数中被间接调用。

(5) model

`model` 是模型相关的包，主要包括两种模型的基类 `Estimator`（可训练的模型）和 `Predictor`（不可训练的模型，由 `Estimator` 训练出来），以及各种继承了这两者的具体的（神经网络或非神经网络）模型。

①数据结构

以下是模型相关的数据结构，主要的基类包括：

- `Estimator`：估计器，可训练的模型
- `Predictor`：预测器，不可训练的模型
 - 经 `Estimator` 训练得来 (`Estimator.train()` 返回)
- `Trainer`：训练器，主要描述了一系列训练的参数，及训练过程，被传递给 `Estimator`

在已有模型的情况下，最基本的流程为：

1. 设置 `Trainer`，传递给 `Estimator`
2. `estimator.train()`，参数为训练 dataset，训练出 `Predictor` 返回
3. `predictor.predict()`，参数为测试 dataset

②Estimator

```
class Estimator:
```

`Estimator` 是所有可训练模型的抽象基类，子类包括 `GluonEstimator`、`DummyEstimator`。

```
class DummyEstimator(Estimator):
```

`DummyEstimator` 的构造函数直接传入一个 `predictor`，`train` 并不训练，直接返回这个 `predictor`。

```
class GluonEstimator(Estimator):
```

GluonEstimator 是 Estimator 的主要子类，也是所有 gluon 格式算法模型的父类：

- __init__(): 构造函数，传入 trainer (Trainer 的实例)
- 定义了三个抽象方法 (需要子类实现)：
 - create_transformation()
 - create_training_network()
 - create_predictor()
- train_model(): GluonEstimator 的主要函数，流程为：
 - 由 create_transformation()生成 transformation (处理输入数据)
 - 根据训练数据集，经过 transformation，生成 TrainDataLoader 的实例
 - 根据验证数据集生成 ValidationDataLoader 的实例
 - 由 create_training_network()生成训练网络 trained_net
 - 调用 self.trainer()进行训练
 - 由 create_predictor()生成 Predictor
 - 返回 transformation、trained_net、predictor
- train(): 调用 train_model()，并仅返回 predictor

每个具体的算法都对应一个 Estimator 的子类，其中需要实现 3 个基类中的抽象方法。而每个算法也都对应一个训练网络和一个预测网络，分别用于训练和推理。

③Predictor

所有不可训练网络的基类 Predictor 及其子类的继承关系如下：

- Predictor：
 - RepresentablePredictor：
 - ◆ NPTS、Seasonal Naive、R Forecast、Naive2、Prophet、trivial 下的各种
 - GluonPredictor：
 - ◆ SymbolBlockPredictor：
 - TPP (Temporal Point Process) 算法、Rotbaum Tree
 - ◆ RepresentableBlockPredictor：
 - Canonical、Transformer、Deep Factor、DeepAR、Seq2Seq、N beats、Simple Feedforward、Deep State、GP Forecaster、GPVAR、WaveNet 算法均以次为 Predictor
 - ◆ TPP (Temporal Point Process) 算法、Rotbaum Tree
 - FallBackPredictor：ConstantValuePredictor、MeanPredictor
 - ParallelizedPredictor：Not Used
 - Localizer：Not Used

```
class Predictor:
```

```
    def predict(self, dataset: Dataset, **kwargs) -> Iterator[Forecast]:
```

model.predictor.Predictor 是所有 predictor (训练完成的模型) 的抽象基类

- 成员变量包括：

- prediction_length : 预测的长度，单位为 freq
- freq : 时间步骤的频率，比如 ‘h’
- lead_time :
- 其子类需实现抽象方法 predict()，返回 Forecast 的迭代
- 直接子类包括 RepresentablePredictor 和 GluonPredictor

```
class RepresentablePredictor(Predictor):
    def predict_item(self, item: DataEntry) -> Forecast:
```

`model.predictor.RepresentablePredictor` 是 Predictor 的一个子类：

- 用于给各种非 Gluon 模型作为基类，比如 NPTS、Seasonal Naive 等
- 实现了 predict() 函数，但仅仅是将传入的参数 dataset 的每个迭代作为参数调用了 predict_item()
- predict_item() 是需要子类实现的抽象方法，参数为 dataset 的迭代

```
class GluonPredictor(Predictor):
class RepresentableBlockPredictor(GluonPredictor):
```

`mx.model.predictor.GluonPredictor` 是基于 gluon 的模型的基类：

- 有如下成员变量：
 - prediction_net : 网络模型
 - input_transform : 输入前的变换
 - output_transform : 输出前做的变换
 - forecast_generator : 缺省为 SampleForecastGenerator
- predict() : 根据 dataset 建立 InferenceDataLoader，并通过 forecast_generator 迭代 Forecast

`mx.model.predictor.RepresentableBlockPredictor` 是 GluonPredictor 的子类，也是众多具体的 gluon 模型的 predictor 的基类。

④非 Gluon 模型

```
class ConstantPredictor(RepresentablePredictor):
model.trivial.ConstantPredictor 每次都输出同样的一个 Forecast
```

- 成员变量 samples 保存用于建立 SampleForecast 的采样，初始化时赋值
- predict_item() 每次返回一个由 samples 赋值建立的 SampleForecast 的实例

```
class ConstantValuePredictor(RepresentablePredictor, FallbackPredictor):
model.trivial.ConstantValuePredictor 每次输出一个由固定常量构成的 Forecast
```

- 成员变量 value 用于保存该常量，初始化时候被赋值
- predict_item() 中先根据 value 生成 sample，再根据它生成 SampleForecast 返回

```
class MeanPredictor(RepresentablePredictor, FallbackPredictor):
model.trivial.MeanPredictor 基于传入数据集的最后 context_length 个实践步骤的期望
值和标准差，生成 Sample
```

- 成员变量：
 - context_length：用于定义生成 mean 和 std 的数据集时间步骤长度
 - num_samples：生成采样的数量
 - prediction_length：预测的长度
- predict_item()：根据 context_length 算出 mean 和 std，再根据 mean 和 std 的正态分布，以 num_sample 和 prediction_length 的形状采样出预测的张量，以 SampleForecast 形式返回

⑤SFF 模型

以 Simple Feed Forward 算法为例：

```
class SimpleFeedForwardEstimator(GluonEstimator):
```

- 成员 create_transformation()：返回仅有一个 InstanceSplitter 的 Transformation
- 成员 create_training_network()：返回 SimpleFeedForwardTrainingNetwork
- create_predictor()：
 - 实例化一个 SimpleFeedForwardPredictionNetwork，网络参数来自参数
 - 加上 transformation，构成 RepresentableBlockPredictor，返回

```
class SimpleFeedForwardNetworkBase(mx.gluon.HybridBlock):
```

```
class SimpleFeedForwardTrainingNetwork(SimpleFeedForwardNetworkBase):
```

```
class SimpleFeedForwardPredictionNetwork(SimpleFeedForwardNetworkBase):
```

- SFF 的训练网络和预测网络继承同一个基类 SimpleFeedForwardNetworkBase，由 MLP 和 Distribution 构成
 - 成员变量 mlp 为全连接层的 MLP，来自于构造函数的参数
 - get_distr() 函数输入为网络输入，经过 MLP 和 DistributionOutput，返回分布
- 训练网络 SimpleFeedForwardTrainingNetwork：
 - hybrid_forward() 调用 get_distr()，根据分布返回损失值
- 预测网络 SimpleFeedForwardPredictionNetwork：
 - hybrid_forward() 调用 get_distr()，根据分布返回抽签

⑥DeepAR 模型

以 DeepAR 算法为例：

```
class DeepAREstimator(GluonEstimator):
```

- create_transformation()：返回一个 Transformation
- create_training_network()：返回 DeepARTrainingNetwork
- create_predictor()：返回 transformation 和构成的
 - 实例化 DeepARPredictionNetwork 为预测网络
 - 将训练网络（来自函数参数）的参数传给这个预测网络
 - 根据 transformation（来自函数参数）和预测网络，生成一个 RepresentableBlockPredictor 作为 Predictor 返回

- ```

class DeepARNetwork(mx.gluon.HybridBlock):
class DeepARTrainingNetwork(DeepARNetwork):
class DeepARPredictionNetwork(DeepARNetwork):

```
- `DeepARNetwork` 是 DeepAR 算法的训练网络和预测网络的基类：
    - `get_lagged_subsequences()`
      - ◆ 被 `self.unroll_encoder()` 调用
      - ◆ 被 `DeepARPredictionNetwork.sampling_decoder()` 调用
    - `unroll_encoder()`
      - ◆ 被 `DeepARTrainingNetwork.distribution()` 调用
      - ◆ 被 `DeepARPredictionNetwork.hybrid_forward()` 调用
  - DeepAR 算法的训练网络是 `DeepARTrainingNetwork`：
    - `distribution()`：
      - ◆ 被 `self.hybrid_forward()` 调用
    - `hybrid_forward()`：
  - DeepAR 算法的预测网络是 `DeepARPredictionNetwork`：
    - `sampling_decoder()`：
      - ◆ 被 `self.hybrid_forward()` 调用
    - `hybrid_forward()`：

## ⑦Transformer 模型

以 `Transformer` 算法为例：

- ```

class TransformerEstimator(GluonEstimator):

```
- 成员 `encoder` 是编码器，`TransformerEncoder` 的实例
 - 成员 `decoder` 为解码器，`TransformerDecoder` 的实例
 - `create_transformation()`：返回一个 `Transformation`
 - `create_training_network()`：返回 `TransformerTrainingNetwork`
 - `create_predictor()`：返回 `transformation` 和构成的
 - 实例化 `TransformerPredictionNetwork` 为预测网络
 - 将 `trained_network`（来自函数参数）的参数传给这个预测网络
 - 根据 `transformation`（来自函数参数）和预测网络，生成一个 `RepresentableBlockPredictor` 作为 `Predictor` 返回

- ```

class TransformerNetwork(mx.gluon.HybridBlock):
class TransformerTrainingNetwork(TransformerNetwork):
class TransformerPredictionNetwork(TransformerNetwork):

```
- `TransformerNetwork` 是抽象基类，包括两个函数
    - `get_lagged_subsequence()`
      - ◆ 被 `create_network_input()` 调用
      - ◆ 被 `TransformerPredictionNetwork.sampling_decoder()` 调用
    - `create_network_input()`

- ◆ 被两个子类的 `hybrid_forward()` 调用
- `TransformerTrainingNetwork` 是训练网络
  - `hybrid_forward()`
- `TransformerPredictionNetwork` 是预测网络
  - `sampling_decoder()`
  - `hybrid_forward()`

## ⑧包结构

- `model` : 模型相关
  - `forecast` : 预测结果
    - ◆ `Forecast` : 抽象类，表示预测结果
      - `start_date` : 预测结果的开始时间
      - `freq` : 同 `Estimator.freq`
      - `item_id` :
      - `info` :
      - `prediction_length` : 预测多少个时间步骤
      - `mean` : 平均值
      - `plot()` : 将预测结果绘制成图表（与 `matplotlib.pyplot` 结合）
        - `prediction_intervals` : 所绘制的预测的置信区间，为 0-100 间的浮点数的 List，越小表示置信度越高的部分。
        - `show_mean` : 是否显示
        - `color` : 颜色
    - ◆ `SampleForecast` : `Forecast` 的子类
      - `num_samples` : 每个预测的样本个数
      - `samples` : 用于表示预测结果的张量，`ndarray` 类型，如果每个时间步骤为标量，则形如 `(num_samples, prediction_length)`；如果每个时间步骤为 `target_dim` 维向量，则形如 `(num_samples, prediction_length, target_dim)`
      - `mean` : (`num_samples` 个样本的) 平均值
  - `forecast_generator` :
    - ◆ `ForecastGenerator` :
    - ◆ `DistributionForecastGenerator` :
    - ◆ `QuantileForecastGenerator` :
    - ◆ `SampleForecastGenerator` :
  - `predictor` : 预测器，不可训练的模型，由 `Estimator.train()` 生成
    - ◆ `Predictor` : 抽象类，表示不可训练模型
      - `__init__()` : 构造函数
        - `freq` : 时间粒度，每两个数据帧之间的时间间隔
        - `prediction_length` : 预测输出的长度，即预测几个时间步骤
        - `lead_time` :

- `predict()`: 虚函数，预测并返回 `Forecast` 的迭代
- `serialize()`: 序列化，保存到指定位置
- `deserialize()`: 反序列化，类方法，从指定位置读取 `predictor`
- ◆ `GluonPredictior` : gluon 格式模型的基类，`Predictor` 的子类
  - `__init__()`: 构造函数
    - `prediction_net` : 预测网络 (例 : DeepARPredictionNetwork)
    - `forecast_generator` : 缺省为 `SampleForecastGenerator` 实例
    - `output_transform` : 对输出数据的变换
  - `predict()` : 父类 `predict()` 的实现，调用 `forecast_generator`
- ◆ `RepresentableBlockPredictor` :
- `estimator` : 估计器，可训练的模型
  - ◆ `Estimator` : 抽象类，表示可训练模型
    - `freq` : 时间粒度，每两个数据帧之间的时间间隔
    - `prediction_length` : 预测输出的长度，即预测几个时间步骤
    - `lead_time` :
    - `train()` : 子类须实现的虚函数，训练模型并返回 `Predictor`
      - `training_data` : 训练数据集 (`Dataset`)
      - `validation_data` : 验证数据集 (`Dataset`)
  - ◆ `GluonEstimator` : 继承 `Estimator`，其子类为具体算法的 `Estimator`
    - `__init__()` :
      - `trainer` : 训练器，类型为 `Trainer`
    - `from_hyperparameters()` : 类方法
    - `create_transformation()` : 子类实现的抽象方法，返回一个 `Transformation`，用于在相应的 `Estimator` 处理数据之前做变换。
    - `create_training_network()` : 子类实现的抽象方法，返回训练用网络
    - `create_predictor()` : 子类实现的抽象方法，返回 `predictor`
    - `train_model()` : 训练模型，仅被 `train()` 调用。流程中 先后调用上述几个子类实现的抽象函数，返回结果
    - `train()` : 直接调用 `train_model()`，返回 `predictor`
- `simple_feedforward` : 全连接层构成的 Simple Feed Forward 的模型
  - ◆ `_network` : 包含了 Simple Feed Forward 对应的 MXNet 的网络模型
  - `SimpleFeedForwardNetworkBase` : `mx.gluon.HybridBlock` 的子类，下面训练和推理两个网络模型的基类，实现了全 Dense 层的网络模型。
    - `__init__()`: 构造函数
      - ◆ `num_hidden_dimensions` : MLP 的隐藏层维度
      - ◆ `prediction_length` : 预测长度
      - ◆ `context_length` : 被用于预测的背景长度
      - ◆ `batch_normalization` : 是否使用 BN
      - ◆ `mean_scaling` :
      - ◆ `distr_output` : 输出结果的分布类型，`DistributionOutput` 类型，处理网络输出结果，输出分布，缺省为 `StudentTOutput`。
    - `m1p` : 根据构造函数参数创建的 MLP，私有成员
    - `get_distr()` : 根据输入数据 `past_target`，通过 `m1p` 推理出结果，再返回分布 (根据构造函数的参数 `distr_output`)

- `SimpleFeedForwardTrainingNetwork` : 用于训练的 SFF 网络模型，`SimpleFeedForwardNetworkBase` 的子类
  - `hybrid_forward()` : 返回 loss 值
- `SimpleFeedForwardPredictionNetwork` : 用于推理的 SFF 网络模型，`SimpleFeedForwardNetworkBase` 的子类
  - `hybrid_forward()` : 返回一批结果
- ◆ `_estimator.SimpleFeedForwardEstimator` : `GluonEstimator` 的子类，MLP 结构的 estimator (可训练模型)
  - `__init__()` : 构造函数
    - `freq` : 频率
    - `prediction_length` : 预测长度
    - `trainer` : 训练参数，缺省为 `Trainer` 的缺省实例
    - `num_hidden_dimensions` : 各隐藏层的节点数，缺省为 `None` (对应成员变量为 `[40, 40]`)
    - `context_length` : 影响预测结果的时间步骤，缺省为 `None` (对应成员变量为 `prediction_length`)
    - `distr_output` : 分布的类型，缺省为 `StudentTOutput`
    - `batch_normalization` : 是否使用 BN，缺省为 `False`
    - `mean_scaling` : 缺省为 `True`
    - `num_parallel_sample` : 缺省为 `100`
  - `create_transformation()` : 返回一个只有 `InstanceSplitter` 的 `Chain`
  - `create_training_network()` : 以成员变量为参数 (参考构造函数的参数)，返回一个 `SimpleFeedForwardTrainingNetwork`
  - `create_predictor()` : 返回 predictor
- `deepar` : DeepAR 算法
  - ◆ `_estimator.DeepAREstimator` :
- `waevnet` :
  - ◆ `_estimator.WaveNetEstimator` :
- `kernels` :
- `nursery` :
- `shell` :
- `support` :
- `testutil` :
- `time_feature` :
- `trainer` :
  - `_base.Trainer` : 训练器，包括关于训练的一系列参数以及训练函数，包括：
    - ◆ `__init__()` : 构造函数，传入参数
      - `ctx` : CPU 还是 GPU
      - `epoch` : 缺省为 `100`
      - `batch_size` : 缺省为 `32`
      - `num_batches_per_epoch` : 缺省为 `50`
      - `learning_rate` : 缺省为 `0.001`
      - `learning_rate_decay_factor` : 缺省为 `0.5`
      - `patience` :

- minimum\_learning\_rate
- clip\_gradient
- weight\_decay
- init
- ◆ \_\_call\_\_(): 训练
  - net: 输入的训练网络, 例如 DeepARTrainingNetwork
  - input\_names: 上述网络各层的名称
  - train\_iter: 训练数据加载器, TrainDataLoader 类型
  - validation\_iter: 验证数据加载器, ValidationDataLoader 类型

## (6) forecast

model/forecast.py 描述了几个 forecast 相关的类。Forecast 是用于描述结果的形式之一,

```
class Quantile(NamedTuple):
 value: float
 name: str
```

Quantile 用于描述分位数分割点 (quantile level, 指累计分布函数的值, 参考《概念定义 > 概率论 > 分位数》)

- 是一个命名元组, 用于提供两种格式来描述分割点
  - value 为介于 0 到 1 之间的 float, 为累计分布函数的值
  - name 则是对应的字符串, 比如 value 为 0.2, 则 name 为 “0.2”。
- 用于规范化不同的描述格式, 可以通过以下成员函数得到:
  - from\_float(): 输入是 0 到 1 的浮点数
  - from\_str(): 输入是字串, 可以是 “p20” 或者 “0.20” 这两种形式
  - parse(): 输入是 float 或 str 或 Quantile, 返回对应的 Quantile 对象

```
class Forecast:
```

model.forecast.Forecast 是用于表示预测的抽象基类:

- 是 SampleForecast、QuantileForecast、DistributionForecast 等的父类
- 包括如下成员变量:
  - start\_date: pd.Timestamp, 预测的开始时间戳
  - freq: str, 预测时间段的频率
  - item\_id: Optional[str]
  - info: Optional[Dict], Forecast 额外提供的信息
  - prediction\_length: int, 预测的时间长度, 单位为 freq
  - mean: np.ndarray, 均值
  - index: 时间索引, 是一个 (从 start\_date 开始, 间隔为 freq 的, 长为 prediction\_length 的) pandas.DatetimeIndex
- 包括如下成员函数:

- `plot()` :
- `median()` : 参数为 0.5 的 `quantile()` 函数返回
- `as_json_dict()` :
- `quantile_ts()` : `quantile` 加上 `index` 的 pandas.`Series` 形式
- 包括如下需子类实现的纯虚函数 :
  - `quantile()` : 计算分位数，参数为 float 或者 str (比如 0.3, “0.3”, “p30”，表示累计分布函数 CDF 的值)，返回一个 ndarray，形为(时间步骤, 变量元数)，即 CDF 取到该值时，各个时间步骤的随机变量取值
  - `dim()` : 返回单个时间步骤下的数据维数 (多变量的变量元数)
  - `copy_dim()` : 返回 (样本和时间步骤下) 某个指定的子维度 (sub-dimension)
  - `copy_aggregate()` : 利用传入的 `agg_fun` 函数，对 Forecast 中的数据做聚合 (通常是求和或者求平均值)

```
class SampleForecast(Forecast):
```

`SampleForecast` 是 `Forecast` 的子类，用样本形式来表示预测的结果。

- 覆盖父类的成员变量 :
  - `mean` : 所有样例的均值
- 除父类之外的成员变量包括 :
  - `num_samples` : 样本的数量
  - `samples` : 形状为(`num_samples`, `prediction_length`)，或者在多变量情况下形为(`num_samples`, `prediction_length`, `target_dim`)的数组
  - `mean_ts` : `mean` 和 `index` 组成的 pandas.`Series` 格式
- 类内实现的成员函数包括 :
  - `quantile()` :
  - `dim()` :
  - `copy_dim()` : 当 `samples` 的阶 (即 `len(sample.shape)`) 至少为 2，第一个是各个样本，第二个是各个时间步骤，如果大于 2 的情况下，则返回指定第三个，否则直接返回 `samples`
  - `copy_aggregate()` : 如果样本是单元变量 (即 `samples` 的阶 `len(sample.shape)` 为 2)，则直接返回 `samples`，否则调用传入的 `agg_fun` 对样例做聚合
- 覆盖父类的函数包括 :
  - `as_json_dict()` :

```
class QuantileForecast(Forecast):
```

`QuantileForecast` 是 `Forecast` 的子类，用 `Quantile` 和 `均值` 来表示预测。

- 除父类之外的成员变量 :
  - `forecast_arrays` : ndarray 类型，表示预测的 array，其第一阶的维度 (`shape[0]`) 同 `forecast_keys` 数组的长度
  - `forecast_keys` : List[str] 类型，其元素为从“0.1”到“0.9”的 `Quantile`，或者“`mean`”字符串，每一个对应于 `forecast_arrays` 中的一项。
  - `_forecast_dict` : `forecast_keys` 和 `forecast_arrays` 一一对应形成的字典
  - `_nan_out` : np.nan 构成的长度为 `prediction_length` 的 ndarray
- 覆盖父类的成员变量 :
  - `mean` : 返回 `_forecast_dict[“mean”]`，没有则返回“p50”对应的分位数

- 实现父类的纯虚函数：
  - `quantile()`：根据分割点 (`quantile_level`, float 或者 str) 从 `_forecast_dict` 得到对应的分位数，如果没有则返回 `_nan_out`
  - `dim()`：返回数据维度
    - ◆ 如果 `forecast_arrays` 的阶为 2，则是单元变量，维度为 1
    - ◆ 否则是多元变量，维度为 `forecast_arrays.shape[1]`。**FIXME**：因为这里 `forecast_arrays` 的形为 `[num_samples, target_dim, prediction_length]`?
- 覆盖父类的函数：
  - `plot()`：

```
class DistributionForecast(Forecast):
```

`DistributionForecast` 是 `Forecast` 的子类，根据分布来描述预测

- 除父类之外的成员变量包括：
  - `distribution`: `Distribution`，对应的分布
  - `mean_ts`: `mean` 和 `index` 组成的 `pandas.Series` 格式
- 覆盖父类的成员变量包括：
  - `mean`：返回 `distribution` 的 `mean` (期望值)
- 实现父类的纯虚函数：
  - `quantile()`：调用 `distribution.quantile()` 实现
- 除父类之外，`DistributionForecast` 自己的成员函数：
  - `to_sample_forecast()`：通过 `distribution.sample()` 函数进行抽样，返回 `SampleForecast`

## (7) transform

`transform` 是用于对数据做变换的包。

`Transformation` 是所有变换的抽象基类，

- `transform`：对数据做变换
  - `_base.Transformation`：抽象类，所有变换的基类
    - ◆ `__call__()`：`Transformation` 子类的实例都是 `Callable` 的
      - `data_it`: `DataEntry` 的迭代 (`Iterable`)
      - `is_train`: 训练还是推理，`bool` 值
      - 返回：`DataEntry` 的迭代 (`Iterable`)
  - `_base.Chain` : `Transformation` 的子类，串接各种变换，构造函数的输入是 `Transformation` 的 List
  - `_base.FlatMapTransformation` : `Transformation` 的子类，对输入数据集的每个迭代 (`DataEntry`)，迭代出 0 个至若干个结果 (但不将之合并)
    - ◆ `__call__()`：对输入的每个迭代 (`DataEntry`)，都迭代出 0 至若干个输出，具体的输出由 `flatmap_transform()` 生成
    - ◆ `flatmap_transform()`：抽象方法，需要子类实现

- data : 单条数据，DataEntry 类型
- `_base.MapTransformation` : 抽象基类，`Transformation` 的子类，对输入数据集的每个迭代（DataEntry），迭代一个输出
  - ◆ `__call__()` : 对输入（`Iterable[DataEntry]`）的每个迭代（DataEntry），都迭代一个输出，具体处理过程通过 `map_transform` 实现
  - ◆ `map_transform()` : 具体的处理函数，抽象方法，需要子类实现
    - data : 单条数据，DataEntry 类型
    - is\_train : 训练或推理，bool 类型
- `_base.SimpleTransformation` : 抽象类，`MapTransformation` 的子类
  - ◆ `map_transform()` : 实现了父类的抽象方法，实现仅忽略了参数 `is_train`，直接调用抽象方法 `transform()`
  - ◆ `transform()` : 抽象方法，需要子类实现
    - data : 单条数据，DataEntry 类型
- `feature.AddAgeFeature` : `MapTransformation` 的子类，根据 `target_field` 的长度（如果 `is_train` 为 `False` 则加上 `pred_length`），输出从 0 增长的 Age 字段到 `output_field`。如果 `log_scale`，则 `output_field` 字段对数增长，否则为线性增长。
  - ◆ `target_field` : DataEntry 中的目标字段（即 DataEntry dict 的 key）
  - ◆ `output_field` : 保存输出结果的 DataEntry 字段，通常为“feat\_dynamic\_age”
  - ◆ `pred_length` :
  - ◆ `log_scale` : bool 型，`True` 则为对数增长，否则是线性增长
  - ◆ `map_transform()` : 具体的实现函数，实现父类的抽象方法
    - data : 输入的 DataEntry
    - is\_train : bool 型，如果为 `False` 则 `output_field` 的长度要加上 `pred_length`
- `feature.AddObservedValuesIndicator` : `SimpleTransformation` 的子类，检查样本 `target_field` 字段（DataEntry 的某字段，通常是 numpy 张量）的值是否有缺失，输出结果（同形 bool 张量，`False` 表示元素的值缺失）到 `output_field` 字段。如果 `convert_nans` 为 `True`，则将所有缺失元素替换为 `dummy_value`。
  - ◆ `target_field` : DataEntry 中的目标字段（即 DataEntry dict 的 key）
  - ◆ `output_field` : 保存输出结果的 DataEntry 字段，通常为“observed\_values”
  - ◆ `dummy_value` : 用于替换的值
  - ◆ `convert_nans` : 是否转换空值元素
  - ◆ `transform()` : 具体的实现函数，实现父类的抽象方法
- `feature.AddTimeFeatures` :
- `split.InstanceSplitter` : `FlatMapTransformation` 的子类，

## (8) evaluation

### ①数据结构

```
class Evaluator:
```

### ②包结构

- `evaluation` : 预测结果的评价
  - `_base.Evaluator` : 评价器，用于衡量预测结果的准确性
    - ◆ `quantile` : 分位数，缺省为 0.1, 0.2, ...0.9
    - ◆ `sensonality` :
    - ◆ `alpha` :
    - ◆ `calculate_owa` :
    - ◆ `num_workers` : 执行评估的线程数，缺省为 CPU 核心数
    - ◆ `chunk_size` :
    - ◆ `__call__()` : 执行评估
      - `ts_iterator` : 测试数据集（包括数据和验证）
      - `fcst_iterator` : 需要衡量的预测结果，`Forecast` 的迭代
      - `num_series` : 时间序列样本的数量
      - 返回 tuple :
        - `dict` : 衡量结果（各种统计方式）
        - `pandas.DataFrame` : 具体每个时间序列的衡量结果
  - `_base.MultivariateEvaluator` : 多尺度评价器，
  - `backtest` :
    - ◆ `make_evaluation_prediction()` : 做预测/评估
      - `dataset` : 需要评估的测试数据集，类型为 `Dataset`
      - `predictor` : 用于评估的预测器，类型为 `Predictor`
      - `num_samples` : 对每个时间序列给出的预测结果数量
      - 返回 tuple :
        - `Forecast` 的迭代器 : 预测结果，每个迭代是一个
        - `pandas.Series` 的迭代器
    - ◆ `backtest_metrics()` :

## 5 · gluonnlp (自然语言处理)

gluonnlp 是亚马逊推出的自然语言的专用框架。

## (四) CUHK

CUHK（香港中文大学）在计算机视觉方面颇有建树，推出了 `mmcv`、`mmdetection`、`mmsegmentation`、`mmaction` 等几个开源的框架及库。这几个库都基于 Pytorch，其中：

- `mmdetection` 是图像处理方面的
- `mmsegmentation` 是图像分割方面的
- `mmaction` 是视频理解方面的
- `mmcv` 则无关神经网络，是传统 CV 方面的。

### 1 · `mmdetection`（图像处理）

`mmdetection` 是港中文大学推出的图像处理框架，包括图像分类、目标检测、图像分割等任务。`mmdetection` 基于 Pytorch。

github：  
<https://github.com/open-mmlab/mmdetection>

文档：  
<https://mmdetection.readthedocs.io/en/latest/>

`mmdetection` 的 github 仓库目录结构：

- `setup.py`：编译、安装的脚本
- `config`：描述模型的配置文件
- `demo`：存放 demo
- `docker`：docker 相关
- `docs`：文档
- `mmdet`：`mmdetection` 的源码目录，也是安装之后 site-packages 下 `mmdet` 目录的来源
- `requirements`：依赖的 python 包列表
- `tests`：
- `tools`：用于训练、推理的脚本

### 2 · `mmsegmentation`（图像分割）

`mmsegmentation` 是 CUHK 推出的图像分割专用框架。

支持的骨干网络包括：ResNet、ResNeXt、HRNet 等。

支持的模型包括：FCN、PSPNet、DeepLabV3、PSANet、UPerNet、NonLocal Net、EncNet、CCNet、DANet、GCNet、ANN、OCRNet 等

github：

<https://github.com/open-mmlab/mmsegmentation>

参考：

<https://zhuanlan.zhihu.com/p/164489668>

### 3 · mmaction (视频理解)

**mmaction** 是 CUHK 港中文大学推出的视频理解专用框架。

- 全面支持视频动作分析的各种任务，包括**动作识别** (action recognition)、**时序动作检测** (temporal action detection) 以及**时空动作检测** (spatial-temporal action detection)。
  - 时序动作检测目前仅支持基于 **thumos14** 数据集训练的 **SSN**。
  - 时空动作检测目前仅支持基于 **ava** 数据集训练的 **fast-rcnn**。
- 支持多种流行的数据集，包括 **Kinetics**、**THUMOS**、**UCF101**、**ActivityNet**、**Something-Something**、以及 **AVA** 等。
- 已实现多种动作分析算法框架，包括 **TSN**、**I3D**、**SSN**、以及新的 **spatial-temporal action detection** 方法。**MMAction** 还通过 Model Zoo 提供了多个预训练模型，以及它们在不同数据集上的性能指标。
- 采用高度模块化设计。用户可以根据需要对不同模块，比如 **backbone** 网络、采样方案等等进行灵活重组，以满足不同的应用需要。

参考：

<https://baijiahao.baidu.com/s?id=1636836312959143467&wfr=spider&for=pc>

github：

<https://github.com/open-mmlab/mmaction>

**mmaction** 的 github 仓库目录结构如下：

- **compile.sh**：编译脚本
- **setup.py**：安装脚本 (`python setup.py develop`)
- **data**：空目录，用于存放会用到的数据集（由 **data\_tools** 中的脚本下载）
- **data\_tools**：各个数据集的下载脚本
  - **ava**：AVA 数据集的工具脚本
    - ◆ **download\_annotations.sh**：下载标签
    - ◆ **download\_videos.sh**：下载视频
    - ◆ **extract\_frames.sh**：抽取视频帧的脚本
  - **hmdb51**：HMDB51 的工具脚本

- `kinetics400` : Kinetics400 的工具脚本
- `thumos14` : Thumos14 的工具脚本
- `ucf101` : UCF101 的工具脚本
- `mmaction` : mmaction 的源码目录，安装后 python 的 site-pacakge/mmaction.egg-link 也连接至此
- `modelzoo` : 空目录，用于存放下载的模型权重文件
- `configs` 和 `test_configs` : 模型的配置文件
  - ava : 时空行为检测的网络模型，目前仅有 AVA 数据集上的 **Fast-RCNN** 模型
    - ◆ `ava_fast_rcnn_n1_r50_c4_1x_kinetics_pretrain_crop.py`
  - hmdb51 : hmdb51 数据集上的模型
  - thumos14 : 时序行为检测的网络，目前仅有 thumos14 数据集上的 **SSN**
    - ◆ `ssn_thumos14_rgb_bn_inception.py` :
  - CSN : CSN 网络模型的配置
  - I3D\_Flow : I3D 光流模型的配置
  - I3D\_RGB : I3D RGB 模型的配置
  - R2plus1D : R(2+1)D 模型的配置
  - SlowFast : SlowFast 模型的配置
  - SlowOnly : Slow 模型的配置
  - TSN : TSN 模型的配置
- `third_party` : 用到的第三方源码
- `tools` : 训练、测试各种网络的脚本
  - `test_recognizer.py` : 测试行为识别网络
  - `test_localizer.py` : 测试时序行为识别网络
  - `test_detector.py` : 测试时空行为识别网络
- `INSTALL.md` : 安装指南
- `MODEL_ZOO.md` : 下载预训练权重的指南
- `DATASET.md` : 下载数据集的指南
- `GETTING_STARTED.md` : 操作指南

mmaction 的训练、测试需要：

- 程序文件：即 tools 目录下的 py 文件等
- 配置文件：configs 目录下的 py 文件
- 权重文件：.pth 格式文件，一般下载到 modelzoo 中

命令行格式形如：

```
./tools/test_xxx.py configs/xxx.py
```

例如

```
./tools/test_detector.py
```

```
test_configs/TSN/tsn_kinetics400_2d_rgb_r50_seg3_f1s1.py
```

## (五) 图森未来

### 1 · SimpleDet (图像处理)

SimpleDet 是图森未来 (TuSimple) 推出的图像处理框架。包括目标检测和实例分割任务

代码：

<https://github.com/TuSimple/simpledet>

## (六) Hugging Face

Hugging Face 总部位于纽约，是一家专注于自然语言处理、人工智能和分布式系统的创业公司。他们所提供的聊天机器人技术一直颇受欢迎，但更出名的是他们在 NLP 开源社区上的贡献。Huggingface 一直致力于自然语言处理 NLP 技术的平民化(democratize)，希望每个人都能用上最先进(SOTA, state-of-the-art)的 NLP 技术，而非困窘于训练资源的匮乏。

官方网站：

<https://huggingface.co/>

github：

<https://github.com/huggingface>

### 1 · Transformers

参考《研究方向：NLP > 框架 > Transformers》

## (七) 阿里巴巴

### 1 · MNN

参考《边缘计算 > MNN》

## (八) 腾讯

### 1 · ncnn

参考《[边缘计算 > ncnn](#)》

## (九) Redmon

Joseph Redmon 是著名的目标检测模型 YOLO 的作者，YOLO 并没有使用各种流行框架，还是他自己实现了一个简单的 Darknet 框架（C 语言），通过配置文件定义网络模型。

Redmon 自己实现了三个版本的 YOLO（即 YOLO、YOLOv2、YOLOv3），之后为了抗议 YOLO 被用于军事和隐私问题（他自己讲的，我看其实是他觉得烦了），不再做 YOLO 的研发。

而 YOLOv4 算是“半官方”，是他的同事开发的（也一直有在参与之前 YOLO 的开发），主要是加入了很多 trick 使得模型更准确，并没有什么特别创新的思想提出。YOLOv5 则完全不官方，是一家德国公司做的工程化的产品，也是加了大量的 trick，效果和速度据说比 YOLOv4 更好。

### 1 · Darknet

Darknet 是 YOLO 系列原作者 Redmon 开发的一套框架，C 语言的实现和接口。Darknet 虽然通用性不够，但是工程化相当不错。

官网：

<https://pjreddie.com/darknet/yolo/>

代码：

<https://github.com/pjreddie/darknet>

Darknet 是 C 语言的，下载（clone）后通过 make 生成一个名为 darknet 的可执行文件，该文件是 darknet 框架的主文件，可以用于训练、推理等等一系列操作。

**混淆注意！**

Darknet 是有歧义的，可以指代这个深度学习框架，也可以指代使用这个框架开发的一个网络模型，这个网络模型本身可以用于图像分类，同时也是 YOLO（目标检测网络）的骨干网络。

## (1) 使用说明

以下为 Darknet 主程序（用于训练、执行 darknet 格式的网络配置和权重）使用说明

- 下载及编译

```
git clone https://github.com/pjreddie/darknet
cd darknet
可以选择修改 Makefile (GPU=1, CUDNN=1, OPENCV=1)
make
```

- 获取权重

```
cd cfg
wget https://pjreddie.com/media/files/yolov3.weights
wget https://pjreddie.com/media/files/yolov3-tiny.weights
```

- 图像分类

```
./darknet classifier predict cfg/imagenet1k.data cfg/darknet19.cfg
cfg/darknet19.weights data/dog.jpg
```

or input image path on prompt:

```
./darknet classifier predict cfg/imagenet1k.data cfg/darknet19.cfg
cfg/darknet19.weights
```

- 目标检测

```
./darknet detect cfg/yolov3.cfg yolov3.weights data/dog.jpg
```

相当于：

```
./darknet detector test cfg/coco.data cfg/yolov3.cfg yolov3.weights
data/dog.jpg
```

手动输入图片路径：

```
./darknet detect cfg/yolov3.cfg yolov3.weights
```

指定阈值（缺省为 0.25）：

```
./darknet detect cfg/yolov3.cfg yolov3.weights data/dog.jpg -thresh 0.1
```

使用 yolo-tiny：

```
./darknet detect cfg/yolov3-tiny.cfg yolov3-tiny.weights data/dog.jpg
```

摄像头实时检测：

```
./darknet detector demo cfg/coco.data cfg/yolov3.cfg yolov3.weights
```

检测视频文件：

```
./darknet detector demo cfg/coco.data cfg/yolov3.cfg yolov3.weights <video
file>
```

- 训练

在 VOC 上训练（需要设置好其他配置）：

```
./darknet detector train cfg/voc.data cfg/yolov3-voc.cfg darknet53.conv.74
```

在 COCO 上训练（需要设置好其他配置）：

```
./darknet detector train cfg/coco.data cfg/yolov3.cfg darknet53.conv.74
```

- 冻结部分网络（前 N 层，下例为 15）

```
./darknet partial cfg/yolov3-tiny.cfg yolov3-tiny.weights yolov3-tiny.conv.15
15
```

- OpenCV 测试

```
./darknet imtest data/eagle.jpg
```

## (2) 文件说明

- darknet 文件

darknet 的主体，编译生成的 ELF 执行文件

- .cfg 文件

Darknet 的.cfg 文件用于描述**网络结构及超参数**，通常位于 cfg 目录下，其参数定义：

[net] 字段

<https://github.com/AlexeyAB/darknet/wiki/CFG-Parameters-in-the-%5Bnet%5D-section>

层字段

<https://github.com/AlexeyAB/darknet/wiki/CFG-Parameters-in-the-different-layers>

- .weights 文件

.weights 文件是权重数据，须与 cfg 文件描述的网络结构对应

- .data 文件

.data 文件用于配置目标检测器（Object Detection）或者图像分类器（Image classification），通常位于 cfg 目录下。

目标检测器的配置 cfg/coco.data，定义如下：

```
classes= 20 #种类
train = /home/pjreddie/data/voc/train.txt #训练集图片列表
valid = /home/pjreddie/data/voc/2007_test.txt #验证集图片列表
names = data/voc.names #各个分类的名字
backup = backup #存放训练中备份的权重文件的目录
```

- 数据格式

YOLO 有自己的数据格式，训练集或者验证集的图片列表被放在某个 txt 文件里，这些文件

被.data 文件中的 train、valid 字段指定。训练集图片的 label 文件有如下格式：

1. label 文件与图片文件同名，扩展名改为 txt
2. label 文件与图片文件同目录，或者图片文件在 images 目录，label 文件在同级 labels 目录，形如 aaa/bbb/images/pic1.jpg、aaa/bbb/labels/pic1.txt
3. label 文件中每一行代表一个 object，有 5 个值，分别为类别、中心 x、中心 y、宽度、高度，这 5 个值以空格隔开，后面四个值取值范围从 0 到 1，为点或者长度占全宽/全高的比例。

- .names 文件

.names 文件通常位于 data 目录中，按行存放了每个分类的名字

## (十) 其它

### 1 · PyVideoResearch

代码：

<https://github.com/gsig/PyVideoResearch>

### 2 · Theano

Theano 是蒙特利尔大学开发的深度学习框架，流行性低，且已经停止开发。

官方 API 文档：

<http://www.deeplearning.net/software/theano/library/index.html>

### 3 · sklearn

sklearn，即 sciki-learn，严格来说并非一个深度学习的框架，而是一个机器学习框架，也就是说其涵盖的内容比通常的 DL 库要更广。

### 4 · DNN (OpenCV)

OpenCV 的 DNN 模块 (Deep Neural Network)

doc :

[https://docs.opencv.org/master/d2/d58/tutorial\\_table\\_of\\_content\\_dnn.html](https://docs.opencv.org/master/d2/d58/tutorial_table_of_content_dnn.html)

# 五、传统图像视频处理（OpenCV）

本章以 OpenCV 为顺序，介绍一些传统图像视频处理的概念

官网：

<https://opencv.org/>

英文 python 教程：

[https://docs.opencv.org/master/d6/d00/tutorial\\_py\\_root.html](https://docs.opencv.org/master/d6/d00/tutorial_py_root.html)

中文 python 教程：

<http://www.opencv.org.cn/opencvdoc/2.3.2/html/doc/tutorials/tutorials.html>

OpenCV Python：

- `imread()`：从文件中读取图像（返回 `numpy.ndarray` 格式）
- `imwrite()`：将图像写入文件
- `imshow()`：显示图像
- `VideoCapture`：视频流，读取自文件或者摄像头（参数为数字时）
- `read()`：读取一帧图像（返回 `numpy.ndarray` 格式）
- `release()`：释放该视频流
- `get()`：获取视频流的相关信息（高、宽、帧率等）
- `VideoWriter`：写入视频流
- `write()`：写入一帧图像
- `release()`：释放 `img`
- `copyTo()`：掩码

## （一）基本绘图

OpenCV Python：

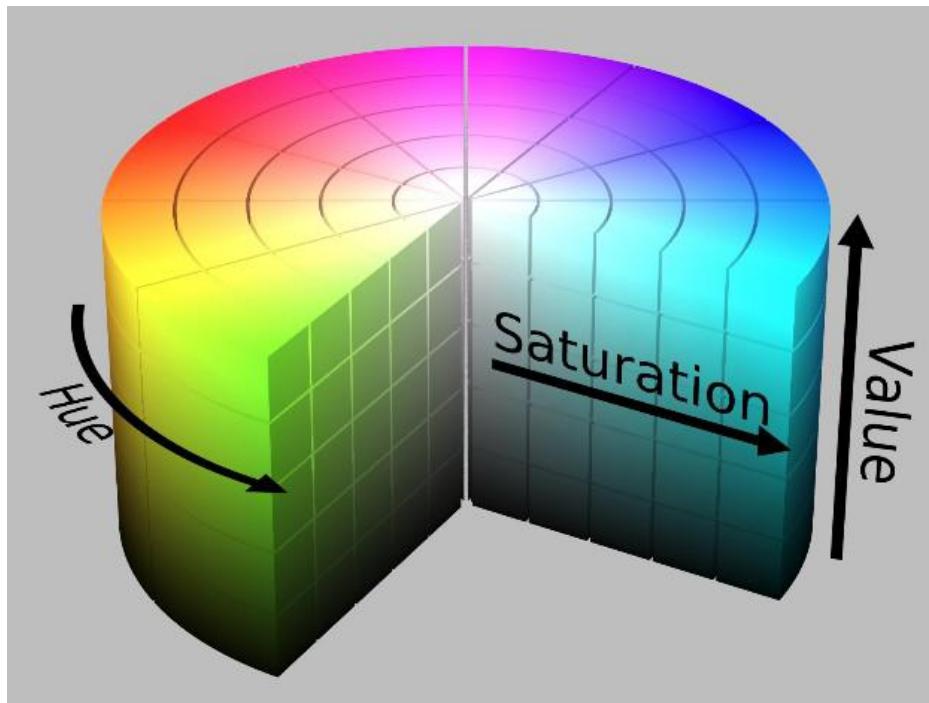
- `line()`：画线
- `circle()`：画圆
  - `img`：输入图像
  - `center`：圆心坐标  $(x, y)$
  - `radius`：半径
  - `color`：颜色  $(R, G, B)$
  - `thickness`：圆环的粗细， $-1$  表示充满整个圆
- `rectangle()`：绘制矩形
  - `img`：输入图像
  - `pt1`：第一个顶点
  - `pt2`：第二个顶点（第一个的对角）

- color : 颜色
- thickness : 轮廓线的粗细

## (二) 色彩空间 (Colorspace)

色彩空间是对色彩的组织方式，RGB 格式的色彩空间很简单，分别用三个分量表示红色 R、绿色 G、蓝色 B 的大小。

相较于 RGB 格式的色彩空间，HSV 色彩空间更加贴合人类的感受。



H : 色相 (Hue) , 即平时所说的颜色，黄色、红色等

S : 饱和度 (Saturation) , 指色彩的纯度，越高色彩越纯，低则逐渐变灰，取 0-100%

V : 明度 (Value) , 即亮度

RGB 到 HSV 的转换公式如下，假设 max 为 R,G,B 中最大者，min 为最小者：

$$h = \begin{cases} 0^\circ & \text{if } max = min \\ 60^\circ \times \frac{g-b}{max-min} + 0^\circ, & \text{if } max = r \text{ and } g \geq b \\ 60^\circ \times \frac{g-b}{max-min} + 360^\circ, & \text{if } max = r \text{ and } g < b \\ 60^\circ \times \frac{b-r}{max-min} + 120^\circ, & \text{if } max = g \\ 60^\circ \times \frac{r-g}{max-min} + 240^\circ, & \text{if } max = b \end{cases}$$

$$s = \begin{cases} 0, & \text{if } max = 0 \\ \frac{max-min}{max}, & \text{if } max = 1 - \frac{min}{max}, \\ & \text{otherwise} \end{cases}$$

$$v = max$$

OpenCV Python :

- `split()`：参数为  $h*w*c$  的图像，返回三路颜色通道，长宽为  $h*w$
- `merge()`：合并  $N$  路颜色通道
- `cvtColor()`：颜色空间的转换，比如 `BGR2GRAY`
  - `src`：输入图像
  - `code`：转换方式，型为 `COLOR_xxx2yyy`，格式可以为 `BGR`、`BGRA`、`BGR565`、`RGB`、`RGBA`、`GRAY`、`HSV`、`XYZ` 等等，例如：
    - ◆ `cv2.COLOR_BGR2BGRA`：`BGR` 通道格式加上 `Alpha` 通道
    - ◆ `cv2.COLOR_BGR2GRAY`：置为灰色
    - ◆ `cv2.COLOR_BGR2HSV`：转为 `HSV` 色彩空间
  - 返回：输出图像
- `inRange()`：检查每个像素是否在某个范围之内
  - `src`：输入图像
  - `lowerb`：像素取值下限
  - `upperb`：像素取值上限
  - 返回：同形灰度矩阵（8 位单通道），对应像素在范围内则为 255（白），否则为 0（黑）

`colorsys` 包中有 `hsv_to_rgb()` 等函数可直接进行 `RGB` 和 `HSV` 三元组之间的函数转换

### (三) 几何变换 (Geometry Transformation)

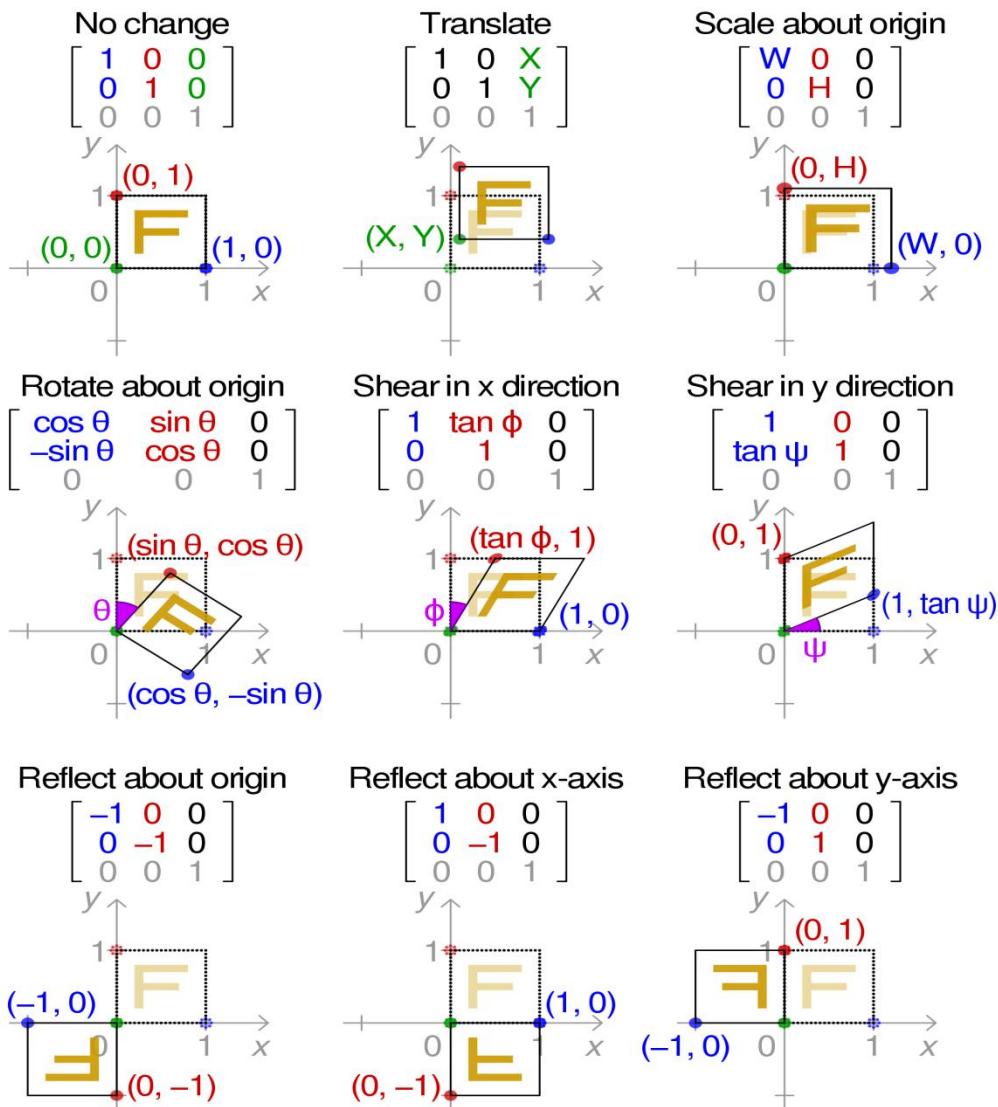
参考《[概念定义 > 线性代数/几何学 > 齐次坐标](#)》

变换模型是指根据待匹配图像和背景图象之间几何畸变的情况，所选择的能最佳拟合两幅图像之间变化的几何变换模型，可采用的变换模型有刚性变换、仿射变换、透视变换等等。

这些变换由一些基础的线性变换构成，包括：

- 平移 (Translation)
- 旋转 (Rotation)
- 镜像 (Reflect)
- 剪切 (Shear)
- 缩放 (Scale)
- Elation

各种基础变换的**变换矩阵**和效果图示如下：



## 1 · 刚体变换 (Rigid Transformation)

如果一幅图像中任意两点的距离经变换后仍保持不变，则这种变换称为刚体变换（刚性变换），刚体变换通常包括：

- 平移 (Translation)
- 旋转 (Rotation)
- 镜像 (Reflect)

在二维空间中，点 $(x, y)$ 经过刚体变换到点 $(x', y')$ 的变换矩阵为：

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} \pm\cos\phi & \pm\sin\phi & t_x \\ \pm\sin\phi & \pm\cos\phi & t_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

上式中 $\phi$ 表示旋转的角度， $[t_x, t_y]^T$ 表示平移变量，参考之前各种基础变换的矩阵，可以看出这个矩阵是由平移、旋转、镜像三个变换的矩阵构成

## 2 · 仿射变换 (Affine Transformation)

仿射变换是对一个向量空间进行一次线性变换，并接上一个平移，变换为另一个向量空间。

仿射变换保持了：

- 点的共线性：在同一直线上的三个或者更多的点在变换后仍然在同一直线上
- 线的平行性：parallelness，在原图中平行的线变换后仍然平行
- 平行线段的长度的比例

二维空间中，点 $(x, y)$ 经过仿射变换到点 $(x', y')$ 的变换矩阵为：

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} a_1 & a_2 & t_x \\ a_3 & a_4 & t_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

其中 $\begin{bmatrix} a_1 & a_2 \\ a_3 & a_4 \end{bmatrix}$ 为线性变换的矩阵， $[t_x, t_y]^T$ 为平移变量。

由于线性变换的矩阵可以为任意值，因此仿射变换涵盖了：

- 平移 (Translation)
- 旋转 (Rotation)
- 镜像 (Reflect)
- 缩放 (Scale)
- 剪切 (Shear)

## 3 · 投影变换 (Projective Transformation)

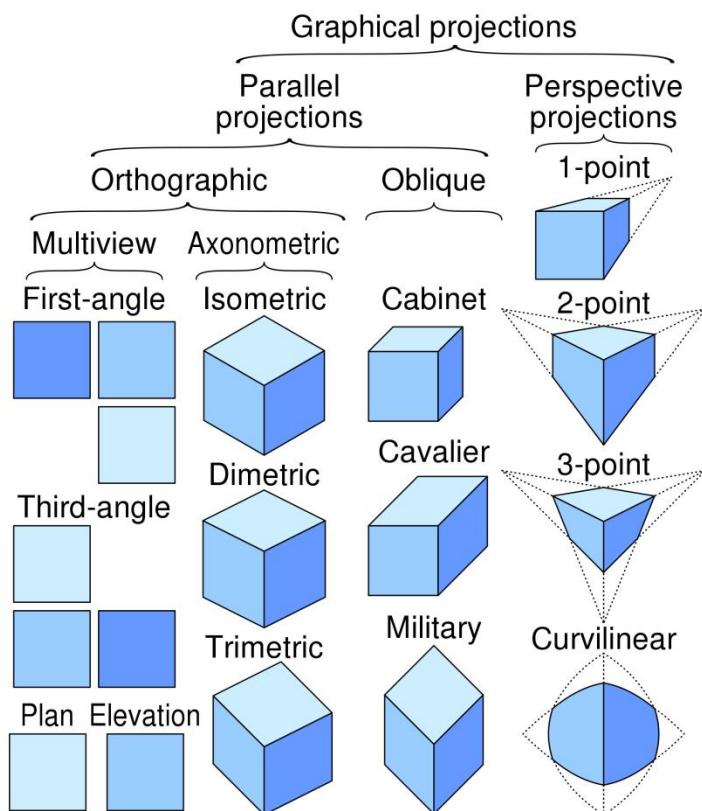
首先来了解一下 3D 投影。

### (1) 投影

投影 (Projection) 指的是将一个 3 维物体显示在一个 2 维的平面（被称为视平面，Viewing Plane，或者叫投影平面，Projection Plane）上，比如设计中用到的三视图，即是一种投影。投影大致分为平行投影和透视投影两大类，详细分类如下：

- 平行投影 (Parallel Projection) : 投影中心和投影平面的距离是无限的，投影线相互平行。因此在 3 维物体中平行的线在投影中依然平行
  - 正投影 (Orthographic Projection) : 投影线垂直于投影平面
    - ◆ 多视图投影 (Multiview Projection) : 物体的 3 个坐标面分别与投影面平行，形成三视图：正视图、侧视图、俯视图

- ◆ 轴测投影 (Axonometric Projection) : 物体的坐标面与投影面都不平行
- 斜投影 (Oblique Projection) : 投影线不垂直于投影平面
- 透视投影 (Perspective Projection) : 投影中心和投影平面的距离是有限的 (此时投影线汇于一点, 即人眼或镜头)。因此在 3 维物体中平行的线在投影中可能不平行。
  - 一点透视 (1-point Perspective) : 有 1 个灭点, 物体有两个维度的棱平行于视平面, 剩下 1 个维度的棱消失于唯一的灭点
  - 两点透视 (2-point Perspective) : 有 2 个灭点, 物体有一个维度的棱平行于视平面, 剩下 2 个维度的棱分别消失于 2 个灭点
  - 三点透视 (3-point Perspective) : 有 3 个灭点, 物体三个维度的棱均不平行于视平面, 而消失于 3 个灭点



投影变换本质上是射影几何的齐次坐标系的 3 维向量, 经过一个  $3 \times 3$  的矩阵变换为另一个 3 维向量, 其变换矩阵为:

$$\begin{bmatrix} u \\ v \\ w \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} \begin{bmatrix} u' \\ v' \\ w' \end{bmatrix}$$

$$\Rightarrow k_{p2} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} k_{p1} \begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix}$$

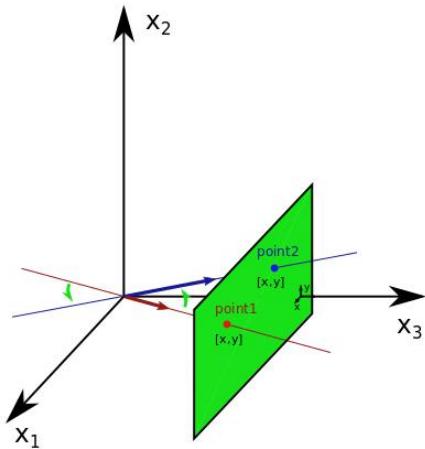
$$\Rightarrow \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \frac{k_{p1}}{k_{p2}} \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} \begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix}$$

因为齐次坐标系中的系数可以为任意值而不会改变意义，将矩阵整体乘以  $1/a_{33}$ ，可得：

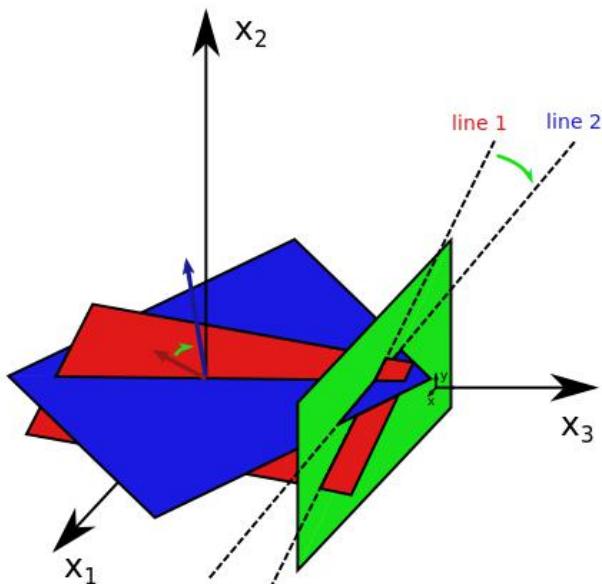
$$\Rightarrow \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = k \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & 1 \end{bmatrix} \begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix}$$

这使得这个变换矩阵的自由度降为 8 个。

下图中齐次坐标系的两个点（即 3 维空间中的红蓝色直线）分别是变换前和变换后的点：



下图中齐次坐标系的两条直线（即 3 维空间中的红蓝色平面）分别为变换前后的线：



图中的绿色箭头表示了变换的方向。

**投影变换**的  $3 \times 3$  矩阵在某些特殊值的情况下，可以表示一些特殊的变换，比如之前提到的刚体变换和仿射变换：

## (2) 等距同构变换

等距同构 (Isometric) 变换指的是变换前后的距离保持不变 (比如旋转、平移)，包括 3 个自由度，即旋转角度  $\theta$  和在两个方向上的平移  $t_x$  和  $t_y$ ，变换矩阵如下：

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} \varepsilon \cos\theta & -\sin\theta & t_x \\ \varepsilon \sin\theta & \cos\theta & t_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

其中  $\varepsilon$  等于  $\pm 1$ 。可以将等距同构变换矩阵拆解为 4 个部分：

$$x' = H_E x = \begin{bmatrix} R & t \\ 0^T & 1 \end{bmatrix} x$$

## (3) 相似变换

相似 (Similar) 变换包括缩放、平移、旋转，包括 4 个自由度：旋转角度  $\theta$ 、缩放系数  $s$  和在两个方向上的平移  $t_x$  和  $t_y$ ，变换矩阵如下：

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} \varepsilon s \cos\theta & -s \sin\theta & t_x \\ \varepsilon s \sin\theta & s \cos\theta & t_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

其中  $s$  为缩放的比例，其他同等距同构矩阵，也可以被分为四个部分：

$$x' = H_s x = \begin{bmatrix} sR & t \\ 0^T & 1 \end{bmatrix} x$$

## (4) 仿射变换

仿射变换的包括了旋转、平移、缩放、剪切，变换矩阵如下：

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} & t_x \\ a_{21} & a_{22} & t_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

仿射变换包括 6 个自由度，对应于矩阵中的 6 个元素，如果将变换矩阵拆解为 4 个部分：

$$x' = H_A x = \begin{bmatrix} A & \mathbf{t} \\ \mathbf{0}^T & 1 \end{bmatrix} x$$

这 6 个自由度的其中 2 个包含在上图中的  $\mathbf{t}$  里（位移  $t_x$  和  $t_y$ ），另外 4 个也可以用更加可视化的方式表现出来，通过将上图中的非奇异矩阵  $A$  拆解为如下几个矩阵的积：

$$A = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \begin{bmatrix} \cos(-\Phi) & -\sin(-\Phi) \\ \sin(-\Phi) & \cos(-\Phi) \end{bmatrix} \dots \begin{bmatrix} \lambda_1 & 0 \\ 0 & \lambda_2 \end{bmatrix} \begin{bmatrix} \cos\phi & -\sin\phi \\ \sin\phi & \cos\phi \end{bmatrix}$$

剩下 4 个自由度为：旋转角度  $\theta$  和  $\phi$ 、两个轴上的缩放系数  $\lambda_1$  和  $\lambda_2$ 。这 6 个自由度的具体化的表示为：

1. 将图像旋转  $\phi$  度
2. 在 x 和 y 轴上分别缩放  $\lambda_1$  和  $\lambda_2$  倍
3. 将图像旋转回去（旋转  $-\phi$  度）
4. 再旋转  $\theta$  度
5. 做两个轴上分别平移  $t_x$  和  $t_y$

## (5) 投影变换 (2D 平面上)

完全的投影变换包括 8 个自由度，其变换矩阵如下：

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & v \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

这里  $v=1$  或者  $0$ ，拆解成 4 个部分则为：

$$x' = H_p x = \begin{bmatrix} A & \mathbf{t} \\ \mathbf{v}^T & v \end{bmatrix} x$$

可以将整个投影变换矩阵表示为四个矩阵之积：

$$H = \begin{bmatrix} s\cos\theta & -\sin\theta & t_x \\ s\sin\theta & s\cos\theta & t_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & k & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \dots \begin{bmatrix} \lambda & 0 & 0 \\ 0 & 1/\lambda & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ v_1 & v_2 & v \end{bmatrix}$$

矩阵链中的每个矩阵也可以拆为四个部分：

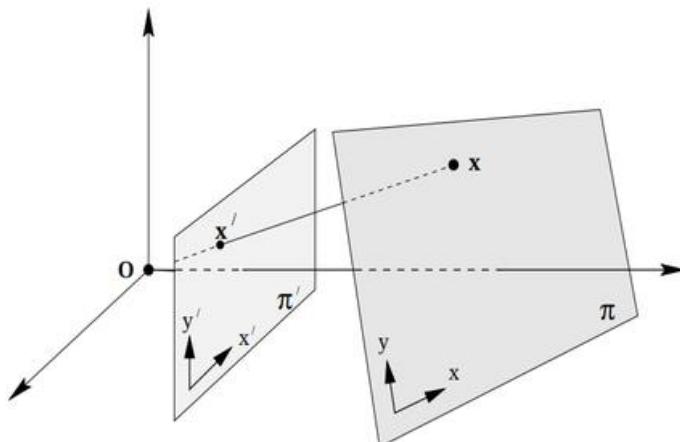
$$H = \begin{bmatrix} sR & \mathbf{t} \\ \mathbf{0}^T & 1 \end{bmatrix} \begin{bmatrix} k & \mathbf{0} \\ \mathbf{0}^T & 1 \end{bmatrix} \begin{bmatrix} \lambda & \mathbf{0} \\ \mathbf{0}^T & 1 \end{bmatrix} \begin{bmatrix} I & \mathbf{0} \\ \mathbf{V}^T & v \end{bmatrix}$$

1. 最左的矩阵为一个相似变换的矩阵，有 4 个自由度
2. 第二个矩阵其实是一个剪切变换矩阵，有 1 个自由度
3. 第三个矩阵是一个 1 个自由度的缩放矩阵，分别在 x 和 y 轴上放大了  $\lambda$  和  $1/\lambda$  倍
4. 最右边的矩阵是一个全新的变换 elation，有 2 个自由度

## (6) 3D 空间的理解

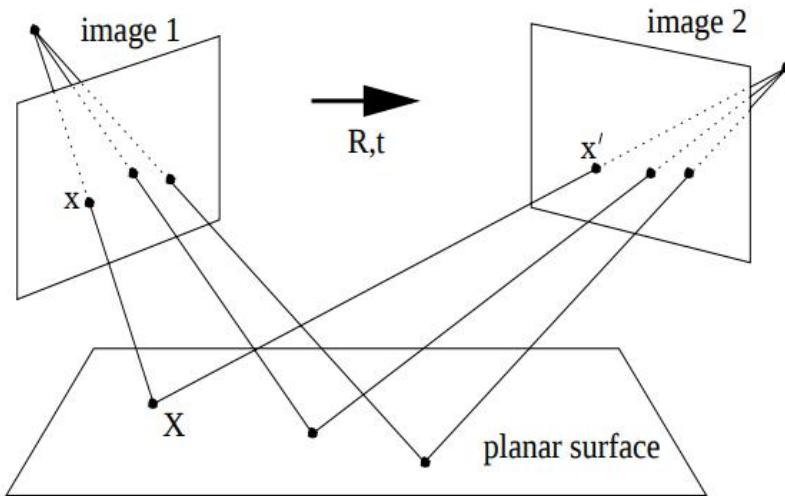
以上是从在 2D 平面做变换的角度去理解投影变换。但既然 **投影变换**包括投影 (Perspective)，自然跟投影也有关系。

通常 **投影变换**都是指**透视变换** (Perspective Transformation)，透视变换是透视投影变换的缩写，和在同一个平面的变换（刚体变换、仿射变换及各种基础变换）不同，透视变换是将物体/点（在视平面  $\pi'$  的成像）投影到一个新的视平面  $\pi'$ 。然后根据这个平面得到物体/点在其中的 2D 坐标。

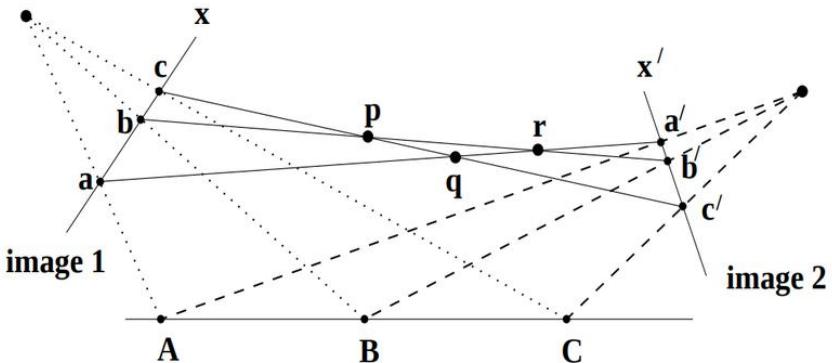


投影变换是一个“群”（Group，数学概念），换句话说，任何一个投影变换的逆变换也是投影变换，任意两个投影变换的组合也是一个投影变换。

举个很常用的例子，从世界平面投影到一个 camera1 的成像平面是一个投影变换，到 camera2 也是。下图是世界平面到 2 个 camera 的投影变换：



考虑到刚才所说，从 camera1 的成像平面，投影回世界平面（逆变换），再投影到 camera2 的成像平面（组合变换），这个变换组合可以被视作一个单独的投影变换，由于 3D 不好表示，用 2D 表示如下：



这种组合变换代表的意义即是对一个平面的不同角度的摄像画面。

参考：

<https://mc.ai/part-ii-projective-transformations-in-2d/>

混淆注意！

透视变换和一个 3D 物体分别投影在两个不同的视平面是有区别的，当物体投影在第一个视平面的时候，其位置信息已经丢失了一部分（3D 变为 2D），因此不可能无损的将一个物体在视平面 A 的投影转换为其在视平面 B 的投影。

透视变换和透视投影这两个名词经常混淆，文中定义的透视变换有时候也会用“透视投影”这个词来表示，那么根据在这里的定义，这两者有什么区别和联系呢？

- 区别：透视投影是 3D 物体到 2D 视平面的投影，而透视变换中不涉及 3D 物体
- 联系：透视变换可以被视作是一个 2D 物体（视平面 A）到 2D 视平面的透视投影

透视变换和仿射变换的区别和联系如下：

- 区别：透视变换中有 2 个视平面，是物体从一个视平面到另一个视平面的投射
- 联系：仿射变换可以被视为是透视变换的一个特例，当透视变换矩阵中的部分元素取特定值时，即变成仿射变换。

## 4 · OpenCV Python

OpenCV Python 中和几何变换相关的各种函数

- `resize()`：缩放图片
  - `src`：输入图片
  - `dsize`：目标大小，格式为 `(width, height)`
  - `dst`：可选参数，输出图片
  - `fx`：可选参数，水平方向放大倍数，`dsize` 需设为 `None`
  - ：可选参数，垂直方向放大倍数，`dsize` 需设为 `None`
  - `interpolation`：可选参数，插值方式
    - ◆ `INTER_LINEAR`：bilinear 插值
    - ◆ `INTER_CUBIC`：bicubic 插值
    - ◆ ...
  - 返回：输出图片
- `getRotationMatrix2D()`：生成旋转变换矩阵
  - `center`：旋转中心的坐标，格式为 `(x, y)`，单位为像素
  - `angle`：旋转的角度，单位为度
  - `scale`：缩放比例
- `getAffineTransform()`：生成仿射变换所需矩阵
  - `src`：原图的三个点，`(3,2)`的 ndarray，`dtype` 为 `float32`
  - `dst`：目标图像的三个对应的点，`(3,2)`的 ndarray，`dtype` 为 `float32`
  - 返回：生成的仿射变换矩阵，`(2,3)`的 ndarray
- `warpAffine()`：根据变换矩阵，对图像进行仿射变换
  - `src`：输入图片
  - `M`：仿射变换矩阵，`(2,3)`的 ndarray，可由 `getAffineTransform` 生成，也可自行定义（参考几何变换）
  - `dsize`：输出图片的尺寸
  - 返回：输出图片
- `getPerspectiveTransform()`：生成透视变换所需矩阵
  - `src`：原图的四个点，`(4,2)`的 ndarray，`dtype` 为 `float32`
  - `dst`：目标图像的四个对应的点，`(4,2)`的 ndarray，`dtype` 为 `float32`
  - 返回：生成的透视变换矩阵，`(3,3)`的 ndarray
- `warpPerspective()`：根据变换矩阵，对图像进行透视变换
  - `src`：输入图片

- M : 透视变换矩阵，(3,3)的 ndarray
- dsize : 输出图片的尺寸
- 返回 : 输出图片

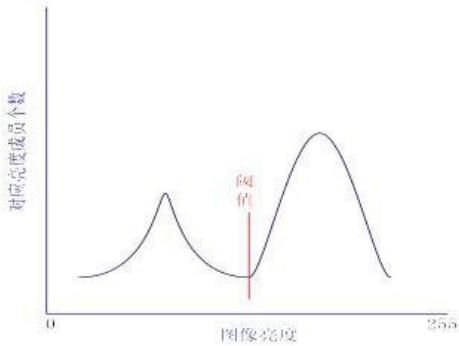
## (四) 阈值处理 (Threshold)

### 1 · 大津算法 (Otsu's Method)

在计算机视觉和图像处理中，大津二值化法用来自动对基于聚类的图像进行二值化，或者说，将一个灰度图像退化为二值图像。该算法以大津命名。

算法假定该图像根据**双模直方图**（前景像素和背景像素）把包含两类像素，于是它要计算能将两类分开的最佳阈值，使得它们的类内方差最小；由于两两平方距离恒定，所以即它们的类间方差最大。

直观的来说，就是找到图像的**直方图**中如下红线位置：



OpenCV Python :

- threshold() : 阈值处理
  - ◆ src : 输入图像，可以为单通道或者多通道（对每个通道分别做阈值处理）
  - ◆ thresh : 阈值
  - ◆ maxval : THRESH\_BINARY 和 THRESH\_BINARY\_INV 用到
  - ◆ type : 类型，如下
    - cv2.THRESH\_BINARY : 大于阈值则为 maxval，否则为 0
    - cv2.THRESH\_BINARY\_INV : 大于阈值则为 0，否则为 maxval
    - cv2.THRESH\_TRUNC : 大于阈值则为 thresh，否则不变
    - cv2.THRESH\_TOZERO : 大于阈值不变，否则为 0
    - cv2.THRESH\_TOZERO\_INV : 大于阈值为 0，否则不变
    - cv2.THRESH\_OTSU : 大津算法，参考《[图像视频处理 > 阈值处理 > 大津算法](#)》
- adaptiveThreshold() : 自适应二值化处理

- ◆ src : 输入图像
- ◆ maxValue : 最大值，即 1 的情况下填的值
- ◆ adaptiveMethod : 自适应阈值处理方法
  - cv2.ADAPTIVE\_THRESH\_MEAN\_C : 阈值为该像素周围区域（长宽都为 blockSize）的平均值减去常量 C
  - cv2.ADAPTIVE\_THRESH\_GAUSSIAN\_C : 阈值为该像素周围区域（长宽都为 blockSize）的加权和（即高斯模糊）减去常量 C
- ◆ thresholdType :
  - cv2.THRESH\_BINARY : 大于阈值时为 maxValue，否则为 0
  - cv2.THRESH\_BINARY\_INV : 小于阈值时为 maxValue，否则为 0
- ◆ blocksize : 区域的长和宽
- ◆ C : 常量 C

## (五) 过滤器-模糊

类似 1 维信号中的 HPF (高通滤波器)，或者 LPF (低通滤波器)，图像中也有类似的处理，可用于过滤噪声，平滑图像。被称为 **图像模糊 (Image Blurring)**，或者 **图像平滑 (Image Smoothing)**。

### 1 · 2D 卷积 (2D Convolution)

根据卷积核对图像做卷积，是下面的均值模糊等操作的一般情况。

OpenCV Python :

- filter2D() : 对图像做卷积运算 (模糊)
  - src : 输入图像
  - ddepth : 图像位数，-1 表示跟输入一致
  - kernel : 卷积核

### 2 · 均值模糊 (Averaging Blur)

取每个点周围区域的平均值作为该点的值。例如一个 3x3 的均值模糊卷积核为：

$$K = \frac{1}{9} \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

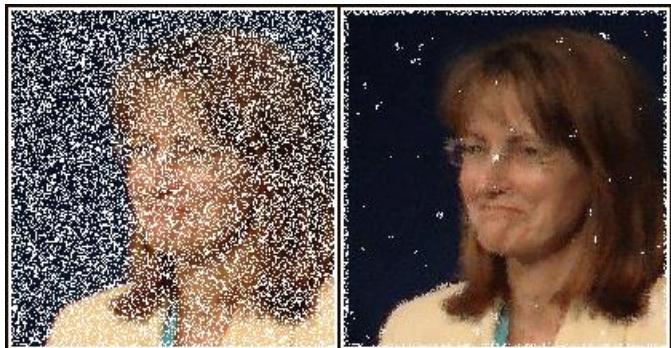
OpenCV Python :

- blur() : 2D 卷积的特殊情况，卷积核固定为 ksize 指定的均值矩阵
  - src : 输入图像
  - ksize : 卷积核尺寸，例如 (5,5)
  - ddepth : 图像位数，-1 表示跟输入一致

### 3 · 中值模糊 (Median Blur)

也叫中值滤波 (Median Filter) ，用于去除图像或者信号中噪声的技术，具有保存边缘的特点。

对于图像来说，其算法就是在像素取值为其周围区域（被称为窗口/window）的中值（注意不是均值）。



OpenCV Python :

- medianBlur() : 中值模糊
  - src : 输入图像
  - ksize : 卷积核尺寸，例如 (5,5)

参考：

[https://en.wikipedia.org/wiki/Median\\_filter](https://en.wikipedia.org/wiki/Median_filter)

### 4 · 高斯模糊 (Gaussian Blur)

也叫高斯平滑 (Gaussian Smoothing) 或者高斯滤波 (Gaussian Filter) ，图像处理手段，用于减少图像噪声，及降低细节。

其算法就是图像与正态分布做卷积，由于正态分布又叫高斯分布，因此被叫做高斯模糊。

OpenCV Python :

- GaussianBlur() : 高斯模糊
  - src : 输入图像，每个通道独立卷积
  - ksize : 卷积核尺寸，例如 (5,5)
  - sigmaX : 标准差，为 0 表示由 ksize 计算得来

参考：

<https://zh.wikipedia.org/wiki/%E9%AB%98%E6%96%AF%E6%A8%A1%E7%B3%8A>

## 5 · 双边滤波 (Bilateral Filter)

在图像处理上，双边滤波器为使影像平滑化的非线性滤波器。

图像去噪的方法很多，如中值滤波，高斯滤波，维纳滤波等等。但这些降噪方法容易模糊图片的边缘细节，对于高频细节的保护效果并不明显。相比较而言，双边滤波器可以很好的边缘保护，即可以在去噪的同时，保护图像的边缘特性。双边滤波 (Bilateral filter) 是一种非线性的滤波方法，是结合图像的空间邻近度和像素值相似度的一种折衷处理，同时考虑空域信息和灰度相似性，达到保边去噪的目的。

双边滤波器之所以能够做到在平滑去噪的同时还能够很好的保存边缘 (Edge Preserve) ，是由于其滤波器的核由两个函数生成：空间域核和值域核

- 空间域核  $w_d$  衡量的是 p, q 两点之间的距离，距离越远权重越低。

$$w_d(i, j, k, l) = \exp\left(-\frac{(i - k)^2 + (j - l)^2}{2\sigma_d^2}\right)$$

- 值域核  $w_r$  衡量的是 p, q 两点之间的像素值相似程度，越相似权重越大。

$$w_r(i, j, k, l) = \exp\left(-\frac{\|f(i, j) - f(k, l)\|^2}{2\sigma_r^2}\right)$$

在平坦区域，临近像素的像素值的差值较小，对应值域权重接近于 1，此时空域权重起主要作用，相当于直接对此区域进行高斯模糊。因此，平坦区域相当于进行高斯模糊。

在边缘区域，临近像素的像素值的差值较大，对应值域权重接近于 0，导致此处核函数下降 (因)，当前像素受到的影响就越小，从而保持了原始图像的边缘的细节信息。

最终的权值则为  $w_d$  和  $w_r$  之积：

$$w(i, j, k, l) = w_d(i, j, k, l) * w_r(i, j, k, l) = \exp\left(-\frac{(i - k)^2 + (j - l)^2}{2\sigma_d^2} - \frac{\|f(i, j) - f(k, l)\|^2}{2\sigma_r^2}\right)$$

OpenCV Python :

- `bilateralFilter()` : 双边滤波
  - `src` : 输入图像
  - `d` : 核的直径
  - `sigmaColor` : 值域核的标准差
  - `sigmaSpace` : 空间域核的标准差

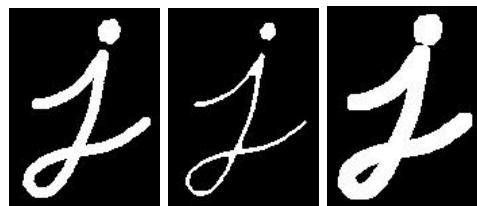
参考：

<https://blog.csdn.net/guyuealiam/java/article/details/82660826>

[https://en.wikipedia.org/wiki/Bilateral\\_filter](https://en.wikipedia.org/wiki/Bilateral_filter)

## (六) 形态变换 (Morphological Transformation)

以下三图分别为原图、侵蚀 (Erosion) 、膨胀 (Dilation)



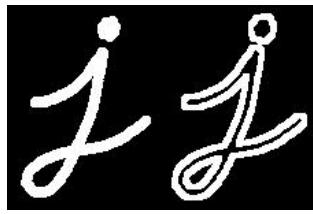
下图为 Opening，是一个侵蚀后跟一个膨胀，可用于消除小的噪点



下图为 Closing，是一个膨胀后跟一个侵蚀，可用于消除小的孔洞



下图为 Morphological Gradient，是膨胀和侵蚀的差，可形成物体的轮廓



OpenCV Python :

- erode() : 缩减，边缘侵蚀
  - src : 输入图像
  - kernel : 核，比如 np.ones((5,5), np.uint8)
- dilate() : 扩张，膨胀
  - src : 输入图像
  - kernel : 核，比如 np.ones((5,5), np.uint8)
- morphologyEx() : 基于 erode 和 dilate 的高级形态变换
  - src : 输入图像
  - op : 变换类型
    - ◆ cv2.MORPH\_ERODE : 同 erode()
    - ◆ cv2.MORPH\_DILATE : 同 dilate()
    - ◆ cv2.MORPH\_OPEN : 先 erode 再 dilate，可以用于消除背景中的噪点
    - ◆ cv2.MORPH\_CLOSE : 先 dilate 再 erode，可以用于填充物体中的小洞
    - ◆ cv2.MORPH\_GRADIENT : dilate() 和 erode() 之差，输出结果物体的轮廓
    - ◆ cv2.MORPH\_TOPHAT : 输入图像和其 Opening 图像之差
    - ◆ cv2.MORPH\_BLACKHAT : 输入图像和其 closing 图像之差
  - kernel : 核，比如 np.ones((5,5), np.uint8)。

## (七) 图像导数 (Image Gradient)

图像导数指的是将像素位置视为 x，像素取值视为 y，得到的函数的导数。其中一阶导数通常用于图像边缘检测，二阶导数的符号可以确定边缘的过渡是从亮到暗还是从暗到亮，在对图像做导数之前最好先进行平滑处理，因为导数操作对噪声敏感。

### 1 · 索伯算子 (Sobel Operator)

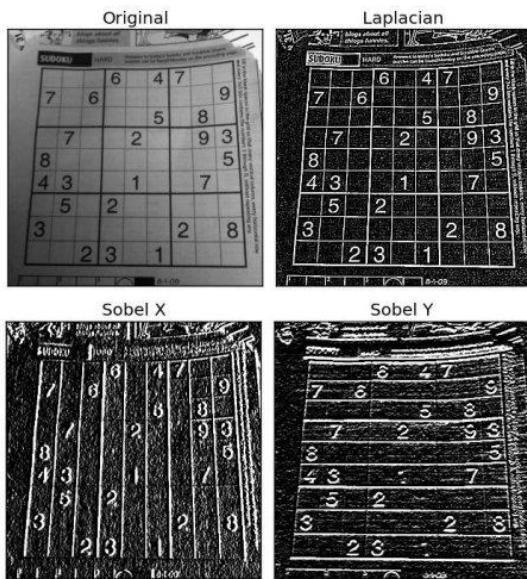
图像处理中的算子之一，在机器视觉领域常被用于做边缘检测。也叫索伯边缘检测、索伯变换。

索伯算子是 2 组  $n \times n$  (通常为  $3 \times 3$ ) 的矩阵，分别为横向边缘检测和纵向边缘检测的卷积核，用该两组卷积核和图像做平面卷积，即可得到横向及纵向的边缘检测结果。

如果  $A$  为原始图像， $G_x$  为经横向边缘检测后的图像， $G_y$  为纵向边缘检测的图像，矩阵大小为  $3 \times 3$ ，则有：

$$G_x = \begin{bmatrix} +1 & 0 & -1 \\ +2 & 0 & -2 \\ +1 & 0 & -1 \end{bmatrix} * A \quad \text{and} \quad G_y = \begin{bmatrix} +1 & +2 & +1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix} * A$$

索伯变换的效果如下图：



参考：

[https://en.wikipedia.org/wiki/Sobel\\_operator](https://en.wikipedia.org/wiki/Sobel_operator)

[https://docs.opencv.org/master/d5/d0f/tutorial\\_py\\_gradients.html](https://docs.opencv.org/master/d5/d0f/tutorial_py_gradients.html)

OpenCV Python：

- `Sobel()`：索伯变换，用于检测横向或纵向的边缘
  - `src`：输入图像
  - `ddepth`：输出图像深度
  - `dx`：x 轴方向的标准差
  - `dy`：y 轴方向的标准差
  - `ksize`：核的尺寸，缺省为 3

## 2 · Scharr 算子 (Scharr Operator)

**Scharr 算子**是索伯算子的优化和变形，在图像处理中通常用以下  $3 \times 3$  卷积核表示：

$$\begin{bmatrix} +3 & 0 & -3 \\ +10 & 0 & -10 \\ +3 & 0 & -3 \end{bmatrix} \quad \begin{bmatrix} +3 & +10 & +3 \\ 0 & 0 & 0 \\ -3 & -10 & -3 \end{bmatrix}$$

以下三幅图分别为原图、索伯变换、Scharr 变换：



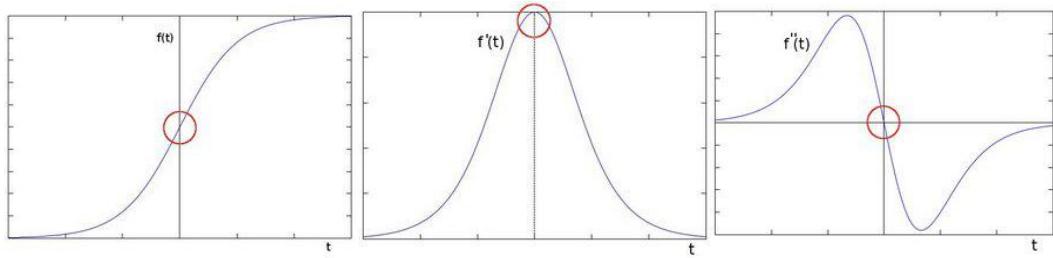
OpenCV Python :

- `Scharr()` : Scharr 变换，索伯变换的变形
  - `src` : 输入图像
  - `ddepth` : 输出图像深度
  - `dx` : x 轴方向的标准差
  - `dy` : y 轴方向的标准差

### 3 · 拉普拉斯算子 (Laplace Operator)

拉普拉斯算子 (Laplace Operator, 或 Laplacian) 是 n 维欧几里德空间中的一个二阶微分算子。

在图像处理中来说，拉普拉斯算子经常被用来作为边缘检测的手段。图像中边缘处的像素值变化更大，其一阶导数必然更大，而一阶导数极值位置，二阶导数必然为 0，下面三图分别为原函数图像、一阶导数、二阶导数：



数字图像处理中，拉普拉斯算子通常被表现为如下的 3×3 卷积核：

$$\mathbf{D}_{xy}^2 = \begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix}$$

OpenCV 中的拉普拉斯函数 `Laplacian()` 缺省采取以上矩阵，此外也采用以下扩展矩阵作为卷积核：

$$\mathbf{D}_{xy}^2 = \begin{bmatrix} 1 & 1 & 1 \\ 1 & -8 & 1 \\ 1 & 1 & 1 \end{bmatrix}.$$

参考：

<https://blog.csdn.net/songzitea/article/details/12842825>

[https://en.wikipedia.org/wiki/Discrete\\_Laplace\\_operator](https://en.wikipedia.org/wiki/Discrete_Laplace_operator)

OpenCV Python：

- `Laplacian()`：拉普拉斯变换
  - `src`：输入图像
  - `ddepth`：输出图像深度

## (八) Canny 边缘检测算法

Canny 边缘检测算子（Canny Edge Detector）是澳洲计算机科学家约翰·坎尼（John F. Canny）于 1986 年开发出来的一个多级边缘检测算法。更为重要的是 Canny 创立了“边缘检测计算理论”（computational theory of edge detection）解释这项技术如何工作。

简单来讲，Canny 将边缘检测需要达到的目标总结为如下三条准则：

1. **好的检测**：检测算法应该精确地找到图像中的尽可能多的边缘，减少漏检和误检。
2. **最优定位**：检测的边缘点应该精确地定位于边缘的中心。
3. **最小响应**：图像中的任意边缘应该只被标记一次，同时图像噪声不应产生伪边缘。

为此，Canny 采用了如下步骤：

- 1 · 使用高斯滤波器，以平滑图像，滤除噪声。

## 2 · 计算图像中每个像素点的梯度强度和方向。

采用 Sobel 算子、Prewitt 算子等等来实现，OpenCV 中用的是 Sobel 算子。

## 3 · 应用非极大值抑制（Non-Maximum Suppression），消除边缘检测带来的杂散响应。

这个步骤的目的是为了上述的第二个准则最优定位，注意这个步骤是沿着梯度方向进行的，简单来说就是使得检测出来的边缘变窄并位于中心。

具体操作是：

- (1) 将当前像素的梯度强度与沿正负梯度方向上的两个像素进行比较。
- (2) 如果当前像素的梯度强度与另外两个像素相比最大，则该像素点保留为边缘点，否则该像素点将被抑制。

## 4 · 应用双阈值（Double-Threshold）检测来确定真实的和潜在的边缘。

相较于简单的单阈值决定是否边缘，双阈值检测的逻辑如下：

- (1) 如果边缘像素梯度高于高阈值，则视为强边缘
- (2) 如果梯度低于低阈值，则视为噪点被抑制
- (3) 如果介于高低阈值之间，则被视为弱边缘

## 5 · 通过抑制孤立的弱边缘最终完成边缘检测。

检测每一个弱边缘像素，如果其相邻的 8 个像素中有一个是强边缘，则该边缘像素被视为强边缘保留。而所有没有连接强边缘的弱边缘则会被抑制。

参考：

<https://www.cnblogs.com/nowgood/p/cannyedge.html>  
[https://en.wikipedia.org/wiki/Canny\\_edge\\_detector](https://en.wikipedia.org/wiki/Canny_edge_detector)

OpenCV Python :

- Canny() : Canny 边缘检测
  - image : 输入图像
  - threshold1 : 低阈值
  - threshold2 : 高阈值

## 1 · Douglas-Peucker 算法 (TODO)

# (九) 图像金字塔 (Image Pyramids)

OpenCV Python :

- `pyrDown()` : 向下采样，采用 Gaussian Pyramid
- `pyrUp()` : 向上采样，采用 Laplacian Pyramid

## (十) 轮廓 (Contour)

### 1 · 矩 (moment)

图像的**矩**是指图像的某些特定像素灰度的加权平均值（矩），或者是图像具有类似功能或意义的属性。

图像矩通常用来描述图像中分割后的对象。可以通过图像的矩来获得图像的部分性质，包括**面积(或总体亮度)**，以及有关**几何中心和方向**的信息。

矩分为**原始矩**（也叫**几何矩**）和**中心矩**，中心矩具有平移不变性。

#### (1) 原始矩

对于二维连续函数  $f(x, y)$ ,  $(p+q)$  阶的原始矩被定义为

$$M_{pq} = \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} x^p y^q f(x, y) dx dy$$

对于灰度图像的像素的强度  $I(x, y)$ , 图像的原始矩  $M_{ij}$  被计算为：

$$M_{ij} = \sum_x \sum_y x^i y^j I(x, y)$$

原始矩包括以下一些有关图像性质的信息：

1.  $M_{00}$  为图像的**灰度质量**（如果是二值图则为**图像的面积**）

$$2. \text{ 图像的几何中心可以表示为 } \{\bar{x}, \bar{y}\} = \left\{ \frac{M_{10}}{M_{00}}, \frac{M_{01}}{M_{00}} \right\}$$

#### (2) 中心矩

中心矩在平移时具有**平移不变性**，**中心矩**定义为：

$$\mu_{pq} = \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} (x - \bar{x})^p (y - \bar{y})^q f(x, y) dx dy$$

其中  $\{\bar{x}, \bar{y}\}$  为图像的几何中心，如果  $f(x, y)$  是一个数字图像，则前一公式等价于

$$\mu_{pq} = \sum_x \sum_y (x - \bar{x})^p (y - \bar{y})^q f(x, y)$$

参考：

<https://www.cnblogs.com/ronny/p/3985810.html>

[https://zh.wikipedia.org/wiki/%E7%9F%A9\\_\(%E5%9B%BE%E5%83%8F\)](https://zh.wikipedia.org/wiki/%E7%9F%A9_(%E5%9B%BE%E5%83%8F))

OpenCV Python：

- `findContours()`：寻找二值图像（8位单通道图像的非0像素均被视为1）中的轮廓
  - `image`：输入图像，8位单通道
  - `mode`：组织模式
    - ◆ `cv2.RETR_EXTERNAL`：仅寻找图像中物体的外部轮廓
    - ◆ `cv2.RETR_LIST`：寻找所有轮廓，但不确立他们的层次结构
    - ◆ `cv2.RETR_CCOMP`：寻找所有轮廓，并分为两级，上一级为外部轮廓，外部轮廓中的为内部轮廓（即物体中洞的轮廓），如果某个内部轮廓中又有轮廓，则又视为外部轮廓
    - ◆ `cv2.RETR_TREE`：寻找所有轮廓，并根据他们的包含关系建立一个完整的层次结构
  - `method`：轮廓近似模式
    - ◆ `cv2.CHAIN_APPROX_NONE`：列出所有的点
    - ◆ `cv2.CHAIN_APPROX_SIMPLE`：只列出线段（横线、竖线、斜线）端点
  - 返回：两个返回值
    - ◆ `contours`：轮廓列表
    - ◆ `hierarchy`：轮廓的层次结构，三阶的 ndarray，第二个阶是各个轮廓，最后一阶的维度为4，表示每个轮廓的[Next, Previous, First\_Child, Parent]
- `drawContours()`：画出轮廓
  - `image`：图像
  - `contours`：轮廓列表
  - `contourIdx`：需要画出的轮廓的索引，-1 表示所有轮廓
  - `color`：轮廓的颜色
  - `thickness`：轮廓线的粗细，缺省为1，-1 表示填满轮廓中的部分
- `moments()`：获得图像的矩
- `contourArea()`：计算轮廓面积
  - `contour`：轮廓
- `arcLength()`：计算弧线（轮廓）周长

- curve : 一段弧线或者轮廓
  - closed : 布尔值，是否为闭环（比如轮廓）
- approxPolyDP() : 得到弧线（轮廓）的更少顶点的近似线段（多边形），利用 Douglas-Peucker 算法
  - curve : 输入的线段（轮廓）
  - epsilon : 新线段和原弧线之间允许的最大距离，用于表示精度
  - closed : 布尔值，是否为闭环（轮廓）
- convexHull() : 得到一系列点的外接凸多边形（有时和 apporxPo1yDP 输出相同）
  - points : 一系列点
- isContourConvex() : 检查轮廓是否为凸多边形
  - contour : 轮廓
- boundingRect() : 得到弧线（轮廓）的 straight bounding box (横平竖直那种)
  - array : 一段弧线或轮廓
  - 返回 : bounding rect 的 x, y (左上角), w, h
- minAreaRect() : 得到一系列点的最小面积 bounding box (可以斜着)
  - points : 一系列点（可以是弧线、轮廓）
- minEnclosingCircle() : 得到一系列点的最小外接圆
  - points : 一系列点（可以是弧线、轮廓）
  - 返回 : 圆心和半径
- fitEllipse()
- fitLine()
- convexityDefects() :
- pointPolygonTest() :
- matchShapes() :

## (十一) 直方图 (Histogram)

**方向梯度直方图 (Histogram of Oriented Gradient)**，是应用在计算机视觉和图像处理领域，用于目标检测的特征描述器。这项技术是用来计算局部图像梯度的方向信息的统计值。

## (十二) 模板匹配 (Matching Template)

**模板匹配 (Matching Template)** 用于在一幅图像中寻找一个小的模板图像

OpenCV Python :

- matchTemplate() : 搜索目标图像，比对模板
  - image : 目标图像，大小为(W, H)
  - templ : 模板图案，大小为(w, h)
  - method : 搜索方法
    - ◆ cv2.TM\_SQDIFF : 目标和模板的平方差，匹配越好，匹配值越小

- ◆ cv2.TM\_SQDIFF\_NORMED : 标准化的平方差 (除以方-和-积-根)
- ◆ cv2.TM\_CCORR : 互相关匹配, 目标和模板的乘积之和, 匹配越好, 匹配值越大
- ◆ cv2.TM\_CCORR\_NORMED : 标准化的互相关匹配
- ◆ cv2.TM\_CCOEFF : “目标像素减平均值” 和 “模板像素减平均值”的乘积之和, 匹配越好, 匹配值越大
- ◆ cv2.TM\_CCOEFF\_NORMED : 标准化的CCOEFF
- 返回: 匹配值矩阵, 大小为 (W-w+1, H-h+1), 每个元素为模板顶点 (而非中心) 在目标图像中的位置
- minMaxLoc() : 从矩阵中找到极大值、极小值以及他们的位置
  - src : 匹配值矩阵
  - 返回: 最小值, 最大值, 最小值位置, 最大值位置

## (十三) 霍夫变换 (Hough Transform)

霍夫变换是一种特征提取, 被广泛应用在图像处理和机器视觉。霍夫变换是用来辨别找出物件中的特征。他的算法流程大致如下, 给定一个物件、要辨别的形状的种类, 算法会在参数空间中执行投票来决定物体的形状, 而这是由累加空间 (accumulator space) 里的局部最大值来决定。

经典的霍夫变换是侦测图片中的直线, 之后, 霍夫变换不仅能识别直线, 也能够识别任何形状, 常见的有圆形、椭圆形。

参考:

<https://zh.wikipedia.org/wiki/%E9%9C%8D%E5%A4%AB%E5%8F%98%E6%8D%A2>

[http://www.opencv.org.cn/opencvdoc/2.3.2/html/doc/tutorials/imgproc/imgtrans/hoough\\_lines/hoough\\_lines.html#hoough-lines](http://www.opencv.org.cn/opencvdoc/2.3.2/html/doc/tutorials/imgproc/imgtrans/hoough_lines/hoough_lines.html#hoough-lines)

OpenCV Python :

- HoughLines() : 标准霍夫线变换, 检测直线
  - 返回: 一组极径和极角的组合( $r, \theta$ )
- HoughLinesP() : 统计概率霍夫变换, 检测线段, 执行效率更高
  - 返回: 一组线段的端点(x0, y0, x1, y1)
- HoughCircles() : 霍夫圆变换, 检测圆形
  - 返回: 描述圆心和直径的三元组(x, y, r)
  - image : 输入图像 (8位、单通道、灰度图像)
  - method : 检测方法, 目前只实现了 cv2.HOUGH\_GRADIENT
  - dp :
  - minDist : 检测到的圆之间的最小距离, 如果这个值太小, 可能会重复检测一个圆 (即一个圆被检测到多次), 如果太大, 可能会漏检。
  - circles : 同返回值

- param1 : 用到的 Canny 边缘检测的高阈值（低阈值为其一半）
- param2 : cv2.HOUGH\_GRADIENT 方法的累加器阈值，越小检测的圆越多
- minRadius : 检测出的圆允许的最小半径
- maxRadius : 检测出的圆允许的最大半径

## (十四) Harris 角检测器 (TODO)

## (十五) SIFT 算法 (TODO)

## (十六) SURF 算法 (TODO)

## (十七) 图像分割 (Image Segmentation)

OpenCV Python :

- watershed() : 分水岭算法，用于分割图像
- grabCut() : GrabCut 算法

## (十八) 视频处理 (Video Process)

### 1 · 均值漂移 (mean-shift)

均值漂移 (mean-shift) 算法，在聚类、图像平滑、图像分割和跟踪方面得到了比较广泛的应用。

1. 在未被标记的数据点中随机选择一个点作为起始中心点 center；
2. 找出以 center 为圆心半径为 radius 的区域中出现的所有数据点。
3. 以 center 为圆心点，计算从 center 开始到集合 M 中每个元素的向量，将这些向量相加，得到向量 shift。
4. center = center + shift。即 center 沿着 shift 的方向移动，移动距离是 ||shift||。
5. 重复步骤 2、3、4，直到 shift 很小（就是迭代到收敛），得到最终收敛的 center。

## 2 · 光流

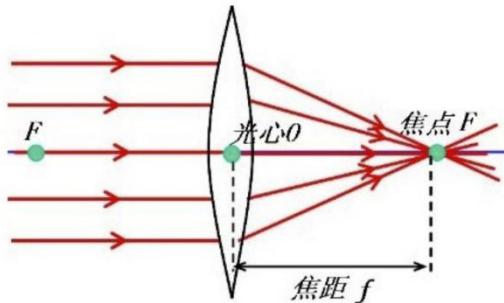
光流(Optical flow, optic flow)是关于视域中的物体运动检测中的概念。用来描述相对于观察者的运动所造成的观测目标、表面或边缘的运动。光流法可用于运动检测、物件切割、碰撞时间与物体膨胀的计算、运动补偿编码，或者通过物体表面与边缘进行立体的测量等等。

### (十九) 相机标定 (Camera Calibration)

相机标定用于纠正相机镜头导致的畸变。

#### 1 · 光心、焦距、焦点

下图可使得光心 (optical center) 、焦距 (focal length) 和焦点 (focal point) 的概念一目了然：



## 2 · 坐标系

图像处理和立体视觉常说到的坐标系有四个，即世界坐标系、相机坐标系、图像坐标系、像素坐标系。

### (1) 世界坐标系

$O_w \sim X_w Y_w Z_w$ ，独立于相机的坐标系，可以用于描述相机位置，单位 m。世界坐标系中的点，即为现实世界中的点。构建世界坐标系只是为了更好的描述相机的位置。

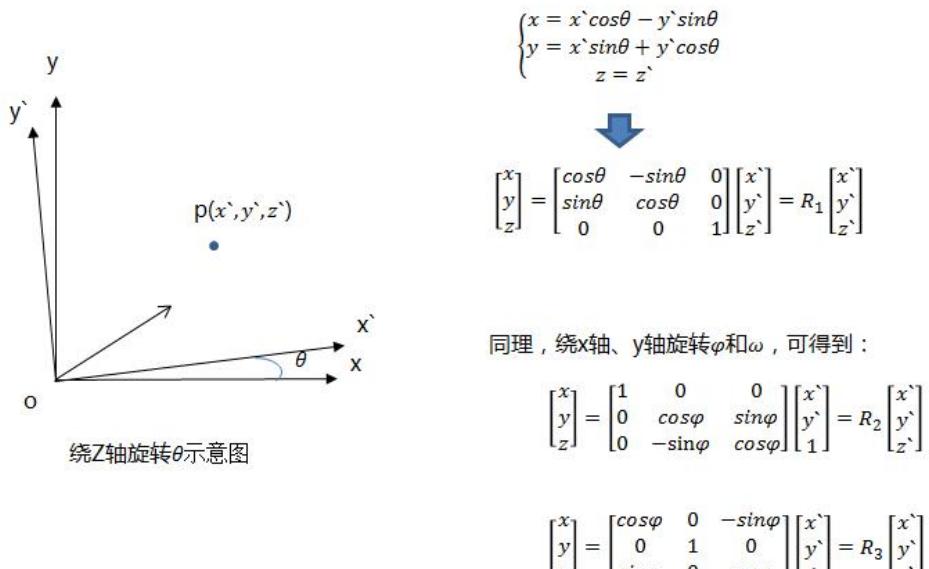
在双目视觉（两个摄像头）中一般将世界坐标系原点定在左相机或者右相机，或者两者 x 轴方向的中点。

对于单目视觉，世界坐标系可以跟相机坐标系重合。

## (2) 相机坐标系

$O_c X_c Y_c Z_c$ ，以光心为原点，单位 m。从世界坐标系转换到相机坐标系为刚体变换，仅涉及到旋转和平移，物体或空间不会发生形变。

以下为旋转矩阵：



于是可以得到旋转矩阵  $R = R_1 R_2 R_3$

<http://blog.csdn.net/chentravelling>

假设平移向量为 T，则点从世界坐标系  $O_w$  到相机坐标系  $O_c$  的变换为：

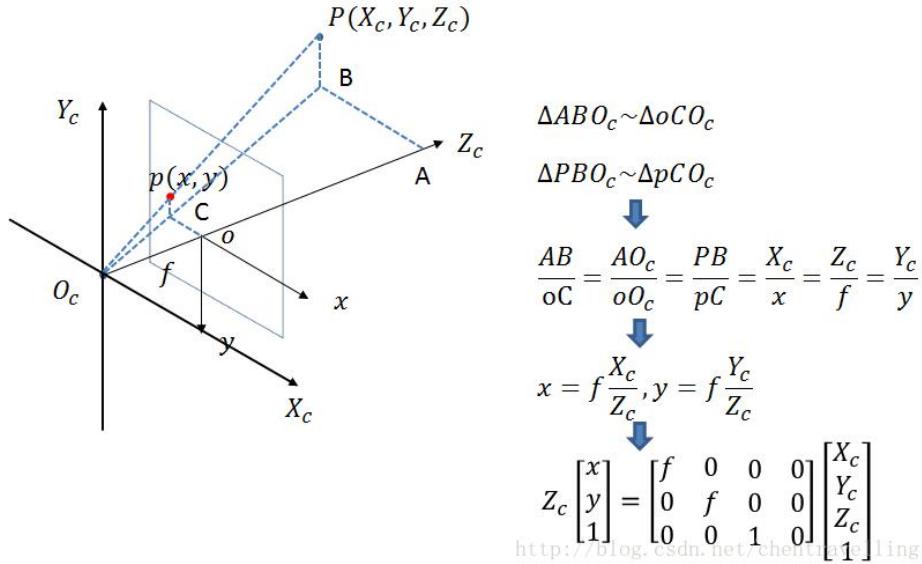
$$\begin{bmatrix} X_c \\ Y_c \\ Z_c \end{bmatrix} = R \begin{bmatrix} X_w \\ Y_w \\ Z_w \end{bmatrix} + T \quad \rightarrow \quad \begin{bmatrix} X_c \\ Y_c \\ Z_c \\ 1 \end{bmatrix} = \begin{bmatrix} R & T \\ \vec{0} & 1 \end{bmatrix} \begin{bmatrix} X_w \\ Y_w \\ Z_w \\ 1 \end{bmatrix}, R: 3 * 3, T: 3 * 1$$

<http://blog1.csdn.net/chentravel>

## (3) 图像坐标系

图像坐标系是二维坐标系，单位是 mm，从相机坐标系到图像坐标系，是透视投影关系。

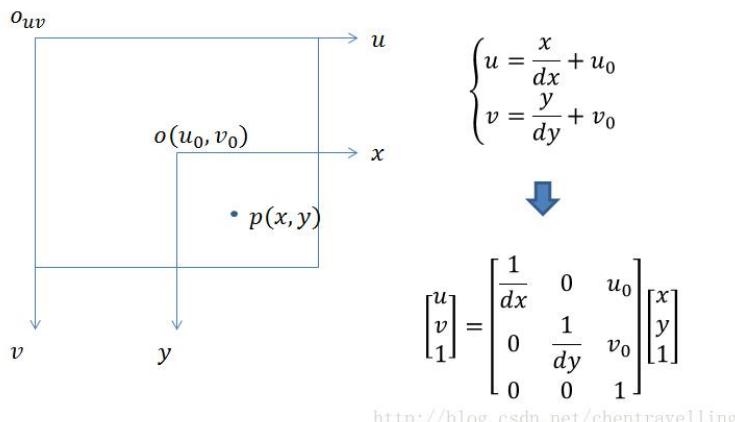
以下是相机坐标系中点的位置  $P(X_c, Y_c, Z_c)$ ，到图像坐标系的位置  $p(x, y)$  的变换：



#### (4) 像素坐标系

像素坐标系和图像坐标系在同一平面，且没有旋转，只是各自的原点和度量单位不一样。图像坐标系的原点为相机光轴与成像平面的交点，通常情况下是成像平面的中点 (principal point)。图像坐标系的单位是 mm，属于物理单位，而像素坐标系的单位是 pixel，我们平常描述一个像素点都是几行几列。

图像坐标系到像素坐标系的转换如下：



## (5) 内参和外参

通过以上四个坐标系的转换可以得到一个点从世界坐标系如何转换到像素坐标系：

$$Z_c \begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = \begin{bmatrix} \frac{1}{dx} & 0 & u_0 \\ 0 & \frac{1}{dy} & v_0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} f & 0 & 0 & 0 \\ 0 & f & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} R & T \\ \vec{0} & 1 \end{bmatrix} \begin{bmatrix} X_w \\ Y_w \\ Z_w \\ 1 \end{bmatrix} = \begin{bmatrix} f_x & 0 & u_0 & 0 \\ 0 & f_y & v_0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \underbrace{\begin{bmatrix} R & T \\ \vec{0} & 1 \end{bmatrix}}_{\text{相机内参}} \underbrace{\begin{bmatrix} X_w \\ Y_w \\ Z_w \\ 1 \end{bmatrix}}_{\text{相机外参}}$$

其中从世界坐标系到相机坐标系的转换矩阵即相机的外参（extrinsic property），而从相机坐标系到像素坐标系的3个转换矩阵被合为一个矩阵，即相机的内参（intrinsic property），也称相机矩阵（camera matrix）。

参考：

<http://blog.csdn.net/chentravelling/article/details/53558096>

参考：

[https://docs.opencv.org/master/dc/dbb/tutorial\\_py\\_calibration.html](https://docs.opencv.org/master/dc/dbb/tutorial_py_calibration.html)

## 3 · 畸变 (distortion)

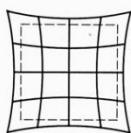
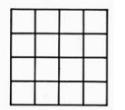
畸变分为两种，径向畸变（radial distortion，或者叫径向失真）和切向畸变（tangential distortion，也叫切向失真）。

### (1) 径向畸变

径向畸变使得直线看上去弯曲，离图像中心越远，径向失真越明显。下图中棋盘的线都应该是直线，然而和标示出来的红色直线相比，棋盘上的线都仿佛凸出来了。



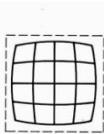
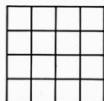
径向畸变的原因主要是镜头的径向曲率造成的（光在远离透镜中心的地方比靠近中心的地方更加弯曲）。导致真实成像点向外或者向内偏离理想成像点。其中如果径向畸变的像点相较于理想像点向外偏移，远离中心的，称为枕形畸变（Pincushion distortion）：



长焦距

[https://blog.csdn.net/qq\\_40369926](https://blog.csdn.net/qq_40369926)

如果径向畸变的像点相对于理想像点沿径向向中心靠拢，称为桶状畸变（Barrel Distortion）：



中短焦距

[https://blog.csdn.net/qq\\_40369926](https://blog.csdn.net/qq_40369926)

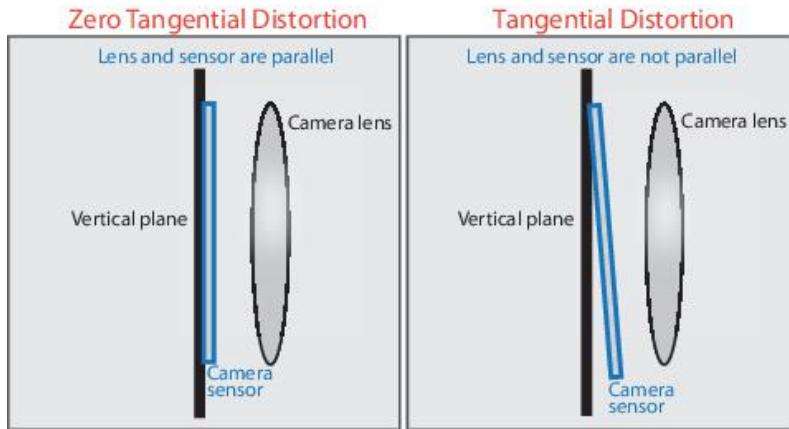
径向畸变可以用如下公式表示：

$$x_{\text{distorted}} = x(1 + k_1 r^2 + k_2 r^4 + k_3 r^6)$$

$$y_{\text{distorted}} = y(1 + k_1 r^2 + k_2 r^4 + k_3 r^6)$$

## (2) 切向畸变

切向畸变发生的原因是因为镜头并非完全与成像平面（imaging plane）平行，因此图像中的某些部分，会显得比实际更加靠近：



镜头和成像平面通常都平行，切向失真相对于径向失真影响小很多。切向失真的公式如下：

$$x_{\text{distorted}} = x + [2p_1xy + p_2(r^2 + 2x^2)]$$

$$y_{\text{distorted}} = y + [p_1(r^2 + 2y^2) + 2p_2xy]$$

### (3) 畸变系数

考虑到径向畸变和切向畸变，我们需要五个参数来表示畸变，通常被称为畸变系数 (distort coefficients)：

$$\text{Distortion coefficients} = (k_1 \ k_2 \ p_1 \ p_2 \ k_3)$$

$k_3$  在  $p_1$  和  $p_2$  的后面是因为有时候会不使用  $k_3$ ，这样会损失些微的精确性。事实上如果要更精确，径向畸变还可以有  $k_4$ 、 $k_5$  等系数。

## 4 · 相机标定

相机标定 (Camera calibration) 有 2 个目的：

- 获得相机的内参矩阵
- 获得相机的畸变系数，以便校正图片

### (1) 张正友标定法

张正友标定法利用棋盘格标定板，在得到一张标定板的图像之后，可以利用相应的图像检测算法得到每一个角点的像素坐标  $(u, v)$ 。

张正友标定法将世界坐标系固定于棋盘格上，则棋盘格上任一点的物理坐标  $(u, v)$ ，由于标定板的世界坐标系是人为事先定义好的，标定板上每一个格子的大小是已知的，我们可以计算得到每一个角点在世界坐标系下的物理坐标  $(U, V, W=0)$ 。

于是我们有了以下这些信息：

- 每一个角点的像素坐标  $(u, v)$
- 每一个角点的世界坐标  $(U, V, W=0)$

来进行相机的标定，获得相机的内外参矩阵、畸变参数。

参考：

<https://zhuanlan.zhihu.com/p/94244568>

## (2) OpenCV calibrate

opencv 的源码中，跟相机标定相关的有：

- samples/python/calibrate.py：校正程序
- samples/data/chessboard.png：棋盘图片，用于打印出来拍摄
- samples/data/left???.jpg：样例图片，calibrate.py 会缺省使用这些图片

calibrate.py 的格式如下：

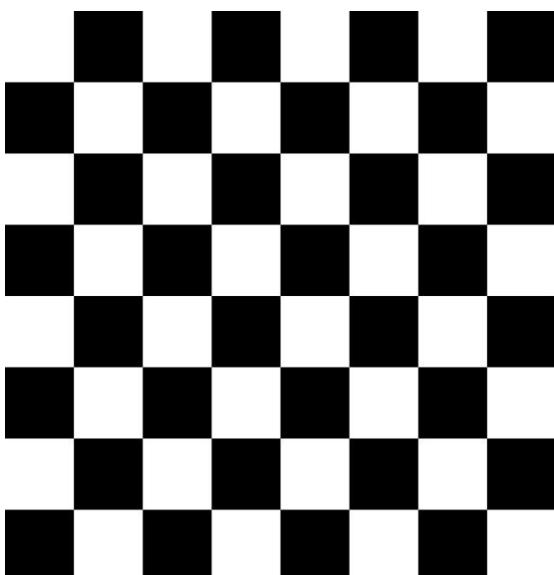
usage:

```
calibrate.py [--debug <output path>] [--square_size] [<image mask>]
```

default values:

```
--debug: ./output/
--square_size: 1.0
<image mask> defaults to ../data/left*.jpg
```

chessboard.jpg 为  $7 \times 7$  的点（黑白格子的交点，calibrate.py 中为  $9 \times 6$ ）



具体的操作步骤：

1. 将 chessboard.jpg 打印在纸上
2. 用需要标定的相机拍摄 10 张以上棋盘纸的图片（参考 left???.jpg）
3. 运行 calibrate.py 得到该相机畸变系数和相机矩阵

## 5 · OpenCV Python

相机标定相关的 OpenCV Python 函数

- findChessboardCorners()：在输入图片中寻找给定长宽的棋盘格，如果成功返回非 0，失败则返回 0。
  - image：输入的带有棋盘格的图片，8bit 的黑白或彩色图片
  - patternSize：棋盘格角点的长宽数量，格式为(columns, rows)
  - corners：可选参数，输出的探测到的角（同返回值中的 corners，C++用）
  - flags：可选参数，标志位
    - ◆ CALIB\_CB\_ADAPTIVE\_THRESH：该函数的默认方式是根据图像的平均亮度值进行图像二值化，设立此标志位的含义是采用变化的阈值进行自适应二值化
    - ◆ CALIB\_CB\_NORMALIZE\_IMAGE：在二值化之前，调用 EqualizeHist() 函数进行图像归一化处理
    - ◆ CALIB\_CB\_FILTER\_QUADS：二值化完成后，函数开始定位图像中的四边形（这里不应该称之为正方形，因为存在畸变），这个标志设立后，函数开始使用面积、周长等参数来筛选方块，从而使得角点检测更准确更严格
    - ◆ CALIB\_CB\_FAST\_CHECK：快速检测选项，对于检测角点极可能不成功检测的情况，这个标志位可以使函数效率提升
  - 返回：
    - ◆ retval：成功则为非 0，失败则为 0
    - ◆ corners：（不精确的）角点坐标，需调用 cornerSubPix() 进一步优化
- cornerSubPix()：进一步改善角点坐标的精度，检测亚像素级角点
  - image：棋盘格图片，8 位单通道图片
  - corners：角点坐标，既是输入（来自 findChessboardCorner()）也是输出
  - winSize：求亚像素角点的窗口大小，格式为(d,d)，通常为(5,5) (d 为半径，窗口大小为(2d+1) \* (2d+1))
  - zeroZone：设置的“零区域”，在搜索窗口内，设置的“零区域”内的值不会被累加，权重值为 0。通常为(-1,-1)，表示没有这样的区域
  - criteria：条件阈值，包括迭代次数阈值和误差精度阈值，一旦其中一项条件满足设置的阈值，则停止迭代，获得亚像素角点。通常为(cv.TERM\_CRITERIA\_EPS + cv.TERM\_CRITERIA\_COUNT, 30, 0.1)。
- drawChessboardCorners()：将角点在棋盘格图片中标示出来
  - image：棋盘格图片，8 位彩色图片
  - patternSize：棋盘格角点的长宽
  - corners：找到的角点（来自 findChessboardConers() 和 CornerSubPix()）
  - patternWasFound：来自 findChessboardCorners() 的 retval
- calibrateCamera()：通过点的世界坐标和像素坐标，标定摄像头
  - objectPoints：点的世界坐标，

- imagePoints : 点的像素坐标，比如 `findChessboardCorners()` 得到
- imageSize :
- cameraMatrix : 同返回的相机矩阵，通常为 `None`
- distCoeffs : 同返回的畸变系数，通常设为 `None`
- 返回：
  - ◆ `retval` :
  - ◆ `cameraMatrix` : 返回的相机矩阵（内参）
  - ◆ `distCoeffs` : 返回的畸变系数
  - ◆ `rvecs` : 外参中的旋转矩阵
  - ◆ `tvecs` : 外参中的平移矩阵
- `getOptimalNewCameraMatrix()` : 根据 `free scaling` 参数，计算新的相机矩阵
  - `cameraMatrix` : 相机矩阵
  - `distCoeffs` : 畸变系数
  - `imageSize` : 原始的图片大小（单位是像素）
  - `alpha` : `free scaling` 系数，介于 0 到 1 之间
    - ◆ 1 : 表示原始图片中所有像素都在校正后的新图片中得到体现（此时新图片是一个扭曲的形状，矩形的其余部分被全黑像素填满）
    - ◆ 0 : 表示校正之后的新图片中所有像素都是有效的（即所有超出有效矩形的部分都被裁剪）
  - `newImageSize` : 可选参数，校正后的图片大小（单位是像素），缺省为 `imageSize`
  - `centerPrinciplePoint` : 可选参数，标志位，决定原点是否在图像中心。缺省根据 `alpha` 自动调整到最适合的值
  - 返回：
    - ◆ `newCameraMatrix` : 新的相机矩阵
    - ◆ `validPixROI` : 有效像素的范围
- `undistort()` : 校正图像
  - `src` : 输入图片
  - `cameraMatrix` : 相机矩阵
  - `distCoeffs` : 畸变系数
  - `dst` : 可选参数，同返回的 `dst` ,
  - `newCameraMatrix` : 可选参数，新的相机矩阵，缺省同 `cameraMatrix`
  - 返回：
    - ◆ `dst` : 输出图片

# 六、网络构成

介绍了构成神经网络的各种结构、微结构、激活函数、损失函数、优化方法、标准化方法等。

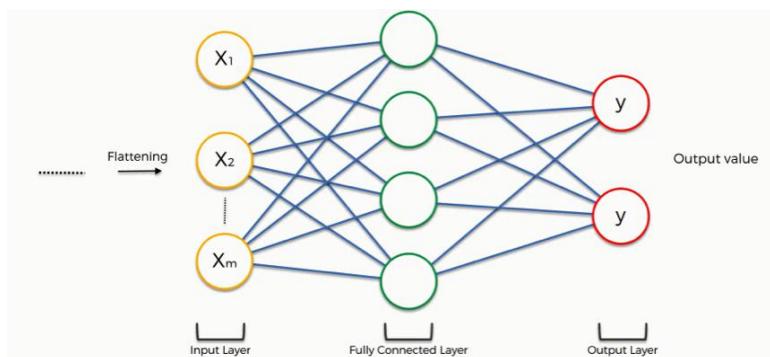
## (一) DNN 及 CNN 微结构

描述了 DNN 和 CNN 的微结构，包括单层的结构（比如全连接层和卷积层），或者若干相邻层组成的组合结构（block，比如 Depthwise Separable 卷积），或者某种构建层结构的思想（比如 3\*3 卷积）。

在后期的基础 CNN 网络构建中，同类 block 的堆叠是一个常用的构建方式。

### 1 · 全连接层 (Dense)

全连接层 (Full-Connection Layer)，又名 FC 层，或者 Dense 层。相邻两层的所有神经元都有连接。



全连接层的参数数量 W (权重数量) ，B (偏置数量) ，P (参数总数) 分别是：

$$W = I * O$$

$$B = O$$

$$P = W+B = (I+1) * O$$

其中 I 为输入神经元的数量，O 为输出神经元的数量。如果输入来自二维卷积层，则

$I = C * H * W$ ，C、H、W 分别为卷积层输出特征图的通道数、长、宽。相对于其他层而言，FC 层的参数数量是最多的。

全连接层的 FLOPs 为：

$$FLOPs = I * O$$

即对每个输出神经元，都要做 I 次权重乘法，I -1 次权重累加，1 次偏置累加。

## 2 · 池化层 (Pooling)

池化层 (Pooling Layer) , 进行缩小长和高方向上的空间的运算的层，池化层不改变通道数量。通常是 Max 池化，也有 Average 池化等。

池化层只有定义池化过滤器大小、padding 大小、stride 大小的超参数，没有可学习的参数。

池化层的计算量为：

$$\text{FLOPs} = H * W * FH * FW * CO$$

H 和 W 为输出特征图的长和宽。

## 3 · 全局池化层

全局池化层 (Global Pooling Layer) 是特化的池化层，全局池化层的过滤器大小跟输入特征图的大小完全一样，整个特征图经过池化之后变为一个标量（加上通道就是一个向量）。全局池化层分为 Global Average Pooling 和 Global Max Pooling。

全局池化层的计算量套用池化层的计算量公式得出：

$$\text{FLOPs} = FH * FW * CO$$

FH 和 FW 为过滤器的大小，也是输入特征图的大小

## 4 · 反池化层

反池化层 (Unpooling Layer) 是上采样的一种方式，通常是 Max 反池化。

池化操作是不可逆的，通过 max 池化时记录下 max 池化时最大值的位置，在反池化时将最大值放回该位置，并将其余部分填 0，反池化操作可以近似实现池化逆操作。

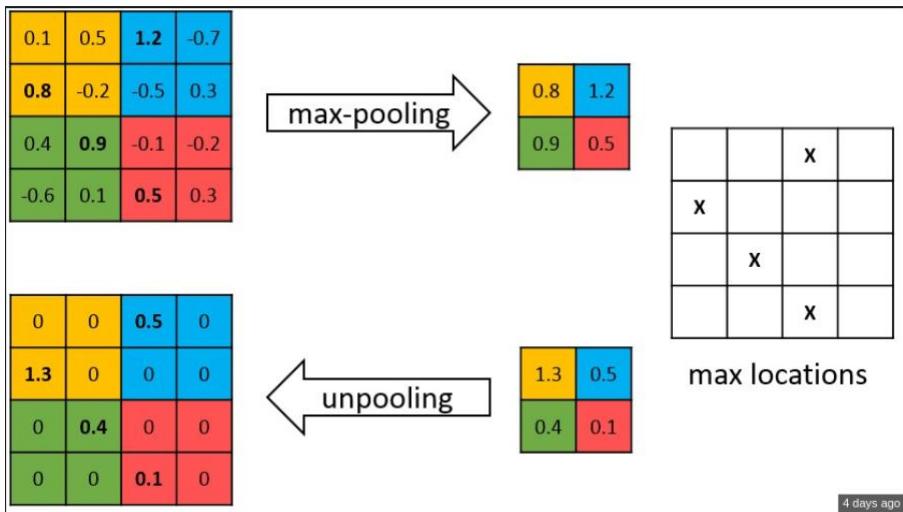


Figure : Max Unpooling

## 5 · 常规卷积层

卷积层（Convolution Layer），进行卷积运算的层，由卷积核组成。CNN 的重要组成部分。

2 维卷积层的参数数量 W（权重数量），B（偏置数量），P（参数总数）分别为：

$$W = FH * FW * CI * CO$$

$$B = CO$$

$$P = W + B = (FH * FW * CI + 1) * CO$$

其中 FH 和 FW 为卷积核（过滤器）的长和宽（通常是一致的，都写作 K），CI 为输入的通道数（输入特征图数），CO 为输出的通道数（输出特征图数）

2 维卷积层的计算量为：

$$\text{FLOPs} = FH * FW * CI * H * W * CO$$

其中卷积乘法操作数量是  $FH * FW * H * W * CI * CO$

每个输入通道内的卷积累加操作数量是  $(FH * FW - 1) * H * W * CI * CO$

输入通道间的累加操作数量是  $(CI - 1) * H * W * CO$

偏置加法操作数量是  $H * W * CO$ 。

这个计算量中：

- $FH * FW$  表示对单个输入特征图的单个卷积核的一次卷积（不是一整张输入特征图）
- 乘以  $CI$  为输出特征图中每个像素的计算量
- 乘以  $H * W$  表示每张输出特征图的计算量
- 再乘以  $CO$ （输出特征图数量）表示所有特征图的计算量

$H$  和  $W$  分别为输出特征图的长和宽，如果 Stride 为 1，则输入和输出特征图大小一样，否则输入特征图为输出特征图的  $S * S$  倍。

## 6 · 本地卷积层

本地卷积层（Locally-Convolutional Layer），类似卷积层，区别在于卷积核不共享，也就是说对矩阵（图像）的每个局部块进行卷积运算的时候，都使用不同的卷积核。

因为参数不共享，本地卷积层的参数数量会远远大于卷积层，各个参数  $W$ （权重数量）， $B$ （偏置数量）， $P$ （参数总数）分别为：

$$W = FH * FW * CI * CO * H * W$$

$$B = CO$$

$$P = W + B = FH * FW * CI * CO * H * W + CO$$

各字母意思参考卷积层。

本地卷积层的计算量和普通的卷积层一致。

## 7 · Dropout 层

Dropout 用于在训练的时候随机的关闭一定比率的神经元，其效果是免于过拟合，这个比率是个超参数，称为 **dropout ratio**。Dropout 只在训练中出现，实际推理的网络的神经元会全部打开。

Dropout 首次提出是在 2012 年的 AlexNet 中。

参考：

12 种主要的 Dropout 方法：如何应用于 DNNs，CNNs，RNNs 中的数学和可视化解释

<https://zhuanlan.zhihu.com/p/146747803>

## 8 · Deconv (ZFNet)

反卷积（Deconvolution，Deconv）的主要目的是为了将 feature map+卷积核可视化，反卷积可能用在三个地方：

- CNN 可视化，比如在 ZFNet 论文中用到的可视化方法，用于分析理解网络
- 图片重建，比如用 GAN 生成图片
- 上采样（upsampling）

反卷积的具体实现请参考 [ZFNet](#)

## 9 · Group Conv (AlexNet)

**Group convolution**, 分组卷积, 或群卷积。通常的卷积过程是：输入特征图尺寸为  $H \times W \times CI$ ，经过  $CO$  个卷积核（尺寸为  $FH \times FW \times CI \times CO$ ），输出特征图尺寸为  $H \times W \times CO$ 。而组卷积则是将输入通道分成若干组  $G$ ，每组  $CI/G$  个通道，每个组内有  $CO/G$  个卷积核，尺寸为  $FH \times FW \times (CI/G) \times (CO/G)$ ，每组输出  $CO/G$  个特征图。这样总的输入输出不变，

分组卷积可以减少参数及计算量，假设分为  $G$  组，采用组卷积之前的参数和计算量为：

Params =  $(FH \times FW \times CI + 1) \times CO$

FLOPs =  $FH \times FW \times CI \times H \times W \times CO$

而分成  $G$  组之后的参数和计算量为：

Params =  $(FH \times FW \times CI/G + 1) \times CO/G \times G$

FLOPs =  $FH \times FW \times (CI/G) \times H \times W \times (CO/G) \times G = FH \times FW \times CI \times H \times W \times CO/G$

可以看到参数和计算量都变为原来的  $1/G$

分组卷积最早的出现应该是 AlexNet 中，受限于硬件条件，前面的卷积层被分成了两组，分别对应两块 GPU，减少训练时间，加快训练速度。

DW 卷积 (Depthwise Conv) 是分组卷积的一个特殊情况，每个通道独立卷积。

分组卷积在后来的 ResNeXt、ShuffleNet 等也有出现，其目的也是为了减小计算量，类似于一种介于常规卷积和 DW 卷积之间的中间状态。

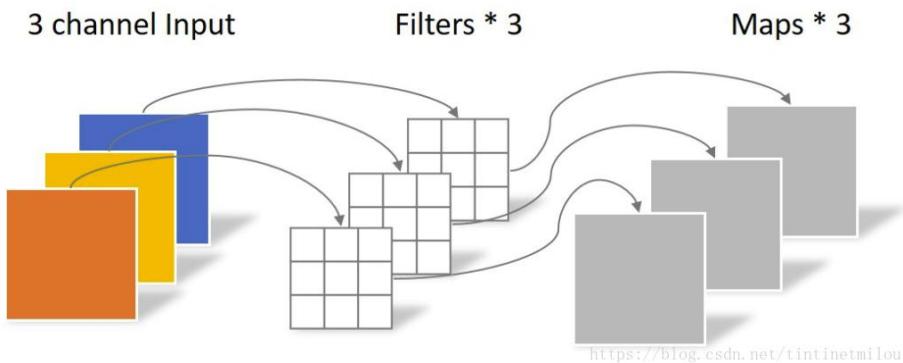
## 10 · Depthwise Separable 卷积 (Xception)

**Depthwise separable 卷积** (深度可分离卷积) 是为了减少参数及计算量，并且能达到常规卷积的效果而出现的。Depthwise Separable 卷积将一个常规的卷积操作分成两层完成，前一层叫 depthwise convolution (DW 卷积)，后一层叫 pointwise convolution (PW 卷积)。

深度可分离卷积在主流 CNN 里首次出现是 Xception 和 MobileNet V1，这两者都是 Google 的，推出时间也差不多。

### (1) Depthwise 卷积层

Depthwise 卷积层，又叫 DW 卷积层，逐层卷积。不同于常规卷积操作中每个卷积核是同时操作所有输入通道，Depthwise Convolution 的每个卷积核只负责一个输入通道，一个通道只被一个卷积核卷积。因此 DW 卷积的卷积核数、输出通道数  $CO$  和输入通道数  $CI$  是一样（如果有 Same padding 则输出特征图的维度和输入特征图完全一致）。



Depthwise Convolution 完成后的 Feature map 数量与输入层的通道数相同，无法扩展 Feature map。而且这种运算对输入层的每个通道独立进行卷积运算，没有有效的利用不同通道在相同空间位置上的 feature 信息。因此需要 Pointwise Convolution 来将这些 Feature map 进行组合生成新的 Feature map。

Depthwise Convolution 算是分组卷积的一种极端情况。

- DW 卷积层的参数数量为：

$$W = FH * FW * CO$$

$$B = CO$$

$$P = W + B = (FH+FW+1) * CO$$

- DW 卷积层的计算量为：

$$FLOPs = FH * FW * H * W * CO$$

其中卷积乘法操作数量： $FH * FW * H * W * CO$

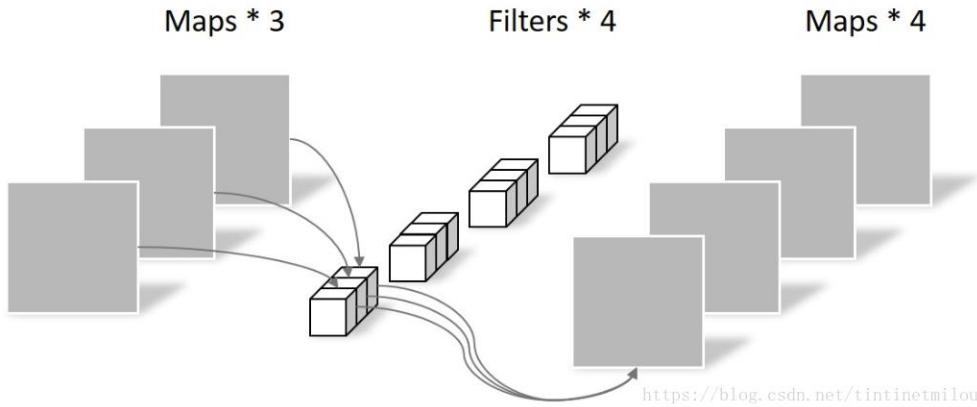
通道内卷积累加操作数量： $(FH * FW - 1) * H * W * CO$

通道间累加操作：无

偏置累加操作数量： $H * W * CO$

## (2) Pointwise 卷积层

Pointwise 卷积层，又叫 PW 卷积层，逐点卷积。Pointwise 卷积就是一个卷积核为  $1 * 1 * CI * CO$  的普通卷积层，所以这里的卷积运算会将上一步的 map 在深度方向上进行加权组合，生成新的 Feature map。



PW 卷积层的参数和计算量参考常规卷积层为：

$$\text{FLOPs} = \text{CI} * \text{CO} * \text{H} * \text{W}$$

### (3) 计算量

根据上面的公式，来计算下能省下多少计算力

假设输入大小为  $56*56*128$ ，输出为也为  $56*56*128$ ，卷积核大小为  $3 * 3$ ，如果用常规卷积，则计算量为：

$$\text{FH} * \text{FW} * \text{CI} * \text{CO} * \text{H} * \text{W} = 3*3*128*128*56*56 = 462,422,016。$$

而如果使用 depthwise separable 卷积（事实上这就是 Mobilenet 的一部分），计算量为：

$$\text{DW_FLOPs} = \text{FH} * \text{FW} * \text{H} * \text{W} * \text{CO} = 2*3*3*56*56*128 = 3,627,672$$

$$\text{PW_FLOPs} = \text{FH} * \text{FW} * \text{H} * \text{W} * \text{CI} * \text{CO} = 1*1*56*56*128*128 = 51,380,224$$

$$\text{FLOPs} = \text{DW_FLOPs} + \text{PW_FLOPs} = 7,255,344 + 102,760,448 = 55,007,896$$

可以看到这里的 Depthwise separable 卷积相对于常规卷积层，计算量只有其的  $1/8$  不到。

## 11 · 3\*3 卷积核 (VGGNet)

在早期的 CNN 中，可以看到一些大卷积核 ( $11*11$ ,  $7*7$ ,  $5*5$ )，但在 2014 年 VGG 的论文中发现，两次  $3*3$  卷积的感受野和一个  $5*5$  卷积一样，三次  $3*3$  卷积的感受野和一次  $7*7$  卷积一样，但是计算量能减少很多。

假设输入特征图为  $H*W$ ，输入通道  $CI$ ，输出通道  $CO$ ，则计算量 FLOPs 如下：

一层  $5*5$  的 FLOPs= $5*5*H*W*CI*CO = 25K$

两层  $3*3$  的 FLOPs= $3*3*H*W*CI*CO * 2 = 18K$

一层  $7*7$  的 FLOPs= $7*7*H*W*CI*CO = 49K$

三层  $3 \times 3$  的 FLOPs =  $3 \times 3 \times H \times W \times CI \times CO \times 3 = 27K$

可以看到计算量下降了不少，因此自 VGG 之后，CNN 大都使用多  $3 \times 3$  卷积来替代大尺度的卷积核。

## 12 · $1 \times 1$ 卷积核

$N \times N$  卷积的目的一般是用来整合相邻区域内的信息，而  $1 \times 1$  的卷积核并不能实现这个目的，它的作用有 2 个：

- 改变维度： $1 \times 1$  卷积核可以用于降维或者升维，比如一个  $H \times W \times CI$  的输入特征图，经过  $CO$  个  $1 \times 1 \times CI$  个卷积核之后，形状变为  $H \times W \times CO$ 。
- 跨通道的交互：类似于一个不同通道间同一位置的全连接，在 ResNet、FCN 等中出现，用于替代 FC 层。

$1 \times 1$  卷积就是 PW 卷积（Pointwise Conv），本质上就是一个常规的卷积层，只不过其卷积核大小为  $1 \times 1$ 。计算量为：

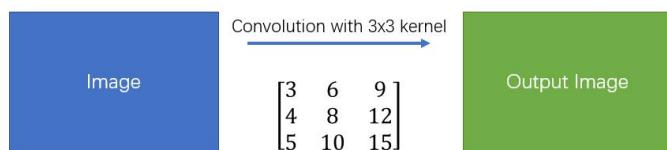
$$\text{FLOPs} = CI * CO * H * W$$

## 13 · Spatial Separable 卷积

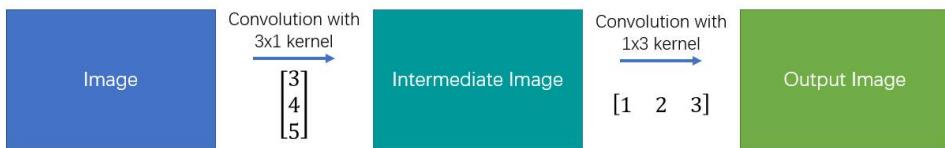
空间可分离卷积（Spatial Separable Convolution，SS 卷积）也是一种将一个常规卷积层分解为 2 个卷积层，以减少计算量的方式。但是空间可分离卷积有其局限性，因此在深度学习中使用的不多。

SS 卷积是将常规的  $N \times N$  卷积核在长和宽上分解为两个  $N \times 1$  和  $1 \times N$  的卷积核，分别作为一个单独的卷积层。

### Simple Convolution



### Spatial Separable Convolution



SS 卷积和深度可分离卷积类似，都是把一个常规卷积层拆分成两个，区别在于拆分的维度不一样。常规卷积层中，每个输出特征图的每个像素，需要在三个维度上卷积（即宽 W、高 H、输入通道 CI）。

- 深度可分离卷积将这三个维度拆分为宽和高先卷积，再在通道上融合。
- SS 卷积则是先在宽和通道上卷积，再在高和通道上卷积

至于计算量的节省，可以简单的通过公式得出，我们以常用的 3\*3 卷积核为例：

原先的计算量 =  $3 \times 3 \times CI \times CO \times H \times W = 9 \times H \times W \times CI \times CO$

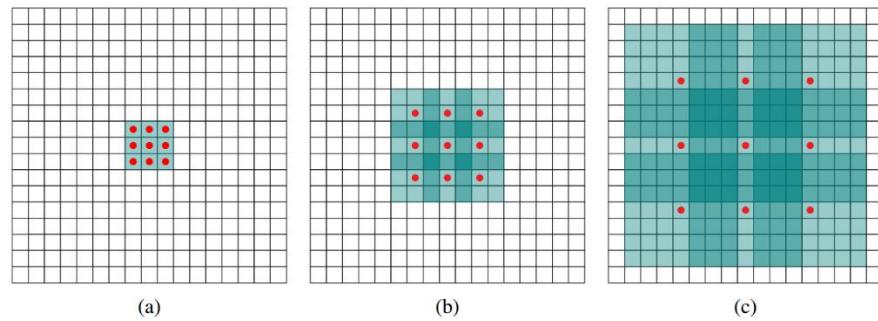
SS 卷积的计算量 =  $1 \times 3 \times CI \times CO \times H \times W + 3 \times 1 \times CI \times CO \times H \times W = 6 \times H \times W \times CI \times CO$

可以看见，计算量的节省并不算非常大。

SS 卷积最大的问题，就是并非所有的卷积核矩阵都可以表示为 1\*N 和 N\*1 的矩阵的点积。

## 14 · 带孔卷积

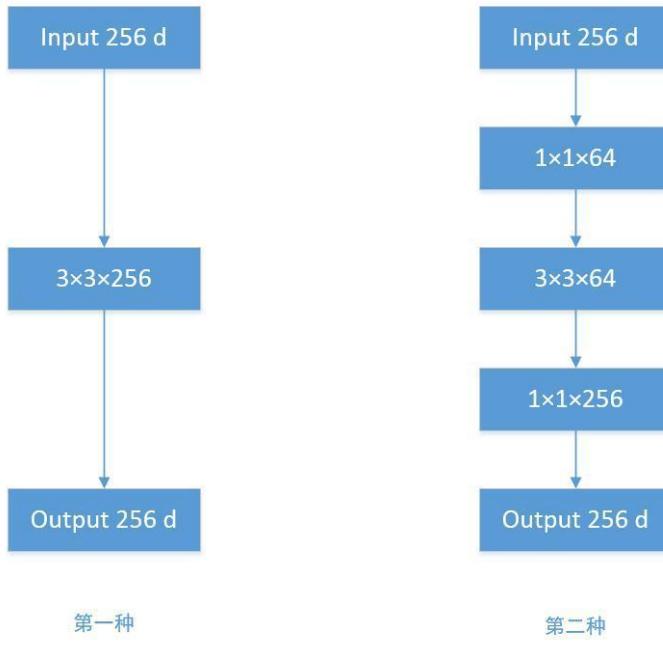
Dilated Convolution，又叫空洞卷积、带孔卷积、扩展卷积。带孔卷积的目的是为了在不增加参数和计算量的情况下扩大感受野。其原理是在卷积的时候，跳着卷积特征图。



上图中红点为卷积核卷积的位置，a 为常规卷积，b 为间隔为 1 的带孔卷积，c 为间隔为 3 的带孔卷积。

## 15 · Bottleneck 结构

Bottleneck 结构也是 CNN 中减小计算量的方式，Bottleneck 通常分成 3 个卷积层：先使用 1\*1 的卷积核（即 PW 卷积）对数据进行降维（减少通道），再进行常规的卷积，最后再用 1\*1 的卷积核进行升维。所谓瓶颈，指的就是中间层的通道数小于前后层。



上面两个卷积过程的输入和输出都是  $H \times W \times 256$ ，左边常规卷积的计算量为：

$$\text{FLOPs} = 3 \times 3 \times 256 \times 256 \times H \times W = 589,824 \times H \times W$$

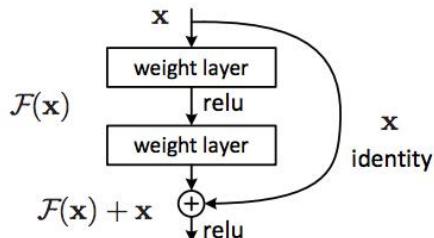
右边 Bottleneck 结构的计算量为：

$$\text{FLOPs} = 1 \times 1 \times 256 \times 64 \times H \times W + 3 \times 3 \times 64 \times 64 \times H \times W + 1 \times 1 \times 64 \times 256 \times H \times W = 69,632 \times H \times W$$

可以看到 Bottleneck 的计算量相较于常规卷积层下了一个数量级

## 16 · Residual Block (ResNet)

**残差结构 (Residual Block)** 于 2015 年由微软的何凯明等人在 ResNet 中提出，其特点是在网络中增加了直连通道（被称作 shortcut 或 skip connection），



传统的卷积网络或者全连接网络在信息传递的时候会存在信息丢失，损耗等问题，同时还有导致梯度消失或者梯度爆炸，导致很深的网络无法训练。残差结构在一定程度上解决了这个问题，通过直接将输入信息绕道传到输出，保护信息的完整性，整个网络只需要学习输入、输出差别的那一部分，简化学习目标和难度。

## (1) Residual bottleneck (ResNet)

在层数较深的实现中，ResNet 使用了一种叫做 **Residual bottleneck** 的结构：

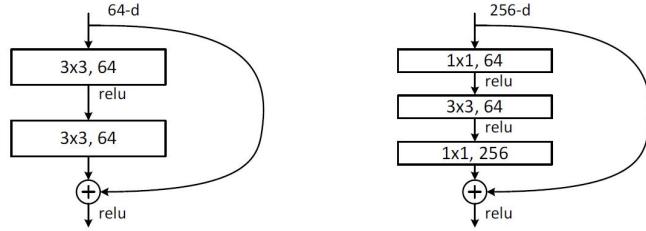


Figure 5. A deeper residual function  $\mathcal{F}$  for ImageNet. Left: a building block (on  $56 \times 56$  feature maps) as in Fig. 3 for ResNet-34. Right: a “bottleneck” building block for ResNet-50/101/152.

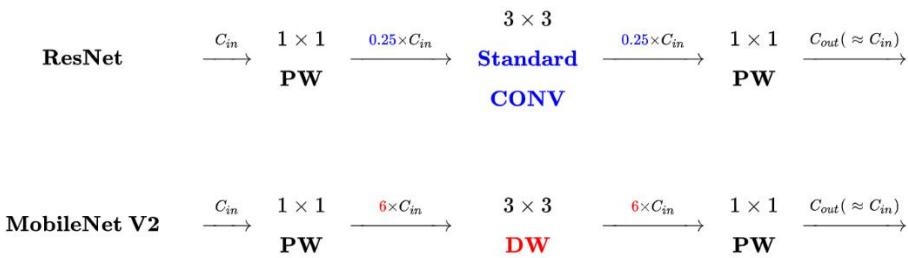
图中左侧是浅层 ResNet 所用的 **Residual block**，而右侧便是 **residual bottleneck**，先通过 PW 卷积降维，接着是一个  $3 \times 3$  的常规卷积，最后又用一个 PW 卷积升维，而 shortcut 连接了这三层的前后。

## 17 · Inverted Residual Block (MobileNet V2)

**Inverted Residual Block** 结构（反残差，倒残差）首次出现于 MobileNetV2 中。在层数比较深的 ResNet 中，会使用 Residual bottleneck 结构，Inverted residual block 结构与之类似，但有 2 点区别：

- Residual bottleneck 中间的  $3 \times 3$  卷积用的是常规卷积，而 Inverted residual block 中间的  $3 \times 3$  卷积用的是 DW 卷积
- Residual bottleneck 前面的  $1 \times 1$  卷积（PW 卷积）作用是降维，而后面是升维（所以才叫“瓶颈”），而 Inverted residual block 正相反，前一个 PW 卷积升维，后一个降维。这么做的原因也和 DW 卷积有关：相较于常规卷积，DW 卷积大幅度降低了计算量，因此没有普通 bottleneck 的需求（降维卷积减少计算量），而在高维提取特征效果更好。

见下图，上面是 Residual bottleneck，下面是 Inverted Residual Block：

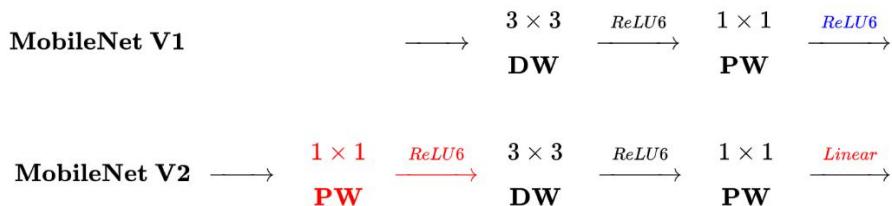


## 18 · Linear bottleneck (MobileNet V2)

Linear bottleneck 结构也是首次出现于 MobileNet V2 中，是一个基于 MobileNet V1 中的 Depthwise Separable 卷积做的改进，其区别在于：

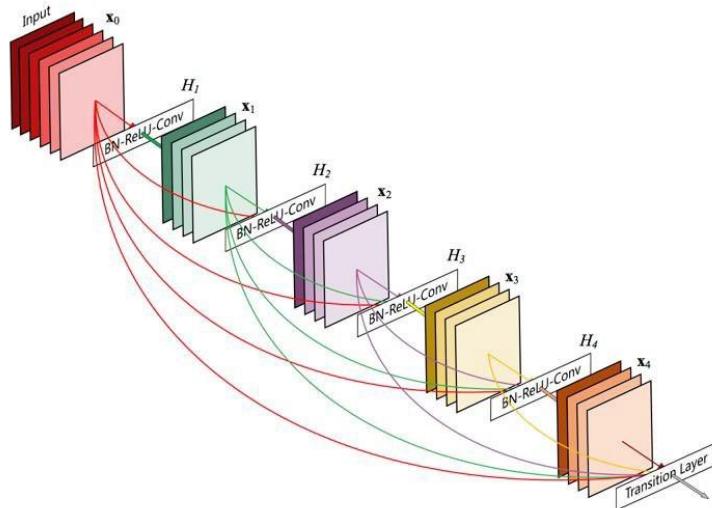
- Linear bottleneck 在 DW 卷积之前增加了一个 PW 卷积用于升维，其原因是 DW 卷积没法提升通道数，效果不够好。
- Linear bottleneck 去掉了第二个 PW 卷积的激活函数 ReLU6，作者认为这个激活函数在高维空间能够有效的增加非线性，但在低维空间会破坏特征。

见下图：



## 19 · Dense Block (DenseNet)

Dense Block 是 Residual Block 的极端化版本，首次出现在 DenseNet 中。在一个 Dense block 中的所有层之间都有连接，下图是一个 Dense block



写成伪代码如下：

```
def dense_block(x, f=32, d=5):
 l = x
 for i in range(d):
 x = conv(l, f)
 l = concatenate([l, x])
 return l
```

## 20 · Inception 结构 (GoogleNet)

Inception 结构首次出现于 2014 年谷歌推出的 GoogLeNet 中，其主要思想是并联，即若干不同大小的卷积层和池化层并联。Inception 是 CNN 历史上的一个里程碑，在此之前，CNN 只是更深的卷积层的简单堆叠。

Inception 的主要思想是，物体在不同图片中占的比例会完全不同，可能是占据了整幅图片的特写，也可能只占据画面一角。因此在同一层使用不同大小的卷积核，用于感受不同的视野上的特征。

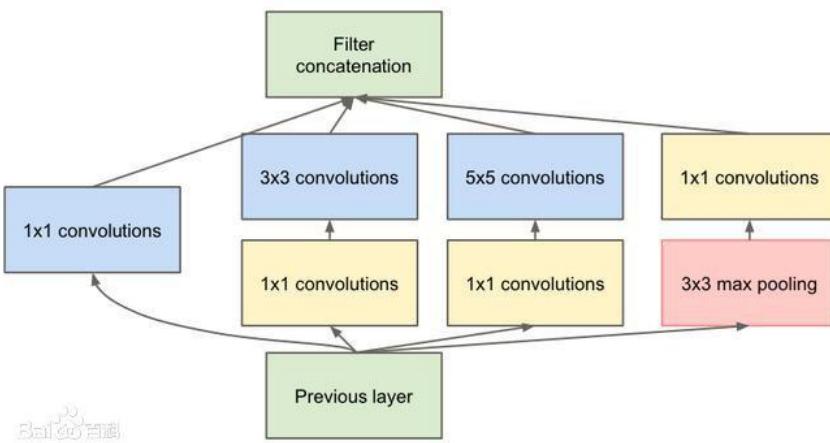
流行的 Inception 版本包括 V1, V2, V3, V4 和 Inception-ResNet。

参考：

<https://towardsdatascience.com/a-simple-guide-to-the-versions-of-the-inception-network-7fc52b863202>

### (1) Inception V1

V1 结构是最早的版本，将  $1 \times 1$ ,  $3 \times 3$ ,  $5 \times 5$  的卷积层和  $3 \times 3$  的池化层并联在一起，其中为了降低  $3 \times 3$  和  $5 \times 5$  卷积层的计算量，先用  $1 \times 1$  卷积进行了降维。



论文：

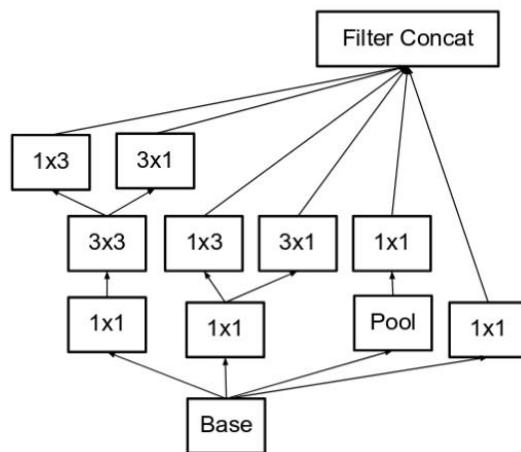
Going Deeper with Convolutions

<https://arxiv.org/pdf/1409.4842.pdf>

## (2) Inception V2 V3

Inception V2 主要做了以下改动：

- 将之前的  $5 \times 5$  卷积核拆成两层  $3 \times 3$  的卷积核，减少计算量（使用了 VGG 中出现的思想）
- 将之前的  $N \times N$  卷积核拆成  $N \times 1$  和  $1 \times N$  两层（即 Spatial Separable 卷积）
- 扩展了 Inception 结构中卷积结构的宽度，而非深度，以降低表现力瓶颈（Representational Bottleneck），如下图



以上三个改动分别被应用在 V2 的三个模型中

Inception V3 则有如下改动：

- 应用了 V2 中所有的三个改动
- 使用了 RMSProp 优化方法替代之前的 SGD
- 辅助分类器中增加了 BN 层
- Label Smoothing

论文：

V2 和 V3 用的是同一篇，Rethinking the Inception Architecture for Computer Vision  
<https://arxiv.org/pdf/1512.00567.pdf>

## (3) Inception V4 / Inception-ResNet

Inception V4 主要的改动是增加了残差网络（ResNet），具体请自行参考论文。

论文：

Inception-v4, Inception-ResNet and the Impact of Residual Connections on Learning

<https://arxiv.org/pdf/1602.07261>

（Inception V4 和 Inception-ResNet 也来自同一篇论文）

## 21 · ResNeXt 结构 (ResNeXt)

ResNeXt 结构是 Inception 和 ResNet 思想的结合，有点类似 Inception-ResNet，但区别在于各个 path 是一样的，下图左为 ResNet 结构，右为 ResNeXt 结构：

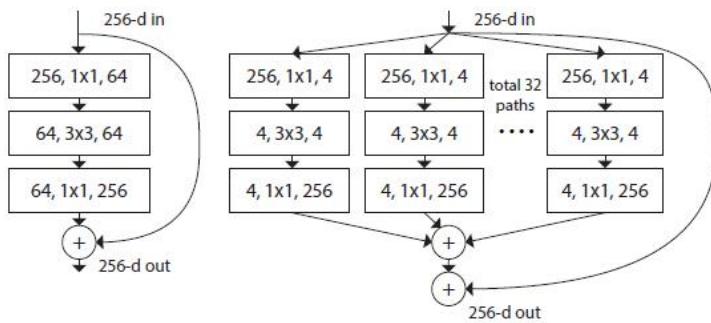


Figure 1. Left: A block of ResNet [14]. Right: A block of ResNeXt with cardinality = 32, with roughly the same complexity. A layer is shown as (# in channels, filter size, # out channels).

## 22 · Fire Module (SqueezeNet)

Fire module 是出现在 SqueezeNet 里的微结构，由两个部分组成，分别是：

- squeeze 层：由 1\*1 卷积组成，作用是降维
- expand 层：由 1\*1 卷积和 3\*3 卷积并联而成，类似 Inception 结构

Fire module 不改变特征图的尺寸，只改变其通道数。

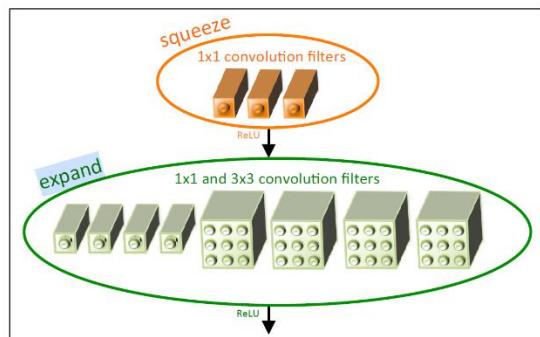
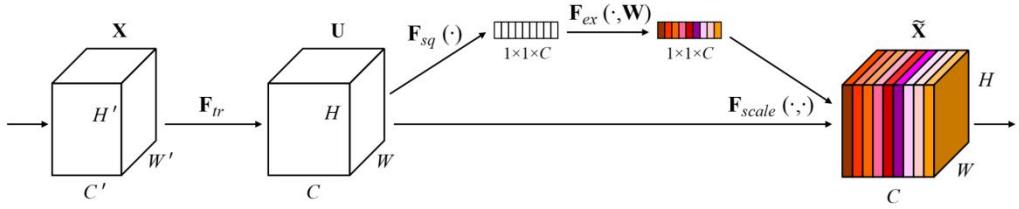


Figure 1: Microarchitectural view: Organization of convolution filters in the **Fire module**. In this example,  $s_{1 \times 1} = 3$ ,  $e_{1 \times 1} = 4$ , and  $e_{3 \times 3} = 4$ . We illustrate the convolution filters but not the activations.

## 23 · SE 结构 (SENet)

SE 结构相当于一个通道间的注意力机制，SE 结构的操作分为三个步骤，Squeeze, Excitation 和 Reweight (Scale)。

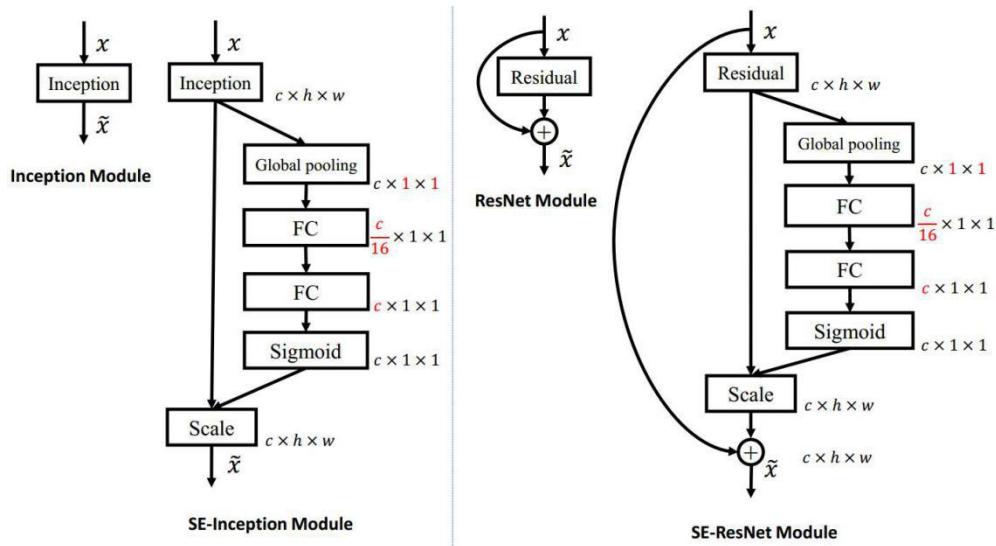
给定一个输入  $x$ ，其特征通道数为  $c_1$ ，通过一系列卷积等一般变换后得到一个特征通道数为  $c_2$  的特征。与传统的 CNN 不一样的是，接下来我们通过三个操作来重标定前面得到的特征。



首先是 **Squeeze 操作**，我们顺着空间维度来进行特征压缩（比如通过 Global Average Pooling），将每个二维的特征通道变成一个实数，这个实数某种程度上具有全局的感受野，并且输出的维度和输入的特征通道数相匹配。它表征着在特征通道上响应的全局分布，而且使得靠近输入的层也可以获得全局的感受野，这一点在很多任务中都是非常有用的。

其次是 **Excitation 操作**，它是一个类似于循环神经网络中门的机制（注意力机制）。通过参数  $w$  来为每个特征通道生成权重，其中参数  $w$  被学习用来显式地建模特征通道间的相关性。

最后是一个 **Reweight 的操作**，我们将 Excitation 的输出的权重看做是进过特征选择后的每个特征通道的重要性，然后通过乘法逐通道加权到先前的特征上，完成在通道维度上的对原始特征的重标定。



上左图是将 SE 模块嵌入到 Inception 结构的一个示例。方框旁边的维度信息代表该层的输出。

这里我们使用 global average pooling 作为 Squeeze 操作。紧接着两个 Fully Connected 层组成一个 Bottleneck 结构去建模通道间的相关性，并输出和输入特征同样数目的权重。我们首先将特征维度降低到输入的  $1/16$ ，然后经过 ReLu 激活后再通过一个 Fully Connected 层升回到原来的维度。这样做比直接用一个 Fully Connected 层的好处在于：

- 具有更多的非线性，可以更好地拟合通道间复杂的相关性；
- 极大地减少了参数量和计算量。

然后通过一个 Sigmoid 的门获得  $0\sim1$  之间归一化的权重，最后通过一个 Scale 的操作来将归一化后的权重加权到每个通道的特征上。

除此之外，SE 模块还可以嵌入到含有 skip-connections 的模块中。上右图是将 SE 嵌入到 ResNet 模块中的一个例子，操作过程基本和 SE-Inception 一样，只不过是在 Addition 前对分支上 Residual 的特征进行了特征重标定。如果对 Addition 后主支上的特征进行重标定，由于在主干上存在  $0\sim1$  的 scale 操作，在网络较深 BP 优化时就会在靠近输入层容易出现梯度消散的情况，导致模型难以优化。

参考：

<https://www.cnblogs.com/bonelee/p/9030092.html>

## 24 · NASNet 单元 (TODO)

## 25 · AmoebaNet 单元 (TODO)

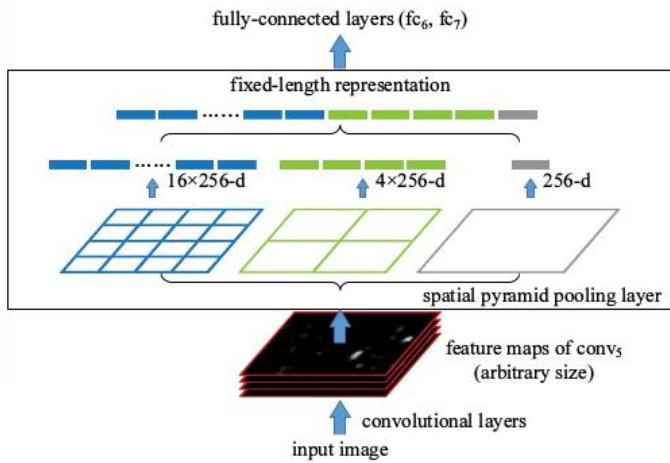
## 26 · SPP 层 (SPP-net)

**Spatial Pyramid Pooling Layer**，空间金字塔池化层，是出现在 SPP-net 中的池化层。

卷积层是输入大小无关的，也就是说无论输入图片的长宽如何，都可以正常的走过整个卷积流程，但全连接层则需要固定输入的大小，因此传统的 CNN 输入的图片大小必须是固定的。

SPP 层的目的是为了使得网络可以接收不同大小的图片（即同一图片中不同大小的区域），其原理是：针对卷积之后的特征图，以不固定大小的池化核进行池化，使得其产生

固定大小的输出，如下图中的空间金字塔池化层使用了 3 个不固定大小的池化核，使得这三个级别的池化层的输出尺寸分别为  $1 \times 1$ ,  $2 \times 2$ ,  $4 \times 4$ ，这样总的输出特征维度为  $(1+4+16) * 256 = 5376$ ：



论文：

Spatial Pyramid Pooling in Deep Convolutional Networks for Visual Recognition  
<https://arxiv.org/pdf/1406.4729.pdf>

## 27 · ROI Pooling 层 (Fast R-CNN)

**RoI (Region of Interest) Pooling Layer (RoI 池化层)** 出现在 **Fast R-CNN** 中，它受到了 SPP 层的启发，其实就是一个特化的 SPP 层：**RoI 池化层** 只有一个金字塔级别（在 Fast R-CNN 中是  $7 \times 7$ ）。假设图片大小为  $m \times n$ ，则将 pooling 的核大小和 stride 都设为  $m/7$  和  $n/7$ ，这样无论输入的图片大小为多少，出来的 feature map 都是  $7 \times 7$ 。

参考《[研究方向：图像 > 目标检测 > R-CNN 系列 > Fast R-CNN](#)》

## (二) CNN 结构

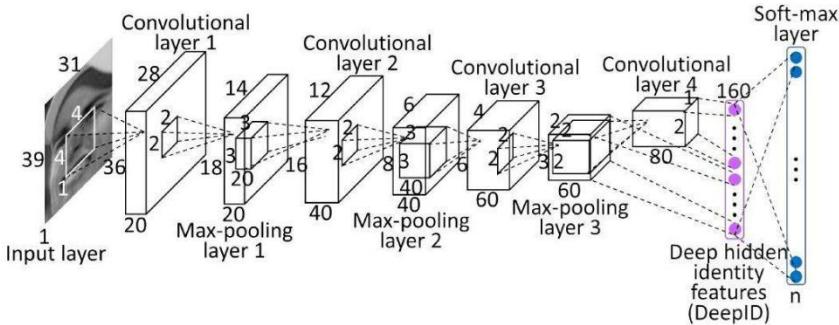
CNN 的基础网络基本上只有一种宏观结构。自深度学习介入到图像处理以来，最基础的**图像分类**任务的 CNN 的基本结构（同时也是很多其他更复杂的图像/视频处理网络的基础网络，即 backbone）

几乎可以在任何一个图像分类网络中看到这种结构，其特点是：

1. 输入为  $h \times w \times 3$  (高、宽、RGB 三个颜色通道)
2. 通过若干微结构，使得特征图越来越小，同时 channel 越来越大
3. 通常这些微结构是重复的，其中最典型和基础的为：卷积层 + 池化层

4. 最后接一个微结构（最典型的情况是一个 FC 层），并变为若干分类（Softmax）。

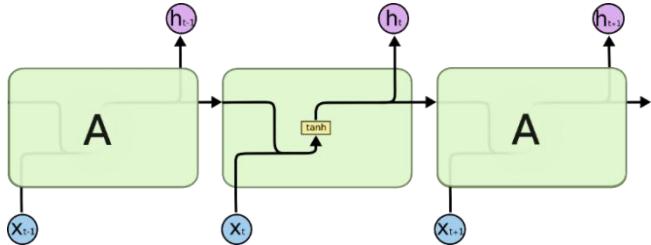
大致如下图所示：



### (三) RNN 结构

循环神经网络单元，**Recurrent Neural Network**，其被设计出来的原因是因为传统的神经网络无法处理时间序列数据，最根本的原因是传统 NN 没有记忆，而 RNN 解决了这个问题。

RNN 可以分为狭义和广义两个概念，狭义的 RNN 指的是微结构内仅有一个 tanh，如下图：



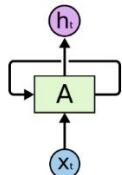
而广义的 RNN 则指代所有包括时间序列的神经网络，包括 LSTM、GRU 等等各种变种。

近年来 RNN 逐渐被 Transformer 代替。

本节描述了 RNN（广义）的神经网络结构。

## 1 · 通用结构

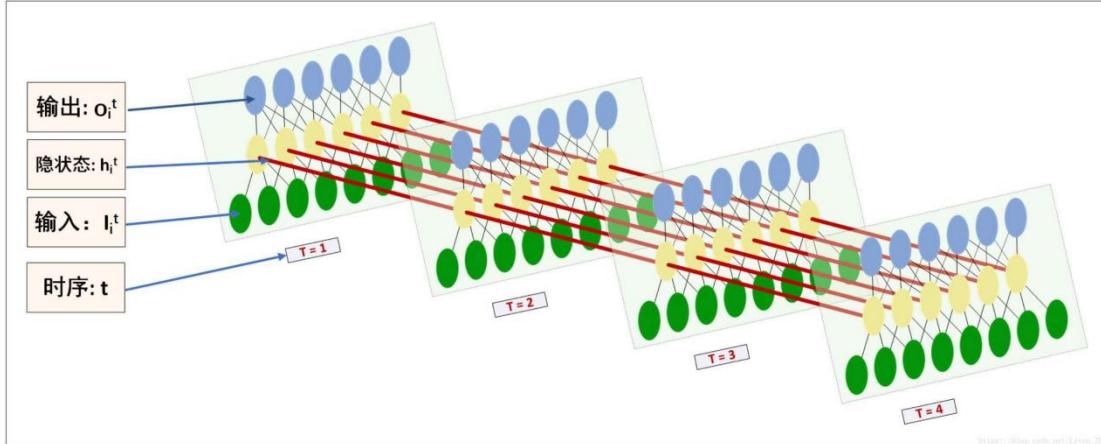
通常各种介绍 RNN、LSTM 等等的文章主要介绍的是各种 RNN 及变体的单元微结构：



从图上看，这个微结构只有 2 个输入，这两个输入其实是两个输入向量，包括：

- 自身的前一个时间步骤的输出向量  $h_{t-1}$
- 当前时间步骤的输入向量  $X_t$

那么这个单一结构的单元如何在宏观上扩展，从而组成了一整个 RNN 网络？请看下图：

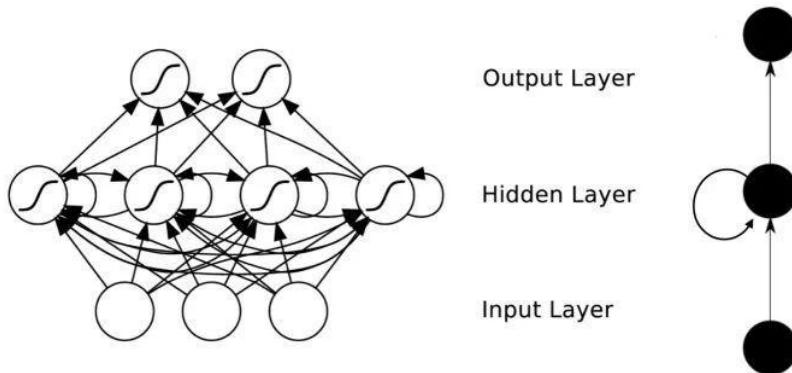


图中的每个图层表示了一个 timestep（时间步骤），也就是说所有图层其实是同一个网络在不同时刻的表现。每个图层（也就是某一时刻），在不考虑时间步骤输入的情况下，就是一个全连接网络。不同于普通全连接层的是：

1. 这个网络处理的不是一个完整的输入（比如 CNN 中的图像），而只是一个完整输入（比如一个句子）的其中一个时间步骤（比如句子中的一个词）
2. 对每个单元来讲，除了输入的向量之外还有一个时间步骤的输入向量，**图中红线并不严谨**，应该是个隐层（前个时间步骤）到隐层（后一个时间步骤）的全连接

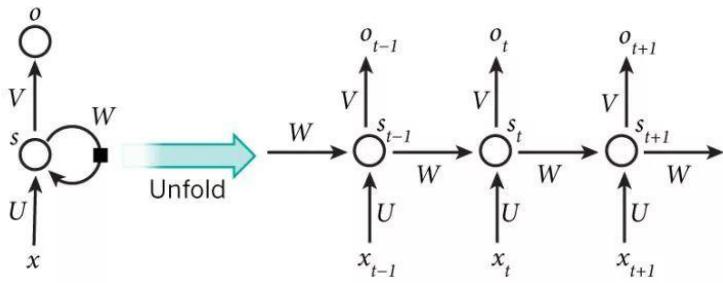
不以时间展开的 RNN 框架可以用以下结构图展示出来：

- 一个三维的输入层（一个时间步骤）
- 隐藏层包括四个 RNN 单元，与输入层全连接，也与自身（相邻时间步骤）全连接
- 一个二维的输出层，与隐藏层全连接



这是最基本的 RNN 宏结构，包括一个输入层( $n \times d$ ,  $n$  为样本数， $d$  为向量维度)、一个输出层和一个隐藏层。这里面的参数包括了 3 个矩阵：

1.  $U$ ：输入层  $\rightarrow$  隐藏层 ( $d \times h$ ,  $d$  为输入层向量维度,  $h$  为隐藏层单元个数)
2.  $V$ ：隐藏层  $\rightarrow$  输出层 ( $h \times q$ ,  $h$  为隐藏层单元个数,  $q$  为输出层单元个数)
3.  $W$ ：隐藏层  $\rightarrow$  隐藏层（不同时间步骤之间共享同一个参数矩阵， $h \times h$ ）



对于给定的时刻  $t$ ，隐藏层  $h$ ，输入  $x$  之间的算法为：

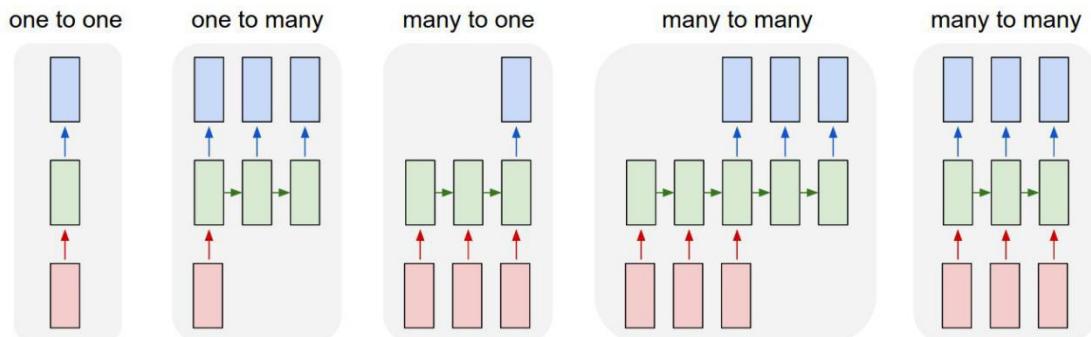
$$h^{(t)} = \phi(Ux^{(t)} + Wh^{(t-1)} + b)$$

其中  $\phi$  为激活函数，一般是  $\tanh$ ， $b$  为偏置。输出  $o$  的算法则为：

$$o^{(t)} = Vh^{(t)} + c$$

## 2 · 变种结构

以下是 RNN（包括 LSTM、GRU 等）宏观结构的几种时序方面的变种：



由于没有时许，图中最左边并非是一个 RNN，而另外四种则是 RNN 宏观结构的四种形态。  
下面一一介绍：

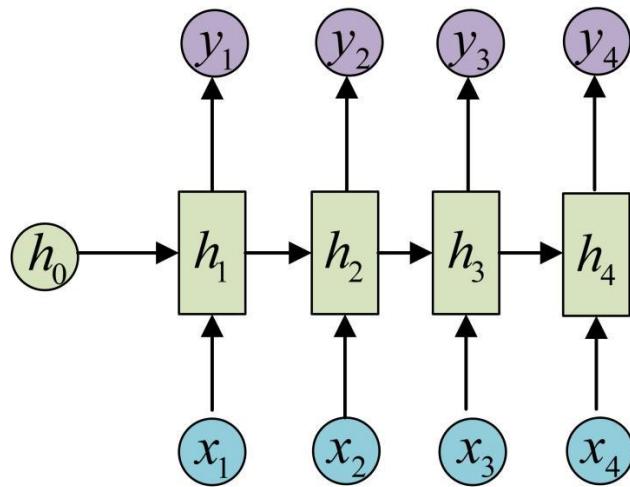
参考：<https://zhuanlan.zhihu.com/p/28054589>

### (1) N vs N

这就是最基本的 RNN 结构，即每一步都输出的通用结构，但是由于输入和输出必须完全完全等长，导致这种结构的实际应用很少，但也有一些问题适合用经典的 RNN 结构建模：

- 计算视频每一帧的分类标签。因为要对每一帧进行计算，因此输入和输出序列等长。

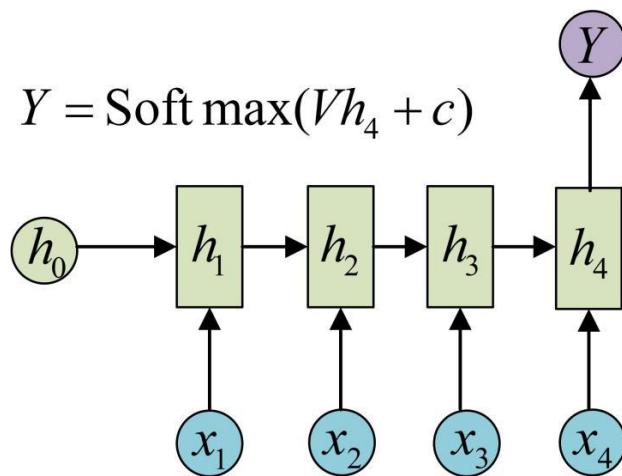
- 输入为字符，输出为下一个字符的概率。这就是著名的 Char RNN（可以用来生成文章，诗歌，甚至是代码）。



## (2) N vs 1

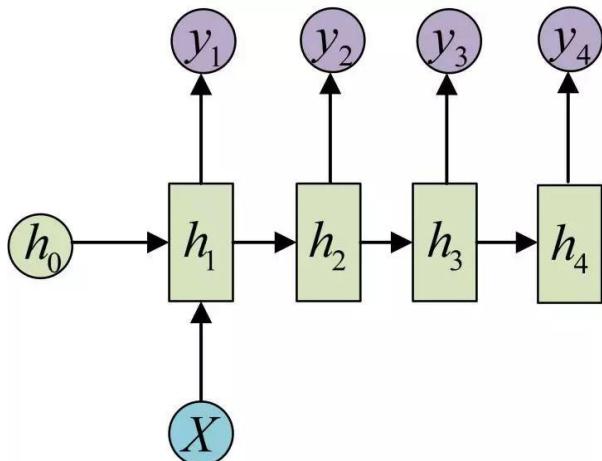
输入一个序列，输出一个非时序的张量。这种情况仅在最后一步进行输出变换，这种结构通常用来处理序列分类问题。比如：

- 输入一段文字判别它所属的类别
  - 输入一个句子判断其情感倾向
  - 输入一段视频并判断它的类别
- 等等。

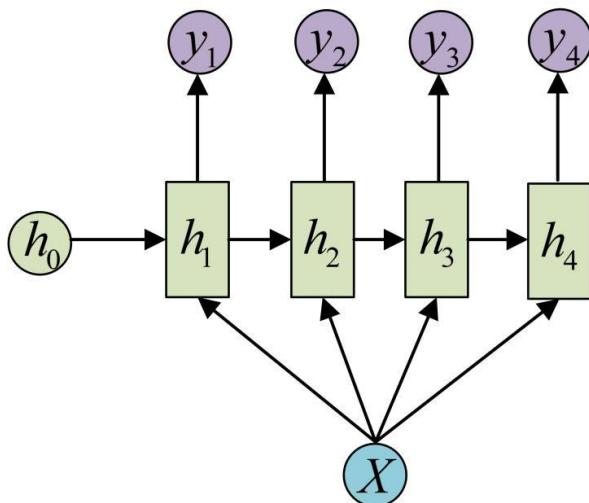


## (3) 1 vs N

输入不是序列而输出为序列的情况怎么处理？我们可以只在序列开始进行输入计算：



还有一种结构是把输入信息  $X$  作为每个阶段的输入：



这种 1 VS N 的结构可以处理的问题有：

- 从图像生成文字 (**image caption**)，此时输入的  $X$  就是图像的特征，而输出的  $y$  序列就是一段句子
- 从类别生成语音或音乐等

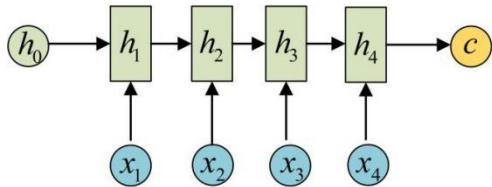
#### (4) N vs M

这是 RNN 通用结构最重要的一个变种，这种结构又叫 **Encoder-Decoder** 结构，或者 **Seq2Seq** 结构。

原始的 N vs N RNN 要求序列等长，然而我们遇到的大部分问题序列都是不等长的，如机器翻译中，源语言和目标语言的句子往往并没有相同的长度。

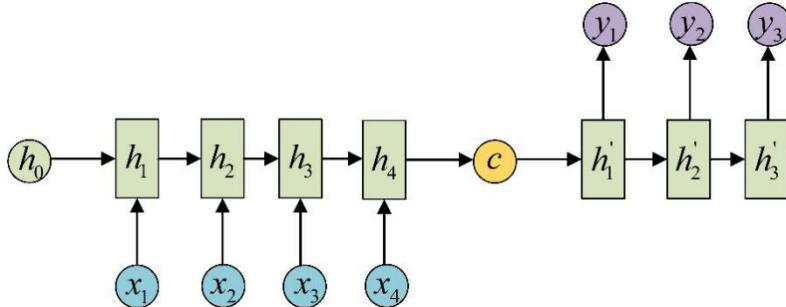
为此，Encoder-Decoder 结构先将输入数据编码成一个上下文向量  $c$ ：

- (1)  $c = h_4$
- (2)  $c = q(h_4)$
- (3)  $c = q(h_1, h_2, h_3, h_4)$

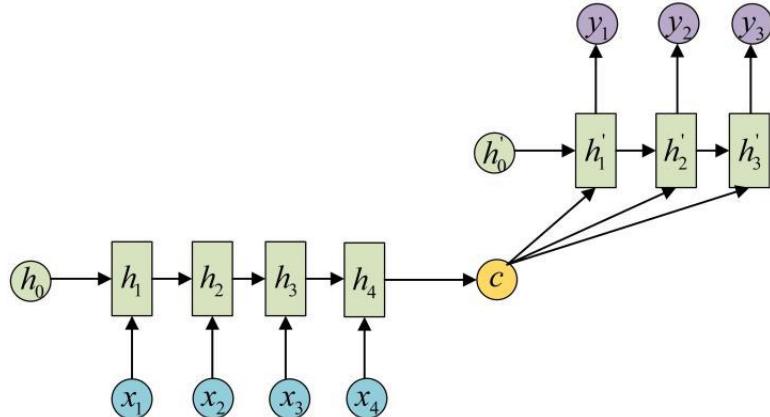


得到  $c$  有多种方式，最简单的方法就是把 Encoder 的最后一个隐状态赋值给  $c$ ，还可以对最后的隐状态做一个变换得到  $c$ ，也可以对所有的隐状态做变换。

拿到  $c$  之后，就用另一个 RNN 网络对其进行解码，这部分 RNN 网络被称为 Decoder。具体做法就是将  $c$  当做之前的初始状态  $h_0$  输入到 Decoder 中：



还有一种做法是将  $c$  当做每一步的输入：



由于这种 Encoder-Decoder 结构不限制输入和输出的序列长度，因此应用的范围非常广泛，比如：

- 机器翻译，Encoder-Decoder 的最经典应用，事实上这一结构就是在机器翻译领域最先提出的
- 文本摘要，输入是一段文本序列，输出是这段文本序列的摘要序列。
- 阅读理解，将输入的文章和问题分别编码，再对其进行解码得到问题的答案。
- 语音识别，输入是语音信号序列，输出是文字序列。

- 视频摘要（Video Caption），输入是一段视频，输出是该视频的描述

论文：

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

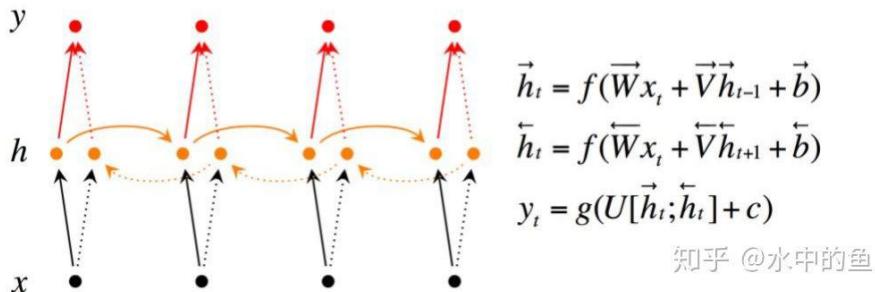
<https://arxiv.org/pdf/1406.1078.pdf>

**Seq2Seq** : Sequence to Sequence Learning with Neural Networks

<https://arxiv.org/pdf/1409.3215.pdf>

### 3 · 双向 RNN

双向 RNN，Bidirectional RNN，顾名思义 RNN 在两个方向同时进行，它假设当前的输出不仅仅和之前的序列有关系，还和之后的序列有关系（比如预测一个缺失的词，需要同时根据上下文进行预测）。Bidirectional RNN 是一个相对简单的 RNNs，由两个 RNNs 上下叠加在一起组成。输出由这两个 RNNs 的隐藏层的状态决定。



我们需要将从前往后和从后往前这两部分的结果拼接起来，如果他们都是  $1000*1$  维的，则拼接起来就是  $2000*1$  维的。

论文

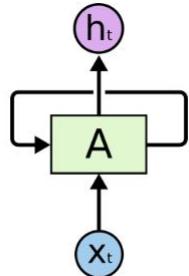
<https://www.di.ufpe.br/~fnj/RNA/bibliografia/BRNN.pdf>

### (四) RNN 微结构

RNN 微结构介绍的是单个 RNN 单元内部的结构

## 1 · 基本 RNN 单元

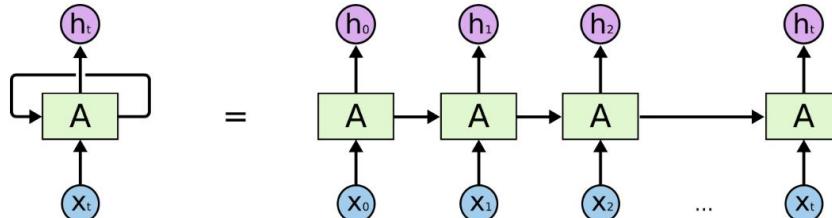
基本 RNN 的单元都带有循环，基本单元如下图：



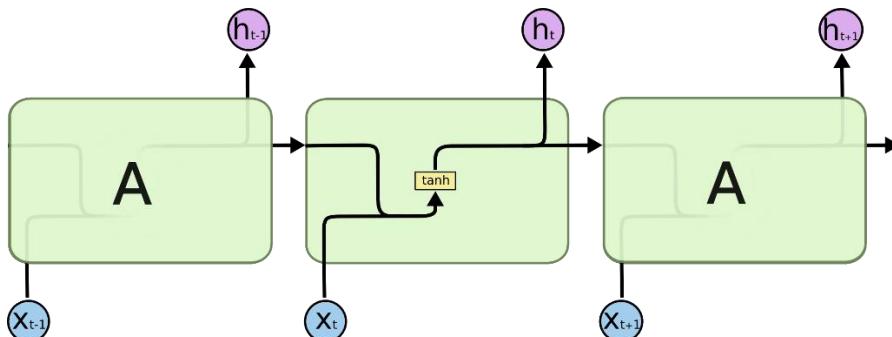
可以看到相较于普通的神经网络（DNN/CNN），RNN 增加了循环，其中 A 为 RNN 单元， $x_t$  表示输入， $h_t$  表示输出， $t$  为表示不同的时间步骤。对于每个 RNN 单元来说，有 2 个输入：

- 一个是用户的输入，也是时间序列输入  $x_t$ ，比如自然语言处理任务中的每个词。
- 另一个是同一个单元的上一步的输出，也就是第  $t$  步的输出  $h_t$  被作为  $t+1$  步的输入。

按时间展开之后如下：



所有的 RNN（广义的 RNN，包括 LSTM、GRU 等）都有类似的结构，但每个不同的 RNN 变种单元中的具体细节有些区别。我们这里的 RNN 特指最简单的 RNN（或者叫标准 RNN，Standard RNN，Keras 中被称为 SimpleRNN），该 RNN 单元内部的结构如下图所示：



相较于 DNN 的隐节点数学表达式为：

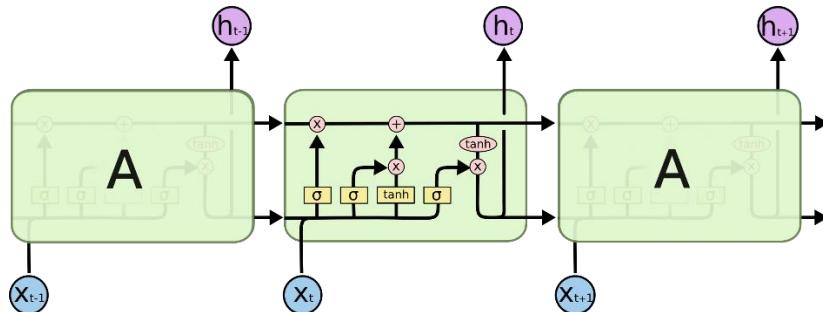
$$h_t = \sigma(x_t \times w_{xt} + b)$$

RNN 隐节点的数学表达式：

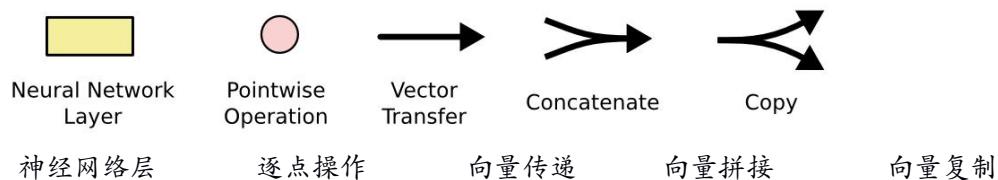
$$h_t = \sigma(x_t \times w_{xt} + h_{t-1} \times w_{ht} + b)$$

## 2 · LSTM 单元

**Long Short-Term Memory Cell**，长短期记忆网络单元，LSTM 算是广义 RNN 的一种，其网络结构也与标准 RNN 类似，区别在于其微结构不同。其微结构如下图：

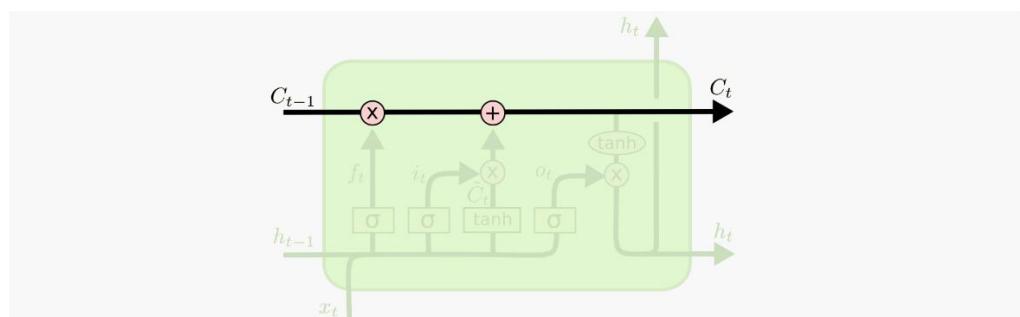


图中各个组成标示如下：



- : sigmoid 函数
- : tanh 函数
- : 向量的逐点相乘
- : 向量的逐点相加

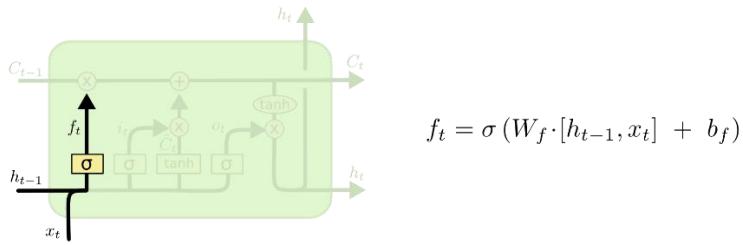
相较于标准 RNN，LSTM 多了一个细胞状态（Cell State）的时间步骤输出（同时也是输入），即图中上面那条横线，用  $C_t$  表示。如下图：



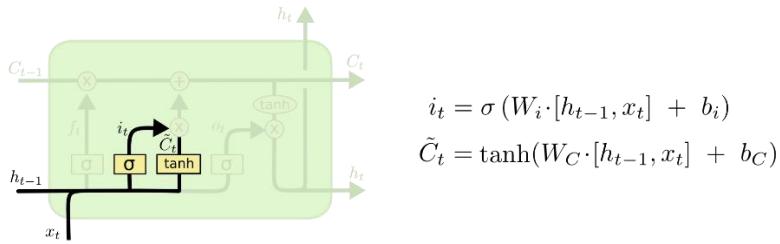
从图中可以看出，在一个时间步骤内， $C_{t-1}$ （上一个时间步骤输出的  $C_t$ ）变到  $C_t$  需要经

过两次逐点操作，分别被称为：

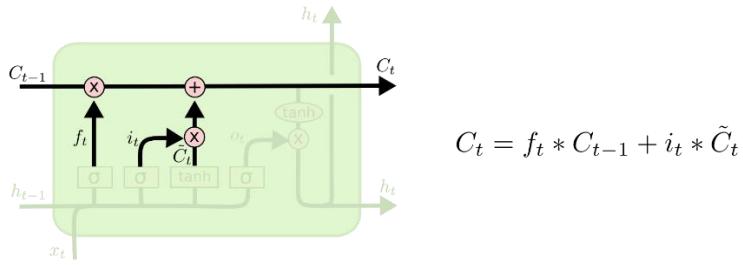
- **遗忘门 (Forget gate layer)**，用于去除不需要的信息， $f_t$  的输出在 0-1 之间（根据 sigmoid 函数），因此可以决定是忘记 (0) 还是记住 (1)，或者介于两者之间。



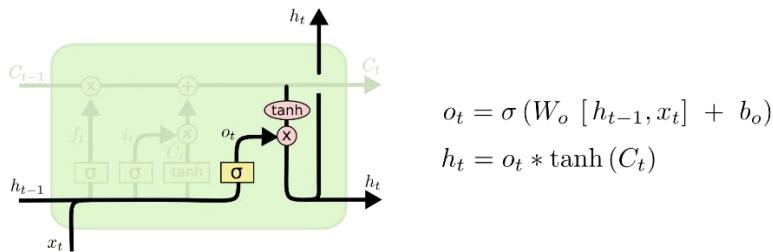
- **输入门 (Input gate layer)**，用于加入新的信息



$C_{t-1}$  到  $C_t$  的更新首先需要逐点乘以  $f_t$ ，以便“忘记”之前的某些信息，然后再逐点加上某些新的信息：



最后，我们需要根据  $C_t$  决定输出的  $h_t$ ：

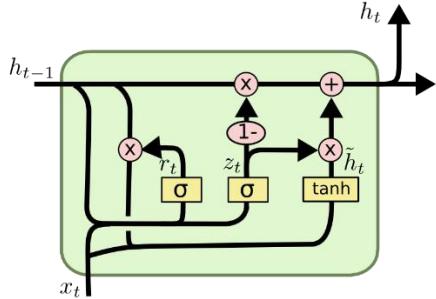


参考：

<https://zhuanlan.zhihu.com/p/42717426>

### 3 · GRU 单元

LSTM 有很多变种，其中最著名的是 GRU (Gated recurrent unit)，其微结构如下：



$$z_t = \sigma(W_z \cdot [h_{t-1}, x_t])$$
$$r_t = \sigma(W_r \cdot [h_{t-1}, x_t])$$
$$\tilde{h}_t = \tanh(W \cdot [r_t * h_{t-1}, x_t])$$
$$h_t = (1 - z_t) * h_{t-1} + z_t * \tilde{h}_t$$

GRU 和 LSTM 均是采用门机制的思想改造 RNN 的神经元，和 LSTM 相比，GRU 更加简单，高效，且不容易过拟合，但有时候在更加复杂的场景中效果不如 LSTM，算是 RNN 和 LSTM 在速度和精度上的一个折中方案。

## (五) 注意力机制 (Attention)

注意力 (Attention) 的概念最早在九几年就提出来了，简单来说就是通过不同的权重，重点关注输入的不同部分，从而得到对应的输出。这就像是人类的视觉中通过快速扫描全局图像，获得需要重点关注的目标区域。

广义的注意力 (Attention) 机制，既不是一个具体的结构，也不是一个微结构，而更象是一个概念和抽象的微结构，在不同的网络中可能会有不同的表现形式。

### ● 理解

我们可以这样来看待 Attention 机制：将 Source 中的构成元素想象成是由一系列的  $\langle \text{Key}, \text{Value} \rangle$  数据对构成，此时给定 Target 中的某个元素 Query，通过计算 Query 和各个 Key 的相似性或者相关性，得到每个 Key 对应 Value 的权重系数，然后对 Value 进行加权求和，即得到了最终的 Attention 数值。所以本质上 Attention 机制是对 Source 中元素的 Value 值进行加权求和，而 Query 和 Key 用来计算对应 Value 的权重系数。即可以将其本质思想改写为如下公式：

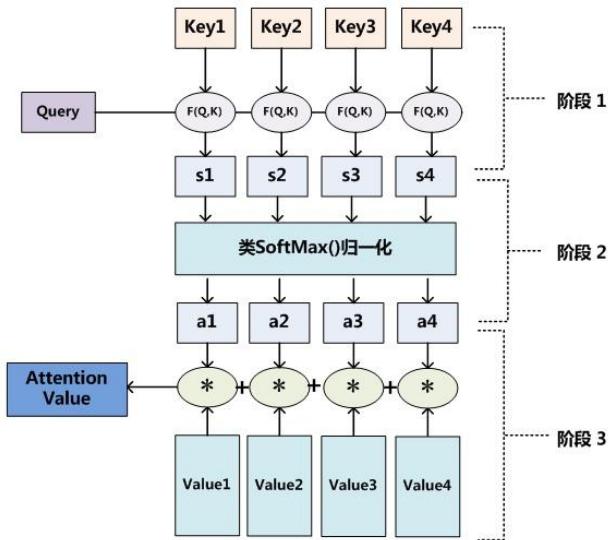
$$\text{Attention}(\text{Query}, \text{Source}) = \sum_{i=1}^{L_x} \text{Similarity}(\text{Query}, \text{Key}_i) * \text{Value}_i$$

也可以将 Attention 机制看作一种软寻址 (Soft Addressing)，Source 可以看作存储器内存储的内容，元素由地址 Key 和值 Value 组成，当前有个  $\text{Key}=\text{Query}$  的查询，目的是取出存储器中对应的 Value 值，即 Attention 数值。通过 Query 和存储器内元素 Key 的地址进行相似性比较来寻址，之所以说是软寻址，指的不像一般寻址只从存储内容里面找出一条内容，而是可能从每个 Key 地址都会取出内容，取出内容的重要性根据 Query 和 Key 的相似性来决定，之后对 Value 进行加权求和，这样就可以取出最终的 Value 值，也即

Attention 值。所以不少研究人员将 Attention 机制看作软寻址的一种特例，这也是非常有道理的。

### ● 计算过程

至于 Attention 机制的具体计算过程，如果对目前大多数方法进行抽象的话，可以将其归纳为两个过程：第一个过程是根据 Query 和 Key 计算权重系数，第二个过程根据权重系数对 Value 进行加权求和。而第一个过程又可以细分为两个阶段：第一个阶段根据 Query 和 Key 计算两者的相似性或者相关性；第二个阶段对第一阶段的原始分值进行归一化处理；这样，可以将 Attention 的计算过程抽象为如下图展示的三个阶段。



在第一个阶段，可以引入不同的函数和计算机制，根据 Query 和某个 Key<sub>i</sub>，计算两者的相似性或者相关性，最常见的方法包括：求两者的向量点积、求两者的向量 Cosine 相似性或者通过再引入额外的神经网络来求值，即如下方式：

$$\text{点积: } \text{Similarity}(\text{Query}, \text{Key}_i) = \text{Query} \cdot \text{Key}_i$$

$$\text{Cosine 相似性: } \text{Similarity}(\text{Query}, \text{Key}_i) = \frac{\text{Query} \cdot \text{Key}_i}{\|\text{Query}\| \|\text{Key}_i\|}$$

$$\text{MLP 网络: } \text{Similarity}(\text{Query}, \text{Key}_i) = \text{MLP}(\text{Query}, \text{Key}_i)$$

第一阶段产生的分值根据具体产生的方法不同其数值取值范围也不一样，第二阶段引入类似 SoftMax 的计算方式对第一阶段的得分进行数值转换，一方面可以进行归一化，将原始计算分值整理成所有元素权重之和为 1 的概率分布；另一方面也可以通过 SoftMax 的内在机制更加突出重要元素的权重。即一般采用如下公式计算：

$$a_i = \text{Softmax}(\text{Sim}_i) = \frac{e^{\text{Sim}_i}}{\sum_{j=1}^{L_x} e^{\text{Sim}_j}}$$

第二阶段的计算结果  $a_i$  即为  $\text{value}_i$  对应的权重系数，然后进行加权求和即可得到 Attention 数值：

$$\text{Attention}(\text{Query}, \text{Source}) = \sum_{i=1}^{L_x} a_i \cdot \text{Value}_i$$

通过如上三个阶段的计算，即可求出针对 Query 的 Attention 数值，目前绝大多数具体的注意力机制计算方法都符合上述的三阶段抽象计算过程。

- RNN

Attention 机制本身和具体 NN 的类型（CNN、RNN）无关，Attention 机制在 2014 年机器翻译的论文《Neural Machine Translation by Jointly Learning to Align and Translate》中重新成为热门，论文将 RNN Encoder-Decoder 和 Attention 机制结合在一起。

- Soft & Hard Attention

在 Kelvin Xu 等人 2015 年关于图像描述（Image Caption）的论文《Show, Attend and Tell》中介绍了硬注意力（Hard Attention）和软注意力（Soft Attention）的概念。这其中 Soft Attention 即是传统的 Attention。

- Global & Local Attention

在 2015 年的另一篇关于机器翻译的论文《Effective Approaches to Attention-based Neural Machine Translation》中提出了 Global Attention 和 Local Attention 的概念。

- Self-Attention

在 2017 年 Google 关于 NLP 的著名论文《Attention is All You Need》中，给出了 Self-Attention 的概念，并且据此提出了 Transformer 结构，成为后来各种 BERT 的基础。

参考：

<https://www.zhihu.com/question/68482809/answer/264632289>

<https://zhuanlan.zhihu.com/p/31547842>

注意力机制：

<https://blog.csdn.net/yimingsilence/article/details/79208092>

5 种 Attention：

[http://www.360doc.com/content/19/0804/02/46368139\\_852849046.shtml](http://www.360doc.com/content/19/0804/02/46368139_852849046.shtml)

Attention 综述：

<https://zhuanlan.zhihu.com/p/62136754>

NLP 中的 Attention 机制：

<https://zhuanlan.zhihu.com/p/53682800>

自然语言中的 Attention：

<https://blog.csdn.net/hahajinbu/article/details/81940355>

论文

An Attentive Survey of Attention Models

<https://arxiv.org/pdf/1904.02874.pdf>

## 1 · CNN 中的 Attention

CNN 中的 SENet 就是一种简单的注意力机制，参考《[网络构成 > DNN/CNN 微结构 > SE 结构](#)》和《[研究方向:图像 > 图像分类 > SENet](#)》

## 2 · RNN 中的 Attention (201409)

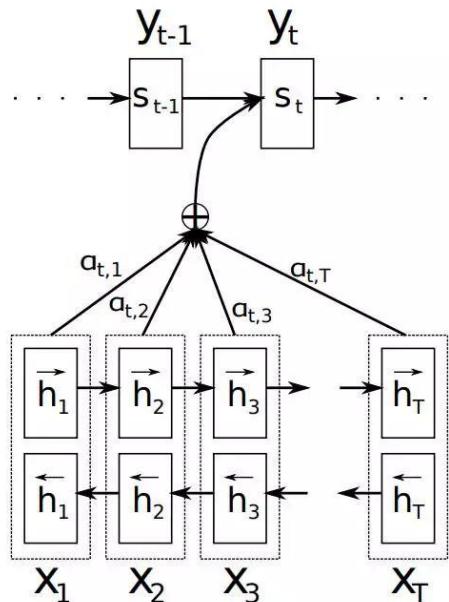
在普通的 RNN Encoder-Decoder 模型中，输入序列 X 和输出序列 Y 之间的传递仅仅通过一个中间向量 C 实现。这就带来如下问题：

- C 的长度是固定的，如果输入序列 X 太长的话，特别是比训练集中最初的句子长度还长时，模型的性能急剧下降。
- 把输入 X 编码成一个固定的长度，对于句子中每个词都赋予相同的权重，这样做是不合理的，比如，在机器翻译里，输入的句子与输出句子之间，往往是输入一个或几个词对应于输出的一个或几个词。因此，对输入的每个词赋予相同权重，这样做没有区分度，往往使得模型性能下降。（这个问题的影响程度存疑，在 X 足够短的情况下，这种偏重应该是可以通过 Encoder 和 Decoder 中的权重体现出来的）

这篇关于机器翻译 (Machine Translation) 的论文主要应用了两个主要技术：

- 双向 RNN，其中每个 RNN 单元为 GRU。
- Attention 机制

如下图：



传统 RNN 的 Encoder-Decoder 机制中，将整个句子的特征向量 C 作为 Decoder 的输入：

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

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

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

其中：

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

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

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

其中  $a$  即是 **Similarity 函数**（文中叫 alignment model，对齐模型，即原语言中的词和翻译语言中词的对齐），表示注意力权重。 $a$  的计算方式有很多种，最简单最常用的是点积计算。

论文：

Neural Machine Translation by Jointly Learning to Align and Translate

<https://arxiv.org/pdf/1409.0473.pdf>

### 3 · Soft & Hard attention (201502)

2015 年发表的论文《Show, Attend and Tell: Neural Image Caption Generation with Visual Attention》，在 Image Caption 中引入了 Attention，当生成第  $i$  个关于图片内容描述的词时，用 Attention 来关联与  $i$  个词相关的图片的区域。论文中使用了两种 Attention Mechanism，即**软注意力 (Soft Attention)**，就是传统的 **Attention** 和**硬注意力 (Hard Attention)**。Soft Attention 是参数化的，因此可导，可以被嵌入到模型中去，直接训练。梯度可以经过 Attention Mechanism 模块，反向传播到模型其他部分。

相反，Hard Attention 是一个随机的过程。Hard Attention 不会选择整个 encoder 的输出做为其输入，Hard Attention 会依概率  $S_i$  来采样输入端的隐状态一部分来进行计算，而不是整个 encoder 的隐状态。为了实现梯度的反向传播，需要采用蒙特卡洛采样的方法来估计模块的梯度。

两种 Attention Mechanism 都有各自的优势，但目前更多的研究和应用还是更倾向于使用 Soft Attention，因为其可以直接求导，进行梯度反向传播。

论文：

Show, Attend and Tell: Neural Image Caption Generation with Visual Attention

<https://arxiv.org/pdf/1502.03044.pdf>

代码

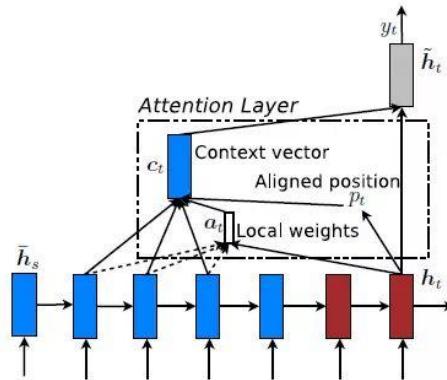
<https://github.com/kelvinxu/arctic-captions>

## 4 · Global & Local Attention (201508)

在 2015 年关于机器翻译的论文《Effective Approaches to Attention-based Neural Machine Translation》中提出了全局注意力 (Global Attention, 即传统的 Attention) 和局部注意力 (Local Attention) 的概念。

Global Attention 有一个明显的缺点就是，每一次，encoder 端的所有 hidden state 都要参与计算，这样做计算开销会比较大，特别是当 encoder 的句子偏长，比如，一段话或者一篇文章，效率偏低。

Local Attention 可以视作是一种介于 Soft Attention 和 Hard Attention 之间的一种 Attention 方式，即把两种方式结合起来：



Local Attention 首先会为 decoder 端当前的词，预测一个 source 端对齐位置 (aligned position)  $p_t$ ，然后基于  $p_t$  选择一个窗口，用于计算背景向量  $c_t$ 。

但是，在实际应用中，Global Attention 应用更普遍，因为 Local Attention 需要预测一个位置向量  $p$ ，这就带来两个问题：

- 1、当 encoder 句子不是很长时，相对 Global Attention，计算量并没有明显减小。
- 2、位置向量  $p_t$  的预测并不非常准确，这就直接计算的到的 local Attention 的准确率。

论文：

Effective Approaches to Attention-based Neural Machine Translation

<https://arxiv.org/pdf/1508.04025.pdf>

github：

[https://github.com/lmthang/nmt\\_matlab](https://github.com/lmthang/nmt_matlab)

## 5 · Transformer & Self-Attention (201706)

参考《研究方向:NLP > NN：基于 Transformer > Transformer》

## 6 · Non-Local Network (201711)

Non-local Network 是 FAIR 在 17 年的论文，其所介绍的 Non-local network 算是 CV 领域里自注意力机制的核心。

Non-Local 的操作的公式如下：

$$\mathbf{y}_i = \frac{1}{\mathcal{C}(\mathbf{x})} \sum_{\forall j} f(\mathbf{x}_i, \mathbf{x}_j) g(\mathbf{x}_j).$$

$\mathbf{x}$ ：输入， $i$  和  $j$  代表不同的位置， $\mathbf{x}_i$  是一个向量，维数同  $\mathbf{x}$  的 channel 数

$\mathbf{y}$ ：输出，和原图大小一致

$f$ ：计算两个点相似关系的函数

$g$ ：映射函数，将一个点映射为一个向量

由于输入和输出形状保持一致，使得 Non-Local 可以很容易的插入 CNN 和 RNN 之中。

论文中  $f$  的形式包括以下四个：

Gaussian :  $f(\mathbf{x}_i, \mathbf{x}_j) = e^{\mathbf{x}_i^T \mathbf{x}_j}.$

Embedded Gaussian :  $f(\mathbf{x}_i, \mathbf{x}_j) = e^{\theta(\mathbf{x}_i)^T \phi(\mathbf{x}_j)}.$

Dot Production :  $f(\mathbf{x}_i, \mathbf{x}_j) = \theta(\mathbf{x}_i)^T \phi(\mathbf{x}_j).$

Concatenation :  $f(\mathbf{x}_i, \mathbf{x}_j) = \text{ReLU}(\mathbf{w}_f^T [\theta(\mathbf{x}_i), \phi(\mathbf{x}_j)]).$

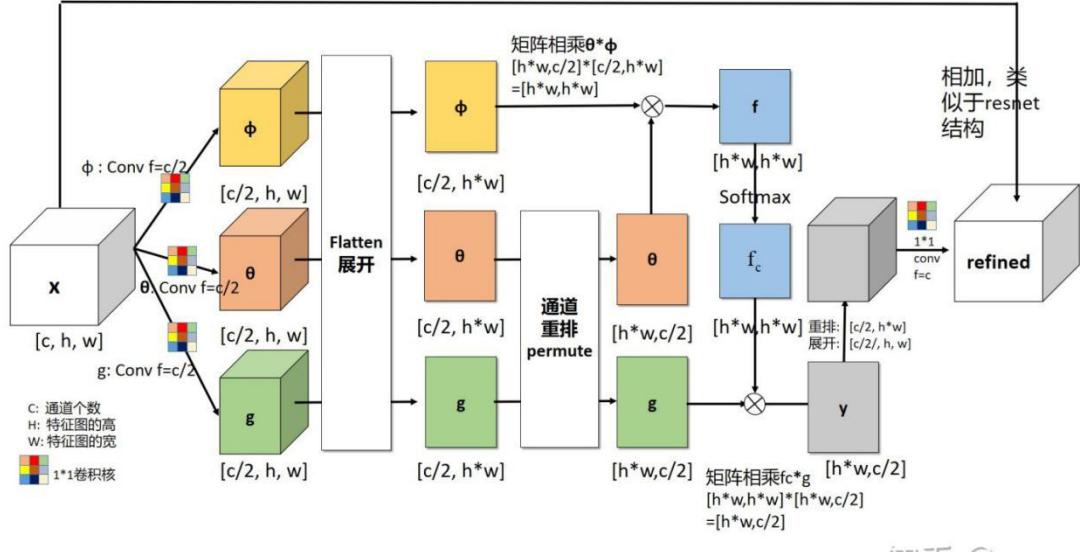
上述公式中的  $\theta$  和  $\phi$  为两个 embedding :

$$\theta(\mathbf{x}_i) = \mathbf{W}_\theta \mathbf{x}_i$$

$$\phi(\mathbf{x}_i) = \mathbf{W}_\phi \mathbf{x}_i$$

以 Embedded Gaussian 为例的整个流程如下图：

$$y_i = \text{softmax}(\theta(x_i)^T \phi(x_j)) g(x_j) = \frac{1}{\sum_{\forall j} e^{\theta(x_i)^T \phi(x_j)}} e^{\theta(x_i)^T \phi(x_j)} W_g x_j$$



知乎 @pprp

<https://zhuanlan.zhihu.com/p/30120220>

论文原文中的流程图更加抽象一点，并且多了一个维度——原文是针对视频的，因此除了表示图片长宽的H和W维度，还有个表示时间步骤的T维度：

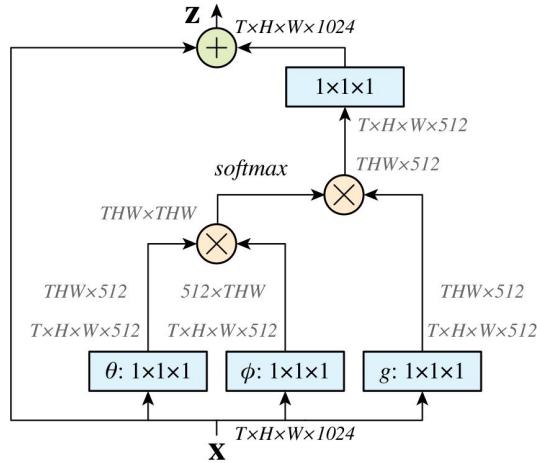


Figure 2. A spacetime **non-local block**. The feature maps are shown as the shape of their tensors, e.g.,  $T \times H \times W \times 1024$  for 1024 channels (proper reshaping is performed when noted). “ $\otimes$ ” denotes matrix multiplication, and “ $\oplus$ ” denotes element-wise sum. The softmax operation is performed on each row. The blue boxes denote  $1 \times 1 \times 1$  convolutions. Here we show the embedded Gaussian version, with a bottleneck of 512 channels. The vanilla Gaussian version can be done by removing  $\theta$  and  $\phi$ , and the dot-product version can be done by replacing softmax with scaling by  $1/N$ .

跟FC层的关系：

如果

- 任意两点的相似性仅跟两点的位置有关，即  $f(x_i, x_j) = W_{ij}$
- $g$  是 identity 函数，即  $g(x_i) = x_i$
- 归一化系数为 1。归一化系数跟输入无关，全连接层不能处理任意尺寸的输入。  
那么此时 Non-Local 变为 FC 层。

跟 Transformer 的关系：

如果  $f$  采用 dot production，即  $f(x_i, x_j) = \theta(x_i)^T \phi(x_j)$ ，而：

$$\theta(x_i) = W_\theta x_i, \text{ 可得 } \theta(x) = W_\theta x$$

$$\phi(x_i) = W_\phi x_i, \text{ 可得 } \phi(x) = W_\phi x$$

从而得到：

$$y = \text{softmax}(x^T W_\theta^T W_\phi x)$$

此时 Non-Local 等同于 Transformer，也就是说，Transformer 可以被视作是 Non-Local 的一个特例。

论文：

<https://arxiv.org/pdf/1711.07971.pdf>

参考：

<https://zhuanlan.zhihu.com/p/33345791>

<https://zhuanlan.zhihu.com/p/102984842>

## (六) GAN 结构

GAN 于 2014 年由 Ian Goodfellow 提出，GAN 的贡献主要是其理念及对应的网络结构。GAN 是深度学习领域的一个重要组成部分。

论文：

<https://arxiv.org/pdf/1406.2661.pdf>

综述参考：

<https://zhuanlan.zhihu.com/p/58812258>

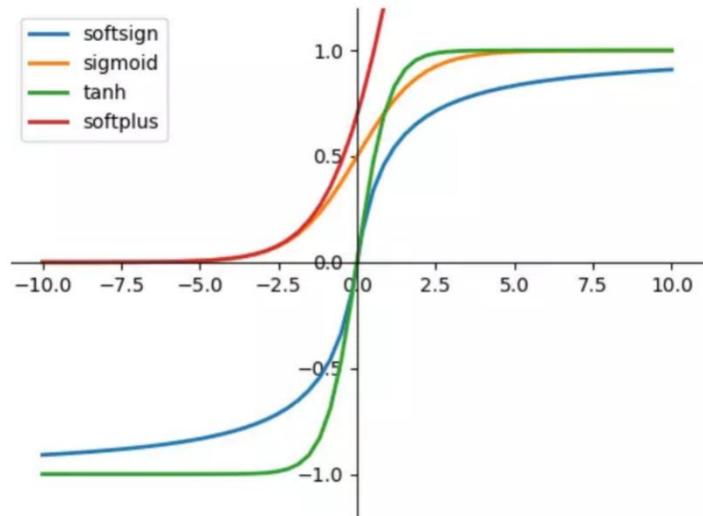
<https://zhuanlan.zhihu.com/p/110581201>

## (七) 激活函数 (Activation Function)

对于某个隐藏节点的计算，该节点的激活值分为 2 步：先是一步线性变换，之后又被作用了一个函数，即激活函数。

激活函数（Activation Function）的作用是，增加神经网络的非线性。如果没有激活层，因为所有层都是线性关系，输出都是输入的线性组合，所有的隐藏层都会变得没有意义。如果使用了激活函数，激活函数引入了非线性，则神经网络可以逼近任何非线性函数。

下图是 **sigmoid**、**tanh**、**softplus** 和 **softsign** 激活函数的曲线图（注意 x 和 y 轴不成比例，曲线图被拉高了）：



下面是各种激活函数的名称、公式、

| Name                                                      | Plot | Equation                                                                                                                                                             |
|-----------------------------------------------------------|------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Identity                                                  |      | $f(x) = x$                                                                                                                                                           |
| Binary step                                               |      | $f(x) = \begin{cases} 0 & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$                                                                                 |
| Logistic (a.k.a. Sigmoid or Soft step)                    |      | $f(x) = \sigma(x) = \frac{1}{1 + e^{-x}}$ [1]                                                                                                                        |
| TanH                                                      |      | $f(x) = \tanh(x) = \frac{(e^x - e^{-x})}{(e^x + e^{-x})}$                                                                                                            |
| Rectified linear unit (ReLU) <sup>[11]</sup>              |      | $f(x) = \begin{cases} 0 & \text{for } x \leq 0 \\ x & \text{for } x > 0 \end{cases} = \max\{0, x\} = x \mathbf{1}_{x>0}$                                             |
| Gaussian Error Linear Unit (GELU) <sup>[6]</sup>          |      | $f(x) = x\Phi(x) = x(1 + \text{erf}(x/\sqrt{2}))/2$                                                                                                                  |
| SoftPlus <sup>[12]</sup>                                  |      | $f(x) = \ln(1 + e^x)$                                                                                                                                                |
| Exponential linear unit (ELU) <sup>[13]</sup>             |      | $f(\alpha, x) = \begin{cases} \alpha(e^x - 1) & \text{for } x \leq 0 \\ x & \text{for } x > 0 \end{cases}$                                                           |
| Scaled exponential linear unit (SELU) <sup>[14]</sup>     |      | $f(\alpha, x) = \lambda \begin{cases} \alpha(e^x - 1) & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$<br>with $\lambda = 1.0507$ and $\alpha = 1.67326$ |
| Leaky rectified linear unit (Leaky ReLU) <sup>[15]</sup>  |      | $f(x) = \begin{cases} 0.01x & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$                                                                             |
| Parameteric rectified linear unit (PReLU) <sup>[16]</sup> |      | $f(\alpha, x) = \begin{cases} \alpha x & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$                                                                  |

|                                                                                                  |  |                                                                                                                                                                                                                             |
|--------------------------------------------------------------------------------------------------|--|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| ArcTan                                                                                           |  | $f(x) = \tan^{-1}(x)$                                                                                                                                                                                                       |
| ElliotSig <sup>[17][18]</sup> Softsign <sup>[19][20]</sup>                                       |  | $f(x) = \frac{x}{1 +  x }$                                                                                                                                                                                                  |
| Square Nonlinearity (SQNL) <sup>[21]</sup>                                                       |  | $f(x) = \begin{cases} 1 & : x > 2.0 \\ x - \frac{x^2}{4} & : 0 \leq x \leq 2.0 \\ x + \frac{x^2}{4} & : -2.0 \leq x < 0 \\ -1 & : x < -2.0 \end{cases}$                                                                     |
| S-shaped rectified linear activation unit (SReLU) <sup>[22]</sup>                                |  | $f_{t_l, a_l, t_r, a_r}(x) = \begin{cases} t_l + a_l(x - t_l) & \text{for } x \leq t_l \\ x & \text{for } t_l < x < t_r \\ t_r + a_r(x - t_r) & \text{for } x \geq t_r \end{cases}$<br>$t_l, a_l, t_r, a_r$ are parameters. |
| Bent identity                                                                                    |  | $f(x) = \frac{\sqrt{x^2 + 1} - 1}{2} + x$                                                                                                                                                                                   |
| Sigmoid Linear Unit (SiLU) <sup>[6]</sup> (AKA SiL <sup>[23]</sup> and Swish-1 <sup>[24]</sup> ) |  | $f(x) = \frac{x}{1 + e^{-x}}$                                                                                                                                                                                               |
| Sinusoid <sup>[25]</sup>                                                                         |  | $f(x) = \sin(x)$                                                                                                                                                                                                            |
| Sinc                                                                                             |  | $f(x) = \begin{cases} 1 & \text{for } x = 0 \\ \frac{\sin(x)}{x} & \text{for } x \neq 0 \end{cases}$                                                                                                                        |
| Gaussian                                                                                         |  | $f(x) = e^{-x^2}$                                                                                                                                                                                                           |
| SQ-RBF                                                                                           |  | $f(x) = \begin{cases} 1 - \frac{x^2}{2} & :  x  \leq 1 \\ \frac{(2- x )^2}{2} & : 1 <  x  < 2 \\ 0 & :  x  \geq 2 \end{cases}$                                                                                              |

参考：

[https://en.wikipedia.org/wiki/Activation\\_function](https://en.wikipedia.org/wiki/Activation_function)

## 1 · sigmoid 函数

$$f(x) = \frac{1}{1 + e^{-x}}$$

也叫 **Logistic 函数**，输出范围在  $(0, 1)$  之间，可以被表示为概率，或者用作数据的归一化（Normalization）。Sigmoid 函数有 2 个问题：

- 软饱和性：当  $x$  趋于无穷时， $f(x)$  的两侧导数逐渐趋向于 0，导致反向传播的梯度非常小，网络很难得到有效训练，这种现象被称为梯度消失。

- 偏置现象：Sigmoid 的输出均大于 0，使得其输出不是 0 均值。

因此现在 sigmoid 经常用在网络最后一层，作为输出层二分类使用，很少用在隐藏层。

## 2 · tanh 函数

$$f(x) = \frac{1 - e^{-2x}}{1 + e^{-2x}}$$

相较于 sigmoid 函数，tanh 函数的输出均值为 0，使得其收敛更快，但是 tanh 函数同样具有软饱和性，从而发生梯度消失。

## 3 · ReLU 函数系列

$$f(x) = \begin{cases} x, & \text{if } x \geq 0 \\ 0, & \text{if } x < 0 \end{cases}$$

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

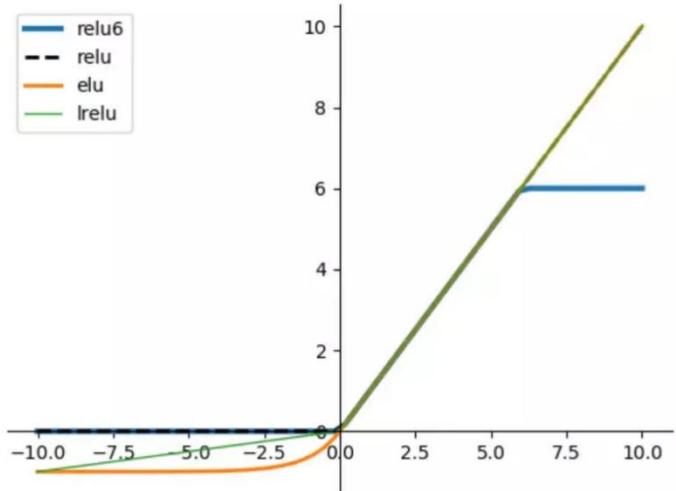
**线性整流单元、修正线性单元 (Rectified Linear Units)**，它的优点有几个：

- 在  $x > 0$  时不存在饱和问题，这让我们无需依赖逐层预训练，在正区间坚决了梯度消失问题
- 计算速度非常快，只需判断是否大于 0
- 收敛速度非常快

缺点有几个：

- 然而随着训练推进，部分输入会落入硬饱和区，导致对应权重无法更新，这被称为“神经元死亡”。
- 与 sigmoid 类似，ReLU 的输出均值也大于 0，所以偏移现象和神经元死亡共同影响网络的收敛性。

ReLU 函数有一系列变体，先上曲线图：



### (1) ReLU6 函数

类似 ReLU 函数，区别是当  $x>6$  时函数值始终取 6。

### (2) Leaky-ReLU 函数

$$f(x) = \begin{cases} x, & \text{if } x \geq 0 \\ \alpha x, & \text{if } x < 0 \end{cases}$$

Leaky-ReLU，为了避免 ReLU 在  $x<0$  时的神经元死亡现象，添加了一个参数  $\alpha$ ，通常  $\alpha$  为 0.01。虽然理论上来看，Leaky-ReLU 有 ReLU 所有的优点，而且还没有 Dead ReLU 问题，但实际使用中并不能证明 ELU 总是好于 ReLU

### (3) ELU 函数

$$f(x) = \begin{cases} x, & \text{if } x \geq 0 \\ \alpha(e^x - 1), & \text{if } x < 0 \end{cases}$$

ELU (Exponential Linear Units) 函数，它结合了 sigmoid 和 ReLU 函数，左侧软饱和，右侧无饱和。和 ReLU 一样，右侧线性部分使得 ELU 能缓解梯度消失，除此之外相较于 ReLU 还有两个优点：

- 左侧软饱和能让对 ELU 对输入变化或噪声更鲁棒。
- ELU 的输出均值接近于 0，所以收敛速度更快。

相较于 ReLU 的缺点是计算量大。同 Leaky ReLU 一样，虽然理论上优于 ReLU，但实际使用中并不能证明 ELU 总是好于 ReLU。

#### (4) softReLU / softplus

也叫 Smooth ReLU 函数，也是对 ReLU 的平滑近似：

$$f(x) = \ln(1 + e^x),$$



#### 4 · softmax 函数

$$S_i = \frac{e^i}{\sum_j e^j}$$

Softmax 函数通常用在分类问题最后一层的激活函数，先对每个  $j$  进行指数幂运算变成非负，然后除以所有项之和归一化。其特点是输出层所有节点的激活值之和为 1，用于表示属于该类别的概率。

#### 5 · softsign 函数

Softsign 函数来自 sign 函数（即正为 1，负为 -1），只是在分母加了个 1，公式为：

$$\text{softsign}(z) = \frac{a}{1 + |a|}$$

就像 Tanh 一样，Softsign 是反对称、去中心、可微分，并返回 -1 和 1 之间的值。其更

平坦的曲线与更慢的下降导数表明它可以更高效地学习。另一方面，导数的计算比 Tanh 更麻烦。

## (八) 损失函数 (Loss Function)

监督式机器学习中，用于衡量预测值和真实值之间的误差的函数，被称为损失函数（Loss Function），损失函数的值越小，说明预测值和真实值之间越接近，模型的预测能力越好。

简单的损失函数大致可分为两类：分类问题的损失函数，和回归问题的损失函数

MSE 和 MAE 是最常用的回归损失函数，交叉熵损失（cross entropy loss）则是最常用的分类问题损失函数。Softmax loss 只是交叉熵损失的另一个名字。

参考：

[https://gombru.github.io/2018/05/23/cross\\_entropy\\_loss/](https://gombru.github.io/2018/05/23/cross_entropy_loss/)

### 1 · MSE (均方误差)

均方误差 MSE (Mean Squared Error)，或者叫 L2 损失，预测值  $y_p$  和真实值  $y$  之差平方的平均值，通常用来做回归问题的损失函数。

$$MSE = \frac{\sum_{i=1}^n (y_i - y_i^p)^2}{n}$$

### 2 · RMSE

均方根误差，(Root Mean Squared Error)，是 MSE 开平方。

$$RMSE = \sqrt{\frac{1}{n} \sum_{i=1}^n (\hat{y}_i - y_i)^2}$$

### 3 · MAE (平均绝对误差)

平均绝对误差 MAE (Mean Absolute Error)，通常用来做回归问题的损失函数

$$MAE = \frac{\sum_{i=1}^n |y_i - \hat{y}_i^p|}{n}$$

由于均方误差（MSE）在误差较大点时的损失远大于平均绝对误差（MAE），它会给异常值赋予更大的权重，模型会全力减小异常值造成的误差，从而使得模型的整体表现下降。

所以当训练数据中含有较多的异常值时，平均绝对误差（MAE）更为有效。当我们对所有观测值进行处理时，如果利用 MSE 进行优化则我们会得到所有观测的均值，而使用 MAE 则能得到所有观测的中值。与均值相比，中值对于异常值的鲁棒性更好，这就意味着平均绝对误差对于异常值有着比均方误差更好的鲁棒性。

但 MAE 也存在一个问题，特别是对于神经网络来说，它的梯度在极值点处会有很大的跃变，即使很小的损失值也会长生很大的误差，这很不利于学习过程。为了解决这个问题，需要在解决极值点的过程中动态减小学习率。MSE 在极值点却有着良好的特性，即使在固定学习率下也能收敛。MSE 的梯度随着损失函数的减小而减小，这一特性使得它在最后的训练过程中能得到更精确的结果

#### 4 · MAPE

平均绝对百分比误差（Mean Absolute Percentage Error），公式如下：

$$MAPE = \frac{100\%}{n} \sum_{i=1}^n \left| \frac{\hat{y}_i - y_i}{y_i} \right|$$

为误差占真实值的百分比，当真实值有数据为 0 时不可用。

#### 5 · sMAPE

对称平均绝对百分比误差（Symmetric Mean Absolute Percentage Error）：

$$SMAPE = \frac{100\%}{n} \sum_{i=1}^n \frac{|\hat{y}_i - y_i|}{(|\hat{y}_i| + |y_i|)/2}$$

当真实值和预测值同时为 0 的时候不可用。

#### 6 · MASE

## 7 · MSIS

## 8 · ND

## 9 · NRMSE

## 10 · OWA

## 11 · Cross-entropy Loss (交叉熵损失)

交叉熵损失 (Cross Entropy loss) 通常用于做分类问题的损失函数。Logistic Loss (逻辑损失)、Softmax Loss 这些都是交叉熵损失的别名而已。

### ● 多分类问题

比如通常的图像分类任务，有若干可能的分类，而每张图片只预测一个分类，所有分类的预测值之和为 1。多分类问题的损失函数，通常为：

$$loss = - \sum_{i=1}^n y_i \log(\hat{y}_i)$$

其中 n 为类别数，而  $y_i$  为真实值， $\hat{y}_i$  为预测值。当 y 为 one-hot 标签时（分类任务通常都是如此），因为除了真实标签之外其余项都为 0，这个函数只剩下了一项：

$$loss = -\log(\hat{y}_i)$$

对应一个 batch 的损失，则为：

$$loss = -\frac{1}{m} \sum_{j=1}^m \sum_{i=1}^n y_{ji} \log(\hat{y}_{ji})$$

m 为 batch 内样本数，n 为类别数

### ● 二分类问题

而对于简单的二分类问题，比如判断图片中是否有枪支，其损失函数由上面的多分类问题的损失函数变为：

$$loss = -y \log(\hat{y}) - (1 - y) \log(1 - \hat{y})$$

其中  $y$  为真实值， $\hat{y}$  为预测值，上式中的  $y$  和  $(1-y)$  相当于多分类中的  $y_i$ ，其和为 1。当真实值为是的时候只有前一项，为否的时候则只有后一项。

### ● 多标签分类问题

多标签分类问题，在实际应用中有点介于图像分类和目标检测任务之间，需要在一张图片中得到若干物体的预测值，因此对于多标签分类问题，一张图片的各个类别预测值之和大于 1（因为一张图片可能有若干物体）。

其单个类别的损失函数可以被视为一个二分类问题的损失函数：

$$loss = -y \log(\hat{y}) - (1 - y) \log(1 - \hat{y})$$

对应每个 batch 的损失为：

$$loss = \sum_{j=1}^m \sum_{i=1}^n -y_{ji} \log(\hat{y}_{ji}) - (1 - y_{ji}) \log(1 - \hat{y}_{ji})$$

$m$  为 batch 内样本数， $n$  为类别数

## 12 · Softmax Loss

严格来说，并没有一个被称为 softmax loss 的学术定义。Softmax loss 指的就是 softmax + cross entropy loss。

通常情况下，分类问题的最后一层 FC 层用 softmax 函数作为激活函数（或者有的人将之视为两层），FC 层出来的是特征值，再经过 softmax 出来的是分类概率，概率被作为输入传给损失函数 cross entropy loss。通常情况下 softmax 和 cross entropy 是成对出现的，这两个函数组合被定义为 softmax loss。

## 13 · Triplet Loss (FaceNet)

## 14 · Contrastive Loss (TODO)

## 15 · Center Loss (2016)

Center loss 是 2016 年的一篇论文，提出了一种新的损失函数 center loss，用于提高 open-set 情况下的人脸识别的准确率。

Close-set 的人脸识别相当于一个分类问题，通常采用 softmax loss 作为损失函数，这在 close-set 中性能良好，但在 open-set 情况下，模型性能会急剧下降。

一个直观的感觉是：如果模型学到的特征判别度更高，那么遇到没见过的数据时，泛化性能会比较好。为了使得模型学到的特征判别度更高，论文提出了一种新的辅助损失函数 center loss，之所以说是辅助损失函数是因为新提出的损失函数需要结合 softmax 交叉熵一起使用，而非替代后者。

文中对每个类别都定义了一个 center，其维度与特征值的维度相同，这些 center 是可以被学习的。

Softmax loss 的定义为：

$$\mathcal{L}_S = - \sum_{i=1}^m \log \frac{e^{W_{y_i}^T \mathbf{x}_i + b_{y_i}}}{\sum_{j=1}^n e^{W_j^T \mathbf{x}_i + b_j}}$$

Center loss 的函数定义为：

$$\mathcal{L}_C = \frac{1}{2} \sum_{i=1}^m \|\mathbf{x}_i - \mathbf{c}_{y_i}\|_2^2$$

总的损失函数为 softmax loss 和 center loss 的加权和，其中  $\lambda$  被用来平衡两个损失函数：

$$\begin{aligned} \mathcal{L} &= \mathcal{L}_S + \lambda \mathcal{L}_C \\ &= - \sum_{i=1}^m \log \frac{e^{W_{y_i}^T \mathbf{x}_i + b_{y_i}}}{\sum_{j=1}^n e^{W_j^T \mathbf{x}_i + b_j}} + \frac{\lambda}{2} \sum_{i=1}^m \|\mathbf{x}_i - \mathbf{c}_{y_i}\|_2^2 \end{aligned}$$

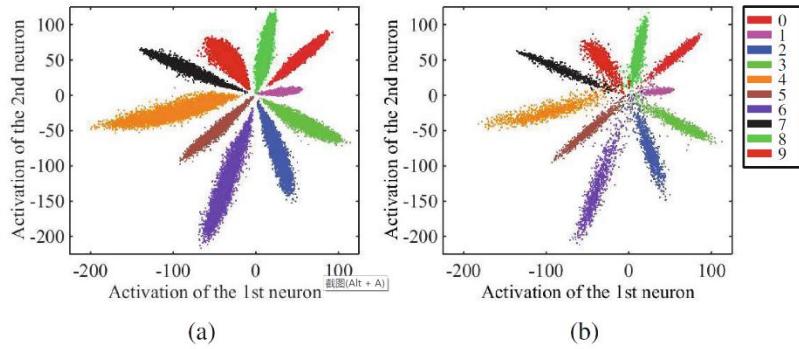
使用两个损失函数的加权和是因为：

- 如果没有 center loss，各个类别的类内变化会比较大（即特征判别度低，feature 不够 discriminative，这样就不能很好的面对陌生数据），也就是说 center loss 用于

保证特征值是 discriminative 的

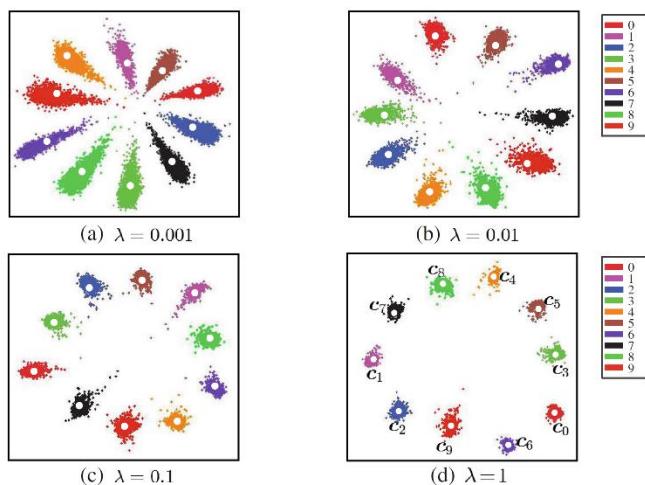
- 而如果没有 softmax loss，那么 center 和特征值都会降到 0（这样 center loss 会非常小），各类别的区分度不够，也就是说 softmax loss 是用于保证特征值 separable 的。

文中使用了一个修改过的 LeNet 提取特征，这个 LeNet 输出的特征只有 2 个，这是为了能在二维坐标系中直观的表示出来，下图为通过传统的 softmax loss 得到的 10 个类别的样本的特征值在二维坐标系中的分布（x 轴和 y 轴分别为两个特征值），左边为训练集，右边为测试集：



**Fig. 2.** The distribution of deeply learned features in (a) training set (b) testing set, both under the supervision of softmax loss, where we use 50K/10K train/test splits. The points with different colors denote features from different classes. **Best viewed in color.** (Color figure online)

而以下是在不同  $\lambda$  的情况下，用组合损失函数习得的特征值分布：



**Fig. 3.** The distribution of deeply learned features under the joint supervision of softmax loss and center loss. The points with different colors denote features from different classes. Different  $\lambda$  lead to different deep feature distributions ( $\alpha = 0.5$ ). The white dots ( $c_0, c_1, \dots, c_9$ ) denote 10 class centers of deep features. **Best viewed in color.** (Color figure online)

而 contrastive loss 和 triplet loss，这两个损失函数的缺点是在样本空间变大的时候，样本对 (sample pair) 和样本三元组 (sample triplet) 的量会变得非常巨大

论文：

A Discriminative Feature Learning Approach for Deep Face Recognition



1600\_CenterLoss  
.pdf

## 16 · A-Softmax Loss (SphereFace)

**A-Softmax Loss** 是在 SphereFace 论文中提出的一种损失函数，A 代表 Angular。

对于一个 2 分类的情况，最后一个 FC 层的 2 维输出（没过 softmax 激活之前）分别是  $W_1^*x+b_1$  和  $W_2^*x+b_2$ ，预测分类是这两者之中大的那个。也就是说 softmax loss 函数的判断边界是  $(W_1 - W_2)x + b_1 - b_2 = 0$ 。如果我们令  $\|W_1\| = \|W_2\| = 1$  且  $b_1 = b_2 = 0$ ，则判断边界就变为  $\|\mathbf{x}\|(\cos \theta_1 - \cos \theta_2) = 0$ ，其中  $\theta_i$  为  $W_i$  和  $x$  的夹角，于是判断边界变为仅依赖于两个夹角  $\theta_1$  和  $\theta_2$ ，直观的理解，就是  $x$  离哪个  $W_i$  更近，则被分为哪个类。

A-Softmax Loss 还加入了一个正整数  $m$  ( $m \geq 1$ ) 来量化的控制判断边界，于是类别 1 的判断边界收缩为  $\|\mathbf{x}\|(\cos(m\theta_1) - \cos(\theta_2)) = 0$ ，类别 2 的为  $\|\mathbf{x}\|(\cos \theta_1 - \cos m\theta_2) = 0$ 。更高的  $m$  会约束每个类别的角度  $\theta$  变得更小，从而使得 inter-class 距离更大，而 intra-class 距离更小。我们可以得出  $m$  的下限，以满足对 open-set 人脸识别的准则：最大的 intra-class 距离要小于最小的 inter-class 距离。

下图中为这二分类神经网络在特征值为二维的情况下（为了方便显示），使用三种不同的损失函数在欧式空间和角空间得到的结果：最左边是原始的 softmax loss，中间两幅图是改进后的 softmax loss（即判断边界改为  $\|\mathbf{x}\|(\cos \theta_1 - \cos \theta_2) = 0$ ），最右两幅图是增加了控制变量  $m$  的结果。

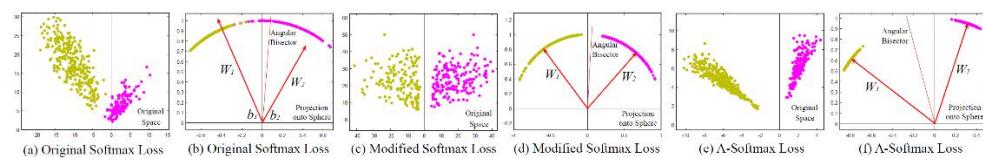


Figure 2: Comparison among softmax loss, modified softmax loss and A-Softmax loss. In this toy experiment, we construct a CNN to learn 2-D features on a subset of the CASIA face dataset. In specific, we set the output dimension of FC1 layer as 2 and visualize the learned features. Yellow dots represent the first class face features, while purple dots represent the second class face features. One can see that features learned by the original softmax loss can not be classified simply via angles, while modified softmax loss can. Our A-Softmax loss can further increase the angular margin of learned features.

A-Softmax Loss 普及到三维特征值，则为一个球面中，二个不同的向量形成的圆分别代表两个不同的分类。下图为二分类神经网络在特征值为二维及三维的情况下，三种不同类型的损失函数在欧式空间和角空间形成的分类结果：两种颜色表示不同分类，上面三个是特

征值为 2D，下面三个是特征值为 3D，最左边两个是使用欧式边缘损失函数，比如 contrastive loss，triplet loss (facenet)，center loss；中间两个是改进后的 softmax loss；最右边两个则是 A-Softmax Loss 的情形：

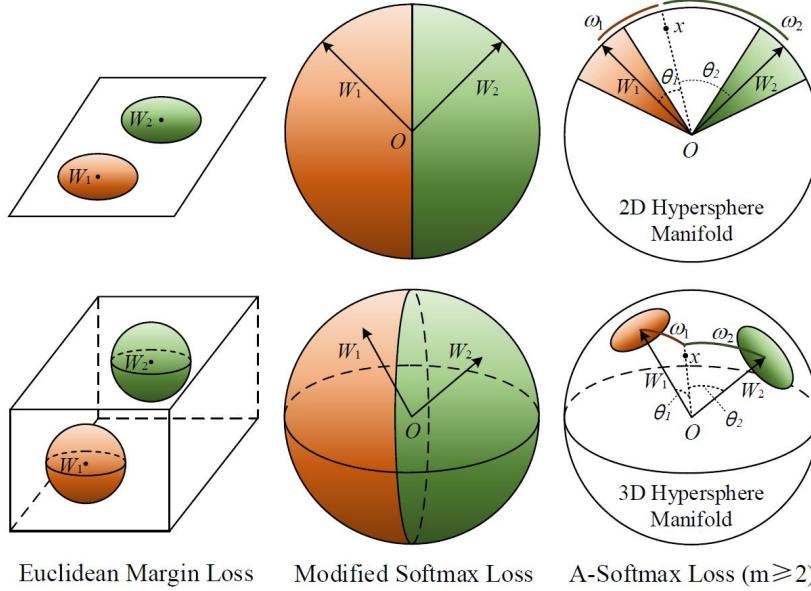


Figure 3: Geometry Interpretation of Euclidean margin loss (e.g. contrastive loss, triplet loss, center loss, etc.), modified softmax loss and A-Softmax loss. The first row is 2D feature constraint, and the second row is 3D feature constraint. The orange region indicates the discriminative constraint for class 1, while the green region is for class 2.

## 17 · Focal Loss (RetinaNet)

Focal Loss 是在 RetinaNet 中被提出来的，是为了应对在单步检测网络（比如 SSD 和 YOLO）中存在的问题：

- 由于正负样本比例严重失衡（网络产生的 bbox 预测大部分为 false negative），大量的 negative example 的损失淹没了 positive 的 loss。
- 大多 negative example 不在前景和背景的过渡区域上，分类很明确（这种易分类的 negative 称为 easy negative），训练时对应的背景类 score 会很大，换个角度看就是单个 example 的 loss 很小，反向计算时梯度小。梯度小造成 easy negative example 对参数的收敛作用很有限，我们更需要 loss 大的对参数收敛影响也更大的 example，即 hard positive/negative example。

这篇论文调整了 loss function，通过增加权重系数来解决这个问题。

传统的二分类交叉熵损失函数 CE 为：

$$\text{CE}(p, y) = \begin{cases} -\log(p) & \text{if } y = 1 \\ -\log(1 - p) & \text{otherwise.} \end{cases}$$

如果定义  $p_t$  为：

$$p_t = \begin{cases} p & \text{if } y = 1 \\ 1 - p & \text{otherwise,} \end{cases}$$

则有：

$$\text{CE}(p, y) = \text{CE}(p_t) = -\log(p_t).$$

通过在 CE 上增加权重系数，Focal Loss 函数 FL 被定义为：

$$\text{FL}(p_t) = -\alpha_t(1 - p_t)^\gamma \log(p_t).$$

其中  $\gamma$  是个大于 0 的值， $\alpha_t$  是个  $[0, 1]$  间的小数， $\gamma$  和  $\alpha_t$  都是固定值，不参与训练。

- 无论是前景类还是背景类， $p_t$  越大，权重  $(1-p_t)$  就越小。也就是说 easy example 可以通过权重进行抑制。换言之，当某样本类别比较明确些，它对整体 loss 的贡献就比较少；而若某样本类别不易区分，则对整体 loss 的贡献就相对偏大。这样得到的 loss 最终将集中精力去诱导模型去努力分辨那些难分的目标类别，于是就有效提升了整体的目标检测准确度。
- $\alpha_t$  用于调节 positive 和 negative 的比例，前景类别使用  $\alpha_t$  时，对应的背景类别使用  $1 - \alpha_t$ ；
- $\gamma$  的作用也是抑制 easy example 对整体 loss 的贡献
- $\gamma$  和  $\alpha_t$  的最优值是相互影响的，所以在评估准确度时需要把两者组合起来调节。作者在论文中给出  $\gamma=2$ 、 $\alpha_t=0.25$  时，ResNet-101+FPN 作为 backbone 的结构有最优的性能。

假设一个 easy 正样本， $p$  为 0.8 ( $p_t=p=0.8$ )，另外有个 hard 正样本， $p$  为 0.4 ( $p_t=p=0.4$ )，在 CE 中，两者对最终 loss 的贡献大约是 1 : 4 ( $\log 0.8$  约等于  $1/4$  的  $\log 0.4$ )，而在 FL 中，通过  $(1-p_t)^\gamma$ ，两者的贡献比例被放大为： $(1-0.8)^2 : (1-0.4)^2 * 4$ ，约等于 1 : 36

论文：

<https://arxiv.org/pdf/1708.02002.pdf>

参考：

<https://zhuanlan.zhihu.com/p/59910080>

## (九) 优化方法 (Optimizer)

## 1 · 梯度下降法 (GD)

梯度下降法 (Gradient Descent, GD) , 最基础的参数优化算法, 计算数据集中样本的梯度, 然后执行决策 (沿着梯度相反的方向更新参数), 重复这个步骤从而逐渐靠近最优解。

根据每次更新前计算的样本数量不同, GD 可以分为 :

### (1) SGD

**Stochastic GD**, 随机梯度下降法, 也叫 One-Sample GD, 每次计算一个随机样本, 然后更新参数。

这种算法的优点包括 :

- 易于理解和实现
- 快速的更新频率使得在某些问题上学习速度很快
- 随机样本的噪声使得模型更易避开局部最小值

缺点 :

- 更新次数太过频繁导致计算量更大
- 随机样本的噪声同样会导致不易稳定在全局最小值

#### 混淆注意 !

由于实际应用中较少使用 One-Sample GD, 而经常使用 Mini-Batch GD, SGD 这个词经常被用于指代 Mini-Batch GD。

### (2) BGD

Batch Gradient Descent, 每次计算所有的样本, 然后更新参数。

这种算法的优点是 :

- 计算的效率比 SGD 更高
- 更少的更新次数带来更稳定的收敛
- 计算更加并行, 方便利用硬件的并行能力

缺点是 :

- 会落入局部极小值的点出不来
- 需要额外的算力来将所有样本的 loss 相加
- 对内存要求太高, 通常需要将整个训练集加载到内存
- 对于大型数据集来说, 训练速度会变得很慢

### (3) MBGD

MBGD，Mini-batch GD，介于 SGD 和 BGD 之间，将训练集随机分割为若干个 Batch，每次计算一个 Batch，然后更新参数。

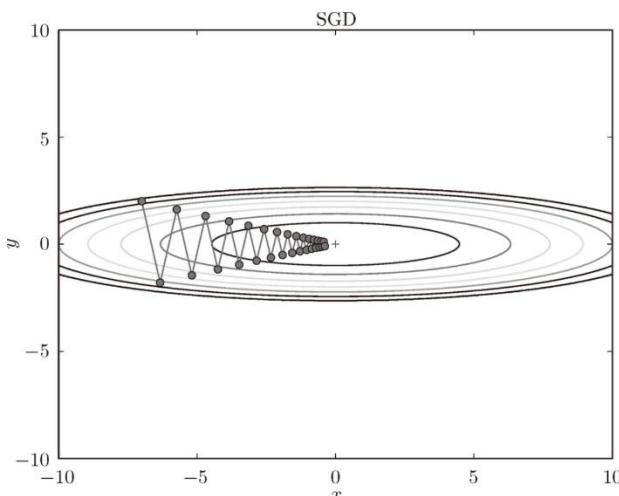
可以看出 MBGD 综合了 SGD 和 BGD 的优缺点，优点是：

- 比 SGD 计算更有效率，更快的收敛速度
- 比 BGD 更鲁棒的收敛，避免了局部极小值
- 在利用了硬件并行算力的同时又无须将整个训练集加载进内存

缺点：

- 需要一个额外的超参数“mini-batch size”
- 也综合了 SGD 和 BGD 的缺点，但总体来说比两者更平衡

MBGD 也是最常用的 GD 方法，同时也是最常用的优化方法，经常被误称为 SGD。



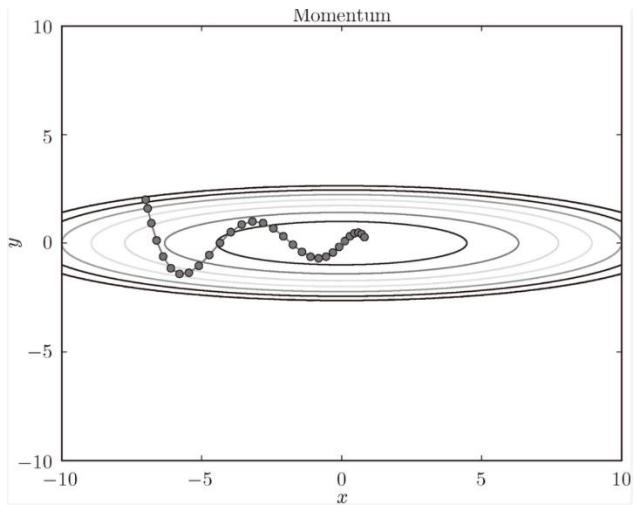
## 2 · 动量法 (Momentum)

Momentum，动量法，公式如下：

$$\mathbf{v} \leftarrow \alpha \mathbf{v} - \eta \frac{\partial L}{\partial \mathbf{W}}$$

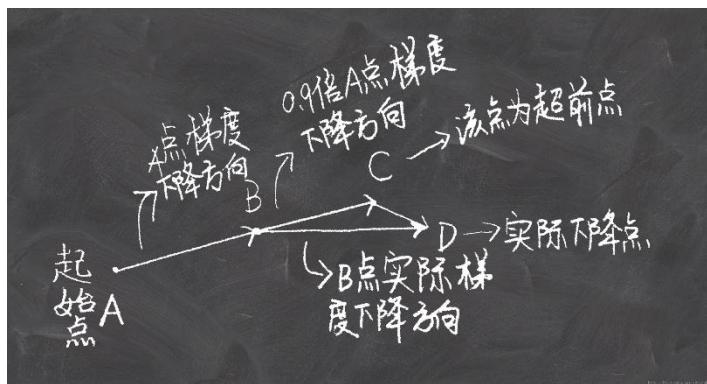
$$\mathbf{W} \leftarrow \mathbf{W} + \mathbf{v}$$

相较于 SGD，增加了一个惯性的概念  $\alpha \mathbf{v}$ ，动量法给人的感觉就好像是小球在函数曲面上滚动，滚动方向，也就是每一次更新的方向都是由惯性（之前的梯度方向残留）和受力方向（本次的梯度方向）组合而成。而相较于此，之前的 GD 则像是小球每次更新到一个新的位置之后立刻静止，只受当前的梯度方向影响。



### 3 · NAG

Nesterov Accelerated Gradient，NAG 和动量法类似，也是“历史梯度+当前梯度”的方法，其区别在于其“当前梯度”采用的不是当前点的梯度，而是当前点在历史梯度上前进之后的超前点的梯度，见下图：



与之相比，动量法如下：



上两图中，A 为起始点，AB 为起始点的梯度方向，B 为当前点，下一个点的位置分别为 D（上图，NAG）和 C（下图，动量法），而 AD/AC 都是 AB 和另一个向量之和，区别在于下图的动量法使用的是当前点 B 的梯度，而上图的 NAG 使用的是超前点 C 的梯度方向 CD。

## 4 · AdaGrad

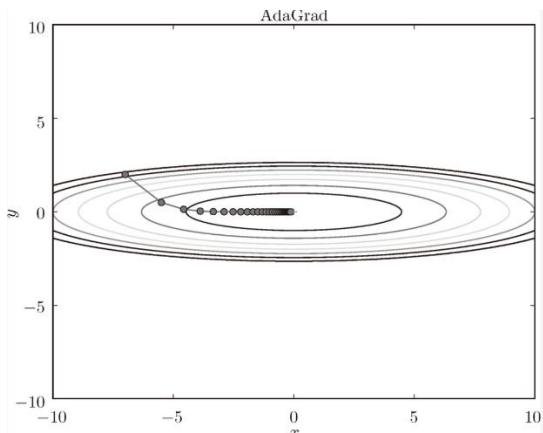
在有关学习率的技巧中，有一种被称为学习率衰减的方法，随着学习的进行，使学习率逐渐减小。而 AdaGrad 方法则进一步发展了这个办法，针对每一个参数，适当的调整学习率，于此同时进行学习。

$$\mathbf{h} \leftarrow \mathbf{h} + \frac{\partial L}{\partial \mathbf{W}} \odot \frac{\partial L}{\partial \mathbf{W}}$$

$$\mathbf{W} \leftarrow \mathbf{W} - \eta \frac{1}{\sqrt{\mathbf{h}}} \frac{\partial L}{\partial \mathbf{W}}$$

公式中  $\mathbf{W}$  表示权重， $\frac{\partial L}{\partial \mathbf{W}}$  表示梯度， $\eta$  表示学习率，新的变量  $\mathbf{h}$  保存了之前所有的梯度的平方和。这样有以下效果：

- 变动大的参数的学习率下降得更快
- 随着梯度的更新，学习率将逐渐变小

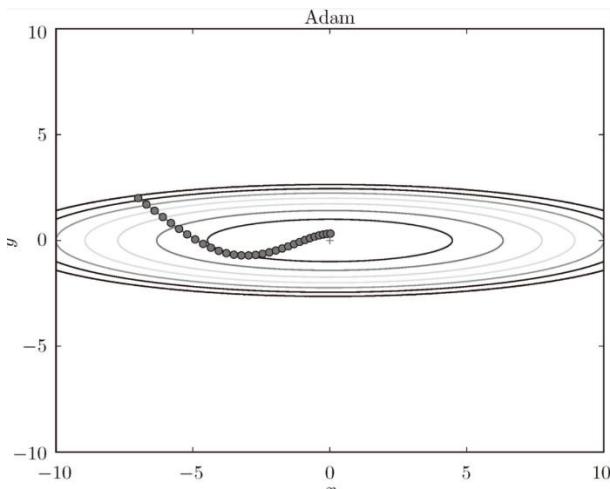


## 5 · RMSprop

AdaGrad 有一个问题，就是学习率总会逐渐地变小（因为  $\mathbf{h}$  会持续的变大），导致最终学习率会降到 0 而无法学习，RMSprop 相较于 AdaGrad 的改进就是添加了一个衰减率，使得  $\mathbf{h}$  不会因为累积而变得太大。

## 6 · Adam

Adam 方法融合了 Momentum 和 AdaGrad 两种方法：



## (十) 标准化方法 (Normalization)

标准化，或者规范化，归一化（Normalization）是神经网络中一种常见的处理方式，可以将数据标准化，其目的是为了增强网络的泛化能力，降低网络的过拟合。

Normalization 的目的是将数据标准化。具体来说，是将数据按行、按列、或者某种维度，映射到一个特定的区间（比如-1 到 1）或者某种特定分布里（比如均值为 0, 方差为 1），使得样本间、批次间、或者该维度间的差距变小，更加趋同。

因此 Normalization 指代两种概念，一种仅指映射的算法，比如 L1N、L2N、min-max normalization、Z-score normalization，另一种则是指代更具体的方法，不仅包括算法，还包括所取的维度，比如：

BN：其标准化的维度 A 为单个批次内，从而使得批次间的数据更加趋同。

LN：其标准化的维度为单个样本内的不同通道，缩小样本间的差异。

IN：标准化的维度为单个通道内的，缩小通道间的差异。

Normalization 这个名词在很多地方都会出现，但是对于数据却有两种容易混淆的处理过程。对于某个多特征的机器学习数据集来说，第一种 Normalization 是对于将数据进行预处理时进行的操作，是对于数据集的各个特征分别进行处理，主要包括 min-max

normalization、Z-score normalization、log 函数转换、atan 函数转换、LN、IN 等。

第二种 Normalization 对于每个样本缩放到单位范数（每个样本的范数为 1），主要有 L1-normalization、L2-normalization、BN 等，可以用于 SVM 等应用

换句话说，第一种类型的 Normalization 的目的是缩小特征间的差异，也叫 Feature Normalization，或者 Feature Scaling。在将输入的多种特征进行趋同化和无量纲化，比如将两个输入特征，第一个高度特征，范围是-10000 米到+30000 米，另一个角度特征，范围是 0 度到 3 度之间。

而通常所指的 Normalization 指的是第二种，其目的是缩小样本间的差异。对于每个样本缩放到单位范数（每个样本的范数为 1）。

参考：

一文搞定深度学习中的规范化 BN,LN,IN,GN,CBN

<https://zhuanlan.zhihu.com/p/115949091>

## 1 · LRN (AlexNet)

局部响应归一化层 (Local Response Normalization)，LRN 的定义如下：

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

具体不做解释，可以参考 AlexNet 的论文。现今的神经网络中已经很少用 LRN，标准化大都通过 BN 层等来实现。

## 2 · Min-max Normalization

对原始数据线性变换，映射到 [0,1] 区间，公式如下：

$$x^* = (x - \text{min}) / (\text{max} - \text{min})$$

## 3 · Z-score Normalization

也叫标准差标准化，这种方法给予原始数据的均值 (mean) 和标准差 (standard deviation) 进行数据的标准化。经过处理的数据符合标准正态分布，即均值为 0，标准差为 1，其转化函数为：

$$x^* = (x - \mu) / \sigma$$

## 4 · L1 Normalization

L1 Normalization，L1 标准化，即向量中的每个元素除以向量的 L1 范数（向量各元素的绝对值之和）

## 5 · L2 Normalization

L2 Normalization，L2 标准化，即向量中的每个元素除以向量的 L2 范数（向量各元素的平方和的平方根）。

## 6 · Batch Normalization (201502)

Batch Normalization，BN，批标准化，是目前最常用的标准化方法。

以进行学习时的 mini-batch 为单位，进行使数据均值为 0 方差为 1 的正规化。优点是可以增加学习速度，不那么依赖初始值，抑制过拟合。

BN 的提出是基于小批量随机梯度下降 (mini-batch SGD) 的。随机梯度下降的缺点是对参数比较敏感，较大的学习率和不合适的初始化值均有可能导致训练过程中发生梯度消失或者梯度爆炸的现象的出现。BN 的出现则有效的解决了这个问题。

参考：

<https://zhuanlan.zhihu.com/p/54171297>

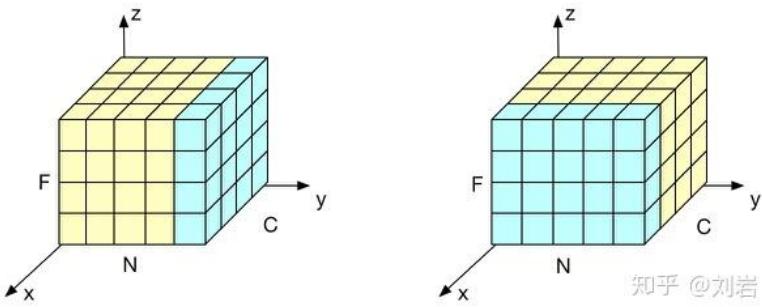
<https://www.cnblogs.com/guoyaohua/p/8724433.html>

论文：

<https://arxiv.org/pdf/1502.03167.pdf>

## 7 · Layer Normalization (201607)

BN 并不适用于 RNN 等动态网络和 batchsize 较小的时候效果不好。**Layer Normalization (LN)** 的提出有效的解决 BN 的这两个问题。LN 和 BN 不同点是归一化的维度是互相垂直的，如下图所示。图中 N 表示样本轴，C 表示通道轴，F 是每个通道的特征数量。BN 如右侧所示，它是取不同样本的同一个通道的特征做归一化；LN 则是如左侧所示，它取的是同一个样本的不同通道做归一化。



左为 LN，右为 BN

参考

<https://zhuanlan.zhihu.com/p/54530247>

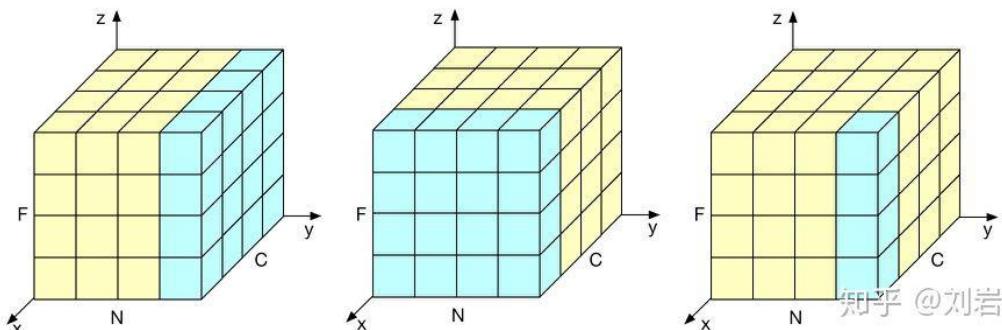
论文：

<https://arxiv.org/pdf/1607.06450.pdf>

## 8 · Instance Normalization (201607)

对于图像风格迁移这类的注重每个像素的任务来说，每个样本的每个像素点的信息都是非常重要的，于是像 BN 这种每个批量的所有样本都做归一化的算法就不太适用了，因为 BN 计算归一化统计量时考虑了一个批量中所有图片的内容，从而造成了每个样本独特细节的丢失。同理对于 LN 这类需要考虑一个样本所有通道的算法来说可能忽略了不同通道的差异，也不太适用于图像风格迁移这类应用。

所以这篇文章提出了 **Instance Normalization (IN)**，一种更适合对单个像素有更高要求的场景的归一化算法（IST，GAN 等）。IN 的算法非常简单，计算归一化统计量时考虑单个样本，单个通道的所有元素。



IN (右) 和 BN (中) 以及 LN (左) 的不同从图中可以非常明显的看出。

参考：

<https://zhuanlan.zhihu.com/p/56542480>

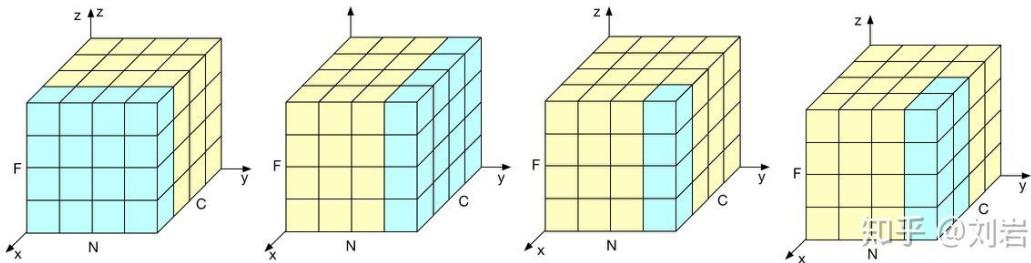
论文：

<https://arxiv.org/pdf/1607.08022.pdf>

## 9 · Group Normalization (201803)

**Group Normalization (GN)** 是针对 **Batch Normalization (BN)** 在 batch size 较小时错误率较高而提出的改进算法，因为 BN 层的计算结果依赖当前 batch 的数据，当 batch size 较小时（比如 2、4 这样），该 batch 数据的均值和方差的代表性较差，因此对最后的结果影响也较大。

Group Normalization (GN) 是何恺明提出的一种归一化策略，它是介于 Layer Normalization (LN) 和 Instance Normalization (IN) 之间的一种折中方案，下图最右。它通过将通道数据分成几组计算归一化统计量，因此 GN 也是和批量大小无关的算法，因此可以用在 batchsize 比较小的环境中。作者在论文中指出 GN 要比 LN 和 IN 的效果要好。



图中从左至右依次是：BN、LN、IN、GN

参考：

<https://zhuanlan.zhihu.com/p/56613508>

<https://blog.csdn.net/u014380165/article/details/79810040>

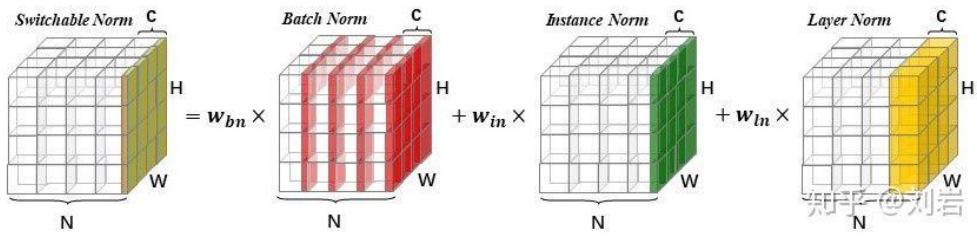
论文：

<https://arxiv.org/pdf/1803.08494.pdf>

## 10 · Switchable Normalization (201806)

虽然 BN、LN、IN 这些归一化方法往往能提升模型的性能，但是当你接收一个任务时，具体选择哪个归一化方法仍然需要人工选择，这往往需要大量的对照实验或者开发者优秀的经验才能选出最合适归一化方法。本文提出了 Switchable Normalization (SN)，它的算法核心在于提出了一个可微的归一化层，可以让模型根据数据来学习到每一层该选择的归一化方法，亦或是三个归一化方法的加权和，如下图所示。所以 SN 是一个任务无关的归一化方法，不管是 LN 适用的 RNN 还是 IN 适用的图像风格迁移 (IST)，SN 均能用到该应用

中。作者在实验中直接将 SN 用到了包括分类，检测，分割，IST，LSTM 等各个方向的任务中，SN 均取得了非常好的效果。



参考：

<https://zhuanlan.zhihu.com/p/57807576>

论文：

<https://arxiv.org/pdf/1806.10779.pdf>

## 七、研究方向：图像

本章介绍了 NN 在图像这个研究方向的概念、分支、使用的技术、技术的演进、使用的数据集等等。

在 NN 兴起之前，人工智能各个领域被很多非 NN 的技术占据，整体而言，在所有这些研究方向里，大致呈现几个趋势：

1. 其他技术渐渐被 NN 取代，有的早期实现是 NN 和其他技术共存，随着发展，NN 渐渐替代了其他部分的非 NN 技术，在很多现有的前沿技术已经全由 NN 构成，输入输出全部实现端到端。
2. 在各个数据集的准确率在逐渐上升
3. 有很多技术被发展出来用于在相同效率的情况下减少计算量

目前在业界有 2 种相反的路线观点，一种观点认为，只要有足够的算力，所有的问题都可以用足够强大的端到端神经网络解决。另一个观点认为，无论算力发展到多强大，NN 仍然不是强人工智能的最终解决方案，因此仍然需要部分人工的介入。

推荐一个网站：[www.paperswithcode.com](http://www.paperswithcode.com)。该网站中包含了很多的研究方向：有论文有代码有排行榜，并且根据网站介绍，该网站是自动化更新的。

目前的深度学习，最大应用方向（一多半的论文都在这个方向）毫无疑问是 CV（计算机视觉），剩下的应用里，NLP（自然语言处理）又占了一大部分，其余包括医疗、打游戏、机器人、音乐等等。从总体比例来看，差不多“NN 研究共一石，CV 独得八斗，NLP 得一斗，其他方向共分一斗”。

而 CV 其中又划分了很多的子类及更小的细分方向，其中几个比较大的，也是实用性比较强的子类为：

- 图像理解
- 视频理解
- 人脸识别（及相关）
- 行人识别（及相关）

图像理解里包括了几个基本且主要的方向：

- 图像分类：Image Classification，解决了 What 的问题
- 目标检测：Object Detection，解决了 What 和 Where（边界框）
- 图像分割：Image Segmentation，解决了 What 和 Where（像素级别），分为：
  - 语义分割：Semantic Segmentation，按同类物体分割
  - 实例分割：Instance Segmentation，按单个物体分割
  - 全景分割：Panoptic Segmentation，前景实例分割，背景语义分割
- 图像描述：Image Caption，根据图像生成描述

- 图像问答：Image QA，图像理解的终极王者，给定图像和问题，NN 给出回答，比如“图中几个人的衣服分别是什么颜色”

图像分类（MNIST，ImageNet）为 NN 的发源地。在大多数研究中，CNN 是作为一个基本结构出现的（一般称为 Backbone，骨干网络，基础网络），比如在目标检测（Object Detection）中的 Faster R-CNN 网络结构，其中用到了 2 个子 CNN 网络。

对于每一个具体的实现而言，包括以下几个因素：

- **数据集**：数据集有很多种，每个数据集基本框定了可以进行的运算（比如著名的 ImageNet 和 LFW）
- **任务**：在同一个数据集中可能可以实现若干个研究任务（比如 LFW 可以进行人脸识别和人脸识别），对于一个任务，最好有一个对应的衡量标准（Metrics，比如 ImageNet 图像分类任务，衡量标准是前一和前五的准确率），衡量标准可能没有。
- **网络结构**：网络结构是网络的基础，一个创新的网络结构通常会有一篇论文来描述，而一个网络结构也可能通过不同实现达到 N 种研究目的（比如 FaceNet 实现人脸验证和人脸识别）
- **实现**：网络结构的具体实现，可能是代码、超参数设置、或者更具体的预训练好的权重

因此对于一个具体的网络，你可能可以找到：

- 一个预训练好的，针对某一数据集，某一目的的，可以直接使用的网络
- 一个开源的网络结构，没有权重，可能有训练方法或者超参数
- 一篇论文，其中描述了网络结构，没有任何开源实现

以下各个网络模型后的时间表示其论文发表的时间，该时间晚于其提出时间。

## （一）图像分类（Image Classification）

在各个研究方向中，**图像分类（Image Classification）** 是比较特别的一支。整个神经网络的复兴，就是起源于 2012 年 AlexNet 在 ImageNet 图像分类赛中的君临天下，此后 CNN 和图像分类也成了神经网络/深度学习的最基础的分支。

对于基础 CNN 网络（也就是单一的 CNN，没有额外的复合架构）的技术演进，图像分类的准确率也是验证其性能的最主要手段。而很多其他的研究方向，要么就是采用基础 CNN 网络的变体，要么就是采用包含了基础 CNN 的复合架构。

因此这一节中描述了关于图像分类的 CNN，都是著名且基础的 CNN。之所以**著名**，是因为这些网络模型的提出带有自己的创新发明，或者是独特的层结构，或者是创新的思想。之所以**基础**，是因为这些都是传统的卷积神经网络，虽然主要的工作目的是图像分类，但是可以被整合在完成其他功能的更复杂的神经网络中，作为其中的一部分（通常被称为其 backbone，骨干网络，基础网络）。

这些基础 CNN 的大致结构都差不多：前面的卷积层+后面的分类器，分类器通常由全局池化层+FC 层+Softmax 三部分组成。

## 1 · 衡量标准

### (1) Top-N 错误率

**top-N error rate**，对图像分类的预测结果是按照概率排序的，top-1 错误率表示仅看第一个时的错误率，top-5 表示预测结果中排前五的包含正确标签的错误率。top-1 和 top-5 是常用的两个标准。

### (2) TPR / FPR

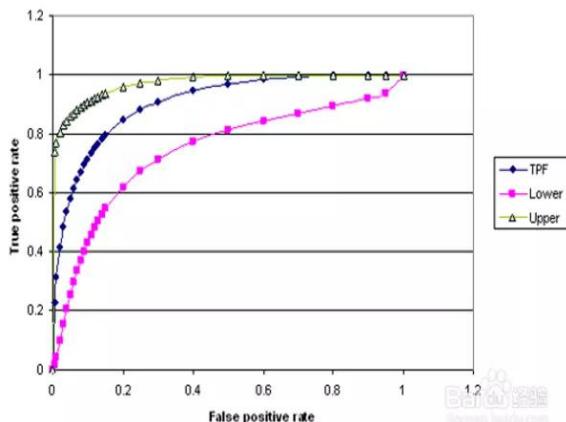
可参考《[概念定义 > NN 相关 > TP/FP/TN/FN](#)》

**TPR (True Positive Ratio)**，也叫查全率，召回率 (Recall)，hit rate，或者 **Sensitivity**，等于  $TP/(TP+FN)$ ，用于衡量所有实际为阳性的样本的检测准确率。

**FPR (False Positive Ratio)**，等于  $FP/(FP+TN)$ ，等于  $1-TNR$ ，用于衡量所有实际为阴性样本的检测错误率。

### (3) ROC 曲线 / AUC 值

**TPR** 和 **TNR** 通常在不同阈值的情况下负相关，TPR 越高，TNR 越低。因此就有了 **ROC 曲线 (ROC Curve)**，以 FPR 为 x 轴，TPR 为 y 轴，ROC 曲线越靠近  $(0, 1)$  点，说明分类器能力越强。



也可以用 AUC (Aera Under Curve) , 即 ROC 曲线下的面积来衡量分类器的能力。

## 2 . 数据集

以下列出部分用于图像分类的数据集，或者多用途数据集的图像分类部分，此外目标检测和图像分割的数据集通常也可以用作图像分类任务，请自行参考。

### ( 1 ) MNIST

MNIST=Modified National Institute of Standard and Technology。

经典的手写数字低分辨率 (28\*28) 图片数据集，图像分类领域的 Hello World，包括超过 6 万张训练图像和 1 万张测试图像。

### ( 2 ) CIFAR-10

CIFAR=Canadian Institute For Advanced Research。

分为 CIFAR-10 和 CIFAR-100。由 Alex Krizhevsky (AlexNet 的作者) 2009 年推出。

CIFAR-10 包括了 6 万张分辨率为 32\*32 的图片，被分为 10 类，每类 6000 张图片。这 10 类为：飞机、小轿车、鸟、猫、鹿、狗、青蛙、马、船和卡车。

CIFAR-100 类似，但被分为了 100 类。

### ( 3 ) ImageNet

这应该是整个神经网络/计算机视觉界最有名的数据集。

由李飞飞创建的庞大的可视化数据集，超过 1400 万图片被人工标注出包含什么物体，至少 100 万图片包含了标注框 (box)，超过 2 万个类别。可以被用于衡量/训练图像分类任务，以及目标检测任务。

官网：

<http://image-net.org>

下载：

<http://image-net.org/download.php>

ImageNet 中的图片是按照 WordNet3.0 进行组织的，对每一个图片类 (synset)，用 wnid (WordNet ID) 标示，比如“军装”的 wnid 是 n03763968。

- synset 的页面：

[http://www.image-net.org/synset?wnid=\[wnid\]](http://www.image-net.org/synset?wnid=[wnid])

- synset 的直接子类：

[http://www.image-net.org/api/text/wordnet.structure.hyponym?wnid=\[wnid\]](http://www.image-net.org/api/text/wordnet.structure.hyponym?wnid=[wnid])

- synset 的所有子类 (直接和间接)：

[http://www.image-net.org/api/text/wordnet.structure.hyponym?wnid=\[wnid\]&full=1](http://www.image-net.org/api/text/wordnet.structure.hyponym?wnid=[wnid]&full=1)

- synset 的名称：

[http://www.image-net.org/api/text/wordnet.synset.getwords?wnid=\[wnid\]](http://www.image-net.org/api/text/wordnet.synset.getwords?wnid=[wnid])

- synset 包含的所有图片的 URL：

[http://www.image-net.org/api/text/imagenet.synset.geturls?wnid=\[wnid\]](http://www.image-net.org/api/text/imagenet.synset.geturls?wnid=[wnid])

- synset 包含的所有图片的图片名和 URL 映射

[http://www.image-net.org/api/text/imagenet.synset.getmapping?wnid=\[wnid\]](http://www.image-net.org/api/text/imagenet.synset.getmapping?wnid=[wnid])

- 所有 synset 的 wnid 和名称的映射：

<http://image-net.org/archive/words.txt>

- 所有 synset 的 wnid 列表

[http://www.image-net.org/api/text/imagenet.synset.obtain\\_synset\\_list](http://www.image-net.org/api/text/imagenet.synset.obtain_synset_list)

- 所有 synset 的 wnid 和解释的映射

<http://image-net.org/archive/gloss.txt>

- WordNet 中的所有的 synset 父类和子类的关系映射 (wnid-wnid)

[http://www.image-net.org/archive/wordnet.is\\_a.txt](http://www.image-net.org/archive/wordnet.is_a.txt)

## (4) COCO

Common Object Context，微软发布的物体识别数据集，包括超过 33 万张图片（其中 20 万张被标注）和 80 个物体类别，包括目标检测，图像分割，关键点检测等任务。

## (5) Pascal VOC

Visual Object Classes，目标检测和图像分类分类数据集，包括超过 11 万张图片和 20 个物体类别，2010 年推出。

## (6) Object365

旷视 2019 年发布的通用目标检测数据集，包括 63 万张图片，其中的物体分为 365 个类别，高达 1000 万的框数。

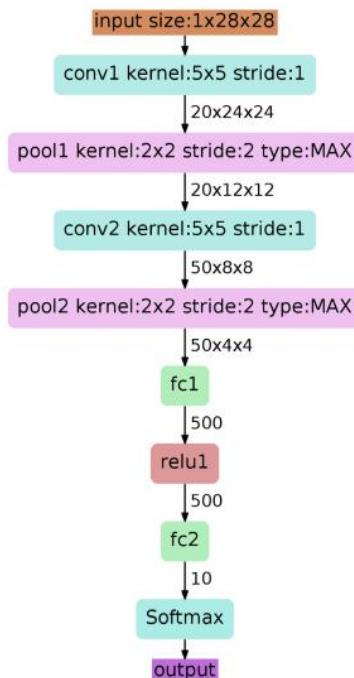
## 3 · 代码

图像分类作为最基础的神经网络，几乎在所有 NN 的通用框架中都有实现：

- darknet: darknet/cfg
- keras: keras/applications
- tensorflow: tensorflow/python/keras/applications
- pytorch: torchvision/models
- mxnet: mxnet/gluon/model\_zoo/vision
- gluon: gluoncv/model\_zoo

## 4 · LeNet (1998)

1998 年 Yann LeCun 设计的 CNN，用于识别手写文字的图片，CNN 的开山鼻祖，也确定了基础 CNN 的大致结构。现在常用的简化改进过的 LeNet-5 包含 5 个层（2 个卷积+池化层，2 个 FC 层，1 个 Softmax 层）。



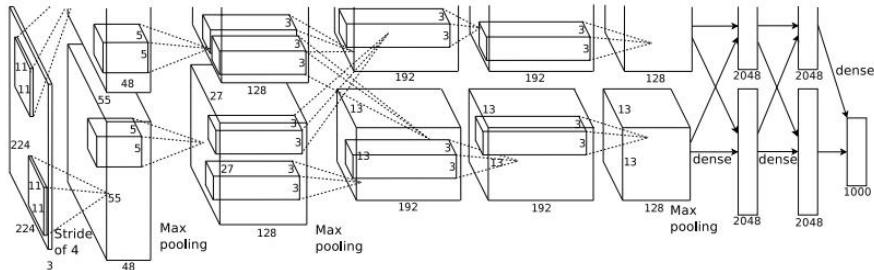
## 5 · AlexNet (2012)

2012 年 Hinton 的学生 Alex Krizhevsky 设计的 CNN，当年 ImageNet ( ILSVRC2012 ) 分类竞赛的冠军，大幅提高了对 ImageNet 的识别率，从而使得深度神经网络成为一门显学。

AlexNet 网络作者是多伦多大学的 Alex Krizhevsky 等人。Alex Krizhevsky 是 Hinton 的学生。在 ILSVRC-2010 竞赛中，AlexNet 取得了 37.5% ( top-1 ) 和 17% ( top-5 ) 的错误率。并且在 ILSVRC-2012 竞赛中获得了 15.3% ( top-5 ) 的错误率，获得第二名的方法错误率是 26.2%，可以说差距是非常的大了，足以说明这个网络在当时给学术界和工业界带来的冲击之大。

AlexNet 网络结构在整体上类似于 LeNet，都是先卷积然后在全连接（之后很多的 CNN 都采取这种形式）：

- AlexNet 有 60 million 个参数和 65000 个 神经元
- 包括五层卷积（其中部分后跟 max 池化层），三层全连接网络
- 最终的输出层是 1000 通道的 softmax
- 为了防止过拟合，采用了 dropout 技术（当时是新技术）
- AlexNet 利用了两块 GPU 进行计算，提高了运算效率



上图是 AlexNet 的网络结构，前半部分分为两个相同的结构，这是因为 AlexNet 是在两个 GPU 上进行运算的，这两部分分别对应两个 GPU。

- 输入层为 224\*224\*3 ( 图片大小+RGB )
- 第一层卷积核为 11\*11\*3，48\*2 个，stride=4，padding=0，输出为 55\*55\*48\*2，后接 Max 池化层，池化核 3\*3，stride=2，输出为 27\*27\*48\*2。
- 第二层卷积核为 5\*5\*48，128\*2 个，stride=1，padding=2，输出为 27\*27\*128\*2，后接 Max 池化层，池化核 3\*3，stride=2，输出为 13\*13\*128\*2。
- 第三层卷积核为 3\*3\*128，192\*2 个，stride=1，padding=1，输出为 13\*13\*192\*2。
- 第四层卷积核为 3\*3\*192，192\*2 个，stride=1，padding=1，输出为 13\*13\*192\*2。
- 第五层卷积核为 3\*3\*192，128\*2 个，stride=1，padding=1，输出为 13\*13\*128\*2，后接 Max 池化层，池化核 3\*3，stride=2，输出为 7\*7\*128\*2。
- 第一层 4096 的全连接层，激活函数为 ReLU，dropout ratio 为 0.5
- 第二层 4096 的全连接层，激活函数为 ReLU，dropout ratio 为 0.5
- 最后一层输出为 1000 的全连接层，激活函数为 Softmax

论文：

ImageNet Classification with Deep Convolutional Neural Networks  
<https://proceedings.neurips.cc/paper/2012/file/c399862d3b9d6b76c8436e924a68c45b-Paper.pdf>

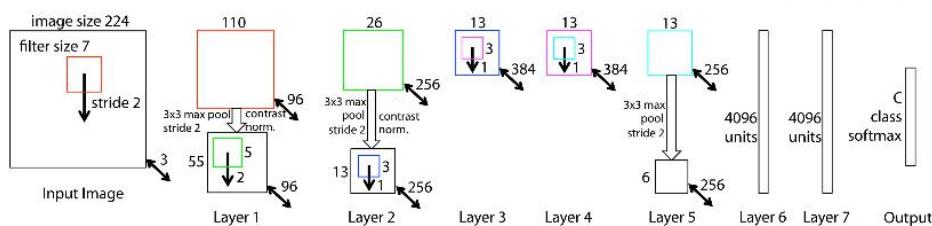
参考：

<https://blog.csdn.net/luoluonuoysuolong/article/details/81750190>

## 6 · ZFNet (201311)

2013 ImageNet (ILSVRC2013) 图像分类赛的冠军，作者是 Zeiler & Fergus，在 CNN 的结构上相较于 AlexNet 只是调整了参数（卷积核大小、stride 等等），并没有什么特别突出的地方。其论文的亮点是可视化技术的应用（ZFNet 的论文 pdf 文件大小是其他论文大小的 10 倍，就是因为其中图片太多）

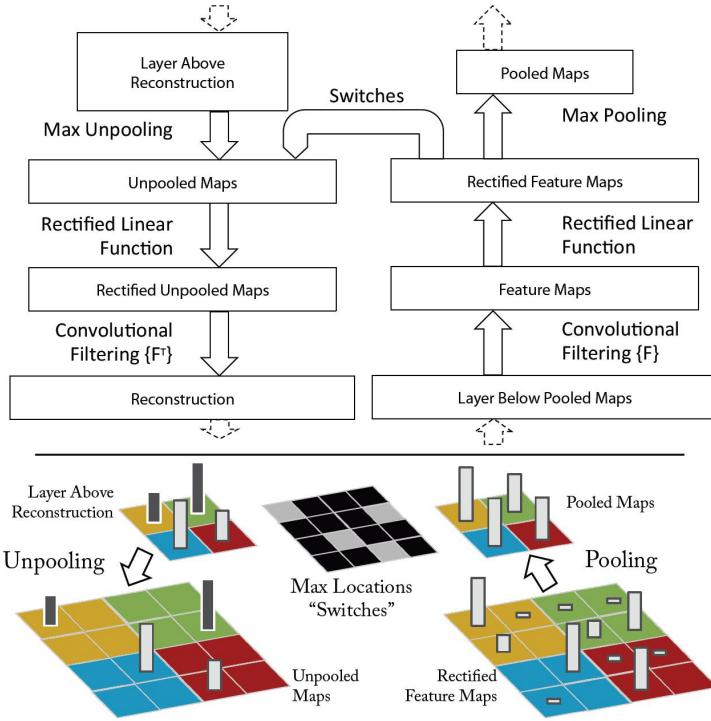
下图为 ZFNet 的结构：



在 ZFNet 中，为了实现对 CNN 的可视化，每一层所谓 **convnet 结构**（包括卷积、ReLU 激活、max 池化三层）都附加有一层 **deconvnet 结构**，deconvnet 的三个步骤分别对应 convnet 的三个步骤：

- **反池化 (unpooling)**：池化操作是不可逆的，通过池化时记录下 max 池化时最大值的位置（被称为 Switchs），在反池化时将最大值放回该位置，并将其余部分填 0，反池化操作可以近似实现池化逆操作。
- **反 ReLU 激活**：同 ReLU 激活函数
- **反卷积**：卷积网使用学习得到的卷积核与输入做卷积得到特征图，为了实现逆过程，反卷积网使用该卷积核的转置作为卷积核，与反 ReLU 之后的特征图进行卷积计算

参考下图：



*Figure 1.* Top: A deconvnet layer (left) attached to a convnet layer (right). The deconvnet will reconstruct an approximate version of the convnet features from the layer beneath. Bottom: An illustration of the unpooling operation in the deconvnet, using *switches* which record the location of the local max in each pooling region (colored zones) during pooling in the convnet.

论文：

Visualizing and Understanding Convolutional Networks

<https://arxiv.org/pdf/1311.2901>

参考：

[https://blog.csdn.net/qq\\_36673141/article/details/78551932](https://blog.csdn.net/qq_36673141/article/details/78551932)

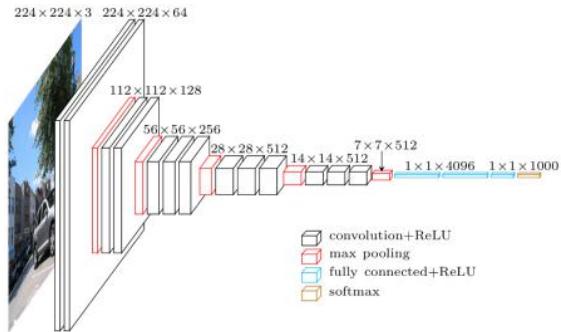
## 7 · VGGNet (201409)

VGGNet 是由牛津大学的视觉 Visual Geometry Group 和 Google Deepmind 一起研发的 CNN，是 2014 年 ImageNet (ILSVRC2014) 分类赛的第二名（冠军是 GoogLeNet）。

VGGNet 有权重的层（卷积层或者全连接层）叠加到了 13 层、16 层或者 19 层，在当时是很深的网络，也称 VGG13、VGG16 或者 VGG19。

VGG 的贡献主要有如下：

- 反复使用  $3 \times 3$  的卷积来替代之前的大卷积核，认为 2 个  $3 \times 3$  的卷积可以起到  $5 \times 5$  的卷积的效果，而计算量会大幅减少。这种思想对后来的网络有很大的影响。具体参考《[网络构成 > DNN/CNN 微结构 >  \$3 \times 3\$  卷积核](#)》
- 发现 LRN 层的作用不大



论文：

Very Deep Convolutional Networks for Large-scale Image Recognition  
<https://arxiv.org/pdf/1409.1556.pdf>

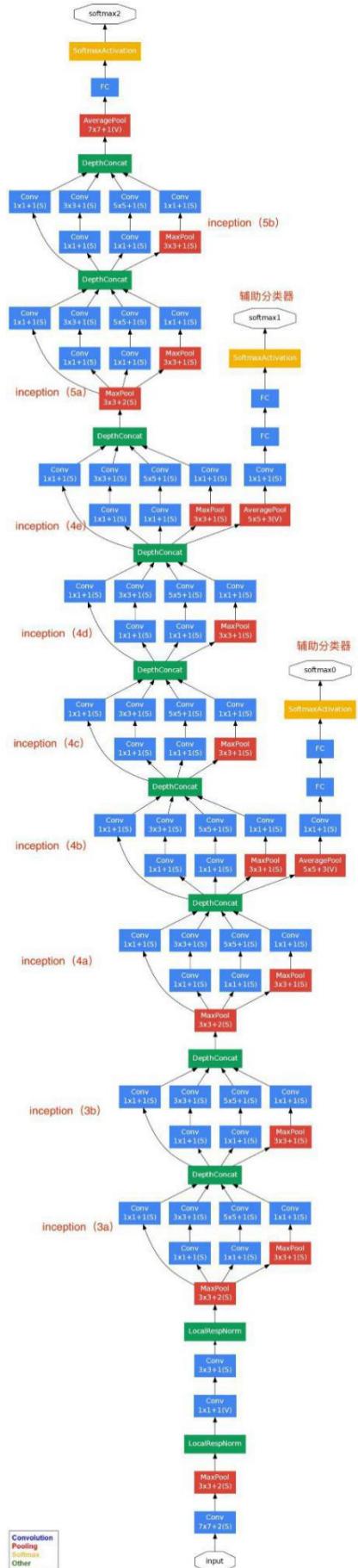
参考：

<https://blog.csdn.net/whz1861/article/details/78111606>

## 8 · GoogLeNet (201409)

Google 开发的 CNN，是 2014 年 ImageNet ( ILSVRC2014 ) 分类赛的冠军，其特点是 Inception 结构，使得网络不仅在纵向上有深度，在横向上有广度。

Inception 结构的目的是感受不同大小视野上的特征，具体请参考《[网络构成 > DNN/CNN 微结构 > Inception 结构](#)》，以下是 GoogLeNet 的结构：



论文：

Going Deeper with Convolutions

<https://arxiv.org/pdf/1409.4842.pdf>

参考：

<https://my.oschina.net/u/876354/blog/1637819>

<https://www.cnblogs.com/A11en-rg/p/5833919.html>

## 9 · ResNet (201509)

ResNet 是微软开发的 CNN，2015 年 ImageNet (ILSVRC2015) 分类赛的第一名，其特点是通过残差结构 (Residual Block) 解决了反向传播梯度消失的问题，使得比以前的网络有更深的深度。请参考《[网络构成 > DNN/CNN 微结构 > Residual block](#)》。

ResNet 论文中的几个 ResNet 变种如下：

| layer name | output size | 18-layer                                                                                  | 34-layer                                                                                  | 50-layer                                                                                                      | 101-layer                                                                                                      | 152-layer                                                                                                      |
|------------|-------------|-------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------|
| conv1      | 112×112     |                                                                                           |                                                                                           | 7×7, 64, stride 2                                                                                             |                                                                                                                |                                                                                                                |
|            |             |                                                                                           |                                                                                           | 3×3 max pool, stride 2                                                                                        |                                                                                                                |                                                                                                                |
| conv2_x    | 56×56       | $\left[ \begin{array}{c} 3 \times 3, 64 \\ 3 \times 3, 64 \end{array} \right] \times 2$   | $\left[ \begin{array}{c} 3 \times 3, 64 \\ 3 \times 3, 64 \end{array} \right] \times 3$   | $\left[ \begin{array}{c} 1 \times 1, 64 \\ 3 \times 3, 64 \\ 1 \times 1, 256 \end{array} \right] \times 3$    | $\left[ \begin{array}{c} 1 \times 1, 64 \\ 3 \times 3, 64 \\ 1 \times 1, 256 \end{array} \right] \times 3$     | $\left[ \begin{array}{c} 1 \times 1, 64 \\ 3 \times 3, 64 \\ 1 \times 1, 256 \end{array} \right] \times 3$     |
| conv3_x    | 28×28       | $\left[ \begin{array}{c} 3 \times 3, 128 \\ 3 \times 3, 128 \end{array} \right] \times 2$ | $\left[ \begin{array}{c} 3 \times 3, 128 \\ 3 \times 3, 128 \end{array} \right] \times 4$ | $\left[ \begin{array}{c} 1 \times 1, 128 \\ 3 \times 3, 128 \\ 1 \times 1, 512 \end{array} \right] \times 4$  | $\left[ \begin{array}{c} 1 \times 1, 128 \\ 3 \times 3, 128 \\ 1 \times 1, 512 \end{array} \right] \times 4$   | $\left[ \begin{array}{c} 1 \times 1, 128 \\ 3 \times 3, 128 \\ 1 \times 1, 512 \end{array} \right] \times 8$   |
| conv4_x    | 14×14       | $\left[ \begin{array}{c} 3 \times 3, 256 \\ 3 \times 3, 256 \end{array} \right] \times 2$ | $\left[ \begin{array}{c} 3 \times 3, 256 \\ 3 \times 3, 256 \end{array} \right] \times 6$ | $\left[ \begin{array}{c} 1 \times 1, 256 \\ 3 \times 3, 256 \\ 1 \times 1, 1024 \end{array} \right] \times 6$ | $\left[ \begin{array}{c} 1 \times 1, 256 \\ 3 \times 3, 256 \\ 1 \times 1, 1024 \end{array} \right] \times 23$ | $\left[ \begin{array}{c} 1 \times 1, 256 \\ 3 \times 3, 256 \\ 1 \times 1, 1024 \end{array} \right] \times 36$ |
| conv5_x    | 7×7         | $\left[ \begin{array}{c} 3 \times 3, 512 \\ 3 \times 3, 512 \end{array} \right] \times 2$ | $\left[ \begin{array}{c} 3 \times 3, 512 \\ 3 \times 3, 512 \end{array} \right] \times 3$ | $\left[ \begin{array}{c} 1 \times 1, 512 \\ 3 \times 3, 512 \\ 1 \times 1, 2048 \end{array} \right] \times 3$ | $\left[ \begin{array}{c} 1 \times 1, 512 \\ 3 \times 3, 512 \\ 1 \times 1, 2048 \end{array} \right] \times 3$  | $\left[ \begin{array}{c} 1 \times 1, 512 \\ 3 \times 3, 512 \\ 1 \times 1, 2048 \end{array} \right] \times 3$  |
|            | 1×1         |                                                                                           |                                                                                           | average pool, 1000-d fc, softmax                                                                              |                                                                                                                |                                                                                                                |
| FLOPs      |             | $1.8 \times 10^9$                                                                         | $3.6 \times 10^9$                                                                         | $3.8 \times 10^9$                                                                                             | $7.6 \times 10^9$                                                                                              | $11.3 \times 10^9$                                                                                             |

Table 1. Architectures for ImageNet. Building blocks are shown in brackets (see also Fig. 5), with the numbers of blocks stacked. Down-sampling is performed by conv3\_1, conv4\_1, and conv5\_1 with a stride of 2.

这些变种通常被称为 ResNet-50、ResNet-101、ResNet-152 等。

论文：

Deep Residual Learning for Image Recognition

<https://arxiv.org/pdf/1512.03385.pdf>

参考：

<https://blog.csdn.net/u013181595/article/details/80990930>

<http://baijiahao.baidu.com/s?id=1598536455758606033&wfr=spider&for=pc>

## 10 · SqueezeNet (201602)

SqueezeNet 在保持了 AlexNet 对 ImageNet 数据集的准确率的情况下，将参数数量压缩到了其 1/50，并且将模型大小限制在了 0.5M 之内（使用了 Deep compression 技术）。

**Fire module** 是 SqueezeNet 所提出的的微结构，由 squeeze 层（ $1 \times 1$  卷积，降维）和 expand 层（ $3 \times 3$  卷积和  $1 \times 1$  卷积并联，升维）组成，fire module 不改变特征图大小，只改变通道数。

SqueezeNet 的网络结构由一个卷积层开始，中间若干 fire module（升维）和池化层（缩小尺寸），最后一个卷积层加一个 softmax，结构如下：

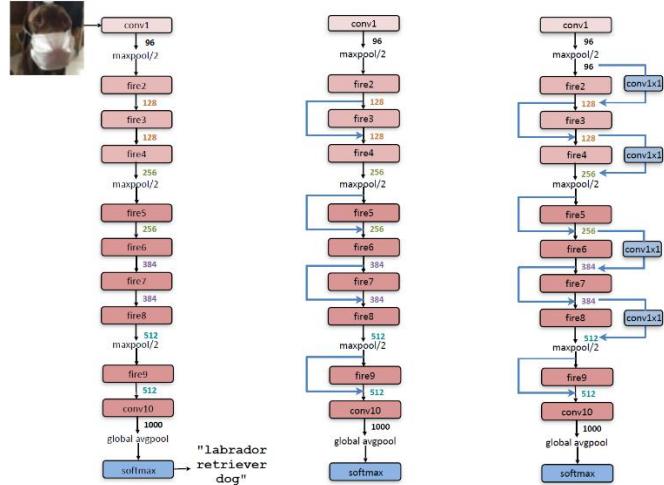


Figure 2: Macroarchitectural view of our SqueezeNet architecture. Left: SqueezeNet (Section 3.3); Middle: SqueezeNet with simple bypass (Section 6); Right: SqueezeNet with complex bypass (Section 6).

参考《[网络构成 > DNN/CNN 微结构 > Fire Module](#)》

论文：

SqueezeNet: AlexNet-level Accuracy with 50x Fewer Parameters and <0.5Mb Model Size

<https://arxiv.org/pdf/1602.07360.pdf>

参考：

<https://blog.csdn.net/csdn1dp/article/details/78648543>

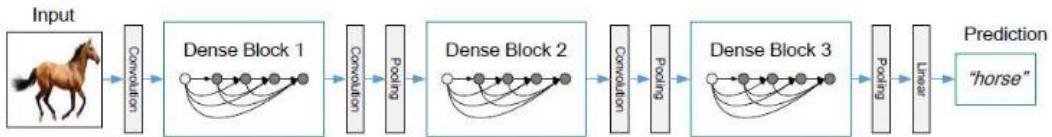
<https://blog.csdn.net/u011995719/article/details/78908755>

代码：

<https://github.com/DeepScale/SqueezeNet>

## 11 · DenseNet (201608)

2016 年发表的 DenseNet 吸收了 ResNet 的特点，是一种具有密集连接的 CNN，在一个 dense block 中的任何两层都有直接的连接，各层的输出在后面的层的输入中以通道的维度进行合并：



每个 Dense Block 中的任意两层都有连接，而每两个 Dense block 之间由于尺寸不同，有一个 Transition 层（包括一个  $1 \times 1$  卷积和一个  $2 \times 2$  的 average 池化）来实现下采样到下一个 dense block 的尺寸，具体的结构如下：

| Layers               | Output Size                      | DenseNet-121                                                                                 | DenseNet-169                                                                                 | DenseNet-201                                                                                 | DenseNet-264                                                                                 |
|----------------------|----------------------------------|----------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------|
| Convolution          | $112 \times 112$                 |                                                                                              | $7 \times 7$ conv, stride 2                                                                  |                                                                                              |                                                                                              |
| Pooling              | $56 \times 56$                   |                                                                                              | $3 \times 3$ max pool, stride 2                                                              |                                                                                              |                                                                                              |
| Dense Block (1)      | $56 \times 56$                   | $\begin{bmatrix} 1 \times 1 \text{ conv} \\ 3 \times 3 \text{ conv} \end{bmatrix} \times 6$  | $\begin{bmatrix} 1 \times 1 \text{ conv} \\ 3 \times 3 \text{ conv} \end{bmatrix} \times 6$  | $\begin{bmatrix} 1 \times 1 \text{ conv} \\ 3 \times 3 \text{ conv} \end{bmatrix} \times 6$  | $\begin{bmatrix} 1 \times 1 \text{ conv} \\ 3 \times 3 \text{ conv} \end{bmatrix} \times 6$  |
| Transition Layer (1) | $56 \times 56$<br>$28 \times 28$ |                                                                                              | $1 \times 1$ conv                                                                            | $2 \times 2$ average pool, stride 2                                                          |                                                                                              |
| Dense Block (2)      | $28 \times 28$                   | $\begin{bmatrix} 1 \times 1 \text{ conv} \\ 3 \times 3 \text{ conv} \end{bmatrix} \times 12$ | $\begin{bmatrix} 1 \times 1 \text{ conv} \\ 3 \times 3 \text{ conv} \end{bmatrix} \times 12$ | $\begin{bmatrix} 1 \times 1 \text{ conv} \\ 3 \times 3 \text{ conv} \end{bmatrix} \times 12$ | $\begin{bmatrix} 1 \times 1 \text{ conv} \\ 3 \times 3 \text{ conv} \end{bmatrix} \times 12$ |
| Transition Layer (2) | $28 \times 28$<br>$14 \times 14$ |                                                                                              | $1 \times 1$ conv                                                                            | $2 \times 2$ average pool, stride 2                                                          |                                                                                              |
| Dense Block (3)      | $14 \times 14$                   | $\begin{bmatrix} 1 \times 1 \text{ conv} \\ 3 \times 3 \text{ conv} \end{bmatrix} \times 24$ | $\begin{bmatrix} 1 \times 1 \text{ conv} \\ 3 \times 3 \text{ conv} \end{bmatrix} \times 32$ | $\begin{bmatrix} 1 \times 1 \text{ conv} \\ 3 \times 3 \text{ conv} \end{bmatrix} \times 48$ | $\begin{bmatrix} 1 \times 1 \text{ conv} \\ 3 \times 3 \text{ conv} \end{bmatrix} \times 64$ |
| Transition Layer (3) | $14 \times 14$<br>$7 \times 7$   |                                                                                              | $1 \times 1$ conv                                                                            | $2 \times 2$ average pool, stride 2                                                          |                                                                                              |
| Dense Block (4)      | $7 \times 7$                     | $\begin{bmatrix} 1 \times 1 \text{ conv} \\ 3 \times 3 \text{ conv} \end{bmatrix} \times 16$ | $\begin{bmatrix} 1 \times 1 \text{ conv} \\ 3 \times 3 \text{ conv} \end{bmatrix} \times 32$ | $\begin{bmatrix} 1 \times 1 \text{ conv} \\ 3 \times 3 \text{ conv} \end{bmatrix} \times 32$ | $\begin{bmatrix} 1 \times 1 \text{ conv} \\ 3 \times 3 \text{ conv} \end{bmatrix} \times 48$ |
| Classification Layer | $1 \times 1$                     |                                                                                              | $7 \times 7$ global average pool                                                             | 1000D fully-connected, softmax                                                               |                                                                                              |

**Table 1:** DenseNet architectures for ImageNet. The growth rate for all the networks is  $k = 32$ . Note that each “conv” layer shown in the table corresponds the sequence BN-ReLU-Conv.

[https://blog.csdn.net/blank\\_tj](https://blog.csdn.net/blank_tj)

论文：

Densely Connected Convolutional Networks

<https://arxiv.org/pdf/1608.06993.pdf>

参考：

[https://blog.csdn.net/blank\\_tj/article/details/82563810](https://blog.csdn.net/blank_tj/article/details/82563810)

[https://blog.csdn.net/sigai\\_csdn/article/details/82115254](https://blog.csdn.net/sigai_csdn/article/details/82115254)

## 12 · Xception (201610)

Xception 是在 Inception V3 的基础上做的改进，其主要特点是加入了 Depthwise Separable 卷积的思想。

参考《[网络构成 > DNN/CNN 微结构 > Depthwise Separable 卷积](#)》

论文：

Xception: Deep Learning with Depthwise Separable Convolutions

<https://arxiv.org/pdf/1610.02357.pdf>

参考：

<https://zhuanlan.zhihu.com/p/50897945>  
<https://blog.csdn.net/u014380165/article/details/75142710>  
<https://www.1eiphone.com/news/201708/KGJYBHPwsRYMhWw.html>

## 13 · ResNeXt (201611)

ResNeXt 是 ResNet 和 Inception 的结合体，不同于 Inception v4 (Inception-ResNet 结构) 的是，ResNeXt 不需要人工设计复杂的 Inception 结构细节，而是每一个分支都采用相同的拓扑结构。ResNeXt 的本质是分组卷积 (Group Convolution)，通过变量基数 (Cardinality) 来控制组的数量。组卷积是普通卷积和深度可分离卷积 (Depthwise Separable Conv) 的一个折中方案，即每个分支产生的 Feature Map 的通道数为  $n$  ( $n > 1$ )。下图左为 ResNet 结构，右为 ResNeXt 结构：

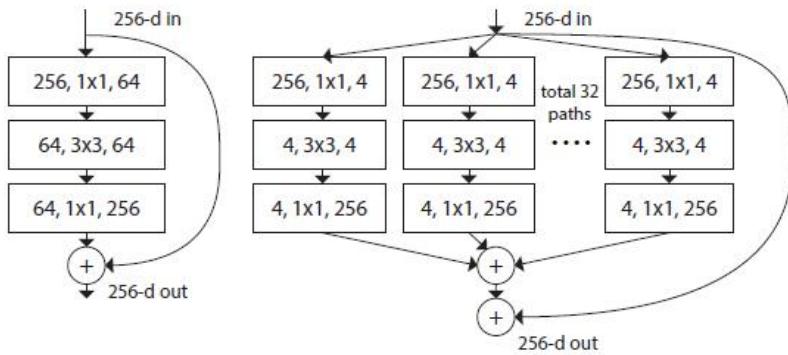


Figure 1. Left: A block of ResNet [14]. Right: A block of ResNeXt with cardinality = 32, with roughly the same complexity. A layer is shown as (# in channels, filter size, # out channels).

参考：

<https://zhuanlan.zhihu.com/p/51075096>

论文：

<https://arxiv.org/pdf/1611.05431.pdf>

## 14 · MobileNet V1 (201704)

MobileNet V1 是谷歌推出的致力于移动设备上使用的轻量化网络，其主要特点和 Xception 一样，也是深度可分离卷积 (Depthwise Separable Convolution)。

MobileNet 的结构如下，可以看出来是连续的深度可分离卷积的堆叠：

Table 1. MobileNet Body Architecture

| Type / Stride   | Filter Shape                         | Input Size                 |
|-----------------|--------------------------------------|----------------------------|
| Conv / s2       | $3 \times 3 \times 3 \times 32$      | $224 \times 224 \times 3$  |
| Conv dw / s1    | $3 \times 3 \times 32$ dw            | $112 \times 112 \times 32$ |
| Conv / s1       | $1 \times 1 \times 32 \times 64$     | $112 \times 112 \times 32$ |
| Conv dw / s2    | $3 \times 3 \times 64$ dw            | $112 \times 112 \times 64$ |
| Conv / s1       | $1 \times 1 \times 64 \times 128$    | $56 \times 56 \times 64$   |
| Conv dw / s1    | $3 \times 3 \times 128$ dw           | $56 \times 56 \times 128$  |
| Conv / s1       | $1 \times 1 \times 128 \times 128$   | $56 \times 56 \times 128$  |
| Conv dw / s2    | $3 \times 3 \times 128$ dw           | $56 \times 56 \times 128$  |
| Conv / s1       | $1 \times 1 \times 128 \times 256$   | $28 \times 28 \times 128$  |
| Conv dw / s1    | $3 \times 3 \times 256$ dw           | $28 \times 28 \times 256$  |
| Conv / s1       | $1 \times 1 \times 256 \times 256$   | $28 \times 28 \times 256$  |
| Conv dw / s2    | $3 \times 3 \times 256$ dw           | $28 \times 28 \times 256$  |
| Conv / s1       | $1 \times 1 \times 256 \times 512$   | $14 \times 14 \times 256$  |
| 5× Conv dw / s1 | $3 \times 3 \times 512$ dw           | $14 \times 14 \times 512$  |
|                 | $1 \times 1 \times 512 \times 512$   | $14 \times 14 \times 512$  |
| Conv dw / s2    | $3 \times 3 \times 512$ dw           | $14 \times 14 \times 512$  |
| Conv / s1       | $1 \times 1 \times 512 \times 1024$  | $7 \times 7 \times 512$    |
| Conv dw / s2    | $3 \times 3 \times 1024$ dw          | $7 \times 7 \times 1024$   |
| Conv / s1       | $1 \times 1 \times 1024 \times 1024$ | $7 \times 7 \times 1024$   |
| Avg Pool / s1   | Pool $7 \times 7$                    | $7 \times 7 \times 1024$   |
| FC / s1         | $1024 \times 1000$                   | $1 \times 1 \times 1024$   |
| Softmax / s1    | Classifier                           | $1 \times 1 \times 1000$   |

MobileNets 作为轻量级网络，其主要特点是在不显著降低准确率的情况下，大幅降低参数和计算量，这在不同的应用场景中都有体现。

图像分类：

Table 8. MobileNet Comparison to Popular Models

| Model             | ImageNet Accuracy | Billion   | Million    |
|-------------------|-------------------|-----------|------------|
|                   |                   | Mult-Adds | Parameters |
| 1.0 MobileNet-224 | 70.6%             | 569       | 4.2        |
| GoogleNet         | 69.8%             | 1550      | 6.8        |
| VGG 16            | 71.5%             | 15300     | 138        |

目标检测：

Table 13. COCO object detection results comparison using different frameworks and network architectures. mAP is reported with COCO primary challenge metric (AP at IoU=0.50:0.05:0.95)

| Framework Resolution | Model        | mAP       | Billion    | Million |
|----------------------|--------------|-----------|------------|---------|
|                      |              | Mult-Adds | Parameters |         |
| SSD 300              | deeplab-VGG  | 21.1%     | 34.9       | 33.1    |
|                      | Inception V2 | 22.0%     | 3.8        | 13.7    |
|                      | MobileNet    | 19.3%     | 1.2        | 6.8     |
| Faster-RCNN 300      | VGG          | 22.9%     | 64.3       | 138.5   |
|                      | Inception V2 | 15.4%     | 118.2      | 13.3    |
|                      | MobileNet    | 16.4%     | 25.2       | 6.1     |
| Faster-RCNN 600      | VGG          | 25.7%     | 149.6      | 138.5   |
|                      | Inception V2 | 21.9%     | 129.6      | 13.3    |
|                      | Mobilenet    | 19.8%     | 30.5       | 6.1     |

人脸识别：

| Model              | 1e-4     | Million   | Million    |
|--------------------|----------|-----------|------------|
|                    | Accuracy | Mult-Adds | Parameters |
| FaceNet [25]       | 83%      | 1600      | 7.5        |
| 1.0 MobileNet-160  | 79.4%    | 286       | 4.9        |
| 1.0 MobileNet-128  | 78.3%    | 185       | 5.5        |
| 0.75 MobileNet-128 | 75.2%    | 166       | 3.4        |
| 0.75 MobileNet-128 | 72.5%    | 108       | 3.8        |

论文：

MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications

<https://arxiv.org/pdf/1704.04861>

参考：

<https://blog.csdn.net/u011974639/article/details/79199306>

<https://blog.csdn.net/mzpmzk/article/details/82976871>

## 15 · NasNet (201707)

NasNet 是 Google 通过自动架构搜索生成的网络。

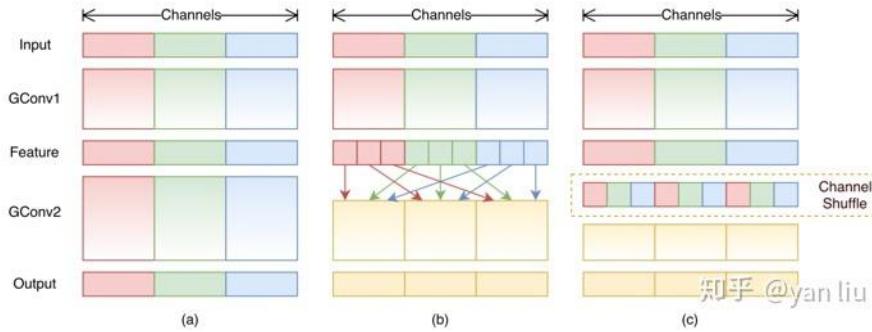
具体请参考《[元学习 > AutoML > 算法：NAS > NasNet](#)》

论文：

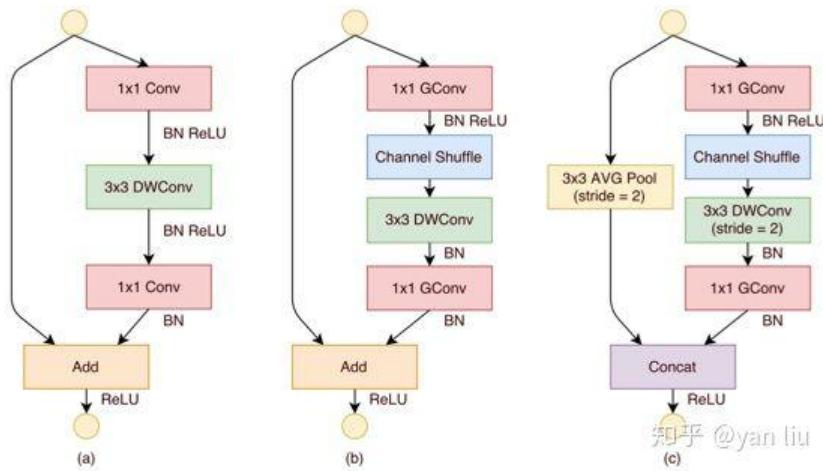
<https://arxiv.org/pdf/1707.07012.pdf>

## 16 · ShuffleNet V1 (201707)

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



上图中左为普通分组卷积，右为 channel shuffle，使得第二层卷积的输入为第一层卷积的各组输入。



上图(a)为一个普通的带有残差结构的深度可分离卷积 (MobileNet, Xception)。  
ShuffleNet v1 的结构图(b), (c)。其中(b)不需要降采样，(c)是需要降采样的情况。

参考：

[ShuffleNet 眇世官方解读](#)

<https://zhuanlan.zhihu.com/p/51566209>

论文：

<https://arxiv.org/pdf/1707.01083.pdf>

github：

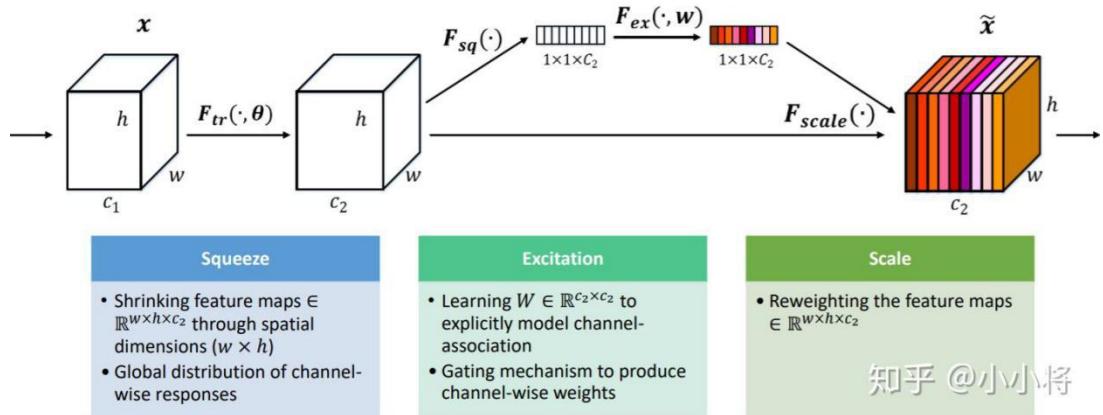
<https://github.com/megvii-model/ShuffleNet-Series>

## 17 · SENet (201709)

SENet 是最后一届 ImageNet (2017) 分类赛的冠军。

SENet 可以被视为是一种注意力模型。通常的卷积的计算方式，对每个 channel 来说，都是以同样卷积核心乘以各个输入 channel 的 feature map 再相加。SENet 网络的创新点在

于关注 channel 之间的关系，希望模型可以自动学习到不同 channel 特征的重要程度。为此，SENet 提出了 Squeeze-and-Excitation (SE) 模块，如下图所示：



SE 模块首先对卷积得到的特征图进行 Squeeze 操作，得到 channel 级的全局特征，然后对全局特征进行 Excitation 操作，学习各个 channel 间的关系，也得到不同 channel 的权重，最后乘以原来的特征图得到最终特征。本质上，SE 模块是在 channel 维度上做 attention 或者 gating 操作，这种注意力机制让模型可以更加关注信息量最大的 channel 特征，而抑制那些不重要的 channel 特征。另外一点是 SE 模块是通用的，这意味着其可以嵌入到现有的网络架构中。

具体 SE 模块请参考《[网络构成 > DNN/CNN 微结构 > SE 结构](#)》

论文：

Squeeze-and-excitation networks

<https://arxiv.org/pdf/1709.01507.pdf>

参考：

<https://zhuanlan.zhihu.com/p/65459972>

github：

<https://github.com/hujielie-frank/SENet>

## 18 · MobileNet V2 (201801)

相对于 MobileNet V1，MobileNet V2 主要引入了两个改动：

- **Linear bottleneck**：在 DW 卷积前增加一个 PW 卷积，去掉 DW 后的 PW 卷积的 ReLU
- **Inverted Residual block**：在层数比较深的 ResNet 中会使用 Residual bottleneck 结构，Inverted residual block 结构与之类似，但将常规卷积改为了 DW 卷积。

整个网络的结构如下：

| Input                 | Operator    | $t$ | $c$  | $n$ | $s$ |
|-----------------------|-------------|-----|------|-----|-----|
| $224^2 \times 3$      | conv2d      | -   | 32   | 1   | 2   |
| $112^2 \times 32$     | bottleneck  | 1   | 16   | 1   | 1   |
| $112^2 \times 16$     | bottleneck  | 6   | 24   | 2   | 2   |
| $56^2 \times 24$      | bottleneck  | 6   | 32   | 3   | 2   |
| $28^2 \times 32$      | bottleneck  | 6   | 64   | 4   | 2   |
| $28^2 \times 64$      | bottleneck  | 6   | 96   | 3   | 1   |
| $14^2 \times 96$      | bottleneck  | 6   | 160  | 3   | 2   |
| $7^2 \times 160$      | bottleneck  | 6   | 320  | 1   | 1   |
| $7^2 \times 320$      | conv2d 1x1  | -   | 1280 | 1   | 1   |
| $7^2 \times 1280$     | avgpool 7x7 | -   | -    | 1   | -   |
| $1 \times 1 \times k$ | conv2d 1x1  | -   | k    | -   | -   |

论文：

MobileNetV2: Inverted Residuals and Linear Bottlenecks  
<https://arxiv.org/pdf/1801.04381>

参考：

<https://zhuanlan.zhihu.com/p/33075914>

## 19 · MNasNet (201807)(TODO)

论文：

<https://arxiv.org/pdf/1807.11626.pdf>

## 20 · ShuffleNet V2 (201807)

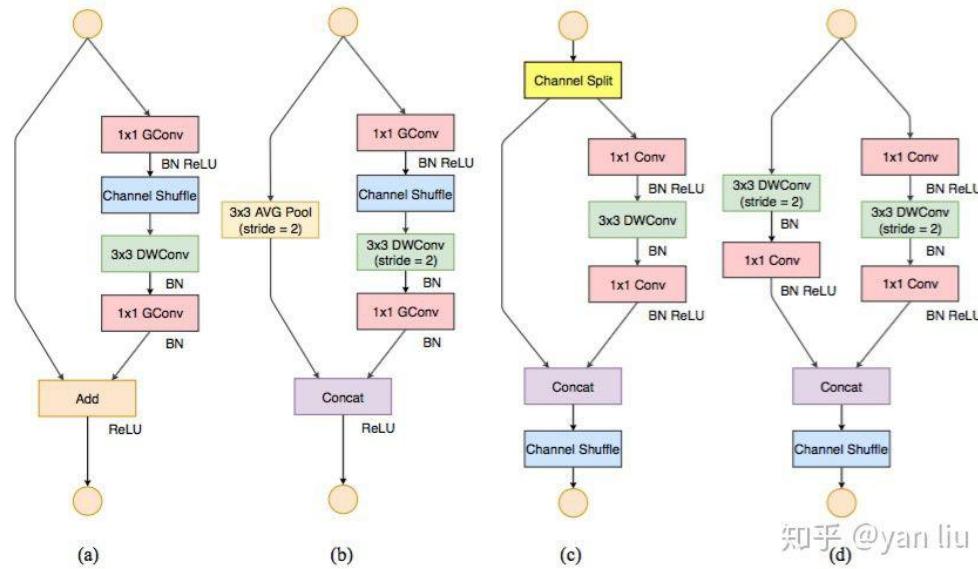
在 ShuffleNet v2 的文章中作者指出现在普遍采用的 FLOPs 评估模型性能是非常不合理的，因为一批样本的训练时间除了看 FLOPs，还有很多过程需要消耗时间，例如文件 I/O，内存读取，GPU 执行效率等等。作者从内存消耗成本，GPU 并行性两个方向分析了模型可能带来的非 FLOPs 的行动损耗，进而设计了更加高效的 **ShuffleNet v2**。ShuffleNet v2 的架构和 **DenseNet** 有异曲同工之妙，而且其速度和精度都要优于 DenseNet。

文中总结了设计高性能网络的四点规则：

1. 使用输入通道和输出通道相同的卷积操作；
2. 谨慎使用分组卷积；
3. 减少网络分支数；

#### 4. 减少 element-wise 操作。

根据以上几条原则，设计出了 ShuffleNet V2 的单元结构，下图中，a 是 ShuffleNet v1 的普通单元，b 是 v1 的降采样单元，c 是 v2 的普通单元，d 是 v2 的降采样单元。



参考：

[ShuffleNet V2 旷世官方解读](#)

<https://zhuanlan.zhihu.com/p/51566209>

<https://zhuanlan.zhihu.com/p/48261931>

论文：

<https://arxiv.org/pdf/1807.11164.pdf>

代码：

<https://github.com/megvii-model/ShuffleNet-Series>

## 21 · EfficientNet (201905)(TODO)

EfficientNet 是当前最强的图像分类 CNN。

论文：

<https://arxiv.org/pdf/1905.11946.pdf>

参考：

EfficientNet 详解。凭什么 EfficientNet 号称当今最强？

<https://zhuanlan.zhihu.com/p/104790514>

EfficientNet-可能是迄今为止最好的 CNN 网络

<https://zhuanlan.zhihu.com/p/67834114>

代码：

<https://github.com/tensorflow/tpu/tree/master/models/official/efficientnet>

## 22 · RegNet (TODO)

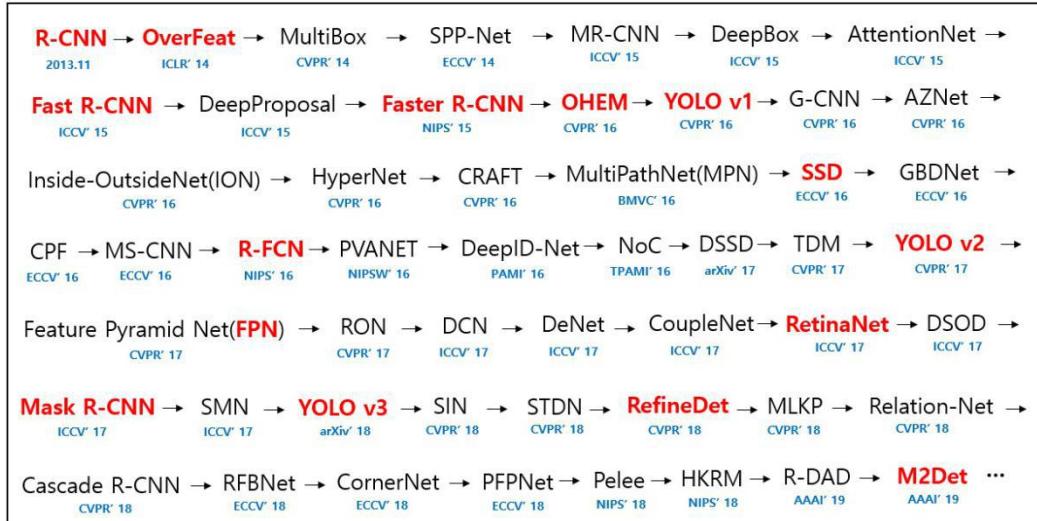
论文：

参考：

何恺明团队最新力作 RegNet：超越 EfficientNet，GPU 上提速 5 倍，这是网络设计新范式  
<https://zhuanlan.zhihu.com/p/122278712>

代码：

## (二) 目标检测 (Object Detection)



如果说图像分类是 NN 最先应用的任务，目标检测（物体检测）就是第二个。如果 Image Classification 是 Black Sabbath，Object Detection 就是 Judas Priest（不了解金属乐的请忽略该句）。

相较于图像分类，**目标检测 (Object Detection)** 任务的难度在于，它需要给出图片中若干个物体的类别及其位置 ( $X, Y, H, W$ )。或者说，目标检测的任务有 2 个，一个是分类，一个定位。

在 2012 年之前，物体检测一直是计算机视觉（CV）加上机器学习（ML）的天下。当时主要的研究方向是如何设计高质量的特征提取器和高效的分类器和回归器。当时特征提取器比较有代表性的算法有 HoG，SIFT 等，而分类任务几乎是 SVM 的天下。

早期的物体检测基本上也遵循四步走的流程：

- Selective Search 选取候选区域；
- 特征提取器提取特征；
- 分类器和回归器预测类别和位置四要素；
- non maximum suppression 合并检测框。

2012 年，Hinton 团队的 AlexNet 在 ILSVRC 大赛中将精度提高了约 10 个百分点。从此，计算机视觉的各个方向都开始考虑使用深度学习解决他们的问题。

卷积网络作为特征提取器首先引起了业内研究者的注意，2014 年，使用深度学习解决物体检测问题的开山之作 R-CNN 应运而生，论文的一作 Ross B. Girshick (RBG) 不仅是该方向的鼻祖，而且其一系列的论文也引领了物体检测的发展方向。R-CNN 的目的也很单纯，只是将特征提取器简单的换成了 CNN。

#### ● 骨干网络

同时在 2014 年，网络模型方向诞生了两个非常经典的网络结构，一是牛津大学计算机视觉组的 VGG，另外一个是谷歌公司的 GoogLeNet。VGG 使用了当时最流行的深度学习框架 Caffe，并非常有先见之明的开源了其训练好的网络模型。VGG 也作为骨干网络成为了之后 3-4 年的检测算法的主要使用骨干网络，代表算法便是 RBG 的 R-CNN 系列的三篇文章。随着数据量的增大和人们对高精度的追求，骨干网络更深的深度成为了一个最容易想到的方向。很幸运，2016 年深度学习领域的另外一尊大神，我国广东省 2003 年的高考理科状元何恺明的残差网络 ResNet 使用 short-cut 解决了深度学习中的退化问题，因为其无限深度的能力成为了近几年物体检测算法骨干网络主要使用的算法，经典算法包括 R-FCN，Mask R-CNN 以及 YOLOv3 等。

#### ● 端到端模型

物体检测的一个非常重要的优化方向是优化传统方法四步走的流程，2015 年 RBG 的 Fast R-CNN 使用 softmax 替代了 SVM，进而将特征提取和分类模型的训练合二为一，算是第一个端到端的物体检测算法。在 R-FCN 中使用了更为快速的投票机制替代了 Fast R-CNN 中的 softmax，因为 softmax 前往往要接最少一层全连接，这也成了制约 Fast R-CNN 速度的一个重要瓶颈。YOLOv3 则是使用 C 路 sigmoid 的多标签模型增强了对覆盖样本的检测能力。

同是在 2015 年，RBG 和何恺明强强联手，推出了使用 RPN 替代了 Selective Search 的 Faster R-CNN 算法。Faster R-CNN 因为其最高的算法精度和在显卡环境下的近实时的速度性能，也成了今年最为流行的算法之一。Faster R-CNN 因其巧妙的设计也是深度学习面试官最爱问的算法之一。

Faster R-CNN 系列虽然在实现上实现了端到端训练，但是其两步走（候选区域提取+位置精校）的策略也被一些人诟病。2016 年，Joseph Redmon 提出了更为革命性的 YOLO 系列算法。不同于 R-CNN 系列分两步走的策略，YOLO 是单次检测检测的算法，YOLO 可以看做是高

精度的 RPN。其更彻底的端到端训练将物体检测的速度大幅提升，在非顶端显卡环境下也实现了实时检测。

### ● 降采样池化

无论是 Selective Search 还是 RPN，得到的候选区域在尺寸和比例上都是不固定的，由此输入到网络中得到的 Feature Map 大小是不同的，最后展开成的特征向量长度也不固定，在目前的开源框架下，暂不支持变长的特征向量作为输入。在 SPP-net 中，作者提出了金字塔池化的方式，通过多尺度分 bin 的形式得到长度固定的特征向量，在 Fast R-CNN 中将其简化为单尺度并命名为 ROI Pooling。Mask R-CNN 发现当 ROI Pooling 应用到语义分割任务中会存在若干个像素的偏移误差，由此设计了更为精确的 ROIAlign。

### ● 锚点

Faster R-CNN 最大的特点是在 RPN 网络中引入了锚点机制，对锚点一个更好的解释是先验框，即对检测框的先验假设。在早期阶段，锚点是根据开发者的经验手动写死的。在 YOLOv2 中，作者在训练集对锚点进行了 k-means 聚类，进而产生了一组更优代表性的锚点。DSSD 中锚点的设置则是根据聚类的结果分析得到的。

### ● 小尺寸物体检测困难

在所有的检测算法中都普遍存在着小尺寸物体检测困难的问题。究其原因，是因为在深层网络中随着语义信息的增强，位置信息也越来越弱，这是深度网络的固有问题。SSD 率先提出使用各个阶段的 Feature Map 都参与损失函数的计算，在 FPN 中则是通过将各个阶段的 Feature Map 融合到一起的方式，融合的方式有 FPN 中从小尺寸向大尺寸融合的双线性插值上采样算法，也是目前最为广泛使用的融合方法；DSSD[11]则是通过反卷积得到不仅将小尺寸 Feature Map 上采样，而且包含语义信息的 Feature Map；而 YOLOv2 采用的是中的大尺寸向小尺寸融合的 space\_to\_depth() 算法。而 YOLOv3 则是接合了 FPN 和锚点机制的思想，为不同深度的 Feature Map 赋予了不同比例，不同尺寸的锚点。

YOLOv2 中采用的另外一个解决方案则是在训练过程中，不同批使用不同尺寸的输入图像。

### ● 半监督学习

系列算法中一个非常有商业前景的方向便是通过半监督学习的方式增加模型可处理的类别。半监督学习即是通过少量的带标签数据和大量的无标签数据，将模型的能力扩展到无标签数据中。YOLO9000 通过 WordTree 融合了 80 类的检测数据集 COCO 和 9418 类的分类数据集 ImageNet，生成了可以检测 9418 类物体的模型。MaskX R-CNN 则是通过权值迁移函数融合了 80 类的分割数据 COCO 和 3000 类的检测数据集 Visual Gnome，生成了可以分割 3000 类物体的模型。

### ● 物体检测和语义分割

近几年物体检测和语义分割的距离越来越小，双方都在汲取对方的算法来获得灵感和优化算法。最典型的算法便是 Mask R-CNN 中融合了分类，检测和分割的三任务模型。DSSD 使用反卷积进行上采样也非常有意思。

最后预测一下未来一段时间物体检测的发展方向：

1. 小尺寸物体检测困难至今尚未有效解决，更有效的多尺度 Feature Map，或者针对小尺寸物体的特定算法是研究的一个热点和难点；

2. 半监督学习：能否将语义分割任务扩展到 ImageNet 类别中，提升非子类或父类物体的无监督学习能力是一大热点；
3. 嵌入式平台的物体检测算法：目前最快的 YOLOv3 的实时运行依然依赖 GPU 环境，能否将检测算法实时的应用到嵌入式平台，例如手机，扫地机器人，无人机等都是有急切需求的场景；
4. 特定领域的物体检测算法：目前在单一领域发展较靠前的是场景文字检测算法。但在一些特定的场景中，例如医学，安检，微生物等依然很有研究前景，也是比较容易有研究成果和应用场景的方向。

总体来讲，当前的目标检测网络大致可以分为两类，一步检测和两步检测。

**两步检测（Multi-shot detector）** 先利用候选区域网络找出可能的候选目标（通常数量固定），这步被称为 region proposal，再用第二个网络来预测每个候选框的置信度（confidence）并修改边界框。早期的 R-CNN 系列就是两步检测架构。

**一步检测（Single-shot detector）** 最著名的即是 YOLO 和 SSD，也可以分为两类：基于锚点（即 anchor box）的检测和基于关键点的检测。

参考：

<https://zhuanlan.zhihu.com/p/43211392>

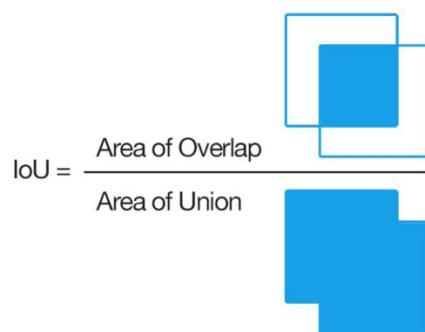
two/one-stage, anchor-based/free 目标检测发展及总结：一文了解目标检测

<https://zhuanlan.zhihu.com/p/100823629>

## 1 · 衡量标准

### (1) IoU

**交并比，Intersection over Union**，用于衡量两个区域的重合度，在目标检测任务中，这两个区域指的是**真实边界框**和**预测边界框**。其定义相当于  $TP/(TP+FP+FN)$ ，图示如下：



基于以上的定义，IoU 应而可以衡量目标检测任务中单个预测的正确与否，比如在 PASCAL VOC 数据集的测评标准中，IoU 大于 0.5 的预测被视为正确。而如果对同一个目标有多个预测（即产生了若干个预测边界框），则第一个预测被视为真阳性，其它的预测被视为假阳性。

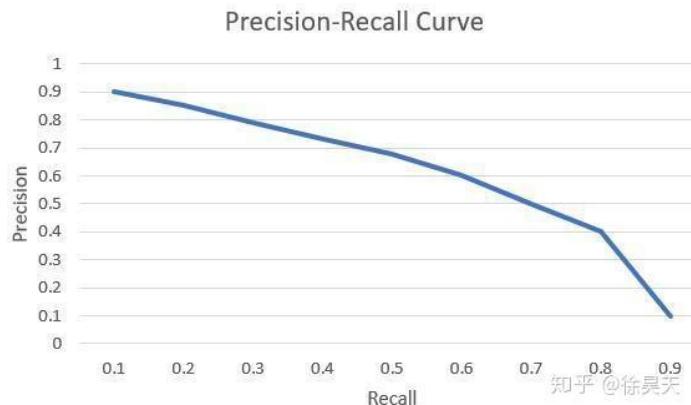
## (2) AP

**精度均值，Average Precision**，P-R 曲线中的 Precision 在不同 Recall（实际是不同置信度导致的）上的均值。

有了 IoU 推出的 TP/FP/TN/FN 作为前提，就可以引出目标检测中的 Precision 和 Recall 的定义，复习一下前文介绍的这两个概念：

- Precision :  $TP / (TP + FP)$ ，所有被检测为阳性的样本的检测正确率
- Recall :  $TP / (TP + FN)$ ，所有实际为阳性的样本的检测正确率

在某个 IoU 阈值（这里指某个 IoU，注意下图中的曲线是在单个 IoU 之下画出来的）之下，根据不同的置信度（confidence 或 score）可以统计出一组不同的 Precision-Recall（通常来说，这两个值负相关，即 Precision 越高，Recall 越低），如果将 Recall 作为横坐标，Precision 作为纵坐标，可形成一条 Precision-Recall 曲线：



AP 指的是在不同 Recall 值的情况下，Precision 的平均值。在 PASCAL VOC 数据集中，是将 Recall 分别 0, 0.1, 0.2…0.9, 1.0 的 11 种情况下的 Precision 取平均值，得到 AP。

## (3) mAP

**平均精度均值，mean Average Precision**，AP 在不同类别上的均值。

参考 AP 的定义，mAP 指的是不同类别的 AP 的平均值。也就是说，A 指的是同一类别不同置信度的平均， $m$  指的是不同类别的平均。

在某些语境下（比如 COCO 数据集中），AP 就表示 mAP，比如 AP@.75，表示在 IoU 为 0.75 的情况下，所有类别的 Precision 的平均值。

## (4) 阈值：置信度和 IoU

置信度的阈值调整通常会使得 Precision 和 Recall 此消彼长，即：

- 置信度阈值升高，则 FP 和 TP 降低，通常导致 Precision 升高，Recall 降低。
- 置信度阈值降低，则 FP 和 TP 升高，通常导致 Precision 降低，Recall 升高。

而 IoU 的阈值调整则使得整个 P-R 曲线和 bounding box 可用性此消彼长，即：

- IoU 阈值降低，则 P-R 曲线整体向右上移动（Precision 和 Recall 都更优），而 bounding box 可用性变差。
- IoU 阈值升高，则 P-R 曲线整体向左下移动（Precision 和 Recall 都更差），而 bounding box 可用性变好。

但 IoU 阈值通常只取几个固定典型值（0.5, 0.75, 0.95 等），因为对于特定任务来说，过差的 IoU 没有意义（想一下 IoU 为 0.1，可能识别出来的汽车 bounding box 中只有某一角是车的一部分）

## 2 · 数据集

### (1) PASCAL VOC

PASCAL (Pattern Analysis, Statistical Modelling and Computational Learning)，VOC (Visual Object Classes)。

PASCAL VOC 竞赛目标主要是目标检测。第一届 PASCAL VOC 举办于 2005 年，然后每年一届，于 2012 年终止。其提供的数据集里包括了 20 类的物体：

- person
- bird, cat, cow, dog, horse, sheep
- aeroplane, bicycle, boat, bus, car, motorbike, train
- bottle, chair, dining table, potted plant, sofa, tv/monitor

PASCAL VOC 的主要 2 个任务是(按照其官方网站所述，实际上是 5 个)：

- 分类：对于每一个分类，判断该分类是否在测试照片上存在（共 20 类）；
- 检测：检测目标对象在待测试图片中的位置并给出矩形框坐标（bounding box）；
- Segmentation：对于待测照片中的任何一个像素，判断哪一个分类包含该像素（如果 20 个分类没有一个包含该像素，那么该像素属于背景）；
- （在给定矩形框位置的情况下）人体动作识别；
- Large Scale Recognition（由 ImageNet 主办）。

VOC 数据集主要包括 VOC2007 和 VOC2012 两部分。

官方网页：

<http://host.robots.ox.ac.uk/pascal/VOC/>

排行榜：

[http://host.robots.ox.ac.uk:8080/leaderboard/main\\_bootstrap.php](http://host.robots.ox.ac.uk:8080/leaderboard/main_bootstrap.php)

参考：

<https://zhuanlan.zhihu.com/p/33405410>

<https://blog.csdn.net/u013832707/article/details/80060327>

<https://blog.csdn.net/mzpmzk/article/details/88065416>

## 1 格式

对于目标检测来说，每一张图片对应一个 xml 格式的标注文件。下面是其中一个 xml 文件的示例：

```
<annotation>
 <folder>VOC2007</folder>
 <filename>001264.jpg</filename>
 <source>
 <database>The VOC2007 Database</database>
 <annotation>PASCAL VOC2007</annotation>
 <image>flickr</image>
 <flickrid>194748336</flickrid>
 </source>
 <owner>
 <flickrid>Hannibal. Or maybe just Rex.</flickrid>
 <name>Minky Paw</name>
 </owner>
 <size>
 <width>500</width>
 <height>375</height>
 <depth>3</depth>
 </size>
 <segmented>0</segmented>
 <object>
 <name>motorbike</name>
 <pose>Unspecified</pose>
 <truncated>0</truncated>
```

```
<difficult>0</difficult>
<bndbox>
 <xmin>69</xmin>
 <ymin>25</ymin>
 <xmax>447</xmax>
 <ymax>348</ymax>
</bndbox>
</object>
</annotation>
```

在这个 xml 格式中：

- bndbox 是一个轴对齐的矩形，它框住的是目标在照片中的可见部分。
- truncated 表明这个目标因为各种原因没有被框完整（被截断了），比如说一辆车有一部分在画面外。
- occluded 是说一个目标的重要部分被遮挡了（不管是被背景的什么东西，还是被另一个待检测目标遮挡）。
- difficult 表明这个待检测目标很难识别，有可能是虽然视觉上很清楚，但是没有上下文的话还是很难确认它属于哪个分类；标为 difficult 的目标在测试成绩的评估中一般会被忽略。

VOC 数据集的目录结构如下，VOC2012 和 VOC2007 一样：

- Annotations：存放上面提到的目标检测的标识 xml
- ImageSets：存放各种数据集的划分，txt 格式，内容是图片文件名
  - Main：目标检测相关的数据集划分，例如 train.txt 中为所有的训练数据集图片，cat\_val.txt 为所有猫分类的验证数据集图片（这种单分类训练集划分里，每行最后的 1/-1 应为该类的正负样本）
  - Action：人体动作识别数据集的划分
  - Layout：
  - Segmentation：分割训练数据集的划分
- JPEGIMages：存放所有原始的图片文件
- SegmentationClass：语义分割掩图图片
- SegmentationObject：实例分割掩图图片

## (2) COCO

COCO 的 全称是 Common Objects in COntext，是微软团队提供的一个可以用来进行图像识别的数据集。MS COCO 数据集中的图像分为训练、验证和测试集。

官网：

<http://cocodataset.org/#home>

参考：

<https://zhuanlan.zhihu.com/p/29393415>

## ① 格式

COCO 的目录格式如下：

- annotation : 标识
- common :
- images : 原始图片
  - train2014 : 2014 训练集
  - val2014 : 2014 验证集
- LuaAPI :
- MatlabAPI :
- PythonAPI :
- results :

## 3 · 代码

目标检测 CNN 的作者很多都来自 Facebook (Ross Girshick, 何恺明等) , YOLOv1 也有 Facebook 的参与，因此目标检测的专业库里，FB 的 Detectron2 涵盖了较多的实现。

github :

<https://github.com/facebookresearch/Detectron2>

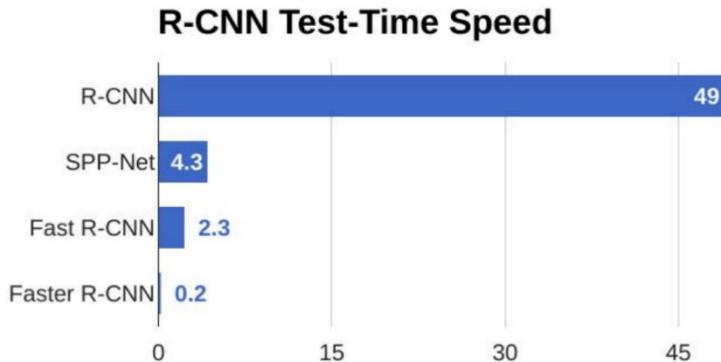
## 4 · R-CNN 系列

传统的 CNN 最主流的应用场景是图像分类 (Image Classification) , 通过若干层的卷积层提取出图像的特征，最后通过全连接层得到若干种分类的可能性。而在目标检测 (Object Detection) 任务中，R-CNN 系列是早期主要的技术演进路线：

R-CNN -> SPP-net -> Fast R-CNN -> Faster R-CNN -> R-FCN

目标检测的一个最简单直观的办法就是：对图片的不同子区域进行类似图像分类 CNN 所进行的操作，但是这个办法的问题在于计算量太大。

因此 R-CNN 系列神经网络登上了历史的舞台，先直观的了解一下这几个神经网络的处理时间，单位是秒：



R-CNN 系列的特点是所谓 two-stage 方法：先通过 region proposal (selective search 算法，或者 CNN) 确定若干候选框，然后对这些候选框进行分类（确定类别）和回归（确定边界）。

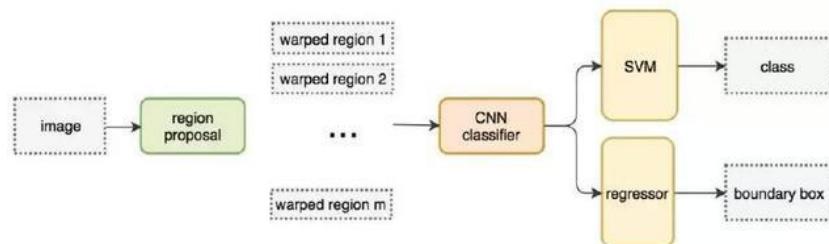
### (1) R-CNN (201311)

为了减少需要通过 CNN 的图像的数量，R-CNN 的流程如下：

1. 首先通过 region proposal (具体是 selective search 算法) 选取了 2000 个候选区域 (RoI, Region of Interest)。
2. 将这些候选区域缩放为统一的大小 (227\*227)，然后送入 CNN 分类器
3. 这个 CNN 分类器提取出 4096 个特征，这些特征被分别送入两个地方：
  - a) N 个 SVM 分类器决定其所属物体类别 (每一个类别对应一个 SVM，用于判断是否属于该类别)；
  - b) 一个岭回归器 (ridge regression)，用来修正物体的位置。
4. 使用贪心的非极大值抑制 (NMS) 合并候选区域，得到输出结果

Selective search 算法通过颜色和纹理的不断合并得到候选区域。

下图为 R-CNN 的结构图：



R-CNN 的主要问题是其计算量仍然很大，2000 个 RoI 都需要走一遍 CNN，导致其对单张图片的操作时间达到了 40 多秒，完全无法用于实时检测。

Selective Search 的区域的合并规则是：

- 优先合并颜色相近的

- 优先合并纹理相近的
- 优先合并合并后总面积小的
- 合并后，总面积在其 BBOX 中所占比例大的优先合并

下图是通过 Selective Search 得到的一组候选区域



参考：

<https://zhuanlan.zhihu.com/p/42731634>

论文：

R-CNN : Rich feature hierarchies for accurate object detection and semantic segmentation Tech report (v5)

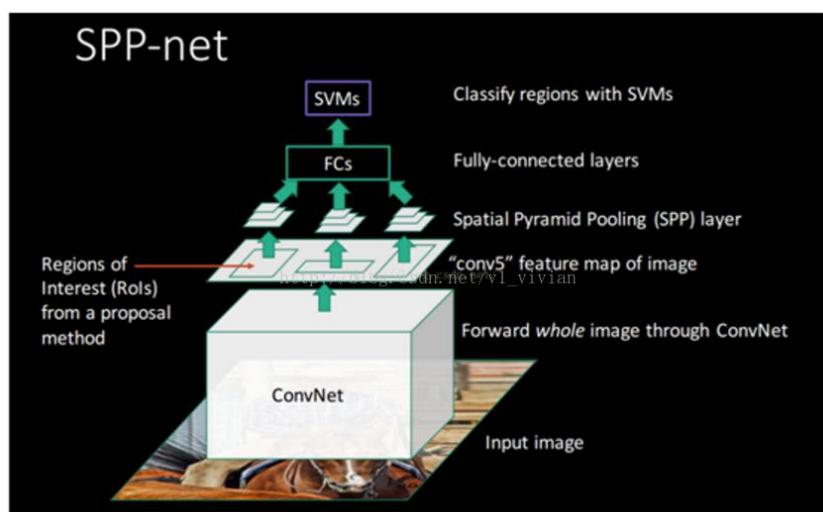
<https://arxiv.org/pdf/1311.2524>

selective search :

<http://www.huppenen.nl/publications/selectiveSearchDraft.pdf>

## (2) SPP-net (201406)

在 R-CNN 系列网络演进的过程中，SPP (Spatial Pyramid Pooling，空间金字塔池化) 的思想对其影响很大。SPP 层具体请参考《[网络构成 > DNN/CNN 微结构 > SPP 层](#)》节，SPP-net 的结构如下：



SPP 层是介于特征层（卷积层的最后一层）和全连接层之间的一种 pooling 结构。

参考：

<https://zhuanlan.zhihu.com/p/42732128>

论文：

Spatial Pyramid Pooling in Deep Convolutional Networks for Visual Recognition

<https://arxiv.org/pdf/1406.4729>

### (3) Fast R-CNN (201504)

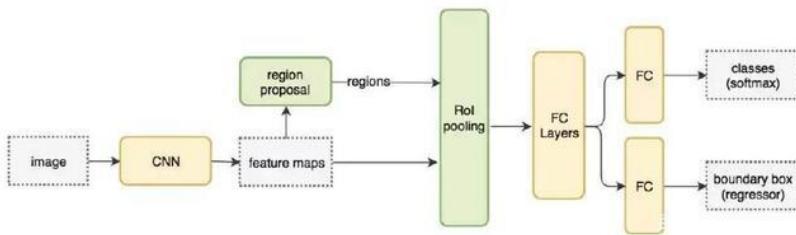
Fast R-CNN 是在 R-CNN 的基础上，结合了 SPP 层的特点，做出了改进，流程如下：

1. 通过 region proposal 在图像上选出 1000-2000 个 ROI
2. 对整个图像卷积一次（而不是每个 ROI 都做一次卷积）
3. 在卷积后的特征图上找到对应的 ROI，并将它们的特征图送入 ROI Pooling（ROI 池化层，即 SPP 层），输出固定维度的特征值。
4. 从 ROI 池化层输出的特征值被送入 FC 层，之后接两个输出
  - a) FC 层+Softmax 函数作为分类器，用于区分物体类别
  - b) FC 层+边界框回归器，用于确定物体位置

相较于 R-CNN，做出的改动有：

1. region proposal 放在 CNN 之后，只卷积一次，从整体特征图中获得 ROI 特征图
2. 不缩放图像，而是用 ROI Pooling 层获得特征值
3. 最后的分类器由 SVM 变为 FC 层(Softmax)

结构图如下：



参考：

<https://zhuanlan.zhihu.com/p/42738847>

论文：

Fast R-CNN

<https://arxiv.org/pdf/1504.08083>

代码：

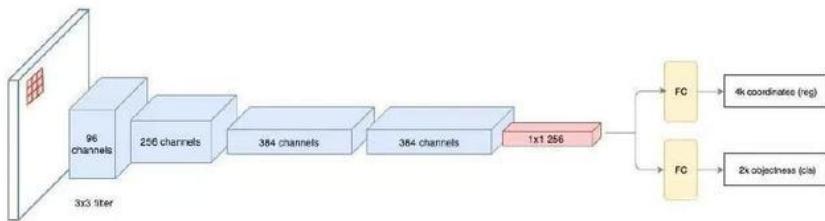
<https://github.com/rbgirshick/fast-rcnn> (已废弃，现在 Detectron2 中)

## (4) Faster R-CNN (201506)

Fast R-CNN 的操作时间降到了 2.3 秒，但是其中有 2 秒左右的时间都是在进行 region proposal。因此想要进一步提高速度，就要找出一个更高效率的 ROI 搜索算法，这个算法就是 RPN。

RPN (Region Proposal Network) 是 Faster R-CNN 所使用的 ROI 搜索器，是一个 CNN，具体在 Faster R-CNN 论文中使用的是 ZFNet 或 VGG。RPN 将第一个 CNN 的输出作为输入，吐出 256 维的特征值，这个特征值被送入两个独立的 FC 层，一个用于做有物体/无物体的分类器，另一个则用于预测边框。

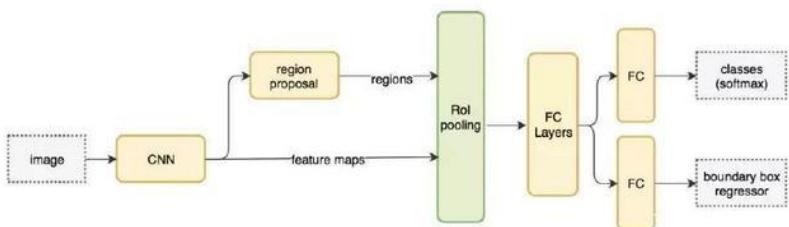
RPN 结构如下：



这样整个 Faster R-CNN 都是由 CNN 实现的，实现了端到端，整个 Faster R-CNN 的流程：

- 对整张图像进行卷积，得到特征图
- 卷积特征输出到 RPN，得到候选框的特征信息
- 对候选框中提取出的特征，分别送去做分类和边框预测

相较于 Fast R-CNN，Faster R-CNN 的改进就是使用 RPN 替代了之前的 selective search 作为 region proposal 算法，因此大幅提高了速度。整体结构如下：



论文：

Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks  
<https://arxiv.org/pdf/1506.01497.pdf>

参考：

<https://zhuanlan.zhihu.com/p/31426458>  
<https://zhuanlan.zhihu.com/p/24916624>  
<https://zhuanlan.zhihu.com/p/42741973>

## (5) R-FCN (201605)

论文：

<https://arxiv.org/pdf/1605.06409.pdf>

参考：

<https://zhuanlan.zhihu.com/p/42858039>

# 5 · YOLO 系列

You Only Look Once，YOLO 之前的目标检测方法（主要是 R-CNN 系列）都分成两步，先通过 regional proposal 产生若干可能包含物体的边界框，再通过分类器判断是具体类别，以及回归器来精确定位物体的边界。而正如其名所示，YOLO 的特点是通过一次卷积过程完成了整个定位和分类的过程。

YOLO 的准确率（定位误差和召回率）比 R-CNN 系列低，优点是速度快（高一个数量级），以及假阳性（FP）更低。YOLO 也是一个系列，包括 V1、V2 和 V3。

代码：

原作者 redmon 版本：<https://github.com/pjreddie/darknet>

Alexy 工程化版本：<https://github.com/AlexeyAB/darknet>

第三方团队 Pytorch 复现版：<https://github.com/ultralytics/yolov3>

## (1) YOLOv1 (201506)

相较于 R-CNN 系列，YOLO V1 的优势在于：

- 计算速度更快（45FPS，能达到 Faster R-CNN 的若干倍）
- 假阳性（False Positive）更低
- 由于其从整体着眼，泛化能力更强，在艺术作品中有更好的效果

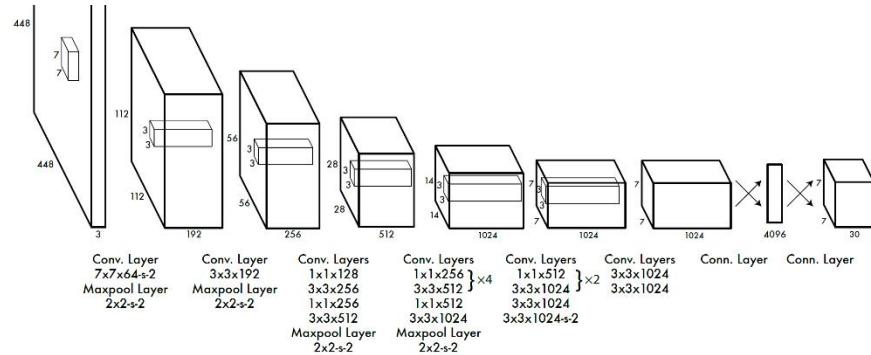
但其缺点也很明显：

- 准确率其实更低，且边界框更不准。
- 由于其原理限制，能预测的物体数量有限

YOLO 将图像分成  $S \times S$  个格子，如果一个物体的中心落在某格子内，则该格子负责预测该物体，每个格子可以预测  $B$  个物体，每个物体用 5 个值用来描述： $x, y, w, h, confidence$ ， $(x, y)$  是该物体的中心位置， $w$  和  $h$  是物体的宽和高， $confidence$  是该物体的置信度。此外每个格子有  $C$  个值来描述分类，每个值表示一个物体分类（注意是每个格子，不是每个物体）。

因此 YOLO 的输出张量为  $S \times S \times (B \times 5 + C)$ ，通常  $S$  为 7， $B$  为 2， $C$  则为 20（PASCAL VOC 数据集），整个输出张量为  $7 \times 7 \times 30$ 。

以下为 YOLO 的网络结构，24 层卷积+2 层全连接：



**Figure 3: The Architecture.** Our detection network has 24 convolutional layers followed by 2 fully connected layers. Alternating  $1 \times 1$  convolutional layers reduce the features space from preceding layers. We pretrain the convolutional layers on the ImageNet classification task at half the resolution ( $224 \times 224$  input image) and then double the resolution for detection.

首先用前 20 层卷积层+1 个平均池化层+1 个全连接层，在 ImageNet 上进行分类训练，最后达到 top-5 有 88% 的正确率。

接着将后 4 层卷积层+2 层全连接层接到之前训练好的 20 层卷积层之后，形成完整的网络，同时将输入分辨率从  $224 \times 224$  提升到  $448 \times 448$ 。

因为在图片的大部分格子中没有物体，这部分权重占比太大，导致网络训练发散，为了避免这种情况，在训练中给有物体的格子和没有物体的格子的损失分配了不同的权重（分别为 5 和 0.5）。

对于不同大小的物体，同样的宽高误差显然有不同的意义，更小的物体对宽高误差更敏感。为了应对这种情况，在损失函数中使用  $w$  和  $h$  的平方根而非本身来衡量误差。

以下是损失函数，其中：

- 第一行是中心点坐标的误差 ( $S \times S \times B \times 2$ )
- 第二行是宽和高的误差 ( $S \times S \times B \times 2$ )
- 第三行是有物体的格子的置信度误差
- 第四行是没有物体的格子的置信度的误差
- 最后一行是类别的误差 ( $S \times S \times B \times C$ )

$$\begin{aligned}
& \lambda_{\text{coord}} \sum_{i=0}^{S^2} \sum_{j=0}^B \mathbb{1}_{ij}^{\text{obj}} \left[ (x_i - \hat{x}_i)^2 + (y_i - \hat{y}_i)^2 \right] \\
& + \lambda_{\text{coord}} \sum_{i=0}^{S^2} \sum_{j=0}^B \mathbb{1}_{ij}^{\text{obj}} \left[ \left( \sqrt{w_i} - \sqrt{\hat{w}_i} \right)^2 + \left( \sqrt{h_i} - \sqrt{\hat{h}_i} \right)^2 \right] \\
& + \sum_{i=0}^{S^2} \sum_{j=0}^B \mathbb{1}_{ij}^{\text{obj}} \left( C_i - \hat{C}_i \right)^2 \\
& + \lambda_{\text{noobj}} \sum_{i=0}^{S^2} \sum_{j=0}^B \mathbb{1}_{ij}^{\text{noobj}} \left( C_i - \hat{C}_i \right)^2 \\
& + \sum_{i=0}^{S^2} \mathbb{1}_i^{\text{obj}} \sum_{c \in \text{classes}} (p_i(c) - \hat{p}_i(c))^2
\end{aligned}$$

论文：

You Only Look Once: Unified, Real-Time Object Detection

<https://arxiv.org/pdf/1506.02640>

## (2) YOLOv2 (201612)

YOLO V2 的出现是为了解决 YOLO V1 的两个主要的问题：定位准确率较低，以及较低的识别率 (TPR)。目的是在不增加网络复杂度的情况下提高这些值，为此文章中使用了一系列手段来实现，或者说，YOLO V2 和 YOLO V1 的区别就是一系列细节，下表中显示了各种不同手段应用在 YOLO 上之后带来的 mAP 上的变化：

	YOLO	YOLOv2						
batch norm?	✓	✓	✓	✓	✓	✓	✓	✓
hi-res classifier?	✓	✓	✓	✓	✓	✓	✓	✓
convolutional?	✓	✓	✓	✓	✓	✓	✓	✓
anchor boxes?	✓	✓						
new network?		✓	✓	✓	✓	✓	✓	✓
dimension priors?			✓	✓	✓	✓	✓	✓
location prediction?				✓	✓	✓	✓	✓
passthrough?					✓	✓	✓	✓
multi-scale?						✓		✓
hi-res detector?							✓	✓
VOC2007 mAP	63.4	65.8	69.5	69.2	69.6	74.4	75.4	76.8
								<b>78.6</b>

**Table 2: The path from YOLO to YOLOv2.** Most of the listed design decisions lead to significant increases in mAP. Two exceptions are switching to a fully convolutional network with anchor boxes and using the new network. Switching to the anchor box style approach increased recall without changing mAP while using the new network cut computation by 33%.

这些手段包括：

### Batch Normalization

BN 可以提高模型收敛速度，且可以降低模型的过拟合，YOLOv2 中每个卷积层后面都增加了 BN，且不再使用 dropout。

### 高分辨率分类器

High-resolution classifier，每个目标检测网络都会使用经 ImageNet 预训练的分类器，YOLOv1 在训练的时候先在 224\*224 的图片上训练分类器，再在 448\*448 的图片上训练目标检测，这意味着网络需要同时应用低分辨率到高分辨率，以及分类到检测的两个转变。对于 YOLOv2，在这两者之间增加了 10 个 epoch 的 448\*448 的图片分类训练做 fine tune，然后在转到 448\*448 的目标检测做 fine-tune。

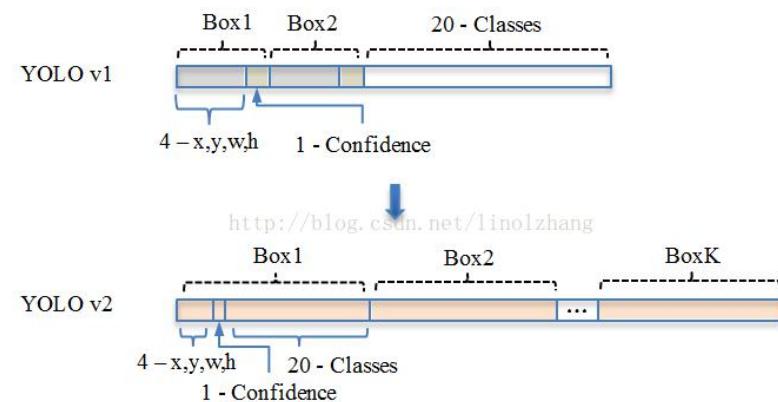
### 带先验框的卷积

锚点（anchor box）即先验框，这个概念是 Faster R-CNN 中提出的。YOLOv1 直接预测每个边界框的坐标，而 YOLOv2 则借鉴了先验框的概念，预测相对于先验框的偏移值，这样可以更好的学习。并且通过设置不同长宽比的先验框，对于不同长宽比的物体可以有很高的预测结果（这也是 YOLOv1 TPR 低的一个原因）。

YOLOv2 去除了 v1 中的全连接层，改用卷积层和先验框来预测边界框。为了使检测所用的特征图分辨率更高，移除了一个池化层。并且不使用 448\*448 而是 416\*416 作为输入，这使得全部下采样（总步长 32）之后得到的特征图大小为 13\*13，这样只有一个中心位置，便于预测中心点在中间的大物体。

YOLOv1 将图像分成 7\*7 的格子（也即其最终特征图的大小），每格可以有 2 个边界框（每个边界框有一个置信度来表示是否有物体，但这两个边界框共享一个分类预测），也就是说 YOLOv1 最多只能预测 98 个物体。

YOLOv2 则大大提高了预测物体的数量，首先有 13\*13 个预测位置，并且每个位置有若干个先验框，每个先验框都有其对应的置信度（有无物体）和分类预测。与 SSD 不同，YOLO 使用置信度来分辨有无物体，而 SSD 将背景作为一个单独的分类，因此 SSD 的分类预测数量是比实际的类别数量多 1。

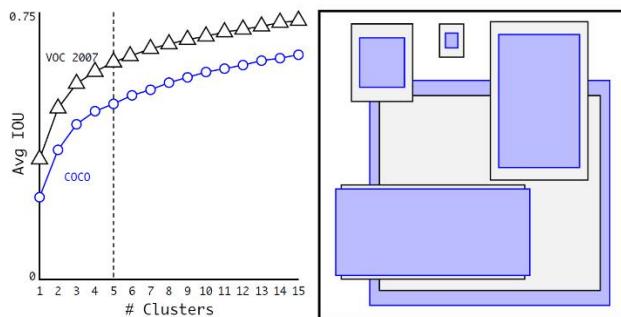


### 维度聚类

Dimension clusters，在 Faster R-CNN 和 SSD 中先验框的维度（长和宽）都是手动设定的，带有一定的主观性。如果选取的先验框维度比较合适，那么模型更容易学习，从而做出更好的预测。因此，YOLOv2 采用 **k-means** 聚类方法对训练集中的边界框做了聚类分析。因为设置先验框的主要目的是为了使得预测框与 ground truth 的 IOU 更好，所以聚类分析时选用 box 与聚类中心 box 之间的 IOU 值作为距离指标：

$$d(\text{box}, \text{centroid}) = 1 - \text{IOU}(\text{box}, \text{centroid})$$

下图为在 VOC 和 COCO 数据集上的聚类分析结果，随着聚类中心数目的增加，平均 IOU 值（各个边界框与聚类中心的 IOU 的平均值）是增加的，但是综合考虑模型复杂度和召回率，作者最终选取 5 个聚类中心作为先验框，其相对于图片的大小如右侧图所示：



### 新的网络

YOLOv2 采用了一个新的基础网络用作特征提取，被称为 Darknet-19，包括 19 个卷积层和 5 个最大池化层：

Type	Filters	Size/Stride	Output
Convolutional	32	$3 \times 3$	$224 \times 224$
Maxpool		$2 \times 2/2$	$112 \times 112$
Convolutional	64	$3 \times 3$	$112 \times 112$
Maxpool		$2 \times 2/2$	$56 \times 56$
Convolutional	128	$3 \times 3$	$56 \times 56$
Convolutional	64	$1 \times 1$	$56 \times 56$
Convolutional	128	$3 \times 3$	$56 \times 56$
Maxpool		$2 \times 2/2$	$28 \times 28$
Convolutional	256	$3 \times 3$	$28 \times 28$
Convolutional	128	$1 \times 1$	$28 \times 28$
Convolutional	256	$3 \times 3$	$28 \times 28$
Maxpool		$2 \times 2/2$	$14 \times 14$
Convolutional	512	$3 \times 3$	$14 \times 14$
Convolutional	256	$1 \times 1$	$14 \times 14$
Convolutional	512	$3 \times 3$	$14 \times 14$
Convolutional	256	$1 \times 1$	$14 \times 14$
Convolutional	512	$3 \times 3$	$14 \times 14$
Maxpool		$2 \times 2/2$	$7 \times 7$
Convolutional	1024	$3 \times 3$	$7 \times 7$
Convolutional	512	$1 \times 1$	$7 \times 7$
Convolutional	1024	$3 \times 3$	$7 \times 7$
Convolutional	512	$1 \times 1$	$7 \times 7$
Convolutional	1024	$3 \times 3$	$7 \times 7$
Convolutional	1000	$1 \times 1$	$7 \times 7$
Avgpool		Global	1000
Softmax			

Darknet-19 与 VGG16 模型设计原则是一致的，主要采用  $3 \times 3$  卷积，采用  $2 \times 2$  的 maxpooling 层之后，特征图维度降低 2 倍，而同时将特征图的 channels 增加两倍。与 NIN(Network in Network)类似，Darknet-19 最终采用 global avgpooling 做预测，并且在  $3 \times 3$  卷积之间使用  $1 \times 1$  卷积来压缩特征图 channels 以降低模型计算量和参数。Darknet-19 每个卷积层后面同样使用了 batch norm 层以加快收敛速度，降低模型过拟合。在 ImageNet 分类数据集上，Darknet-19 的 top-1 准确度为 72.9%，top-5 准确度为 91.2%，但是模型参数相对小一些。使用 Darknet-19 之后，YOLOv2 的 mAP 值没有显著提升，但是计算量却可以减少约 33%。

## 直接位置预测

Direct location prediction，直接 Anchor Box 回归导致模型不稳定，对应公式也可以参考 Faster-RCNN 论文，该公式没有任何约束，中心点可能会出现在图像任何位置，这就有可能导致回归过程震荡，甚至无法收敛。

针对这个问题，作者在预测位置参数时采用了强约束方法：

- 1) 对应 Ce11 距离左上角的边距为  $(Cx, Cy)$ ， $\sigma$  定义为 sigmoid 激活函数，将函数值约束到  $[0, 1]$ ，用来预测相对于该 Ce11 中心的偏移（不会偏离 ce11）；
- 2) 预定 Anchor (文中描述为 bounding box prior) 对应的宽高为  $(Pw, Ph)$ ，预测 Location 是相对于 Anchor 的宽高 乘以系数得到

## 细粒度特征

YOLOv2 的输入图片大小为 416\*416，经过 5 次 maxpooling 之后得到 13\*13 大小的特征图，并以此特征图采用卷积做预测。13\*13 大小的特征图对检测大物体是足够了，但是对于小物体还需要更精细的特征图 (Fine-Grained Features)。因此 SSD 使用了多尺度的特征图来分别检测不同大小的物体，前面更精细的特征图可以用来预测小物体。YOLOv2 提出了一种 **passthrough** 层来利用更精细的特征图。YOLOv2 所利用的 Fine-Grained Features 是 26\*26 大小的特征图 (最后一个 maxpooling 层的输入)，对于 Darknet-19 模型来说就是大小为 26\*26\*512 的特征图。passthrough 层与 ResNet 网络的 shortcut 类似，以前面更高分辨率的特征图为输入，然后将其连接到后面的低分辨率特征图上。前面的特征图维度是后面的特征图的 2 倍，passthrough 层抽取前面层的每个 2\*2 的局部区域，然后将其转化为 channel 维度，对于 26\*26\*512 的特征图，经 passthrough 层处理之后就变成了 13\*13\*2048 的新特征图 (特征图大小降低 4 倍，而 channels 增加 4 倍，图 6 为一个实例)，这样就可以与后面的 13\*13\*1024 特征图连接在一起形成 13\*13\*3072 的特征图，然后在此特征图基础上卷积做预测。在 YOLO 的 C 源码中，passthrough 层称为 reorg layer。在 TensorFlow 中，可以使用 `tf.extract_image_patches` 或者 `tf.space_to_depth` 来实现 passthrough 层

## 多尺度训练

由于 YOLOv2 模型中只有卷积层和池化层，所以 YOLOv2 的输入可以不限于 416\*416 大小的图片。为了增强模型的鲁棒性，YOLOv2 采用了多尺度输入训练策略，具体来说就是在训练过程中每间隔一定的 iterations 之后改变模型的输入图片大小。由于 YOLOv2 的下采样总步长为 32，输入图片大小选择一系列为 32 倍数的值输入图片。最小为 320\*320，此时对应的特征图大小为 10\*10，而输入图片最大为 608\*608，对应的特征图大小为 19\*19，在训练过程，每隔 10 个 iterations 随机选择一种输入图片大小，然后只需要修改对最后检测层的处理就可以重新训练。

采用 Multi-Scale Training 策略，YOLOv2 可以适应不同大小的图片，并且预测出很好的结果。在测试时，YOLOv2 可以采用不同大小的图片作为输入，在 VOC 2007 数据集上的效果如下图所示。可以看到采用较小分辨率时，YOLOv2 的 mAP 值略低，但是速度更快，而采用高分辨输入时，mAP 值更高，但是速度略有下降，对于 544\*544，mAP 高达 78.6%。注意，这只是测试时输入图片大小不同，而实际上用的是同一个模型 (采用 Multi-Scale Training 训练)。

Detection Frameworks	Train	mAP	FPS
Fast R-CNN [5]	2007+2012	70.0	0.5
Faster R-CNN VGG-16[15]	2007+2012	73.2	7
Faster R-CNN ResNet[6]	2007+2012	76.4	5
YOLO [14]	2007+2012	63.4	45
SSD300 [11]	2007+2012	74.3	46
SSD500 [11]	2007+2012	76.8	19
YOLOv2 $288 \times 288$	2007+2012	69.0	91
YOLOv2 $352 \times 352$	2007+2012	73.7	81
YOLOv2 $416 \times 416$	2007+2012	76.8	67
YOLOv2 $480 \times 480$	2007+2012	77.8	59
YOLOv2 $544 \times 544$	2007+2012	<b>78.6</b>	40

论文：

YOLO9000: Better, Faster, Stronger

<https://arxiv.org/pdf/1612.08242>

参考：

<https://blog.csdn.net/shanlepu6038/article/details/84778770>

<https://blog.csdn.net/anqian123321/article/details/82627332>

### (3) YOLOv3 (201804)

YOLOv3 在 YOLOv2 的基础上做了一些改动，包括如下：

#### 新的基础网络 Darknet-53

相较于上一版本中的 Darknet19，Darknet53 通过加入残差结构，大大增加了网络深度。此外 YOLOv3 中是完全没有池化层和全连接层的，张量的尺寸变化是通过改变卷积核的步长来实现的。

Type	Filters	Size	Output	
1x	Convolutional	32	$3 \times 3$	$256 \times 256$
	Convolutional	64	$3 \times 3 / 2$	$128 \times 128$
	Convolutional	32	$1 \times 1$	
	Convolutional	64	$3 \times 3$	
2x	Residual			$128 \times 128$
	Convolutional	128	$3 \times 3 / 2$	$64 \times 64$
	Convolutional	64	$1 \times 1$	
	Convolutional	128	$3 \times 3$	
8x	Residual			$64 \times 64$
	Convolutional	256	$3 \times 3 / 2$	$32 \times 32$
	Convolutional	128	$1 \times 1$	
	Convolutional	256	$3 \times 3$	
8x	Residual			$32 \times 32$
	Convolutional	512	$3 \times 3 / 2$	$16 \times 16$
	Convolutional	256	$1 \times 1$	
	Convolutional	512	$3 \times 3$	
4x	Residual			$16 \times 16$
	Convolutional	1024	$3 \times 3 / 2$	$8 \times 8$
	Convolutional	512	$1 \times 1$	
	Convolutional	1024	$3 \times 3$	
	Residual			$8 \times 8$
	Avgpool		Global	
	Connected		1000	
	Softmax			

Table 1. Darknet-53.

### 多尺度检测

Predictions across scales，YOLOv2 中使用 passthrough 结构来进行检测细粒度特征，在 v3 中得到了进一步发扬光大，吸取了 FPN 的特点，YOLOv3 输出 3 个不同尺度的特征图，分别是  $13 \times 13$ ， $26 \times 26$ ， $52 \times 52$ ，

论文：

YOLOv3: An Incremental Improvement

<https://arxiv.org/pdf/1804.02767.pdf>

github：第三方团队在 Pytorch 上复现的 YOLOv3

<https://github.com/ultralytics/yolov3>

## (4) YOLOv4 (202004)

YOLOv3 之后其作者 Joseph Redmon 宣布退出 CV 研究（因为他觉得自己的开源算法被用在军事上），于是 YOLOv4 的作者 Alexey 就“接手”了之后的开发。Alexey 跟 Redmon 认识，而且之前就做了很多 YOLO 的工作，主要是一些工程化方面的东西，比如移植到 Windows、fix bug、增加模型等等，Alexey 的 github 库也是除官方之外最权威的。

YOLOv4 中没有之前几个版本的 YOLO 的创新性的改进，更多的贡献是在工程化的方面的，在 YOLOv3 之上测试并应用了很多新出来的技术（数据增强、激活函数、Loss 函数、dropout 等等）。

论文：

<https://arxiv.org/pdf/2004.10934.pdf>

github：Alexey 版本的各个 YOLO

<https://github.com/AlexeyAB/darknet>

参考：

<https://www.zhihu.com/question/390194081>

<https://zhuanlan.zhihu.com/p/135899403>

## (5) YOLOv5 (Ultralytics)

YOLOv5 并不是一篇论文。

如果说 YOLOv3 是官方（Joseph Redmon），YOLOv4 是半官方（AlexeyAB），那 YOLOv5 就是民间的了，其作者是 Ultralytics，这家公司之前就一直在做 YOLO 上的各种工作，包括移植到 Pytorch 上，之前对 YOLOv3 的移植在 github 上也获得了各种移植版本中最高的 star：<https://github.com/ultralytics/yolov3>

事实上 v5 这个称号也是它自封的，和 YOLOv4 一样，YOLOv5 也是在 YOLOv3 之上加了各种 tricks 形成的，但是并没有发表论文，具体的 tricks 目前尚未看到介绍。

和 YOLOv4 最大的区别在于它是 Pytorch 实现的。

github：

<https://github.com/ultralytics/yolov5>

## 6 · SSD (201512)

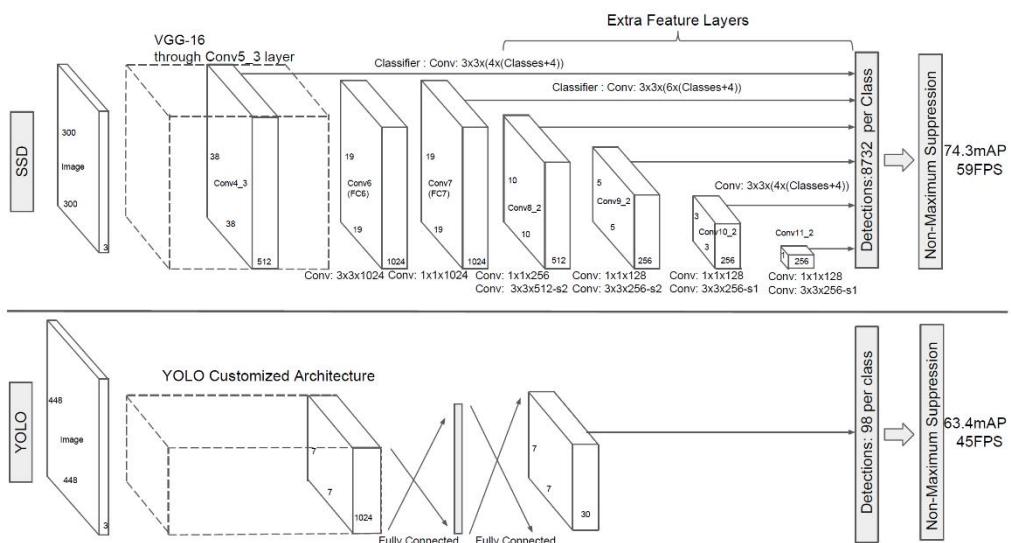
SSD 同 YOLO 一样，是个一步检测的框架，其特点在于：

- 相较于 YOLO，提供了更高的准确性，可以达到和两步检测方法（比如 Faster R-CNN）差不多的水准（SSD 出来的时候 YOLOv2 尚未出来）
- SSD 的关键是用一组缺省的边界框来预测类别置信度和边界，这些缺省边界框是通过将小的卷积核应用在特征图上得出的
- 在不同尺度的特征图上得到缺省边界框，以及不同宽高比的边界框
- 以上设计使得整个网络是端到端的，易于训练，高准确率，即便是在低分辨率的图上，进一步提高了速度/准确性

SSD 使用一个传统的图像分类网络（这里是 VGG）作为基础网络，最后用于分类的层被去掉，此外全连接层 FC6 和 FC7 被替换为卷积层，还增加了以下辅助结构来产生目标检测结果：

- 通过在不同尺寸的特征图（feature map，即不同卷积层）上追加特征层（feature layer）来实现对不同尺寸物体的检测：在大的（即靠前的）特征图上预测小物体，在小的（靠后的）特征图上预测大的物体。
- 这些特征层可以产生固定数量的预测。特征层是卷积层，一个  $m \times n \times p$  的特征层（ $p$  为通道数）所对应的卷积核为  $3 \times 3 \times p$ 。对于每个预测位置 ( $m \times n$  个) 的每个通道 ( $p$  个)，会产生某个类别的置信度，或者对位置的预测 ( $x, y, w, h$ )
- 每个预测位置可以产生不同长宽比的若干个 ( $k$  个) 先验框，与单个先验框相关的输出有  $c+4$  个， $c$  为类别数，表示每个类别的置信度，4 为坐标，分别为中心位置 ( $x, y$ ) 和长宽 ( $w, h$ )。因此共有  $(c+4) \times k$  个卷积核被应用到特征图的每个预测位置上，从而对于每个特征层，有  $p = (c+4) \times k$  个通道。这个先验框（default box）的想法来自于 Faster R-CNN 中的锚点（anchor box）

下图为 SSD 的结构，以及 YOLO 的结构



再用简单的描述总结一下：

- 基础网络上的 N 层（特征图）可以产生若干预测，越靠前的层预测越小的物体，越靠后的层预测越大的物体；
  - 每层有  $m \times n$  个位置（特征图大小）；
  - 每个位置可以产生 k 个预测（不同长宽比的先验框）；
  - 每个先验框包括  $c+4$  个输出（c 为类别数），4 对应每个类别的置信度和中心坐标（x, y）和长宽（w, h）。
- 每个特征图总共有  $m \times n \times k \times (c+4)$  个输出

### 先验框匹配策略

在训练过程中，首先要确定训练图片中的 ground truth（真实目标）与哪个先验框来进行匹配，与之匹配的先验框所对应的边界框将负责预测它。在 Yolo 中，ground truth 的中心落在哪个单元格，该单元格中与其 IOU 最大的边界框负责预测它。但是在 SSD 中却完全不一样，SSD 的先验框与 ground truth 的匹配原则主要有两点。

- 对于图片中每个 ground truth，找到与其 IOU 最大的先验框，该先验框与其匹配，这样，可以保证每个 ground truth 一定与某个先验框匹配。通常称与 ground truth 匹配的先验框为正样本，反之，若一个先验框没有与任何 ground truth 进行匹配，那么该先验框只能与背景匹配，就是负样本。
- 第二个原则是，对于剩余的未匹配先验框，若某个 ground truth 的 IOU 大于某个阈值（一般是 0.5），那么该先验框也与这个 ground truth 进行匹配。这个原则简化了学习过程。而这意味着某个 ground truth 可能与多个先验框匹配，这是可以的。但是反过来却不可以，因为一个先验框只能匹配一个 ground truth，如果多个 ground truth 与某个先验框 IOU 大于阈值，那么先验框只与 IOU 最大的那个先验框进行匹配。第二个原则一定在第一个原则之后进行，仔细考虑一下这种情况，如果某个 ground truth 所对应最大 IOU 小于阈值，并且所匹配的先验框却与另外一个 ground truth 的 IOU 大于阈值，那么该先验框应该匹配谁，答案应该是前者，首先要确保某个 ground truth 一定有一个先验框与之匹配。但是，这种情况我觉得基本上是不存在的。由于先验框很多，某个 ground truth 的最大 IOU 肯定大于阈值，所以可能只实施第二个原则既可了。

### 损失函数

损失函数定义为位置误差（loc）与置信度误差（conf）的加权和：

$$L(x, c, l, g) = \frac{1}{N} (L_{conf}(x, c) + \alpha L_{loc}(x, l, g))$$

其中 N 是先验框的正样本数量。这里  $x_{pij} \in \{1, 0\}$  为一个指示参数，当  $x_{pij}=1$  时表示第 i 个先验框与第 j 个 ground truth 匹配，并且 ground truth 的类别为 p。c 为类别置信度预测值。l 为先验框的所对应边界框的位置预测值，而 g 是 ground truth 的位置参数。对于位置误差，其采用 Smooth L1 loss，定义如下

$$L_{loc}(x, l, g) = \sum_{i \in Pos}^N \sum_{m \in \{cx, cy, w, h\}} x_{ij}^k \text{smooth}_{L_1}(l_i^m - \hat{g}_j^m)$$

$$\hat{g}_j^{cx} = (g_j^{cx} - d_i^{cx})/d_i^w \quad \hat{g}_j^{cy} = (g_j^{cy} - d_i^{cy})/d_i^h$$

$$\hat{g}_j^w = \log\left(\frac{g_j^w}{d_i^w}\right) \quad \hat{g}_j^h = \log\left(\frac{g_j^h}{d_i^h}\right)$$

$$\text{smooth}_{L_1}(x) = \begin{cases} 0.5x^2 & \text{if } |x| < 1 \\ |x| - 0.5 & \text{otherwise,} \end{cases}$$

### 数据增广

采用 **数据扩增 (Data Augmentation)** 可以提升 SSD 的性能，主要采用的技术有水平翻转 (horizontal flip)，随机裁剪加颜色扭曲 (random crop & color distortion)，随机采集块域 (Randomly sample a patch) (获取小目标训练样本) 等。

论文：

SSD: Single Shot MultiBox Detector

<https://arxiv.org/pdf/1512.02325>

参考：

<https://blog.csdn.net/xiaohu2022/article/details/79833786>

## 7 · FPN (201612)(TODO)

**特征金字塔网络 (Feature Pyramid Network)**

论文：

<https://arxiv.org/abs/1612.03144>

参考：

代码：

## 8 · RetinaNet (201708)

RetinaNet 的主要贡献是 **Focal Loss**，参考《[网络构成 > 损失函数 > Focal Loss](#)》

论文：

<https://arxiv.org/pdf/1708.02002.pdf>

## 9 · MaskX R-CNN (201711)(TODO)

论文：

<https://arxiv.org/pdf/1711.10370.pdf>

## 10 · CenterNet (201904)(TODO)

论文：

<https://arxiv.org/pdf/1904.07850.pdf>

代码：

<https://github.com/xingyizhou/CenterNet>

## 11 · EfficientDet (201911)(TODO)

EfficientDet 是以 EfficientNet 为 backbone 实现的目标检测网络

论文：

<https://arxiv.org/pdf/1911.09070.pdf>

参考：

<https://zhuanlan.zhihu.com/p/129016081>

代码：

<https://github.com/zlyol117/Yet-Another-EfficientDet-Pytorch>

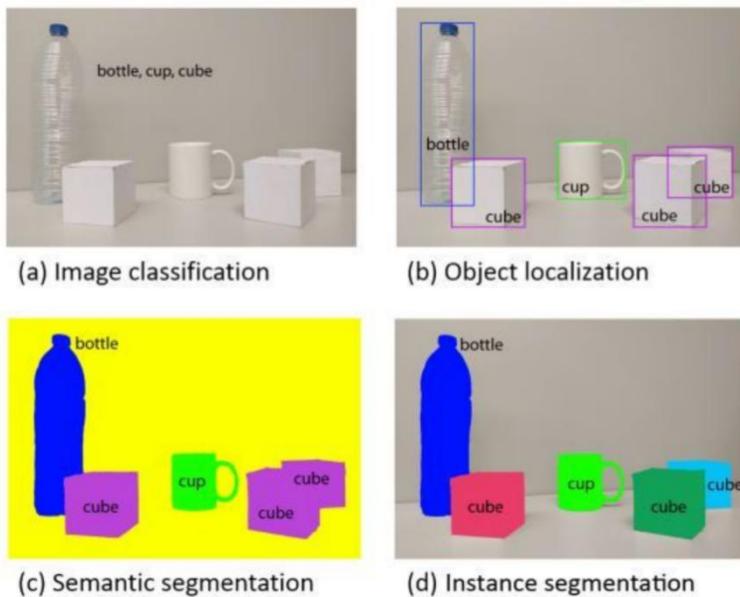
## (三) 图像分割 (Image Segmentation)

图像分类识别整个图片，目标检测识别图片中的物体，图像分割则是更进一步的任务——它的精细度到了单个像素，图像分割的目的是找出图片中各个物体的轮廓。

图像分割有几个子任务：

- 语义分割 (Semantic Segmentation)：按照物体的类别分割
- 实例分割 (Instance Segmentation)：按照单个物体进行分割
- 全景分割 (Panoptic Segmentation)：前景实例分割+背景语义分割

请注意区别语义分割 (Semantic Segmentation) 和实例分割 (Instance Segmentation) 的区别，语义分割按类别区分，而实例分割按个体区分，例如，当图像中有多只猫时，语义分割会将多只猫整体的所有像素预测为“猫”这个类别。与此不同的是，实例分割需要区分出哪些像素属于第一只猫、哪些像素属于第二只猫。下图中左下为语义分割，右下为实例分割：



一般的图像分割架构可以被认为是一个编码器-解码器网络，编码器通常是一个预训练的分类网络（VGG、ResNet 等），后面接一个解码器网络，它的任务是将编码器学到的低分辨率的可判别特征投射到高分辨率的像素空间，以获得清晰的类别区分。

参考：

<https://meetshah1995.github.io/semantic-segmentation/deep-learning/pytorch/vision/2017/06/01/semantic-segmentation-over-the-years.html>

上文翻译：[https://blog.csdn.net/qq\\_20084101/article/details/80432960](https://blog.csdn.net/qq_20084101/article/details/80432960)

## 1 · FCN (2014)

FCN (全卷积网络，Fully Convolutional Network) 在近几年的语义分割论文中有开创性的意义，之后的几篇语义分割相关的网络都是基于 FCN 的理论基础。FCN 是一个编码器-解码器网络，编码器是一个预训练的分类网络（VGG、ResNet 等），后面接一个解码器网络。

FCN 这个概念（网络全都是卷积构成）在其他的研究方向也有应用。

FCN 的特点包括：

- 分类网络中的全连接层被替换为全卷积层以保持特征图维度（FC 输出形状是一维，卷积层输出形状是二维）
- 输入给解码器的特征是由编码器的不同阶段合并而成，这些阶段的语义特征有着不同的粗糙程度。
- 低分辨率的语义特征的上采样是通过经双线性插值滤波器初始化的反卷积实现的
- 从 VGG、AlexNet 等分类器网络进行知识迁移来实现语义细分

下图可以看到不同粗糙程度的阶段有不同的上采样倍数（FCN-32s、FCN-16s、FCN-8s）

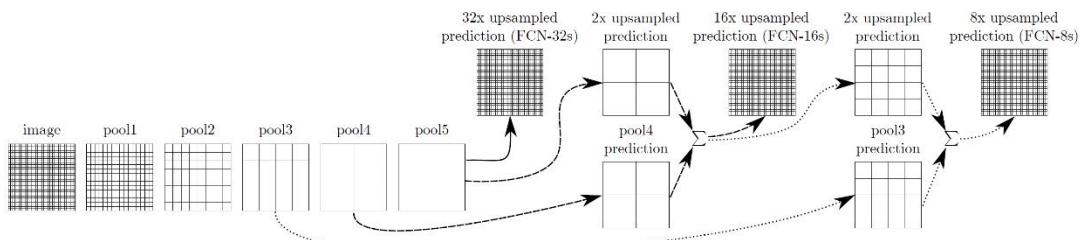


Figure 3. Our DAG nets learn to combine coarse, high layer information with fine, low layer information. Layers are shown as grids that reveal relative spatial coarseness. Only pooling and prediction layers are shown; intermediate convolution layers (including our converted fully connected layers) are omitted. Solid line (FCN-32s): Our single-stream net, described in Section 4.1, upsamples stride 32 predictions back to pixels in a single step. Dashed line (FCN-16s): Combining predictions from both the final layer and the pool4 layer, at stride 16, lets our net predict finer details, while retaining high-level semantic information. Dotted line (FCN-8s): Additional predictions from pool3, at stride 8, provide further precision.

参考：

<https://www.cnblogs.com/xuanxufeng/p/6249834.html>

论文：

Fully Convolutional Networks for Semantic Segmentation

<https://arxiv.org/pdf/1605.06211.pdf>

## 2 · SegNet (201511)

SegNet 的新颖之处在于解码器对其较低分辨率的输入特征图进行上采样的方式。具体地说，解码器使用了在相应编码器的最大池化步骤中计算的池化索引来执行非线性上采样。这种方法消除了学习上采样的需要。经上采样后的特征图是稀疏的，因此随后使用可训练的卷积核进行卷积操作，生成密集的特征图。我们将我们所提出的架构与广泛采用的 FCN 以及众所周知的 DeepLab-LargeFOV，DeconvNet 架构进行比较。比较的结果揭示了在实现良好的分割性能时所涉及的内存与精度之间的权衡。

简单地说，SegNet 的特点是：

- 上采样使用了反池化层

其网络架构如下：

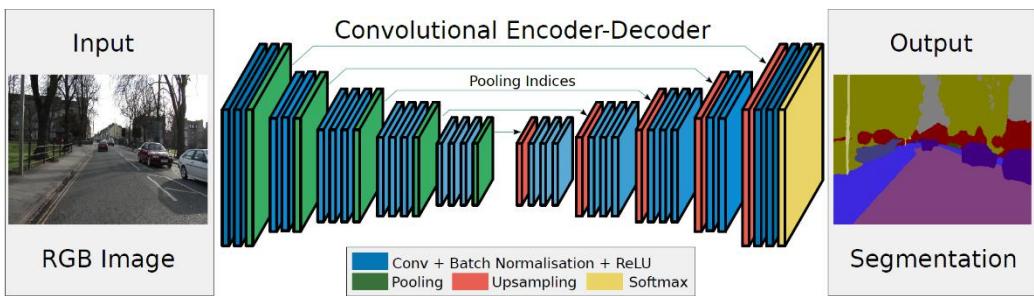


Fig. 2. An illustration of the SegNet architecture. There are no fully connected layers and hence it is only convolutional. A decoder upsamples its input using the transferred pool indices from its encoder to produce a sparse feature map(s). It then performs convolution with a trainable filter bank to densify the feature map. The final decoder output feature maps are fed to a soft-max classifier for pixel-wise classification.

论文：

SegNet: A Deep Convolutional Encoder-Decoder Architecture for Image Segmentation

<https://arxiv.org/pdf/1511.00561>

### 3 · DeepLab 系列(TODO)

(1) DeepLab (201412)

(2) DeepLabV2 (201606)

(3) DeepLabV3 (201706)

(4) DeepLabV3+ (201802)

## 4 · Mask R-CNN (201703)

以 Faster R-CNN 为基础，在现有的边界框识别分支基础上添加一个并行的预测目标掩码的分支。Mask R-CNN 很容易训练，仅仅在 Faster R-CNN 上增加了一点小开销，运行速度为 5fps。此外，Mask R-CNN 很容易泛化至其他任务，例如，可以使用相同的框架进行姿态估计。我们在 COCO 所有的挑战赛中都获得了最优结果，包括实例分割，边界框目标检测，和人关键点检测。在没有使用任何技巧的情况下，Mask R-CNN 在每项任务上都优于所有现有的单模型网络，包括 COCO 2016 挑战赛的获胜者。

Mask R-CNN 的特点：

- 在 Faster R-CNN 上添加辅助分支以执行语义分割
- 对每个实例进行的 RoIPool 操作被修改为 RoIAAlign，它避免了特征提取的空间量化，因为在最高分辨率中保持空间特征不变对于语义分割很重要。
- Mask R-CNN 与 Feature Pyramid Networks（类似于 PSPNet，它对特征使用了金字塔池化）相结合，在 MS COCO 数据集上取得了最优结果。

论文：

Mask R-CNN

<https://arxiv.org/pdf/1703.06870>

## 5 · PointRend (201912) (TODO)

论文：

PointRend: Image Segmentations as Rendering

<https://arxiv.org/pdf/1912.08193.pdf>

## (四) 人脸识别和建模 (Face Recognition)

**Facial Recognition and Modelling**，人脸相关的神经网络，大致有以下几个部分，这些概念之间有交叉的部分，同样的技术和网络也可能用在不同部分上。

- **人脸检测 (Face Detection)**：找出图片中的人脸，输入是图片，输出是人脸的位置和大小。通常情况下其它人脸相关的深度学习会需要人脸检测作为第一步。人脸检测也可以单独作为一个模块使用（比如拍照时候的自动对焦）。
- **人脸对齐 (Face Alignment)**：将人脸图片标准化，包括 2D 对齐（歪->正）和 3D 对齐（侧脸->正脸）。人脸对齐的输入是人脸的图片，输出是对齐后的人脸。现在有些神经网络已经不需要人脸对齐这个步骤，而是直接用未对齐的人脸图片做端到端的计

算。

- **人脸验证 (Face Verification)**：验证两张图片中的人脸是否为同一人（1：1），输入是通常是两张人脸图片（对齐或未对齐），输出是为同一人的概率。具体步骤包括人脸图片的特征提取，以及两个特征值的比对。
- **人脸识别 (Face Identification)**：识别出图片中人脸是谁（1：N）。输入是一个人脸图片，以及一个特征值数据库，输出为该人脸是谁（或者谁都不是）的概率向量？
- **人脸聚类 (Face Clustering)**：找出图片中哪些人脸是同一个人（非监督学习）
- **表情识别 (Facial Expression Recognition)**：识别人脸的表情

相较于以上的概念，**Face Recognition** 倒是一个相对不严格的定义，有时候指代 **Face Identification**，有时候指代人脸验证和 **Face Identification**，有时候包括前面人脸检测和人脸对齐环节，有时候则不包括。

**人脸检测**和**人脸对齐**是其它部分的前导环节，之后通过不同的特征提取手段，在后来的一些人脸相关的 NN 上，人脸对齐这个阶段有时候会被忽略，直接将检测出来的人脸图片拿去提取特征。

早期的人脸相关的研究，有很多非神经网络的特征提取方法，比如 **LBP**、**HOG**、**SIFT** 等。而后渐渐增大神经网络在整个模型中的比例，这在早期的论文中可以明显的看到 NN 的部分占比越来越大，现在基本都是用神经网络完成端到端的工作了。

相对于 **close-set** 人脸识别的数据，对 **Open-set** 人脸识别的数据，有一个准则，即**最大的 intra-class 距离要小于最小的 inter-class 距离**（这个准则应该可以普及到所有 open-set 分类问题）。这可以通俗的理解为，**close-set** 的数据必然会落入某个分类，因此类与类之间不能有空隙，不然不知道该分入哪个类，而 **open-set** 的数据则得留下足够大的空隙，不然应该被标记为“**unknown**”的陌生数据也会落入某个已知分类。

比较近期的论文（**FaceNet**，**Center loss**，**SphereFace**，**CosFace**，**ArcFace** 等）大都是在发掘不同的损失函数，以使得提取出来的特征值不仅是可分离的（**Separable**），而且特征判别度高（**Discriminative**），以提高在 **open-set** 的情况下的识别能力。这些损失函数要么是通过增加提高类间的**欧式边缘 (Euclidean margin)**，要么是通过增加类间的角度边缘 (**angular margin**) 来实现。

Github 上的一个人脸相关的资料仓库：

[https://github.com/ChanChiChoi/awesome-Face\\_Recognition](https://github.com/ChanChiChoi/awesome-Face_Recognition)

参考：

<https://blog.csdn.net/tkyjqh/article/details/70139718>

<https://paperswithcode.com/task/face-verification>

# 1 · 数据集

人脸相关数据集介绍

## (1) LFW

**Labelled Face in the Wild**，LFW 人脸数据库主要用来研究非受限情况下的人脸识别问题。LFW 主要是从互联网上搜集图像，一共含有 13000 多张人脸图像，这些人脸图片有着不同的表情、朝向、光照，分别属于 5000 多个人，每张图像都被标识出对应的人的名字，其中有 1680 人对应不只一张图像。因此 LFW 被广泛用于衡量 Face Verification 的准确性。

LFW 可以随机生成一个 pairs.txt 文件，包含了 6000 对图片的 ID，其中 3000 对为同一个人，格式为“名字 n1 n2”，另外 3000 对为不同的人，格式为“名字 1 n1 名字 2 n2”

网站：<http://vis-www.cs.umass.edu/lfw/>

## (2) YTF (Youtube Faces)

采集自 Youtube 的脸部视频数据集，包括 3000 多段视频，其中有 1595 个人。

官网：<https://www.cs.tau.ac.il/~wolf/ytfaces/>

## (3) CelebA

CelebA 是 **CelebFaces Attribute** 的缩写，意即名人人脸属性数据集，其包含 10,177 个名人身份的 202,599 张人脸图片，每张图片都做好了特征标记，包含人脸 bbox 标注框、5 个人脸特征点坐标以及 40 个属性标记。这些属性包括“秃头”、“眼镜”、“双下巴”等等。

CelebA 由香港中文大学开放提供，广泛用于人脸相关的计算机视觉训练任务，可用于人脸属性标识训练、人脸检测训练以及 landmark 标记等。

官方网址：<http://mm1lab.ie.cuhk.edu.hk/projects/CelebA.html>

CelebFaces+

## (4) FDDB

FDDB (Face Detection Data Set and Benchmark) , FDDB 数据集主要用于约束人脸检测研究，该数据集选取野外环境中拍摄的 2845 个图像，从中选择 5171 个人脸图像。是一个被广泛使用的权威的人脸检测平台。

<http://vis-www.cs.umass.edu/fddb/index.html>

## (5) WIDER

WIDER FACE 是香港中文大学的一个提供更广泛人脸数据的人脸检测基准数据集，由 YangShuo , Luo Ping , Loy , Chen Change , Tang Xiaoou 收集。

1. 它包含 32203 个图像和 393703 个人脸图像，在尺度，姿势，闭塞，表达，装扮，关照等方面表现出了大的变化。
2. WIDER FACE 是基于 61 个事件类别组织的，对于每一个事件类别，选取其中的 40% 作为训练集，10% 用于交叉验证 (cross validation) ，50% 作为测试集。
3. 和 PASCAL VOC 数据集一样，该数据集也采用相同的指标。
4. 和 MALF 和 Caltech 数据集一样，对于测试图像并没有提供相应的背景边界框。

### ● 格式

Annotation 包中的 wider\_face\_train\_bbox\_gt.txt 和 wider\_face\_val\_bbox\_gt.txt 用于标记脸的位置大小等信息，格式如下：

文件路径

Bbox 数量

Bbox1 信息

Bbox2 信息

...

Bbox 信息格式为：x1, y1, w, h, blur, expression, illumination, invalid, occlusion, pose

官网：

<http://shuoyang1213.me/WIDERFACE/index.html>

## (6) AFW

Annotated Face in the Wild, AFW 数据集是使用 Flickr (雅虎旗下图片分享网站) 图像建立的人脸图像库，包含 205 个图像，其中有 473 个标记的人脸。对于每一个人脸都包含一个长方形边界框，6 个地标和相关的姿势角度。数据库虽然不大，额外的好处是作者给出了其 2012 CVPR 的论文和程序以及训练好的模型。

<http://www.ics.uci.edu/~xzhu/face/>

## (7) AFLW

Annotated Facial Landmarks in the Wild, AFLW 人脸数据库是一个包括多姿态、多视角的大规模人脸数据库，而且每个人脸都被标注了 21 个特征点。此数据库信息量非常大，包括了各种姿态、表情、光照、种族等因素影响的图片。AFLW 人脸数据库大约包括 25000 已手工标注的人脸图片，其中 59% 为女性，41% 为男性，大部分的图片都是彩色，只有少部分是灰色图片。该数据库非常适合用于人脸识别、人脸检测、人脸对齐等方面的研究，具有很高的研究价值。

<https://www.tugraz.at/institute/icg/research/team-bischof/1rs/downloads/aflw/>

## (8) IJB-A

IJB-A 是一个用于人脸检测和识别的数据库，包含 24327 个图像和 49759 个人脸。

<https://www.nist.gov/itl/iad/image-group/ijb-dataset-request-form>

## (9) MegaFace

MegaFace 资料集包含一百万张图片，代表 690000 个独特的人。所有数据都是华盛顿大学从 Flickr (雅虎旗下图片分享网站) 组织收集的。这是第一个在一千万规模级别的面部识别算法测试基准。现有脸部识别系统仍难以准确识别超过百万的数据量。为了比较现有公开脸部识别算法的准确度，华盛顿大学在去年年底开展了一个名为 “MegaFace Challenge” 的公开竞赛。这个项目旨在研究当数据库规模提升数个量级时，现有的脸部识别系统能否维持可靠的准确率。

<http://megaface.cs.washington.edu/dataset/download.html>

## 2 · 代码

这里介绍了一些开源的实现，因为都是工程方面的改进，因此没有相关论文

### (1) 超轻量级人脸检测 (201910)

Ultra Light Fast Generic Face Detector 是一个开源的大小只有 1MB 的人脸检测模型。

介绍及代码如下：

<https://github.com/Linzaer/Ultra-Light-Fast-Generic-Face-Detector-1MB>

## 3 · DeepFace (201406)

DeepFace 是 FaceBook 2014 年推出的一篇关于人脸验证的论文。被认为是神经网络使用在人脸识别领域的开山之作。

DeepFace 算法分成四个步骤：人脸检测、人脸对齐、特征提取（CNN）、分类。这其中特征提取部分完全由神经网络实现。DeepFace 所使用的人脸对齐算法比较复杂，并且在之后的研究中发现，跳过这个步骤，直接将检测到的人脸图片送到特征提取模块，效果也不差，因此在 DeepFace 之后的论文中大都没有使用人脸对齐这个部分。

在人脸识别领域，DeepFace 相较于之前的方法，是只使用 CNN 提取特征，而不是 LBP（局部二值模式），HOG 等方法。

整个流程的如下：

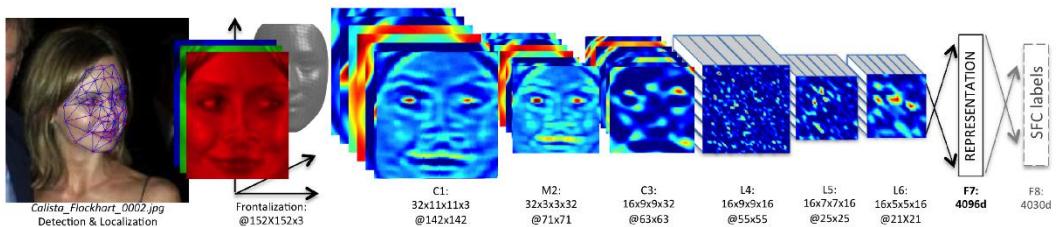


Figure 2. Outline of the DeepFace architecture. A front-end of a single convolution-pooling-convolution filtering on the rectified input, followed by three locally-connected layers and two fully-connected layers. Colors illustrate feature maps produced at each layer. The net includes more than 120 million parameters, where more than 95% come from the local and fully connected layers.

第一步人脸检测，第二步人脸对齐，后面是 8 层 CNN：卷积、最大池化、卷积，然后是 3 层本地连接层，再接 2 层全连接层。五层连接层的参数数量占了整个网络参数量的 95%。

论文：



1406\_DeepFace.  
pdf

参考：

<https://blog.csdn.net/stdcoutzyx/article/details/46776415>

[https://blog.csdn.net/hh\\_2018/article/details/80576290](https://blog.csdn.net/hh_2018/article/details/80576290)

[https://blog.csdn.net/hh\\_2018/article/details/80581612](https://blog.csdn.net/hh_2018/article/details/80581612)

## 4 · DeepID 系列 (2014)

**Deep ID** 是香港中文大学的汤晓欧（也是商汤的创始人）推出的人脸识别算法，一共出了 DeepID、DeepID2、DeepID2+、DeepID3 几个版本。

### (1) DeepID (2014)

DeepID 的亮点包括：

- 将特征的维数大幅度降低，只用了 160 维特征
- 使用同一个人脸图片的不同 patch (即部分) 作为测试集训练，提高了准确性
- 有个 Multi-scale 的概念，将最后两层卷积层和 160 维 DeepID 层连接
- 分别使用了联合贝叶斯 (Joint Bayesian) 和 NN 作为最后的分类器

DeepID 所用的网络整体框架如下：

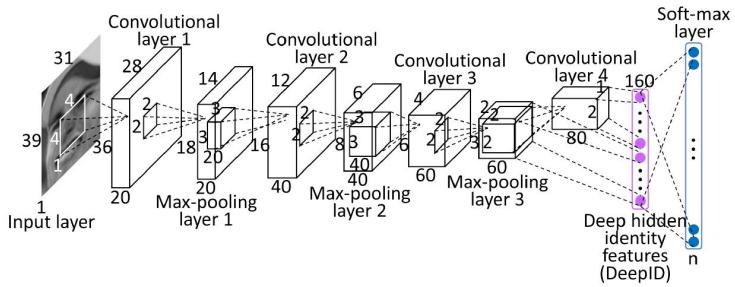


Figure 2. ConvNet structure. The length, width, and height of each cuboid denotes the map number and the dimension of each map for all input, convolutional, and max-pooling layers. The inside small cuboids and squares denote the 3D convolution kernel sizes and the 2D pooling region sizes of convolutional and max-pooling layers, respectively. Neuron numbers of the last two fully-connected layers are marked beside each layer.

最前面是  $39 \times 31 \times 3$  (或  $31 \times 31 \times 3$ ) 的输入，接着是 4 层卷积层+3 层最大池化层，然后是一个吐出 160 维特征值的全连接层（被称为 DeepID 层），最后一层是一个根据类别数量决定的全连接+Softmax 层。

注意图中最后一个隐藏层（即生成 160 维特征值的 FC 层）同时连接了第三个池化层和第四个卷积层，作者认为这样提高了对多种尺寸特征的感受，并且避免了这个特征 FC 层因为神经元太少而成为信息传播的瓶颈。

论文：

[Deep Learning Face Representation from Predicting 10,000 Classes](#)



[1400\\_DeepID.pdf](#)

参考：

[https://blog.csdn.net/a\\_1937/article/details/40425397](https://blog.csdn.net/a_1937/article/details/40425397)

<https://blog.csdn.net/stdcoutzyx/article/details/42091205>

## (2) DeepID2 (201406)

论文：

[Deep Learning Face Representation by Joint Identification-Verification](#)

<https://arxiv.org/pdf/1406.4773>

参考：

[https://blog.csdn.net/hh\\_2018/article/details/80540067](https://blog.csdn.net/hh_2018/article/details/80540067)

### (3) DeepID3

## 5 · FaceNet (201503)

FaceNet 是谷歌推出的人脸识别算法/结构，这个结构可以用来进行人脸的验证、识别和聚类。

论文：

FaceNet: A Unified Embedding for Face Recognition and Clustering

<https://arxiv.org/pdf/1503.03832.pdf>

代码：

<https://github.com/davidsandberg/facenet>

参考：

<https://blog.csdn.net/stdcoutzyx/article/details/46687471>

<https://medium.com/intro-to-artificial-intelligence/one-shot-learning-explained-using-facenet-dff5ad52bd38>

## 6 · MTCNN (201604)

MTCNN (Multi-Task CNN) 是一个比较著名的人脸检测+人脸对齐网络，它主要包括了 3 个级联的 CNN，分别简称 P-Net、R-Net 和 O-Net，MTCNN 的整体流程如下：

- 首先将输入图片缩放到不同的大小，以建立一个图像金字塔
- 用一个简单的全卷积网络 Proposal-Net (P-Net) 来找出候选区域。
- 接着从 P-Net 中选出的候选区域被送到更复杂一点的 Refine-Net (R-Net)，R-Net 会拒绝掉大量的伪候选区域，
- 最后从 R-Net 的输出会进入到最复杂的 Output-Net (O-Net)，O-Net 的工作和 P-Net 差不多，也是从输入滤除掉更多的伪候选区域，并对人脸框进行回归，除此之外还要给出脸部 landmark 的位置。

训练的时候，每个网络最后都分成三个分支，分别对应：人脸判定（是否是人脸）、人脸框回归（确定位置）、脸部特征（左眼、右眼、鼻子、左嘴角、右嘴角）定位，实际使用时候不用全部使用（应该是关闭 P-Net 和 R-Net 的脸部特征定位）。

论文：

Joint Face Detection and Alignment using Multi-task Cascaded Convolutional Networks

[https://arxiv.org/pdf/1604.02878](https://arxiv.org/pdf/1604.02878.pdf)

代码：

[https://github.com/kpzhang93/MTCNN\\_face\\_detection\\_alignment](https://github.com/kpzhang93/MTCNN_face_detection_alignment)

参考：

<https://blog.csdn.net/u014380165/article/details/78906898>

## 7 · CenterLoss (2016)

参考《[网络构成 > 损失函数 > Center Loss](#)》

## 8 · SphereFace (201704)

SphereFace 提出了一种基于角边缘 (angular margin) 分离的损失函数 A-Softmax Loss，用于使得特征值更加 discriminative (判别度高)，具体参考《[网络构成 > 损失函数 > A-Softmax Loss](#)》。

论文：

SphereFace: Deep Hypersphere Embedding for Face Recognition

[https://arxiv.org/pdf/1704.08063](https://arxiv.org/pdf/1704.08063.pdf)

代码：

<https://github.com/wyliu/sphereface>

## 9 · FacePoseNet (201708)

简称 FPN

## 10 · CosFace (201801)

## 11 · ArcFace/InsightFace (201801)

## 12 · SeqFace (201803)

## 13 · MobileFaceNets (201804)

**MobileFaceNet** 是一个轻量化的网络，是在 **MobileNet V2** 的基础上，针对人脸识别做了些改动。

最主要的改动是：大部分基础 CNN 最后都有一个 **global avg pooling**（全局平均池化层），但研究发现在人脸识别任务中，这层的存在反而降低了准确性，MobileFaceNet 论文给出了一个解释：因为人脸识别任务中，图片中间部分的重要性高于周围。为此 MobileFaceNet 将这层全局平均池化层改为全局 DW 卷积层（Global Depthwise Conv）

MobileFaceNet 结构如下：

**Table 1.** MobileFaceNet architecture for feature embedding. We use almost the same notations as MobileNetV2 [3]. Each line describes a sequence of operators, repeated  $n$  times. All layers in the same sequence have the same number  $c$  of output channels. The first layer of each sequence has a stride  $s$  and all others use stride 1. All spatial convolutions in the bottlenecks use  $3 \times 3$  kernels. The expansion factor  $t$  is always applied to the input size. GDCConv $7 \times 7$  denotes GDCConv of  $7 \times 7$  kernels.

Input	Operator	$t$	$c$	$n$	$s$
$112^2 \times 3$	conv3x3	-	64	1	2
$56^2 \times 64$	depthwise conv3x3	-	64	1	1
$56^2 \times 64$	bottleneck	2	64	5	2
$28^2 \times 64$	bottleneck	4	128	1	2
$14^2 \times 128$	bottleneck	2	128	6	1
$14^2 \times 128$	bottleneck	4	128	1	2
$7^2 \times 128$	bottleneck	2	128	2	1
$7^2 \times 128$	conv1x1	-	512	1	1
$7^2 \times 512$	linear GDCConv $7 \times 7$	-	512	1	1
$1^2 \times 512$	linear conv1x1	-	128	1	1

论文：

MobileFaceNets: Efficient CNNs for Accurate RealTime Face Verification on Mobile Devices

<https://arxiv.org/pdf/1804.07573>

参考：

[https://blog.csdn.net/Fire\\_Light/article/details/80279342](https://blog.csdn.net/Fire_Light/article/details/80279342)

<https://www.jianshu.com/p/10fd3e32fc91>

## (五) 人体姿态估计 (Pose Estimation)

又叫人体关键点检测，人体姿态估计是计算机视觉中一个很基础的问题。从名字的角度来看，可以理解为对人体的姿态（关键点，比如头，左手，右脚等）的位置估计。一般我们可以这个问题再具体细分成 4 个任务：

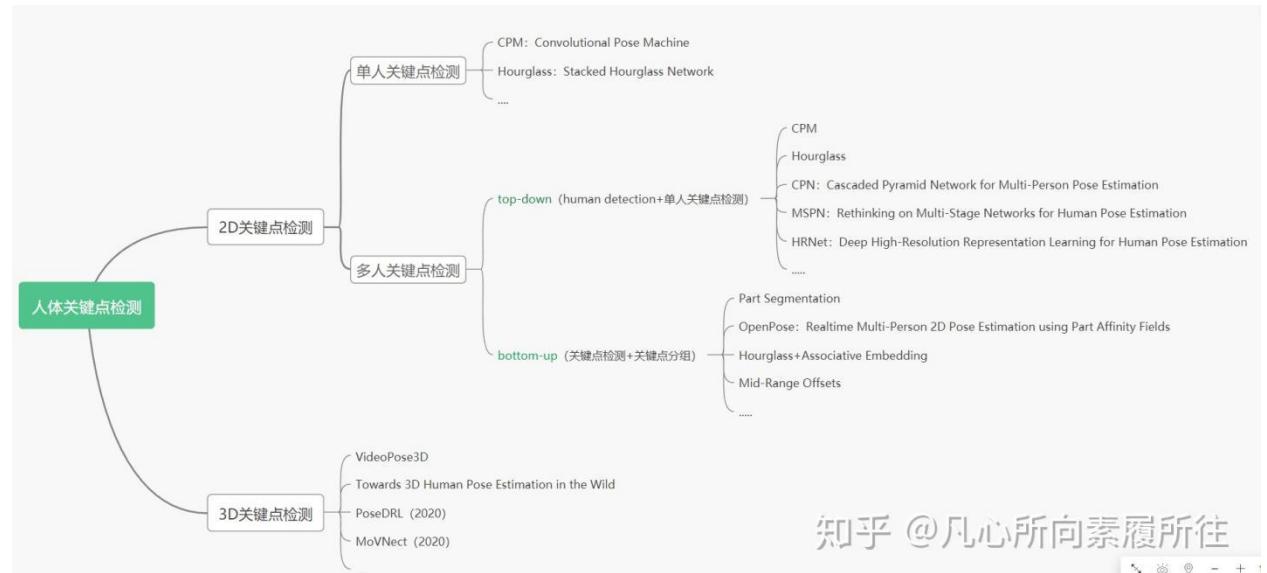
- 单人姿态估计 (Single-Person Skeleton Estimation)
- 多人姿态估计 (Multi-person Pose Estimation)
- 人体姿态跟踪 (Video Pose Tracking)
- 3D 人体姿态估计 (3D Skeleton Estimation)

单人姿态估计的输入是已经 crop 过的人体图像（相当于行人检测的结果），这个方向单独应用并不是很实用，通常在论文中都是多人姿态估计的一个子任务。

应用最广泛的是多人姿态估计，多人姿态估计算法大致可以分为两种：

- Top-down：先通过目标检测，从图片中找到人体，抠出来，再送到网络中进行姿态估计。换言之，top-down 是将多人姿态估计的问题转化为多个单人姿态估计的问题。
- Bottom-up：先找出图片中所有关键点，然后对关键点进行分组，得到一个个人。

具体算法分类如下：



参考：

人体关键点检测（姿态估计）简介+分类汇总

<https://zhuanlan.zhihu.com/p/102457223>

人体姿态估计(Human Pose Estimation)经典方法整理

<https://zhuanlan.zhihu.com/p/104917833>

人体姿态估计的过去，现在，未来

<https://zhuanlan.zhihu.com/p/85506259>

重新思考人体姿态估计 Rethinking Human Pose Estimation

<https://zhuanlan.zhihu.com/p/72561165>

## 1 · 数据集

MPII

COCO

## 2 · CPM (201602)

论文：

<https://arxiv.org/pdf/1602.00134.pdf>

参考：

<https://zhuanlan.zhihu.com/p/102468356>

代码：

<https://github.com/timccho/convolutional-pose-machines-tensorflow>

### 3 · HourGlass (201603)

论文：

<https://arxiv.org/pdf/1603.06937.pdf>

参考：

<https://zhuanlan.zhihu.com/p/102470330>

代码：

<https://github.com/Naman-ntc/Pytorch-Human-Pose-Estimation>

### 4 · OpenPose (201611)

论文：

<https://arxiv.org/pdf/1611.08050.pdf>

参考：

<https://zhuanlan.zhihu.com/p/102472863>

代码：

旧：[https://github.com/ZheC/Realtime\\_Multi-Person\\_Pose\\_Estimation](https://github.com/ZheC/Realtime_Multi-Person_Pose_Estimation)

新：<https://github.com/CMU-Perceptual-Lab/openpose>

### 5 · CPN (201711)

论文：

<https://arxiv.org/pdf/1711.07319.pdf>

参考：

<https://zhuanlan.zhihu.com/p/102475395>

代码：

<https://github.com/chenyilun95/tf-cpn>

## 6 · MSPN (201901)

论文：

<https://arxiv.org/pdf/1901.00148.pdf>

参考：

<https://zhuanlan.zhihu.com/p/102494631>

代码：

<https://github.com/megvii-detection/MSPN>

## 7 · HRNet (201902)

论文：

<https://arxiv.org/pdf/1902.09212.pdf>

参考：

<https://zhuanlan.zhihu.com/p/102494979>

代码：

<https://github.com/leoxiaobin/deep-high-resolution-net.pytorch>

# 八、研究方向：视频

视频理解的主要关注点是人的行为，当然也可以应用到非人类出现的场景。大致可以分为以下几个方向：

- **行为识别**：Action Recognition，又叫 Video Action Recognition 或者 Video Classification，就是给一段视频分类，差不多对应图像处理中的图像分类。解决了 What 的问题，但是跟图像分类一样，在实际应用中不是很多。
- **时序行为检测**：Temporal Action Detection，给定一段视频，要检测其中的行为片段，也就是这段视频里包括几个行为，每个行为的开始和结束的时间点是什么。这个方向比行为识别高级，解决了 What 和 When。
- **时空行为检测**：Spatio-temporal Action Detection，这个相较于时序行为检测又高级了一些，对给定视频，要检测出其中每个行为开始和结束的时间点，以及每一帧中该行为的位置（bounding box），解决了 What、When 和 Where 的问题。
- **视频描述**：Video Caption，根据视频生成描述
- **视频问答**：Video QA，这是视频理解的终极王者，给定一段视频和一个问题，NN 可以给出回答，比如“视频中的女孩一共跳了几下”

## (一) 行为识别 (Video Action Classification)

视频动作分类，Video Action Classification，或者行为识别，Action Recognition，或者视频识别，Video Recognition，是 NN 在 CV 应用中，关于视频的一个最简单的方向，这个应用的目的是在给定一小段视频的情况下，预测其所属的行为类别（类似图像分类）。

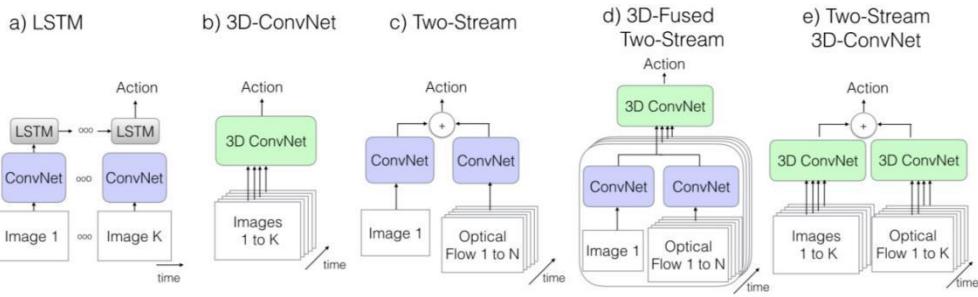
在 2014 年之前，深度学习就应用在视频理解上，但是效果并不好（没有传统方法好，比如 iDT）。

直到 2014 年双流 (two-stream) 网络出来，时空双网络也成为视频理解的一大流派。之后又有 TSN 等网络走这条路。

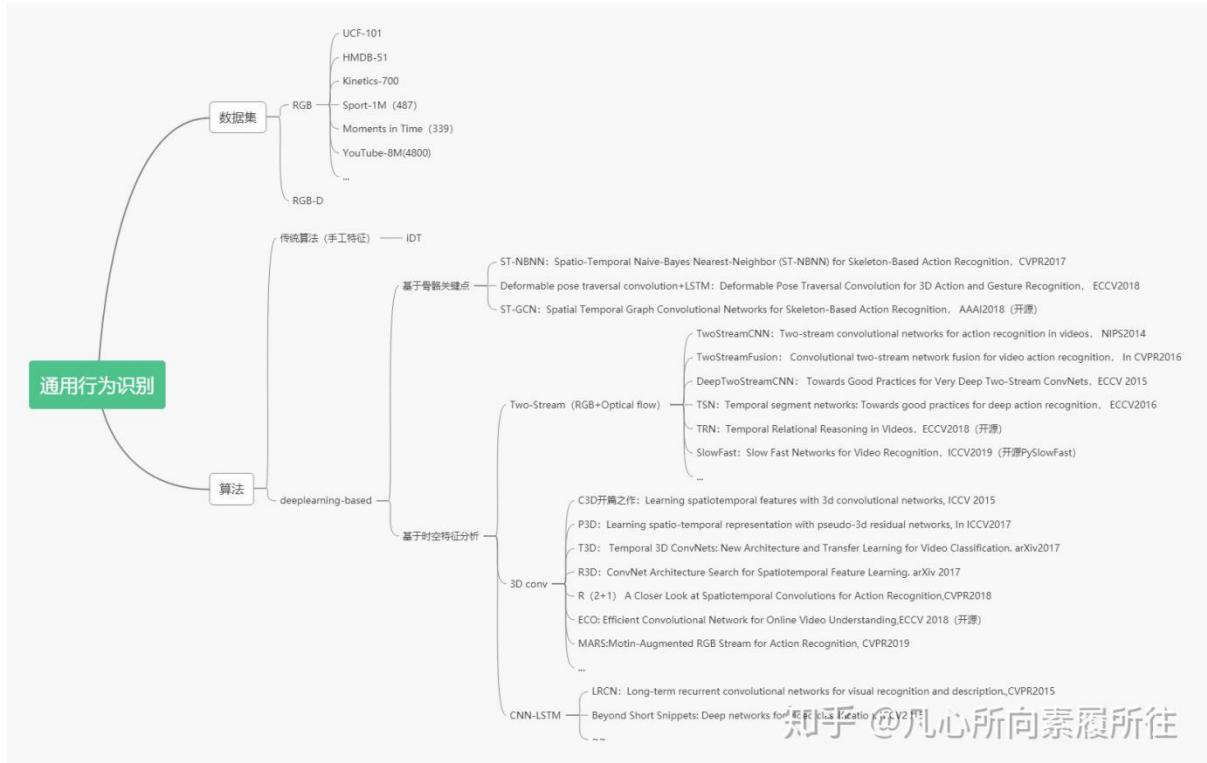
视频理解的另外一个方向是 3D 网络，起始于 C3D，即加上了时间方向（帧）的 3D 卷积。

目前的视频理解论文大都基于这两个方向，也有两者的融合。

除了 3D 网络和双流网络之外，还有两个小点的方向，LSTM+CNN 和 skeleton-based（即人体姿态估计 Pose Estimation），LSTM 的方向很久都没什么 paper 出来，而 skeleton 方向缺乏数据集。



具体的包括：



参考：

视频理解近期研究进展

<https://zhuanlan.zhihu.com/p/36330561>

<https://zhuanlan.zhihu.com/p/33040925>

简评 | Video Action Recognition 的近期进展

<https://zhuanlan.zhihu.com/p/59915784>

一文了解通用行为识别 ActionRecognition：了解及分类

<https://zhuanlan.zhihu.com/p/103566134>

基于 3D 骨架的深度学习行为识别综述  
<https://zhuanlan.zhihu.com/p/107983551>

Deep learning for video action recognition review  
<http://blog.qure.ai/notes/deep-learning-for-videos-action-recognition-review>

PWC：  
<https://paperswithcode.com/task/action-recognition-in-videos>

## 1 · 数据集

参考：  
<https://zhuanlan.zhihu.com/p/36330561>

### (1) UCF101

- 101 类、13320 个视频剪辑、每个视频不超过 10 秒、共 27 小时、分辨率 320\*240、共 6.5 GB。
- 视频源于 YouTube，内容包含化妆刷牙、爬行、理发、弹奏乐器、体育运动五大类。
- 每类动作由 25 个人做动作，每人做 4-7 组。
- 在摄像机运动、物体外观和姿态、物体尺度、视点、杂乱背景、光照条件等方面存在较大的差异

官网：  
<https://www.crcv.ucf.edu/data/UCF101.php>

论文：  
<https://arxiv.org/pdf/1212.0402.pdf>

### (2) HMDB51

HMDB51 包括 51 类、6766 剪辑视频、每个视频不超过 10 秒、分辨率 320\*240、共 2 GB。视频源于 YouTube 和谷歌视频，内容包括人面部、肢体、和物体交互的动作这几大类。

论文：

[https://serre-lab.clps.brown.edu/wp-content/uploads/2012/08/Kuehne\\_etal\\_iccv11.pdf](https://serre-lab.clps.brown.edu/wp-content/uploads/2012/08/Kuehne_etal_iccv11.pdf)

官网：

<https://serre-lab.clps.brown.edu/resource/hmdb-a-large-human-motion-database/>

### (3) Sports1M

487 类、1,100,000 视频（70%训练、20%验证、10%测试）。内容包含各种体育运动。

论文：Large-scale video classification with convolutional neural networks

### (4) Kinetics

Kinetics 是谷歌 DeepMind 推出的视频数据集，先后推出了 Kinetics-400、Kinetics-600 和 Kinetics-700 三个版本，数字表示视频类别的数量。视频源于 YouTube，Kinetics 官网上的 csv 文件描述了每个视频的类别、在 Youtube 的 ID、事件开始时间、事件结束时间等。

可以用下面的官方代码通过 csv 下载视频：

<https://github.com/activitynet/ActivityNet/tree/master/Crawler/Kinetics>

也有非官方的下载器：

<https://github.com>Showmax/kinetics-downloader>

Kinetics-400 目前包括了 400 类、22 万多训练视频、1.8 万验证视频、3.5 万测试视频。

Kinetics-600 目前包括了 600 类、37 万多训练视频、2.8 万验证视频、5.7 万测试视频。

Kinetics-700 目前包括了 700 类、53 万多训练视频、3.4 万验证视频、6.8 万测试视频。

Kinetics 是一个大规模数据集，其在视频理解中的作用有些类似于 ImageNet 在图像识别中的作用，有些工作用 Kinetics 预训练模型迁移到其他视频数据集。

官网：

<https://deepmind.com/research/open-source/kinetics>

论文：

Kinetics-400：<https://arxiv.org/pdf/1705.06950.pdf>

Kinetics-600：<https://arxiv.org/pdf/1808.01340.pdf>

Kinetics-700：<https://arxiv.org/pdf/1907.06987.pdf>

## (5) Something-something V2

Something-something 是 20BN 推出的视频数据集

官网：

<https://20bn.com/datasets/something-something/v2>

## 2 · 衡量标准

由于跟 Image Classification 的相似性，衡量标准同图像分类。

## 3 · 传统方法

传统方法中最出色的是 iDT，迄今为止 iDT 和神经网络方法的差距都不是很大。

### (1) iDT

## 4 · Two-Stream

### (1) Two-stream (201406)

two-stream 这篇论文的主要贡献有 3 个：

- 1、首先，提出了一种融合时空网络的双流 convnet 体系结构。
- 2、第二，证明了在多帧密集光流上训练的 convnet 的方法，即使在训练数据有限的情况下仍能获得很好的性能。
- 3、最后，将多任务学习应用于两种不同的动作分类数据集，可以增加训练数据量，提高训练数据的性能。

two-stream 的概念是这篇文章最大的贡献，在之后的网络中屡次被用到，这里的两个 stream 指的是两个独立的 CNN，分别用于处理空间数据（Spatial Stream ConvNet）和时间数据（Temporal Stream ConvNet）。

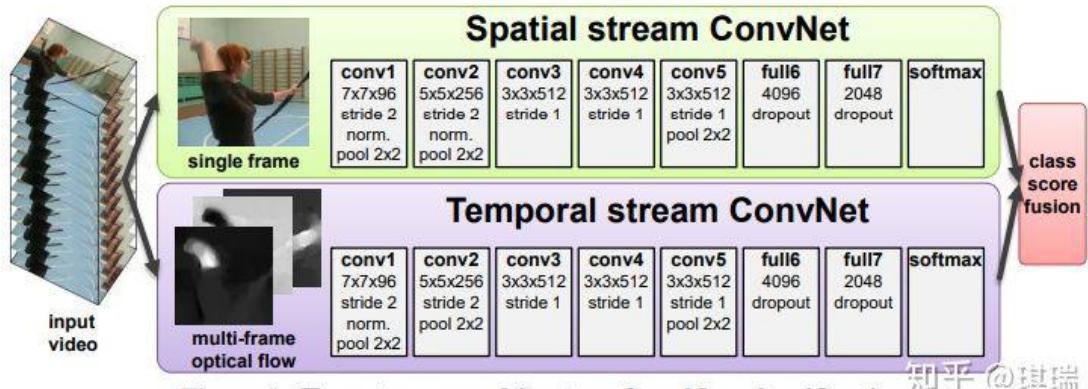


Figure 1: Two-stream architecture for video classification.

知乎@琪瑞

上层的 Spatio Stream Convnet 处理静态的帧画面，输入是单个帧画面，得到画面中物体的特征，还可以使用 ImageNet challenge 数据集种有标注的图像数据进行预训练。temporal stream convnet 则是通过多帧画面的光流位移来获取画面中物体的运动信息，其输入由多个连续帧之间的光流位移场叠加而成，这种输入显式地描述了视频帧之间的运动，这使得识别更加容易，因为网络不需要隐式地估计运动。两个不同的 stream 都通过 CNN 实现，最后进行信息融合。

optical flow 是由一些位移矢量场 (displacement vector fields) (每个矢量用  $dt$  表示) 组成的，其中  $dt$  是一个向量，表示第  $t$  帧的 displacement vector，是通过第  $t$  和第  $t+1$  帧图像得到的。 $dt$  包含水平部分  $dtx$  和竖直部分  $dty$ ，可以看下图中的 (d) 和 (e)。因此如果一个 video 有  $L$  帧，那么一共可以得到  $2L$  个 channel 的 optical flow，然后才能作为 Figure1 中 temporal stream convnet 网络的输入。

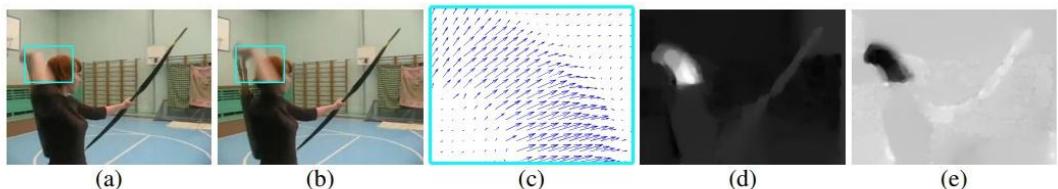


Figure 2: Optical flow. (a), (b): a pair of consecutive video frames with the area around a moving hand outlined with a cyan rectangle. (c): a close-up of dense optical flow in the outlined area; (d): horizontal component  $d^x$  of the displacement vector field (higher intensity corresponds to positive values, lower intensity to negative values). (e): vertical component  $d^y$ . Note how (d) and (e) highlight the moving hand and bow. The input to a ConvNet contains multiple flows (Sect. 3.1).

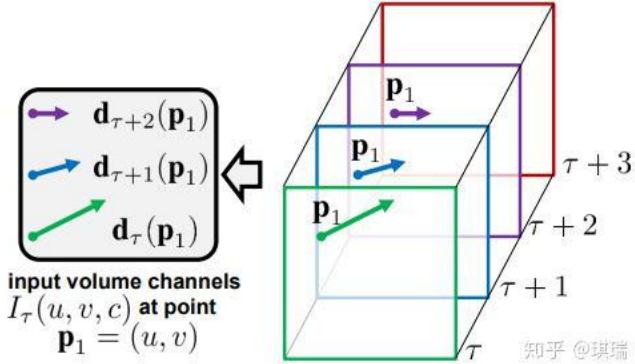
上图中的 (a) 和 (b) 表示连续的两帧图像，(c) 表示一个 optical flow，(d) 和 (e) 分别表示一个 displacement vector field 的水平和竖直两部分。

所以如果假设一个 video 的宽和高分别是  $w$  和  $h$ ， $L$  表示连续的帧数，那么 Figure1 中 temporal stream convnet 的输入维度应该是下面这样的。其中  $\tau$  表示任意的一帧。

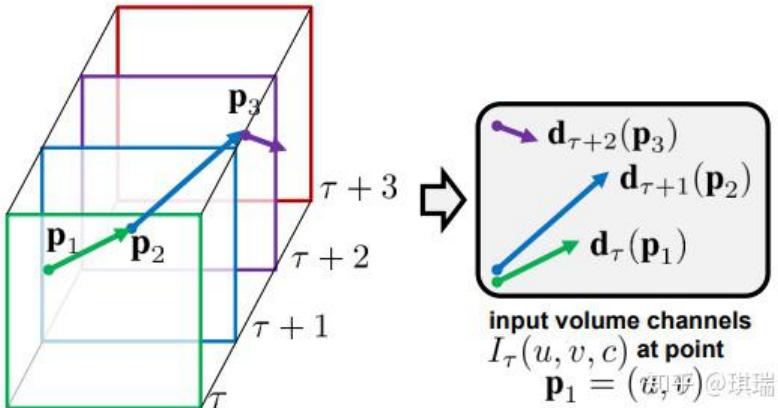
$$I_\tau \in \mathbb{R}^{w \times h \times 2L}$$

光流数据作为 Temporal ConvNet 的输入，有以下几种形式：

1. Optical flow stacking: 输入的大小为  $w \times h \times 2L$ , L 表示堆叠的帧数。可以描述输入 map 某个位置的光流变化。



2. 轨迹叠加(Trajectory stacking): 在基于轨迹的描述符的启发下, 另一种运动表示法将在多个帧的相同位置采样的光流替换为沿运动轨迹采样的流。



3. Bi-directional optical flow: 输入大小为  $w \times h \times 2L$ , 由于是双向的,  $L/2$  表示堆叠的帧数。

4. Mean flow subtraction: 减掉平均光流, 用来去除部分相机运动对光流的影响。

论文的第三个贡献, 就是用“multi-task learning”来克服数据量不足的问题。其实就是 CNN 的最后一层连到多个 softmax 的层上, 对应不同的数据集, 这样就可以在多个数据集上进行 multi-task learning。spatial stream convnet 因为输入是静态的图像, 因此其预训练模型容易得到(一般采用在 ImageNet 数据集上的预训练模型), 但是 temporal stream convnet 的预训练模型就需要在视频数据集上训练得到, 但是目前能用的视频数据集规模还比较小(主要指的是 UCF-101 和 HMDB-51 这两个数据集, 训练集数量分别是 9.5K 和 3.7K 个 video)。因此作者采用 multi-task 的方式来解决。怎么做呢? 首先原来的网络 (temporal stream convnet) 在全连接层后只有一个 softmax 层, 现在要变成两个 softmax 层, 一个用来计算 HMDB-51 数据集的分类输出, 另一个用来计算 UCF-101 数据集的分类输出, 这就是两个 task。这两条支路有各自的 loss, 最后回传 loss 的时候采用的是两条支路 loss 的和。

论文：

<https://arxiv.org/pdf/1406.2199v2.pdf>

github：

<https://github.com/woodfrog/ActionRecognition>

参考：

<https://zhuanlan.zhihu.com/p/61605147>

<https://zhuanlan.zhihu.com/p/34929782>

## (2) Two-stream Fusion (201604)

更好的双流融合

论文：

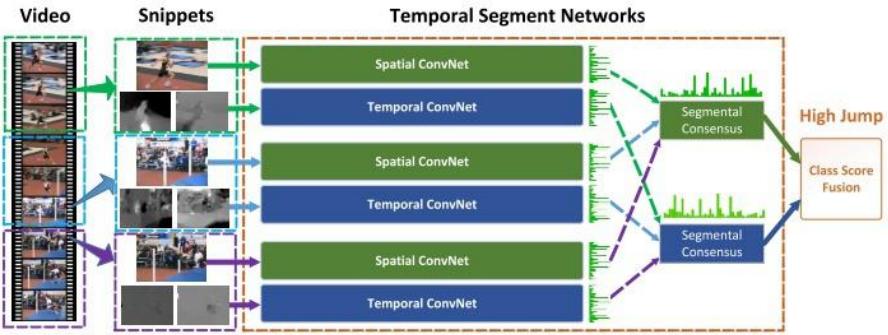
Convolutional Two-Stream Network Fusion for Video Action Recognition

<https://arxiv.org/pdf/1604.06573.pdf>

## (3) TSN (201608)

由于信息的来源是单帧图像和光流图，two-stream 的缺点是对长视频的学习能力不足，已经有的解决方法是 dense temporal sampling，但缺点是计算量太大。此外 CNN 训练需要大量的数据，而当时的数据量不够大，容易过拟合。

TSN (Temporal Segment Network) 也是建立在 two-stream 的基础上的，在 TSN 中，作者首先将数据等时间的划分成 K 个 segments (原文 K=3)，然后在每一个 segment 中随机的采样出一个 snippet，这样就生成了 K 个 snippets。每一个 snippet 都作为网络的 input 就能得到 K 个关于视频所属类别的 scores，然后这 K 个 scores 通过一个融合函数 G 来生成最终评分。G 主要有 max, average 和 weighted average 这三种，作者通过对比实验最终选择 average。



**Fig. 1.** Temporal segment network: One input video is divided into  $K$  segments and a short snippet is randomly selected from each segment. The class scores of different snippets are fused by an the segmental consensus function to yield segmental consensus, which is a video-level prediction. Predictions from all modalities are then fused to produce the final prediction. ConvNets on all snippets share parameters. @Ooops

也就是说，给定一段视频  $V$ ，把它按相等间隔分为  $K$  段  $\{S_1, S_2, \dots, S_K\}$ 。接着，TSN 按如下方式对一系列片段进行建模：

$$TSN(T_1, T_2, \dots, T_K) = H(G(F(T_1; W), F(T_2; W), \dots, F(T_K; W)))$$

- $(T_1, T_2, \dots, T_K)$  代表片段序列，每个片段  $T_k$  从它对应的段  $S_k$  中随机采样得到。
- $F(T_k; W)$  函数代表采用  $W$  作为参数的卷积网络作用于短片段  $T_k$ ，函数返回  $T_k$  相对于所有类别的得分。
- 段共识函数  $G$  (The segmental consensus function) 结合多个短片段的类别得分输出以获得他们之间关于类别假设的共识。（本文  $G$  为取平均值）
- 基于这个共识，预测函数  $H$  预测整段视频属于每个行为类别的概率（本文  $H$  选择了 Softmax 函数）。
- 结合标准分类交叉熵损失 (cross-entropy loss)，关于部分共识的最终损失函数的形式为：

$$\mathcal{L}(y, G) = - \sum_{i=1}^C y_i \left( G_i - \log \sum_{j=1}^C \exp G_j \right)$$

其中  $C$  表示类别数， $y_i$  是标签。

网络内部采用 BN-Inception。

此外，作者还提出了 **RGB diff** 与 **Warped Optical Flow** 是这两种 modality，来作为 RGB 和 optical flow 的补充。最后作者发现  $RGB + optical flow + warped optical flow$  的组合是最好的，引入  $RGB$  diff 反而会带来较差的结果。

- **RGB diff** :  $RGB$  差值，连续的两个 frames 相减，能凸显出运动的部分，补充运动信息
- **Warped Optical Flow** : 扭曲光流，由于现实拍摄的视频中，通常存在摄像机的运动，这样光流场就不是单纯体现出人类行为。由于相机的移动，视频背景中存在大量的水平运动。受到 iDT (improved dense trajectories) 工作的启发，作者提出将扭曲的光流场作为额外的输入。通过估计单应性矩阵 (homography matrix) 和补偿相机运动来提取扭曲光流场。图 2 中，扭曲光流场抑制了背景运动，使得专注于视频中的人物运动

作者还使用了多种减轻过拟合的技术。用 RGB 模型初始化 optical flow 模型、freeze 除了第一层以外的其余 BN 层的均值和方差、采用多种数据增强技术。

论文：

Temporal Segment Network: Towards Good Practices for Deep Action Recognition

<https://arxiv.org/pdf/1608.00859.pdf>

github：

<https://github.com/yjxiong/temporal-segment-networks>

参考：

<https://zhuanlan.zhihu.com/p/32777430>

<https://zhuanlan.zhihu.com/p/62121305>

<https://blog.csdn.net/chen1234520nnn/article/details/104901072>

## (4) TRN (201711)

论文：

<https://arxiv.org/pdf/1711.08496.pdf>

参考：

代码：

## 5 · 3D Conv

### (1) C3D (201412)

3D 卷积是在图像的二维卷积（长、宽）上加一维时间方向的维度。

C3D 中 C 指的是 Convolution，C3D 网络中的视频片段尺寸定义为  $c * 1 * h * w$  ( $c$  为通道、 $1$  为帧数、 $h$  和  $w$  为高和宽)，下图为 C3D 的结构：

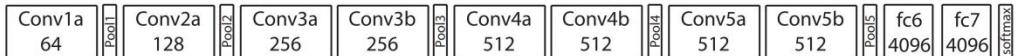


Figure 3. **C3D architecture.** C3D net has 8 convolution, 5 max-pooling, and 2 fully connected layers, followed by a softmax output layer. All 3D convolution kernels are  $3 \times 3 \times 3$  with stride 1 in both spatial and temporal dimensions. Number of filters are denoted in each box. The 3D pooling layers are denoted from pool1 to pool5. All pooling kernels are  $2 \times 2 \times 2$ , except for pool1 is  $1 \times 2 \times 2$ . Each fully connected layer has 4096 output units.

C3D 网络有 8 个卷积层，5 个最大池化层和 2 个全连接层，最后是 softmax 输出层。所有的 3D 卷积核都是  $3 \times 3 \times 3$ ，在空间和时间上都有步长 1。卷积核的数量表示在每个框中。3D 池化层由 pool1 到 pool5 表示。所有池化核为  $2 \times 2 \times 2$ ，除了 pool1 为  $1 \times 2 \times 2$ 。每个全连接层有 4096 个输出单元。

C3D 的一个贡献就是发现  $3 \times 3 \times 3$  的卷积核效果最好。

论文：

<https://arxiv.org/pdf/1412.0767.pdf>

参考：

<https://www.jianshu.com/p/0b4964261673>

<https://www.jianshu.com/p/09d1d8ffe8a4>

## (2) I3D (201705)

I3D 中的 I 表示 Inflated，是一个双流网络和 3D 卷积的融合。

论文提到，当前 3D 网络的主要问题是：

- 相对于 2D 网络，参数量有较大增加，更难训练
- 因为是全新的网络，无法利用已经成熟的预训练 2D 网络，只能用层数较少的 CNN 在小数据集上从头训练

I3D 把 two-stream 结构中的 2D 卷积扩展为 3D 卷积。并且利用了预训练的模型

Inception，其方法是将  $N \times N$  的 Inception module 复制  $N$  遍（即 Inflate，膨胀），变为  $N \times N \times N$  的 module。

由于时间维度不能缩减过快，前两个汇合层的卷积核大小是  $1 \times 2 \times 2$ ，最后的汇合层的卷积核大小是  $2 \times 7 \times 7$ 。和之前文章不同的是，two-stream 的两个分支是单独训练的，测试时融合它们的预测结果。

论文：

<https://arxiv.org/pdf/1705.07750v3.pdf>

参考：

<https://zhuanlan.zhihu.com/p/34919655>

github :

<https://github.com/deepmind/kinetics-i3d>

### (3) T3D (201711)

T3D 中的 T 表示 Temporal，来自于论文中的 TTL 层（Temporal Transition Layer）。

T3D 的三维卷积主要是在 densenet 的基础上改进得到，具体来说就是将原始网络中二维卷积修改为三维卷积，二维 pooling 修改为三维 pooling。

T3D 最主要的创新来源于 TTL 层（Temporal Transition Layer），TTL 层包含几个不同大小和时间域深度的卷积 kernel 和三维 pooling 层构成，所希望达到的效果是能够对短、中、长三个不同的时间长度的序列信息进行建模。

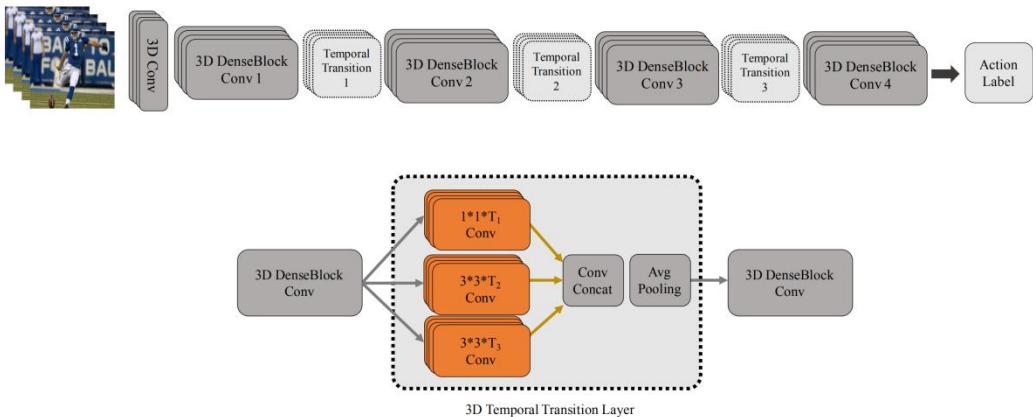


Figure 1: **Temporal 3D ConvNet (T3D).** Our Temporal Transition Layer (TTL) is applied to our DenseNet3D. T3D uses video clips as input. The 3D feature-maps from the clips are densely propagated throughout the network. The TTL operates on the different temporal depths, thus allowing the model to capture the appearance and temporal information from the short, mid, and long-range terms. The output of the network is a video-level prediction.

论文：

<https://arxiv.org/pdf/1711.08200.pdf>

参考：

<https://zhuanlan.zhihu.com/p/90374193>

github :

<https://github.com/MohsenFayyaz89/T3D>

## (4) P3D (201711)

P3D 中的 P 表示 Pseudo，P3D 用一个  $1 \times 3 \times 3$  的空间方向卷积和一个  $3 \times 1 \times 1$  的时间方向卷积来近似原  $3 \times 3 \times 3$  卷积。这种技巧用于减少运算量，类似于 2D 卷积中的深度可分离卷积 (Depthwise Separable Conv) 和 SS 卷积 (Spatial Separable Conv)。

通过组合三种不同的模块结构，进而得到 P3D ResNet。P3D ResNet 在参数数量、运行速度等方面对 C3D 作出了优化。

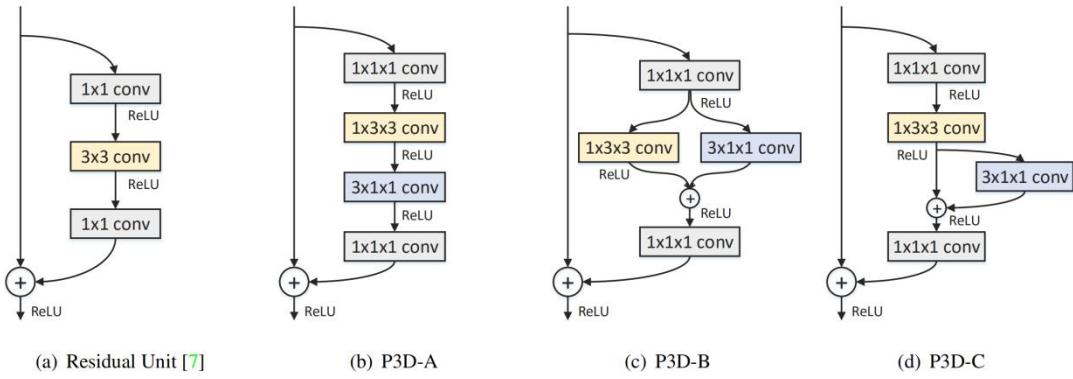


Figure 3. Bottleneck building blocks of Residual Unit and our Pseudo-3D.

论文：

<https://arxiv.org/pdf/1711.10305.pdf>

## (5) R(2+1)D (201711)

R(2+1)D 中的 R 表示 Resnet。

R(2+1)D 的创新包括 2 个：

- 新的混合型结构：在浅层使用 3 维卷积，在深层接上 2 维卷积。
- 新的微结构：2+1 维卷积块，就是把 3 维卷积操作分解成两个接连进行的子卷积块，2 维空间卷积和 1 维时间卷积（类似 P3D-A）。

论文：

<https://arxiv.org/pdf/1711.11248.pdf>

参考：

<https://zhuanlan.zhihu.com/p/48267318>

## (6) ECO (201804)

论文：

<https://arxiv.org/pdf/1804.09066.pdf>

## (7) S1owFast (201812)

S1owFast 创造了一种新的时空交互的方法：

1. 两路 3d 卷积，分别侧重时间 (fast) 和空间 (s1ow)
2. 将侧重时间的支路信息融合入空间支路

两个支路 s1ow 和 fast 指的是 T 维的卷积核大小核跨步不同。

- Fast 路的时间维度卷积核大小为  $aT$ , 时间维度 stride 为  $s/a$ , 通道数为  $c/a$ ，比较轻量；
- S1ow 支路时间维度卷积核大小为  $T$ ，时间维度 stride 为  $s$ , 通道数为  $c \cdot a=8$ . 配置如下：

stage	<i>Slow</i> pathway	<i>Fast</i> pathway	output sizes $T \times S^2$
raw clip	-	-	$64 \times 224^2$
data layer	stride 16, $1^2$	stride $\underline{2}$ , $1^2$	<i>Slow</i> : $4 \times 224^2$ <i>Fast</i> : $\underline{32} \times 224^2$
conv <sub>1</sub>	$1 \times 7^2$ , 64 stride 1, $2^2$	$\underline{5} \times 7^2$ , $\underline{8}$ stride 1, $2^2$	<i>Slow</i> : $4 \times 112^2$ <i>Fast</i> : $\underline{32} \times 112^2$
pool <sub>1</sub>	$1 \times 3^2$ max stride 1, $2^2$	$1 \times 3^2$ max stride 1, $2^2$	<i>Slow</i> : $4 \times 56^2$ <i>Fast</i> : $\underline{32} \times 56^2$
res <sub>2</sub>	$\left[ \begin{array}{l} 1 \times 1^2, 64 \\ 1 \times 3^2, 64 \\ 1 \times 1^2, 256 \end{array} \right] \times 3$	$\left[ \begin{array}{l} \underline{3} \times 1^2, \underline{8} \\ \underline{1} \times 3^2, \underline{8} \\ 1 \times 1^2, \underline{32} \end{array} \right] \times 3$	<i>Slow</i> : $4 \times 56^2$ <i>Fast</i> : $\underline{32} \times 56^2$
res <sub>3</sub>	$\left[ \begin{array}{l} 1 \times 1^2, 128 \\ 1 \times 3^2, 128 \\ 1 \times 1^2, 512 \end{array} \right] \times 4$	$\left[ \begin{array}{l} \underline{3} \times 1^2, \underline{16} \\ \underline{1} \times 3^2, \underline{16} \\ 1 \times 1^2, \underline{64} \end{array} \right] \times 4$	<i>Slow</i> : $4 \times 28^2$ <i>Fast</i> : $\underline{32} \times 28^2$
res <sub>4</sub>	$\left[ \begin{array}{l} \underline{3} \times 1^2, 256 \\ \underline{1} \times 3^2, 256 \\ 1 \times 1^2, 1024 \end{array} \right] \times 6$	$\left[ \begin{array}{l} \underline{3} \times 1^2, \underline{32} \\ \underline{1} \times 3^2, \underline{32} \\ 1 \times 1^2, \underline{128} \end{array} \right] \times 6$	<i>Slow</i> : $4 \times 14^2$ <i>Fast</i> : $\underline{32} \times 14^2$
res <sub>5</sub>	$\left[ \begin{array}{l} \underline{3} \times 1^2, 512 \\ \underline{1} \times 3^2, 512 \\ 1 \times 1^2, 2048 \end{array} \right] \times 3$	$\left[ \begin{array}{l} \underline{3} \times 1^2, \underline{64} \\ \underline{1} \times 3^2, \underline{64} \\ 1 \times 1^2, \underline{256} \end{array} \right] \times 3$	<i>Slow</i> : $4 \times 7^2$ <i>Fast</i> : $\underline{32} \times 7^2$
	global average pool, concat, fc		# classes

Table 1. **An example instantiation of the SlowFast network.** The dimensions of kernels are denoted by  $\{T \times S^2, C\}$  for temporal, spatial, and channel sizes. Strides are denoted as  $\{\text{temporal stride}, \text{spatial stride}^2\}$ . Here the speed ratio is  $\alpha = 8$  and the channel ratio is  $\beta = 1/8$ .  $\tau$  is 16. The green colors mark *higher* temporal resolution, and orange colors mark *fewer* channels, for the Fast pathway. Non-degenerate temporal filters are underlined. Residual blocks are shown by brackets. The backbone is ResNet-50 ©孟让

网络结构就是以上两种 3d 卷积的支路，中间将 slow 支路融合到 fast，有三种方法：

1. time2channel1 : reshape fast 维度的  $[bC, aT, S, S]$  to  $[abC, T, S, S]$
2. 对 fast 路的特征图进行 temporal stride sample 统一时间维度
3. fast 接跨步 3d 卷积统一时间维度

论文：

<https://arxiv.org/pdf/1812.03982v3.pdf>

参考：

<https://zhuanlan.zhihu.com/p/103577209>

代码：

<https://github.com/facebookresearch/SlowFast>

## 6 · Skeleton-based

Skeleton-based 类型的神经网络需要先对视频做**人体关键点检测**（Keypoint Detection，或称**人体姿态评估** Pose Estimation），然后将得到的人体关键点作为输入给到神经网络，得出视频分类。这类神经网络的论文不多，或许与数据集太少有关。

### (1) ST-GCN (201801)

论文：

<https://arxiv.org/pdf/1801.07455.pdf>

参考：

<https://zhuanlan.zhihu.com/p/108255643>

代码：

<https://github.com/yysijie/st-gcn>

### (2) TSM (201811)

论文：

<https://arxiv.org/pdf/1811.08383.pdf>

github：

<https://github.com/MIT-HAN-LAB/temporal-shift-module>

### (3) Multigrid (201912)

论文：

<https://arxiv.org/pdf/1912.00998v2.pdf>

github：（其中使用了 multigrid 方法）  
<https://github.com/facebookresearch/SlowFast>

## 7 · LSTM-based

### (1) LRCN (201411)

论文：  
<https://arxiv.org/pdf/1411.4389.pdf>

参考：  
<https://zhuanlan.zhihu.com/p/90378536>

## (二) 时序行为识别 (Temporal Action Recognition)

时序行为识别，Temporal Action Recognition 或者时域动作检测，Temporal Action Detection，Temporal Action Localization。

时序行为识别的目的是给定一段视频，算法需要检测视频中的行为片段，包括其开始时间、结束时间以及类别，一段视频中可能包括一个或者若干个片段。

Video Action Recognition 和 Temporal Action Recognition 之间的关系就像 Image Classification 和 Object Detection。Temporal Action Detection 不仅要识别动作的类别，还需要知道动作的起始帧和结束帧。

参考：  
<https://zhuanlan.zhihu.com/p/26603387>  
<https://zhuanlan.zhihu.com/p/52524590>  
<https://www.jianshu.com/p/92bb5cad2319>

## 1 · 数据集

### (1) THUMOS14

该数据集即为 THUMOS Challenge 2014。该数据集包括行为识别和时序行为检测两个任务。它的训练集为 UCF101 数据集，包括 101 类动作，共计 13320 段分割好的视频片段。THUMOS2014 的验证集和测试集则分别包括 1010 和 1574 个未分割过的视频。在时序行为检测任务中，只有 20 类动作的未分割视频是有时序行为片段标注的，包括 200 个验证集视频（包含 3007 个行为片段）和 213 个测试集视频（包含 3358 个行为片段）。这些经过标注的未分割视频可以被用于训练和测试时序行为检测模型。实际上之后还有 THUMOS Challenge 2015，包括更多的动作类别和视频数，但由于上面可以比较的方法不是很多，所以目前看到的文章基本上还是在 THUMOS14 上进行实验。

官网：

<http://crcv.ucf.edu/THUMOS14/>

### (2) ActivityNet

目前最大的数据库，同样包含分类和检测两个任务，这个数据集仅提供视频的 youtube 链接，而不能直接下载视频，所以还需要用 python 中的 youtube 下载工具来自动下载。该数据集包含 200 个动作类别，20000（训练+验证+测试集）左右的视频，视频时长共计约 700 小时。

## 2 · 衡量标准

由于跟 Object Detection 的相似性，时序行为识别的衡量标准也是 IoU 和 mAP。

2017 的大多数工作在  $\text{IOU}=0.5$  的情况下达到了 20%-30% 的 MAP，虽然较 2016 年提升了 10% 左右，但是在  $\text{IOU}=0.7$  时直接降低到了 10% 以下，2018 年  $\text{IOU}=0.5$  有 34% 的 mAP。

## 3 · SSN

### (三) 时空行为识别

时空行为识别（Spatio-Temporal Detection）比时序行为识别更进一步，不仅需要在视频中检测出每个动作的起始帧和结束帧，还需要检测出这个动作在每一帧中的位置。

#### 1 · 数据集

##### (1) AVA

AVA (Atomic Vision Action) 是谷歌推出的数据集，类似 Kinetics，其具体视频也是在Youtube 上，官网上下载的只是 Annotation。

官网：

<https://research.google.com/ava/>

### (四) 视频字幕 (Video Captioning)

#### 1 · 数据集

YouCook2

### (五) 视频问答 (Video QA)

如果行为识别是视频理解的入门，那么视频问答就是视频理解的终极难度。任务是输入一段视频和一个文本问题，得到一个答案。

举个例子，输入一段几个小孩拍球的视频，问：蓝色衣服的小朋友拍了几次球？

## 1 · Zhou (201804)

论文：

<https://arxiv.org/pdf/1804.00819v1.pdf>

github：

<https://github.com/salesforce/densecap>

## 2 · Video-BERT (201904)

论文：

<https://arxiv.org/pdf/1904.01766.pdf>

## (六) 视频目标跟踪 (Video Object Tracking)

广义的 VOT (Video Object Tracking) 任务分为单目标跟踪 (SOT, Single Object Tracking) 和多目标跟踪 (MOT, Multiple Object Tracking)，而狭义的 VOT 指的是单目标跟踪 (SOT)。

目标跟踪的意思，就是在一段视频上对移动物体做跨时间的跟踪。那么为什么不直接针对每帧图像做 Object Detection 呢？因为用目标检测的方法会有一些问题：

- 如果图像中有若干个物体，那如何确定上下两帧之间物体的对应关系？
- 如果跟踪的物体离开画面若干帧（比如被柱子挡住），之后另一个物体出现在画面中，如何确定这两个物体是否是同一个？

或者说，如果仅仅针对每帧做图像处理，我们无法得知物体的移动情况。

此外还会有一些目标检测中多余的计算，也许在 VOT 中并不需要：

- 直接对视频中每帧图片做目标检测，其计算量代价是比较大的，可以通过一些算法节省算力，比如根据物体在  $t$  时刻和  $t+1$  时刻的位置，预测其在  $t+2$  时刻的位置
- 基本的 VOT 算法是不需要做物品分类的，分类只是个可选项而已。

再者就是考虑到视频前后帧之间的联系，可以

- 有效的去除错误的目标检测结果
- 补上遗漏的目标检测结果

目前整个目标跟踪领域的发展都比较停滞，依然有很多都是非深度学习的方法，远落后于目标检测（表现为有些目标检测的算法基础上修改为目标跟踪算法，效果都好于传统目标跟踪算法）。深度学习在目标跟踪领域尚未和相关滤波类的算法拉开什么差距。

论文：

Deep Learning for Visual Tracking: A Comprehensive Survey

<https://arxiv.org/pdf/1912.00535.pdf>

A Review of Visual Trackers and Analysis of its Application to Mobile Robot

<https://arxiv.org/pdf/1910.09761.pdf>

参考

<https://cv-tricks.com/object-tracking/quick-guide-mdnet-goturn-rolo/>

<https://www.zhihu.com/question/26493945/answer/156025576>

<https://zhuanlan.zhihu.com/p/44265232>

## 1 · 衡量标准

参考：

<https://www.jianshu.com/p/8cd0bcc9792c>

### (1) Precision Slot / Success Plot

- **Precision Plot**：预测位置中心点与标注的中心位置间的欧式距离，以像素为单位。  
结果用 average precision plot 来表示，即为该视频序列所有帧的平均误差。
- **Success plot**：主要指的是预测目标所在 benchmark 的重合程度，即 IOU (交并比)

### (2) Accuracy / Robustness / EAO

- **Accuracy**：准确率，是指跟踪器在单个测试序列下的平均重叠率（两矩形框的相交部分面积除以两矩形框的相并部分的面积。即 average success plot）。
- **Robustness**：鲁棒性，是指单个测试序列下的跟踪器失败次数，当重叠率为 0 时即可判定为失败。
- **EAO (Expected Average Overlap)**：平均重叠期望，对每个跟踪器在一个短时图像序列上的非重置重叠的期望值，即在一个 N 帧视频上每帧的 IOU 的均值。

$$\phi_{N_s} = \frac{1}{N_s} \sum_{i=1}^N \phi_i$$

这三者也是 VOT2017 所使用的评价标准。

## 2 · 数据集

### (1) VOT

**Visual-Object-Tracking Challenge (VOT)** 是当前国际上在线目标跟踪领域最权威的测评平台，由伯明翰大学、卢布尔雅那大学、布拉格捷克技术大学、奥地利科技学院联合创办，旨在评测在复杂场景下单目标跟踪的算法性能。

官网：

<http://www.votchallenge.net/>

VOT Challenges

<https://votchallenge.net/challenges.html>

### (2) OTB

分为 OTB50 (OTB-2013) 和 OTB100 (OTB-2015) ，

## (七) 多目标跟踪 (Multiple Object Tracking)

广义的 VOT 任务分为单目标跟踪 (SOT, Single Object Tracking) 和多目标跟踪 (MOT, Multiple Object Tracking) ，而通常 VOT 指的是单目标跟踪。

MOT 通常的工作流程如下：

1. 给定视频的原始帧
2. 运行对象检测器以获得对象的边界框 (目标检测)
3. 对于每个检测到的物体，计算出不同的特征，通常是视觉和运动特征；
4. 之后，相似度计算步骤计算两个对象属于同一目标的概率；
5. 最后，关联步骤为每个对象分配数字 ID。

MOT 和 SOT 之间的区别还是比较大的。

按照初始化方法，MOT 可以分为 Detection-Based Tracking (DBT) 和 Detection-Free Tracking (DFT) 两种：

- DBT：首先检测目标，然后链接到轨迹中，给定一个序列，在每帧中对特定类型的目标检测，然后进行跟踪。可自动发现新目标，自动终止消失的目标。现行的 MOT 通常指的是这种方式
- DFT：需要在第一帧手动初始化一定数量的目标，然后在后续帧定位这些物体。

按照处理模式分，分为 Online 追踪和 Offline 追踪：

- Online 用的是当前帧及之前的信息
- Offline 可以用到未来帧的信息

参考：

<https://zhuanlan.zhihu.com/p/97449724>  
<https://blog.csdn.net/yuhq3/article/details/78742658>  
[https://zhuanlan.zhihu.com/c\\_1102212337087401984](https://zhuanlan.zhihu.com/c_1102212337087401984)

论文：

Multiple Object Tracking: A Literature Review  
<https://arxiv.org/pdf/1409.7618.pdf>

Deep Learning in Video Multi-Object Tracking: A Survey  
<https://arxiv.org/pdf/1907.12740.pdf>

## 1 · 衡量标准

参考：

<https://zhuanlan.zhihu.com/p/75776828>

### (1) MOTA/MOTP

MOTA (MOT Accuracy) 和 MOTP (MOT Precision) 都是 MOT Challenge 数据集提出的 MOT 衡量标准 Clear MOT 的一部分。

**MOTA**：多目标跟踪的准确度，体现在确定目标的个数，以及有关目标的相关属性方面的准确度，用于统计在跟踪中的误差积累情况，包括 FP、FN、ID Sw。

$$MOTA = 1 - \frac{\sum_t (m_t + fp_t + mme_t)}{\sum_t g_t}$$

**m<sub>t</sub>**：是 FP, 缺失数（漏检数），即在第 t 帧中该目标，没有假设位置与其匹配。

**fp<sub>t</sub>**：是 FN，误判数，即在第 t 帧中给出的假设位置，没有跟踪目标与其匹配。

**mme<sub>t</sub>**：是 ID Sw，误配数，即在第 t 帧中跟踪目标发生 ID 切换的次数，多发生在这档情况下。

**MOTP**：多目标跟踪的精确度，体现在确定目标位置上的精确度，用于衡量目标位置确定的精确程度。

$$MOTP = \frac{\sum_{i,t} d_t^i}{\sum_t c_t}$$

c<sub>t</sub>：表示第 t 帧目标 o<sub>i</sub> 和假设 h<sub>j</sub> 的匹配个数

d<sup>i</sup><sub>t</sub>：表示第 t 帧目标 o<sub>i</sub> 与其配对假设位置之间的距离，即匹配误差。

参考：

<https://zhuanlan.zhihu.com/p/75776828>

## 2 · 数据集

### (1) MOT Challenge

MOT Challenge 是当前用的最多的多目标追踪数据集

官网

<https://motchallenge.net/>

参考：

<https://zhuanlan.zhihu.com/p/133670271>

### (2) KITTI

用的较少

### 3 · 传统算法

当前的 MOT (及 VOT) 尚未有端到端的神经网络的实现 (其原因应该是计算量过大、数据集不够)，通常都要加上一些其他的算法。

#### (1) 卡尔曼滤波 (Kalman Filter)

卡尔曼滤波 (Kalman filter) 是一种高效率的递归滤波器 (自回归滤波器)，它能够从一系列的不完全及包含噪声的测量中，估计动态系统的状态。卡尔曼滤波会根据各测量量在不同时间下的值，考虑各时间下的联合分布，再产生对未知变数的估计，因此会比只以单一测量量为基础的估计方式要准。

简单来说，卡尔曼滤波解决的是如何从多个不确定数据中提取相对精确的数据。前提是：

- 实践前提是这些数据满足高斯分布。
- 理论前提是一个高斯斑乘以另一个高斯斑可以得到第三个高斯斑，第三个高斯斑即为提取到相对精确的数据范围。

参考：

<https://www.zhihu.com/question/23971601/answer/375355599>

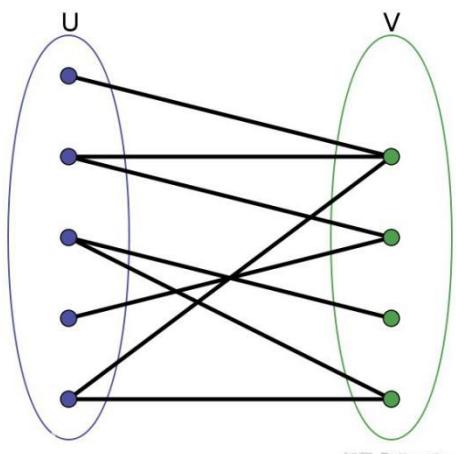
<https://www.zhihu.com/question/23971601/answer/194464093>

<https://zh.wikipedia.org/wiki/%E5%8D%A1%E5%B0%94%E6%9B%BC%E6%BB%A4%E6%B3%A2>

#### (2) 匈牙利算法 & KM 算法

匈牙利算法 (Hungarian Algorithm) 和 KM 算法 (Kuhn—Munkres 算法，Munkres 分配算法) 是都是求解二分图的最大匹配问题的组合优化算法。

二分图就是能分成两组，U,V。其中，U 上的点不能相互连通，只能连去 V 中的点，同理，V 中的点不能相互连通，只能连去 U 中的点。这样，就叫做二分图。



可以把二分图理解为视频中连续两帧中的所有检测框，第一帧所有检测框的集合称为  $U$ ，第二帧所有检测框的集合称为  $V$ 。同一帧的不同检测框不会为同一个目标，所以不需要互相关联，相邻两帧的检测框需要相互通连，最终将相邻两帧的检测框尽量完美地两两匹配起来。而求解这个问题的最优解就要用到**匈牙利算法**或者**KM 算法**（KM 算法解决的是带权二分图的最优匹配问题）。

参考：

<https://zhuanlan.zhihu.com/p/62981901>

<https://zh.wikipedia.org/wiki/%E5%8C%88%E7%89%99%E5%88%A9%E7%AE%97%E6%B3%95>

## 4 · SORT (201602)

现在多目标跟踪算法的效果，与目标检测的结果息息相关，因为主流的多目标跟踪算法都是TBD (Tracking-by-Detecton) 策略。

SORT 采用的是在线跟踪的方式，不使用未来帧的信息。在保持 100fps 以上的帧率的同时，也获得了较高的 MOTA (在当时 16 年的结果中)。

SORT 的贡献主要有三：

- 利用强大的 CNN 检测器的检测结果来进行多目标检测
- 使用基于卡尔曼滤波与**匈牙利算法**的方法来进行跟踪
- 开源了代码，为 MOT 领域提供一个新的 baseline

论文：

Simple Online and Real-time Tracking

<https://arxiv.org/pdf/1602.00763.pdf>

github：

<https://github.com/abewley/sort>

参考：

<https://zhuanlan.zhihu.com/p/62858357>

<https://zhuanlan.zhihu.com/p/59148865>

## 5 · DeepSORT (201703)

DeepSORT 是在 SORT 基础上做的修改，增加了 Deep Association Metric。

论文：

Simple Online and Real-time Tracking with a Deep Association Metric

<https://arxiv.org/pdf/1703.07402.pdf>

github：

[https://github.com/nwojke/deep\\_sort](https://github.com/nwojke/deep_sort)

## 6 · MOTDT (201809)

论文：

<https://arxiv.org/pdf/1809.04427.pdf>

## 7 · JDE(201909)

论文：

<https://arxiv.org/pdf/1909.12605.pdf>

github：

<https://github.com/Zhongdao/Towards-Realtime-MOT>

## 8 · FairMOT (202004)

论文：

<https://arxiv.org/pdf/2004.01888v3.pdf>

github :

<https://github.com/ifzhang/FairMOT>

## (八) 行人识别 (Person Recognition)

类似 Face Recognition，Person Recognition 泛指跟行人（Person）相关的一些研究方向（这些研究方向会有交叉），比如：

- **行人检索：Person Retrieval**，总结图像/视频中的行人的特征，比如头发长短、性别、衣服颜色，是否带眼镜等等，以便在图像/视频中检索出具有特定特征的行人。
- **行人检测：Pedestrian Detection (Person Detection)**，从图像找出行人，是 Object Detection 的特例。
- **行人识别：Person Identification**，类似人脸识别，根据行人图像，从行人特征值数据库找出该图像属于谁。
- **行人再识别：Person Re-Identification，Person Re-ID**，指的是在一组监控摄像头中交叉跟踪一个人（从某个摄像头的范围中走出，之后出现在另一个摄像头范围中），由于不同摄像头的角度、光照、色差、分辨率等等的区别，使得神经网络需要去消除这些区别，Re-ID 作为一个较为通用的技术，在其他领域，比如视频目标跟踪（VOT）也有用到。
- **行人搜索：Person Search**，与行人再识别的区别在于，需要从完整的图像/视频（即监控画面）中找出需要找的人，而 re-ID 的任务是根据目标人，从（裁剪出来的）人形图像库中搜索。也就是说多了个 Person Detection 的任务。

### 1 · 数据集

SoftBioSearch

Market1501

CUHK03

DPM

## 2 · MLCNN (2015ICB)

**Multi-Label CNN** 将人体分为 15 个部分，从上到下 5 行，左中右 3 列。每个部分之间有重叠（overlapping）。然后将这 15 个部分接入一个多标签的卷积神经网络（Multi-Label CNN），每个部分单独卷积。每个标签会和相对应的部分连接（比如头发和头部，衣服和身体部分，短裤和腿部），如下：

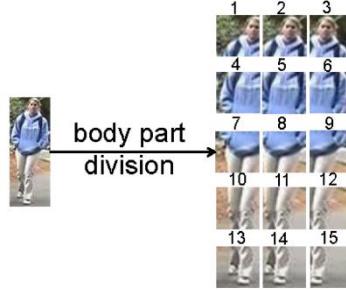


Figure 2. One person is divided into 15 overlapping body parts.

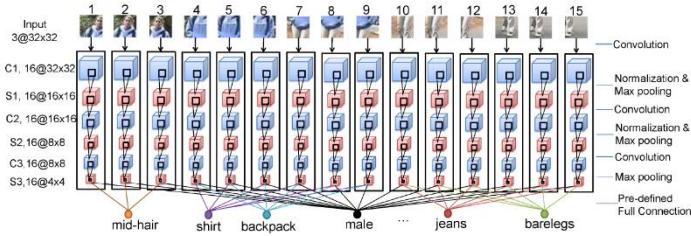


Figure 3. The structure of the multi-label convolutional neural network (MLCNN) used in our method.

每个标签所对应的部分都是预定义好的：

Table 1. The pre-defined connection relationships between parts and attributes.

attribute	parts	attribute	parts	attribute	parts
male	1-15	redshirt	4-9	skirt	10-15
midhair	1-3	blueshirt	4-9	barelegs	10-15
darkhair	1-3	nocoats	4-9	shorts	10-15
bald	1-3	patterned	4-9	lightbottoms	10-15
darkshirt	4-9	hassatchel	4-9	darkbottoms	10-15
lightshirt	4-9	hasbackpack	4-9	jeans	10-15
greenshirt	4-9	hashandbag	7-12	notlightdark	
		carrierbag		jeanscolour	10-15

两张行人图像之间的距离（Fusion Distance）由属性距离（Attribute-based Distance）和底层距离（Low-level based Distance）共同构成：

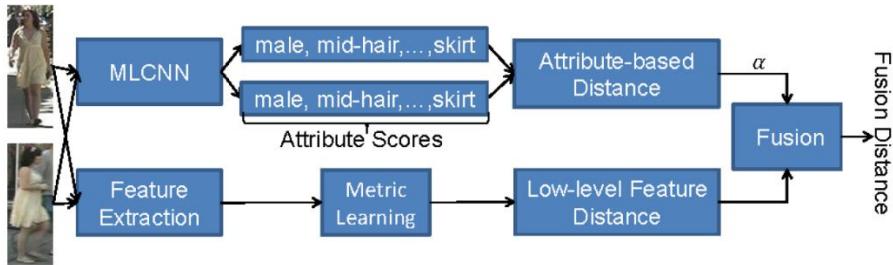


Figure 4. The framework of attribute assisted person re-identification.

论文：

Multi-label CNN Based Pedestrian Attribute Learning for Soft Biometrics



15ICB\_MLCNN.pdf

### 3 · OIM 损失函数 (201604)

《Joint Detection of and Identification Feature Learning for Person Search》的论文作者认为自己的贡献在 3 个方面：

1. 提出了一个新的神经网络
2. 提出了一个新的损失函数 OIM (Online Instance Matching)
3. 创建标注了一个新的数据集

以下为该论文提出的神经网络：

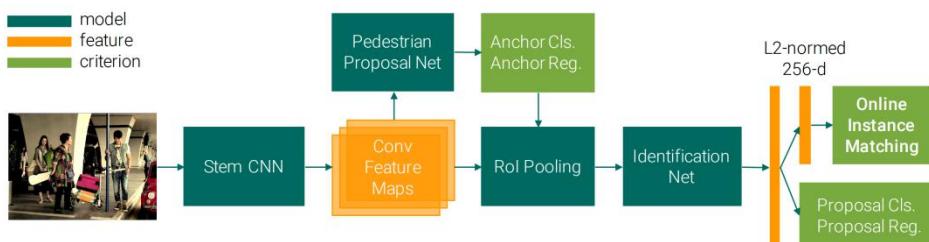


Figure 2. Our proposed framework. Pedestrian proposal net generates bounding boxes of candidate people, which are fed into an identification net for feature extraction. We project the features to a L2-normalized 256-d subspace, and train it with a proposed Online Instance Matching loss. Both the pedestrian proposal net and the identification net share the underlying convolutional feature maps.

- 使用 ResNet-50 作为基础网络，首先通过 Stem CNN (ResNet-50 的前半部分) 提取出特征图。
- 基于 Anchor Box，Pedestrain Proposal Net 负责从特征值中找到并定位行人。
- 接着用 RoI Pooling 池化特征图

- ROI Pooling 出来的结果通过 Identification Net (ResNet-50 的后半部分) 提取出一个 2048 维的特征值
- 这个特征值一方面被送过去过滤可能的假阳性结果 (图中右下)，另一方面被 L2-Normalize 为一个 256 维的 id feat，OIM 使用这个 id feat 来计算和目标行人的余弦距离。

论文提出了一个 LUT (Lookup Table) 用于保存 Labeled Identities 和一个 CQ (Circular Queue) 用于保存 Unlabeled Identities：



Figure 3. Online Instance Matching. The left part shows the labeled (blue) and unlabeled (orange) identity proposals in an image. We maintain a lookup table (LUT) and a circular queue (CQ) to store the features. When forward, each labeled identity is matched with all the stored features. When backward, we update LUT according to the id, pushing new features to CQ, and pop out-of-date ones. Note that both data structures are external buffer, rather than the parameters of the CNN.

论文：

Joint Detection and Identification Feature Learning for Person Search

<https://arxiv.org/pdf/1604.01850.pdf>

代码：

[https://github.com/ShuangLI59/person\\_search](https://github.com/ShuangLI59/person_search)

## 4 · PCB 和 RPP (201711)

《Beyond Part Models : Person Retrieval with Refined Part Pooling (and A Strong Convolutional Baseline)》论文有两个贡献：

- 提出了一个新的神经网络 PCB (Part-based Convolutional Baseline)
- 提出了一个新的分块 (Part) 方法 RPP (Refined Part Pooling)

通常现行的 Part 方法分成两种：

- 利用外部方法进行分块，比如使用 Human Pose Estimation 网络
- 不利用外部方法，比如 PAR 和本文

PCB 的基础网络可以是任何主流分类网络，比如 Inception 或 ResNet，本文使用了 ResNet50 作为基础网络。

论文：

Beyond Part Models: Person Retrieval with Refined Part Pooling  
(and A Strong Convolutional Baseline)

<https://arxiv.org/pdf/1711.09349>

代码：

[https://github.com/layumi/Person\\_reID\\_baseline\\_pytorch](https://github.com/layumi/Person_reID_baseline_pytorch)

## 5 · Height, Color, Gender (201810)

《Person Retrieval in Surveillance Video using Height, Color and Gender》这篇论文提出了一个若干步骤的方法，通过身高、颜色和性别进行行人检索：

1. 首先通过 Mask R-CNN 得到视频/图像中的行人的语义分割（像素级别）
2. 将所有得到的行人（语义分割）输入 Height Filter，根据相机的参数（安装位置、角度、焦距等），通过矩阵计算（Tsai Camera Calibration）得到行人的身高（取视频中各帧的平均值），对比输入的身高条件，过滤掉一部分行人。
3. 将剩下的行人输入 Color Filter，根据 Height Filter 计算出的身高，切取从上到下 20% 到 50% 的部分作为上身，50% 到 100% 作为腿，将这两部分人体切片送入 AlexNet 为基础的颜色分类器得到颜色，与输入的颜色条件，得到匹配的结果
4. 如果需要，将上一步得到的行人结果（全身的语义分割）送入 AlexNet 为基础的性别分类器得到性别，与输入的性别条件比对，得到结果。

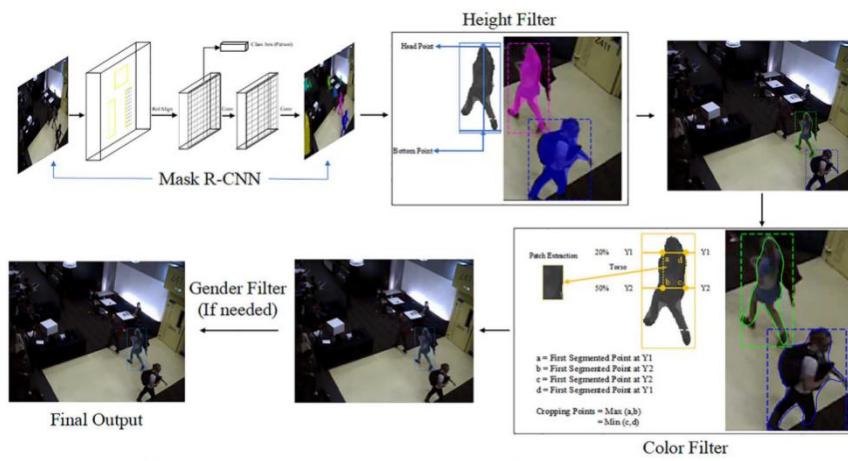


Figure 1: Proposed approach of person retrieval using height, cloth color and gender.

论文：

Person Retrieval in Surveillance Video using Height, Color and Gender

<https://arxiv.org/pdf/1810.05080>

## 6 · st-ReID (201812)

论文：

Spatial-Temporal Person Re-identification

[https://arxiv.org/pdf/1812.03282](https://arxiv.org/pdf/1812.03282.pdf)

代码：

<https://github.com/Wanggcong/Spatial-Temporal-Re-identification>

## 7 · DG-net (201904)

论文：

Joint Discriminative and Generative Learning for Person Re-identification

[https://arxiv.org/pdf/1904.07223](https://arxiv.org/pdf/1904.07223.pdf)

代码：

<https://github.com/NV1abs/DG-Net>

# 九、研究方向：自然语言处理（NLP）

Natural Language Processing，自然语言处理，这个范畴本身也是超越深度学习/神经网络的。在起初，RNN 是 NLP 的一个非常重要的研究方向，Transformer 则后来居上。

早期的 NLP 使用 One-hot 或者词袋模型来表示词，之后 Word2Vec 的出现使词嵌入成为表示词汇的主流方式，再之后预训练模型（GPT 系列、BERT 系列）的出现使得 Transformer 成为主流。

## 论文

中文 NLP 模型综述

<https://arxiv.org/pdf/2004.13922.pdf>

## 参考

NLP 的巨人肩膀

<https://zhuanlan.zhihu.com/p/50443871>

自然语言处理中注意力机制综述

[https://zhuanlan.zhihu.com/p/54491016?utm\\_source=qq&utm\\_medium=social&utm\\_oi=616755169208307712](https://zhuanlan.zhihu.com/p/54491016?utm_source=qq&utm_medium=social&utm_oi=616755169208307712)

<https://blog.csdn.net/jiaowoshouzi/article/details/89073944>

## （一）概念

### 1 · 语言学相关

这里是语言学相关的一些概念，对理解 NLP 任务有帮助。

- Phonology（音位）：语言学分支，研究语音的功能
- Orthography（正写）：书写一门语言的惯例，比如英语中的拼写、大小写、标点等
- Morphology（词法）：语言学的分支，研究单词的内部结构和形成方式
- Morpheme（语素）：指最小的语法单位，是最小的语音语义结合体，英语中比如 unbreakable 这个词可以分为 un-、break、-able 三个语素。
- Syntax（句法）：即句子的格式，主谓宾这样的结构是否正确，句法正确的句子不一定有意义。

- Semantics (语义) : 简言之即语言 (或者逻辑等) 的意义
- Pragmatics (语用) : 语用研究的是不同语境对语言含义的影响，但其和语义间的区别变得越来越模糊
- Grammar (语法) : 也叫句法，是个范围比较大的概念，包括 Phonology 、 Morphology 、 Syntax ，也包括 Phonetics 、 Semantics 、 Pragmatics 。
- Part of Speech (POS) : 词性，比如动词、名词、形容词。

论文：

<http://www.jmlr.org/papers/volume3/bengio03a/bengio03a.pdf>

## 2 · Tokenize (分词)

Tokenize 即分词，这是 NLP 的第一步，将文本分割为模型可识别的单位 token (通常是词，有的时候是子词)，因此 tokenizer (分词器) 和模型是对应的。其中还有一个隐含的步骤是将切割后的词转换为对应的 id，最终转换为模型可识别的 encodings

## 3 · TF-IDF

- TF : Term Frequency (词频) , 词的量化表示的一种方式。某词在文档中出现的频率，通常用词频除以文档总的词数来归一化。
- IDF : Inverse Document Frequency (逆文档频率) , 词的量化表示的一种方式。语料库中的文档总数/包含该词的文档数。

## 4 · 语言模型 (LM)

语言模型 (Language Model) 可以简单理解为一个句子  $s$  在所有句子中出现的概率分布  $P(s)$ 。而考虑到  $s$  是一个由词构成的序列  $w_1, w_2, \dots, w_n$ ， $P(s)$  的概率可表示为：

$P(w_1, w_2, \dots, w_n)$

再展开得到：

$$P(w_1, w_2, \dots, w_n) = P(w_1) \cdot P(w_2 | w_1) \cdot P(w_3 | w_1, w_2) \cdots P(w_n | w_1, w_2, \dots, w_{n-1})$$

参考：

<https://zhuanlan.zhihu.com/p/43453548>

## 5 · n-grams (n 元语法)

语言模型的计算和存储复杂度会随着词汇表和文本序列的增长而变得过大而无法计算。n-grams (n 元语法) 通过马尔可夫假设（虽然并不一定成立）简化了语言模型的计算。这里的马尔可夫假设是指一个词的出现只与前面  $n-1$  个词相关，即  $n-1$  阶马尔可夫链 (Markov chain of order  $n$ )。例如  $n=1$ ，那么有  $P(w_3 | w_1, w_2) = P(w_3 | w_2)$ 。如果基于  $n-1$  阶马尔可夫链，我们可以将语言模型改写为：

$$P(w_1, w_2, \dots, w_T) \approx \prod_{t=1}^T P(w_t | w_{t-(n-1)}, \dots, w_{t-1})$$

以上也叫 **n 元语法 (n-grams)**。它是基于  $n-1$  阶马尔可夫链的概率语言模型。当  $n$  分别为 1、2 和 3 时，我们将其分别称作一元语法 (unigram)、二元语法 (bigram) 和三元语法 (trigram)。例如，长度为 4 的序列  $w_1, w_2, w_3, w_4$  在一元语法、二元语法和三元语法中的概率分别为：

一元语法： $P(w_1, w_2, w_3, w_4) = P(w_1) \cdot P(w_2) \cdot P(w_3) \cdot P(w_4)$

二元语法： $P(w_1, w_2, w_3, w_4) = P(w_1) \cdot P(w_2 | w_1) \cdot P(w_3 | w_2) \cdot P(w_4 | w_3)$

三元语法： $P(w_1, w_2, w_3, w_4) = P(w_1) \cdot P(w_2 | w_1) \cdot P(w_3 | w_1, w_2) \cdot P(w_4 | w_2, w_3)$

当  $n$  较小时， $n$  元语法往往并不准确。例如，在一元语法中，由三个词组成的句子“你走先”和“你先走”的概率是一样的。然而，当  $n$  较大时， $n$  元语法需要计算并存储大量的词频和多词相邻频率。

## (二) 任务

NLP 的任务比较复杂，并且还在不断发展，并没有一个较为官方的或者统一的分类。

传统的 NLP 任务定义泛指和自然语言相关的任务，因此从传统意义上说：

- 语音识别 (Speech recognition, 语音转文字)
- 语音合成 (Text-to-Speech, 文字转语音)
- 语音分段 (Speech segmentation)
- OCR (Optical Character Recognition, 光学字符识别)

这些也应该算是 NLP。但现在在深度学习研究领域里，所说的 NLP 任务只包括文字相关的任务，这可能是因为大部分难点和主要的研究方向都集中在这部分。

NLP 的任务种类比较复杂：

- 种类繁多，且任务之间的关系交织密切（有交叉，有包含）
- 根据关注点和能力（比如知识图谱）的发展，也在不断的发展新任务
- 分类方式和维度也各有不同，不同分类方式中的任务名称和定义可能又有区别

- 由于语言的相关特性，以及该语言 NLP 的发展，不同语言的任务也会有所区别

## 1 · 语言学 NLP 任务分类

以下是维基百科的 **Natural Language Processing** 词条里，对 NLP 任务的分类，这里有些是可以应用的真实世界的任务，有些则是子任务。

维基百科的 NLP 任务分类的特点是：

- 基于英文
- 比较详细
- 更学术化
- 更偏向语言学

可以将此作为理解神经网络 NLP 方向的任务分类基础。

- **Text and Speech Processing** : 文本和语音处理
  - **OCR** : 光学字符识别，识别图像中的文字
  - **Speech Recognition** : 语音识别，语音片段转化为文字
  - **Speech Segmentation** : 语音分段，将语音片段分词，语音识别的子任务
  - **TTS (Text to Speech)** : 语音合成，文字转化为语音
  - **Tokenization** : 也叫 **Word Segmentation** , 分词，将文字片段拆分成词
- **Morphological Analysis** : 词法分析
  - **Lemmatization** : 词元化，将词还原为其基本形式，例如去掉各种时态语态的变化，通常是通过字典来完成
  - **Stemming** : 词干提取，类似 Lemmatization，区别在于 Stemming 的工作方式相对简单粗暴，一般是通过切除前缀或者后缀来实现。
  - **Morphological Segmentation** : 词分割，将词拆分成语素，例如 “unbreakable” 输出 “un-break-able”
  - **POST (Part-of-speech Tagging)** : 词性标注，标定词汇的词性（动词还是名词）
- **Syntactic Analysis** : 句法分析
  - **Grammar Induction** : 语法规纳，根据输入，产生出该语言的语法规则
  - **Sentence Breaking** : 句子分割，给定一段文本，切分成句子
  - **Parsing** : 解析，输入句子，生成解析树 (Parsing tree)
- **Lexical Semantics** : 词汇语义
  - **Lexical Semantics** : 词汇语义，单个词汇在上下文中的计算意义（比如词嵌入）
  - **Distributional Semantics** : 分布式语义，分布式语义基于一个分布式假设：有类似分布的语义元素有类似的意义（即在同样的上下文环境中出现的词有类似的意思）
  - **NER (Named Entity Recognition)** : 命名实体识别，识别文本中具有特定意义的实体，包括人名、地名、机构名、专有名词等，以及时间、数量、货币、比例等文字。
  - **Sentiment Analysis** : 情感分析，分析文本的情感倾向（正面/负面/中性）
  - **Terminology Extraction** : 关键词提取，从文本中提取相关的关键词

- **Word Sense Disambiguation**：词义消歧，很多词都有若干意义，该任务是为了选择词汇在上下文环境中到底是哪个意义。
- **Relational Semantics**：关系语义
  - **RE (Relationship Extraction)**：关系抽取，类似 Information Extraction，从输入中检测并分类语义关系
  - **Semantic Parsing**：语义解析，将自然语言表达转换成逻辑形式，通常由几个更基本的子任务构成。可以应用于更高级的任务比如机器翻译、智能问答中。
  - **SRL (Semantic Role Labeling)**：也叫 **Shallow Semantic Parsing**，语义角色标注，给句子中的词添加语义角色标签，简单的比如主谓宾等。
- **Discourse**：言谈
  - **Coreference resolution**：共指消解，从一段文本或者对话中找出指代同一个人或事物的所有词汇
  - **DA (Discourse Analysis)**：言谈分析，这是个更偏语言学的任务
  - **Implicit Semantic Role Labelling**：隐性语义角色标注，在给句子做语义角色标注的基础上，给词标注出句子间的语义角色，有点类似共指消解。
  - **TE (Recognizing Texture entailment)**：文字蕴涵，判断两段文本（文本 T 和假设 H）的关系是正向蕴涵、矛盾蕴涵还是独立蕴涵。
  - **Topic Segmentation and Recognition**：主题分割和识别，将文本分割为段落，并识别出段落的主题
- **High-Level NLP Application**：高级 NLP 应用，每个都包含了若干较下层的子任务
  - **Text Summarization**：文本摘要，也叫 **Automatic Summarization**，提取文章的主要内容
  - **Book Generation**：书籍生成
  - **Dialogue Management**：对话管理，也叫 **CA (Conversational Agent)**，类似智能问答，区别在于并不一定是以问答的形式
  - **Document AI**：文档 AI，从文档中抽取所需的特定数据
  - **Grammatical Error Correction**：语法错误纠正，包括语法错误的检测和纠正
  - **Machine Translation**：机器翻译
  - **NLG (Natural Language Generation)**：自然语言生成，这里指的是从数据库等结构化数据中转换为（人类易于读懂的）文本。
  - **NLU (Natural Language Understanding)**：自然语言理解，跟 NLG 相反，将文本段落转换为计算机更易于理解的结构
  - **Question Answering**：智能问答，通常是简单的有确定答案的问题（比如“今天星期几”），但有时候也会需要考虑开放式的问题。

参考：

[https://en.wikipedia.org/wiki/Natural\\_language\\_processing](https://en.wikipedia.org/wiki/Natural_language_processing)

## 2 · 神经网络 NLP 任务分类

神经网络中的 NLP 任务和维基百科中的语言学 NLP 任务分类多有重合，下面为在语言学 NLP 的基础上所做的补充：

- Language Modelling :
- Constituency Parsing (成分句法分析) : 属于 Semantic Parsing
- Dependency Parsing (依赖句法分析) : 属于 Semantic Parsing
- Stance Detection (立场检测) : 分类任务，输入是两个序列，输出是一个类别，表示后面的序列是否与前面的序列站在同一立场。常用的立场检测包括 SDQC 四种标签，支持 (Support)，否定 (Denying)，怀疑 (Querying)，Commenting (注释)。
- Veracity Prediction (事实验证) : ，也叫 Fake News Detection。
- Search Engine (搜索引擎) : 模型的输入是一个关键词或一个问句和一堆文章，输出是每篇文章与该问句的相关性。谷歌有把 BERT 用在搜索引擎上，在语义理解上得到了提升。
- New Word Detection (新词发现) :
- Text Classification (文本分类) :
- 文本匹配
- IE (Information Extraction, 信息抽取) :
- Reading Comprehension (阅读理解)
- Knowledge Graph (知识图谱)
- AMR (Abstract Meaning Representation, 抽象意义表示)

参考：

<https://paperswithcode.com/area/natural-language-processing>

<https://zhuanlan.zhihu.com/p/163281686>

<https://zhuanlan.zhihu.com/p/50755570>

透过现象看本质，仅从任务的输出、输出的角度看，可以将 NLP 大致分为 NLU 和 NLG：

- NLU : Natural Language Understand，自然语言理解，通常输入是文本序列，输出是分类（或者标签），又可以分为：
  - 输入是句子，输出是分类：情感分析、文本分类
  - 输入是句子对，输出是分类
  - 输入是句子，输出是标签
  - ...
- NLG : Natural Language Generate，自然语言生成，通常输入是文本序列，输出也是文本序列
  - 机器翻译
  - 智能问答
  - 文本摘要
  - ...

## (三) 衡量标准

### 1 · Accuracy

参考《概念定义 > NN 相关 > 二分类任务衡量标准》

### 2 · F1-Score

参考《概念定义 > NN 相关 > 二分类任务衡量标准》

### 3 · BLEU

**BiLingual Evaluation Understudy**，一个用于衡量机器翻译的指标。其背后的原则是：机器翻译的结果越接近人工翻译的结果，得分越高。

## (四) 数据集

NLP 的数据集也叫语料库 (corpus)

### 1 · ChnSentiCorp

中文的情感分析语料库，来自携程评论，文本分类任务，将每个样本（评论）标示为正面或者负面。

下载地址：

<http://file.hankcs.com/corpus/ChnSentiCorp.zip>

### 2 · SQuAD

<https://rajpurkar.github.io/SQuAD-explorer/>

### 3 · GLUE

GLUE (General Language Understanding Evaluation) 是目前业界通用的，用于测试 NLP 模型能力的 Benchmark，包含了若干任务（的数据集及衡量标准），主流的 NLP 模型都在上面测试过。GLUE 全部是分类任务，包括以下数据集/任务：

- **CoLA** : The Corpus of Linguistic Acceptability，语言可接受性语料库，即判断输入可接受程度（是否合乎语法），语料来自语言理论的书籍和期刊，每个句子被标注为是否合乎语法的单词序列。**二分类任务**，结果为 1 (合乎语法) 或者 0 (不合乎语法)。
- **SST-2** : The Stanford Sentiment Treebank，斯坦福感情树库。包含电影评论中的句子和它们情感的人类注释。是判断句子的情感分析任务，是个**二分类任务**，结果为正面或者负面。
- **MRPC** : The Microsoft Research Paraphrase Corpus，微软研究院释义语料库，**相似性和释义任务**，是从在线新闻源中自动抽取句子对成为语料库。**二分类任务**，判断输入的两者是否互为释义。
- **STSB** : The Semantic Textual Similarity Benchmark，语义文本相似性基准测试，**相似性和释义任务**，判断输入句子对的相似性。是从新闻标题、视频标题、图像标题以及自然语言推断数据中提取的句子对的集合，每对都是由人类注释的，任务就是预测这些相似性得分（从 0 到 5），本质上是一个回归问题，但是依然可以用分类的方法，可以归类为句子对的文本**五分类任务**。
- **QQP** : The Quora Question Pairs，Quora 问题对数集，**相似性和释义任务**，是社区问答网站 Quora 中问题对的集合。任务是确定一对问题在语义上是否等效。**二分类任务**，等效或者不等效。
- **MNLI** : The Multi-Genre Natural Language Inference Corpus，多类型自然语言推理数据库，**自然语言推断任务**，是通过众包方式对句子对进行文本蕴含标注的集合。给定前提 (premise) 语句和假设 (hypothesis) 语句，任务是预测前提语句是否包含假设 (蕴含, entailment)，与假设矛盾 (矛盾, contradiction) 或者两者都不 (中立, neutral)，是个**三分类任务**。
- **QNLI** : Qusetion-answering NLI，问答自然语言推断，**自然语言推断任务**。QNLI 是从另一个数据集 The Stanford Question Answering Dataset(斯坦福问答数据集, SQuAD 1.0)转换而来的。SQuAD 1.0 是有一个问题-段落对组成的问答数据集，其中段落来自维基百科，段落中的一个句子包含问题的答案。这里可以看到有个要素，来自维基百科的段落，问题，段落中的一个句子包含问题的答案。通过将问题和上下文（即维基百科段落）中的每一句话进行组合，并过滤掉词汇重叠比较低的句子对就得到了 QNLI 中的句子对。相比原始 SQuAD 任务，消除了模型选择准确答案的要求；也消除了简化的假设，即答案适中在输入中并且词汇重叠是可靠的提示。是个**二分类任务**，蕴含和不蕴含。
- **RTE** : The Recognizing Textual Entailment datasets，识别文本蕴含数据集，**自然语言推断任务**，它是将一系列的年度文本蕴含挑战赛的数据集进行整合合并而来的，包含 RTE1, RTE2, RTE3, RTE5 等，这些数据样本都从新闻和维基百科构建而来。将这些所有数据转换为**二分类任务**（蕴含和不蕴含），对于三分类的数据，为了保持一致

性，将中立（neutral）和矛盾（contradiction）转换为不蕴含（not entailment）。

- **WNLI**：Winograd NLI，Winograd 自然语言推断，自然语言推断任务，数据集来自于竞赛数据的转换。Winograd Schema Challenge，该竞赛是一项阅读理解任务，其中系统必须读一个带有代词的句子，并从列表中找到代词的指代对象。这些样本都是手动创建的，以挫败简单的统计方法：每个样本都取决于句子中单个单词或短语提供的上下文信息。为了将问题转换成句子对分类，方法是通过用每个可能的列表中的每个可能的指代去替换原始句子中的代词。任务是预测两个句子对是否有关（蕴含、不蕴含），因此是个**二分类任务**。训练集两个类别是均衡的，测试集是不均衡的，65%是不蕴含。

可以看出来，GLUE 包含的任务全部是**分类任务**，输入是单个句子或者句子对：

- 输入单句，输出二分类：CoLA、SST-2
- 输入句对，输出二分类：MRPC、QQP、QNLI、RTE、WNLI
- 输入句对，输出多分类：STSB、MNLI

以下为 GLUE 官方网站上的排行榜 LeaderBoard：

Rank Name	Model	URL	Score	CoLA	SST-2	MRPC	STS-B	QQP	MNLI-m	MNLI-mm	QNLI	RTE	WNLI	AX	
1	DeBERTa Team - Microsoft	DeBERTa / TuringNLVRv4		90.8	71.5	97.5	94.0/92.0	92.9/92.6	76.2/90.8	91.9	91.6	99.2	93.2	94.5	53.2
2	HFL IFLYTEK	MacALBERT + DKM		90.7	74.8	97.0	94.5/92.6	92.8/92.6	74.7/90.6	91.3	91.1	97.8	92.0	94.5	52.6
3	Alibaba DAMO NLP	StructBERT + TAPT		90.6	75.3	97.3	93.9/91.9	93.2/92.7	74.8/91.0	90.9	90.7	97.4	91.2	94.5	49.1
4	PING-AN Omni-SinTIC	ALBERT + DAAF + NAS		90.6	73.5	97.2	94.0/92.0	93.0/92.4	76.1/91.0	91.6	91.3	97.5	91.7	94.5	51.2
5	ERNIE Team - Baidu	ERNIE		90.4	74.4	97.5	93.5/91.4	93.0/92.6	75.2/90.9	91.4	91.0	96.6	90.9	94.5	51.7
6	T5 Team - Google	T5		90.3	71.6	97.5	92.8/90.4	93.1/92.8	75.1/90.6	92.2	91.9	96.9	92.8	94.5	53.1
7	Microsoft D365 AI & MSR AI & GATECHMT-DNN-SMART		89.9	69.5	97.5	93.7/91.6	92.9/92.5	73.9/90.2	91.0	90.8	99.2	89.7	94.5	50.2	
8	Huawei Noah's Ark Lab	NEZHA-Large		89.8	71.7	97.3	93.3/91.0	92.4/91.9	75.2/90.7	91.5	91.3	96.2	90.3	94.5	47.9
9	Zihang Dai	Funnel-Transformer (Ensemble B10-10-H1024)		89.7	70.5	97.5	93.4/91.2	92.6/92.3	75.4/90.7	91.4	91.1	95.8	90.0	94.5	51.6
10	ELECTRA Team	ELECTRA-Large + Standard Tricks		89.4	71.7	97.1	93.1/90.7	92.9/92.5	75.6/90.8	91.3	90.8	95.8	89.8	91.8	50.7
11	Microsoft D365 AI & UMD	FreeLB-RoBERTa (ensemble)		88.4	68.0	96.8	93.1/90.8	92.3/92.1	74.8/90.3	91.1	90.7	95.6	88.7	89.0	50.1
12	Junjie Yang	HIRE-RoBERTa		88.3	68.6	97.1	93.0/90.7	92.4/92.0	74.3/90.2	90.7	90.4	95.5	87.9	89.0	49.3
13	Facebook AI	RoBERTa		88.1	67.8	96.7	92.3/89.8	92.2/91.9	74.3/90.2	90.8	90.2	95.4	88.2	89.0	48.7
14	Microsoft D365 AI & MSR AI	MT-DNN-ensemble		87.6	68.4	96.5	92.7/90.3	91.1/90.7	73.7/89.9	87.9	87.4	96.0	86.3	89.0	42.8
15	GLUE Human Baselines	GLUE Human Baselines		87.1	66.4	97.8	86.3/80.8	92.7/92.6	59.5/80.4	92.0	92.8	91.2	93.6	95.9	-
16	Adrian de Wynter	Bort (Alexa AI)		83.6	63.9	96.2	94.1/92.3	89.2/88.3	66.0/85.9	88.1	87.8	92.3	82.7	71.2	51.9
17	Lab LV	ConvBERT base		83.2	67.8	95.7	91.4/88.3	90.4/89.7	73.0/90.0	88.3	87.4	93.2	77.9	65.1	42.9

论文：

<https://arxiv.org/pdf/1804.07461.pdf>

官方 github：

<https://github.com/nyu-mll/GLUE-baselines>

官方网站：

<https://gluebenchmark.com/>

参考：

<https://zhuanlan.zhihu.com/p/135283598>

## 4 · SuperGLUE

论文：

<https://arxiv.org/pdf/1905.00537.pdf>

官方网站：

<https://super.gluebenchmark.com/>

代码：

<https://jiant.info/>

## 5 · CLUE

CLUE (Chinese GLUE) ，即中文 GLUE。中文语言理解测评基准，包括代表性的数据集、基准(预训练)模型、语料库、排行榜，是用于衡量中文 NLP 模型能力的一个通用的数据集/任务集合。

具体的任务如下：

- **AFQMC** : Ant Financial Question Matching Corpus，蚂蚁金融语义相似度。输入为句子对，输出二分类。每一条数据有三个属性，从前往后分别是 句子1，句子2，句子相似度标签。其中 label 标签，1 表示 sentence1 和 sentence2 的含义类似，0 表示两个句子的含义不同。
- **TNEWS'** : Short Text Classification for News。文本分类任务。输入为句子，输出为多分类。每一条数据有三个属性，从前往后分别是 分类 ID，分类名称，新闻字符串（仅含标题）。该数据集来自今日头条的新闻版块，共提取了 15 个类别的新闻，包括旅游，教育，金融，军事等。
- **IFLYTEK'** : Long Text Classification，长文本分类任务。输入为长文本，输出为多分类。该数据集共有 1.7 万多条关于 app 应用描述的长文本标注数据，包含和日常生活相关的各类应用主题，共 119 个类别：“打车”:0，“地图导航”:1，“免费 WIFI”:2，“租车”:3, …, “女性”:115, “经营”:116, “收款”:117, “其他”:118(分别用 0-118 表示)。每一条数据有三个属性，从前往后分别是类别 ID，类别名称，文本内容。
- **OCNLI** : Original Chinese Natural Language Inference，中文原版自然语言推理。输入为句子对，输出为分类。是第一个非翻译的、使用原生汉语的大型中文自然语言

推理数据集。OCNLI 包含 5 万余训练数据，3 千验证数据及 3 千测试数据。除测试数据外，我们将提供数据及标签。测试数据仅提供数据。OCNLI 为中文语言理解基准测评（CLUE）的一部分。中文原版数据集 OCNLI 替代了 CMNLI，使用 bert\_base 作为初始化分数。

- **CMNLI** : Chinese Multi-Genre NLI，中文语言推理任务（已被 OCNLI 替代）。CMNLI 数据由两部分组成：XNLI 和 MNLI。数据来自于 fiction, telephone, travel, government, slate 等，对原始 MNLI 数据和 XNLI 数据进行了中英文转化，保留原始训练集，合并 XNLI 中的 dev 和 MNLI 中的 matched 作为 CMNLI 的 dev，合并 XNLI 中的 test 和 MNLI 中的 mismatched 作为 CMNLI 的 test，并打乱顺序。该数据集可用于判断给定的两个句子之间属于蕴涵、中立、矛盾关系。每一条数据有三个属性，从前往后分别是 句子 1，句子 2，蕴含关系标签。其中 label 标签有三种：neutral, entailment, contradiction。
- **WSC2020** : WSC Winograd 模式挑战中文版，Winograd 模式是图灵测试的一个变种，旨在判定 AI 系统的常识推理能力。参与挑战的计算机程序需要回答一种特殊但简易的常识问题：代词消歧问题，即对给定的名词和代词判断是否指代一致。其中 label 标签，true 表示指代一致，false 表示指代不一致。
- **CSL** : Keyword Recognition, 论文关键词识别任务。中文科技文献数据集 (CSL) 包含中文核心论文摘要及其关键词。用 tf-idf 生成伪造关键词与论文真实关键词混合，生成摘要-关键词对，关键词中包含伪造的则标签为 0。每一条数据有四个属性，从前往后分别是 数据 ID，论文摘要，关键词，真假标签。
- **CMRC2018** : Reading Comprehension for Simplified Chinese，简体中文阅读理解任务。CMRC2018 是讯飞的中文阅读理解评测。
- **DRCD** : Reading Comprehension for Traditional Chinese，繁体中文阅读理解任务。台達閱讀理解資料集 Delta Reading Comprehension Dataset (DRCD)，屬於通用領域繁體中文機器閱讀理解資料集。本資料集期望成為適用於遷移學習之標準中文閱讀理解資料集。
- **ChID** : 成语阅读理解填空 Chinese IDiom Dataset for Cloze Test。成语完形填空，文中多处成语被 mask，候选项中包含了近义的成语。
- **C3** : Multiple-Choice Chinese Machine Reading Comprehension，中文多选阅读理解数据集，包含对话和长文等混合类型数据集。

CLUE 包含的任务全部是**分类任务**，输入是单个句子或者句子对：

- 输入单句，输出二分类：CoLA、SST-2
- 输入句对，输出二分类：MRPC、QQP、QNLI、RTE、WNLI
- 输入句对，输出多分类：STS-B、MNLI

官方网站：

<https://www.cluebenchmarks.com/>

论文：

<https://arxiv.org/pdf/2004.05986.pdf>

github :

旧版：<https://github.com/chineseGLUE/chineseGLUE>

新版：<https://github.com/CLUEbenchmark/CLUE>

## (五) 框架

严格来说，这里列出的也包括各种 NLP 模型的库/工具包，而不仅仅是框架。

### 1 · SpaCy

### 2 · NLTK

### 3 · Stanford CoreNLP (2014)

Stanford CoreNLP 是个已经训练好的 NLP 模型，支持多种语言。

在各个语言上的能力如下表：

Annotator	ar	zh	en	fr	de	es
Tokenize / Segment	✓	✓	✓	✓		✓
Sentence Split	✓	✓	✓	✓	✓	✓
Part of Speech	✓	✓	✓	✓	✓	✓
Lemma			✓			
Named Entities		✓	✓	✓	✓	✓
Constituency Parsing	✓	✓	✓	✓	✓	✓
Dependency Parsing		✓	✓	✓	✓	
Sentiment Analysis			✓			
Mention Detection		✓	✓			
Coreference		✓	✓			
Open IE			✓			

官方：

<https://stanfordnlp.github.io/CoreNLP/>

论文：

<https://nlp.stanford.edu/pubs/StanfordCoreNLP2014.pdf>

github：

<https://github.com/stanfordnlp/CoreNLP>

参考：

[CSDN：StanfordNLP 的安装及使用](#)

[知乎：Stanford CoreNLP 入门指南](#)

[StanfordCoreNLP 的简单使用](#)

## 4 · Gensim

Gensim 是用于特定文本主题建模的高端行业级软件。它的功能非常强大，独立于平台，并且具有可扩展性。不仅可以用来判断两个报纸文章之间的语义相似性，而且可以利用简单的函数调用执行此操作并返回其相似度分数，非常方便！

任务：主题建模，文本摘要，语义相似度

官网：

<https://radimrehurek.com/gensim/>

github：

<https://github.com/RaRe-Technologies/gensim>

## 5 · OpenNMT

OpenNMT 是用于机器翻译和序列学习任务的便捷而强大的工具。其包含的高度可配置的模型和培训过程，让它成为了一个非常简单的框架。

官网：

<https://opennmt.net/>

github：

<https://github.com/OpenNMT/OpenNMT-py>

## 6 · Par1AI

Par1AI 是 Facebook 的 #1 框架，用于共享、训练和测试用于各种对话任务的对话模型。其提供了一个支持多种参考模型、预训练模型、数据集等的多合一环境。

官网：

<https://par1.ai/>

github：

<https://github.com/facebookresearch/Par1AI>

## 7 · DeepPavlov

官网：

<https://deppavlov.ai/>

github：

<https://github.com/deepmipt/DeepPavlov>

## 8 · SnowNLP

SnowNLP 是个非神经网络的 NLP 库，比较老，最后更新已经是 2017 年，优点是 **非常易用**，且由于并未使用神经网络，**速度较快**。

SnowNLP 是一个 python 写的类库，可以方便的处理中文文本内容，是受到了 TextBlob 的启发而写的，由于现在大部分的自然语言处理库基本都是针对英文的，于是写了一个方便处理中文的类库，并且和 TextBlob 不同的是，这里没有用 NLTK，所有的算法都是自己实现的，并且自带了一些训练好的字典。

github：

<https://github.com/isnowfy/snownlp>

使用示例（情感分析）：

```
$ pip install snownlp
from snownlp import SnowNLP
s = SnowNLP(content)
result = s.sentiments
```

训练示例（情感分析）：

```
from snownlp import sentiment
sentiment.train('neg.txt', 'pos.txt')
sentiment.save('sentiment.marshall')
```

之后修改 snownlp/sentiment/\_\_init\_\_.py 里的 data\_path 指向刚训练好的文件即可

## 9 · Senta

百度推出的情感分析的 NLP 库

github：

<https://github.com/baidu/senta>

论文：

<https://arxiv.org/abs/2005.05635>

使用示例（情感分析）：

```
from senta import Senta
my_senta = Senta()
my_senta.init_model(task="sentiment_classify")
result1 = my_senta.predict(text)
```

## 10 · HuggingFace Transformers

Transformers 是 [HuggingFace](#) 推出的包含各种基于 transformer 结构的 NLP 模型的工具包。包含了 Pytorch 和 Tensorflow 两种格式的模型。

2017 年，transformer 结构被提出，之后 NLP 的方向开始从 RNN 转向 Transformer，2018 年，Google 发布了基于 Transformer 的 NLP 预训练模型 BERT，之后基于 BERT 的各种模型（XLNet、ALBERT、RoBERTa...）层出不穷，不断刷新纪录。

BERTs 模型虽然很香，但是用起来还是有一些障碍，比如：

- 预训练需要大量的资源，一般研究者无法承担。以 RoBERTa 为例，它在 160GB 文本上利用 1024 块 32GB 显存的 V100 卡训练得到，如果换算成 AWS 上的云计算资源的话，但这一模型就需要 10 万美元的开销。
- 很多大机构的预训练模型被分享出来，但没有得到很好的组织和管理。

- BERT 系列的各种模型虽然师出同源，但在模型细节和调用接口上还是有不少变种，用起来容易踩坑

为了让这些预训练语言模型使用起来更加方便，Huggingface 在 github 上开源了 Transformers。这个项目开源之后备受推崇，截止 2020 年 5 月，已经累积了 26k 的 star 和超过 6.4k 的 fork。

Transformers 最早的名字叫做 pytorch-pretrained-bert，推出于 google BERT 之后。顾名思义，它是基于 pytorch 对 BERT 的一种实现。pytorch 框架上手简单，BERT 模型性能卓越，集合了两者优点的 pytorch-pretrained-bert 自然吸引了大批的追随者和贡献者。

其后，在社区的努力下，GPT、GPT-2、Transformer-XL、XLNET、XLM 等一批模型也被相继引入，整个家族愈发壮大，这个库适时地更名为 pytorch-transformers。

之后又加入了 pytorch 和 TF2.0+ 的互操作性，使得 pytorch 和 tensorflow 两大阵营的模型之间可以相互转换，项目的名字，也改成了现在的 Transformers。时至今日，Transformers 已经在 100+ 种人类语言上提供了 32+ 种预训练语言模型。

官方文档：

<https://huggingface.co/transformers/>

哈工大和讯飞联合实验室训练的中文模型：

<https://huggingface.co/hf1>

参考：

<https://zhuanlan.zhihu.com/p/141527015>

github：

Transformers：<https://github.com/huggingface/transformers>

数据集操作库（原名 nlp）：<https://github.com/huggingface/datasets>

论文：

<https://arxiv.org/pdf/1910.03771.pdf>

## (1) 示例

安装：

```
$ pip install transformers
```

使用示例（情感分析）：

```
from transformers import pipeline
p = pipeline("sentiment-analysis")
```

```
result = p(text)
```

训练：

<https://huggingface.co/transformers/training.html>

用自有数据集进行 fine-tuning：

[https://huggingface.co/transformers/custom\\_datasets.html](https://huggingface.co/transformers/custom_datasets.html)

## (2) 类

Transformers 提供了三个主要的组件：

- **Configuration** 配置类。存储模型和分词器的参数，诸如词表大小，隐层维数，dropout rate 等。配置类对深度学习框架是透明的。
- **Tokenizer** 分词器类。每个模型都有对应的分词器，存储 token 到 index 的映射，负责每个模型特定的序列编码解码流程，比如 BPE(Byte Pair Encoding)，SentencePiece 等等。也可以方便地添加特殊 token 或者调整词表大小，如 CLS、SEP 等等。
- **Model** 模型类。提供一个基类，实现模型的计算图和编码过程，实现前向传播过程，通过一系列 self-attention 层直到最后一个隐藏状态层。在最后一层基础上，根据不同的应用会再做些封装，比如 XXXForSequenceClassification，XXXForMaskedLM 这些派生类。

Transformers 的作者们还为以上组件提供了一系列 Auto Classes，能够从一个短的别名（如 bert-base-cased）里自动推测出来应该实例化哪种配置类、分词器类和模型类。

Transformers 提供两大类的模型架构，一类用于语言生成 NLG 任务，比如 GPT、GPT-2、Transformer-XL、XLNet 和 XLM，另一类主要用于语言理解任务，如 Bert、DistilBert、RoBERTa、XLM。

- AutoTokenizer：根据模型名自动找到的对应的 tokenizer
- AutoModel：根据模型名自动返回对应的模型
- Trainer：训练器类

[https://huggingface.co/transformers/main\\_classes/trainer.html](https://huggingface.co/transformers/main_classes/trainer.html)

## 11 · HanLP

HanLP 目前由青岛自然语义公司维护，提供了 RESTful 和 Python 本地接口。HanLP 分成 1.x 和 2.x 两个版本，其中 1.x 主要是机器学习算法实现，2.x 则使用了深度学习。

官方网站：

<https://hanlp.hankcs.com/>

官方文档：

<https://hanlp.hankcs.com/docs/index.html>

github：

<https://github.com/hankcs/HanLP>

## 12 · AllenNLP

基于 Pytorch 的开源项目

官方网站：

<https://allennlp.org/>

github：

<https://github.com/allenai/allennlp>

PaddleHub

## (六) NN：词的表征

NLP 的基础是词，所有 NLP 任务的第一步，是将输入的文本拆分成 Token（词），这个步骤被称为 Tokenize（分词），分词这步比较简单，可以是一些很粗糙的逻辑（比如英文中用空格和标点分词，当然实际情况要更复杂）。

分词之后的问题就是词的表征（word representation），就是如何表示词好输入给下一步的模型。NLP 有不少经典论文都是关于词向量的表示。

词通常是用向量来表示，最简单就是 One-hot 形式，此外还有词袋模型、n-gram、TF-IDF、Word2Vec、动态表示（BERT）等等。

论文：

[http://www.iro.umontreal.ca/~vincentp/Publications/1m\\_jmlr.pdf](http://www.iro.umontreal.ca/~vincentp/Publications/1m_jmlr.pdf)

参考：

语言模型：从 n 元模型到 NNLM

<https://zhuanlan.zhihu.com/p/43453548>

史上最全词向量讲解（LSA/word2vec/Glove/FastText/ELMo/BERT）

<https://zhuanlan.zhihu.com/p/75391062>

## 1 · NNLM (2003)

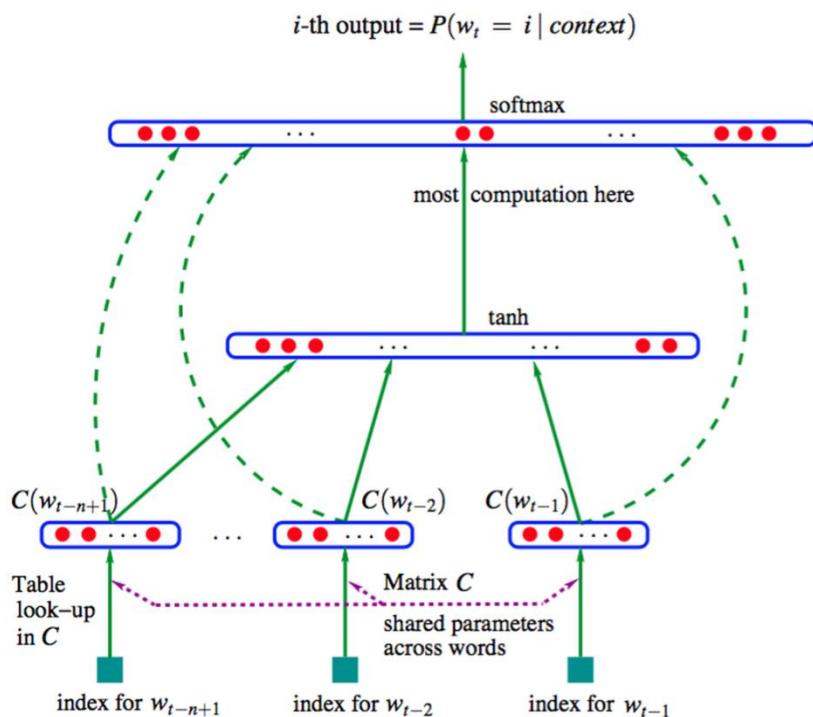
A Neural Probabilistic Language Model，深度学习三巨头之一的 Yoshua Bengio 于 2003 年发表的论文，也是第一篇神经网络的语言模型的论文，在得到语言模型的同时，也产出了副产品——词向量。

参考《[研究方向:NLP > 概念 > 语言模型](#)》

NNLM 基于 n-grams，输入为（之前的）n-1 个词

文中的语言模型如下，包括三层（或者可以将后两层理解为一层）：

- 第一层是输入层，将输入的 n-1 个词映射为向量，并首尾拼接得到新的向量
- 第二层是隐藏层，将第一层的输出向量进行矩阵运算，激活函数是 tanh
- 第三层是输出层，是个 softmax 多分类器



而第一层中的向量即为副产品——词向量。

论文：

[http://www.iro.umontreal.ca/~vincentp/Publications/1m\\_jmlr.pdf](http://www.iro.umontreal.ca/~vincentp/Publications/1m_jmlr.pdf)

参考：

<https://zhuanlan.zhihu.com/p/21240807>

【语言模型】NNLM(神经网络语言模型)

## 2 · Word2Vec (201301)

词嵌入 (Word Embeddings, 或者叫词向量, Word Vectors) 是用于在低维空间 (通常是几十到几百) 表示原本在高维空间 (词的 one-hot 表示, 词的词向量稀疏表示法)。

最早用于表示一个词用的是 one-hot 标签的向量, 即每个向量维度等同于词库中词的数量, 向量中该词对应的元素为 1, 其余为 0。后来逐渐从这种原始的词向量稀疏表示法过渡到现在的低维空间中的密集表示。用稀疏表示法在解决实际问题时经常会遇到维数灾难, 并且语义信息无法表示, 无法揭示 word 之间的潜在联系。而采用低维空间表示法, 不但解决了维数灾难问题, 并且挖掘了 word 之间的关联属性, 从而提高了向量语义上的准确度。

在多年对各种 NLP 任务的研究中发现, 词向量的生成作为很多 NLP 任务的第一步, 可以被独立出来, 在一个模型中被单独学习生成, 并且被应用到不同的 NLP 任务中。这篇论文就将注意力集中在如何通过简单的模型来学习词向量。

在 NLP 中, 把  $x$  看做一个句子里的一个词语,  $y$  是这个词语的上下文词语, 那么这里的  $f$ , 便是 NLP 中经常出现的『语言模型』 (language model), 这个模型的目的, 就是判断  $(x, y)$  这个样本, 是否符合自然语言的法则, 更通俗点说就是: 词语  $x$  和词语  $y$  放在一起, 是不是人话。

Word2vec 正是来源于这个思想, 但它的最终目的, 不是要把  $f$  训练得多么完美, 而是只关心模型训练完后的副产物——模型参数 (这里特指神经网络的权重), 并将这些参数, 作为输入  $x$  的某种向量化的表示, 这个向量便叫做——词向量

模型训练的复杂度为  $O = E \cdot T \cdot Q$ , 其中  $O$  为复杂度,  $E$  为 epoch 数量,  $T$  为训练集中的单词数量,  $Q$  则是每个模型所定义的。

文中提出了两种模型, 第一种称为 CBOW (Continuous Bag-of-Word Model), 连续词袋模型。第二种称为 Continuous Skip-gram 模型。

- 如果是用一个词语作为输入, 来预测它周围的上下文, 那这个模型叫做『Skip-gram 模型』
- 而如果是拿一个词语的上下文作为输入, 来预测这个词语本身, 则是『CBOW 模型』

### CBOW 模型

基于语料库, CBOW 的任务是通过中心词周围的词来预测中心词 (及其概率), 比如对于“今天下午我去钓鱼”这句话, 如果设“我”是中心词, 则通过今天、下午、去、钓鱼来推测出中心词。之所以叫做 Bag, 表明不考虑输入的这些中心词周围词的顺序。

而 Skip-gram 的任务正好相反, 输入是某个中心词, 而输出是各个周围词 (及其概率)。

两种模型的结构图如下，左侧为 CBOW，右侧为 Skip-gram，输入和输出的  $w$  均为词向量（在实际的代码中每个词有两个词向量，分别作为中心词和周围词时使用）：

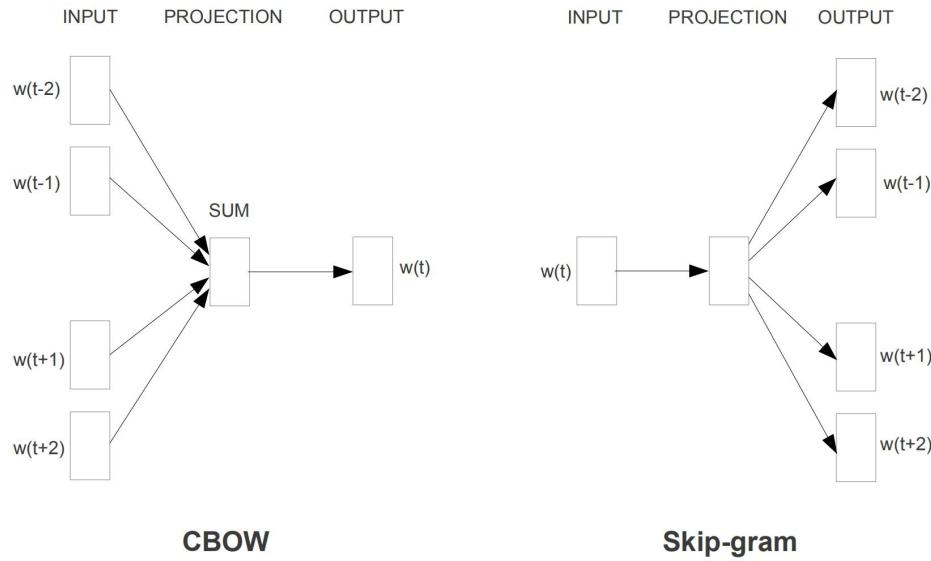
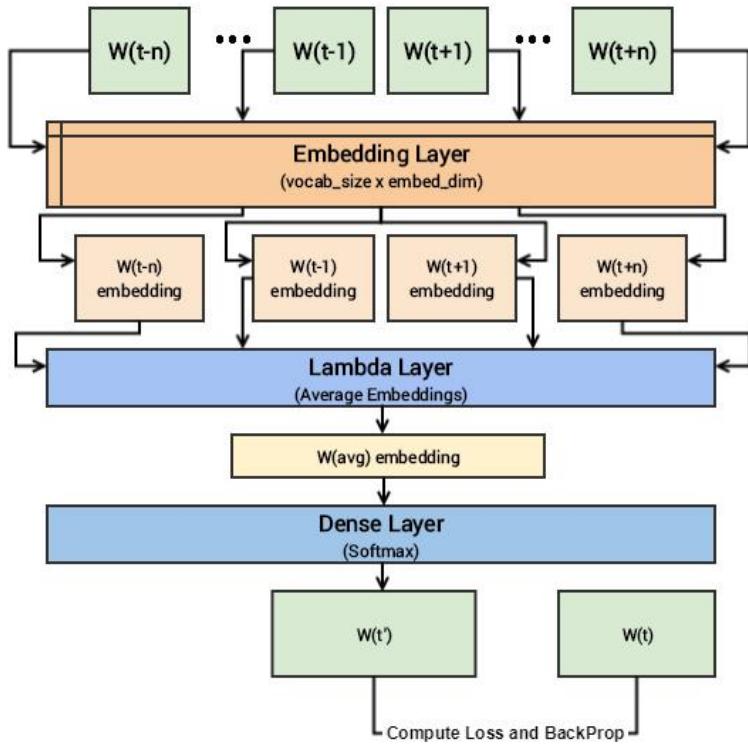


Figure 1: New model architectures. The CBOW architecture predicts the current word based on the context, and the Skip-gram predicts surrounding words given the current word.

以 CBOW 为例解释一下模型是如何生成词嵌入的。图左为 CBOW，可以看到输出为中心词  $w(t)$ ，而输入为  $w(t)$  之前的词  $w(t-1)$  和  $w(t-2)$ ，以及之后的词  $w(t+1)$  和  $w(t+2)$ 。比如“今天早上我吃的玉米”， $w(t)$  是“我”，则  $w(t-2)$  是“今天”， $w(t-1)$  是“早上”， $w(t+1)$  是“吃的”， $w(t+2)$  是“玉米”。

1. 输入层： $w$  的格式就是词的 one-hot 向量，其维度为  $V$  ( $1 \times V$  矩阵，即 vocabulary，词库中词的数量，通常  $V$  很大，能到 10 万级别)。CBOW 的输入为  $N$  (图中  $N=4$ ) 个 one-hot 向量。
2. projection 层：中间的 projection 层有权重矩阵  $W$  (大小为  $V \times D$ )，projection 层生成  $D$  维向量， $D$  即为词嵌入的维度 (通常为几十到几千)。 $N$  个 one-hot 向量分别和  $W$  做点积，取生成的  $N$  个  $D$  维向量的平均值作为输出。由于 one-hot 向量的稀疏性，每个 one-hot 向量仅和  $W$  中的一行做运算，其计算复杂度为  $D$ ，projection 层总的计算复杂度为  $N \times D$ 。
3. 输出层：输出层有权重矩阵  $W'$  (大小为  $D \times V$ )，生成  $V$  维向量 (即对输出词的预测)，激活函数为 softmax，因此这  $V$  维向量的和为 1，与 ground truth 的 one-hot 做对比，误差越小越好。其计算复杂度为  $D \times \log_2 V$ ，(为什么不是  $D \times V$  下文解释)。因此 CBOW 模型总的计算复杂度  $Q$  为： $Q = N \times D + D \times \log_2 V$ 。

下图描述了整个模型：



训练结束之后，我们需要的并不是这个模型，而是其中的权重矩阵  $W$ ，每个词的 one-hot 向量和  $W$  点积之后得到的  $D$  维向量（即  $W$  中对应该词的那一行）即是该词的  $D$  维词嵌入。

但在不同的解读文章中，对 CBOW 模型（以及 Skip-gram 模型）有不同的解读，比如在 Amazon 的《动手学习深度学习》中：

[https://zh.d2l.ai/chapter\\_natural-language-processing/word2vec.html](https://zh.d2l.ai/chapter_natural-language-processing/word2vec.html)

在这些论述中，每个词都有两个词嵌入，分别对应其作为中心词和背景词时，如果对比上文中的模型，projection 层的权重矩阵（ $V \times D$  大小）的每一行（ $D$  维）可以被视为背景词嵌入。

而通过其中的计算方式可知，中心词嵌入无法对应输出层的权重矩阵，即：

- 上文中  $V$  维输出的计算方式是背景词嵌入的平均值（ $D$  维）与  $D \times V$  矩阵（输出层的权重）点积生成的。
- 而 d2l 文中的  $V$  维输出是背景词嵌入的平均值（ $D$  维）分别与所有（ $V$  个）中心词嵌入（ $D$  维）点积得出的  $V$  个标量。

在对 Word2Vec 论文质疑的文章中：

<https://zhuanlan.zhihu.com/p/68401048>

我们可以看到，也许这就是 word2vec 论文和代码的区别

### Skip-gram 模型

对于 Skip-gram 模型来说，整个过程与 CBOW 类似，区别仅在于，输入的中心词 one-hot 向量只有一个，而输出的周围词有  $C$  个，计算复杂度为： $Q = C \times (D + D \times \log_2 V)$

## Hierarchical Softmax

从输入层到 projection 层是个 V 维向量（one-hot 向量）到 D 维向量（N 个词嵌入的均值）的矩阵乘法，但考虑到 V 维向量是 one-hot 向量，其计算复杂度只有  $N*D$ ，而从 projection 层到输出层是个 D 维向量到 V 维向量的矩阵乘法，正常来讲其复杂性为  $D*V$ ，因为 V 很大，因此这部分消耗了绝大部分算力。

因此，利用霍夫曼树的 Hierarchical Softmax 作为激活函数（从而实际上形成了若干层隐藏层），使得通过若干次 ( $\log_2 V$  次) 二元判断可以得到最后的 V 维向量，从而把计算复杂度降低为  $D * \log_2 V$ 。

<https://www.cnblogs.com/pinard/p/7243513.html>

## Negative Sampling

负采样（Negative Sampling）是论文中用于降低计算量的另外一个手段。对于训练语言模型来说，softmax 层非常难算，由于要预测的是当前位置是哪个词，那么这个类别数就等同于词典规模，因此动辄几万几十万的类别数对算力有很大的消耗。

负采样的思想是，不直接让模型从整个词表找最可能的词了，而是直接给定这个词（即正例）和几个随机采样的噪声词（即采样出来的负例），只要模型能从这里面找出正确的词就认为完成目标。

参考：

<https://cloud.tencent.com/developer/news/84841>

<https://blog.csdn.net/bitcarmanlee/article/details/82291968>

秒懂词向量 Word2vec 的本质

<https://zhuanlan.zhihu.com/p/26306795>

论文：

Efficient Estimation of Word Representations in Vector Space

<https://arxiv.org/pdf/1301.3781v3.pdf>

代码：

官方代码：<https://code.google.com/archive/p/word2vec/>

网友搬运到 github：<https://github.com/dav/word2vec>

## 3 · GloVe(2014)

GloVe 这个名字来自 Global Vector，类似 Word2Vec，也是一个计算词向量的算法。

共现矩阵（Co-occurrence Probabilities Matrix）的元素  $X_{ij}$  的意义为，在整个语料库中，单词  $i$  和单词  $j$  共同出现在一个上下文窗口的次数（早期的一种词向量表征工具 LSA，就是基于共现矩阵的）。GloVe 是通过共现矩阵来计算词向量。

官方连接：

<https://nlp.stanford.edu/projects/glove/>

论文：

<https://nlp.stanford.edu/pubs/glove.pdf>

参考：

<https://zhuanlan.zhihu.com/p/60208480>

通俗易懂理解——Glove 算法原理

<https://zhuanlan.zhihu.com/p/42073620>

四步理解 Glove

<https://zhuanlan.zhihu.com/p/79573970>

GloVe 详解

<https://www.fanyeong.com/2018/02/19/glove-in-detail/>

## 4 · fastText(201607)

在 word2vec 中，我们并没有直接利用构词学中的信息。无论是在 skip-gram 模型还是连续词袋模型中，我们都将形态不同的单词用不同的向量来表示。例如，“dog”和“dogs”分别用两个不同的向量表示，而模型中并未直接表达这两个向量之间的关系。鉴于此，fastText 提出了子词嵌入（subword embedding）的方法，从而试图将构词信息引入 word2vec 中的 Skip-gram 模型。

在 fastText 中，每个中心词被表示成子词的集合。下面我们用单词“where”作为例子来了解子词是如何产生的。首先，我们在单词的首尾分别添加特殊字符“<”和“>”以区分作为前后缀的子词。然后，将单词当成一个由字符构成的序列来提取 n 元语法。例如，当 n=3 时，我们得到所有长度为 3 的子词：“<wh”、“whe”、“her”、“ere”、“re>”以及特殊子词“<where>”。

fastText 也用到了基于霍夫曼树的 Hierarchical Softmax。

论文：

<https://arxiv.org/pdf/1607.01759.pdf>

代码：

<https://github.com/facebookresearch/fastText>

参考：

<https://zhuanlan.zhihu.com/p/158043574>

<http://yzstr.com/2018/11/30/fasttext/>

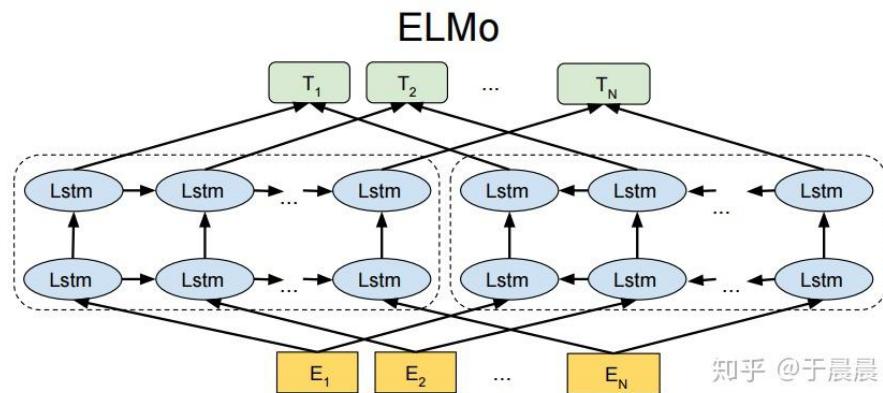
fastText 原理及实践

<https://zhuanlan.zhihu.com/p/32965521>

## 5 · ELMo(201802)

之前的词向量表示（Word2Vec，GloVe，fastText 等）都是固定的，对多义词的表示无能为力，ELMo 的工作是对于多义词提出了一个较好的解决方案。

ELMo 中的词-向量转换不再是一个对应关系，而是一个预训练好的模型，输入词，输出词向量。这个模型是个两个单向双层 LSTM，如下：



参考：

ELMo 原理解析及简单上手使用

<https://zhuanlan.zhihu.com/p/51679783>

词嵌入：ELMo 原理

<https://zhuanlan.zhihu.com/p/88993965>

论文：

Deep Contextualized word Representation

<https://arxiv.org/pdf/1802.05365.pdf>

## (七) NN：基于 RNN

### 1 · Encoder-Decoder (201406)

《Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation》的贡献有 2 个：

- 提出了 RNN Encoder-Decoder，一种 NvsM 类型 RNN 的实现。
- 提出了 GRU 微结构

Encoder-Decoder 由两个 RNN 组成，即通过一个 Nvs1 的 RNN (Encoder) 先将输入序列转换为一个向量，而另一个 1vsN 的 RNN (Decoder) 则将这个向量转换为一个输出序列。这种模型可以处理输入和输出序列不等长的情况，机器翻译等 NLP 任务基本上使用的都是此类模型。

RNN Encoder-Decoder 的结构如下：

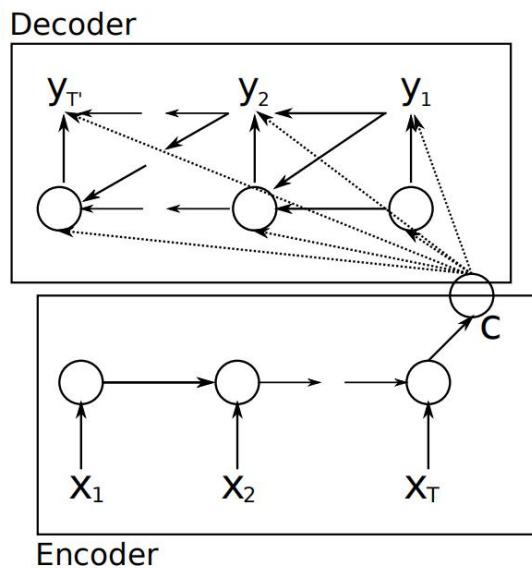


Figure 1: An illustration of the proposed RNN Encoder-Decoder.

其中下面的框是 encoder， $X$  是输入句子中的各个词，而上面的框是 decoder， $Y$  是输出句子的各个词。

对于 encoder 来说，每个步骤的隐藏状态表示为：

$$\mathbf{h}_{\langle t \rangle} = f(\mathbf{h}_{\langle t-1 \rangle}, x_t)$$

对于 decoder 来说，每个步骤的隐藏状态为：

$$\mathbf{h}_{\langle t \rangle} = f(\mathbf{h}_{\langle t-1 \rangle}, y_{t-1}, \mathbf{c})$$

每个输出 symbol（词）的概率分布为：

$$P(y_t | y_{t-1}, y_{t-2}, \dots, y_1, \mathbf{c}) = g(\mathbf{h}_{\langle t \rangle}, y_{t-1}, \mathbf{c})$$

Encoder 和 Decoder 这两个 RNN 会被联合训练，以便使得以下表达式尽量大：

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

其中  $\theta$  为模型参数， $(x_n, y_n)$  为训练集的输入和标签。

模型训练完成之后，可以通过 2 种方式使用：

- 给定一个输入序列，生成一个输出序列
- 给定一对输入序列-输出序列，得到该序列对的可能性得分

GRU 这里不做介绍，参考《[网络构成 > RNN 微结构 > GRU 单元](#)》

论文：

Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation  
<https://arxiv.org/pdf/1406.1078.pdf>

## 2 · Seq2Seq (201409)

Seq2Seq 有时候指代这篇 Google 的论文《Sequence to Sequence Learning with Neural Network》，但通常更广义的指代所有这种类型的模型（参看 RNN 中的 N vs M 类型的模型），与 Encoder-Decoder 同义。

Seq2Seq 通常会和 Encoder-Decoder 被视为同一概念，这两篇论文出现的时间也差不多，和 Encoder-Decoder 的区别在于：

- 使用 LSTM 而非 GRU 作为 Encoder 和 Decoder 的单元
- 使用多层（4 层）网络而非 1 层网络
- 倒序输入序列（句子），能大幅提高网络性能

论文：

Sequence to Sequence Learning with Neural Network

<https://arxiv.org/pdf/1409.3215.pdf>

### 3 · Attention 机制 (201409)

参考《[网络构成 > Attention 机制 > RNN 中的 Attention](#)》

参考：

<https://zhuanlan.zhihu.com/p/42724582>

[https://blog.csdn.net/qq\\_42189083/article/details/89326085](https://blog.csdn.net/qq_42189083/article/details/89326085)

## (八) NN：基于 Transformer

Attention is all you need 是 Google 在 2017 年发表的论文，文中提出了 Transformer 单元（Self-Attention 架构）。

Transformer 的出现，使得 NLP 模型的通用性大为增加，同时模型的大小及需要的算力也大增，在 Transformer 的基础上，出现了 GPT 系列和 BERT 系列两个系列的通用预训练 NLP 网络（这些网络的特点都是使用了 Transformer 的一部分，GPT 用的是去掉 Encoder-Decoder Attention 层的 Decoder Transformer）。

BERT 源自 Google，之后不同的团队也开发了各种变种（比如 ALBERT，RoBERTa 等）。这些 NLP 的预训练网络类似 CV 中不同任务中的模型的骨干网络（比如 ResNet、VGG、MobileNet 等），针对各种任务（主要是分类任务）所需要的附加结构简单，通常只是一些分类的 Dense + Softmax 层，在后期 Fine-tuning 的工作量也更小。

GPT 系列来自 OpenAI 团队，GPT 系列的思想和 BERT 不一样，GPT 是完全舍弃 Fine-Tuning，转而使用一个更大、更通用的预训练网络来完成各种不同的任务。因此 GPT 的网络也更大，比如 GPT-2 的参数量是 15 亿，GPT-3 的参数量更是达到了惊人的 1750 亿。

参考：

Transformer 结构及其应用详解--GPT、BERT、MT-DNN、GPT-2

<https://zhuanlan.zhihu.com/p/69290203>

推荐一个宝藏博主，让你搞懂 Transformer、BERT、GPT！

<https://zhuanlan.zhihu.com/p/137858220>

## 1 · Transformer(201706)

2017 年，Google 团队发布了关于 NLP 的著名论文《Attention is All You Need》，提出了 **Self-Attention** 的概念，并据此提出了 **Transformer** 架构，Transformer 架构也成为之后 NLP 模型的基础。之后随着发展，Transformer 在其他领域也有大量的应用。

在 Self-Attention 之前，NLP 的任务大都是 RNN Seq2Seq + Attention 的方式，传统的基于 RNN 的 Seq2Seq 模型难以处理长序列的句子，无法实现并行，并且面临对齐的问题。这类模型的发展基本有以下几个方向：

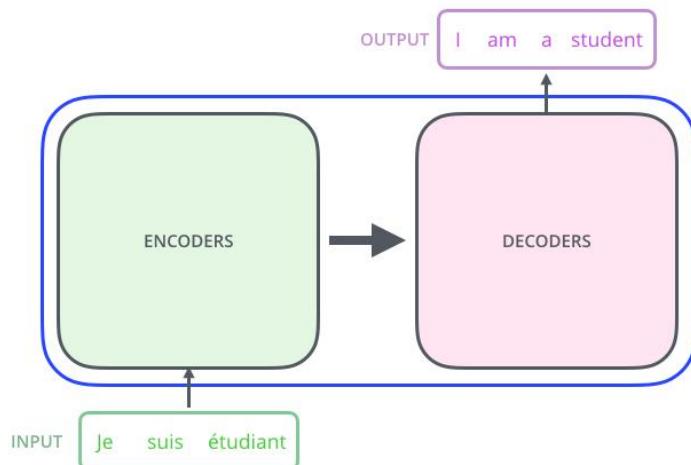
- 输入方向性：单向 -> 双向
- 深度：单层 -> 多层
- 单元类型：基础 RNN -> LSTM/GRU

再然后 CNN 由计算机视觉也被引入到 deep NLP 中，CNN 不能直接用于处理变长的序列样本但可以实现并行计算。完全基于 CNN 的 Seq2Seq 模型虽然可以并行实现，但非常占内存，很多的 trick，大数据量上参数调整不容易。

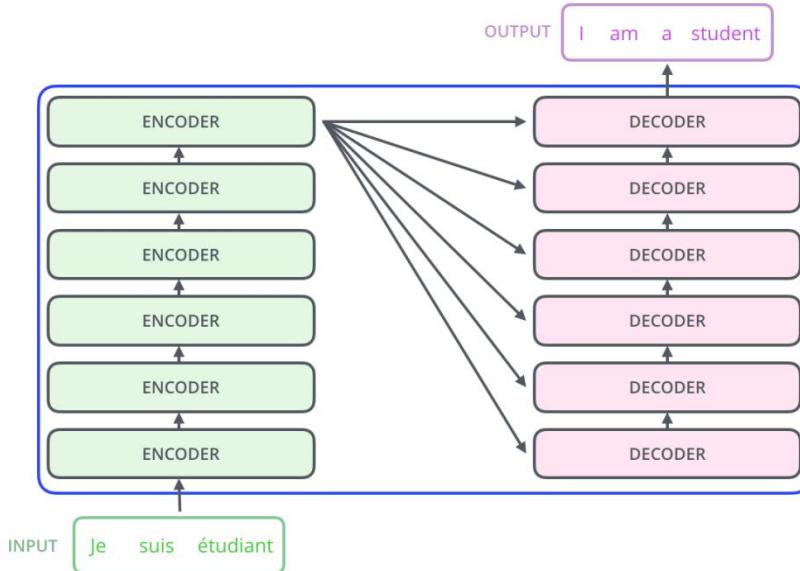
而 Self-Attention 的出现替代了 RNN 或者 CNN，只用 Attention 处理 NLP。这里的“Self”意思是 Query、Key、Value 皆从同一个 Source 中得出。

### (1) 网络结构

- 整个网络分成了 Encoder 部分和 Decoder 部分



- Encoder 部分由 6 层 Encoder Transformer 单元组成，6 层 Encoder Transformer 在结构上完全一样，但不共享权重。
- Decoder 则由 6 层 Decoder Transformer 单元组成，他们的结构也一样，不共享权重



其中每个 Encoder 层被称为一个 Encoder Transformer 单元，由两个子层组成：Self-Attention 层和 Dense 层。每个 Decoder 层被称为一个 Decoder Transformer 单元，由三个子层组成：Self-Attention 层、Encoder-Decoder Attention 层、Dense 层。

## (2) Self-Attention

Self Attention 就是句子中的某个词对于本身的所有词做一次 Attention。算出每个词对于这个词的权重，然后将这个词表示为所有词的加权和。每一次的 Self Attention 操作，就像是为每个词做了一次 Convolution 操作或 Aggregation 操作。

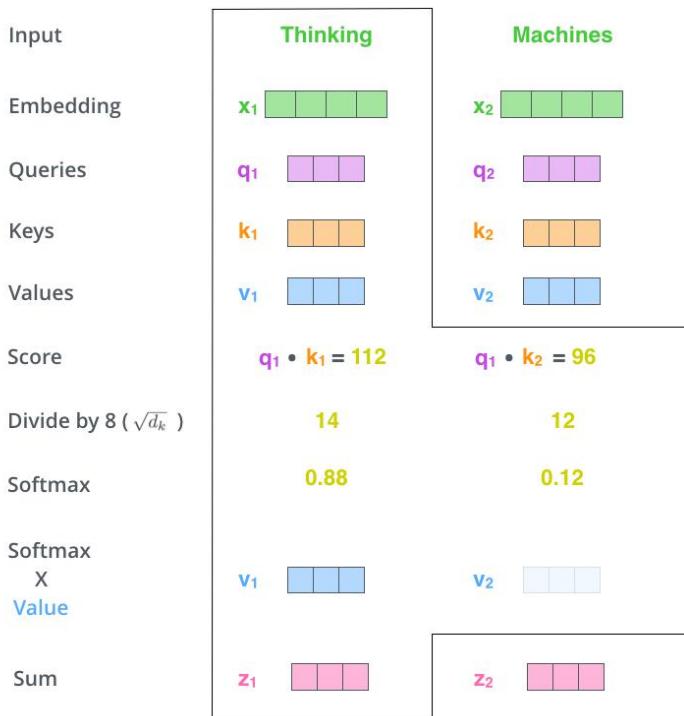
输入序列长度为 N（比如 512，长度不够的通过空白单词<pad>补足），输入文本的开始和结束用特殊单词<s>和<e>表示。

论文中的词向量维度为 512，query、key、value 三个向量维度为 64，具体流程如下：

- 首先每个词的词向量  $X_i$  ( $1 \times 512$ ) 都要分别点积三个矩阵  $W^q$  ( $512 \times 64$ )， $W^k$  ( $512 \times 64$ )， $W^v$  ( $512 \times 64$ )，生成每个词自己的 query ( $q_i$ ， $1 \times 64$ )，key ( $k_i$ ， $1 \times 64$ )，value ( $v_i$ ， $1 \times 64$ ) 三个向量
- 以序列中某个位置的词为中心进行 Self Attention 时，都是用该词的 key 向量 ( $k_i$ ， $1 \times 64$ ) 与每个词的 query 向量的转置 ( $q_i^T$ ， $64 \times 1$ ) 做点积，得到 N 个 score。
- 将该词的所有 score (N 个) 除以 8 (key 向量的维度 64 的平方根)，再通过 Softmax 归一化出权重 (N 个)，这些权重表示了序列中各个词对当前位置的影响（毫无疑问影响最大的是该词本身）。

- 然后通过这些权重 (N个) 算出所有词的 value ( $1 \times 64$ ) 的加权和，作为这个位置的输出  $Z_i$  ( $1 \times 64$ )。

流程如下：



而整个序列 X 的计算过程，则可以用矩阵运算来表示：

- 首先，输入序列 X 的词嵌入矩阵 ( $N \times 512$ ) 通过  $W^Q$ 、 $W^K$ 、 $W^V$  (皆为  $512 \times 64$ ) 分别转换为三个矩阵 Q ( $N \times 64$ )、K ( $N \times 64$ )、V ( $N \times 64$ )

$$\begin{array}{ccc}
 X & W^Q & Q \\
 \begin{matrix} \text{[4 green boxes]} \end{matrix} \times \begin{matrix} \text{[4x4 purple boxes]} \end{matrix} & = & \begin{matrix} \text{[4x4 purple boxes]} \end{matrix} \\
 X & W^K & K \\
 \begin{matrix} \text{[4 green boxes]} \end{matrix} \times \begin{matrix} \text{[4x4 orange boxes]} \end{matrix} & = & \begin{matrix} \text{[4x4 orange boxes]} \end{matrix} \\
 X & W^V & V \\
 \begin{matrix} \text{[4 green boxes]} \end{matrix} \times \begin{matrix} \text{[4x4 blue boxes]} \end{matrix} & = & \begin{matrix} \text{[4x4 blue boxes]} \end{matrix}
 \end{array}$$

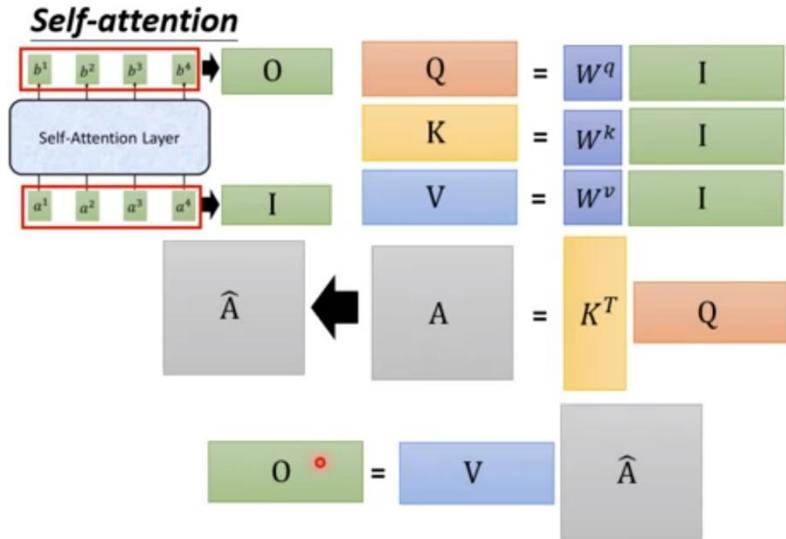
- 然后转置 K ( $N \times 64$ )，与 Q ( $N \times 64$ ) 做点积，生成表示序列中每个位置对每个位置的影响的 score 矩阵 ( $N \times N$ )，再做 softmax 运算得出每个输入的权重 ( $N \times N$ )，乘以 V ( $N \times 64$ )，得出表示各个位置的输出的矩阵 Z ( $N \times 64$ )

$$\begin{aligned}
 & \text{softmax} \left( \frac{Q \times K^T}{\sqrt{d_k}} \right) V \\
 & = \begin{matrix} \text{[4x4 pink boxes]} \end{matrix}
 \end{aligned}$$

公式为：

$$\begin{aligned} Q &= XW_Q \\ K &= XW_K \\ V &= XW_V \\ \text{Attention}(Q, K, V) &= \text{Softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V \end{aligned}$$

下图是台大李宏毅视频中的截图，同样表示了从输入矩阵 I (序列×嵌入) 到输出矩阵 O (序列×嵌入) 的整个过程：



### (3) Multi-head 结构

Transformer 论文中还提出了 Multi-head (多头结构)。

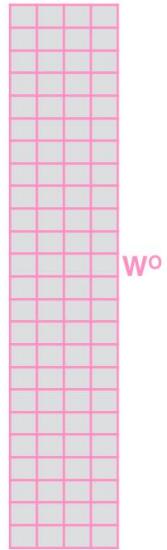
Multi-head 将数据分别输入到若干个 (8 个) 不同的 self-attention 结构中，得到 8 个加权的特征矩阵  $Z_i$  ( $N \times 64$ )，之后按列将它们拼成一个大的特征矩阵 ( $N \times (64 \times 8)$ )，经过一层全连接层，即图中的  $W^o$  权重矩阵 ( $512 \times 512$ ) 之后得到输出  $Z$  ( $N \times 512$ )。

1) Concatenate all the attention heads



2) Multiply with a weight matrix  $W^o$  that was trained jointly with the model

$\times$



3) The result would be the  $Z$  matrix that captures information from all the attention heads. We can send this forward to the FFNN

$$= \begin{matrix} Z \\ \begin{matrix} \text{---} \\ \text{---} \end{matrix} \end{matrix}$$

1) This is our input sentence\*

2) We embed each word\*

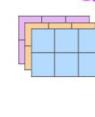
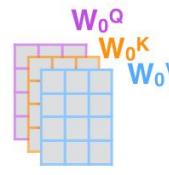
Thinking Machines



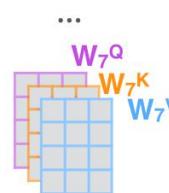
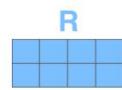
3) Split into 8 heads. We multiply  $X$  or  $R$  with weight matrices

4) Calculate attention using the resulting  $Q/K/V$  matrices

5) Concatenate the resulting  $Z$  matrices, then multiply with weight matrix  $W^o$  to produce the output of the layer



\* In all encoders other than #0, we don't need embedding. We start directly with the output of the encoder right below this one



#### (4) Position-encoding 位置编码

因为模型中不包括 Recurrent 和 CNN，因此输入是位置无关的，也就是说，无论输入数据的顺序如何（输入句子中词的顺序，或者输入图像中各个像素的位置等），结果是类似的。因此论文还提出了 Position Encoding (位置编码) 概念。

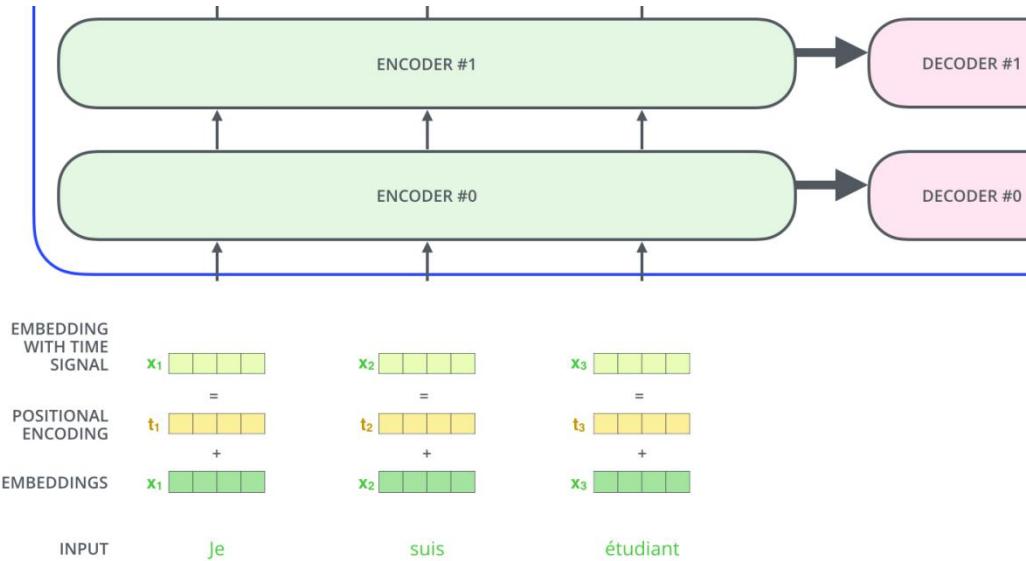
文中 Position Encoding 就是直接在输入的词向量 ( $1 \times 512$ ) 上加上一个 hard-coded 的 Position Encoding 向量 ( $1 \times 512$ )，从而输出一个带位置信息的嵌入 positional embedding ( $1 \times 512$ )。

位置编码并非学习来的，其公式为：

$$PE_{(pos,2i)} = \sin(pos/10000^{2i/d_{\text{model}}})$$

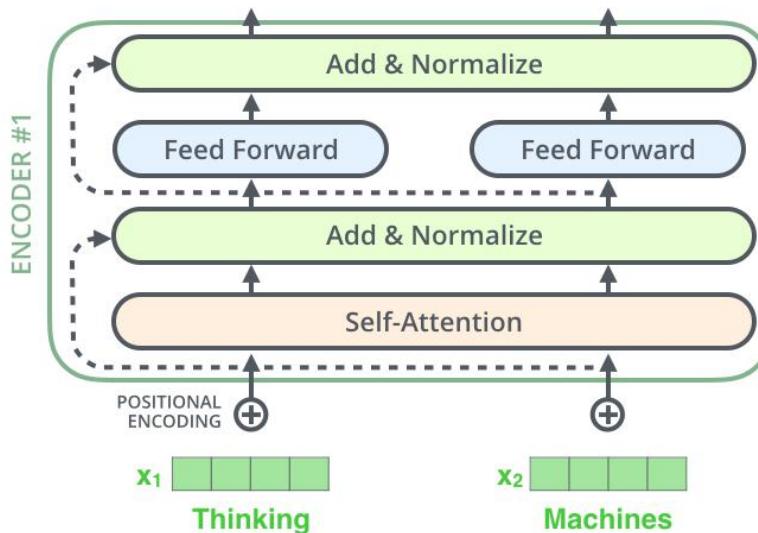
$$PE_{(pos,2i+1)} = \cos(pos/10000^{2i/d_{\text{model}}})$$

结构如下图：



## (5) Encoder Transformer 单元

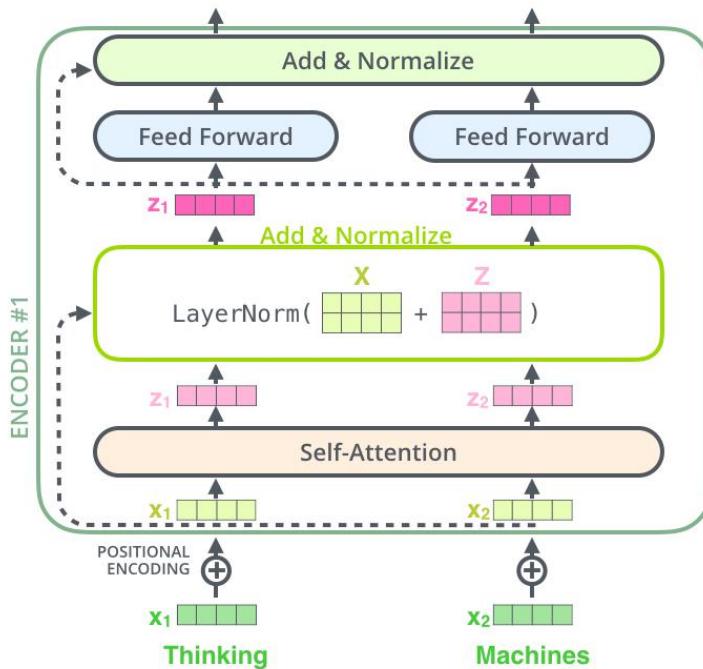
整个 Encoder 包括了 6 层这样的 Encoder Transformer：



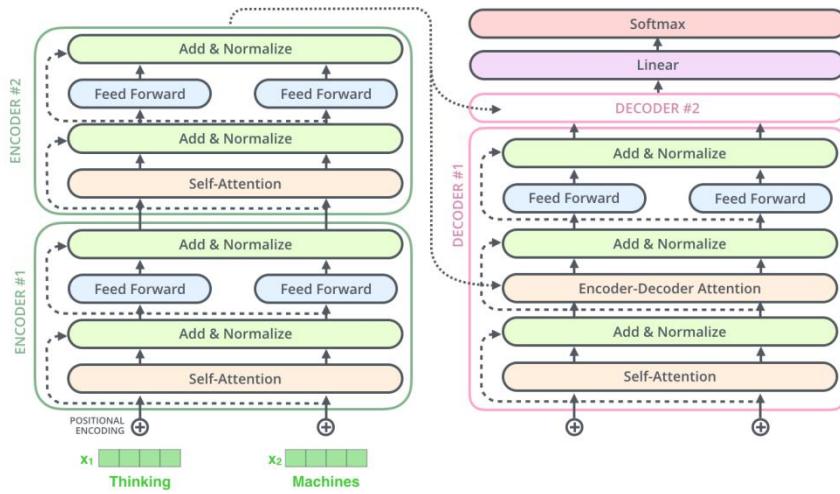
每个 Encoder Transformer 单元都有两个子层组成：

- Self-Attention 层
- 全连接层 (Feed Forward)

这两个子层都有残差 (Residual) 结构，即图中的 Add&Norm，具体是将该子层的输入和输出相加 (Add)，并做归一化 (Normalize)：



## (6) Decoder Transformer



Decoder Transformer 类似 Encoder Transformer，区别在于：

- Self-Attention 和 Dense 层之间多了一层 Encoder-Decoder Attention 层  
Encoder 最上面一层的输出转化为了 K 和 V，被作为 Decoder 中的 Encoder-Decoder Attention 层的输入的一部分，来帮助 Decoder 聚焦在输入序列的正确位置。该层的 Q 用的是下面的 Self-Attention 层输出的 Q，而 K 和 V 则来自于 Encoder
- Decoder 的 Self-Attention 层的输入只允许先于当前位置的词，其后的位置都被 mask  
Masked Attention 层相较于 Self-Attention 层（Encoder Transformer 中）的区别在于，在进入 Softmax 之前，与 (&) 一个掩码矩阵

	I	have	a	dream
I	■			
have		■		
a			■	
dream				■

公式如下：

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

## (7) 最后的 Linear 层和 Softmax 层

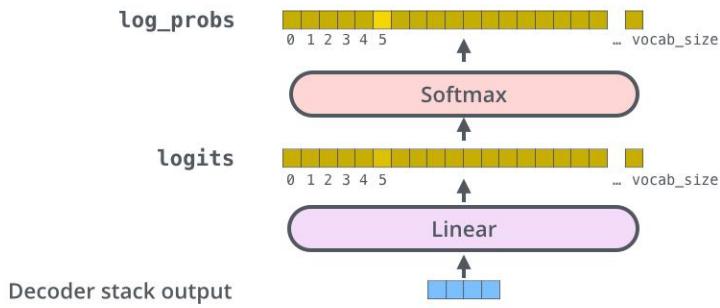
Decoder 的输出是一个向量，这个向量会经过一个 Linear 层（全连接层），该层的输出维度很大，等同于词库中词汇的数量，再经过一个 Softmax 层，得出对每个词的预测的结果概率。

Which word in our vocabulary  
is associated with this index?

am

Get the index of the cell  
with the highest value  
(`argmax`)

5



## (8) 整体结构

整体结构如下：

- Encoder
  - Encoder Transformer × 6
    - ◆ Self-Attention 层 (+Add&Norm)
    - ◆ 全连接层 (+Add&Norm)
- Decoder
  - Decoder Transformer × 6
    - ◆ Masked Self-Attention 层 (+Add&Norm)
    - ◆ Encoder-Decoder Attention 层
    - ◆ 全连接层 (+Add&Norm)
  - Linear 层 + Softmax

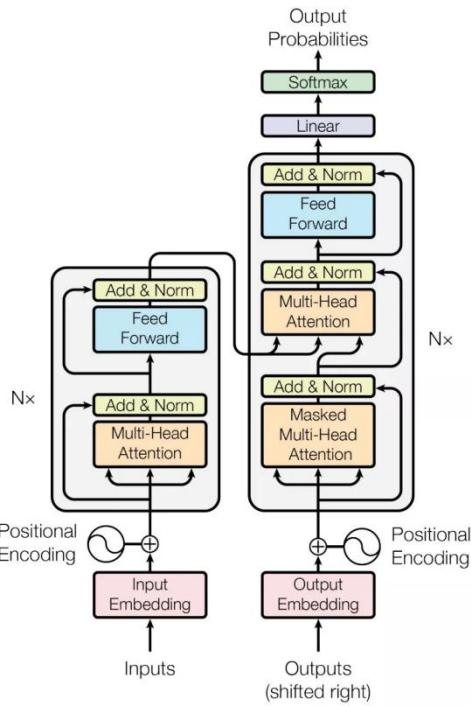


Figure 1: The Transformer - model architecture.

论文：

Attention is All You Need

<https://arxiv.org/pdf/1706.03762.pdf>

参考：

The Illustrated Transformer :

<http://jalammar.github.io/illustrated-transformer/>

Transformer 结构及其应用详解--GPT、BERT、MT-DNN、GPT-2

<https://zhuanlan.zhihu.com/p/69290203>

详解 Transformer (Attention is all you need)

<https://zhuanlan.zhihu.com/p/48508221>

李宏毅视频：<https://www.bilibili.com/video/BV15b411g7Wd?p=92>

<https://mp.weixin.qq.com/s/RLxWevVWHXgX-UcoxDS70w>

Weighted Transformer Network for Machine Translation

<https://arxiv.org/pdf/1711.02132.pdf>

代码：

<https://github.com/Kyubyong/transformer>

<https://github.com/JayParks/transformer>

## 2 · GPT 系列

GPT 系列是 OpenAI 团队（一个非盈利的 AI 组织）开发的，基于 Transformer 的 NLP 预训练模型。除了模型尺度和训练数据增加之外，GPT 三代之间的结构基本没有什么变化。

GPT 和 BERT 是当前两个最主流的 NLP 模型类型，其共同点是都基于 Transformer，而不同点包括：

GPT 使用 Decoder Transformer 单元（少一层），BERT 使用 Encoder Transformer 单元  
基于上面的原因，GPT 是个生成模型，而 BERT 则不是

参考：

<https://zhuanlan.zhihu.com/p/228857593>

### (1) GPT (201806)

GPT 采用单向 Transformer 单元，就是去掉 Encoder-Decoder Attention 层的 Decoder Transformer（或者也可以被认为是将 Encoder Transformer 里的 Self-Attention 层替换为 Decoder Transformer 中的 Masked Self-Attention 层），使得模型只能看得见上文的词。

而训练的过程其实非常的简单，就是将句子 n 个词的词向量（第一个为特殊单词<s>，表示 start）加上 Positional Encoding 后输入到前面提到的 Transfomer 中，n 个输出分别预测该位置的下一个词（<s>预测句子中的第一个词，最后一个词的预测结果不用于语言模型的训练）。

由于使用了 Masked Self-Attention，所以每个位置的词都不会“看见”后面的词，也就是预测的时候是看不见“答案”的，保证了模型的合理性，而因此 GPT 模型每次都是吐出一个单词，而这个单词会被加入下一轮的输入。

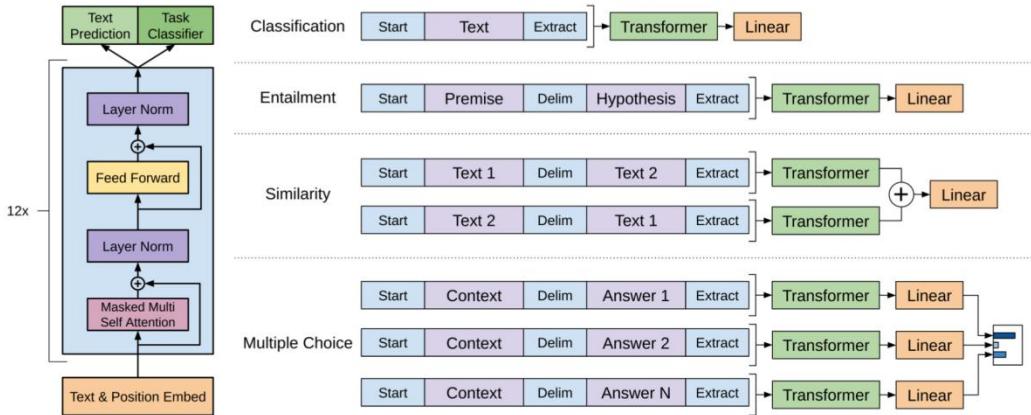


Figure 1: (**left**) Transformer architecture and training objectives used in this work. (**right**) Input transformations for fine-tuning on different tasks. We convert all structured inputs into token sequences to be processed by our pre-trained model, followed by a linear+softmax layer.

论文：

[https://s3-us-west-2.amazonaws.com/openai-assets/research-covers/language-unsupervised/language\\_understanding\\_paper.pdf](https://s3-us-west-2.amazonaws.com/openai-assets/research-covers/language-unsupervised/language_understanding_paper.pdf)

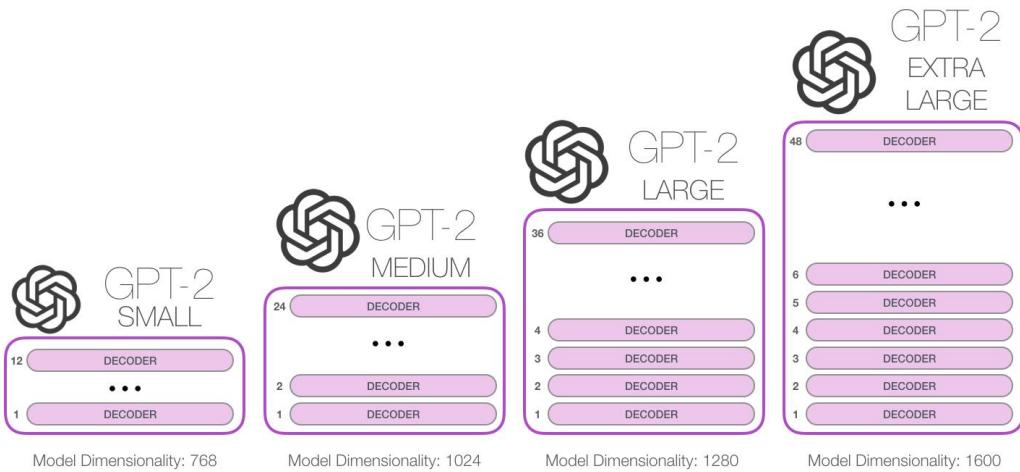
## (2) GPT-2 (201902)

GPT-2 继续沿用了原来 GPT 中使用的单向 **Transformer** 单元来建立模型，而这篇文章的目的就是尽可能利用单向 **Transformer** 的优势，做一些 BERT 使用的双向 Transformer 所做不到的事。那就是通过上文生成下文文本。

GPT-2 和 GPT 的结构基本没有区别，主要就是更大的结构和更多的训练数据：

GPT-2 的想法就是完全舍弃 Fine-Tuning 过程，转而使用一个容量更大、无监督训练、更加通用的语言模型来完成各种各样的任务。我们完全不需要去定义这个模型应该做什么任务，因为很多标签所蕴含的信息，就存在于语料当中。

GPT-2 中使用了不同大小的网络（维度和深度），例如其中 GPT-2 small 堆叠了 12 层单向 Transformer 单元，词向量的维度为 768：



而最大的 GPT-2 Extra Large 的深度是 48 层，使用的词向量宽度为 1600。

训练后的 GPT-2 包含两个权值矩阵：词向量矩阵和位置编码矩阵（参考 Transformer）

## 论文

<https://d4mucfpksywv.cloudfront.net/better-language-models/language-models.pdf>

## 参考

The Illustrated GPT-2

<http://jalammar.github.io/illustrated-gpt2/>

完全图解 GPT-2：看完这篇就够了（一）

<https://zhuanlan.zhihu.com/p/79714797>

## 代码

<https://github.com/openai/gpt-2>

## (3) GPT-3 (202005)

2020 年，OpenAI 推出了参数量高达 1750 亿的 GPT-3。其结构跟之前的 GPT-2 一样，依然是单向 Transformer 单元的堆叠，主要的区别就还是大小。GPT-3 的层数达到了 96 层。

## 参考

最新最全 GPT-3 模型网络结构详细解析

<https://zhuanlan.zhihu.com/p/174782647>

How GPT3 works - Visualizations and Animations

<http://jalammar.github.io/how-gpt3-works-visualizations-animations/>

## 论文

<https://arxiv.org/pdf/2005.14165.pdf>

## github

<https://github.com/openai/gpt-3>

## 3 · BERT 系列

Transformer 出来之后，成为了在 NLP 领域替代 LSTM 的一大杀器，而原始 Transformer（即《Attention is all you need》中提到的 Encoder-Decoder 结构）作为机器翻译这类 NLG 任务的模型很合适，那么该如何处理 NLU 这种分类任务呢，Google 提出了 **BERT** 作为解决方法。

**BERT** (**Bidirectional Encoder Representations from Transformers**)，是 Google 团队推出的基于 Transformer 的 NLP 模型，在 BERT 的基础上，不同的团队推出了各种 NLP 模型。

BERT 系列的模型是预训练模型，对其的使用类似图像领域中的卷积神经网络（比如 VGGNet、ResNet 等），使用者需要进行几步相对简单且不费算力的操作：

- 下载预训练模型（官方或者第三方已经在大语料库上进行过无监督的预训练）
- 添加全连接层（向量->类别）+ Softmax
- 通过对应的语料进行有监督的 fine-tuning

### (1) BERT(201810)

2018 年 10 月，Google 发布的论文及预训练模型 BERT，成功的在 11 项 NLP 任务中取得 SOTA 的成绩。

BERT 基于 Transformer，从此开始了基于 Transformer 的预训练模型（BERT 系列和 GPT 系列）在 NLP 任务上独占鳌头的局面。BERT 的特点从其论文的名字上可以看出来（**BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding**）：

- Deep：网络深度比原始的 transformer 深
- Bidirectional Transformers：基于双向 Encoder Transformer，这是和使用单向

Transformer 的 GPT 的区别，并且没有使用到原始 Transformer 中的 Decoder 部分

- Pre-training : BERT 是大量数据无监督预训练出的模型，之后不同下游任务根据需要，用 supervised 数据进行对应的训练 (fine-tuning) 7

BERT 和 GPT 以及 ELMo 的区别如下：

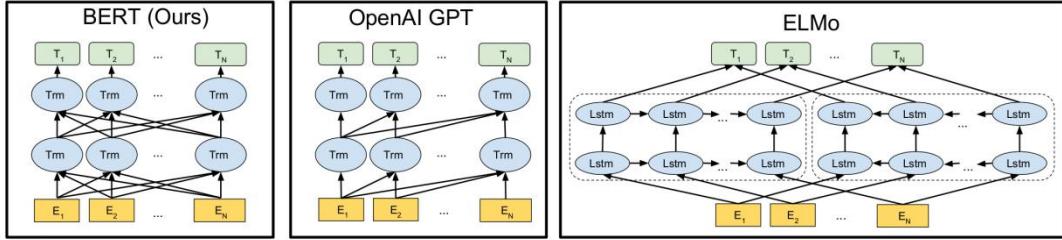


Figure 3: Differences in pre-training model architectures. BERT uses a bidirectional Transformer. OpenAI GPT uses a left-to-right Transformer. ELMo uses the concatenation of independently trained left-to-right and right-to-left LSTMs to generate features for downstream tasks. Among the three, only BERT representations are jointly conditioned on both left and right context in all layers. In addition to the architecture differences, BERT and OpenAI GPT are fine-tuning approaches, while ELMo is a feature-based approach.

可以看到：

- GPT 使用的是单向 Transformer Encoder
- 而 ELMo 则是将两个单向的 LSTM 的输出做拼接
- BERT 则是直接使用双向 Transformer

BERT 吸取了之前的几篇论文的经验：

- Transformer : Tranformer 单元
- ELMo : 双向、动态词嵌入
- ULM-FiT : fine-tuning 的流程
- GPT : 采用部分 Transformer (BERT 用的是 Encoder)

相较于原始的 Transformer，BERT 有如下改动：

## 网络结构

BERT 用的是双向 Transformer 单元的堆叠，这里的双向 Transformer 就是 Transformer 中的 Encoder Transformer 单元，所谓“双向”是指用的是用的是 Self-Attention 层，而非类似 GPT 所用的单向 Transformer 单元中的 Masked Self-Attention 层（只能看到左边的信息）。

BERT 分为两个大小的版本，Base 版和 Large 版本，L (模型层数) 、H (向量维度) 和 A (多头结构的 head 数量) 分别为：

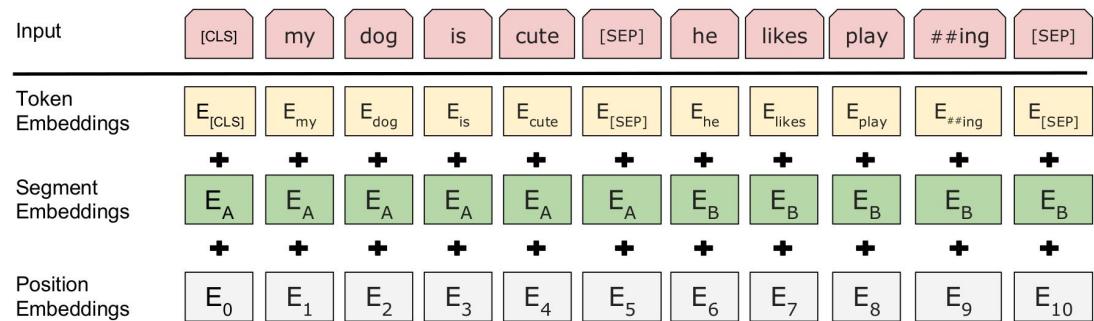
- Base 版本：L=12，H=768，A=12，参数总量是 110M
- Large 版本：L=24，H=1024，A=16，参数数量为 340M
- 原始 Transformer：L=6，H=512，A=8

## 输入向量和输出向量

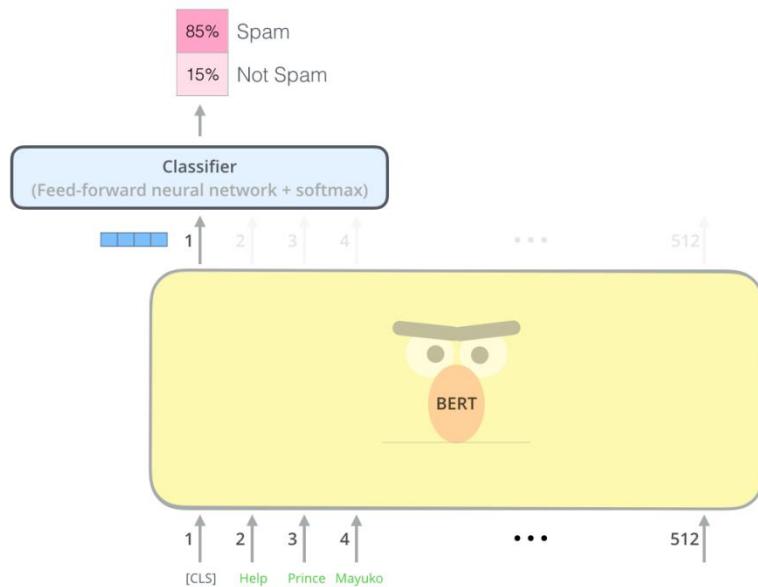
BERT 的输入向量是三个向量的和，比原始 Transformer 多了一个分割嵌入：

- **WordPiece 嵌入**：是指将单词划分成一组有限的公共子词单元，能在单词的有效性和字符的灵活性之间取得一个折中的平衡。例如图中 ‘playing’ 被拆分成了 ‘play’ 和 ‘ing’ 。
- **位置嵌入 (Position Embedding)**：对词位置的标示，参考 Transformer
- **分割嵌入 (Segment Embedding)**：用于区分两个句子，例如 B 是否是 A 的下文，对于句子对，第一个句子的特征值为 0，第二个的特征值为 1

下图中的每个块都是一个向量 (base 版的维度为 768)，红色为输入的词向量，由 WordPiece 嵌入 (黄色)、分割嵌入 (绿色)、位置嵌入 (灰色) 相加而来。



输入序列的第一个位置为**特殊单词[CLS]**，用于给之后的分类 (classification) 留出位置。第一个输出向量 (base 版维度也是 768) 用于分类，通过一个全连接层 (输入为该向量，输出为分类类别数) +Softmax 就可以得到很好的分类效果，比如下图中的区分垃圾邮件的二分类。

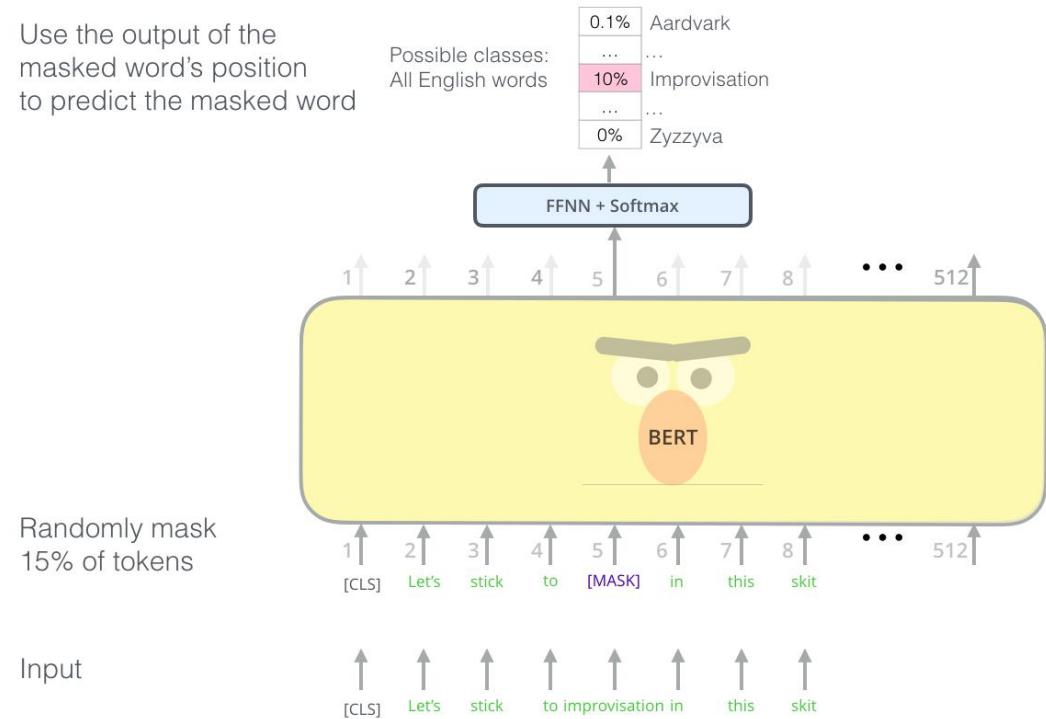


BERT 的预训练任务包括两个：MLM ( Masked Language Model ) 和 NLP ( Next Sentence Prediction )

## Masked Language Model

MLM 预训练任务，指的是在训练的时候随机从输入语料上 mask 掉一些 token，然后通过上下文预测该 token。注意，模型只需要预测该 token，而无须重建（计算）整个输入。

这个随机 mask 的比例是 15%，并且并不是每次都 mask 掉这些单词，而是有 80% 比例使用一个特殊单词 [mask] 替换该词，有 10% 的几率不替换，而剩下 10% 则随机替换为其它任意单词。

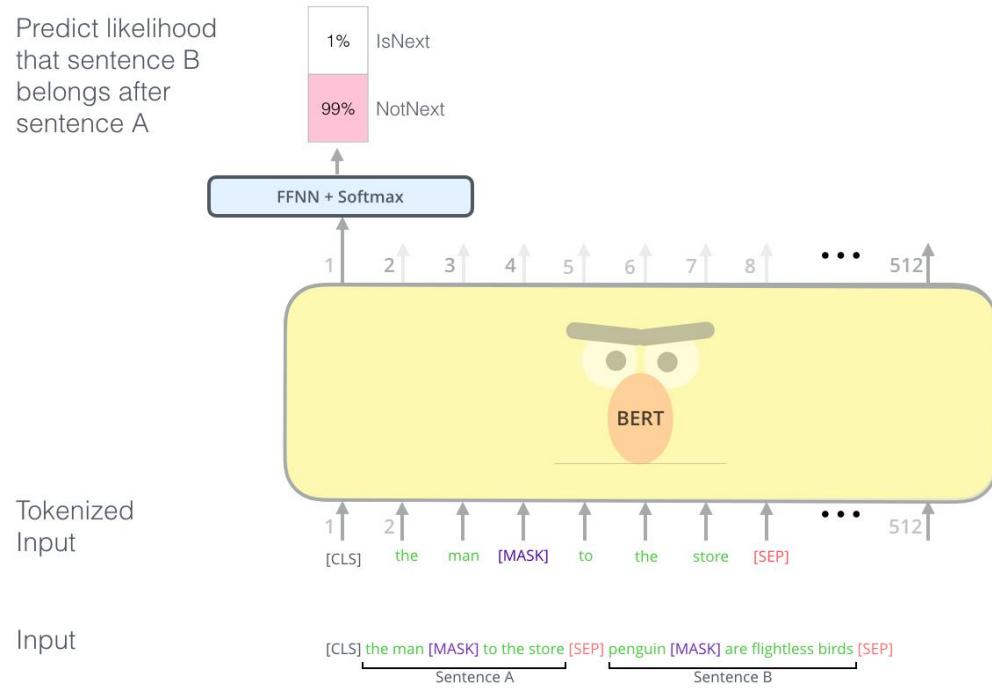


## Next Sentence Prediction

为了让 BERT 可以更好的把握两个句子之间的关系，比如问答判断（判断两句是否互为问答），蕴含关系（Entailment，判断两句是否互相蕴含），BERT 还使用了另一个无监督预训练任务 NSP (Next Sentence Prediction)。

NSP 任务是判断句子 B 是否是句子 A 的下文，如果是的话输出 “IsNext”，否则输出 “NotNext”。

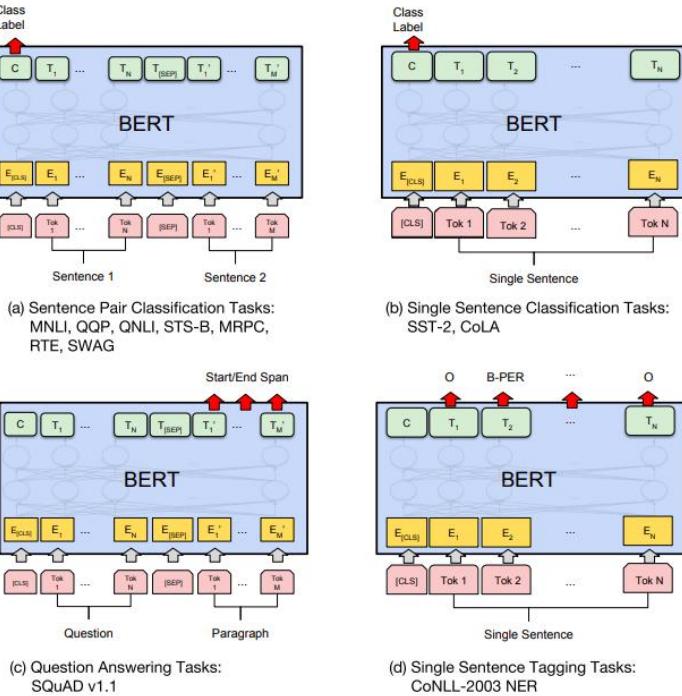
训练数据的生成方式是从平行语料中随机抽取的连续两句话，其中 50% 保留抽取的两句话，它们符合 IsNext 关系，另外 50% 的第二句话是随机从预料中提取的，它们的关系是 NotNext 的。这个关系保存在图中的第一个输出向量中（类似 MLM 任务中第一个输出向量表示分类）。



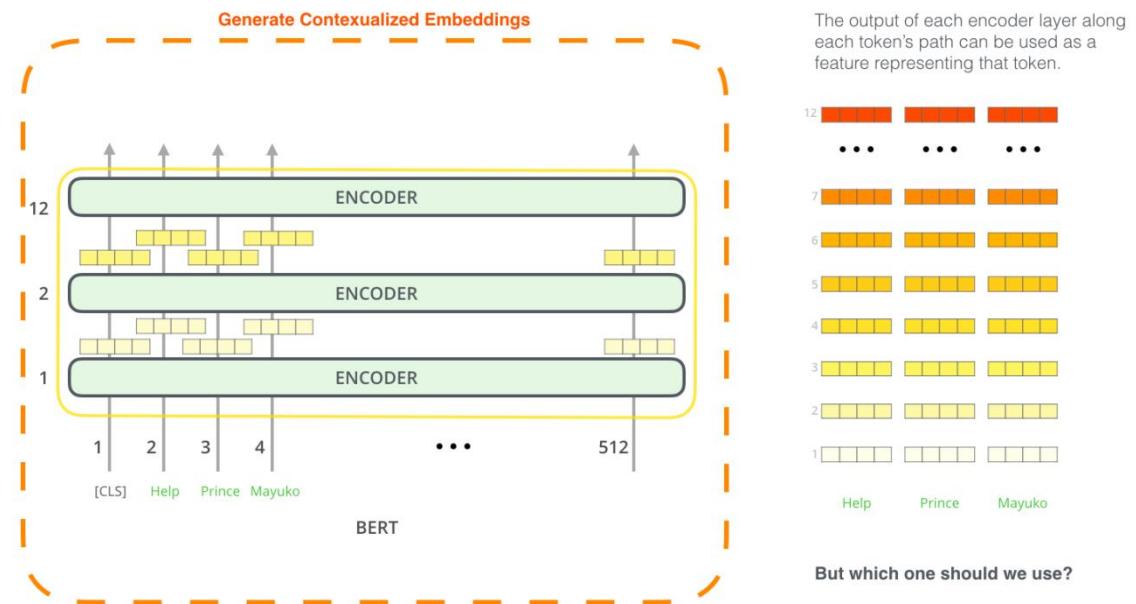
BERT 在具体任务的应用

对于各种不同类型的任务，BERT 需要添加不同的输出层，并加以 Fine-tuning，具体的模型如下图所示，其中：

- 左上为输入为句子对，输出为分类的任务，比如 MNLI、QQP、QNLI 等
  - 右上为输入为单个句子，输出为分类的任务，比如 SST-2、CoLA
  - 左下为输入为句子对，输出为句子的任务，比如 SQuAD v1.1
  - 右下为输入为单个句子，输出为多个标签的任务，比如 CoNLL-2003 NER



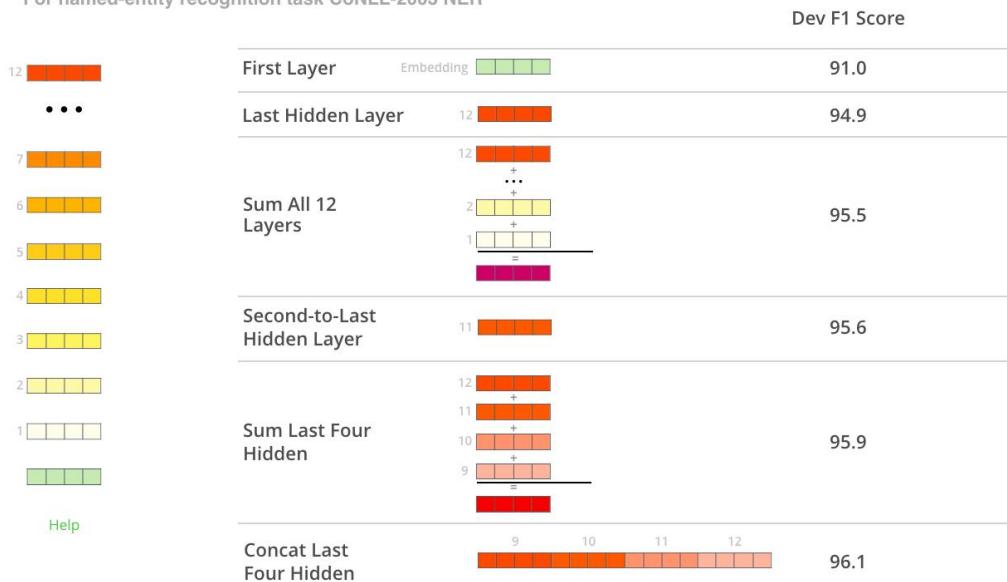
Fine-tuning 并不是唯一使用 BERT 的方式，就像 ELMo，也可以使用 BERT 来生成上下文相关的词向量，再将这些向量输入到自己的模型。



问题是该用哪一层的输出作为词向量最合适？BERT 论文中尝试了 6 种不同的选择：

- 第一层
- 最后一层
- 所有 12 层之和
- 倒数第二层
- 最后四层之和
- 最后四层的连接 (Concatenate)

What is the best contextualized embedding for “**Help**” in that context?  
For named-entity recognition task CoNLL-2003 NER



## 论文

BERT:Pre-training of Deep Bidirectional Transformers for Language Understanding  
<https://arxiv.org/pdf/1810.04805.pdf>

## github

Google 官方：

<https://github.com/google-research/bert>

哈工大中文版本：

<https://github.com/ymcui/Chinese-BERT-wwm>

## 参考

Illustrated BERT

<http://jalammar.github.io/illustrated-bert/>

A Visual Guide to Using BERT for the First Time

<http://jalammar.github.io/a-visual-guide-to-using-bert-for-the-first-time/>

词向量之 BERT

<https://zhuanlan.zhihu.com/p/48612853>

【NLP】Google BERT 模型原理详解

<https://zhuanlan.zhihu.com/p/46652512>

NLP 必读：十分钟读懂谷歌 BERT 模型  
<https://zhuanlan.zhihu.com/p/51413773>

## (2) ERNIE (201904)

论文：  
<https://arxiv.org/pdf/1904.09223.pdf>

## (3) XLNet (201906)

XLNet 试图融合自回归语言模型（Auto Regressive，比如 GPT）和自编码语言模型（Auto Encoder，比如 BERT）的优点。

论文：  
<https://arxiv.org/pdf/1906.08237.pdf>

参考：  
XLNet: 运行机制及和 Bert 的异同比较  
<https://zhuanlan.zhihu.com/p/70257427>

什么是 XLNet，它为什么比 BERT 效果好？  
<https://zhuanlan.zhihu.com/p/107350079>

github：  
官方：<https://github.com/zihangdai/xlnet>  
哈工大中文版本：<https://github.com/ymcui/Chinese-XLNet>

## (4) ERNIE2.0 (201907)

论文：  
<https://arxiv.org/pdf/1907.12412.pdf>

github：  
<https://github.com/PaddlePaddle/ERNIE>

参考：

<https://www.ramlinbird.com/2019/08/06/ernie%E5%8F%8Aernie-2-0%E8%AE%BA%E6%96%87%E7%AC%94%E8%AE%B0/>

## (5) RoBERTa (201907)

论文：

<https://arxiv.org/pdf/1907.11692.pdf>

github：

RoBERTa 中文版本：[https://github.com/brightmart/roberta\\_zh](https://github.com/brightmart/roberta_zh)

## (6) ALBERT (201909)

ALBERT 也是 Google 团队提出的，是 BERT 的一个变体。

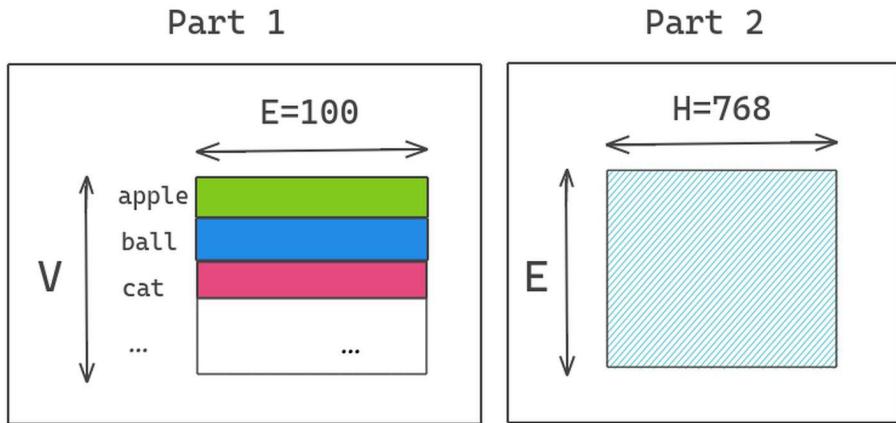
ALBERT 主要有以下几点贡献：

- Factorized embedding parameterization
- Cross-layer parameter Sharing
- Sentence-order prediction

### Factorized embedding parameterization

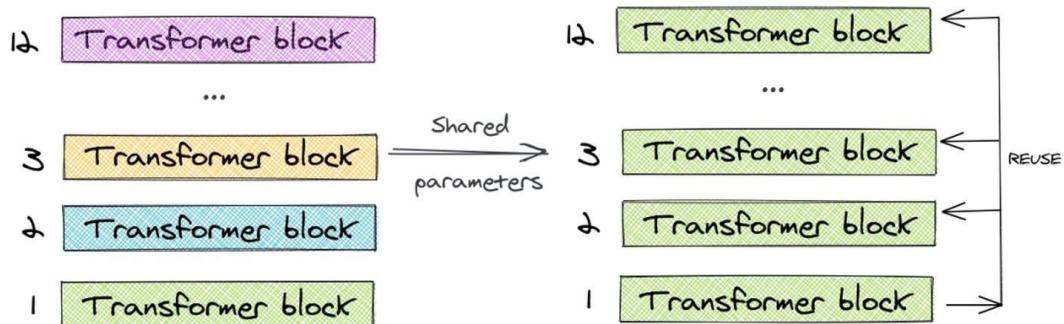
参数分解，在 BERT、XLNet、RoBERTa 等模型中，由于模型结构的限制，WordPiece embedding 的大小  $E$  总是与隐层大小  $H$  相同，即  $E = H$ 。从建模的角度考虑，词嵌入学习的是单词与上下文无关的表示，而隐层则是学习与上下文相关的表示。显然后者更加复杂，需要更多的参数，也就是说模型应当增大隐层大小  $H$ ，或者说满足  $H \gg E$ 。但实际上词汇表的大小  $V$  通常非常大，如果  $E = H$  的话，增加隐层大小  $H$  后将会使 embedding matrix 的维度  $V \times E$  非常巨大。

因此本文想要打破  $E$  与  $H$  之间的绑定关系，从而减小模型的参数量，同时提升模型表现。具体做法是将 embedding matrix 分解为两个大小分别为  $V \times E$  和  $E \times H$  矩阵，也就是说先将单词投影到一个低维的 embedding 空间  $E$ ，再将其投影到高维的隐藏空间  $H$ 。这使得 embedding matrix 的维度从  $O(V \times H)$  减小到  $O(V \times E + E \times H)$ 。当  $H \gg E$  时，参数量减少非常明显。在实现时，随机初始化  $V \times E$  和  $E \times H$  的矩阵，计算某个单词的表示需用一个单词的 one-hot 向量乘以  $V \times E$  维的矩阵（也就是 lookup），再用得到的结果乘  $E \times H$  维的矩阵即可。两个矩阵的参数通过模型学习。



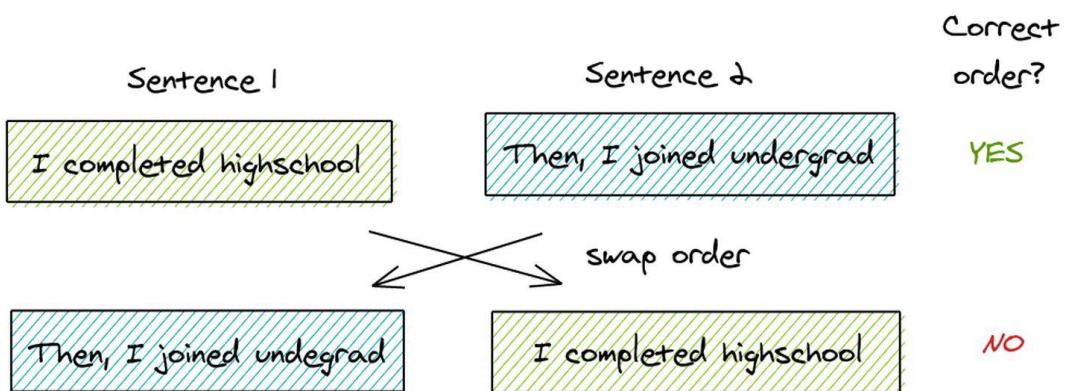
### Cross-Layer Parameter Sharing

跨层参数共享，本文提出的另一个减少参数量的方法就是层之间的参数共享，即多个层使用相同的参数。参数共享有三种方式：只共享 feed-forward network 的参数、只共享 attention 的参数、共享全部参数。ALBERT 默认是共享全部参数的。



### Sentence-order Prediction

在先前的研究中，已经证明 NSP 是并不是一个合适的预训练任务。因此本文提出了 SOP（句子顺序预测）任务来取代 NSP。其正例与 NSP 相同，都是两句连续的句子对，但负例是通过选择一篇文档中的两个连续的句子并将它们的顺序交换构造的。这样两个句子就会有相同的话题，模型学到的就更多是句子间的连贯性。



论文：

<https://arxiv.org/pdf/1909.11942.pdf>

参考：

【论文阅读】ALBERT

<https://zhuanlan.zhihu.com/p/87562926>

BERT 的 youxiu 变体：ALBERT 论文图解介绍

<https://zhuanlan.zhihu.com/p/142416395>

github：

中文版：[https://github.com/brightmart/albert\\_zh](https://github.com/brightmart/albert_zh)

## (7) ELECTRA (202003)

ELECTRA 是 Google 和斯坦福共同提出的，本质上还是一个 BERT 模型。

其最主要的贡献是：

- 提出了新的预训练框架，由 **Generator**（生成器）和 **Discriminator**（判别器）组成，有点类似 GAN
- 及其对应的新预训练任务 **RTD (Replaced Token Detection)**，判断当前 token 是否被语言模型替换过

ELECTRA 的预训练框架由两个部分构成，一个 BERT-small 作为 **Generator** 运行 **MLM 任务**，而另外一个 BERT 模型（即 ELECTRA 模型）作为 **Discriminator** 运行 **RTD 任务**，即负责判别这些生成出来的 token 有哪些被替换掉了。

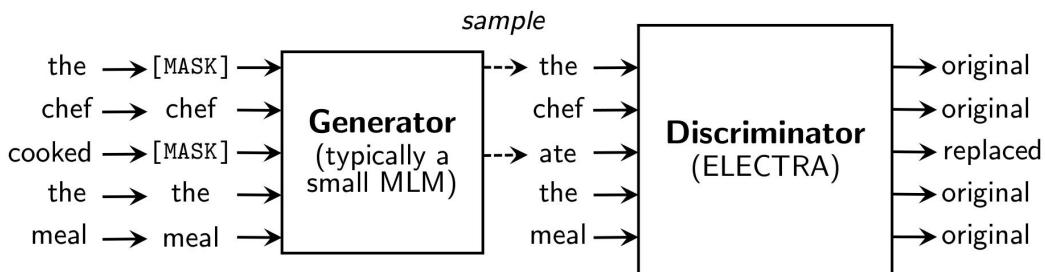


Figure 2: An overview of replaced token detection. The generator can be any model that produces an output distribution over tokens, but we usually use a small masked language model that is trained jointly with the discriminator. Although the models are structured like in a GAN, we train the generator with maximum likelihood rather than adversarially due to the difficulty of applying GANs to text. After pre-training, we throw out the generator and only fine-tune the discriminator (the ELECTRA model) on downstream tasks.

上述结构有个问题，输入句子经过生成器，输出改写过的句子，因为句子的字词是离散的，所以梯度在这里就断了，判别器的梯度无法传给生成器，于是生成器的训练目标还是

MLM（作者在后文也验证了这种方法更好），判别器的目标是序列标注（判断每个 token 是真是假），两者同时训练，但判别器的梯度不会传给生成器，目标函数如下：

$$\min_{\theta_G, \theta_D} \sum_{x \in \mathcal{X}} \mathcal{L}_{\text{MLM}}(x, \theta_G) + \lambda \mathcal{L}_{\text{Disc}}(x, \theta_D)$$

因为判别器的任务相对来说容易些，RTD loss 相对 MLM loss 会很小，因此加上一个系数，作者训练时使用了 50。

另外要注意的一点是，在优化判别器时计算了所有 token 上的 loss，而以往计算 BERT 的 MLM loss 时会忽略没被 mask 的 token。作者在后来的实验中也验证了在所有 token 上进行 loss 计算会提升效率和效果。

事实上，ELECTRA 使用的 Generator-Discriminator 架构与 GAN 还是有不少差别，作者列出了如下几点：

	ELECTRA	GAN
输入	真实文本	随机噪声
目标	生成器学习语言模型，判别器学习区分真假文本	生成器尽可能欺骗判别器，判别器尽量区分真假图片
反向传播	梯度无法从 D 传到 G	梯度可以从 D 传到 G
特殊情况	生成出了真实文本，则标记为正例	生成的都是负例（假图片）

论文：

<https://arxiv.org/pdf/2003.10555.pdf>

github：

官方：<https://github.com/google-research/electra>

哈工大的中文版 ELECTRA：<https://github.com/ymcui/Chinese-ELECTRA>

参考

ELECTRA：超越 BERT，19 年最佳 NLP 预训练模型

<https://zhuanlan.zhihu.com/p/89763176>

超越 BERT 模型的 ELECTRA 代码解读

<https://zhuanlan.zhihu.com/p/139898040>

## (8) DeBERTa (202006)

论文：

<https://arxiv.org/pdf/2006.03654.pdf>

github：

<https://github.com/microsoft/DeBERTa>

## 4 · T5 (201910)

论文：

<https://arxiv.org/pdf/1910.10683.pdf>

github：

<https://github.com/google-research/text-to-text-transfer-transformer>

# 十、研究方向：时间序列 (TS)

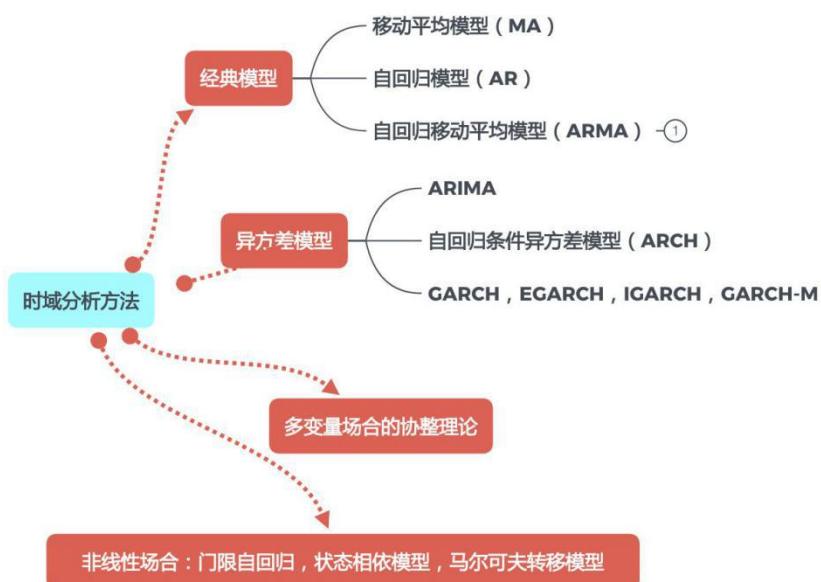
**时间序列 (Time Series)** 是一组按照时间发生先后顺序进行排列的数据点序列。通常一组时间序列的时间间隔为一恒定值（如 1 秒，5 分钟，12 小时，7 天，1 年），因此时间序列可以作为离散时间数据进行分析处理。

早期的时序分析通常都是直观的数据比较或绘图观测，寻找序列中蕴涵的发展规律，这种分析方法称为**描述性时序分析**。20 世纪 20 年代开始，学术界利用数理统计学原理来分析时间序列。研究的重心从总结表面现象（描述性时序分析）转移到分析序列值内的相关关系上（**统计时序分析**），由此开辟来一门应用统计学学科——时间序列分析。



在使用神经网络之前，时间序列分析使用了一系列传统模型，比如 AR, MA, ARMA 以及结合了这几个的 ARIMA 模型，而在此基础之上，ARCH 模型则解决了传统计量经济学中方差恒定这个不符合实际的假设（也是 2003 年的诺奖的成果之一）。

总体来说，时序分析的传统方法可以分为时域分析和频域分析。



时域分析包括：ACF、XCF、ARMA、ARIM、ARCH 等

频域分析包括：傅立叶变换、小波变换等

目前传统方法中效果最佳的方式是 COTE (35 个分类器的集合) 及其后续 HIVE-COTE (37 个分类器的集合)，但是为了实现高精度，HIVE-COTE 的计算量变得非常之大。

时序预测算法按照其实现原理可以分为：

- 传统统计学
- 机器学习（非深度学习）
- 深度学习

按照预测步长来分，可以分为：

- 单步预测：一次预测一个时间步骤
- 多步预测：一次预测多个时间步骤

按照输入变量区分，可分为：

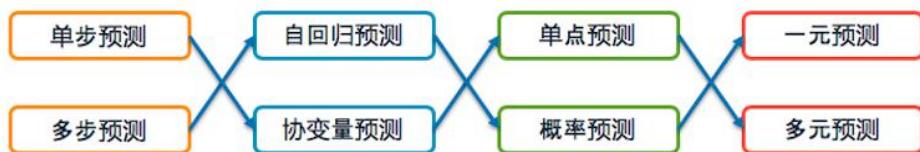
- 自回归预测：只以预测数据项和时间作为输入
- 协变量预测：也接受其他相关变量（协变量）作为输入

按输出结果分：

- 单点预测：输出结果为一个具体的值
- 概率预测：输出结果为一个具体的概率分布

按目标个数区分，可分为：

- 一元预测：预测目标只有一个
- 多元预测：预测目标有多个



#### 1. 简单移动平均 SMA



#### 2. ARIMA



#### 3. LSTM



#### 4. DeepAR



参考：

使用 DeepAR 进行时间序列预测

<https://amazonaws-china.com/cn/blogs/china/time-series-prediction-with-deep/>

The Complete Guide to Time Series Analysis and Forecasting

<https://towardsdatascience.com/the-complete-guide-to-time-series-analysis-and-forecasting-70d476bfe775>

传统方法：

<https://zhuanlan.zhihu.com/p/83511434>

《基于深度学习的时间序列分类》：

[https://blog.csdn.net/qq\\_32796253/article/details/88414656](https://blog.csdn.net/qq_32796253/article/details/88414656)

《深度学习在时间序列分类中的应用》：

<https://zhuanlan.zhihu.com/p/83130649>

综述论文：《Deep learning for time series classification: a review》

<https://arxiv.org/pdf/1809.04356.pdf>

翻译：《基于深度学习时间序列分类研究综述》

[https://blog.csdn.net/qq\\_32796253/article/details/88538231](https://blog.csdn.net/qq_32796253/article/details/88538231)

综述论文：《Deep Neural Network Ensembles for Time Series Classification》

<https://arxiv.org/pdf/1903.06602.pdf>

翻译：《用于时间序列分类的集成深度神经网络》

<https://zhuanlan.zhihu.com/p/60835712>

深度学习的时间序列预测有没有综述？ - Ada 的回答

<https://www.zhihu.com/question/405169480/answer/1329637122>

## 1 · 数据集

时间序列数据集中的每个样本为一个**时间序列（Time Series）**，每个时间序列分为若干**时间步骤（Time Step）**，每个时间步骤通常为一个标量或者向量。

## 2 · 衡量标准

### (二) 传统方法及概念

时序分析有很多传统（非 NN）的方法，下面介绍一些比较主流的。

参考：

<https://zhuanlan.zhihu.com/p/83511434>

[https://en.wikipedia.org/wiki/Time\\_series](https://en.wikipedia.org/wiki/Time_series)

## 1 · AR 模型

**Auto-regressive Model**，自动回归模型，一种处理时间序列的方法，用同一个变量  $x$  的之

前各周期，即  $x_1$  到  $x_{t-1}$ ，来预测本期  $x_t$  的取值，并假设它们为一线性关系。

因为这是从回归分析中的线性回归发展而来，只是不用  $x$  预测  $y$ ，而是用  $x$  预测  $x$ （自己），所以叫做自回归。

$X$  的当期值等于一个或数个前期值的线性组合，加常数项，加随机误差。公式如下：

$$X_t = c + \sum_{i=1}^p \varphi_i X_{t-i} + \varepsilon_t$$

其中：

$c$  是常数项。

$\varepsilon_t$  被假设为平均数等于 0，标准差等于  $\sigma$  的随机误差值； $\varepsilon_t$  被假设为对于任何的  $t$  都不变。

参考：

<https://zh.wikipedia.org/wiki/%E8%87%AA%E8%BF%B4%E6%AD%B8%E6%A8%A1%E5%9E%8B>

## 2 · VAR 模型

**Vector Auto-regression Model**，向量自回归模型，扩充了只能使用一个变量的 AR 模型，用在多变量时间序列模型上。

一个 VAR( $p$ ) 模型可以写成为：

$$y_t = c + A_1 y_{t-1} + A_2 y_{t-2} + \cdots + A_p y_{t-p} + e_t,$$

其中：

- $c$  是  $n \times 1$  常数向量
- $A_i$  是  $n \times n$  矩阵。
- $e_t$  是  $n \times 1$  误差向量，满足：

■  $E(e_t) = 0$ ：误差项的均值为 0

■  $E(e_t e_t') = \Omega$ ：误差项的协方差矩阵为  $\Omega$ （一个  $n \times n$  正定矩阵）

■  $E(e_t e_{t-k}') = 0$ （对于所有不为 0 的  $k$  都满足）—误差项不存在自相关

参考：

<https://zh.wikipedia.org/wiki/%E5%90%91%E9%87%8F%E8%87%AA%E5%9B%9E%E5%BD%92%E6%A8%A1%E5%9E%8B>

### 3 · MA 模型

**Moving Average Model**，滑动平均模型，是一种对单一变量时间序列进行建模的方法。移动平均模型和自回归模型都是时间序列中 **ARMA 模型** 和 **ARIMA 模型** 的重要组成部分，也是一种特殊情况。与自回归模型不同，移动平均模型总是平稳的。

$q$  阶移动平均模型通常简记为 **MA( $q$ )** :

$$x_t = \mu + \varepsilon_t - \theta_1 \varepsilon_{t-1} - \theta_2 \varepsilon_{t-2} - \cdots - \theta_q \varepsilon_{t-q}$$

或：

$$X_t = \mu + \varepsilon_t + \sum_{i=1}^q \theta_i \varepsilon_{t-i}$$

其中  $\mu$  是序列的均值， $\theta_1, \dots, \theta_q$  是参数， $\varepsilon_t, \varepsilon_{t-1}, \dots, \varepsilon_{t-q}$  都是白噪声。

参考：

<https://zh.wikipedia.org/wiki/%E7%A7%BB%E5%8A%A8%E5%B9%B3%E5%9D%87%E6%A8%A1%E5%9E%8B>

### 4 · 白噪声

**白噪声**，是一种功率谱密度为常数的随机信号或随机过程。即此信号在各个频段上的功率一致。

由于白光是由各种频率（颜色）的单色光混合而成，因而此信号的平坦功率谱性质称为“白色”，此信号也因此得名为白噪声。相对的，其他不具有这一性质的噪声信号则称为有色噪声。

理想的白噪声具有无限带宽，因而其能量是无限大，这在现实世界是不可能存在的。实际上，人常常将有限带宽的平整信号视为白噪声，以方便进行数学分析。

参考：

<https://zh.wikipedia.org/wiki/%E7%99%BD%E9%9B%9C%E8%A8%8A>

### 5 · ARMA 模型

**Auto-regressive moving average model**，自回归滑动平均模型。是研究时间序列的重要方法，由自回归模型（AR 模型）与移动平均模型（MA 模型）为基础混合构成。

自回归模型 **AR( $p$ )** :

$$X_t = c + \sum_{i=1}^p \varphi_i X_{t-i} + \varepsilon_t$$

移动平均模型 MA(q)：

$$X_t = \mu + \varepsilon_t + \sum_{i=1}^q \theta_i \varepsilon_{t-i}$$

自回归滑动平均模型 ARMA(p, q)包含了 p 个自回归项和 q 个移动平均项：

$$X_t = c + \varepsilon_t + \sum_{i=1}^p \varphi_i X_{t-i} + \sum_{j=1}^q \theta_j \varepsilon_{t-j}$$

有时 ARMA 模型可以用滞后算子 L 来表示， $L^i X_t = X_{t-i}$ 。这样 AR(p)模型可以写成：

$$\varepsilon_t = \left( 1 - \sum_{i=1}^p \varphi_i L^i \right) X_t = \varphi(L) X_t$$

MA(q)模型可以写成：

$$X_t = \left( 1 + \sum_{i=1}^q \theta_i L^i \right) \varepsilon_t = \theta(L) \varepsilon_t$$

因此，ARMA(p, q)模型可以表示为：

$$\left( 1 - \sum_{i=1}^p \varphi_i L^i \right) X_t = \left( 1 + \sum_{i=1}^q \theta_i L^i \right) \varepsilon_t$$

参考：

<https://zh.wikipedia.org/wiki/ARMA%E6%A8%A1%E5%9E%8B>

## 6 · ARIMA 模型

ARIMA 模型（Auto-regressive Integrated Moving Average Model，差分整合移动平均自回归模型），时间序列的预测模型之一。AR、MA、ARMA 模型都可以视作 ARIMA 模型的特例。

ARIMA(p, d, q)中，p 为回归项数，q 为滑动平均项数，d 为使之成为平稳序列的差分次数（阶数）：

$$\left(1 - \sum_{i=1}^p \phi_i L^i\right) (1 - L)^d X_t = \left(1 + \sum_{i=1}^q \theta_i L^i\right) \varepsilon_t$$

其中  $L$  是滞后算子 (Lag operator) ,  $d \in \mathbb{Z}, d > 0$

参考 :

<https://zh.wikipedia.org/wiki/ARIMA%E6%A8%A1%E5%9E%8B>

## 7 · ARFIMA 模型

Auto-regressive fractionally integrated moving average ,

参考 :

[https://en.wikipedia.org/wiki/Autoregressive\\_fractionally\\_integrated\\_moving\\_average](https://en.wikipedia.org/wiki/Autoregressive_fractionally_integrated_moving_average)

## 8 · ARCH 模型

Auto-regressive conditional heteroskedasticity model , 自回归条件异方差模型。解决了传统的计量经济学对时间序列变量的第二个假设 (方差恒定) 所引起的问题。这个模型是获得 2003 年诺贝尔经济学奖的计量经济学成果之一。

参考 :

<https://zh.wikipedia.org/wiki/ARCH%E6%A8%A1%E5%9E%8B>

<https://zhuanlan.zhihu.com/p/21962996>

## 9 · DTW

Dynamic Time Warping , 动态时间规整 , 是一种衡量两个长度不同的时间序列的相似度的方法。

参考 :

<https://blog.csdn.net/zouxy09/article/details/9140207>

[https://en.wikipedia.org/wiki/Dynamic\\_time\\_warping](https://en.wikipedia.org/wiki/Dynamic_time_warping)

## 10 · COTE

论文：

<https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=7069254>

## 11 · HIVE-COTE (2016ICDM)

论文：

<https://core.ac.uk/download/pdf/77027925.pdf>

# (三) 神经网络方法

## 1 · WaveNet (201609) (TODO)

论文：

<https://arxiv.org/pdf/1609.03499.pdf>

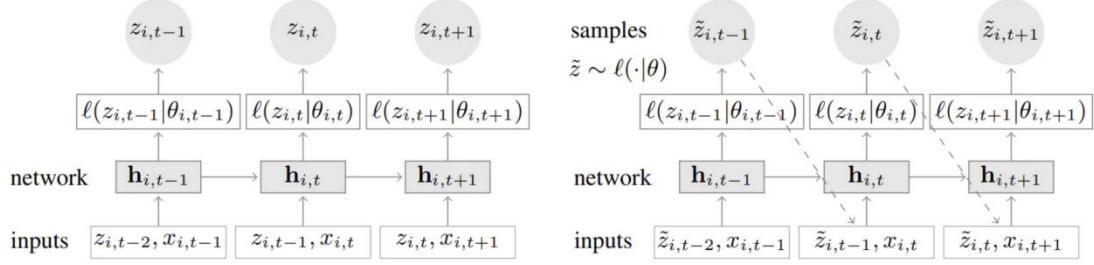
## 2 · DeepAR (201704)

DeepAR 是 Amazon 提出来的基于深度学习的时间序列预测方法，被收入 GluonTS。

DeepAR 相较于传统的 TS 方法，区别如下：

- 纳入了对额外特征的考虑，也接受其他协变量作为输入（传统方法只以预测数据项和时间作为输入）
- 其预测目标是序列在每个时间步骤上的概率分布（传统方法是单点预测）

下图左为训练过程，右为预测过程



训练时，在每个实践步骤  $t$ ，网络的输入特征包括  $x_{i,t}$ 、上一个实践步骤的取值  $z_{i,t-1}$ ，以及上一个时间步骤的隐状态  $h_{i,t-1}$ 。计算当前的隐状态的函数为：

$$h_{i,t} = h(h_{i,t-1}, z_{i,t-1}, x_{i,t})$$

进而计算似然函数  $l(z | \theta)$  的参数  $\theta_{i,t} = \theta(h_{i,t})$ ，最后将最大对数似然函数作为 Loss：

$$\mathcal{L} = \sum_i \sum_t \log l(z_{i,t} | \theta(\vec{h}_{i,t}))$$

来学习网络的参数。

DeepAR 本质上是一个 RNN，具体来说是多层的 LSTM（参考 gluonts 里的实现）

论文：

<https://arxiv.org/pdf/1704.04110.pdf>

参考：

<https://amazonaws-china.com/cn/blogs/china/time-series-prediction-with-deep/>

<https://www.jianshu.com/p/8a900b9ad3d3>

### 3 · Deep state(201800) (TODO)

论文：

<https://papers.nips.cc/paper/8004-deep-state-space-models-for-time-series-forecasting.pdf>

### 4 · Deep factor (201905) (TODO)

论文：

<https://arxiv.org/pdf/1905.12417.pdf>

参考：

# 十一、实践应用

应用类的方向不是一个理论研究方向，而是基于某个或者若干个理论方向的实际应用、工程化、数据集、系统集成等等。

## (一) 车牌识别

车牌识别用到的研究方向包括目标检测（车牌检测）和 OCR（车牌识别）

### 1 · 数据集

#### (1) CCPD

Chinese City Parking Dataset，用于车牌识别的大型数据集，由中科大构建。

优点：

- 包含各种复杂环境，不同天气、车牌倾斜、模糊、过亮过暗等等
- 数据量大，包含了 25 万张车牌

缺点：

- 每张图片只包含了一张车牌
- 由于所有数据采集都是在合肥（中科大所在地），因此车牌以徽 A 居多

### ① 格式

车牌的 ground truth 直接标记在文件名中，文件名包含 7 个部分，以“-”分隔，分别是：

- 车牌与整张图片的占比
- 倾斜角度，包括水平和垂直的倾斜角度，以“\_”分隔
- Bounding Box 的左上角、右下角，以“\_”分隔
- 车牌四角位置的坐标
- 车牌号，具体格式参考官网说明
- 亮度
- 模糊度

官网：<https://github.com/detectRecog/CCPD>

## 2 · 代码：EasyPR

EasyPR 是比较早期的一个中文车牌识别项目。

github：  
<https://github.com/liuruoze/EasyPR>

## 3 · 代码：HyperLPR

HyperLPR 也是 github 上比较目前仍在维护

github：  
<https://github.com/szad670401/HyperLPR>

## (二) 无人机识别

无人机识别是图像分类、目标检测、视频目标跟踪、视频分类的一个具体应用。

### 1 · 数据集

无人机数据集总览：  
<https://zhuanlan.zhihu.com/p/53151892>

SSD (Stanford Drone Dataset)：[http://cvgl.stanford.edu/projects/uav\\_data/](http://cvgl.stanford.edu/projects/uav_data/)  
Okutama Action Dataset：<http://okutama-action.org/>

## (三) 视频监控异常检测

Anomaly Detection in Surveillance Video，也不算是个研究方向，而应该是视频理解  
(比如动作识别、时序动作检测、时空动作检测的特化)

# 1 · Real-World Anomaly Detection in Surveillance Videos (201801) (TODO)

论文：

<https://arxiv.org/pdf/1801.04264.pdf>

参考：

<https://zhuanlan.zhihu.com/p/34553884>

## (四) 红绿灯识别

### 1 · 数据集

<https://github.com/ytzhaorobotics/wiki/TS-and-TL-Dataset>

[https://blog.csdn.net/weixin\\_42419002/article/details/100605115#5\\_ApolloScape\\_49](https://blog.csdn.net/weixin_42419002/article/details/100605115#5_ApolloScape_49)

#### (1) Lara 交通灯识别

巴黎的信号灯数据集

<http://www.lara.prd.fr/benchmarks/trafficlightsrecognition>

#### (2) WPI 数据集

包括交通灯、行人和车道检测

<http://computing.wpi.edu/dataset.html>

### (3) Bosch 交通灯数据集

<https://hci.iwr.uni-heidelberg.de/content/bosch-small-traffic-lights-dataset>

## 2 · 基于 OpenCV 的红绿灯识别

参考：

<https://zhuanlan.zhihu.com/p/93867116>

github：

<https://github.com/ZhiqiangHo/code-of-csdn/tree/master/Traffic%20Light%20Detection%20using%20Python%20OpenCV>

## 3 · 用深度学习识别交通灯

参考：

<https://zhuanlan.zhihu.com/p/24955921>

github：

## 十二、元学习

本章介绍了和深度学习本身相关的一些研究方向

### (一) 迁移学习

以下是王晋东的《迁移学习简明手册》：

<https://zhuanlan.zhihu.com/p/35352154>

[http://jd92.wang/assets/files/transfer\\_learning\\_tutorial\\_wjd.pdf](http://jd92.wang/assets/files/transfer_learning_tutorial_wjd.pdf)

## 1 · Fine-tuning

Transfer learning 和 Fine tuning 的区别和联系是什么？在 Tensorflow 语境里，Fine tuning 是 Transfer learning 的最后步骤，比如在预训练的图像分类 CNN 上迁移学习一个分类网络：

1. 去掉 top (include\_top=False)，将 base model 冻结，增加分类的 Dense 层
2. 训练分类的 Dense 层
3. 解冻 base model 的后面几层，训练。这步被叫做 Fine tuning

## (二) 集成学习

集成学习（ensemble learning），并不是一个单独的机器学习算法，而是通过构建并结合多个机器学习器来完成学习任务。集成学习往往被视为一种元算法（meta-algorithm）。

集成学习可以用于分类问题集成，回归问题集成，特征选取集成，异常点检测集成等等，可以说所有的机器学习领域都可以看到集成学习的身影。

对于训练集数据，我们通过训练若干个个体学习器（learner），通过一定的结合策略，就可以最终形成一个强学习器，以达到博采众长的目的。

也就是说，集成学习有两个主要的问题需要解决：

- 如何得到若干个个体学习器
- 如何选择一种结合策略，将这些个体学习器集合成一个强学习器。

而个体学习器按照个体学习器之间是否存在依赖关系可以分为两类：

- 个体学习器之间存在强依赖关系，一系列个体学习器基本都需要串行生成，代表算法是 boosting 系列算法
- 个体学习器之间不存在强依赖关系，一系列个体学习器可以并行生成，代表算法是 bagging 和随机森林（Random Forest）系列算法。

## 1 · Boosting

Boosting，指的是一系列集成学习元算法（ensemble meta algorithm），用于减小监督式学习中误差的。

Boosting 通过一系列弱学习算法（weak learner，准确率较低的学习算法，可能结果只比随机分类略好）的集合来生成一个强学习算法（strong learner）。

参考：

### (1) AdaBoost

**Adaptive Boosting**，是第一个成功的 Boost 算法，用于二元分类。

AdaBoost 方法的自适应在于：前一个分类器分错的样本会被用来训练下一个分类器。

AdaBoost 方法是一种迭代算法，在每一轮中加入一个新的弱分类器，直到达到某个预定的足够小的错误率。每一个训练样本都被赋予一个权重，表明它被某个分类器选入训练集的概率。如果某个样本点已经被准确地分类，那么在构造下一个训练集中，它被选中的概率就被降低；相反，如果某个样本点没有被准确地分类，那么它的权重就得到提高。通过这样的方式，AdaBoost 方法能“聚焦于”那些较难分（更富信息）的样本上。在具体实现上，最初令每个样本的权重都相等，对于第 k 次迭代操作，我们就根据这些权重来选取样本点，进而训练分类器  $C_k$ 。然后就根据这个分类器，来提高被它分错的样本的权重，并降低被正确分类的样本权重。然后，权重更新过的样本集被用于训练下一个分类器  $C_k$ 。整个训练过程如此迭代地进行下去。

参考：

<http://www.uml.org.cn/sjjmwj/2019030721.asp>

## (三) 模型压缩

模型压缩可以使得模型在时间和空间两个维度上更加的节省，通常来说，模型压缩分成两个主要的方式：

- **prune**：剪枝，改变模型的结构
- **quantize**：量化，将 float 型权重改为 int 或 binary 等更为简单和易于计算的形式

### 1 · 剪枝

#### (1) Activation Rank Filter(1607)

Activation APoZ Filter：基于指标 APoZ（平均百分比零）的剪枝过滤器，该指标测量（卷积）图层激活中零的百分比。

Activation Mean Rank Filter：基于计算输出激活最小平均值指标的剪枝过滤器

论文：

<https://arxiv.org/abs/1607.03250>

## (2) L1Filter 和 L2Filter(1608)

在卷积层中具有最小 L1 权重规范和 L2 权重规范的剪枝过滤器（用于 Efficient Convnets 的剪枝过滤器）。

论文：

<https://arxiv.org/pdf/1608.08710.pdf>

## (3) Slim(1708)

通过修剪 BN 层中的缩放因子来修剪卷积层中的通道。

论文：

<https://arxiv.org/pdf/1708.06519.pdf>

## (4) AGP(1710)

自动的逐步剪枝（是否剪枝的判断：基于对模型剪枝的效果）

论文：

<https://arxiv.org/pdf/1710.01878.pdf>

## (5) Lottery Ticket(1803)

反复修剪模型

论文：

<https://arxiv.org/pdf/1803.03635.pdf>

## (6) FPGM(1811)

论文：

<https://arxiv.org/pdf/1811.00250.pdf>

## 2 · 量化

### (1) BNN (201602)

论文：Binarized Neural Network

<https://arxiv.org/pdf/1602.02830.pdf>

### (2) DoReFa (201606)

论文：

<https://arxiv.org/pdf/1606.06160.pdf>

### (3) QAT(2018CVPR)

论文：

[http://openaccess.thecvf.com/content\\_cvpr\\_2018/papers/Jacob\\_Quantization\\_and\\_Training\\_CVPR\\_2018\\_paper.pdf](http://openaccess.thecvf.com/content_cvpr_2018/papers/Jacob_Quantization_and_Training_CVPR_2018_paper.pdf)

#### (4) XGBoost

eXtreme Gradient Boosting，

github : <https://github.com/dmlc/xgboost>

#### (5) CatBoost

#### (6) LightGBM

github : <https://github.com/Microsoft/LightGBM>

#### (7) GBDT

### (四) AutoML

**AutoML**，**Automated Machine Learning**，自动机器学习，是自动化应用机器学习于解决现实世界问题的过程。AutoML 覆盖了从原始数据处理到可部署模型的整个过程。高等级的 AutoML 可以使得普通用户也可以使用机器学习。

针对某个问题的解决，传统神经网络需要人类来进行网络模型的选择，算法的选择，数据的预处理，特征值的提取和选择，超参数的优化等等操作。AutoML 则希望将以上这些流程全部自动完成。

AutoML 可以做的工作包括：

- 数据准备：data preparation 数据预处理、清洗
- 特征工程：feature engineering，选择、构造、提取合适的特征
- 网络架构搜索：NAS (Network Architecture Search)
- 超参数优化：hyperparameter optimization
- 模型压缩：model compression

- 结果分析：result analysis，分析模型输出结果  
其中 NAS 被视为相对更有技术含量的部分。

目前的 AutoML 系统（这里只包括深度学习，不包括其他机器学习）包括：

- Auto-sklearn: sklearn 的 AutoML 实现
- Cloud AutoML: 谷歌的自动机器学习系统（收费）
- AutoKeras: Keras 的 AutoML 实现，被视为 Cloud AutoML 的开源替代
- NNI：微软推出的 AutoML 开源包，后端支持 Pytorch、Tensorflow、sklearn 等
- AutoGluon：Amazon 推出的 AutoML 系统

目前这些 AutoML 系统支持的主要任务见下表（Y 表示支持）：

	AutoKeras	Cloud AutoML	AutoGluon	NNI	Auto sklearn
图像分类	Y	Y (+Edge)	Y		
图像回归	Y				
目标检测		Y (+Edge)	Y		
视频分类		Y			
目标追踪		Y			
文本分类	Y	Y (NLP)	Y		
文本回归	Y				
结构数据处理	Y (分类+回归)	Y	Y (表格预测)		
机器翻译		Y			

这里有篇文章讨论四者的区别：

<https://medium.com/@santiagof/auto-is-the-new-black-google-automl-microsoft-automated-ml-autokeras-and-auto-sklearn-80d1d3c3005c>

参考：

<https://www.jiqizhixin.com/articles/2018-11-07-18>

<https://www.automl.org/automl/>

<https://www.jianshu.com/p/b59fe5dd1b93>

## 1 · AutoKeras(开源)

AutoKeras 是德州 A&M 大学的团队研发的开源项目，是 Google Cloud AutoML 的替代品。

官网：

<https://autokeras.com/>

论文：Auto-Keras: An Efficient Neural Architecture Search System

<https://arxiv.org/pdf/1806.10282.pdf>

github :

<https://github.com/keras-team/autokeras>

AutoKeras 目前支持的任务包括：

- Image Classification
- Image Regression
- Text Classification
- Text Regression
- Structured Data Classification
- Structured Data Regression

## (1) Python 包

AutoKeras 的 python 包名为 autokeras，具体结构如下：

- `autokeras`
  - `auto_model`
    - ◆ `AutoModel`
  - `encoder`
    - ◆ `Encoder`
    - ◆ `LabelEncoder`
    - ◆ `OneHotEncoder`
    - ◆ `serialize()`
    - ◆ `deserialize()`
  - `hypermodel`
    - ◆
  - `meta_model`
    - ◆ `Assembler`
    - ◆ `ImageAssembler`
    - ◆ `StructuredDataAssembler`
    - ◆ `TimeSeriesAssembler`
    - ◆ `assemble()`
  - `task`
    - ◆ `SupervisedImagePipeline`, `SupervisedStructuredDataPipeline`,  
`SupervisedTextPipeline`：分别是以下几个 class 的基类
    - ◆ `ImageClassifier`
    - ◆ `ImageRegressor`
    - ◆ `StructuredDataClassifier`
    - ◆ `StructuredDataRegressor`
    - ◆ `TextClassifier`
    - ◆ `TextRegressor`

- ◆ TimeSeriesForecaster
- tuner
  - ◆ AutoTuner
  - ◆ RandomSearch
  - ◆ Hyperband
  - ◆ BayesianOptimization
  - ◆ GreedyOracle
  - ◆ Greedy
- utils

## 2 · NNI (Microsoft 开源)

NNI (Neural Network Intelligence) 是一个轻量但强大的工具包，帮助用户自动的进行特征工程，神经网络架构搜索，超参调优以及模型压缩。

NNI 并没有提供其他 AutoML (AutoKeras、AutoGluon) 所提供的具体任务实现（比如 ImageClassification、TextClassification 之类），而是提供了一些相对更加底层且灵敏的工具。

NNI 管理自动机器学习 (AutoML) 的 Experiment，调度运行由调优算法生成的 Trial 任务来找到最好的神经网络架构和/或超参，支持各种训练环境，如本机，远程服务器，OpenPAI，Kubeflow，基于 K8S 的 FrameworkController (如，AKS 等)，以及其它云服务。

支持 Pytorch，Tensorflow，Keras，MXNet，sklearn 等框架。

文档：<https://nni.readthedocs.io/en/latest/Overview.html>

中文文档：<https://nni.readthedocs.io/zh/latest/Overview.html>

github：<https://github.com/microsoft/nni>

中文 github：[https://github.com/microsoft/nni/blob/master/README\\_zh\\_CN.md](https://github.com/microsoft/nni/blob/master/README_zh_CN.md)

### (1) 概念

- Trial：是将一组超参数在模型上的一次尝试，实现 trial 有两种方式，NNI API 和 NNI Annotation
- Experiment：实验，实验是一次找到模型的最佳超参组合，或最好的神经网络架构的任务。它由 Trial 和自动机器学习算法所组成。

- Configuration：配置是来自搜索空间的一个参数实例，每个超参都会有一个特定的值。
- Annotation：标记，NNI 的一种语法，通过加入一些注释，就可以启动 NNI，完全不影响代码原先的逻辑。
- Tuner：调参算法，Tuner 从 trial 接收结果 (metrics)，评估一组网络结构/超参的性能，然后将下一组网络结构/超参发给 trial。NNI 有内置的 tuner，也可以自定义。
- Assessor：评估算法，为了节省资源，可通过创建 Assessor 来配置提前终止策略。NNI 有内置的 Assessor，用户也可以自定义 assessor。
- Advisor：是 Tuner 和 Assessor 的结合体
- 训练平台：是 Trial 的执行环境。根据 Experiment 的配置，可以是本机，远程服务器组，或其它大规模训练平台（如，OpenPAI，Kubernetes）。

## (2) 实现

NNI 提供了两种模式来实现 AutoML：

- NNI API：在训练的 trial 文件中显式调用 nni（比如通过 nni.get\_next\_parameter() 来得到下一组参数），需要 3 个文件：
  - config.yml：主配置文件，其中定义了 trial command（执行 trial 文件），以及搜索空间文件，以及其他各种配置（比如 tuner、assessor 的选择等）。
  - xxxx.py：trial 文件，由普通的训练主程序增加对 nni 接口的调用实现，实现了单次训练（一个 Trial）
  - search\_space.json：定义了超参数的搜索空间
- NNI Annotation：在训练的主程序增加 Annotation，需要 2 个文件：
  - config.yml：主配置文件，其中定义了 trial command（执行 trial 文件），及其他各种配置。
  - xxxx.py：trial 文件，由普通的训练主程序增加 Annotation（以注释形式，因此即便没有 nni 环境也可以正常执行单次训练）来定义搜索空间

## (3) Python 包

NNI 的 Python 包名为 nni：

- **nni**
  - **tuner.Tuner**：Tuner 的基类
  - **xxx\_tuner**：各内置 tuner
  - **xxx\_advisor**：各内置 advisor
  - **assessor.Assessor**：Assessor 的基类
  - **xxx\_assessor**：各内置 assessor
  - **nas**：网络架构搜索算法

- ◆ tensorflow : 目前不支持
- ◆ pytorch : Pytorch 的 NAS
  - base\_mutator.BaseMutator :
  - base\_trainer.BaseTrainer :
  - mutator.Mutator :
  - trainer.Trainer :
  - classic-nas.mutator.ClassicMutator : NAS 算法
  - enas : ENAS 算法
    - trainer.EnasTrainer :
    - mutator.EnasMutator :
  - darts : DARTS 算法
    - mutator.DartsMutator :
    - trainer.DartsTrainer :
  - pdarts : P-DARTS 算法
    - mutator.PdartsMutator :
    - trainer.PdartsTrainer :
  - random.mutator.RandomMutator
  - spos
    - trainer.SPOSSupernetTrainer
    - mutator.SPOSSupernetTrainingMutator
    - evolution.SPOSEvolution
- compression : 模型压缩算法，主要包括剪枝算法和量化算法
  - ◆ tensorflow : tensorflow 的实现
    - compressor : tensorflow 压缩算法的抽象类
      - Compressor : tensorflow 压缩算法的基类
      - Pruner : tensorflow 剪枝算法的基类，继承 Compressor
      - Quantizer : tensorflow 量化算法的基类，继承 Compressor
    - builtin\_pruners : tf 内置的剪枝算法，包括 LevelPruner、AGP\_Prune 和 FPGMPruner
    - builtin\_quantizers : tf 内置的量化算法，包括 NaiveQuantizer、QAT\_Quantizer 和 DoReFaQuantizer
  - ◆ torch : Pytorch 的实现
    - compressor : Pytorch 压缩算法的抽象类
      - Compressor : Pytorch 压缩算法的基类
      - Pruner : Pytorch 剪枝算法的基类，继承 Compressor
      - Quantizer : Pytorch 量化算法的基类，继承 Compressor
      - QuantGrad :
    - pruners : 剪枝算法，包括 LevelPruner、AGP\_Pruner、SlimPruner 和 LotteryTicketPruner
    - quantizers : 量化算法，包括 NaiveQuantizer、QAT\_Quantizer、DoReFaQuantizer 和 BNNQuantizer
    - activation\_rank\_filter\_pruners :
      - ActivationRankFilterPruner : 以下两种剪枝算法的基类
      - ActivationAPoZRankFilterPruner

- ActivationMeanRankFilterPruner
- weight\_rank\_filter\_pruners :
  - WeightRankFilterPruner : 以下三种剪枝算法的基类
  - L1FilterPruner :
  - L2FilterPruner :
  - FPGMPruner :
- trial : 被 import 进 nni 模块，在 trial 文件中被调用
  - ◆ get\_next\_parameter() : 用于在 NNI API 模式中得到下一组超参数
  - ◆ get\_current\_parameter()
  - ◆ report\_intermediate\_result()
  - ◆ report\_final\_result()
  - ◆ get\_experiment\_id()
  - ◆ get\_trial\_id()
  - ◆ get\_sequence\_id()
- smartparam : NNI Annotation 的超参数选择，被 import 进 nni 模块
  - ◆ choice() : 超参数是输入的参数之一
  - ◆ randint() : 超参数是 round(uniform(low, high)) 的点
  - ◆ uniform() : 超参数是 low 和 high 之间均匀分布的某个值
  - ◆ loguniform() : 超参数是 exp(uniform(low, high)) 的点
- feature\_engineering : 特征工程
  - ◆ feature\_selector.FeatureSelector : 特征选择算法的抽象类
  - ◆ gbdt\_selector.gbdt\_selector.GBDTSelector :
  - ◆ gradient\_selector :

### 3 · Cloud AutoML (Google)

谷歌推出的在线 AutoML 服务，要钱。

官网：<https://cloud.google.com/automl/>

目前支持的任务包括：

- AutoML Vision
  - Image Classification : 图像分类
  - Image Classification (Edge) : 图像分类（边缘计算）
  - Object Detection : 目标检测
  - Object Detection (Edge) : 目标检测（边缘计算）
- AutoML Video Intelligence
  - Classification : 视频分类
  - Object Tracking : 目标追踪
- AutoML Natural Language : 自然语言处理
- AutoML Translation : 机器翻译
- AutoML Tables : 结构数据处理

## 4 · AutoGluon (Amazon)

AutoGluon 是亚马逊推出的 AutoML 框架。

官网：

<https://autogluon.mxnet.io/>

github:

<https://github.com/awslabs/autogluon>

AutoGluon 目前支持的任务包括：

- Tabular Prediction：表格预测，根据表格的其他列，预测某个列的值
- Image Classification：传统的图像分类任务
- Object Detection：传统的目标检测任务
- Text Classification：文本分类，比如情感分类

### (1) Python 包

- `autogluon.task`
  - `base`
    - ◆ `base_task`
      - `BaseDataset`
      - `BaseTask`
    - ◆ `base_predictor`.`BasePredictor`
  - `image_classification`
    - ◆ `image_classification`.`ImageClassification`
    - ◆ `classifier`.`Classifier`
    - ◆ `dataset`
      - `RecordDataset`
      - `NativeImageFolderDataset`
      - `ImageFolderDataset`
    - ◆ `losses` :
    - ◆ `metrics` :
    - ◆ `nets` :
      - `Identity`
      - `ConvBNRelu`
      - `ResUnit`
    - ◆ `pipeline` :
    - ◆ `utils` :

- `object_detection`
  - ◆ `object_detection.ObjectDetection` :
  - ◆ `detector.Detector`
  - ◆ `dataset` : 数据集载入
    - `get_dataset()` :
    - `base.DatasetBase` : 数据集 loader 的基类
    - `coco.COCO` : COCO 格式数据集 loader
    - `voc.CustomVOCDetection` :
  - ◆ `nets` :
  - ◆ `pipeline` :
  - ◆ `utils` :
- `tabular_prediction` :
  - ◆ `tabular_prediction.TabularPrediction` :
  - ◆ `predictor.TabularPredictor` :
  - ◆ `dataset.TabularDataset` : 加载数据集，返回统一的数据集格式
- `text_classification` :
  - ◆ `text_classification.TextClassification` :
  - ◆ `predictor.TextClassificationPredictor` :
  - ◆ `dataset` :
  - ◆ `transforms.BERTDatasetTransform`
  - ◆ `network` :
  - ◆ `pipeline` :
- `autogluon.`

## 5 · 算法：NAS

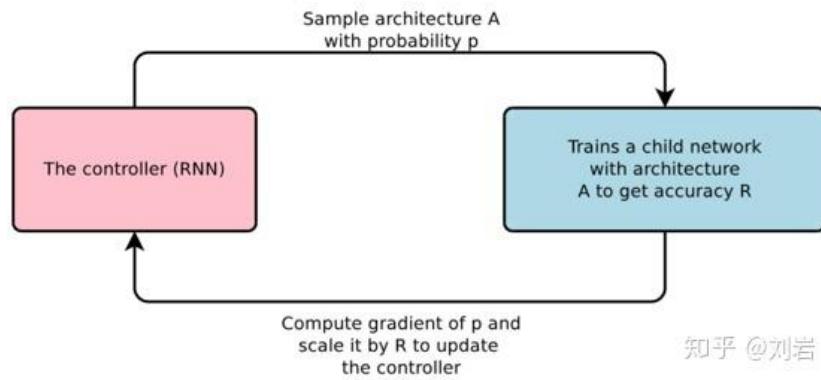
NAS (Neural Architecture Search) ,

### (1) NAS (201611)

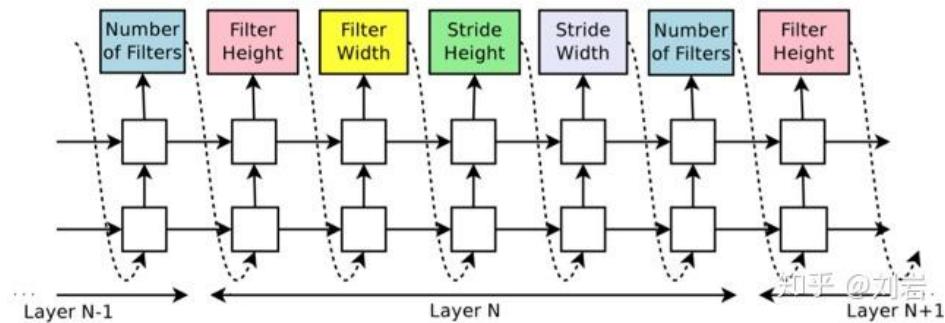
这篇文章提出了 **NAS** 这个概念，即 Neural Architecture Search，通过强化学习寻找最优的网络架构，包括一个 Image Classification 的卷积部分，和 RNN 的一个 Cell (类似 LSTM)。

由于现在的神经网络一般采用堆叠 block 的方式搭建而成，这种堆叠的超参数可以通过一个序列来表示。而这种序列的表示方式正是 RNN 所擅长的工作。

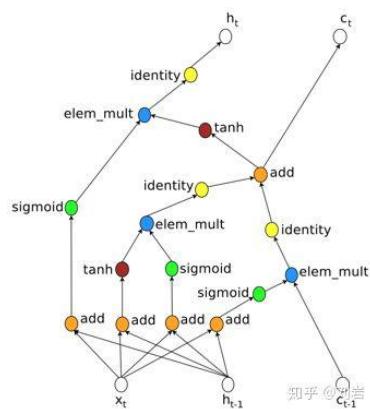
所以，NAS 会使用一个 RNN 构成的控制器 (controller) 以概率  $p$  随机采样一个网络结构  $A$ ，接着在 CIFAR-10 上训练这个网络并得到其在验证集上的精度  $R$ ，然后在使用  $R$  更新控制器的参数，如此循环执行直到模型收敛，如图所示：



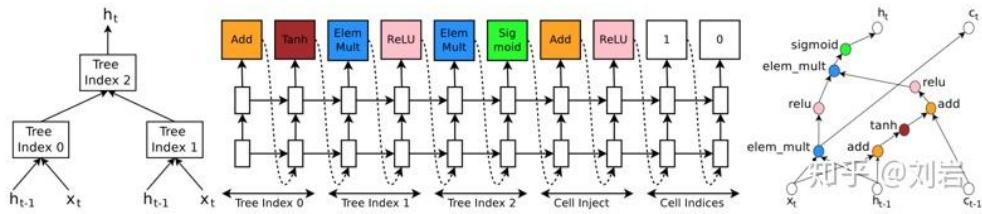
首先我们考虑最简单的 CNN，即只有卷积层构成。那么这种类型的网络是很容易用控制器来表示的。即将控制器分成 N 段，每一段由若干个输出，每个输出表示 CNN 的一个超参数，例如 Filter 的高，Filter 的宽，横向步长，纵向步长以及 Filter 的数量，如图所示：



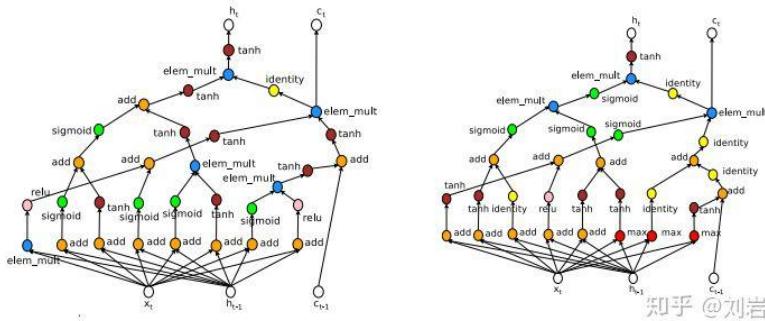
NAS 同样可以用来生成一个 RNN 的 Cell，传统的 LSTM 的计算图如下：



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



上面例子是使用“base 2”的超参作为例子进行讲解的，在实际中使用的是 base 8，得到下图两个 RNN 单元。左侧是不包含 max 和 sin 的搜索空间，右侧是包含 max 和 sin 的搜索空间（控制器并没有选择 sin）。



## 参考

<https://zhuanlan.zhihu.com/p/52471966>

## 论文：

<https://arxiv.org/pdf/1611.01578.pdf>

## (2) NASNet (201707)

## 论文：

<https://arxiv.org/pdf/1707.07012.pdf>

## (3) PNASNet (201712)

## 论文：

<https://arxiv.org/pdf/1712.00559.pdf>

## (4) ENAS (201802)

论文 : Efficient Neural Architecture Search Via Parameter Sharing

<https://arxiv.org/pdf/1802.03268.pdf>

## (5) AmoebaNet (201802)

参考 :

<https://zhuanlan.zhihu.com/p/57489362>

论文 :

<https://arxiv.org/pdf/1802.01548.pdf>

## (6) DARTS (201806)

论文 : DARTS: Differentiable Architecture Search

<https://arxiv.org/pdf/1806.09055.pdf>

## (7) P-DARTS (201904)

论文 : Progressive Differentiable Architecture Search: Bridging the Depth Gap between Search and Evaluation

<https://arxiv.org/pdf/1904.12760.pdf>

## (8) EvaNet (201811)

论文：

<https://arxiv.org/pdf/1811.10636.pdf>

代码：

<https://github.com/google-research/google-research/tree/master/evanet>

## (9) AssembleNet (201905)

论文：

<https://arxiv.org/pdf/1905.13209.pdf>

## (10) TinyVideoNetworks (201910)

论文：

<https://arxiv.org/pdf/1910.06961.pdf>

# 6 · 算法：特征工程

## (1) Gradient Selector(1908)

Gradient Feature Selector，基于梯度搜索算法的特征选择。

1. 该方法扩展了一个近期的结果，即在亚线性数据中通过展示计算能迭代的学习（即，在迷你批处理中），在线性的时间空间中的特征数量 D 及样本大小 N。
2. 这与在搜索领域的离散到连续的放松一起，可以在非常大的数据集上进行高效、基于梯度的搜索算法。
3. 最重要的是，此算法能在特征和目标间为  $N > D$  和  $N < D$  都找到高阶相关性，这与只考虑一种情况和交互式的方法所不同。

论文：Feature Gradients: Scalable Feature Selection via Discrete Relaxation

<https://arxiv.org/pdf/1908.10382.pdf>

## (五) 多示例学习

多示例学习，**Multi-Instance Learning**，MIL，在 20 世纪 90 年代在机器学习领域中提出的方法。在 MIL 中，“包”被定义为多个示例的集合，其中“正包”中至少包含一个正示例，而“负包”中则只有负示例（此处示例的概念与样本相同，以下不区分）。MIL 的目的是得到一个分类器，使得对于待测试的示例，可以得到其正负标签。

## (六) 半监督学习

Semi-supervised Learning

参考：

<https://zhuanlan.zhihu.com/p/33196506>

<https://zhuanlan.zhihu.com/p/138085660>

# 十三、硬件支持

由于神经网络的运算特点，使用 GPU 相较于 CPU 更加高效。传统的 GPU 公司 NVIDIA 和嵌入式芯片设计公司 ARM 因此都推出了一些相应的神经网络接口，以便直接对接上层的神经网络框架。

除此之外，神经网络运算和图形计算还是有些区别，比如神经网络计算对精度的要求并不高，8 位精度就能满足要求（只是使用网络的时候，训练网络的时候还是需要浮点精度），因此很多公司有 NPU (Neural Processing Unit) 推出，这其中包括 Google 设计并仅供自己使用的 TPU (Tensor Processing Unit)。

基本来说，适合运行神经网络的处理器可以分为三类：

- 独立的处理器：比如谷歌的 TPU、Intel Nervana NNP、Mobileye Myriad 2 等
- 基于 GPU 的产品：Nvidia Tesla 系列、AMD Raedon Instinct 系列等
- 协处理器：高通 Hexagon DSP、Apple Neural Engine、谷歌 Pixel Visual Core、ARM ML Processor 等，此外三星和华为的 SoC 都有内部的 NPU 协处理器

[https://en.wikipedia.org/wiki/AI\\_accelerator](https://en.wikipedia.org/wiki/AI_accelerator)

关于 GPU 和 CPU 之间的关系，有一个很形象的比喻，CPU 像是一个大学教授，而 GPU 像是一个班的小学生，小学生自然没有大学教授厉害，但是如果要完成 1000 道四则运算，反倒是小学生快点。而 NPU 的处理单元则像是一大帮幼儿园大班的小朋友，因为只需要算 5 以内的加法。

## (一) GPGPU

图形处理单元上的通用计算 (General-purpose computing on graphics processing units, GPGPU)，是利用处理图形任务的图形处理器来计算原本由中央处理器处理的通用计算任务。这些通用计算任务通常与图形处理没有任何关系。由于现代图形处理器有强大的并行处理能力和可编程流水线，令图形处理器也可以处理非图形数据。特别是在面对单指令流多数据流 (SIMD) 且数据处理的运算量远大于数据调度和传输的需要时，通用图形处理器在性能上大大超越了传统的中央处理器应用程序。

GPGPU 的概念在 2012 年神经网络开始复兴之前便被提出和实现，GPGPU 的目的并不仅仅是为了神经网络，而是为了以一种通用的方式把 GPU 的强大计算能力提供出来。

## (二) OpenCL

OpenCL (Open Computing Language，开放计算语言) 是一个为异构平台编写程序的框架，

此异构平台可由 CPU、GPU、DSP、FPGA 或其他类型的处理器与硬件加速器所组成。OpenCL 由一门用于编写 kernels（在 OpenCL 设备上运行的函数）的语言（基于 C99）和一组用于定义并控制平台的 API 组成。OpenCL 提供了基于任务分割和数据分割的并行计算机制。

### （三）CUDA（NVIDIA）

CUDA（Compute Unified Device Architecture，统一计算架构）是由 [NVIDIA](#) 所推出的一种 GPGPU 技术。CUDA 可以兼容 [OpenCL](#) 或者自家的 C-编译器。无论是 CUDA C-语言或是 OpenCL，指令最终都会被驱动程序转换成 PTX 代码，交由显示核心计算。

### （四）cuDNN（NVIDIA）

CUDA Deep Neural Network，是 NVIDIA 推出的 DNN 的 GPU 加速库，cuDNN 提供了专门针对类似卷积、池化、正规化等 DNN 计算的高性能实现。

CMSIS-NN 则是其中专为神经网络的应用而提供的统一抽象接口。

### （五）TPU（Google）

TPU（Tensor Processing Unit，张量处理单元），Google 于 2016 年 5 月发布的专为机器学习和 TensorFlow 定制的专用集成电路。第一代 TPU 提供高吞吐量的低精度运算（8 位），面向使用或运行模型，而不是训练模型。Google 宣布他们已经在数据中心中运行 TPU 长达一年多，发现它们对机器学习提供一个数量级更优的每瓦特性能。

2017 年 5 月，Google 推出第二代 TPU，并在 Google Compute Engine 中使用。第二代 TPU 提供最高 180TFLOPS 的性能，组装成 64 个 TPU 的集群时提供最高 11.5 PFLOPS 的性能。相较于第一代，TPU v2 不仅可以计算整数，也可以计算浮点数，这使得它不仅仅可以用于使用网络，也可以被用于训练网络模型。

2018 年 5 月，Google 推出第三代 TPU，Google 宣布这一代 TPU 比上一代的性能翻倍。

## （六）Linux 下 Nvidia/CUDA/cuDNN 的安装

Nvidia 驱动、CUDA 和 cuDNN，首先搞清楚这三者的关系

1. 你得有一块 NV 的显卡，才需要装 NVidia 的驱动。
2. 在有了显卡和驱动的情况下，才可以安装 CUDA（在显卡和驱动版本支持的情况下）

3. cuDNN 基于 CUDA 的神经网络的驱动，是在 CUDA 的基础上安装的

## 1. NVidia 驱动

Nvidia 驱动可以从 Nvidia 官网下载安装，也可以从系统包安装

### (1) Nvidia 官网

- 下载

<https://www.nvidia.com/Download/index.aspx>

从官网下载对应的可执行安装程序 NVIDIA-Linux-x86\_64-xxx.xx.run (64 位)

- 安装 Nvidia 驱动

```
service lightdm stop #首先需要停止 X server，某些情况下需要手动杀死 Xorg
./NVIDIA-Linux-x86_64-xxx.xx.run #执行安装脚本
service lightdm start #启动 X server
```

如果装完之后出现循环登录，可以卸载之后添加--no-opengl-files 选项重新安装一次

- 卸载 Nvidia 驱动

```
./NVIDIA-Linux-x86_64-xxx.xx.run --uninstall #用下载的安装程序进行卸载
或者
nvidia-uninstall #用安装出来的可执行脚本卸载
```

### (2) Ubuntu 官方

- Ubuntu 安装

```
apt install nvidia-xxx #xxx 为版本，例如 nvidia-418
```

- Ubuntu 卸载

```
apt remove --purge nvidia-xxx
```

## 2 · CUDA

CUDA 同理，可以从 NV 官网直接下载或者通过 Linux 系统 apt 方式安装

## (1) NVidia 官网

- 下载

```
https://developer.nvidia.com/cuda-downloads
```

- 安装 CUDA

官网提供了四种 CUDA 安装方式：

1. runfile(local)：下载可执行文件到本地，运行安装
2. deb(local)：增加一个本地的仓库，而该仓库的建立通过安装一个 deb 实现，通过 apt install 安装该仓库中的包
3. deb(network)：增加一个远程的仓库，通过 apt install 安装该仓库中的包
4. cluster(local)：.tar.gz 文件

- 卸载 CUDA

```
apt purge --autoremove cuda
```

## (2) Ubuntu 官方

- Ubuntu 安装

```
apt install nvidia-cuda-toolkit
```

- Ubuntu 卸载

```
apt purge --autoremove nvidia-cuda-toolkit
```

## 3 · CuDNN

CuDNN 也是一样，安装来源可以是 NV 官网或者 Linux 系统

### (1) Nvidia 官网

- 下载 CuDNN

```
https://developer.nvidia.com/rdp/cudnn-archive
```

下载内容包括三个，运行库，开发库，代码样例和使用说明，都是 deb 的形式

- 安装 CuDNN

```
dpkg -i libcudnn7_7.6.3.30-1+cuda10.1_amd64.deb
```

## (2) Ubuntu 官方

- 安装

```
apt install libcudnn7
```

## (七) Keras 中 GPU/CPU 的切换

```
import tensorflow as tf

from keras import backend as K
CPU_config = tf.ConfigProto(intra_op_parallelism_threads=4,\n inter_op_parallelism_threads=4, allow_soft_placement=True,\n device_count = {'CPU' : 1, 'GPU' : 0})
GPU_config = tf.ConfigProto(intra_op_parallelism_threads=4,\n inter_op_parallelism_threads=4, allow_soft_placement=True,\n device_count = {'CPU' : 1, 'GPU' : 1})

session = tf.Session(config=CPU_config)
K.set_session(session)
```

# 十四、Nvidia GPU

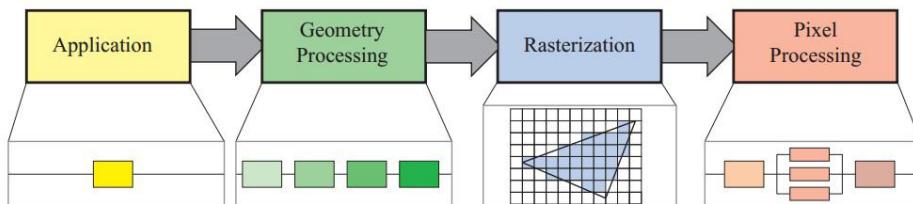
## (一) 图形渲染流水线

**Graphics Rendering Pipeline** (图形渲染流水线，或者叫做图形渲染管线)，其主要功能是将一个三维场景（给定的虚拟相机、三维物体、光源、照明模式、纹理等），生成一副二维图像。

图形渲染管线类似CPU中的指令流水线（事实上管线和流水线是同一个词 pipeline），其使得图形渲染的过程并发，也就是说GPU中的不同部分同时工作，处理不同的图形数据。比如当第一帧的图像执行到第三阶段的时候，第二帧的数据在第二阶段处理，而第三帧的数据在第一阶段处理，因而无须等第一帧的数据全部处理完毕之后才可处理第二帧数据。毫无疑问，图形渲染管线的速度由其中最慢的那个阶段决定。

渲染管线可以被分为四个大的阶段：

- 应用程序阶段 (Application Stage)
- 几何阶段 (Geometry Processing Stage)
- 光栅化阶段 (Rasterization Stage)
- 像素处理阶段 (Pixel Processing Stage)



**Figure 2.2.** The basic construction of the rendering pipeline, consisting of four stages: application, geometry processing, rasterization, and pixel processing. Each of these stages may be a pipeline in itself, as illustrated below the geometry processing stage, or a stage may be (partly) parallelized, as shown below the pixel processing stage. In this illustration, the application stage is a single process, but this stage could also be pipelined or parallelized. Note that rasterization finds the pixels inside a primitive, e.g., a triangle.

其中应用程序阶段的工作是由**应用程序**在CPU上完成的，而几何阶段和光栅化阶段的工作通常是在GPU上完成的。

参考：

《Real-Time Rendering 3rd》 提炼总结

<https://zhuanlan.zhihu.com/p/26527776>

实时渲染第四版翻译：第二章图形渲染管道

<https://zhuanlan.zhihu.com/p/97586708>

## 1 · 应用程序阶段

**应用程序阶段 (Applicaton Stage)** 一般是图形渲染管线概念上的第一个阶段。应用程序阶段是通过软件方式来实现的阶段（应用程序在 CPU 上执行），开发者能够对该阶段发生的情况进行完全控制，可以通过改变实现方法来改变实际性能。其他阶段，他们全部或者部分建立在硬件基础上，因此要改变实现过程会非常困难。

而与此相对的，应用程序阶段基于软件实现的一个结果是它不能被划分成一些子阶段，像图形处理，光栅化，像素处理阶段那样。然而，为了提升性能，这个阶段通常在多个处理器核心上并行的执行。

在应用程序阶段的末端，将需要在屏幕上（具体形式取决于具体输入设备）显示出来绘制的几何体（也就是绘制图元，rendering primitives，如点、线、矩形等）输入到绘制管线的下一个阶段（**几何阶段**）。

## 2 · 几何阶段

从**几何阶段 (Geometry Stage)** 主要负责大部分多边形操作和顶点操作。可以将这个阶段进一步划分成如下几个功能阶段：

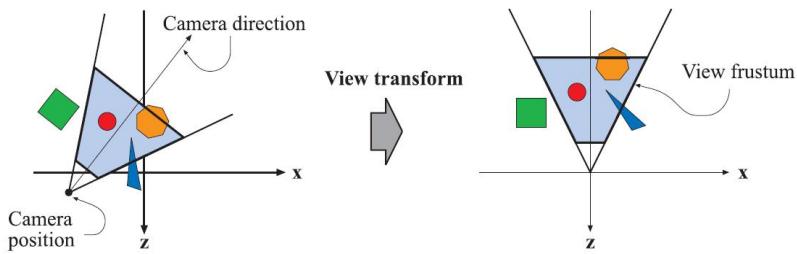
- **模型视点变换 Model & View Transform**：模型变换的目的是将模型变换到适合渲染的空间当中，而视图变换的目的是将摄像机放置于坐标原点，方便后续步骤的操作。
- **顶点着色 Vertex Shading**：顶点着色的目的在于确定模型上顶点处材质的光照效果
- **投影 Projection**：投影阶段就是将模型从三维空间投射到了二维的空间中的一个过程。投影阶段也可以理解为将视体变换到一个对角顶点分别是 $(-1, -1, -1)$ 和 $(1, 1, 1)$ 单位立方体内的过程。
- **裁剪 Clipping**：裁剪阶段的目的，就是对部分位于视体内部的图元进行裁剪操作
- **屏幕映射 Screen Mapping**：屏幕映射阶段的主要目的，就是将之前步骤得到的坐标映射到对应的屏幕坐标系上



### (1) 模型视点变换 (Model & View Transform)

**模型变换 (Model Transform)** 的目的是将模型变换到适合渲染的空间当中。

**视点变换 (View Transform)** 目的就是要把相机放在原点，然后进行视点校准，使其朝向 Z 轴负方向，y 轴指向上方，x 轴指向右边。



## (2) 顶点着色 (Vertex Shading)

确定材质上的光照效果的操作被称为着色 (shading)。

通常，这些计算中的一些在几何阶段期间在模型的顶点上执行，而其他计算可以在每像素光栅化 (per-pixel rasterization) 期间执行。可以在每个顶点处存储各种材料数据，诸如点的位置，法线，颜色或计算着色方程所需的任何其它数字信息。顶点着色的结果（其可以是颜色，向量，纹理坐标或任何其他种类的阴着色数据）计算完成后，会被发送到光栅化阶段以进行插值操作。

顶点着色还包括几个可选的子阶段，包括：

- 曲面细分 Tessellation：想象一下你有一个弹球物体。如果你仅仅用一系列三角形来代表弹球，你会遇到质量或性能的问题。你的球或许在 5 米外看起来很好，但是靠近每个独立的三角形，尤其是沿着轮廓，轮廓会变得可见。如果为了改进质量你用更多的三角形代表球，当球离得很远，只在屏幕上占很少的像素，你会浪费很可观的处理时间和内存。使用曲面细分，曲面可以通过适当数量的三角形来生成。曲面细分又由三个子阶段构成，分别是外壳着色 Hull Shading，曲面细分 Tessellation，和域着色 Domain Shading
- 几何着色 Geometry Shading：着色器早于曲面细分着色器，因此在 GPU 上更常见。就像曲面细分着色器一样，它接收各种类型的图元然后产生新的顶点。这是一个更简单的阶段在于它创建顶点的范围是受限的并且输出图元的类型也有更多的限制。几何着色器有多种用途，其中最受欢迎的是生成粒子。想象一下模拟烟花爆炸。每个火球可以用一个顶点来代表。几何着色器可以把每个顶点变成正方形（由两个三角形组成），该正方形面向观察者，覆盖几个像素，因此为我们提供了一个更可信的图元用来着色。
- 流输出 Stream Output：这个阶段使我们把 GPU 当作几何引擎来使用。取而代之的把我们处理过的顶点发往剩余的管道渲染到屏幕上，在这里我们可以选择把这些输出到一个数组里，以便将来处理。这些数据可以被 CPU 或是 GPU 使用，在之后的过程中。此阶段通常用于粒子模拟，比如我们烟花的粒子。

这三个阶段是以这样的顺利来执行的——曲面细分，几何着色，和流输出，每个都是可选的。无论哪个可选的管道被使用（也可能都没使用），如果我们继续沿着管道，我们会有 一系列齐次坐标的顶点将要用来被检查相机是否能看到它们。

### (3) 投影 (Projection)

投影 (Projection) 就是将模型从三维空间投射到了二维的空间中的过程。

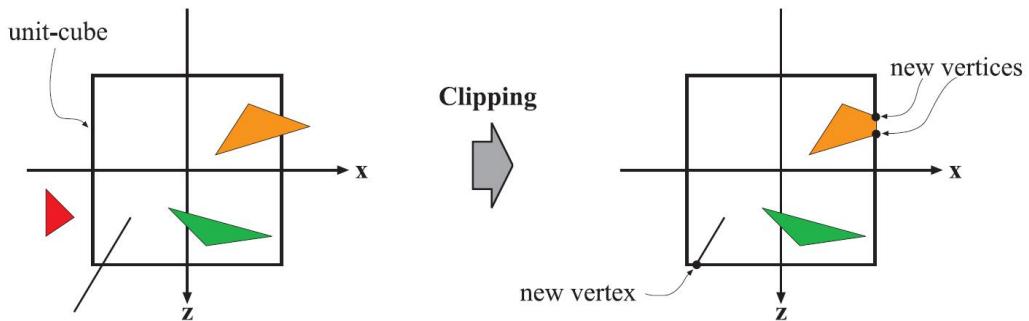
光照处理之后，渲染系统就开始进行投影操作，即将视体变换到一个对角顶点分别是 $(-1, -1, -1)$ 和 $(1, 1, 1)$ 单位立方体 (unit cube) 内，这个单位立方体通常也被称为规范立方体 (Canonical View Volume, CVV)。

目前，主要有两种投影方法，即：

- 正交投影 (orthographic projection, 或称 parallel projection)。
- 透视投影 (perspective projection)。

### (4) 裁剪(Clipping)

只有当图元完全或部分存在于视体（也就是上文的规范立方体，CVV）内部的时候，才需要将其发送到光栅化阶段，裁剪就是对部分位于视体内部的图元进行裁剪操作。

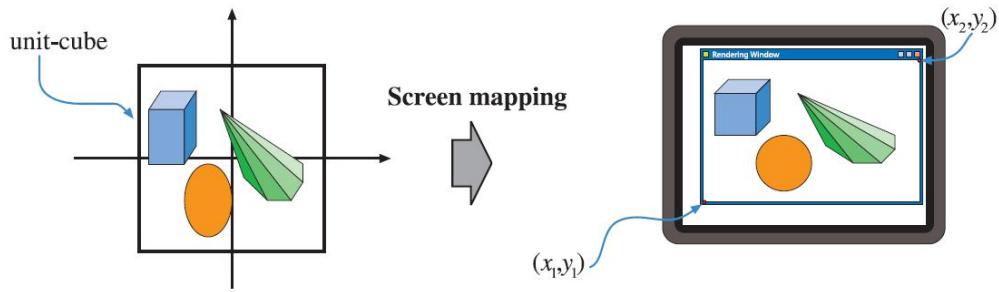


### (5) 屏幕映射 (Screen Mapping)

屏幕映射 (Screen Mapping) 的主要目的，就是将之前步骤得到的坐标映射到对应的屏幕坐标系上。

进入到这个阶段时，坐标仍然是三维的（但显示状态在经过投影阶段后已经成了二维），每个图元的 x 和 y 坐标变换到了屏幕坐标系中，屏幕坐标系连同 z 坐标一起称为窗口坐标

系。



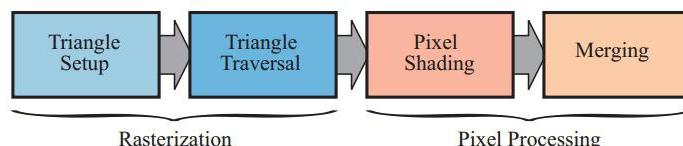
### 3 · 光栅化阶段 & 像素处理

给定经过变换和投影之后的顶点，颜色以及纹理坐标（均来自于几何阶段），给每个像素（Pixel）正确配色，以便正确绘制整幅图像。这个过程叫光栅化（rasterization）或扫描变换（scan conversion），即从二维顶点所处的屏幕空间（所有顶点都包含Z值即深度值，及各种与相关的着色信息）到屏幕上的像素的转换。

与几何阶段相似，该阶段细分为几个功能阶段：

- **三角形设定（Triangle Setup）**：这个阶段主要计算三角形的一些数据。这些数据用于三角形遍历阶段，也被几何阶段产生的插值的着色器数据所使用。这个阶段由固定的硬件执行。
- **三角形遍历（Triangle Traversal）**：找到哪些采样点或像素在三角形中的过程。
- **像素着色（Pixel Shading）**：像素着色阶段的主要目的是计算所有需逐像素计算操作的过程。
- **融合（Merging）**：融合阶段的主要任务是合成当前储存于缓冲器中的由之前的像素着色阶段产生的片段颜色。此外，融合阶段还负责可见性问题（Z缓冲相关）的处理。

这四个阶段又被分为两个部分：光栅化（Rasterization）和像素处理（Pixel Processing）：



**Figure 2.8.** Left: rasterization split into two functional stages, called triangle setup and triangle traversal. Right: pixel processing split into two functional stages, namely, pixel shading and merging.

## (1) 三角形设定

三角形设定阶段主要用来计算三角形表面的差异和三角形表面的其他相关数据。该数据主要用于扫描转换 (scan conversion)，以及由几何阶段处理的各种着色数据的插值操作所用。该过程在专门为设计的硬件上执行。

## (2) 三角形覆盖

在三角形遍历阶段将进行逐像素检查操作，检查该像素处的像素中心是否由三角形覆盖，而对于有三角形部分重合的像素，将在其重合部分生成片段 (fragment)。

## (3) 像素着色

像素着色 (Pixel Shading) 的主要目的是计算所有需逐像素操作的过程。

所有逐像素的着色计算都在像素着色阶段进行，使用插值得来的着色数据作为输入，输出结果为一种或多种将被传送到下一阶段的颜色信息。纹理贴图操作就是在这阶段进行的。

像素着色是在可编程 GPU 内执行的，在这一阶段有大量的技术可以使用，其中最常见，最重要的技术之一就是纹理贴图 (Texturing)。简单来说，纹理贴图就是将指定图片“贴”到指定物体上的过程。而指定的图片可以是一维，二维，或者三维的，其中，自然是二维图片最为常见：



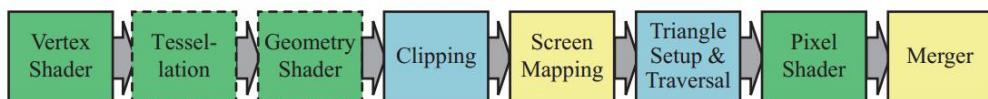
## (4) 融合

每个像素的信息都储存在颜色缓冲器中，而颜色缓冲器是一个颜色的矩阵（每种颜色包含红、绿、蓝三个分量）。融合阶段的主要任务是合成当前储存于缓冲器中的由之前的像素着色阶段产生的片段颜色。不像其它着色阶段，通常运行该阶段的 GPU 子单元并非完全可编程的，但其高度可配置，可支持多种特效。

此外，这个阶段还负责可见性问题的处理。这意味着当绘制完整场景的时候，颜色缓冲器中应该还包含从相机视点处可以观察到的场景图元。对于大多数图形硬件来说，这个过程是通过 Z 缓冲（也称深度缓冲器）算法来实现的。Z 缓冲算法非常简单，具有  $O(n)$  复杂度（ $n$  是需要绘制的像素数量），只要对每个图元计算出相应的像素 z 值，就可以使用这种方法。

## (二) GPU 的构成

GPU 的构成和图形渲染流水线基本对应，大致如下：



**Figure 3.2.** GPU implementation of the rendering pipeline. The stages are color coded according to the degree of user control over their operation. Green stages are fully programmable. Dashed lines show optional stages. Yellow stages are configurable but not programmable, e.g., various blend modes can be set for the merge stage. Blue stages are completely fixed in their function.

可以看到分别对应图形渲染流水线中的某个部分，其中绿色表示高度可编程，蓝色表示是固定流程，黄色则介于两者之间，虽然不可编程但是高度可配置。虚线表示可选阶段。

在早期 GPU 的参数中，可以看到核心结构的参数为 a:b:c:d，这四个值分别是：

顶点着色器：像素着色器：纹理单元：光栅单元

后期的 GPU 中（从 GeForce 8 系列开始），由于统一着色器架构的出现，编程了 a:b:c，三个值分别为：流处理器：纹理单元：光栅单元

其中流处理器（Unified Shader）即统一渲染结构的着色器（顶点着色器/几何着色器/像素着色器）

## 1 · 着色器

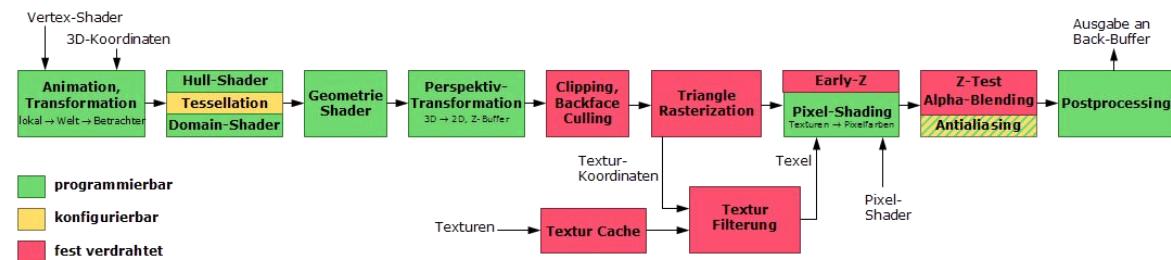
计算机图形学领域中，着色器（shader）是一种计算机程序，原本用于进行图像的浓淡处理（计算图像中的光照、亮度、颜色等），但近来，它也被用于完成很多不同领域的工  
作，比如处理 CG 特效、进行与浓淡处理无关的影片后期处理、甚至用于一些与计算机图形学无关的其它领域。

GPU 的可编程图形流水线已经全面取代传统的固定流水线，可以使用着色器语言对其进行编程。构成最终图像的像素、顶点、纹理，它们的位置、色相、饱和度、亮度、对比度也都可以利用着色器中定义的算法进行动态调整。调用着色器的外部程序，也可以利用它向着色器提供的外部变量、纹理来修改这些着色器中的参数。

随着图形处理器的进步，OpenGL 和 Direct3D 等主要的图形软件库都开始支持着色器。第一批支持着色器的 GPU 仅支持像素着色器，但随着开发者逐渐认识到着色器的强大，很快便出现了顶点着色器。2000 年，第一款支持可编程像素着色器的显卡 Nvidia GeForce 3 (NV20) 问世。Direct3D 10 和 OpenGL 3.2 则引入了几何着色器。

着色器可以分为二维着色器和三维着色器，目前二维着色器只有像素着色器一种，而三维着色器则包括顶点着色器和几何着色器等，即：

- 二维着色器
  - 像素着色器（像素处理->像素着色阶段）
- 三维着色器
  - 顶点着色器（几何阶段->顶点着色阶段）
  - 几何着色器（几何阶段->几何着色阶段）
  - 曲面细分着色器 Tesselation Shader (几何阶段->曲面细分阶段)



## 2 · 统一着色器/流处理器

Unified Shader，也叫统一渲染单元，Nvidia 管它叫流处理器 (Stream Processor, SP) 或者 CUDA 核心 (CUDA Core)。

微软在 DirectX 10 提出了统一着色器的概念，统一了各种着色器，Nvidia 则从 GeForce 8 系列 (Tesla 微架构) 开始，使用了统一着色器架构，不再区分具体的着色器。

## 3 · CUDA 核心

CUDA Core，这是 Nvidia 独有的定义 (CUDA 是 Nvidia 的商标)，即可以运行 CUDA 程序的

核心，基本对应于通用概念中的统一着色器/流处理器。

## 4. 纹理单元

也叫纹理映射单元（TMU，Texture Mapping Unit），TMU 能够旋转，调整大小和扭曲位图图像（执行纹理采样），以作为理放置在给定 3D 模型的任意平面上。此过程称为纹理制图映射。

## 5. 光栅单元

光栅单元（ROP，Raster Operation Pipeline），也叫渲染输出单元（Render Output Unit）。是 GPU 渲染过程的最后步骤之一。图形管线取像素（每个像素是一个无量纲点），和纹理像素信息，并处理它，经由特定的矩阵和向量运算，变成最终像素或深度值。此过程对应图形渲染流水线中的光栅化阶段。

## 6. 光线追踪核心

光追核，Ray Tracing Core（RT Core），专门用于处理光线追踪运算的硬件。Nvidia 从 RTX 开始（即 RTX 20 系列，之前没有 RT 核的都叫 GTX）加入了 RT 核。

## 7. 张量核心

Tensor Core，是部分 Nvidia GPU 中提供的，用于进行矩阵运算的硬件（GeForce 系列是从 RTX 20 系列开始使用第二代张量核心）。更多用在专用的用于进行神经网络运算的 GPU 中，有的没有显示输出，俗称无头卡，比如 Tesla 系列。

## 8. GPU 大核

Streaming Multiprocessor，SM，也叫 GPU 大核。在 Nvidia 不同世代的 GPU 中也有被叫做 SMM、SMX 等，有时候被误翻译成流处理器。

一个 GPU 中包含一个到若干个 SM，比如 RTX 2080 Ti 包含 68 个 SM。

一组线程（Thread Block）会被打包交给一个 SM 进行处理，这个 Thread Block 会被切成

若干 Thread Warp，在 SM 中的执行粒度是一个 Thread Warp。

SM 中包含了：

- 成千上万个寄存器
- Load/Store Units
- 各种 cache：
  - 线程共享的内存
  - L1 Cache
  - Constant Cache
  - 纹理 Cache
- Warp 调度器，用于进行 Warp 间的切换
- 执行单元（流处理器，CUDA 核）
  - 整数和单精度浮点执行单元
  - 双精度浮点执行单元
  - Special Function Unit (SFU)
- 纹理单元
- 光栅单元
- Tensor Core (Volta、Ampere 微架构)

这些不同的组成部分在 Nvidia 每一代不同的微架构都有调整。

Pascal 微架构：





Volta 微架构：



参考：

NVidia 流处理器发展史

<https://zhuanlan.zhihu.com/p/139801202>

<https://www.zhihu.com/question/267104699/answer/320361801>

### (三) NVidia 微架构 (Microarchitecture)

目前，图形硬件正在朝统一着色器架构（Unified Shading Architecture）发展。统一着色器模型的好处是更加灵活。

自 GeForce8 系列开始采用统一着色器架构之后，NVIDIA 的 GPU 微架构的 codename 依次是：

**Tesla**（并非指产品系列 Tesla）、**Fermi**、**Kepler**、**Maxwell**、**Pascal**，之后分为 **Turing**（消费级产品使用）和 **Volta**、**Ampere**（专业产品/图形工作站）

#### 1 · Pascal 架构

采用帕斯卡微架构的显卡包括 Tesla P100、Titan X、GeForce GTX 10 系列、Quadro P 系列

参考：

[https://en.wikipedia.org/wiki/Pascal\\_\(microarchitecture\)](https://en.wikipedia.org/wiki/Pascal_(microarchitecture))

#### 2 · Volta 架构

伏特微架构用于图形工作站等专业设备，采用伏特架构的产品包括 Titan V、Quadro GV100、Titan V CEO Edition，Tesla V100

参考：

[https://en.wikipedia.org/wiki/Volta\\_\(microarchitecture\)](https://en.wikipedia.org/wiki/Volta_(microarchitecture))

#### 3 · Turing 架构

采用图灵架构的产品包括 GeForce GTX 16 系列、GeForce RTX 20 系列、Quadro RTX 系列、Tesla T4

参考：

[https://en.wikipedia.org/wiki/Turing\\_\(microarchitecture\)](https://en.wikipedia.org/wiki/Turing_(microarchitecture))

## 4 · Ampere 架构

采用安培架构的产品包括 GeForce RTX 30 系列、A100（属于之前的 Tesla 产品线，但不再使用 Tesla 这个名字）

参考：

[https://en.wikipedia.org/wiki/Ampere\\_\(microarchitecture\)](https://en.wikipedia.org/wiki/Ampere_(microarchitecture))

## (四) NVidia 产品系列及型号

NVIDIA 将显示核心分为三大系列：

- GeForce 系列：用于提供家庭娱乐
- Quadro 系列：用于专业绘图设计
- Tesla 系列：用于大规模的并联电脑运算，从基于 Ampere 微架构的 A100 开始弃用了 Tesla 这个系列名称，改用同微架构一样的 Ampere 作为产品名称

### 混淆注意！

1. 这里的 Tesla 是产品系列的名称，请注意不要和 NVidia 的 GPU 微架构 Tesla 混淆。之所以使用 Tesla 的原因应该是从 Tesla 微架构开始使用统一着色器架构。
2. Tesla 的产品先后采用了若干种微架构，比如 P100（Pascal 微架构），V100（Volta 微架构），T4（Turing 微架构）
3. 从 A100（Ampere 微架构）开始，不再使用 Tesla 作为其产品名称，而改为使用微架构名称 Ampere 作为产品名称。

## 1 · GeForce

GeForce 系列是 NVidia 推出的针对桌面和移动个人电脑的 GPU。其首个型号是 1999 年推出的 GeForce 256，它之前的型号为 Riva TNT 2。历年推出过的型号包括：

- GeForce 256：1999 年推出
- GeForce 2 系列：2000 年，包括 GeForce 2 MX、GTS、Pro、Ti 等型号
- GeForce 3 系列：2001 年，包括 GeForce 3、GeForce 3 Ti 等型号
- GeForce 4 系列：2002 年，包括 GeForce MX 4x0、PCX 4xx0、Ti 4xx0 等型号
- GeForce FX 系列：2003 年，包括 GeForce FX 5xx0 和 PCX 5xx0 等型号
- GeForce 6 系列：2004 年，包括 GeForce 6xx0 等型号

- **GeForce 7 系列**：2005 年，包括 GeForce 7xx0 等型号
- **GeForce 8 系列**：2006 年，包括 GeForce 8xx0 等型号
- **GeForce 9 系列**：2008 年，包括 GeForce 9xx0 等型号
- **GeForce 200 系列**：2008 年，包括 GeForce 2xx、G2xx、GTS2xx、GTX2xx
- **GeForce 100 系列**：2009 年，系 GeForce 9 系列重命名而成，包括 G1xx、G1xxM、GT1xx、GT1xxM、GTS1xx、GTS1xxM
- **GeForce 300 系列**：2009 年，也是之前的 GPU 重命名，包括 GeForce 3xx、3xxM、GT3xx、GT3xxM、GTS3xxM
- **GeForce 400 系列**：2010 年，包括 GeForce 4xx、4xxM、GT4xx、GT4xxM、GTS4xx、GTX4xx、GTX4xxM
- **GeForce 500 系列**：2010 年，包括 GeForce 5xx、GT5xx、GT5xxM、GT5xxMX、GTX5xx、GTX5xxTi、GTX5xxM
- **GeForce 600 系列**：2012 年，包括 GeForce GT6xx、GTX6xx、GTX6xxTi
- **GeForce 700 系列**：2013 年，包括 GeForce GT7xx、GTX7xx、GTX7xxTi、GTX Titan 旗舰系列
- **GeForce 800M 系列**：2014 年，包括 GeForce 8xxM、GTX8xxM
- **GeForce 900 系列**：2014 年，包括 GeForce GTX9x0、GTX9x0M、GTX9x0Ti、NVIDIA Titan X
- **GeForce 10 系列**：2016 年，包括 GeForce GT10x0、GTX10x0、GTX10x0Ti
- **GeForce 20 系列**：2018 年，包括 GeForce RTX20x0、RTX20x0 Super、RTX 20x0Ti、Titan RTX
- **GeForce 16 系列**：2019 年，是 20 系列去光线追踪功能的弱化版本，型号包括 GeForce GTX16xx、GTX16xx Super、GTX16xxTi
- **GeForce 30 系列**：2020 年，包含 GeForce RTX 30x0

## (1) GeForce 900 系列

GeForce 900 系列 GPU 发布于 2014 年，采用第二代 Maxwell 微架构。

900 系列支持 DirectX12，OpenGL4.6，OpenCL1.2，Vulkan1.2。下表列出的部分 900 系列 GPU 制程工艺均为 28 纳米，显存均为 GDDR5，总线接口均为 PCIe 3.0 x16：

型号	核心代号	晶体管数 & 晶粒面积	核心配置		时钟频率 MHz	填充率		显示存储器		运算性能 (GFLOPS)			TDP (W)	SLI 支持	
			核心配置 SPs:TMUs:ROPs	二级缓存 (MB)		像素 (GP/s)	材质 (GT/s)	容量 (GB)	带宽 (GB/s)	接口带宽 (bit)	单精度 (加速)	双精度 (加速)	半精度 (加速)		
GeForce GTX 950	GM206	29.4 亿 227mm <sup>2</sup>	768:48:32 (6 SMM)	1	1024/ 1188	32.7	49.2	2,4	106	128	1572	49.1		90	2 路
GeForce GTX 960			1024:64:32 (8 SMM)		1127/ 1178	39.3	72.1	2,4	112		2308	72.1		120	2 路
GeForce GTX 970	GM204	52 亿 398mm <sup>2</sup>	1664:104:56 (13 SMM)	1.75	1050/ 1178	54.6	109.2	3.5 +0.5	196 +28	224	3494	109		145	3 路
GeForce GTX 980			2048:128:64 (16 SMM)	2	1126/ 1216	72.1	144	4	224	256	4612	144		165	4 路

GeForce GTX TITAN X	GM200	80 亿 601mm <sup>2</sup>	3072:192:96 (24 SMM)	3	1000/ 1089	96	192	12	336	384	6144	192		250	
---------------------	-------	----------------------------	-------------------------	---	---------------	----	-----	----	-----	-----	------	-----	--	-----	--

## (2) GeForce 10 系列

GeForce 10 系列的 GPU 发布于 2016 年，采用了 Pascal 微架构。

支持 DirectX12，OpenGL4.6，OpenCL1.2，Vulkan1.2。下表列出的部分 10 系列 GPU 制程均为 16 纳米，总线接口均为 PCIe 3.0 x16，均支持 SLI：

型号	核心代号	晶体管数 & 晶粒面积	核心配置		时钟频率		填充率		显示存储器			运算性能 (GFLOPS)			TDP (W)			
			核心配置 SPs:TMUs:ROPs	二级缓存 (MB)	默认/ 加速 MHz	存储器 (MT/s)	像素 (GP/s)	材质 (GT/s)	容量 (GB)	带宽 (GB/s)	类型	接口带宽 (bit)	单精度 (加速)	双精度 (加速)	半精度 (加速)			
GeForce GTX 1070	GP104	72 亿 314mm <sup>2</sup>	1920:80:48 (15 SM)	2	1506/ 1683	8000	96.4	180.7	8	256	GDDR5	256	5783 (6463)	181 (202)	90 (101)	150		
GeForce GTX 1070 Ti	GP104		2432:152:64 (19 SM)		1607/ 1683		244.3	102.8					7816 (8186)	244 (256)	122 (128)	180		
GeForce GTX 1080	GP104		2560:160:64 (20 SM)		1607/1733	10000	257.1	320		8228 (8873)			257 (277)	128 (139)				
GeForce GTX 1080Ti	GP102	120 亿 471mm <sup>2</sup>	3584:224:88 (28 SM)	3	2.75	1480	11000	130.2	331.5	11	484	352	10609-11340	332-354	166-177	250		
NVIDIA TITAN X	GP102		3584:224:96 (28 SM)		1417	10000	136	317.4	12	480	384		10157 (10974)	317 (343)	159 (171)			
NVIDIA TITAN Xp	GP102		3840:240:96 (30 SM)		1405	11410	135	337.2		547.7			10790 (12150)	337 (380)	169 (190)			

## (3) GeForce 16 系列

GeForce 16 系列的 GPU 发布于 2019 年，采用 Turing 微架构，16 系列在 20 系列之后推出，与 20 系列最大的区别在于取消了 RT 核和张量核心。

支持 DirectX12，OpenGL4.6，OpenCL1.2，Vulkan1.2。下表列出的部分 16 系列 GPU 制程均为 12 纳米，总线接口均为 PCIe 3.0 x16：

型号	核心代号	晶体管数 & 晶粒面积	核心配置		时钟频率		填充率		显示存储器			运算性能 (GFLOPS)			TDP (W)		
			核心配置 SPs:TMUs:ROPs	一级缓存 (KB)	二级缓存 (MB)	默认/ 加速 MHz	存储器 (MT/s)	像素 (GP/s)	材质 (GT/s)	容量 (GB)	带宽 (GB/s)	类型	接口带宽 (bit)	单精度 (加速)	双精度 (加速)		
GTX 1650	TU117	47 亿 200mm <sup>2</sup>	896:56:32 (14 SM)	896	1	1485/ 1665	8000	53.28	93.24	4	128	GDDR5	128	2661 (2984)	83.16 (93.24)	5322 (5967)	75
GTX 1650 (GDDR6)						1410/ 1590	12000	50.88	89.04		192			2527 (2849)	79 (89)	5053 (5699)	

<b>GTX 1650 (TU106)</b>	TU106	108 亿 445mm2														90		
<b>GTX 1650 (TU116)</b>	TU116															75		
<b>GTX 1650 Super</b>	TU116		1280:80:32 (20 SM)	1280		1530/ 1725		55.2	110.4						3916 (4416)	122 (138)	7832 (8832)	100
<b>GTX 1660</b>							8000											120
<b>GTX 1660 Super</b>	TU116	66 亿 284mm2	1408:88:48 (22 SM)	1408	1.5	1530/ 1785	14000	73	135	6	336	GDDR5	192	4308 (5027)	135 (157)	8616 (10053)	125	
<b>GTX 1660 Ti</b>	TU116		1536:96:48 (24 SM)	1536		1500/ 1770	12000	88.6	177.1		288	GDDR6		4608	144	9216	120	

## (4) GeForce 20 系列

GeForce 20 系列的 GPU 发布于 2018 年，采用 **Turing 微架构**，具有实时光线追踪功能（Ray Tracing），通过使用 RT（Ray Tracing）核心可以加速这一过程。

支持 DirectX12，OpenGL4.6，OpenCL1.2，Vulkan1.2。下表列出的部分 16 系列 GPU 制程均为 **12 纳米**，显存类型均为 **GDDR6**，总线接口均为 **PCIe 3.0 x16**

其中只有 2080 系列支持 2 路 NVLink（用于连接 CPU 和 GPU，或多个 GPU 之间的连接）：

型号	晶体管数 & 晶粒面积	核心配置				时钟频率		填充率		显示存储器			运算性能 (GFLOPS)			光线追踪性能		
		核心配置 SPs:TMUs:ROPs	光线 追踪 核心	张量 核心	二级 缓存 (MB)	默认/ 加速 MHz	存储器 (MT/s)	像素 (GP/s)	材质 (GT/s)	容量 (GB)	带宽 (GB/s)	接口 带宽 (bit)	单精度 (加速)	双精度 (加速)	半精度 (加速)	每秒 光线数 (十亿)	RTX- OPS (万亿)	张量 浮点 (万亿)
RTX 2060	108 亿 445mm2	1920:120:48 (30 SM)	30	240	3	1365/ 1680	14000	65.52	163.8	6	336	192	5242 (6451)	164 (202)	10483 (12902)	5	37	51.6
RTX 2060 Super		2176:136:64 (34 SM)	34	272	4	1470/ 1650		90.5	191.4	8	448	256	6123 (7181)	191 (224)	12246 (14362)	6	41	57.4
RTX 2070		2304:114:64 (36 SM)	36	288		1410/ 1620		90.24	203.04				6497 (7465)	203 (233)	12994 (14930)		45	59.7
RTX 2070 Super	136 亿 545mm2	2560:160:64 (40 SM)	40	320		1605/ 1770	15500	102.72	256.8				8218 (9062)	257 (283)	16435 (18125)	7	52	72.5
RTX 2080		2944:184:64 (46 SM)	46	368	4	1515/ 1710		96.96	278.76				8920 (10068)	279 (315)	17840 (20137)	8	60	80.5
RTX 2080 Super		3072:192:64 (46 SM)	48	384		1650/ 1815		105.6	316.8				10138 (11151)	317 (349)	20275 (22303)		63	89.2
RTX 2080 Ti	186 亿 745mm2	4352:272:88 (68 SM)	68	544	5.5	1350/ 1545	14000	118.8	367.2	11	616	352	11750 (13448)	367 (421)	23500 (26896)	10	78	107.6
Titan RTX		4608:288:96 (72 SM)	72	576	6	1350/ 1770		129.6	388.8	24	672	384	12442 (16312)	389 (510)	24884 (32625)	11	84	130.5

## (5) GeForce 30 系列

GeForce 30 系列于 2020 年 9 月推出，采用了 **Ampere 微架构**。其性能是之前的 RTX 20 系列的两倍。

30 系列用的制程是三星的 **8 纳米工艺**，总线接口是 **PCIe 4.0 x16**：

型号	核心配置	时钟频率				显示存储器				运算性能 (GFLOPS)			TDP (W)	NVLink
		SPs:TMUs	默认 (MHz)	加速 (MHz)	存储器 (MT/s)	容量 (GB)	带宽 (GB/s)	类型	接口 带宽 (bit)	单精度 (加速)	双精度 (加速)	半精度 (加速)		
RTX 3070	5888:368	1500	1730	16000	8	512	GDDR6	256	17664 (20372)	552 (637)	35328 (40745)	220	否	

<b>RTX 3080</b>	8704:544	1440	1710	19000	10	760	GDDR6X	320	25068 (29768)	783 (930)	50135 (59535)	320	否
<b>RTX 3090</b>	10496:656	1400	1700	19500	24	936	GDDR6X	384	29389 (35686)	918 (1115)	58778 (71373)	350	2 路

## 2 · Quadro

Quadro 系列定位于专业绘图工作站领域，用于运行专业的 CAD ( Computer-Aided Design ) 、 CGI ( Computer-Generated Imaginary ) 、 DCC ( Digital Content Creation ) 。

多数产品的核心实质上与定位于个人领域的 GeForce 完全相同，但与 GeForce 相比 Quadro 强调与行业软件的兼容性、稳定性以及高效率。其驱动程式对行业软件及编程界面有相应的优化。

- **Quadro** : 第一代产品，基于 GeForce 256，专业级图形处理器。
- **Quadro 2** : 基于 GeForce 2，Quadro 改良型专业级图形处理器。
- **Quadro DCC** : 基于 GeForce 3，针对数位内容创作。
- **Quadro 4** : 基于 GeForce 4，专业级图形处理器。
- **Quadro FX** : 针对台式电脑和行动工作站的专业级图形处理器，提供高效能的专业 3D 应用处理。例如多媒体，视像模拟等。2010 年 7 月 27 日以后，Quadro FX 系列重新划分为 Quadro 系列。
- **Quadro NVS** : 专业级商用图形处理器。提供多显示功能，例如金融交易等。2010 年 12 月 1 日以后，Quadro NVS 系列独立成为 NVS 系列。
- **Quadro CX** : 专为 Adobe Creative Suite 设计的加速器。
- **Quadro VX** : 专为满足中国 AutoCAD 专业人士的需求而设计的高性价比图形处理器。
- **Quadro Plex** : 针对最复杂的重度绘图和运算问题的专业级图形处理器。例如制造业设计，地球科学，数位内容创建等。
- **Quadro RTX** : 加入即时光影追踪 Ray tracing 技术。而且 Ray tracing 技术能够令显卡进行即时的运算，渲染与光影追踪

## 3 · Tesla

Tesla 是一个新的（相较于 GeForce）显示核心系列品牌，主要用于服务器高性能电脑运算，用于对抗 AMD 的 FireStream 系列。这是继 GeForce 和 Quadro 之后，第三个显示核心商标。

早期的 Tesla 分为三个子系列：

- C : 外形类似普通显卡，不设任何显示输出
- D : Desktop，D870 包含两张 C870 GPU，可多个设备互联
- S : Server，外形类似 1U 服务器，S870 包含四张 C870，可多个设备互联

下表列出了部分 Tesla GPU 的参数：

型号	微架构	芯片	着色器			显示存储器				运算性能(GFLOPS)		CUDA core version	TDP (W)	总线接口	
			CUDA核心(总共)	默认(MHz)	加速(MHz)	类型	接口带宽(bit)	容量(GB)	时钟频率(MHz)	带宽(GB/s)	单精度(MAD or FMA)	双精度(FMA)			
P4	Pascal	1×GP104	2560	810	1063	GDDR5	256	8	6000	192	4147–5443	129.6–170.1	6.1	50-75	PCIe
P6		1×GP104	2048	1012	1506	GDDR5	256	16	3003	192.2	6169	192.8	6.1	90	MXM
P40		1×GP102	3840	1303	1531	GDDR5	384	24	7200	345.6	10007–11758	312.7–367.4	6.1	250	PCIe
P100		1×GP100	3584	1328	1480	HBM2	4096	16	1430	732	9519–10609	4760–5304	6	300	NVLink
P100 (16 GB card)		1×GP100		1126	1303						8071–9340	4036–4670		250	PCIe
P100 (12 GB card)		1×GP100		3072	12						549	8071–9340	4036–4670		
V100 (mezzanine)	Volta	1×GV100	5120	N/A	1455	HBM2	4096	16/32	1750	900	14899	7450	7	300	NVLink
V100 (PCIe card)		1×GV100		N/A	1370						14028	7014		250	PCIe
T4 (PCIe card)	Turing	1×TU104	2560	585	1590	GDDR6	256	16	N/A	320	8100	N/A	7.5	70	PCIe

## (五) API

### 1 · DirectX

DirectX (Direct eXtension，缩写：DX) 是由微软公司创建的一系列专为多媒体以及游戏开发的应用程序接口。旗下包含 Direct3D、Direct2D、DirectCompute 等等多个不同用途的子部分，因为这一系列 API 皆以 Direct 字样开头，所以 DirectX (只要把 X 字母替换为任何一个特定 API 的名字) 就成为这一巨大的 API 系列的统称。目前最新版本为 DirectX 12，随附于 Windows 10 操作系统之上。

DirectX 被广泛用于 Microsoft Windows、Microsoft Xbox 电子游戏开发，并且只能支持这些平台。除了游戏开发之外，DirectX 亦被用于开发许多虚拟三维图形相关软件。Direct3D 是 DirectX 中最广为应用的子模块，所以有时候这两个名词可以互相代称。

DirectX 主要基于 C++ 编程语言实现，遵循 COM 架构。

- DirectX 5.0 - 雾化效果，Alpha 混合

- DirectX 6.0 - 纹理映射
- DirectX 7.0 - 硬件 T&L
- DirectX 8.0 - Shader Model 1.1
- DirectX 8.1 - Pixel Shader 1.4, Vertex Shader 1.1
- DirectX 9.0 - Shader Model 2.0
- DirectX 9.0b - Pixel Shader 2.0b, Vertex Shader 2.0
- DirectX 9.0c - Shader Model 3.0
- DirectX 9.0Ex- Windows Vista 版本的 DirectX 9.0c, Shader Model 3.0, DXVA 1.0
- DirectX 10- Shader Model 4.0, Windows Graphic Foundation 2.0, DXVA 2.0
- DirectX 10.1- Shader Model 4.1, Windows Graphic Foundation 2.1, DXVA 2.1
- DirectX 11- Shader Model 5.0, Tessellation 镶嵌技术, 多线程渲染, 计算着色器
- DirectX 12.0 - Windows 10, low-level rendering API, GPGPU

## 2 · OpenGL

OpenGL (Open Graphics Library, 开放图形库) 是用于渲染 2D、3D 矢量图形的跨语言、跨平台的应用程序编程接口 (API)。这个接口由近 350 个不同的函数调用组成，用来从简单的图形比特绘制复杂的三维景象。

而另一种程序接口系统是仅用于 Microsoft Windows 上的 Direct3D。OpenGL 常用于 CAD、虚拟现实、科学可视化程序和电子游戏开发。

OpenGL 的高效实现 (利用图形加速硬件) 存在于 Windows, 部分 UNIX 平台和 Mac OS。这些实现一般由显示设备厂商提供，而且非常依赖于该厂商提供的硬件。

- OpenGL 1.1 - 纹理对象
- OpenGL 1.2 - 3D 纹理, BGRA 压缩像素格式
- OpenGL 1.3 - 多重渲染, 多重采样, 纹理压缩
- OpenGL 1.4 - 深度纹理
- OpenGL 1.5 - 物体顶点缓冲, 遮面查询
- OpenGL 2.0 - GLSL 1.1, 多渲染目标, 可编程着色语言, 双面模板
- OpenGL 2.1 - GLSL 1.2, 物体像素缓冲, sRGB 纹理
- OpenGL 3.0 - GLSL 1.3, 纹理阵列, 条件渲染, FBO
- OpenGL 3.1 - GLSL 1.4, 纹理缓冲对象, 统一缓冲对象, 符号正常化纹理, 基本元素重启, 实例化, 拷贝缓冲接口
- OpenGL 3.2 - GLSL 1.5, 着色器可直接处理纹理采样, 改进管线可编程设计性
- OpenGL 3.3 - GLSL 3.3, 同 OpenGL 4.0, 大量新的 ARB 扩展, 使 OpenGL 3.x 级别硬件尽可能多的支持 OpenGL 4.x 级别硬件的特性
- OpenGL 4.0 - GLSL 4.0, 两种新的着色阶段, 增加渲染质量和反锯齿灵活性, 数据绘图由外部 API 负责, 加强 GPU 通用计算, 64 位双精度浮点着色器
- OpenGL 4.1 - GLSL 4.1, 支持着色器二进制信息提取和加载, 64 位浮点组件支持顶点着色器输入, 完全兼容于 OpenGL ES 2.0 APIs
- OpenGL 4.2 - GLSL 4.2, 允许多种操作的 Shader 同处在一个级别的纹理单元内, 捕

捉 GPU 细分几何图形，绘制多个实例用来改变反馈结果，一个 32bit 精度的数值可以包含多个 8bit 和 16bit 精度数值

- OpenGL GLSL 着色器 4.3 - 4.30 利用 GPU 的并行计算，着色器存储缓冲区对象，高质量的工作 / EAC 纹理压缩，提高存储的安全，多应用鲁棒性扩展
- OpenGL 4.4 GLSL 4.40 缓冲配置控制，高效异步查询，着色器变量布局，高效的多目标约束，流线型的 Direct3D 应用程序的移植，无纹理延伸，稀疏纹理扩展
- OpenGL GLSL 4.5 - 4.50

### 3 · OpenGL ES

是 OpenGL 的子集，用于移动设备

### 4 · Mesa

Mesa 3D 是一个在 MIT 许可证下开放源代码的三维计算机图形库，以开源形式实现了 OpenGL 的应用程序接口。

OpenGL 的高效实现一般依赖于显示设备厂商提供的硬件，而 Mesa 3D 是一个纯基于软件的图形应用程序接口。由于许可证的原因，它只声称是一个“类似”于 OpenGL 的应用程序接口。

### 5 · Vulkan

Vulkan 是一个低开销、跨平台的二维、三维图形与计算的应用程序接口（API），最早由科纳斯组织在 2015 年游戏开发者大会（GDC）上发表。与 OpenGL 类似，Vulkan 针对全平台即时 3D 图形程序（如电子游戏和交互媒体）而设计，并提供高性能与更均衡的 CPU 与 GPU 占用，这也是 Direct3D 12 和 AMD 的 Mantle 的目标。与 Direct3D（12 版之前）和 OpenGL 的其他主要区别是，Vulkan 是一个底层 API，而且能执行并行任务。除此之外，Vulkan 还能更好地分配多个 CPU 核心的使用。

# 十五、边缘计算

本章介绍在边缘计算中，即各种手机、移动设备、嵌入式设备、物联网设备中如何使用神经网络。包括：

- 移动设备上的计算单元
- 各公司的移动计算框架
- ARM 相关介绍

## (一) 达芬奇 NPU (华为) (TODO)

## (二) 寒武纪 (寒武纪)

国内的 AI 芯片公司及其产品

## (三) TensorRT (NVIDIA)

TensorRT 是 NVIDIA 推出的针对自己产品做的加速包。

官网：

<https://developer.nvidia.com/tensorrt>

参考：

<https://zhuanlan.zhihu.com/p/64933639>

## (四) ncnn (腾讯)

ncnn 是腾讯开发的移动设备运行库，类似 TensorFlow Lite，但 ncnn 是个 c++ 的库，只支持 NDK。Ncnn 库中提供了 MXNet、Caffe 和 ONNX 的模型转换到 ncnn 的工具。

ncnn 是一个为手机端极致优化的高性能神经网络前向计算框架。ncnn 从设计之初深刻考虑手机端的部署和使用。无第三方依赖，跨平台，手机端 cpu 的速度快于目前所有已知的开源框架。

github：

<https://github.com/Tencent/ncnn>

## (五) TNN (腾讯)

TNN 是腾讯最新推出的新一代开源移动端推理框架

github :

<https://github.com/Tencent/TNN>

## (六) MNN (阿里)

MNN 是阿里推出的移动端神经网络推理框架

github :

<https://github.com/alibaba/MNN>

doc :

<https://www.yuque.com/mnn/en/usage>

## (七) mace (小米)

github :

<https://github.com/XiaoMi/mace>

## (八) Paddle-Lite (百度)

github :

<https://github.com/PaddlePaddle/Paddle-Lite>

## (九) Google

# 1 · TensorFlow Lite (Google)

2017 年 5 月，Google 宣布从 Android Oreo 开始，提供一个专用于 Android 开发的软件栈 TensorFlow Lite。

官网：

<https://www.tensorflow.org/lite>

TensorFlow Lite 主要由两个组件构成，解释器（Interpreter）和转换器（Converter）。

## (1) 基本工作流程

1. 选择模型：选择一个自定义的 TensorFlow 网络模型，也可以是 TF Lite 里自带的预训练网络 (<https://www.tensorflow.org/lite/models>)
2. 转换模型：如果是自定义的网络，使用转换器转换成 TF Lite 的格式 (.tflite)
3. 部署到设备：通过 TF Lite Interpreter 在设备上运行这个模型，可以通过多种语言的 API 实现。
4. 优化模型：通过模型优化工具对模型进行优化：在尽量不影响准确率的情况下缩减尺寸、提高

[https://www.tensorflow.org/lite/performance/model\\_optimization](https://www.tensorflow.org/lite/performance/model_optimization)

## (2) Converter

转换器（Converter）可以将一些框架下的网络模型转换成解释器可以使用的格式 (.tflite 文件)，转换器可以通过命令行（toco，tfLite\_convert）或者 Python 接口进行调用。

Converter 可以转换几种格式的数据为 tflite 文件：

- TensorFlow GraphDef (.pb) 文件
- TensorFlow SavedModel
- Keras 模型 (.h5) 文件
- TensorFlow 2.0 Concrete Function

简单命令：

`tfLite_convert --keras_model_file=MODEL.h5 --output_file=MODEL.tflite`

参考：

[https://tensorflow.google.cn/lite/convert/cmdline\\_reference](https://tensorflow.google.cn/lite/convert/cmdline_reference)

<https://www.tensorflow.org/lite/convert>  
<https://www.tensorflow.org/lite/r2/convert>

### (3) Interpreter

解释器（Interpreter）通过以下步骤来运行模型：

1. 加载模型：将描述网络模型的 .tflite 文件加载到内存中
2. 数据转换：将输入数据转换为网络模型认可的格式，比如图像的缩放
3. 运行模型：
4. 解释输出：将模型输出的张量转换为应用程序可以利用的格式

解释器可以以库的形式运行在安卓、iOS 和 Linux 上。

参考：

<https://www.tensorflow.org/lite/guide/inference>

### (4) 模型量化

TF1lite 和 Tensorflow 提供了量化（Quantization）这个方法/工具来减小模型的复杂度。

量化通过降低模型权重的精确度（从 32 位 float 到 16 位 float，或者 8 位整数）来实现对推理时间、模型大小、读写时间的降低，其代价是模型准确性的下降。许多 CPU 等硬件提供 SIMD 指令的支持，可以大幅提高量化后模型的运算时间。

目前量化有两种方式：

- 训练后量化（Post-training quantization）：对已有的训练好的模型进行量化。
- 量化训练（Quantization-aware training）：由于在训练时就针对量化做优化，可以使得模型的准确性下降最小，但这只适用于部分 CNN 架构。

#### ①训练后量化

训练后量化针对一个已经训练好的模型进行转换，有三种转换方式：

方式	优化结果	硬件
权重量化 Weights quantization	模型大小变为 1/4 2-3 倍速度提升	CPU
全整数量化 Full integer quantization	模型大小变为 1/4 3 倍以上速度提升	CPU, Edge TPU 等

16 位浮点量化 Float 16 quantization	模型大小变为 1/2 潜在的 GPU 加速	CPU/GPU
-----------------------------------	--------------------------	---------

### 1) 权重量化：

Weights quantization，将权重量化为 8 位整数，代码如下：

```
import tensorflow as tf

converter = tf.lite.TFLiteConverter.from_saved_model(saved_model_dir)

converter.optimizations = [tf.lite.Optimize.OPTIMIZE_FOR_SIZE]

tflite_quant_model = converter.convert()
```

权重量化也可以通过执行 converter 时增加参数实现：

```
tflite_convert \
--output_file=/tmp/foo.tflite \
--graph_def_file=/tmp/some_quantized_graph.pb \
--inference_type=QUANTIZED_UINT8 \
--input_arrays=input \
--output_arrays=MobilenetV1/Predictions/Reshape_1 \
--mean_values=128 \
--std_dev_values=127
```

### 2) 全整数量化

Full integer quantization of weights and activations，将权重和激活函数量化为 8 位整数，代码如下：

```
import tensorflow as tf

def representative_dataset_gen():

 for _ in range(num_calibration_steps):

 # Get sample input data as a numpy array in a method of your choosing.

 yield [input]
```

```
converter = tf.lite.TFLiteConverter.from_saved_model(saved_model_dir)

converter.optimizations = [tf.lite.Optimize.DEFAULT]

converter.representative_dataset = representative_dataset_gen

tflite_quant_model = converter.convert()
```

### 3) 16 位浮点量化

Float16 quantization of weights，将权重量化为 16 位浮点，代码如下：

```
import tensorflow as tf

converter = tf.lite.TFLiteConverter.from_saved_model(saved_model_dir)

converter.optimizations = [tf.lite.Optimize.DEFAULT]

converter.target_spec.supported_types = [tf.lite.constants.FLOAT16]

tflite_quant_model = converter.convert()
```

## (5) 预训练模型

TF Lite 官网自带了五种功能的预训练网络模型：

- **图像分类 (Image Classification)**：给图像分类，包括人、动物、植物、活动、地点等。模型会输出一组 0 到 1 之间的数字（和为 1），表示各种类别的可能性。图像分类的缺省模型是 MobileNetV1。  
([https://www.tensorflow.org/lite/models/image\\_classification/overview](https://www.tensorflow.org/lite/models/image_classification/overview))
- **目标检测 (Object Detection)**：检测图像中的若干物体，用矩形框标示出位置。模型的输出为一组数据，其中每个数据由类别、得分（置信度）、位置三部分构成，位置则包括上、下、左、右四个数据。目标检测的缺省模型是用 coco 数据集训练的 MobileNetV1。  
([https://www.tensorflow.org/lite/models/object\\_detection/overview](https://www.tensorflow.org/lite/models/object_detection/overview))
- **智能问答 (Smart Reply)**：智能问答可以根据聊天内容自动生成回复，而且是上下文相关的。  
([https://www.tensorflow.org/lite/models/smart\\_reply/overview](https://www.tensorflow.org/lite/models/smart_reply/overview))

- **姿势评估 (Pose Estimation)** : 评估图像/视频中人的姿势，或者说是定位其中人的各个身体节点（包括鼻、眼、耳、肩、肘、腕、臂、膝、踝）的位置。姿势评估用的模型是 PoseNet。  
([https://www.tensorflow.org/lite/models/pose\\_estimation/overview](https://www.tensorflow.org/lite/models/pose_estimation/overview))
- **图像分割 (Segmentation)** : 图像分割有点类似目标检测，其区别在于标示的方式不是矩形框，而是物体的准确轮廓。图像分割用的缺省模型是 DeepLabV3。  
(<https://www.tensorflow.org/lite/models/segmentation/overview>)

其他已经在 TF Lite 中可以工作的预训练好的模型：

[https://www.tensorflow.org/lite/guide/hosted\\_models](https://www.tensorflow.org/lite/guide/hosted_models)

或者可以从 TensorFlow Hub 上自行下载需要的模型进行转换：

<https://www.tensorflow.org/hub>

## (6) Transfer Learning

迁移学习 (Transfer Learning)，指的是在已经训练好的网络上做适度训练，以便解决一个类似的问题，比如用于辨识小轿车的模型可以通过少量训练用于辨识卡车。迁移学习的好处是不用从头对网络进行训练，大大减少了训练时间和训练数据。

参考：

<https://codelabs.developers.google.com/codelabs/recognize-flowers-with-tensorflow-on-android/#1>

## (7) 交叉编译 label\_image

label\_image 是 TF Lite 里自带的一个对 tflite 的 C++ 使用样例，位于：

`tensorflow/tensorflow/lite/examples/label_image`

对其在 x86 android 上进行交叉编译的步骤如下：

- 修改 tensorflow/tensorflow/WORKSPACE 文件，增加：

```
android_ndk_repository(
 name = "androidndk", # Required. Name *must* be "androidndk".
 path = "/home/xxx/ndk", # Optional. Can be omitted if `ANDROID_NDK_HOME` environment variable is set.
)
```

其中 path 指向 ndk 的所在目录

- 在 tensorflow/ 中执行编译：

```
bazel build //tensorflow/lite/examples/label_image:label_image
--crosstool_top=//external:android/crosstool --cpu=x86
--host_crosstool_top=@bazel_tools//tools/cpp:toolchain --cxxopt="-std=c++11"
可进行在 x86 android 上的交叉编译，如果编译中出现 std::round 相关的错误，去掉 std
即可。
```

- 将程序和 mobilenet 网络模型上传到设备上的同一个目录下：

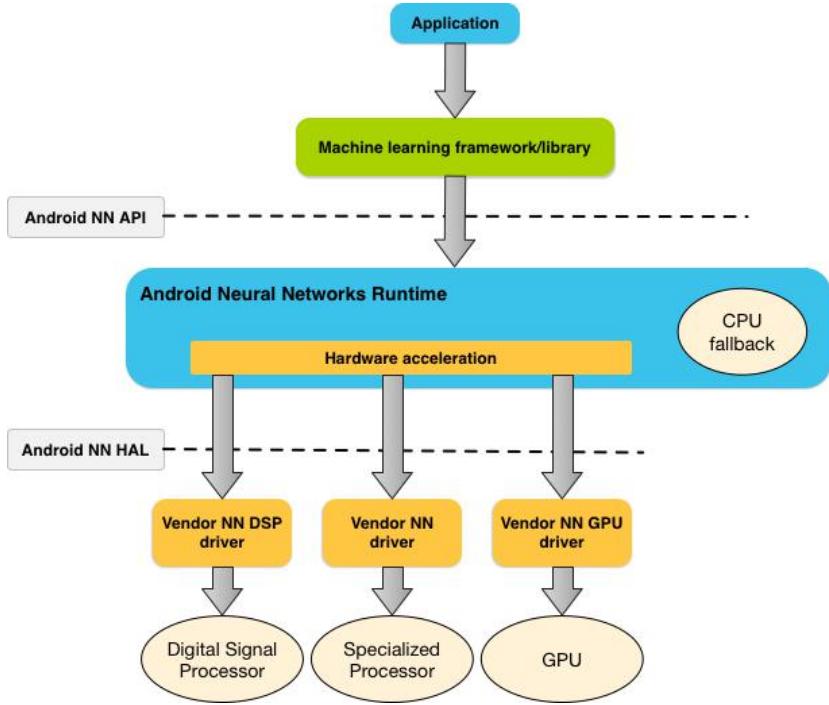
- tensorflow/tensorflow/bazel-  
bin/tensorflow/lite/examples/label\_image/label\_image
- examples/lite/examples/image\_classification/android/app/src/main/assets  
下的 tflite 网络模型文件和 label 文件

## 2 · NNAPI (Google)

NNAPI (Neural Network API) 是 Google 推出的 Android 的 C 语言 API，用于提高神经网络在安卓设备上的计算效率。NNAPI 位于上层的机器学习库 (TensorFlow Lite、Caffe2 等) 的下层，这些上层机器学习库在设备外搭建并训练神经网络模型，之后通过 NNAPI 在安卓设备上运行。

通常 App 不会直接调用 NNAPI，而是直接使用上层的机器学习库。而这些机器学习库会通过调用 NNAPI 来实现加速。

基于 App 的需求和硬件的条件，NNAPI runtime 会机动的分配计算载荷到设备的计算部件上，这些计算部件可以是专用的 NN 硬件、GPU 以及 DSP，对于没有这些计算部件（或者驱动）的设备，NNAPI runtime 会优化代码并在 CPU 上执行。



NNAPI 和 ARMNN 的相同之处在于，都是为了将上层的神经网络框架和下层的嵌入式硬件桥接起来，区别在于 NNAPI 仅为 Android 提供，下层可以由多种硬件实现。而 ARMNN 仅封装下层的 ARM 硬件，而可以供给不同的上层使用。

参考：

<https://developer.android.com/ndk/guides/neuralnetworks>

## (十) Facebook

### 1 · Pytorch Mobile (Facebook)

Pytorch Mobile 是 Facebook 推出的移动计算框架，支持 QNNPACK 和 ARM CPU。

官网：

<https://pytorch.org/mobile/home/>

### 2 · QNNPACK (Facebook)

Quantized Neural Network PACKage，Facebook 为移动计算优化的开源高性能内核库

```
github :
https://github.com/pytorch/QNNPACK
```

## (十一) ARM

ARM，Advanced RISC Machine，英国的一家公司，及其推出的 RISC 体系架构处理器家族，适用于移动通信领域。ARM 本身不生产销售芯片，而是销售自己的设计授权。客户可以选择：

- IP Core 授权：直接购买 ARM 设计的核心，并基于此设计 SoC
- 指令集架构授权：基于某版本的架构自己设计核心，并基于此设计 SoC。

### 1 · 指令集/架构/核心

首先来清晰一下架构的概念。

在谈到体系结构时，有若干种/层分类方式，非常绕人，其中有很多都可以被叫做“架构（architecture）”，从 ARM 这个体系结构本身，到最终手机/移动设备上的主板。而这些架构的概念，是有概念上和层次上的区分的。目前并没有一个通用的名词列表去定义这些概念，以下的名词定义和解释主要基于 ARM 官网和维基百科。

#### 混淆注意！

1. 请注意 ARMv7 和 ARM7 的区别，前者是指令集版本（或者称为 ARM 架构版本），后者是指一个核心家族或者核心（Core Family 或者 Core）
2. 核心的划分被分为两个阶段，ARMv6 之前官方核心是按照数字划分的（ARM1-ARM11），ARMv7 之后被称为 Cortex 核心

下面是具体内容：

- RISC 和 CISC 架构，这是 CPU 的一个二分类方式，RISC（Reduced Instruction Set Computing，精简指令计算机）的代表就是 ARM，此外还有 MIPS、PowerPC、RISC-V、SPARC 等。CISC（Complex Instruction Set Computing，复杂指令计算机）的代表则是 x86 体系结构，此外还有 Motorola 68000、PDP-10、System Z 等。
- 各种体系结构，比如 ARM 体系结构（ARM Architecture），这里指整个 ARM 处理器家族。或者著名的 x86 体系结构
- ARM 架构（ARM Architecture）划分，也被称为指令集版本或者指令集架构，目前版本从 ARMv1 到 ARMv8。
  - ARMv6 之前的部分指令集版本又有不同的子版本，比如 ARMv5 有 ARMv5TE、ARMv5TEJ 两个子版本。

- **Profile**，在 ARMv6 之前并没有做此种划分，因此被归为 Classic，ARMv7 开始（以及 ARMv6-M）开始将 ARM 架构分为三种，即 A (Application) ，R (Real-time) ，M (Microcontroller) ，比如 ARMv6-M，ARMv7-M，ARMv7-A，ARMv8-R，ARMv8-A 等等。
- **核心家族 (Core Family)**，处理器核心的划分，分为官方 (ARM Holdings) 和第三方（比如三星、高通等）。
  - 官方核心的划分在 ARMv6 之前跟指令集版本部分对应，也有一定的交错（比如 ARM7，ARM9、ARM11）。
    - ◆ 根据不同的配置又有细分子类，比如 ARM7 可分为 ARM7、ARM7T、ARM7EJ 三个子类
      - T 表示支持 Thumb 指令
      - J 表示支持 Jazzele
      - D 表示支持 JTAG Debug
      - M 表示支持 fast Multiplier
      - I 表示支持 enhanced ICE
      - S 表示 Synthesizable
    - ARMv7 之后则按照 Profile 分为 Cortex-A、Cortex-R、Cortex-M。
    - 第三方设计的 Core 指的是其他公司在拿到指令集架构授权之后自己设计的 Core，苹果、三星、都有做第三方设计
- **片上系统 (SoC)**，这个才是真正能在手机主板上看到的封装好的芯片，功能类似 PC 中的主板，其中主处理器即是上条中的 Core。和 x86 架构的 PC 不一样，考虑到便携性，手机等移动设备的集成度更高，一个 SoC 里不仅包含了处理器，还有很多外设比如 DMA 控制器、LCD 控制器、内存控制器、摄像头接口、Flash 等等。

以下是详细的指令集版本和不同 Core Family，以及所包括的 Core 的列表，可以看到两者之间相关，但也有一定交错：

指令集版本		官方 Core family	官方 Core	第三方 Core Family	第三方 Core	
ARMv1		ARM1				
ARMv2	ARMv2	ARM2	ARM2	Amber (Open Source)	Amber23 Amber25	
	ARMv2a		ARM250			
		ARM3	ARM3			
ARMv3		ARM6				
		ARM7	ARM60			
			ARM600			
ARMv3	ARMv4T		ARM610			
			ARM700			
			ARM710			
			ARM710a			
	ARM7	ARM7TDMI				
		ARM710T				
		ARM720T				
		ARM7T			ARM740T	
					ARM7TDMI-S	

		ARM9	ARM9T	ARM9TDMI			
				ARM920T			
		ARMv4	SecureCore		StrongARM(DEC)	SA-110	
			SC100			SA-1100	
	ARMv5	ARM8		ARM810		FA510	
						FA526	
	ARMv5TE	ARM10	ARM10E	ARM1020E	Faraday (Faraday)	FA626	
						FA606TE	
		ARM9	ARM9E	ARM946E-S ARM966E-S ARM968E-S ARM996HS	XScale (Intel/Marvell)	FA626TE	
						FMP626TE	
		ARMv5TEJ		ARM926EJ-S		FA726TE	
	ARMv6	ARM7	ARM7EJ	ARM7EJ-S			
		ARM10	ARM10E	ARM1026EJ-S			
		ARMv6-M	ARM11		ARM1136J(F)-S		
					ARM1156T2(F)-S		
			Cortex-M		ARM1176JZ(F)-S		
			SecureCore		ARM11MPCore		
	ARMv7-M	SecureCore		SC000			
		Cortex-M		Cortex-M0			
				Cortex-M0+			
	ARMv7E-M	Cortex-M		Cortex-M1			
		Cortex-M3					
	ARMv7-R	Cortex-R		Cortex-M4			
				Cortex-M7			
				Cortex-R4			
				Cortex-R5			
				Cortex-R7			
				Cortex-R8			

ARMv7-A	Cortex-A (32bit)	Cortex-A5	Snapdragon (Qualcomm)	Scorpion Krait
		Cortex-A7		
		Cortex-A8		
		Cortex-A9	Ax (Apple)	A6(Swift)
ARMv8-A	Cortex-A (64bit)	Cortex-A12		
		Cortex-A15		
		Cortex-A17		
		Cortex-A32	Snapdragon (Qualcomm)	Kryo
		Cortex-A34	Denver (NVidia)	Denver
		Cortex-A35	K12 (AMD)	K12
		Cortex-A53	Exynos (Samsung)	M1/M2 (Mongoose)
		Cortex-A57		M3(Meerkat)
		Cortex-A72	Ax (Apple)	A7(Cyclone)
		Cortex-A73		A8(Typhoon)
ARMv8	ARMv8.1-A	Cortex-A55		A9(Twister)
		Cortex-A65E		A10(Hurricane)
		Cortex-A75		A11(Monsoon)
		Cortex-A76		
		Cortex-A77		
ARMv8	ARMv8.2-A	Neoverse	NeoverseN1 NeoverseE1	Exynos (Samsung) M4(Cheetah)
		ARMv8.3-A		Ax (Apple) A12(Vortex)
				A13(Lightning)
		ARMv8-M	Cortex-M	Cortex-M23 Cortex-M33 Cortex-M35P
		ARMv8-R	Cortex-R	Cortex-R52

参考：

[https://en.wikipedia.org/wiki/List\\_of\\_ARM\\_microarchitectures](https://en.wikipedia.org/wiki/List_of_ARM_microarchitectures)

[https://en.wikipedia.org/wiki/ARM\\_architecture](https://en.wikipedia.org/wiki/ARM_architecture)

特性	ARM V8	ARM V7
指令集	64 位指令集 AArch64，并且兼容 32 位指令集 AArch32	32 位指令集 A32 和 16 位指令集 T16

特性	ARM V8	ARM V7
支持地址长度	64 位	32 位
通用寄存器	31 个 x0-x30 (64 位) 或者 w0-w30 (32 位)	15 个, r0-r14 (32 位)
异常模式	4 层结构 ELO-EL3	2 层结构 vector table
NEON	默认支持	可选支持
LAPE	默认支持	可选支持
Virtualization	默认支持	可选支持
big.LITTLE	支持	支持
TrustZone	默认支持	默认支持
SIMD 寄存器	32 个 X 128 位	32 个 X 64 位

## (1) Armv7

从 V7 版本后开始变成了 Cortex 架构。

- Cortex-A 系列：应用处理器，主要用于移动计算、智能手机、车载娱乐、自动驾驶、服务器、高端处理器等领域。时钟频率超过 1GHZ，支持 Linux、Android、Windows 等完整操作系统需要的内存管理单元 MMU。
- Cortex-R 系列：实时处理器，可用于无线通讯的基带控制、汽车传动系统、硬盘控制器等。时钟频率 200HZ 到大于 1GHZ，多数不支持 MMU，具有 MPU、Cache 和其他针对工业设计的存储器功能。响应延迟非常低，不支持完整版本的 Linux 和 Windows，支持 RTOS，
- Cortex-M 系列：微控制器处理器，时钟频率较低容易使用，应用于单片机和深度嵌入式市场。

## (2) Armv8

ARM V8 是 ARM 公司的第一款 64 位处理器架构，包括 AArch64 和 AArch32 两种主要执行状态。其中前者引入了一套新的指令集“A64”专门用于 64 位处理器，后者用来兼容现有的 32 位 ARM 指令集。目前我们看到的 Cortex-A53, Cortex-A57 (现在被 A72 替代了) 二款处理器便属于 Cortex-A50 系列，首次采用 64 位 V8 架构，是 ARM 在 2012 年下半年发布的二款产品。

参考：

[https://blog.csdn.net/weixin\\_42325069/article/details/84070376](https://blog.csdn.net/weixin_42325069/article/details/84070376)

## 2 · big.LITTLE (ARM)

**big.LITTLE** 是 ARM 公司推出的技术， “big” 处理器有更好的计算性能，用于执行对性能要求高的任务，“LITTLE” 处理器则更加节能，用于处理其他对性能没有高要求的任务。

Armv8 中的处理器组合：

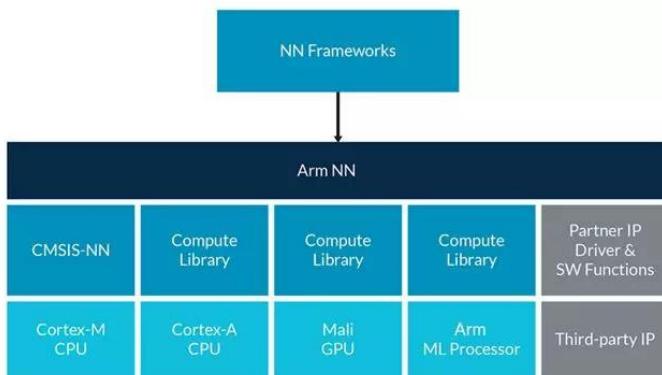
- big 处理器：Cortex-A73, Cortex-A75, Cortex-A76
- LITTLE 处理器：Cortex-A53, Cortex-A55

## 3 · ARM NN (ARM)

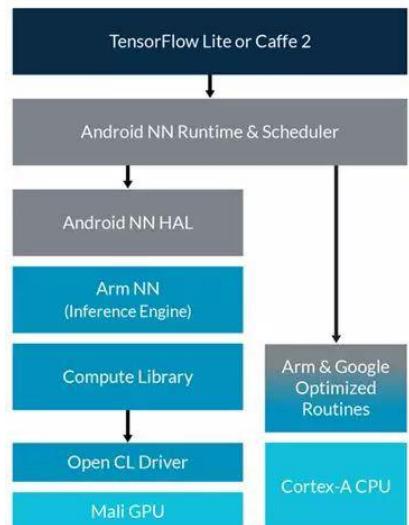
ARM NN (ARM Neural Network) 是 ARM 推出的一套开源软件/工具，使得神经网络可以高效的运行在 ARM 设备上，它桥接了上层的神经网络框架（TensorFlow，Caffe，TensorFlow Lite，ONNX）和下层的 ARM 系列硬件（Cortex CPU、Mali GPU、ARM ML 处理器）。这样，在其它主机上训练的网络，可以通过 ARM NN 在 ARM 硬件上无缝运行。



ARM NN 通过 Compute Library 使用 Cortex-A, Mali 和 ML 处理器，通过 CMSIS-NN 使用 Cortex-M 处理器：



ARM NN 也提供对谷歌的 Android NN API 的支持：



## (1) ARM Compute Library

ARM Compute Library 是一个专为 ARM 的 CPU 和 GPU 优化过的底层函数的集合，这个函数集基于 Cortex-A、Mali 和 ARM 的 ML 处理器，向外提供了图像处理、计算机视觉和机器学习的功能。

## (2) CMSIS-NN

CMSIS (Cortex Microcontroller Software Interface Standard) 是 ARM 的 Cortex-M 系列处理器提供的具体硬件无关的 HAL 层接口，其基本组件是 CMSIS-Core，除此之外还有 CMSIS-SVD、CMSIS-RTOS、CMSIS-DSP、CMSIS-Driver、CMSIS-Packs、CMSIS-DAP。

CMSIS-NN 则是其中专为神经网络的应用而提供的统一抽象接口。

## 4 · Neon(ARM)

Neon 是 ARM 提供的一个 SIMD 架构，工作在 Cortex-A 系列和 Cortex-R52 处理器上

## (十二) Vulkan

Vulkan 是一个低开销、跨平台的二维、三维图形与计算的应用程序接口（API），最早由科纳斯组织在 2015 年游戏开发者大会（GDC）上发表。与 OpenGL 类似，Vulkan 针对全平台即时 3D 图形程序（如电子游戏和交互媒体）而设计，并提供高性能与更均衡的 CPU 与 GPU 占用，这也是 Direct3D 12 和 AMD 的 Mantle 的目标。与 Direct3D（12 版之前）和 OpenGL 的其他主要区别是：

- Vulkan 是一个底层 API，而且能执行并行任务。
- Vulkan 还能更好地分配多个 CPU 核心的使用。

## (十三) NNIE(TODO)

## (十四) AidLearning

**Aid Learning FrameWork** 是一个在 Android 手机上构建了一个带图形界面的 Linux 系统，同时支持 GUI，Python 以及 AI 编程。这意味着当它安装时，你的 Android 手机拥有一个可以在其中运行 AI 程序的 Linux 系统。

现在 Aid Learning 已经完美支持 **Caffe**，**Tensorflow**，**Mxnet**，**ncnn**，**Keras**，**pytorch**，**opencv** 这些框架。此外提供了一个名为 Aid\_code 的 AI 编码开发工具。它可以通过在 Aid Learning 框架上使用 Python（支持 Python2 和 Python3）来提供可视化的 AI 编程体验。

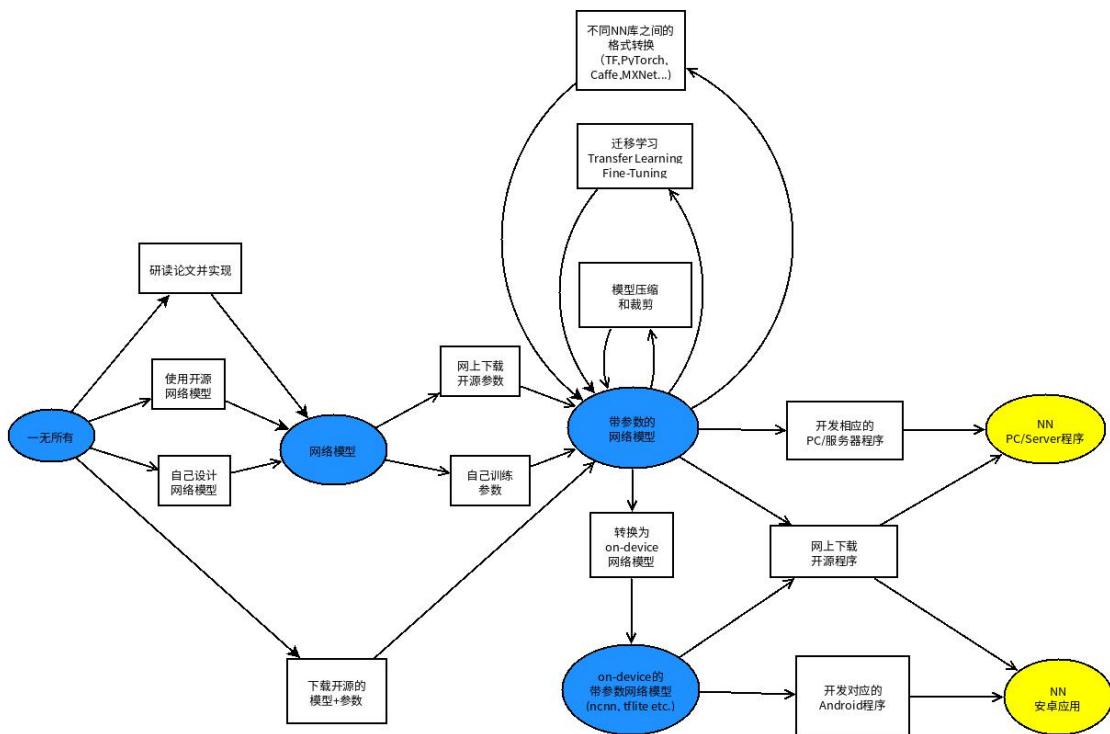
官网：<http://aidlearning.net/>

Github：<https://github.com/aidlearning/AidLearning-FrameWork>



# 十六、其它

## (一) NN 上的工作



## (二) 神经网络类型

网络类型是神经网络（NN、DNN）的最基本分类，只包含全连接层（或者叫 FC 层、Dense 层）的网络被称为多层感知机。包含卷积层的神经网络被称为卷积神经网络。包含循环层的被称为循环神经网络。

### 1 · 多层感知机（MLP）

多层感知器（Multilayer Perceptron, MLP）是一种前向结构的人工神经网络，映射一组输入向量到一组输出向量。MLP 可以被看作是一个有向图，由多个的节点层所组成，每一层都全连接到下一层。除了输入节点，每个节点都是一个带有非线性激活函数的神经元（或称处理单元）。一种被称为反向传播算法的监督学习方法常被用来训练 MLP。MLP 是感知器的推广，克服了感知器不能对线性不可分数据进行识别的弱点。

从神经网络的角度来看，MLP 就是最基本的 DNN，即都是全连接层的 DNN。

MLP 在 80 年代的时候曾是相当流行的机器学习方法，拥有广泛的应用场景，譬如语音识别、图像识别、机器翻译等等，但自 90 年代以来，MLP 遭到来自更为简单的支持向量机的强劲竞争。近来，由于深度学习的成功，MLP 又重新得到了关注。

### 2 · 卷积神经网络（CNN）

卷积神经网络（Convolutional Neural Network, CNN）是一种前馈神经网络，它的人工神经元可以响应一部分覆盖范围内的周围单元，对于大型图像处理有出色表现。

卷积神经网络由一个或多个卷积层和顶端的全连接层（对应经典的神经网络）组成，同时也包括关联权重和池化层（pooling layer）。这一结构使得卷积神经网络能够利用输入数据的二维结构。与其他深度学习结构相比，卷积神经网络在图像和语音识别方面能够给出更好的结果。这一模型也可以使用反向传播算法进行训练。相比较其他深度、前馈神经网络，卷积神经网络需要考量的参数更少，使之成为一种颇具吸引力的深度学习结构。

### 3 · 递归神经网络（RNN）

递归神经网络（RNN，Recurrent Neural Network），也叫循环神经网络，是神经网络的一种。

RNN 的定义包括两个概念，广义的定义是指宏观结构，基础 RNN、LSTM 和 GRU 用的都是这个宏观结构，狭义的是指基础 RNN 的微结构。

时间递归神经网络可以描述动态时间行为，因为和前馈神经网络（feedforward neural network）接受较特定结构的输入不同，RNN 将状态在自身网络中循环传递，因此可以接受更广泛的时间序列结构输入。手写识别是最早成功利用 RNN 的研究结果。

单纯的 RNN 因为无法处理随着递归，权重指数级爆炸或梯度消失的问题（Vanishing gradient problem），难以捕捉长期时间关联；而结合不同的 [LSTM](#) 可以很好解决这个问题。

## 4 · 生成对抗网络 (GAN)

生成对抗网络 (Generative Adversarial Network, GAN) 是非监督学习的一种方法，通过让两个神经网络相互博弈的方式进行学习。

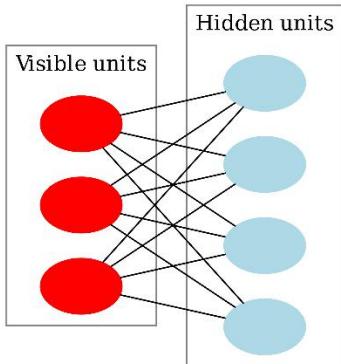
生成对抗网络由一个生成网络与一个判别网络组成。生成网络从潜在空间 (latent space) 中随机采样作为输入，其输出结果需要尽量模仿训练集中的真实样本。判别网络的输入则为真实样本或生成网络的输出，其目的是将生成网络的输出从真实样本中尽可能分辨出来。而生成网络则要尽可能地欺骗判别网络。两个网络相互对抗、不断调整参数，最终目的是使判别网络无法判断生成网络的输出结果是否真实。

生成对抗网络常用于生成以假乱真的图片。此外，该方法还被用于生成视频、三维物体模型等。

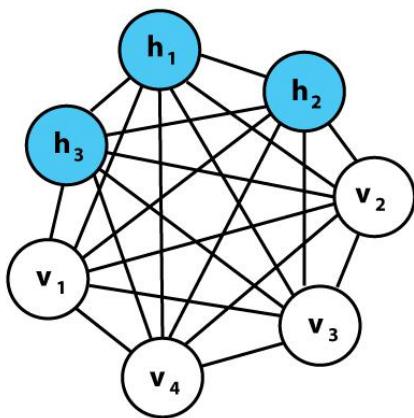
## 5 · 受限玻尔兹曼机 (RBM)

受限玻尔兹曼机 (restricted Boltzmann machine, RBM) 是一种可通过输入数据集学习概率分布的随机生成神经网络。受限玻兹曼机在降维、分类、协同过滤、特征学习和主题建模中得到了应用。根据任务的不同，受限玻兹曼机可以使用监督学习或无监督学习的方法进行训练。

受限玻兹曼机是一种玻兹曼机的变体，但限定模型必须为二分图，由一个显层 (Visible Layer) 和一个隐层 (Hidden Layer) 构成，显层和隐层之间的神经元为双向全连接。



**玻尔兹曼机（Boltzmann machine）**是随机神经网络和递归神经网络的一种，由 Geoffrey Hinton 和 Terry Sejnowski 在 1985 年发明。



## 6 · 深度置信网络 (DBN)

**深度置信网络（Deep Belief Network，DBN）**，DBN 是一个概率生成模型，与传统的判别模型的神经网络相对，生成模型是建立一个观察数据和标签之间的联合分布，对  $P(\text{Observation}|\text{Label})$  和  $P(\text{Label}|\text{Observation})$  都做了评估，而判别模型仅仅而已评估了后者，也就是  $P(\text{Label}|\text{Observation})$ 。

DBN 由多个**限制玻尔兹曼机**层组成。这些网络被“限制”为一个可视层和一个隐层，层间存在连接，但层内的单元间不存在连接。隐层单元被训练去捕捉在可视层表现出来的高阶数据的相关性。

### (三) 神经网络可视化

神经网络的可视化大体分为三种，即模型结构本身的可视化；训练的可视化；以及特征图/

卷积核的可视化。

## 1 · 模型可视化

模型可视化可以让用户更加直观的了解模型的结构

### (1) Netron

将 NN 模型可视化的开源工具，github：<https://github.com/lutzroeder/netron>，目前支持：

1. **ONNX** (.onnx, .pb, .pbtxt)
2. **Keras** (.h5, .keras)
3. **Core ML** (.mlmodel)
4. **Caffe** (.caffemodel, .prototxt)
5. **Caffe2** (predict\_net.pb, predict\_net.pbtxt)
6. **MXNet** (.model, -symbol.json)
7. **TorchScript** (.pt, .pth)
8. **NCNN** (.param)
9. **TensorFlow Lite** (.tflite)

部分支持：

1. **PyTorch** (.pt, .pth)
2. **Torch** (.t7)
3. **CNTK** (.model, .cntk)
4. **Deeplearning4j** (.zip)
5. **PaddlePaddle** (.zip, \_\_model\_\_)
6. **Darknet** (.cfg)
7. **scikit-learn** (.pk1)
8. **ML.NET** (.zip)
9. **TensorFlow.js** (model.json, .pb)
10. **TensorFlow** (.pb, .meta, .pbtxt)

### (2) Keras

#### ● `model.summary()`

通过 `model.summary()` 可以打印出 Keras 模型的基本结构及参数数量，结果如下：

- `plot_model()`

`plot_model()`函数则以一种稍微图形化一点的方式显示模型的结构，可以生成一张结构图，代码如下：

```
from keras.utils.vis_utils import plot_model
plot_model(model, to_file="xxx.png", show_shapes=True)
```

生成的结构图：

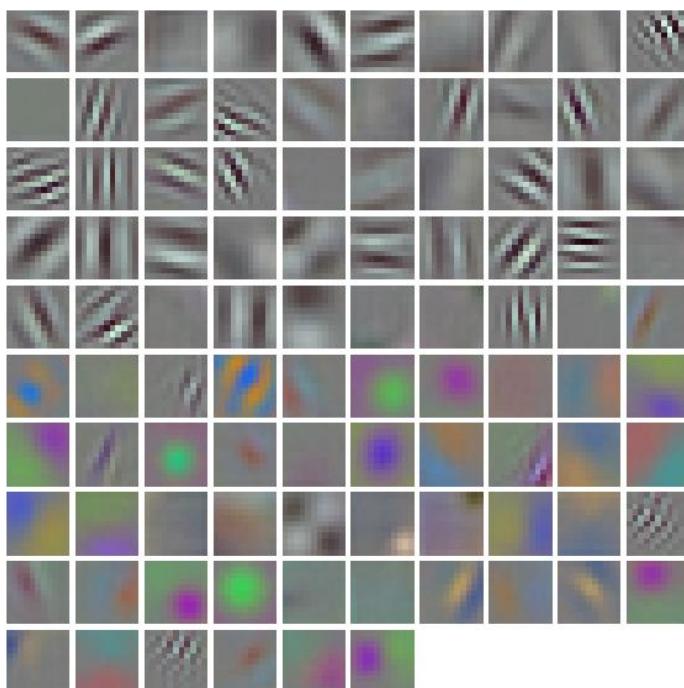
## 2 · 训练可视化

训练可视化的目的是为了帮助更好的了解训练的情况，以更快速的训练。**Tensorflow** 以及使用 tensorflow 作为后端的 **keras** 可以使用 **tensorboard** 进行训练的可视化。

## 3 · 卷积核/特征图可视化

卷积核和特征图的可视化是图像处理 NN 中会用到的一个手段。

卷积核的可视化可以看出卷积核所关心的图案，但是卷积核的可视化仅限于之前的大卷积核，在  $3 \times 3$  卷积核兴起之后就没有意义了（ $3 \times 3$  的图片能看出什么？），下面是 AlexNet 论文中作者对 AlexNet 第一层卷积核的可视化图：



虽然卷积核可视化的意义在现在已经不大，特征图可视化还是有意义的，可以让用户直观的了解“神经网络都在图像中寻找什么”

<https://github.com/keplr-io/quiver>

<https://github.com/raghakot/keras-vis>

<https://raghakot.github.io/keras-vis/>

## (四) 相关学习资料

### 1 · 视频

网易云课堂：台大李宏毅——《机器学习》2017 版

<https://study.163.com/course/introduction/1208946807.htm>

哔哩哔哩：台大李宏毅——《机器学习》2020 版

<https://www.bilibili.com/video/av94534906/>

网易云课堂：吴恩达——《机器学习》

<https://study.163.com/course/courseMain.htm?courseId=1004570029>

### 2 · 课程

[威斯康辛大学《机器学习导论》2020 秋季课程](#)

### 3 · 电子书

当当云阅读：Francois Chollet——《Python 深度学习》（付费）

<http://e.dangdang.com/pc/reader/index.html?id=1901100618>

当当云阅读：斋藤康毅——《深度学习入门》（付费）

<http://e.dangdang.com/pc/reader/index.html?id=1901100563>

github：Ian Goodfellow、Yoshua Bengio、Aaron Courville——《Deep Learning》中文

<https://github.com/exacity/deeplearningbook-chinese>

## (五) 著名人士 (TODO)

1 · Michael Jordan

2 · Bengio

3 · Hinton

4 · Yann LeCun

5 · 李飞飞

6 · 吴恩达

7 · Ian Goodfellow

8 · 汤晓欧

香港中文大学教授，DeepID 系列论文的一作，商汤科技创始人。