_peaks _peaks _peaks _peaks

Machine or Deep Learning

冯迁

2019 年 3 月 3 日

目录

前言

本文档是我的一个学习过程，学习时间开始于 2016 年末，打算将其写出来于 2017 年中。本着不是科班出生，很多方面都不太扎实，包括程序设计，对算法的性能分析，以及囿于匮乏的经验对市场的状况认识还不太合格。将其写出来一来梳理自己，二来也可以利用自己的数学优势以及先行几步减少部分初学者学习上的困惑，最后也能知道自己的不足之处以及理解有误的地方。

首先以词条人工智能开始维基百科之旅: 人工智能。

AI 的核心问题包括推理, 知识, 规划, 学习, 交流, 感知, 移动和操作物体等。目前比较流行的方法包括统计方法，计算智能和传统意义的 AI。大量应用的人工智能包括搜索和数学优化, 逻辑推演。

接着自由选择进入机器学习页面：机器学习是人工智能的一个分支。在 30 多年的发展中，已成为一门多领域的交叉学科，涉及概率论，统计学，逼近论，凸分析，计算复杂性理论等多门学科。其算法主要实现从数据中自动分析获得规律，并利用规律对未知数据进行预测。主要分为四类：监督学习 (回归分析和统计分类)，无监督学习 (聚类)，半监督学习和增强学习 (基于环境行动，以取得最大化的预期利益)。

具体的机器学习算法:

1. 构造间隔理论分布: 聚类分析和模式识别
   (a) 人工神经网络
   (b) 决策树
   (c) 感知器
   (d) 支持向量机
   (e) 集成学习 AdaBoost
   (f) 降维与度量学习
   (g) 聚类
   (h) 贝叶斯分类器
2. 构造条件概率: 回归分析和统计分类
   (a) 高斯过程回归

   (b) 线性判别分析
   (c) 最近邻法
   (d) 径向基函数核
3. 通过再生模型构造概率密度函数:
   (a) 最大期望算法
   (b) 概率图模型: 贝叶斯网和 Markov 随机场
   (c) Generative Topographic Mapping
4. 近似推断技术:
   (a) 马尔科夫链
   (b) 蒙特卡洛方法
   (c) 变分法
5. 最优化: 大多数以上方法，直接或间接使用最优化算法。

以上是中文版的粗略分类，关于更详尽的分类可参考:Outline_of_ml. 假设我们已经浏览了此页面上所有内容，包括其链接内容。这样我们对人工智能整体有了个一个简单的认识: 算法庞杂，理论繁多，近几年活跃度很大。一方面方法可以很简单，比如 KNN,PCA,BP..., 另一方面也可以很复杂, 比如变分法，最优传输等。以现有的目光来看，很多基本的问题还需要学者们去解答。包括一个统一的理论框架，对深度黑箱的解释等。另外如此多的算法，在面对实际问题时，往往局限于模型的理想化，以及问题的类型，需要根据实际选择并改装。这也就促使我们选择自己感兴趣的分支，并掌握所需算法的精髓。似乎一下就变成了生命能承受之重了，... 似轻非轻..., 道路很长。

* 选择：CV
* 工具:TensorFlow,Pytorch,sklearn,Numpy,Opencv...
* 理论: 相似与度量学习, 凸优化, 概率统计, 最优传输
* 问题: 怎样学到最少的东西, 解决更多的问题?

关于工具的问题，纯粹学习工具本身，是一个挺无聊的过程。带着问题或者项目学习，在一定程度上能减少一些莫名的痛苦。学习工具之前要有一定的理论基础，单单学习框架是无意义的，假设不从长远来看的话，那就没问题了。理论学习若能结合具体问题进行比较，在面向工程时，感觉会过渡得更加自然，反过来，也可能会给理论研究注入新鲜东西。

第一章　数据处理

数据处理是机器学习的关键一步，不论是在训练前还是在训练当中，都存在对数据的各种处理。训练前，数据收集，数据质量评判，数据表示，数据特征抽取，数据降维，数据归一化，训练中，数据批归一化，数据重构...。

接下来假设我们已经拥有了比较完整均匀的数据。

数据预处理

Multivariate Statistical Analysis

多变量分析主要用于分析拥有多个变数的资料，探讨资料彼此之间的关联性或是厘清资料的结构，而有别于传统统计方法所着重的参数估计以及假设检定。常见的分析方法有 PCA,CCA,MDS,SEM 等。

PCA

主成分分析:PCA

PCA 分析计算的核心就是矩阵的奇异值分解，奇异值分解属于谱定理的一小部分，数学上谱定理是个很精彩的定理，但这里我们只能介绍 SVD。

假设 M 是一个 $m \times n$ 阶矩阵，其中的元素全部属于域 K，也就是实数域或复数域。如此则存在一个分解使得

$$M = U\Sigma V^*$$

其中 U 是 $m \times m$ 阶酉矩阵； 是 $m \times n$ 阶非负实数对角矩阵；而 V^*，即 V 的共轭转置，是 $n \times n$ 阶酉矩阵。这样的分解就称作 M 的奇异值分解。Σ 对角线上的元素 Σ_i, i 即为 M 的奇异值。

对于 PCA，我们要分解的就是数据的经验协方差阵，因为协方差阵是对称的，在线性代数里，我们知道每个正规矩阵都可以被一组特征向量对角化。即:

$$M = U\Sigma U^*.$$

实际上对于对称矩阵我们还可以做到

$$M = U\Sigma U^{-1}.$$

意义自明，U 的第 i 列表示 M 的第 i 个特征值对应的特征向量 (这里假设特征值是按顺序排列了)。现在我们需要多大比例的保持方差极大信息，选择一定数量的特征值及其特征向量即可。

“PCA 具有保持子空间拥有最大方差的最优正交变换的特性。然而，当与离散余弦变换相比时，它需要更大的计算需求代价。非线性降维技术相对于 PCA 来说则需要更高的计算要求。”

CCA

典型相关分析:CCA

CCA 寻找两个具有相互关系的随机变量的特征的线性组合，使其表示成的新特征之间具有最大的相关性。可以说是一种保持特征相关性的特征重构。具有降维的作用。其计算过程和 PCA 差不多，首先根据两随机向量 X, Y 计算其互协方差矩阵，然后求解向量 a, b 使得 $\rho = corr(a^{'}X, b^{'}Y)$ 最大，其中 $U = a^{'}X, V = b^{'}Y$ 是第一对典型变量，然后依次求得不相关的典型变量对。而这个问题最后被转化成一个求由协方差阵组合成的某对称矩阵的特征向量问题。

相关代码参考:PyCCA

Multidimensional scaling

多维标度:MDS

代码参考:PyMDS

AutoEncoder

AutoEncoder

从维基上我们看到，自编码是一种无监督式的数据重构方法，其理论比较简单，相应的利用 Tensorflow 或者 Pytorch 实现它也很简单，其扩展方式很多。

现在我们来看看采用概率图模型的自编码方法:Variational autoencoder。这里算了提前进入机器学习概率这一板块了，讲道理，这块是我的弱项。算是提前在这里熟悉概率的一些基本的东西吧。

通俗 VAE此讲解作为第一次阅读，以及后面的彩蛋，都不错。结合入门 VAE该文章小错误比较多，作为入门理解，还是不错的，且不可关注过多细节。入门 2AVE

基础阅读材料:TutorialVAE以及简短的变分推理 Blei, David M. ”Variational Inference.” Lecture from Princeton。

传统图像预处理

Pycode

第二章　机器学习

本章包含了机器学习的经典算法. 经典的机器学习算法有很多库都已经实现，我们没必要所有都去造轮子，我的选择是理解其数学部分，使用现有的库 sklearn，并在实践中分析理论与和实际的差距。假设我们对理论和库的调用都不太熟悉，实际上 sklearn 的 document 本身就是一个很好的学习地方，那里包含了算法的相关参考文献。后面的章节我们首先以这种方式来学习经典机器学习。

开始页面：sklearn-user-guide

4.6 核近似

4.7 Pairwise metrics, Affinities and Kernels

4.8 变换目标值

5 数据导入

6 大数据

以上是 sklearn 指导文档首页的部分目录，现在我们随机选择一些东西学习，比如我这里选择了接下来的四节内容。这只是一个初步的学习，剩下的就是在实践中不断的深化理解实际和理论上的差别，然后再反过来思考理论上的问题。这部分的理论相对简单，但这些优化方法却是人工智能的基础。

2.1 Regression

进入一般线性模型，玲琅满目，眼花缭乱。

2.1.1 Logistic regression

逻辑回归是一个二分类概率模型，其很容易扩展到多元情形，它将特征向量映射为一个概率向量，其每个分量表示特征属于其对应标签的概率。模型可表示如下：

$$P^{LR}(W) = C\sum_{i=1}^{n} log(1 + e^{-y_i W^T \mathrm{x_i}}) + 1/2W^T W \tag{2.1}$$

其中 $\{\mathrm{x_i}, y_i\}_{i=1}^{n}$ 表示数据以及其标签，$\mathrm{x_i} \in R^m, y_i \in \{1, -1\}.C > 0, W$ 是要学习的参数。

给定数据及其标签，我们可以用如下公式来表示条件概率。

$$P_W(y = \pm 1|\mathrm{x}) = \frac{1}{1 + e^{-yW^T\mathrm{x}}} \tag{2.2}$$

根据极大似然原理，我们很容易由 (2.2) 得到 (2.1)，如果我们去掉 (2.1) 的 $1/2W^TW$，这个多余的东西其实就是正则项，用来限制参数 W，防止过拟合的技巧。这个后面详说。在实际情况中，往往需要很多额外起脚来使模型更加实用。

现在的问题是如何得到模型参数 W, C?

答案：Coordinate descent approch, quasi-Newton method,iterative scaling method, exponetiatel gradient ... 如果你学过数值分析的话，你会觉得很多似曾相识，如果你学过凸分析的话，你会觉得很亲切，随着学习的深入，我们会逐渐建立更清晰的理论框架。现在不妨将视角转向 SVM。

Linear regression Logisticregression

2.2 支持向量机

SVM，一个二分类线性模型，简单的说，就是我们高中遇见的线性规划问题的推广。我们知道直线 $y = kx + b$ 将平面 xy 分成两部分，其实也就是两类，一类在“上面”，一类在“下面”。现在我们的情况只是在维度上增加了，也就是寻找一个超平面 $y = Wx + b$ 能将数据分类出来。模型可表示如下：

$$P^{SVM}(W) = C\sum_{i=1}^{n} max(1 - y_i W^T \mathrm{x_i}, 0) + 1/2 W^T W \tag{2.3}$$

很明显超平面是依赖训练数据的。从文档上看，该分类其的好处有：

- 高维空间上比较高效。
- 对特征维度大于样本量时，仍然有效 (大过多时，需要适当的选择核函数和正则项)。
- 用部分数据来得到决策函数，也即支持向量，能减少内存。

上面只是最简单的情形，多分类，多元回归呢?

我们先来看看文档里的情况，对于多分类，从文档里我们了解到两种方法：SVC 的“one-against-one”，n 类标签构建 $\binom{n}{2}$ 个分类器;LinearSVC 的“one-vs-the-rest”，训练 n 个模型。

对于回归问题，其对应的模块名叫 SVR。自行查看即可。现在的问题是：怎么从分类模型过度到回归模型呢？完整想出来，似乎还是有点难度，但是我们看到网页所给参看文献SVR，好了，又到了真正学习的时候了。

此段讲理论，以及代码分析。

然而实际上我们对多分类问题的处理方式还是失望的，我们并不想重复二分类模型，针对这个问题，表明我们该看看 1.12 节Multiclass-Multilabel了。

Maximum Entropy

蹦，问题又来了，所谓超平面，直观上看，毕竟是个“平”的。实际问题中，数据往往需要用一个弯曲的面才能将其较好的分类出来，这时怎么办呢？自然的我们有两种想法，一种直接把超曲面算出来，但这样不太好，考虑到曲面的表示方式，能控制的范围太小了，而实际变化范围太大；另一种方法就是保持超平面不变，直接映射输入数据，使其能被平面分割。这就是所谓的核技巧。

核技巧讲完了。

现在我们来看看以上模型该如何学习参数。

随机选择一个 Reference:Dual coordinate descent for LR and ME

2.3 Adaboost

略, 目前没有相关实战问题.

2.4 高斯混合模型

会在后面补上一个用 WGAN 来生成混合高斯分布数据.

第三章　理论知识

3.1　一些文章

这里主要列出一些参考文章，因为理论还没有建立起来。

- An explicit expression for the global minimizer network
- Understanding Deep Neural Networks with Renyi' s -entropy Functional
- Deep Neural Networks Learn Non-Smooth Functions Effectively
- Depth Efficiency of Deep Mixture Models and Sum-Product Networks using Tensor Analysis
- Universal approximations of invariant maps by neural networks
- Information based regularization for deep learning
- Theory of Deep Learning II: Landscape of the Empirical Risk in Deep Learning
- Theory of Deep Learning III: explaining the non-overfitting puzzle
- Deep Linear Networks with Arbitrary Loss: All Local Minima Are Global
- On the Optimization of Deep Networks: Implicit Acceleration by Overparameterization
- The Dynamics of Learning: A Random Matrix Approach

以上是以前随便列的一些文章, 很多文章也没有看. 现更新一些主题相对连贯的文章, 并在后小节补充说明一些细节内容. 以下内容主要消化自offconvex. 关于非凸优化的文章可参考nonconvex, 该网页实时更新.

1 The Loss Surfaces of Multilayer Networks(2015.Anna Choromanska)

2 Escaping From Saddle Points — Online Stochastic Gradient for Tensor Decomposition(2015.Rong Ge)

3 Efficient approaches for escaping higher order saddle points in non-convex optimization(2016.Anima Anandkumar)

4 How to Escape Saddle Points Efficiently(2017.Chi Jin)

5 Hessian-based Analysis of Large Batch Training and Robustness to Adversaries(2018)

6 Over- Deep Neural Networks Have No Strict Local Minima For Any Continuous Activations (2018)

7 Gradient descent with identity initialization efficiently learns positive definite linear transformations by deep residual networks (2018)

8 Deep linear neural networks with arbitrary loss: All local minima are global (2017)

文章 1,2,3 主要来自此篇博文:Escaping from Saddle Points. 论文 1 说明了在深度学习中, 几乎所有的局部极值点和全局最优点的函数差值都不大, 因此只要收敛到局部极小值点就差不多了. 而文章 3 又指出寻找一般非凸函数的局部极值点是一个 NP-hard 问题. 除此之外, 文章 3 主要考虑了非凸优化问题中的 degenerate saddle points , 并给出利用三阶导数信息来逃离鞍点并收敛到三阶局部最下值的算法. 文章 2 中作者定义了 strict saddle 概念, 并给出随机梯度下降法能在多项式时间内收敛到局部极值点的证明 (有限制条件), 也即 $y = x - \eta\nabla f(x) + \epsilon$, 其中 ϵ 就是梯度随机因子. 毕竟鞍点是不稳定的, 加扰动因子, 是很容易逃离鞍点附近的. 但是, 问题是如果鞍点附近平稳区域太大, 或者逃离的方向不太对, 怎么办? 这样就很容易跑到无穷小或无穷大去了. 关于这一点, 博主又更新了一篇文章Saddles Again, 文章表明一般优化算法很难收敛到鞍点, 除非你精心设计初始点, 调节参数等.

考虑简单的不定二次型, $f(x) = \frac{1}{2}\sum_{i=1}^{d} a_i x_i^2$ 其中前 k 个系数为正, 后 $d-k$ 个系数为负数, 因此我们很容易得到其梯度迭代形式:

$$x^{(k+1)} = x^{(k)} - t\nabla f\left(x^{(k)}\right)$$

考虑第 i 维度, 容易得到

$$x_i^{(k)} = (1 - ta_i)^k x_i^{(0)}$$

因 $\exists a_i > 0$, 于是从任何非零点开始都将以指数般的速度发散到无穷远. 当且仅当初始点为 0 时, 方才收敛, 但凡有一点扰动, 都将发散 (原文似乎有一些不严密的地方). 文章 4 也是在梯度方向上增加扰动, 给出了几乎与维度无关的收敛到二阶驻点的多项式时间复杂度算法. 而且收敛速度和众所周知的梯度下降收敛到一阶驻点的收敛速度同步差仅一个对数因子. 该方法可直接用于矩阵分解问题. 我将在 3.2.3 中详细展开该论文.

文章 5 实验了使用不同大小批量训练时, 模型的收敛邻域的局部几何之间的区别. 结果显示, 此前人们普遍相信的鞍点困扰优化的论据其实并不存

在在大批量训练过程中, 真正的原因是大批量训练的模型会逐渐收敛于具有更大的谱的区域, Hessian 矩阵的谱越大, 其对应的极值点更尖锐, 因此泛化更低, 但, 锐度可能不是唯一的因素.

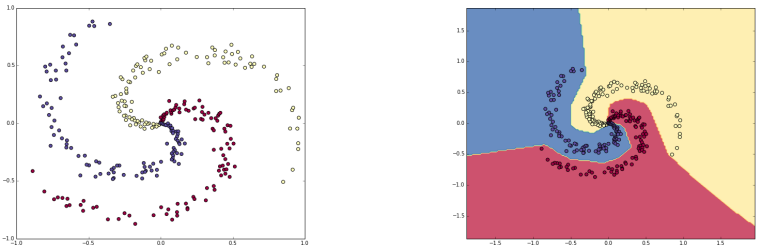
6,7,8 均还没有看, 文章的筛选, 是挺费时的一件事.

3.2 和深度学习相关的基础知识

上一节列了一些零散的深度学习理论文章，因为理论的东西还没有完全建立起来，将来更加成熟后会增加这一块的内容，这一节将组织一些零散的基础知识，因为面面俱到似乎没有必要。

3.2.1 多层感知机与任意函数的拟合

首先如下一张非线下可分的数据图，逻辑回归是搞不定的，但简单的 2 层感知机就可以做到。



(a) spiral_raw (b) spiral_net

图 3.1: perception

具体代码见：two-perception

关于其函数拟合能力的分析，参考：MLP-Function. 可视化理解参考:MLP-Function-Approximate。将来会补充一些实验代码。

3.2.2 卷积

数字图像处理基础：Conv-Matrix 维基百科卷积：Convolution-wiki

最早接触卷积是在学习傅里叶分析的时候，最后在解偏微分方程中有一些应用。两个函数的卷积相当于做了一次积分变换，其表达式为: $(f*g)(t)\int_{-\infty}^{\infty} f(\tau)g(t-\tau)d\tau$ 很直观的就能写出其离散的形式: $(f*g)[n] = \sum_{m=-\infty}^{\infty} f[m]g[n-m]$. 容易看出上面的形式和两个多项式的乘积的系数关

系一样，即：

$$\begin{gathered} p(x).q(x) = (p*q)(x) = a.b = [c_0, c_1, c_2, \ldots, c_{n+m}], \text{ where} \\ c_k = \sum_{i,j:i+j=k} a_i b_j \quad k = 0, 1, \ldots, n+m \end{gathered} \tag{3.1}$$

将其写成矩阵形式为：

$$y = a * b = \begin{bmatrix} a_1 & 0 & \ldots & 0 & 0 \\ a_2 & a_1 & \ldots & \vdots & \vdots \\ a_3 & a_2 & \ldots & 0 & 0 \\ \vdots & a_3 & \ldots & a_1 & 0 \\ a_{m-1} & \vdots & \ldots & a_2 & a_1 \\ a_m & a_{m-1} & \vdots & \vdots & a_2 \\ 0 & a_m & \ldots & a_{m-2} & \vdots \\ 0 & 0 & \ldots & a_{m-1} & a_{m-2} \\ \vdots & \vdots & \vdots & a_m & a_{m-1} \\ 0 & 0 & 0 & \ldots & a_m \end{bmatrix} \begin{bmatrix} b_1 \\ b_2 \\ b_3 \\ \vdots \\ b_n \end{bmatrix} \tag{3.2}$$

而式 (3.2) 中的矩阵也就是 Toeplitz matrix. 我们熟知的循环矩阵如下 C 就是其特殊形式。遥想当年考研的时候，探究过循环矩阵的性质，遂提一提，我想其在有限域中对编码的构造也许有用。

$$C = \begin{bmatrix} c_0 & c_{n-1} & \ldots & c_2 & c_1 \\ c_1 & c_0 & c_{n-1} & & c_2 \\ \vdots & c_1 & c_0 & \ddots & \vdots \\ c_{n-2} & & \ddots & \ddots & c_{n-1} \\ c_{n-1} & c_{n-2} & \ldots & c_1 & c_0 \end{bmatrix}. \tag{3.3}$$

一维卷积可以用 Toeplitz 矩阵来快速实现，二维卷积呢? 答案是 doubly block-Toeplitz。所谓 doubly block-Toeplitz 就是将任意 Toeplitz 中的元素替换为 Toeplitz 矩阵. 具体操作查看上面给的 Conv-Matrix 链接即可. 那么在如今流行的深度学习框架里面, 其实现方式又是如何? 当然是和上面的思路是一样的, 但是有一些出入的地方在工具的使用, 以及加速的设计上. 较深入的比如 cuda 等我就暂时不考虑了. 其 numpy 的实现方式, 参考我在第七章给出的代码块. 这里我将其主要意思翻译一下 (来源 cs231 课程卷积实现 cs231);

- 用函数 im2col 将输入图像中的各局部区域 (卷积核对应的区域) 展开为列。例如，输入是 [227x227x3]，用大小为 11x11x3, 步长为 4 的的

卷积核去做卷积，那么我们将输入图像的 [11x11x3] 大小的像素块，用 im2col 展开为 11*11*3=363 的列向量。因为步长为 4，故在长宽上都要做 (227-11)/41=55 次这样的操作，从而得到一个大小为 [363x3025] 的矩阵 X_{col}，其中每一列都是一个展平的接受域，总共有 55*55=3025 个。注意，由于感受野的重叠，输入被卷积块的每个数字可能在多个不同的列中重复。

- 卷积层的权重系数也同样被拉成行。例如，如果有大小为 [11x11x3] 的 96 个卷积核，这将给出一个大小为 [96x363] 的矩阵 W_{row}。
- 于是卷积的结果现在等价于执行一个大矩阵的乘法运算 $np.dot(W_{row}\ X_{col})$，它计算了每个卷积核和每个接收域之间的点积。在我们的例子中，这个操作的输出将是 [96x3025]，给出每个卷积核在每个位置块的点积输出。
- 最终须将结果重新调整到其适当的输出大小 [55x55x96]。

二维卷积的自实现代码测试可以用 signal.convolve2d(a,b) 完成.

3.2.3 同深度学习相关的优化

更基础的一些优化问题可以参考本章 3.1 节的简单阐述.

以下应该我是以前从哪里找的面试问题清单之一:

深度学习发展过程中，已衍生出较多的训练优化算法，且在实际应用中发挥越来越重要的作用。请根据你的经验，回答下面这些问题。1）训练优化算法的目的是什么？2）SGD 的原理和不足有哪些？3）在 SGD 基础上，已逐渐优化出 Momentum、AdaGrad、AdaDelta、Adam 等算法，请简述这 4 种算法的原理及其改良出发点。4）请根据你的理解阐述上述几种算法分别适应哪些问题类型的优化，或根据你的实际经验阐述应用了何种优化算法取得了较好效果及其原因分析

因此, 在这一节, 我将把框架中使用的优化器, 做一个简单的总结 (此时在我脑海里首先出现的是两篇文章):

一个框架看懂优化算法之异同 SGD/AdaGrad/Adam

优化诗

第一篇文章实际为三篇, 我这里将其做个缩减, 摘录到这里. 第二篇文章的优化诗不错, 将会附在本节后面.

总体框架:

0 首先定义：待优化参数：w ，目标函数：$f(w)$ ，初始学习率 α。

而后，开始进行迭代优化。在每个 epoch t ：

1 计算目标函数关于当前参数的梯度：$g_t = \nabla f(w_t)$

2 根据历史梯度计算一阶动量和二阶动量：$m_t = \phi(g_1, g_2, \cdots, g_t); V_t = \psi(g_1, g_2, \cdots, g_t)$,

3 计算当前时刻的下降梯度：$\eta_t = \alpha \cdot m_t / \sqrt{V_t}$

4 根据下降梯度进行更新：$w_{t+1} = w_t - \eta_t$

各个算法的具体情况:

- SGD:$m_t = g_t; V_t = I^2$, 没有动量.
- SGDM: $m_t = \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t$, 一阶动量是各个时刻梯度方向的指数移动平均值，约等于最近 $1/(1 - \beta_1)$ 个时刻的梯度向量和的平均值, 于是下降方向由当前梯度方向和累计梯度方向共同决定.
- SGDN: $g_t = \nabla f\left(w_t - \alpha \cdot m_{t-1} / \sqrt{V_{t-1}}\right)$, 按累计梯度方向走一步后的梯度方向和历史动量相结合.
- AdaGrad: $V_t = \sum_{\tau=1}^{t} g_\tau^2$, 用二阶动量去度量历史更新频率, 来动态调节学习率 (频率越快学习率越小).
- AdaDelta / RMSProp: $V_t = \beta_2 * V_{t-1} + (1 - \beta_2)\, g_t^2$, 利用指数移动平均值来计算过去一段时间梯度的二阶动量, 避免学习率递减的过快.
- Adam: $m_t = \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t$, $V_t = \beta_2 * V_{t-1} + (1 - \beta_2)\, g_t^2$, 同时考虑了一二阶动量, 且都是指数移动平均值.
- Nadam: Nesterov + Adam

问题:

1. SGD 收敛速度慢, 且不容易逃离局部最优点 (实际上我在第七章的实验发现, 针对同样的参数, 其他优化方法在给定的函数上也很难跳出局部最优点).

2. Adam 所谓的学习率自适应并不是真的自适应, 在一些优化函数上可能导致不收敛 (后期学习率有震荡现象), 可能错过全局最优 (跳出去了就很难回来),

3. 缺乏对数据的理解.

经验:

1. 优先考虑 SGD+Nesterov Momentum 或者 Adam

2. Adam 等自适应学习率算法对于稀疏数据具有优势，且收敛速度很快；但精调参数的 SGD（+Momentum）往往能够取得更好的最终结果。

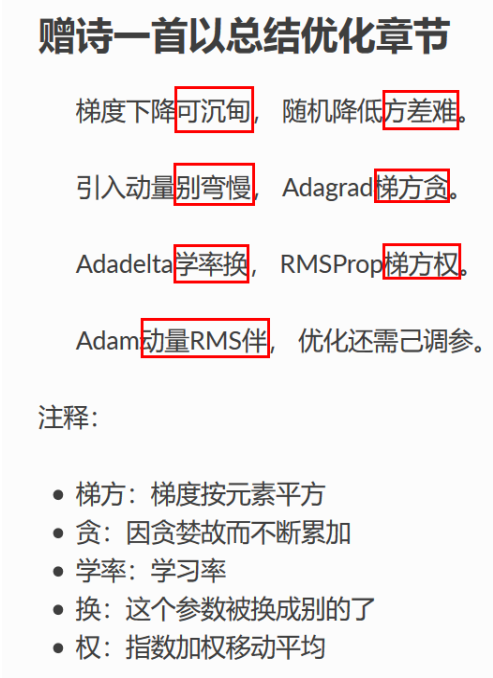
3. 先用 Adam 快速下降，再用 SGD 调优:Improving Generalization Performance by Switching from Adam to SGD

4. 学习率策略 (见 7.1.2)

深度学习中 7 种最优化算法的可视化与理解该文是在一维函数上做的实验, 而我在第 7 章做了一些二维上的实验, 可以用来加深直观认识.

数学形式的分析参考深度学习中的优化算法-张戎，这个分析针对一些具体的二元函数, 相对实际情况是比较简单的.

优化算法暂时就以链接优化诗中的一首诗来结束吧.



3.2.4 前向, 后向传播

后面结合框架写一写.

第四章　强化学习

强化学习将在 9102 年给补上.
首先推荐的书籍:neuralnetworks

$$v_\pi(s) = \sum_{a \in A} \pi(a|s) q_\pi(s,a)$$

$$q_\pi(s,a) = R_s^a + \gamma \sum_{s'} P_{ss'}^a v_\pi\left(s'\right) q_\pi(s,a) = R_s^a + \gamma \sum_{s'} P_{ss'}^a v_\pi\left(s'\right)_{(2.3)}$$

$$v_\pi(s) = \sum_{a \in A} \pi(a|s) \left(R_s^a + \gamma \sum_{s' \in S} P_{ss'}^a v_\pi\left(s'\right) \right) \quad (2.4)$$

DL

4.1　经典强化学习算法

4.2　深度强化学习算法

GAN 与 actor-critic

4.2.1　DQN

4.2.2　A3C

4.2.3　DDPG

4.3　游戏中的应用

第五章　传统智能

遗传算法，最优路径等。后面将增加一些和优化相关的内容。

5.1　路径规划

等有相关工作任务时在写吧。

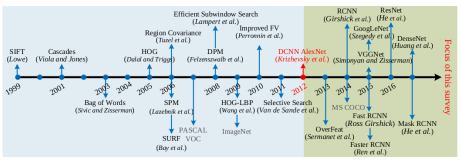
第六章　论文阅读

6.1　物体检测

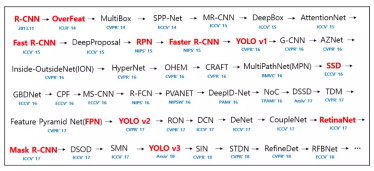
本文主要梳理一下我对:SSD,YOLO v3, Mask R-CNN 三个算法的理解，包括算法基本架构，物体检测的基本思想，以及源码解析，目前属于第一版，有诸多不顺畅及欠缺的地方，待将来优化。

6.1.1　历史

就我自己来说，历史的所有细节 (包括每个算法之间的不同, 各自优势, 各自解决的问题等) 没精力去详细考察，把这些细节留给研究者去做，然后自己筛选几个关键算法做深入了解 (理论上包括: 算法理解，实际应用，算法改进) 即可。如此，选了如下总结图:



(a) fig1



(b) fig2

图 6.1: History

Fig1 来自 Deep Learning for Generic Object Detection: A Survey，它给出了图像检测从传统人工提取特征的巅峰算法 SIFT 到目前 (2018) 神经网络提取特征的 state-of-art 算法 Mask R-CNN 的时间节点图，Fig2 来自网络推文，它对深度学习这块的发展情况做了更详细的补充。我将从 Fig2 中标红的关键算法中选择三个节点:SSD,YOLO v3, Mask R-CNN 作为学习对象，假设一段恍惚的时间过去，将理解梳理如下:

6.1.2 结构图

展示 SSD, YOLO v3, Mask R-CNN 的结构图，并给出说明。

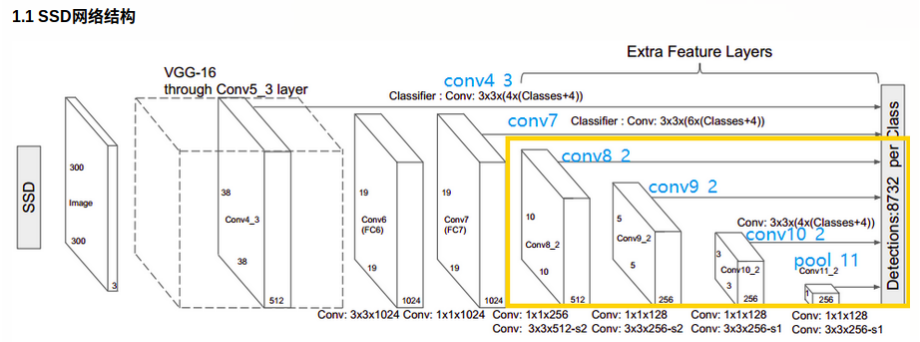


图 6.2

从上图看到 SSD 的特征提取网络为 VGG-16，其预测部分利用了输入图像的多层特征，这和后来的 Mask R-CNN，YOLO v3 所用的 FPN 网络结构殊俗同归，而基本思想可以说来源于 1999 年的 SIFT 算子中利用的特征金字塔结构。主要目的还是检测不同尺度的对象，以及融合不同层级的特征来提高对象的表示能力。其分类和回归均利用卷积完成，后来的 Mask RCNN, YOLO v3 也借鉴了这种处理方式，减少了全连接的运算时间。

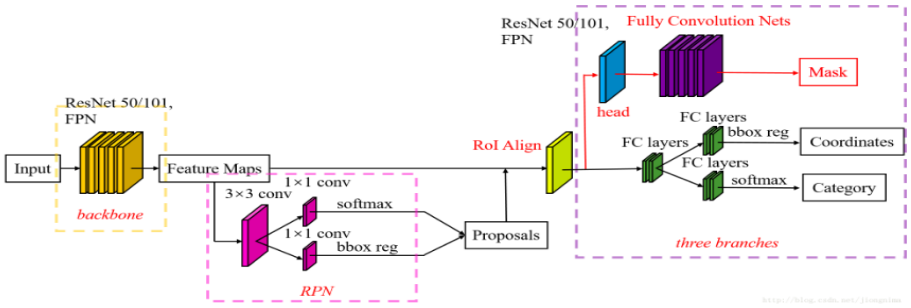


图 6.3

如上我们看到，其特征提取网络为 ResNet 系列加 FPN(FPN 可以理解成将传统金字塔思路和 SSD 中的多层级思路推广出的一个通用网络结构)，而 YOLO v3 的特征提取网络就是在其 DarkNet 的基础上借鉴了此思路 (DarkNet+Residual+FPN)，接着 RPN 层用来初步提取感兴趣区域，以及后面的分割块，检测和分类模块做最后的完善。相关细节将在后面的代码分析部分展开。

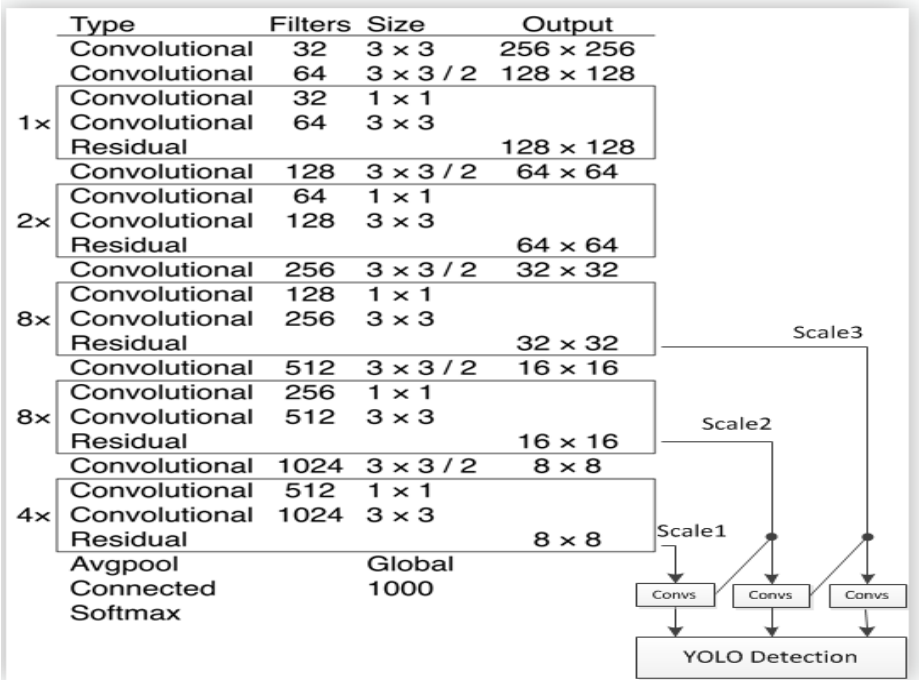


图 6.4

如上我们看到 YOLO 的升级版 (相对 yolov2) 特征提取网络:Darknet-53 以及右侧的检测模块即构成 YOLOv3 的网络结构图。看得出来，它和 SSD 的思想架构是一样的：特征提取网络和多层级预测。

6.1.3 异同补充

三个算法的检测部分都是采用 anchor 的思想来实现，首先是用各自不同的特征提取网络提取特征，然后根据特征图构造 anchor，并给所构造的 anchor 打上标签，用于分类，然后对 anchor 进行编码 (目标 anchor 也要进行编码)，编码的必要性来自于神经网络的需求，输入数据归一化到 0-1 之间，保持网络顺利学习模型参数。

三个算法的特征提取网络不同，anchor 的构造也不同。

首先 SSD 在不同特征层构造不同 scale 且在同一特征层有不同 aspect ratio 的 anchor，以此达到构造的 anchor 能覆盖住输入图片中不同大小不同形状的 object。具体计算规则如下：

1. 以特征图上每个点的中心点位中心 (offset=0.5), 生成一系列同心的 anchor(具体实现的时候都需要将此 anchor 映射到原图中去，也即乘以 step/stride)

2. 正方形 anchor 最小边长 min_size, 最大边长 $\sqrt{min\_size \times max\_size}$, 相应比例的长宽分别为:$\sqrt{aspect\_ratio} \times min\_size$, $\frac{1}{\sqrt{aspect\_ratio}} \times min\_size$.

3. 每个特征层对应的 anchor 的 min_size 和 max_size 由如下公式确定:

$$s_k = s_{min} + \frac{s_{max} - s_{min}}{m-1}(k-1), k \in [1, m]$$

其中 m 是使用特征层的数量 (SSD300,m=6), 第一特征层对应 $min\_size = s_1, max\_size = s_2$, 第二层对应 $min\_size = s_2, max\_size = s_3$ 等, 其分别对应 SSD 结构图的蓝色特征层。这块内容在 ssd.pytorch 的 config 中即可看得很清楚。可参考:SSD 精述。

其次 YOLO v3 的 anchor 是用 K-means 从数据的标注信息聚类得到 9 类不同尺度的先验框 (将每个 box 的宽和高相对整张图片的比例 (w_r, h_r) 进行聚类, 得到 9 个 anchor box)。以宽度大小作升排序, 分成三个层级, 分别对应 YOLO v3 框架图中的 scale3,scale2,scale1, 并用各自层的先验框给特征层的每个特征点分配 1 个 anchor(与真实 box 具有 max_iou)。

最后 Mask R-CNN 的 anchor 和 SSD 基本思想是相同的: 在不同层级的特征图上生成不同 scale 的且同一层特征图上给不同 aspect_ratio 的 anchor。只是具体的生成方式比上面 123 条要简洁。具体看代码即可。

如上 SSD 和 Mask R-CNN 在 anchor 的构造上基本一致, YOLO v3 独树一帜, 另外三个算法对 anchor 的编码基本保持一致: 中心坐标相对归一化, 尺度上映射到对数空间中, 即 $t_x = (x - x_a)/w_a, t_w = log(w/w_a)$, 其中 x, x_a, w, w_a 分别表示特征位置 x 值, 对应 anchor 的 x 值, 特征图宽度, 对应 anchor 的宽度。然而在位置损失函数上, YOLO v3 又有所不同, 首先前两者均是根据 IOU 值较大时, 可用线性拟合去估计位置偏移量的思想, 估计变量经过编码 (映射) 后, 学习函数即可表示为:

$$W_* = argmin_{\hat{W}_*} \sum_i^N (t_*^i - \hat{W}_*^T \Phi_5(P^i))^2 + \lambda ||\hat{W}_*||^2$$

其中 $t_* = (t_x, t_y, t_w, t_h)$ 表示真实值与 anchor 的偏移量和缩放因子, 预测时, 采用逆变换。而 YOLO v3 的坐标误差:

$$\lambda_{\text{coord}} \sum_{i=0}^{S^2} \sum_{j=0}^{B} \mathbb{1}_{ij}^{\text{obj}} \left[(x_i - \hat{x}_i)^2 + (y_i - \hat{y}_i)^2 \right] \quad \text{坐标误差}$$

$$+ \lambda_{\text{coord}} \sum_{i=0}^{S^2} \sum_{j=0}^{B} \mathbb{1}_{ij}^{\text{obj}} \left[\left(\sqrt{w_i} - \sqrt{\hat{w}_i} \right)^2 + \left(\sqrt{h_i} - \sqrt{\hat{h}_i} \right)^2 \right]$$

图 6.5

预测 x, y 偏移量采用了 logistic 变换。与此同时，其综合损失函数也有较大差异。YOLO v3 的综合误差包含三个部分：坐标误差，IOU 误差和分类误差，而另外两个则采用了 smooth_l1 和交叉熵的组合形式。

6.1.4 插曲

闲聊一段深度学习。目前深度学习基本采用端到端的模式从数据中学习数据的某些特性，期望达到应用的目的。大部分实际工作主要围绕如下词汇展开：data , network, loss function, optimizer。实际应用中，数据处理所占比重较大；网络结构目前属于实验范畴，也有了一些基本的网络结构，包括 ResNet,Inception,DenseNet 等的基本构件，正在发展的 GCN 应该会对网络结构这块带来更丰富，理论化的认识；损失函数基本围绕三种类型进行组合, 变形：MSE，Hinge，Cross Entropy；优化方法即古老的梯度下降法以及其三类推广形式 1 增加动量项，动态改变梯度更新方向 (NAG)，2 根据梯度范数等动态改变学习率 (Adagrad,Adadelta)，3 前两者的结合 (Adam)。从上面可以看到，深度学习目前仍处于较初级的状态，另外对抗学习 GAN 用到的数学理论也足以说明，深度学习刚起步。要想用到更高级的数学理论，以及建立合适的深度学习理论都将任重而道远，虽然以上提到的各个板块都有很多需要进一步做下去的问题。

6.1.5 代码分析

- SSD:ssd.pytorch
- YOLO v3:keras-yolo3;
- Mask R-CNN: Mask R-CNN

想了下，只分析 Mask RCNN，其实MASK_RCNN 代码详解 已经分析了一遍，他将代码分成了如下四个部分:

- Backbone Network 代码
- Region Proposal Network(RPN) 代码
- Network Heads 代码
- Losses 代码

其每个部分都做了详细的分析，但缺乏整体性，在这里我就将这些细节略过，然后对整体性做一些补充，并对一些实际中会面对的基本问题做一些经验梳理。

首先我们从 Mask RCNN 的结构图就能清楚的看到算法由如下逻辑连接：特征提取网络 (ResNet-101+FPN) 提取多层特征得到 Feature Maps；接着一个 RPN 网络对 Feature Maps 进行二分类和边框回归，得到 Pro-

posals；然后一个 ROI Align 用传统方式将 Proposals 统一到固定大小 (7×7) 放入最后的三并行网络，mask 预测，类别预测和框回归。这刚好对应了MASK_RCNN 代码详解。正如上面所言，你可能看完还是有点迷，故，我换个深度学习一般采用的模式来做一些修补说明。

问题：如何写 Data 类，如何给数据标签编码，如何写损失函数，有哪些辅助函数? 提这些问题的原因在于我觉得网络结构可变性不大，主要还是如何处理数据，如何根据损失函数表示数据，如何加速训练以及如何评估模型的好坏等。现随机回答以上问题:

Dataset 类具有 add_class,add_image,load_image,load_mask 等基本功能，image_info,class_info,image_ids 等基本属性。这些在不同的框架下都是如此。对于 Mask RCNN 数据处理的主要难点在于 mask 这一块。这里我说下源码是如何处理的 (以 coco 数据为例):

首先 CocoDataset 类调用其功能函数 load_coco 得到数据的所有信息，然后将其放入数据生成器 data_generator 进行 batch 包装，在其中 load_image_gt 调用 CocoDataset 类的 load_mask 函数返回输入图片的 mask 以及对应的 class_id，因为一张图片有多个 mask，故返回多个 mask 的 stack 和 class_ids, 此处 mask 为原图大小的 bool array。在源码中还会做一个 RLE 的编码解码过程，以节约内存，得到 inputs[-1] 即 batch_gt_masks。最后在类 MaskRCNN 中将 inputs[-1] 放入 DetectionTargetLayer 类得到 target_mask，shape=[batch,TRAIN_ROIS_PER_IMAGE,height,width]，在其中首先调用 detection_targets_graph 对 batch_gt_masks 做筛选 (根据得到的 Proposals 与原始框的 iou 值) 得到 config 中设定的 TRAIN_ROIS_PER 个 mask 即 shape=[TRAIN_ROIS_PER_IMAGE,height,width]。

个人吐槽，刚看看源码时，有一个地方会让人迷糊，请源码搜索如下关键字：random_rois,generate_random_rois,build_detection_targets。其实源码说明中说得很清楚，这块不参与正常训练，是在 debugging 或者用 generate_random_rois 生成的 rois 训练 Mask RCNN heads 时用，但是由于 rpn 网络所用 targets 是由 build_rpn_targets 给出的，而且 build_detection_targets 的 mask 构造维数为 4 维，这样名字上，理解上难免会让人绕一个圈子。

rpn 网络由 build_rpn_model 构建，其接受 P2-P6 特征层，给出 rpn_class, rpn_bbox,rpn_class_logits 然后进过 ProposalLayer 进行筛选 (anchor score,iou，nms，个数) 得到 rpn_rois，在这里关键的还是如何编码目标值以及训练数据的筛选，分类 one-hot，坐标回归，位置归一化，尺度映射到对数空间，训练筛选做到均衡，有利于损失函数的优化。以 build_ron_targets 为例：将 iou 小于 0.3 的赋为-1，>= 0.7 的赋为 1，并在数量上做到不超过 RPN_TRAIN_ANCHORS_PER_IMAGE。rpn_bbox 为 anchor 与原

始 bbox 的相对偏移量和尺度比率的对数值，这里有点拗口，公式清晰易懂。另外 rpn_rois 是经过预测偏移量修正后的坐标值。以上描述看下图加以复习:

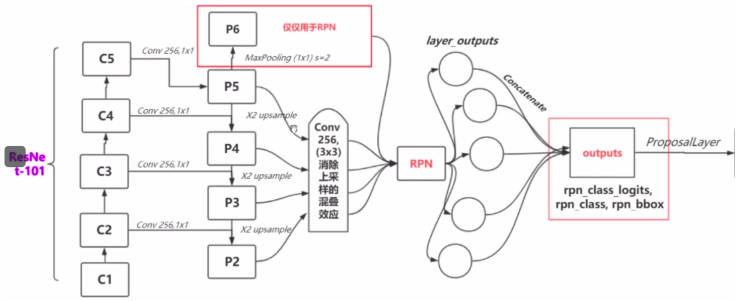


图 6.6

现接着对下图做点补充:

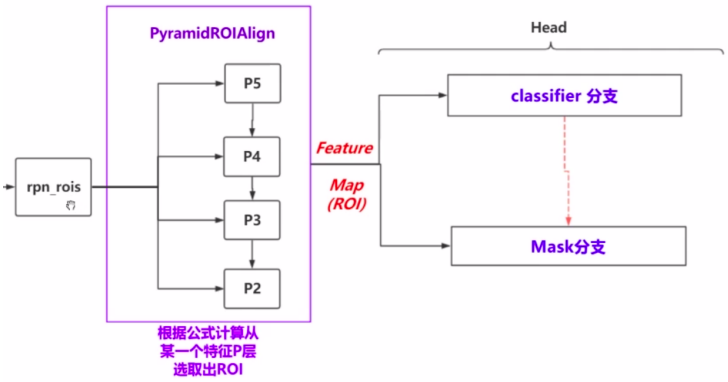


图 6.7

PyramidROIAlign 为一个自定义层，在 keras 中，自定义层主要由 ___init___(**kwargs) 字典类继承，call 方法，compute_output_shape 方法构成。PyramidROIAlign 的主要目的是将 rpn_rois 从其对应的 feature_maps 中提取出相同大小的特征信息作为最后的 Head 分支的输入。这里有两点需要说明:

- 1. rpn_rois 的层级算法，来源于 FPN 论文:

$$k = \lfloor k_0 + log_2(\sqrt{wh}/224) \rfloor$$

也可参考：rois_leavel。其意思是说将我们预测得到的 anchor 根据其面积划分到相应的特征层，从而在相应的特征层上进行 Pooling，源码所用的函数为 tf.image.crop_and_resize。

- 2. 每个 box 均要提取对应的 Pool 特征。最后将得到的 rois 送入 Head 分支，即 fpn_classifier_graph 和 build_fpn_mask_graph 中，分别得到 mrcnn_class_logits, mrcnn_probs, mrcnn_bbox 和 masks: [batch, roi_count, heights, width, num_classes]。

损失函数:smooth_l1_loss, rpn_class_loss_graph, rpn_box_loss_graph, mrcnn_class_loss_graph, mrcnn_bbox_loss_graph, mrcnn_mask_loss_graph. 若我们掌握了数据的高维表示，那么损失函数就很好写了。比如函数 mrcnn_mask_loss_graph:

```
def mrcnn_mask_loss_graph(target_masks, target_class_ids, pred_masks):
"""Mask binary cross-entropy loss for the masks head.
target_masks: [batch, num_rois, height, width].
A float32 tensor of values 0 or 1. Uses zero padding to fill array.
target_class_ids: [batch, num_rois]. Integer class IDs. Zero padded.
pred_masks: [batch, proposals, height, width, num_classes] float32 tensor
with values from 0 to 1.
"""
  # 将个数信息reshape成一维,高维数据操作，为了好算,基本上所有损失函数都这样
  target_class_ids = K.reshape(target_class_ids, (-1,))
  mask_shape = tf.shape(target_masks)
  target_masks = K.reshape(target_masks, (-1, mask_shape[2], mask_shape[3]))
  pred_shape = tf.shape(pred_masks)
  pred_masks = K.reshape(pred_masks,
  (-1, pred_shape[2], pred_shape[3], pred_shape[4]))
  # 将类别信息放在第二维度，方便tf.gather_nd作位置筛选，[个数，类分数，
    mask_h, mask_w]
  pred_masks = tf.transpose(pred_masks, [0, 3, 1, 2])

  # tf.gather 选出正样本，贡献损失函数值
  positive_ix = tf.where(target_class_ids > 0)[:, 0]
  positive_class_ids = tf.cast(
  tf.gather(target_class_ids, positive_ix), tf.int64)
  # indices.shape = [num_positive, 2] ，[下标位置，标签],这里标签其实就是类
    别位置
  indices = tf.stack([positive_ix, positive_class_ids], axis=1)

  y_true = tf.gather(target_masks, positive_ix)  # 形状 [num_positive,
    mash_h, mask_w], type:bool
  y_pred = tf.gather_nd(pred_masks, indices) # tf.gather_nd根据前两维度的位
    置选出mask,结果维度如y_true

  # K.switch三目表达式， K.binary_crossentropy对mask中像素做二分类
  loss = K.switch(tf.size(y_true) > 0,
  K.binary_crossentropy(target=y_true, output=y_pred),
  tf.constant(0.0))
  loss = K.mean(loss)
  return loss
```

我们看到写损失函数，主要就是做有效信息筛选以及数据的结构调整。需要注意的是，在源码的训练中有一个 (前面也有提到) 非正常 (normal) 训练 Head，它没有用到 RPN 网络所筛选出来的 rois 进行训练，而是从原始 bbox 中随机生成的 anchor 进行后续训练，关键地方在于，其 target_masks 维度是 4，加 batch 则为 5 维，且其 mask 进行的是多分类。

现在剩下最后一块内容：数据增强，辅助功能函数。除开我们不同工程写的各自函数外，普遍的辅助函数，在以上列出的三项源码中均有体现，且实现都较为灵活，故暂时略去这块内容。

6.2 GAN

本章主要尝试梳理 GAN 算法的一些结果，搭建基本认识，包含当前理论阐述，一些问题，解决方案以及部分代码实现。理论升级，工程实践待添加 (后面两个才是关键. 汗.jpg)。

论文列表

1 Generative Adversarial Networks(2014.7)

2 Wasserstein gan(2017.2)

3 Spectral Norm Regularization for Improving the Generalizability of Deep Learning(2017.5)

4 Spectral Normalization for Generative Adversarial Networks(2018.2)

5 The relativistic discriminator: a key element missing from standard GAN(2018.9)

6 GANs for Medical Image Analysis(2018.9)

7 Auto-Encoding Variational Bayes(2014.5)

- ...

6.2.1 基本思想, 问题及解决方案

利用神经网络的函数拟合能力，结合博弈的思想，论文 [1] 首次提出了对抗神经网络 GAN。一个判别网络 D，一个生成网络 G，一个优化对象 V。数学表达即为：

$$\min_G \max_D V(D,G) = \mathbb{E}_{\boldsymbol{x}\sim p_{data}[(\boldsymbol{x})}[logD(\boldsymbol{x})] + \mathbb{E}_{\boldsymbol{z}\sim p_{\boldsymbol{z}}(\boldsymbol{z})}[log(1-D(G(\boldsymbol{z})))] \tag{6.1}$$

表达式 (6.1) 清晰的给出了该算法的优化目标，首先固定 G 更新 D 最大化 V 得到 D^*，其次固定 D 更新 G 最小化 V_{D^*} 得到最优网络参数 θ_d, θ_g。容易得到当 $D = D^*$ 时，生成网络的优化目标变为原始数据 p_{data} 与生成数据 p_g 之间的 JS 散度，从而整个算法的思想就很明了了。不过，理论上如此，问题还是有很多，比如网络结构的设计 (问题不大)，模型优化困难等。对此论文1.5,[2] 具体分析原模型的问题，并给出了相关解决方案。中文细节可参考:WGAN 拍案叫绝。

原始 GAN 的主要问题:

a 判别器越好，生成器梯度消失越严重

b 生成器损失函数不合理 ($\mathbb{E}_{\boldsymbol{z}\sim p_{\boldsymbol{z}}(\boldsymbol{z})}[-log(D(G(\boldsymbol{z})))]$)，导致梯度不稳定以及 collapse model。

问题 a：由于随机生成分布很难与真实分布有不可忽略的重叠 (支撑集交集小或低维嵌入) 以及 JS 散度的突变特性 (JS 散度刻画两分布 “距离” 并不

连续)，使得生成器面临梯度消失的问题；
问题 b：优化生成器时 (trick -D)，既要最小化 P_r, P_g 的 KL 散度，又要最大化其 JS 散度，矛盾，导致梯度不稳定，而且 KL 散度的不对称性使得生成器为了生成样本的准确性丢到了生成样本的丰富性，导致 collapse model 现象。
注：条目 b 中的生成器损失函数为原始的改写，殊俗同归。不难推导优化改写后的损失函数等价于优化:$KL(P_g||P_r) - 2JS(P_r||P_g)$，写成这样就能较容易分析出原始 GAN 难以训练的原因了。

原始模型因为最优化判别模型后导致生成模型优化对象为一个 JS 散度或者 $KL - 2JS$ 而出现种种问题。如何解决呢？一个自然的想法就是改变生成模型的优化对象，既然 KL，JS 都不太好，就让 Wasserstein 距离来吧。首先去 Wikipedia 看看:Wasserstein_metric。

数学上的 Wasserstein_metric 是在一般概率度量空间中给出的，论文 [2] 采用了其特殊情形 Wasserstein-1，也即计算机科学中常遇见的 Earth-Mover distance:

$$W(P_r, P_g) = \inf_{\gamma \in \Pi(P_r, P_g)} \mathbb{E}_{(x,y)\sim\gamma}[||x - y||] \tag{6.2}$$

其中 $\Pi(P_r, P_g)$ 表示 P_r, P_g 的联合分布 $\gamma(x, y)$ 集。

但是 (6.2) 式在实际情况中难以操作，至少联合分布采样就无法完成，然而根据 Kantorovich-Rubinstein duality 此距离可改写成如下形式:

$$W(P_r, P_g) = \sup_{||f||_L \leq K} \mathbb{E}_{x\sim P_r}[f(x)] - \mathbb{E}_{x\sim P_g}[f(x)] \tag{6.3}$$

其中$||f||_L \leq K$表示满足 K-Lipschitz 条件的函数。

以上即得到了我们要最小化的生成损失函数 $W(P_r, P_g)$，将其和 (6.1) 对比，WGAN 的优化目标 V 可表示为:

$$\min_G \max_{D_f} V(D_f, G) = \min_{P_g} \sup_{||f||_L \leq K} \mathbb{E}_{x\sim P_r}[f(x)] - \mathbb{E}_{x\sim P_g}[f(x)] \tag{6.4}$$

对比 (6.1)，(6.4) 不同的地方还是很明显的。首先，前者判别网络输出 0,1 类的概率值，而后者为一个距离值 (Wasserstein distance)，一个分类，一个回归，所以网络结构上要去掉 sigmoid 函数层，其次判别网络函数被限制为满足 K-Lipschitz 的函数，由它得到的生成优化对象 Wasserstein distance 连续可微且无跳跃点，距离合理且梯度不会常为 0 或者爆炸，从而算是解决了前面的问题。

注:Wasserstein metric 本身和最优传输息息相关，法国数学家 Cédric Villani 写过一本Optimal transport, old and new，有难度，另外最优经典问

题 Monge-Kantorovich Transportation problem 可参考Monge-Kantorovich Transportation Problem .该文有对上面对偶公式的说明。

另附加一个公式 (将来有用) 两个 n 维高斯分布的 Wasserstein 距离为:

$$W_2(\mu_1,\mu_2)^2 = \|m_1 - m_2\|_2^2 + \operatorname{trace}\big(C_1 + C_2 - 2\big(C_2^{1/2}C_1C_2^{1/2}\big)^{1/2}\big) \quad (6.5)$$

最后用算法伪代码来复习一下这一小节内容:

Algorithm 1 Minibatch stochastic gradient descent training of generative adversarial nets. The number of steps to apply to the discriminator, k, is a hyperparameter. We used $k = 1$, the least expensive option, in our experiments.

for number of training iterations **do**
  for k steps **do**
    • Sample minibatch of m noise samples $\{\boldsymbol{z}^{(1)}, \ldots, \boldsymbol{z}^{(m)}\}$ from noise prior $p_g(\boldsymbol{z})$.
    • Sample minibatch of m examples $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ from data generating distribution $p_{\text{data}}(\boldsymbol{x})$.
    • Update the discriminator by ascending its stochastic gradient:

$$\nabla_{\theta_d} \frac{1}{m} \sum_{i=1}^{m} \left[\log D\left(\boldsymbol{x}^{(i)}\right) + \log\left(1 - D\left(G\left(\boldsymbol{z}^{(i)}\right)\right)\right)\right].$$

  end for
  • Sample minibatch of m noise samples $\{\boldsymbol{z}^{(1)}, \ldots, \boldsymbol{z}^{(m)}\}$ from noise prior $p_g(\boldsymbol{z})$.
  • Update the generator by descending its stochastic gradient:

$$\nabla_{\theta_g} \frac{1}{m} \sum_{i=1}^{m} \log\left(1 - D\left(G\left(\boldsymbol{z}^{(i)}\right)\right)\right).$$

end for
The gradient-based updates can use any standard gradient-based learning rule. We used momentum in our experiments.

(a) GAN

Algorithm 1 WGAN, our proposed algorithm. All experiments in the paper used the default values $\alpha = 0.00005$, $c = 0.01$, $m = 64$, $n_{\text{critic}} = 5$.

Require: : α, the learning rate. c, the clipping parameter. m, the batch size. n_{critic}, the number of iterations of the critic per generator iteration.
Require: : w_0, initial critic parameters. θ_0, initial generator's parameters.
while θ has not converged **do**
  for $t = 0, ..., n_{\text{critic}}$ **do**
    Sample $\{x^{(i)}\}_{i=1}^{m} \sim \mathbb{P}_r$ a batch from the real data.
    Sample $\{z^{(i)}\}_{i=1}^{m} \sim p(z)$ a batch of prior samples.
    $g_w \leftarrow \nabla_w \left[\frac{1}{m}\sum_{i=1}^{m} f_w(x^{(i)}) - \frac{1}{m}\sum_{i=1}^{m} f_w(g_\theta(z^{(i)}))\right]$
    $w \leftarrow w + \alpha \cdot \text{RMSProp}(w, g_w)$
    $w \leftarrow \text{clip}(w, -c, c)$
  end for
  Sample $\{z^{(i)}\}_{i=1}^{m} \sim p(z)$ a batch of prior samples.
  $g_\theta \leftarrow -\nabla_\theta \frac{1}{m}\sum_{i=1}^{m} f_w(g_\theta(z^{(i)}))$
  $\theta \leftarrow \theta - \alpha \cdot \text{RMSProp}(\theta, g_\theta)$
end while

(b) WGAN

图 6.8: GANs

需要说明的是,图 b 中 W 的截断方式,是为了使判别函数满足 Lipschitz 的工程实现方式，后续论文将会对此进行改进，也即论文 [3,4]，具体见下节。

6.2.2 模型的泛化

模型的泛化意指模型对输入数据的微小扰动不敏感。可用 Lipschitz 约束来刻画:

$$||f_w(x) - f_w(x+\epsilon)|| \le C(w) \cdot ||\epsilon|| \quad (6.6)$$

其中函数 f_w 的变动被其系数 w 控制。模型泛化能力越强，表示 ϵ 小幅度变化时 $C(w)$ 能保持很小的值。

由于深度学习的映射函数是线性映射加一个非线性变换的基本结构按层递推而得，所以分析某一层以及各层连接关系即可。对第 l 层，不难写出:

$$x^l = f^l(W^l x^{l-1} + b^l) \tag{6.7}$$

其中 $x^{l-1} \in \mathbb{R}^{n_{l-1}}$ 为 $l-1$ 层的输出，$f^l : \mathbb{R}^{n_l} \to \mathbb{R}^{n_l}$ 为 l 层的激活函数 (ReLU,maxout,maxpooling)，$W^l \in \mathbb{R}^{n_l \times_{l-1}}$ 表示映射 $\mathbb{R}^{n_{l-1}} \to \mathbb{R}^{n_l}$。于是我们可得到 l 层的 Lipschitz 关系:

$$||f^l(W^l x^{l-1} + b^l) - f^l(W^l(x^{l-1} + \epsilon) + b^l)|| \leq C(W^l) \cdot ||\epsilon|| \tag{6.8}$$

当 ϵ 足够小时，左边即可用 Taylor 一阶项来估计，于是:

$$||\frac{\partial f^l}{\partial x^{l-1}} W^l \epsilon|| \leq C(W^l) \cdot ||\epsilon|| \tag{6.9}$$

考虑到神经网络的激活函数导数都是有界的，故可以将其去掉，于是我们得到第 l 层权重参数的 Lispschitz 约束关系 (其他层类似)。我们希望 $C(W)$ 足够小使得我们的模型有足够的泛化能力，其实就是计算 W 的谱范数。因为谱范数的定义:

$$||W||_2 = \max_{x \neq 0} \frac{||Wx||}{||x||} \tag{6.10}$$

我们熟知的 $l1, l2$ 正则化，分别是矩阵的 1 范数，Frobenius 范数 (2 范数)，当然也有 p 范数详见Matrix_norm。从而，可以看出矩阵范数在模型泛化中的位置。根据 (6.10) 容易得到谱范数对应了 $W^T W$ 的最大特征根，而矩阵特征根我们可以用迭代法得到:

$$\boldsymbol{u}^l \leftarrow W^l \boldsymbol{v}^l, \boldsymbol{v}^l \leftarrow (W^l)^T \boldsymbol{u}^l, \sigma^l \leftarrow ||\boldsymbol{u}^l||/||\boldsymbol{v}^l|| \tag{6.11}$$

至此，一般模型的泛化到此结束。

其算法伪代码如下:

Algorithm 1 SGD with spectral norm regularization

for $\ell = 1$ to L **do**
  $\boldsymbol{v}^\ell \leftarrow$ a random Gaussian vector.
for each iteration of SGD **do**
  Consider a minibatch, $\{(\boldsymbol{x}_{i_1}, y_{i_1}), \ldots, (\boldsymbol{x}_{i_k}, y_{i_k})\}$, from training data.
  Compute the gradient of $\frac{1}{k}\sum_{i=1}^{k} L(f_\Theta(\boldsymbol{x}_{i_j}), y_{i_j})$ with respect to Θ.
  for $\ell = 1$ to L **do**
    for a sufficient number of times **do** ▷ One iteration was adequate in our experiments
      $\boldsymbol{u}^\ell \leftarrow W^\ell \boldsymbol{v}^\ell$, $\boldsymbol{v}^\ell \leftarrow (W^\ell)^\top \boldsymbol{u}^\ell$, $\sigma^\ell \leftarrow \|\boldsymbol{u}^\ell\|/\|\boldsymbol{v}^\ell\|$
      Add $\lambda \sigma^\ell \boldsymbol{u}^\ell (\boldsymbol{v}^\ell)^\top$ to the gradient of W^ℓ.
  Update Θ using the gradient.

图 6.9: SNR

Keras 实现: 返回正则项的梯度雅克比矩阵

```
def spectral_normal(w, lamda, r=5):
  w_shape = k.int_shape(w)
  in_dim = np.prod(w_shape[:-1]).astype(int)
  out_dim = w_shape[-1]
  w = K.reshape(w, (in_dim, out_dim))
  v = K.ones((out_dim, 1))
  for i in range(r):
    u = K.dot(w, v)
    v = K.dot(K.transpose(w), u)
    sigma = K.l2_normalize(u) / K.l2_normalize(v)
  return lamda*sigma*(K.dot(u, K.transpose(v)))
```

那么如何将谱范数应用到 GAN 的训练中? 论文 [4] 给出了答案。

回想一下，在上一节中，我们想让 GAN 的生成器更好训练，最终分析得到只需要给判别器增加 Lipschitz 约束即可，而实现方式有截断权重，梯度惩罚，但其方式粗糙，且样本多样化后效果减弱，而接下来就是更为科学的 Lipschitz 约束实现方式: 谱归一化。谱归一化和谱正则化是有区别的，谱正则化希望每一层的上下映射满足 Lipschitz 约束，这点从图 6.8 的伪代码中即可看出，而谱归一化希望最终的函数满足 Lispschitz 约束，这点从 (6.4) 式即可看出，我们约束的是最终的判别函数 D_f。根据递推关系 (见论文 [4] 式 6,7,8) 我们只需要限制每一层的 W 的谱范数为 1 即可，也即:

$$\hat{W}_{SN}(W) := W/\sigma(W). \tag{6.12}$$

从而谱归一化就是将每一层的权重参数除以其谱范数。其 Keras 实现方式:Keras_SN

```
def spectral_norm(w, r=5):
  w_shape = K.int_shape(w)
  in_dim = np.prod(w_shape[:-1]).astype(int)
  out_dim = w_shape[-1]
  w = K.reshape(w, (in_dim, out_dim))
  u = K.ones((1, in_dim))
  for i in range(r):
    v = K.l2_normalize(K.dot(u, w))
    u = K.l2_normalize(K.dot(v, K.transpose(w)))
  return K.sum(K.dot(K.dot(u, w), K.transpose(v)))

def spectral_normalization(w):
  return w / spectral_norm(w)
```

6.2.3 重分析:RGAN

GAN 算法的基本问题，在前两节已有所分析，并给出了相应解决方案。原始 GAN 一出，各种 GAN 文章铺天盖地而来，有滥竽充数的，也有良心的。论文 [5] 重新分析了原始 GAN，指出其模型不如 WGAN 等稳定的关键地方，认为在优化判别器的同时应该降低其对真实样本的概率值，并给出了新的相对判别器，取得当前最好结果。

作者将 GAN 分为两类，non-IPM-GANs: 标准的 SGAN,f-GAN，IPM-GANs: 为解决 GAN 模型不稳定而提出的 WGAN-GP,Fisher GAN, Sobolev GAN，然后从三个角度分析 SGAN 缺失的关键特性: 假样本 $D(x_f)$ 增加的同时，真样本 $D(x_r)$ 应该减小。

- 先验知识论: 第一波训练后 D 将绝大多数样本判别为 $1(D(x)\uparrow)$，这与真假样本各一半矛盾;
- 散度最小化论: 从式 (6.1) 可看出，优化过程仅让 $D(x_f)\uparrow$，这与优化一个纯粹的 JS 散度不一致, 见图 6.10;
- 梯度论: 想让 SGAN 和 IPM-GANS 一致，需要有 $D(x_r)=0$，也即生成器应该要间接影响判别器。

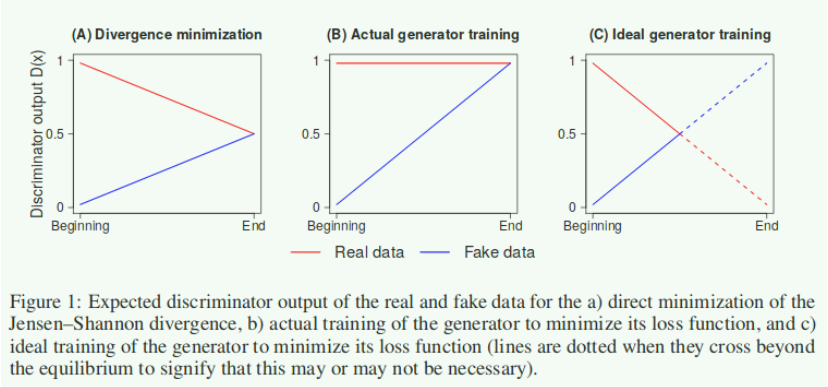


Figure 1: Expected discriminator output of the real and fake data for the a) direct minimization of the Jensen–Shannon divergence, b) actual training of the generator to minimize its loss function, and c) ideal training of the generator to minimize its loss function (lines are dotted when they cross beyond the equilibrium to signify that this may or may not be necessary).

图 6.10: RGAN

因此作者提出将 $D(x)=sigmoid(C(x))$ 替换为 $\hat{x}=(x_r,x_f), D(\hat{x})=sigmoid(C(x_r)-C(x_f))$ 表示判别器将认为真实样本比随机抽取的假样本更真实，那么对应的生成器就是 $1-D(\hat{x})$(因为 sigmoid，$sigmoid(-x)=1-sigmoid(x)$，所以恰好这样)

于是得到最终的优化函数:

$$L_D^{RSGAN} = -\mathbb{E}_{(x_r,x_f)\sim(\mathbb{P},\mathbb{Q})}[log(sigmoid(C(x_r) - C(x_f)))] \qquad (rgan1)$$
$$L_G^{RSGAN} = -\mathbb{E}_{(x_r,x_f)\sim(\mathbb{P},\mathbb{Q})}[log(sigmoid(C(x_f) - C(x_r)))] \qquad (rgan2)$$

写为 (6.1) 的形式即为

$$\min_G \max_D = \mathbb{E}_{(x_r,x_f)\sim(\mathbb{P},\mathbb{Q})}[log(D(\hat{x}))] - \mathbb{E}_{(x_r,x_f)\sim(\mathbb{P},\mathbb{Q})}[log(1 - D(\hat{x}))] \quad (6.13)$$

直接看 (6.13) 式，看不懂，且分析有矛盾，所以还是不合并为好，直接看其伪代码即可。

Algorithm 1 Training algorithm for non-saturating RGANs with symmetric loss functions

Require: The number of D iterations n_D ($n_D = 1$ unless one seeks to train D to optimality), batch size m, and functions f which determine the objective function of the discriminator (f is f_1 from equation 10 assuming that $f_2(-y) = f_1(y)$, which is true for many GANs).

while θ has not converged **do**
  for $t = 1, \ldots, n_D$ **do**
    Sample $\{x^{(i)}\}_{i=1}^m \sim \mathbb{P}$
    Sample $\{z^{(i)}\}_{i=1}^m \sim \mathbb{P}_z$
    Update w using SGD by ascending with $\nabla_w \frac{1}{m} \sum_{i=1}^m \left[f(C_w(x^{(i)}) - C_w(G_\theta(z^{(i)})))\right]$
  end for
  Sample $\{x^{(i)}\}_{i=1}^m \sim \mathbb{P}$
  Sample $\{z^{(i)}\}_{i=1}^m \sim \mathbb{P}_z$
  Update θ using SGD by ascending with $\nabla_\theta \frac{1}{m} \sum_{i=1}^m \left[f(C_w(G_\theta(z^{(i)})) - C_w(x^{(i)}))\right]$
end while

图 6.11: RGAN_pseudocode

论文 [5] 中还提出了 RaGAN，这里略去。其实讲了三节了，其中有个有趣的隐藏点，一直没说: 生成模型除了 GAN 还有更早的 VAE，以及后来的 Glow，GAN 的原始论文也引用了一篇 AE 文章，我们去阅读 VAE 会发现，两者相似度很大，但，为何 GAN 却掀起了一场 GAN 热? 且听下节分解。

6.2.4 VAE

略。

6.2.5 应用

这一节主要讲讲 GAN 和 VAE 在实际应用中的一些现状：医疗数据集:X-ray images 上的异常检测实验 GAN for X-ray anomaly detection 使用网络 Bi-GAN 以及 Alpha-GAN，前者基本无法完成任务，后者根据 L1

损失值的在 area=0.65 情况下的 ROC 能做到检测，但实际准确率未知。Anomaly-XRay-GANs

ADVERSARIAL FEATURE LEARNING 也就是 Bi-GAN, 给 SGAN 网络增加了一个编码器。而后苏剑林在 The relativistic discriminator: a key element missing from standard GAN(2018.9) 的启发下，在此文中 GAN-QP(2018.12) 给出了 GAN 的统一架构:GAN-QP。

GAN 在异常检测的实验上，大概如下文章: GANomaly(2018.5) 此文章在 AnoGAN 和 Efficient-GAN-Anomaly 之后。前两篇文章主要是在比较图像的分布，而此文章主要在编码空间中做比较，其基本出发点为: 对于正常的数据，编码解码再编码得到的潜在空间和第一次编码得到的潜在空间差距不会特别大。但是，在正常样本训练下的 AE 用作于从未见过的异常样本编码解码，再编码后其在潜在空间中的差距应该更大。从论文中的实验结果来看，其效果更佳。

Adversarially Learned Anomaly Detection Efficient-GAN-Anomaly 作者的最新文章 (2018.12). 似乎和 GANomaly 差别不大。

而以上的异常检测文章在实际中用处并不大, 如果你的异常物体占全图的比重很小的话, 那么如何在图像中针对小物体的异常检测, 以及如何针对有大量正样本的情况下, 侦查出出现的未曾见过的负样本, 且其异常物体占图的比例很小? 我觉得可以建立正常图像的网格特征图谱来实现, 但要想得到网格之间的权重, 还得需要少量负样本, 样本之间的网格对齐是个问题.

6.3 Loss Surface

关于神经网络的流形结构，目前很多东西都还不成熟，大多处于实验性的结论或者有很强的假设，以下仅摘录一些论文。

deep fully connected networks with general activation functions and could show that almost every critical point is a global minimum if one layer has more neurons than the number of training point

Understanding the Loss Surface of Neural Networks for Binary Classification(2018)

The Loss Surfaces of Multilayer Networks(2015)

Visualizing the Loss Landscape of Neural Nets(2018)

6.4 其他与 GAN 相关的主题

ACGAN: 在噪音上增加了类别标签用于生成不同类别的图片, 同时在判别器上增加了类别判别输出, 其实就是将分类和生成两个功能合在一起了. 于是分类任务也可以用它来完成.

CVAE-GAN 参考 CVAE-GAN这篇文章说的很清晰, 结合 VAE 的生成能力和 GAN 的判别能力, 将 VAE 的弱判别能力和 GAN 的弱生成能力 (普通 GAN) 替换. 从而实现更稳定的效果.

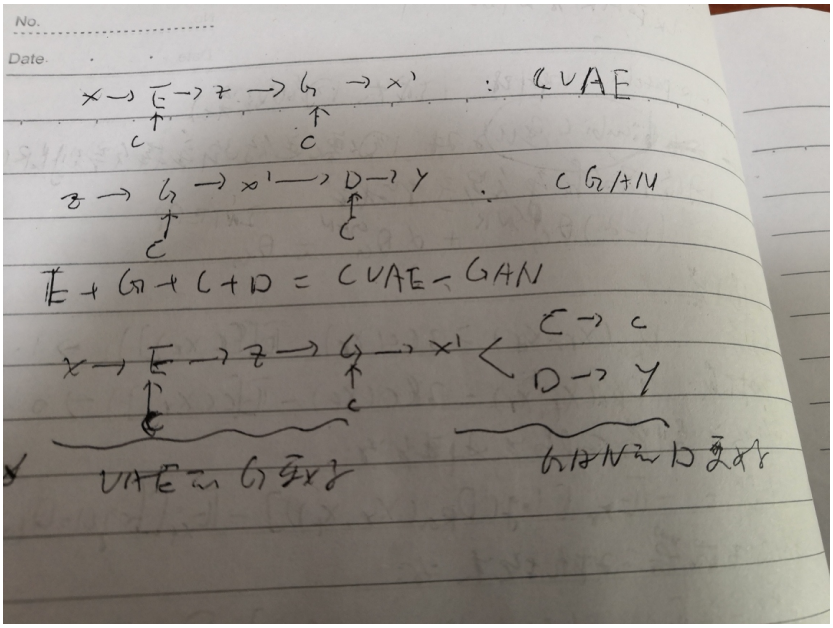


图 6.12: CVAE-GAN

CycleGAN: 实现了两个图域之间的转换. 不太清楚在两类样本的容量有较大差距的时候, 效果会怎样, 待做实验.

StarGAN: 实现了多类图之间的相互转换。这一个我在具体是实验中发现, 它对细小特征以及特征的位置移动的效果还不太好, 当两类之间有增加或减少某些特征的时候, 效果是很容易达到的.

ESRGAN: 超分辨率的性能提升文章, 从中我们能学到如何提升模型性能的一些方法. 比如功能结构的分析和微调, 然后重新设计, 损失函数的重新设计, 以及一些实验中的技巧. 参考ESRGAN (该文损失判别损失函数有误).

超分辨率主要问题点在于提供更尖锐的边缘, 提高生成图片的质量. 从这两点出发以及对 SRGAN 的分析, 作者提出了如下提升方法:

- 网络结构: 生层器增加 RRDB 结构单元, 并去掉 BN 层 (提升纹理);
- 对抗损失: 来源 RaGAN, 提高更丰富的边缘和细节;

- 感知损失: 使用激活前的特征 (为亮度一致和纹理恢复提供更强的监督), 使用材料识别网络预训练模型 (纹理多),
- 残差块之间乘以一个缩减系数, 减小初始化的方差值等实验技巧

问题: 用 RaGAN 生成 256 或者 512 大小的数据, 然后用 ESRGAN 将生成数据提升到 1024 的大小, 不知效果如何.

StyleGAN: 实现了 1024 大小的高分辨率图像生成任务. 时间原因, 其连带的论文就略去了, 先把目前的粗略理解情况梳理一下:

该论文同 ESRGAN 一样, 也是在生成器上做文章, 其主要贡献在于将输入噪声向量经由 8 层的 MLP 映射到另一个分布空间, 因生成网络采用的是 Progressive GANs 的结构, 于是在不同分辨率上分别进行自适应实例归一化, 来达到对隐变量空间的特征功能性表达的实验观察, 于是得出不同层控制不同特征的结论. 因为一般我们在做特征的定向表达都是在生成模型的输入向量中增加类别编码向量, 然后在判别时增加类别损失, 比如 acgan. 但是这里不同, 因为输入的向量是经过一个映射网络变换的, 这样一来, 输入的类别我们知道, 但是在放入生成模型的向量中我们并不知道是哪一块表示了输入类别的信息, 于是作者在论文的 4.2 节 Linear separability 给出了同判别器结构相同的类别预训练分类模型作为评判标准, 另外作者给出了转化后的向量空间 W 与所设计的 AdaIN 结合, 能实现特征的可分离表达的定量方法:Perceptual path length.

其中自适应实例归一化 AdaIN 为:

$$\operatorname{AdalN}\left(\mathbf{x}_i, \mathbf{y}\right)=\mathbf{y}_{s,i} \frac{\mathbf{x}_i-\mu\left(\mathbf{x}_i\right)}{\sigma\left(\mathbf{x}_i\right)}+\mathbf{y}_{b,i} \tag{6.14}$$

$\mathbf{x_i}$ 表示输入特征的第 i 通道,$\mathbf{y}$ 为为风格向量, 由 $\mathcal{W}$ 转换而来.

Perceptual path length 的具体表达式为 (lerp 见原文参考文献):

$$l_{\mathcal{W}}=\mathbb{E}\left[\frac{1}{\epsilon^2} d\left(g\left(\operatorname{lerp}\left(f\left(\mathbf{z}_1\right), f\left(\mathbf{z}_2\right) ; t\right)\right)\right)\right] ., \mathbf{z}_1, \mathbf{z}_2 \sim P(\mathbf{z}), t \sim U(0,1). \tag{6.15}$$

在从如下生成网络的结构图复习一下, 大概认识就差不多了.

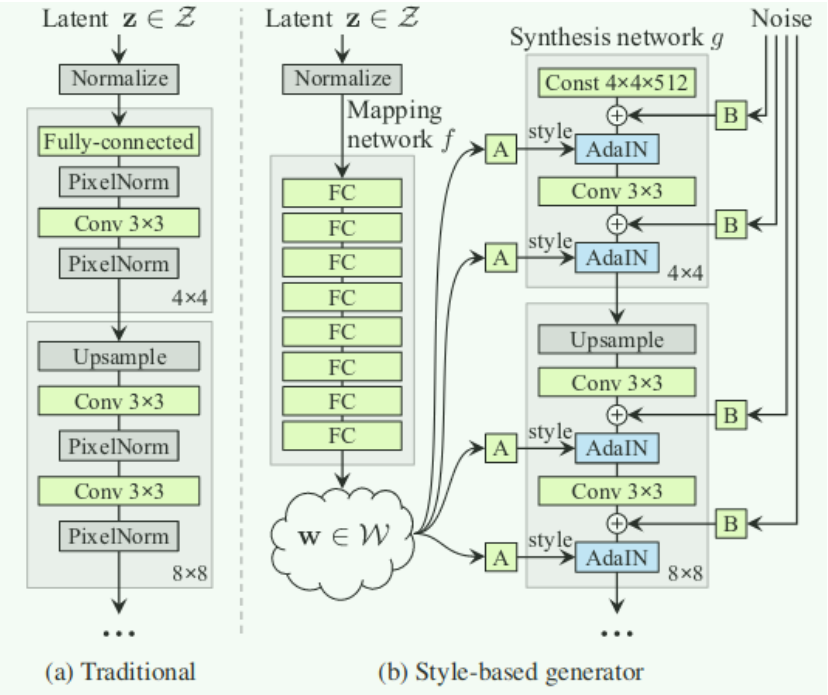


图 6.13: StyleGAN

	Method	CelebA-HQ	FFHQ
A	Baseline Progressive GAN [26]	7.79	8.04
B	+ Tuning (incl. bilinear up/down)	6.11	5.25
C	+ Add mapping and styles	5.34	4.85
D	+ Remove traditional input	5.07	4.88
E	+ Add noise inputs	**5.06**	4.42
F	+ Mixing regularization	5.17	**4.40**

Table 1. Fréchet inception distance (FID) for various generator designs (lower is better). In this paper we calculate the FIDs using 50,000 images drawn randomly from the training set, and report the lowest distance encountered over the course of training.

图 6.14: StyleGAN-test

第七章　实战技巧

本章主要包括深度学习中物体检测的一些必要组件，模型优化的一些经验以及一些最新论文提出的新方法的实验，代码多数来自 github。

7.1　基本代码块

卷积，谱归一化，优化器 AdamW

以下摘录 cnn-numpy中 cnn 层代码实现。相关的从零实现细节后面补充。

```
import numpy  as np
class Convolution():

  def __init__(self, nc_in, nc_out, kernel_size, stride=2,padding=1):
    self.kernel_size = kernel_size
    self.weights = np.random.randn(nc_in * kernel_size[0] * kernel_size[1] ,
    nc_out) * np.sqrt(2/nc_in)
    self.biases = np.zeros(nc_out)
    self.stride = stride
    self.padding = padding

  def forward(self,x):
    mb, ch, n, p = x.shape
    y = np.matmul(arr2vec(x,self.kernel_size,self.stride,self.padding), self
    .weights) + self.biases
    y = np.transpose(y,(0,2,1))
    n1 = (n-self.kernel_size[0]+ 2 * self.padding) //self.stride + 1
    p1 = (p-self.kernel_size[1]+2 * self.padding )//self.stride + 1
    return y.reshape(mb,self.biases.shape[0],n1,p1)

  def backward(self,grad):
    mb, ch_out, n1, p1 = grad.shape
    grad = np.transpose(grad.reshape(mb,ch_out,n1*p1),(0,2,1))
    self.grad_b = grad.sum(axis=1).mean(axis=0)
    self.grad_w = (np.matmul(self.old_x[:,:,:,None],grad[:,:,None,:])).sum(
    axis=1).mean(axis=0)
    new_grad = np.matmul(grad,self.weights.transpose())
```

```
    return vec2arr(new_grad, self.kernel_size, self.old_size, self.stride,
     self.padding)
```

```
def arr2vec(x, kernel_size, stride=1,padding=0):
  k1,k2 = kernel_size
  mb, ch, n1, n2 = x.shape
  y = np.zeros((mb,ch,n1+2*padding,n2+2*padding))
  y[:,:,padding:n1+padding,padding:n2+padding] = x
  start_idx = np.array([j + (n2+2*padding)*i for i in range(0,n1-k1+1+2*
     padding,stride) for j in range(0,n2-k2+1+2*padding,stride) ])
  grid = np.array([j + (n2+2*padding)*i + (n1+2*padding) * (n2+2*padding) *
     k for k in range(0,ch) for i in range(k1) for j in range(k2)])
  to_take = start_idx[:,None] + grid[None,:]
  batch = np.array(range(0,mb)) * ch * (n1+2*padding) * (n2+2*padding)
  return y.take(batch[:,None,None] + to_take[None,:,:])


def vec2arr(x, kernel_size, old_shape, stride=1,padding=0):
  k1,k2 = kernel_size
  n,p = old_shape
  mb, md, ftrs = x.shape
  ch = ftrs // (k1*k2)
  idx = np.array([[[i-k1i, j-k2j] for k1i in range(k1) for k2j in range(k2)]
      for i in range(n) for j in range(p)])
  in_bounds = (idx[:,:,0] >= -padding) * (idx[:,:,0] <= n-k1+padding)
  in_bounds *= (idx[:,:,1] >= -padding) * (idx[:,:,1] <= p-k2+padding)
  in_strides = ((idx[:,:,0]+padding)%stride==0) * ((idx[:,:,1]+padding)%
     stride==0)
  to_take = np.concatenate([idx[:,:,0] * k2 + idx[:,:,1] + k1*k2*c for c in
     range(ch)], axis=0)
  to_take = to_take + np.array([ftrs * i for i in range(k1*k2)])
  to_take = np.concatenate([to_take + md*ftrs*m for m in range(mb)], axis=0)
  in_bounds = np.tile(in_bounds * in_strides,(ch * mb,1))
  return np.where(in_bounds, np.take(x,to_take), 0).sum(axis=1).reshape(mb,
     ch,n,p)
```

谱归一化 (可见六章): 来源于 SPECTRAL NORMALIZATION FOR GENERATIVE ADVERSARIAL NETWORKS , 其主要是了为了 GAN 模型训练的稳定性而提出的. 具体原因在于当优化距离被改为 Wasserstein 度量后, 你需要让判别器满足 Lipschitz 条件. 而谱归一化的 Lipschitz 实现方式当然要比 weight clippingand gradient penalty 来得更漂亮. 算法流程见 7.1.

从图 7.1 可以看到, 实现它只需要将判别器每层的权重矩阵用其最大特征值作一次归一化即可, 而快速得到最大特征值的方式是双参数迭代, 这方面, 快速了解可参考: Spectral Normalization. 另外参考代码:spectral-normal.

Algorithm 1 SGD with spectral normalization

- Initialize $\tilde{\boldsymbol{u}}_l \in \mathcal{R}^{d_l}$ for $l = 1, \ldots, L$ with a random vector (sampled from isotropic distribution).
- For each update and each layer l:
  1. Apply power iteration method to a unnormalized weight W^l:

$$\tilde{\boldsymbol{v}}_l \leftarrow (W^l)^{\mathrm{T}} \tilde{\boldsymbol{u}}_l / \|(W^l)^{\mathrm{T}} \tilde{\boldsymbol{u}}_l\|_2 \tag{20}$$

$$\tilde{\boldsymbol{u}}_l \leftarrow W^l \tilde{\boldsymbol{v}}_l / \|W^l \tilde{\boldsymbol{v}}_l\|_2 \tag{21}$$

  2. Calculate $\bar{W}_{\mathrm{SN}}$ with the spectral norm:

$$\bar{W}^l_{\mathrm{SN}}(W^l) = W^l / \sigma(W^l), \text{ where } \sigma(W^l) = \tilde{\boldsymbol{u}}_l^{\mathrm{T}} W^l \tilde{\boldsymbol{v}}_l \tag{22}$$

  3. Update W^l with SGD on mini-batch dataset $\mathcal{D}_M$ with a learning rate α:

$$W^l \leftarrow W^l - \alpha \nabla_{W^l} \ell(\bar{W}^l_{\mathrm{SN}}(W^l), \mathcal{D}_M) \tag{23}$$

图 7.1: Spectral normalization

AdamW：算法伪代码:

Algorithm 2 Adam with L_2 regularization and Adam with weight decay (AdamW)

given $\alpha = 0.001, \beta_1 = 0.9, \beta_2 = 0.999, \epsilon = 10^{-8}, w \in \mathbb{R}$
initialize time step $t \leftarrow 0$, parameter vector $\boldsymbol{x}_{t=0} \in \mathbb{R}^n$, first moment vector $\boldsymbol{m}_{t=0} \leftarrow \boldsymbol{0}$, second moment vector $\boldsymbol{v}_{t=0} \leftarrow \boldsymbol{0}$, schedule multiplier $\eta_{t=0} \in \mathbb{R}$
repeat
  $t \leftarrow t + 1$
  $\nabla f_t(\boldsymbol{x}_{t-1}) \leftarrow \text{SelectBatch}(\boldsymbol{x}_{t-1})$ ▷ select batch and return the corresponding gradient
  $\boldsymbol{g}_t \leftarrow \nabla f_t(\boldsymbol{x}_{t-1}) + w\boldsymbol{x}_{t-1}$
  $\boldsymbol{m}_t \leftarrow \beta_1 \boldsymbol{m}_{t-1} + (1 - \beta_1)\boldsymbol{g}_t$ ▷ here and below all operations are element-wise
  $\boldsymbol{v}_t \leftarrow \beta_2 \boldsymbol{v}_{t-1} + (1 - \beta_2)\boldsymbol{g}_t^2$
  $\hat{\boldsymbol{m}}_t \leftarrow \boldsymbol{m}_t / (1 - \beta_1^t)$ ▷ β_1 is taken to the power of t
  $\hat{\boldsymbol{v}}_t \leftarrow \boldsymbol{v}_t / (1 - \beta_2^t)$ ▷ β_2 is taken to the power of t
  $\eta_t \leftarrow \text{SetScheduleMultiplier}(t)$ ▷ can be fixed, decay, or also be used for warm restarts
  $\boldsymbol{x}_t \leftarrow \boldsymbol{x}_{t-1} - \eta_t \left(\alpha \hat{\boldsymbol{m}}_t / (\sqrt{\hat{\boldsymbol{v}}_t} + \epsilon) + w\boldsymbol{x}_{t-1} \right)$
until *stopping criterion is met*
return optimized parameters $\boldsymbol{x}_t$

图 7.2: AdamW

参考代码: AdamW

物体检测相关功能函数

物体检测的基本模块：非最大抑制，Rpn 层，RoiAlign 层，框的变换辅助函数等。

```
# IOU计算
# 假设box1维度为[N,4]    box2维度为[M,4]
def iou(self, box1, box2):
    N = box1.size(0)
    M = box2.size(0)

    lt = torch.max(  # 左上角的点
    box1[:, :2].unsqueeze(1).expand(N, M, 2),    # [N,2]->[N,1,2]->[N,M,2]
    box2[:, :2].unsqueeze(0).expand(N, M, 2),    # [M,2]->[1,M,2]->[N,M,2]
    )

    rb = torch.min(
    box1[:, 2:].unsqueeze(1).expand(N, M, 2),
    box2[:, 2:].unsqueeze(0).expand(N, M, 2),
    )

    wh = rb - lt  # [N,M,2]
    wh[wh < 0] = 0    # 两个box没有重叠区域
    inter = wh[:,:,0] * wh[:,:,1]    # [N,M]

    area1 = (box1[:,2]-box1[:,0]) * (box1[:,3]-box1[:,1])  # (N,)
    area2 = (box2[:,2]-box2[:,0]) * (box2[:,3]-box2[:,1])  # (M,)
    area1 = area1.unsqueeze(1).expand(N,M)  # (N,M)
    area2 = area2.unsqueeze(0).expand(N,M)  # (N,M)

    iou = inter / (area1+area2-inter)
    return iou
```

```
# NMS算法
# bboxes维度为[N,4], scores维度为[N,], 均为tensor
def nms(self, bboxes, scores, threshold=0.5):
    x1 = bboxes[:,0]
    y1 = bboxes[:,1]
    x2 = bboxes[:,2]
    y2 = bboxes[:,3]
    areas = (x2-x1)*(y2-y1)   # [N,] 每个bbox的面积
    _, order = scores.sort(0, descending=True)    # 降序排列

    keep = []
    while order.numel() > 0:       # torch.numel()返回张量元素个数
        if order.numel() == 1:     # 保留框只剩一个
            i = order.item()
            keep.append(i)
            break
        else:
            i = order[0].item()    # 保留scores最大的那个框box[i]
            keep.append(i)

        # 计算box[i]与其余各框的IOU(思路很好)
```

```
    xx1 = x1[order[1:]].clamp(min=x1[i])   # [N-1,]
    yy1 = y1[order[1:]].clamp(min=y1[i])
    xx2 = x2[order[1:]].clamp(max=x2[i])
    yy2 = y2[order[1:]].clamp(max=y2[i])
    inter = (xx2-xx1).clamp(min=0) * (yy2-yy1).clamp(min=0)   # [N-1,]

    iou = inter / (areas[i]+areas[order[1:]]-inter)  # [N-1,]
    idx = (iou <= threshold).nonzero().squeeze() # 注意此时idx为[N-1,] 而
    order为[N,]
    if idx.numel() == 0:
      break
    order = order[idx+1]  # 修补索引之间的差值
  return torch.LongTensor(keep)   # Pytorch的索引值为LongTensor
```

最新工作:

MetaAnchor: Guided-Anchor

7.1.1 关于初始化

初始化对寻找更优的极值点来说，极为重要。这节我以几个具体的二元函数为例，来直观说明此问题。而关于神经网络的初始化，对模型性能的提升同样很重要，在同样的优化策略下，好的初始化方式，也许会带来几个百分点的性能提升，相反糟糕的初始化也许让你不经怀疑模型是否有问题等严重影响判断的现象。然而，神经网络的初始化和本节讲的具体的二元函数的初始化还是有其不同的地方，首先，神经是没有具体表达式的，其初始化的是所有可能的函数集的函数的参数，因此不同的初始化方式，将覆盖不同的函数集合，从而也必将影响最终模型所代表的函数与最优函数的差距。

目前深度学习中，初始化方式从 pytorch 官方文档 torch.init中可知大概有 11 种方式，比如 uniform，normal，xavier_uniform，kaiming_uniform，orthogonal，sparse，constant 等。

以下是模型初始化的方式:

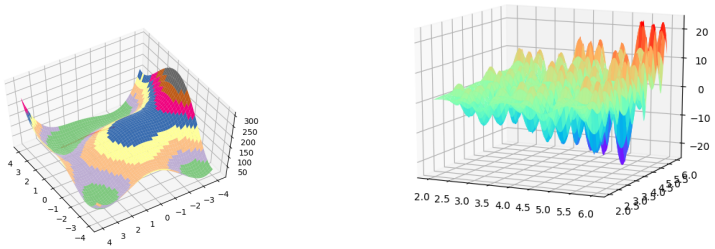
```
import torch.nn.init as init

def get_mean_and_std(dataset):
  '''Compute the mean and std value of dataset.'''
  dataloader = torch.utils.data.DataLoader(dataset, batch_size=1, shuffle=
    True, num_workers=2)
  mean = torch.zeros(3)
  std = torch.zeros(3)
  print('==> Computing mean and std..')
  for inputs, targets in dataloader:
    for i in range(3):
    mean[i] += inputs[:,i,:,:].mean()
    std[i] += inputs[:,i,:,:].std()
  mean.div_(len(dataset))
```

```
  std.div_(len(dataset))
  return mean, std

def init_params(net):
'''Init layer parameters.'''
  for m in net.modules():
    if isinstance(m, nn.Conv2d):
      init.kaiming_normal(m.weight, mode='fan_out')
    if m.bias:
      init.constant(m.bias, 0)
    elif isinstance(m, nn.BatchNorm2d):
      init.constant(m.weight, 1)
      init.constant(m.bias, 0)
    elif isinstance(m, nn.Linear):
      init.normal(m.weight, std=1e-3)
      if m.bias:
        init.constant(m.bias, 0)
```

其中第一块代码期望从数据中获得一定信息，以此来指导网络参数的初始化，具体如何操作，目前推测需要仔细研究网络参数和输入数据被映射后的空间 (不太准确的说法) 之间的关系，方能决定。



(a) himmelblau (b) compleax_peaks

图 7.3: peaks_function

以下即为实验中用到例子:

$$
\begin{aligned}
f(x,y) &= (x^2+y-11)^2+(x+y^2-7)^2 \qquad (himmelblau)\\
f(x,y) &= x^3-3xy^2 \qquad (monkey\_saddle)
\end{aligned}
$$

$$
f(x,y) = 3*(1-x)^2*e^{-x^2-(y+1)^2} - 10*(x/5-x^3-y^5)* e^{-(x^2+y^2)} - \frac{e^{-(x+1)^2-y^2}}{3} - \frac{\sin(x^2+y^2)-cos(x^2+y^2)}{2}
$$

$$f(x,y) = \sin(wx)^2 * \sin(wy)^2 * e^{\frac{x+y}{\sigma^2})} - \\ 2 * \sin(2 * w(x+2))^2 * \sin(2 * w(y+2))^2 * e^{\frac{x+y}{\sigma^2}} + \\ sin(3 * w(x-2))^2 * \sin(3 * w(y-2))^2 * e^{\frac{x+y}{\sigma^2}} \\ (compleax\_peaks)$$

简单的实验代码:

```
import numpy as np
from matplotlib import pyplot as  plt
from mpl_toolkits.mplot3d import Axes3D
import torch

def himmelblau(x):
  return (x[0]**2 + x[1] -11)**2 + (x[0] + x[1]**2 - 7)**2

def monkey_saddle(x):
  return (x[0]**3 - 3*x[0]*x[1]**2)

def complex_peaks(x, w=2, sigma=2):
  return torch.sin(w*x[0])**2 * torch.sin(w*x[1])**2 * torch.exp((x[0]+x[1])
    /sigma**2) - \
  2*torch.sin(2*w*(x[0]+2))**2 * torch.sin(2*w*(x[1]+2))**2 * torch.exp((x
    [0]+x[1])/sigma**2) + \
  torch.sin(3*w*(x[0]-2))**2 * torch.sin(3*w*(x[1]-2))**2 * torch.exp((x[0]+
    x[1])/sigma**2)

x = np.arange(-4, 4, 0.01)
y = np.arange(-4, 4, 0.01)
# for complex_peaks function :
#x = np.linspace(2, 6, 400)
#y = np.linspace(2, 6, 400)
X, Y = np.meshgrid(x, y)
# print('X, Y maps:', X, Y)

# Z = himmelblau([X, Y])
# Z = monkey_saddle([X, Y])
Z = many_peak_bumps([X, Y])
#Z = complex_peaks([torch.from_numpy(X), torch.from_numpy(Y)]).numpy()

# print(Z.shape, Z)
fig = plt.figure('himmelblau')
ax = fig.gca(projection='3d')
ax.plot_surface(X, Y, Z, cmap='Accent')


grad_x_sets = []
grad_y_sets = []
```

```
grad_z_sets = []
x_0 = torch.tensor([0.,  0.], requires_grad=True)
# optimizer = torch.optim.SGD([x_0], lr = 0.01)
optimizer = torch.optim.SGD([x_0], lr = 0.01, momentum=0.05)
# optimizer = torch.optim.RMSprop([x_0], lr=10, alpha=0.9)
# optimizer = torch.optim.RMSprop([x_0], lr=10, alpha=0.9, momentum=0.85)
# optimizer = torch.optim.Adam([x_0], betas=(0.9, 0.99), lr=20)
#lr_scheduler = CyclicLR(optimizer, base_lr=1e-2, max_lr=1e-0)
for step in range(100):
  pred = himmelblau(x_0)

  optimier.zero_grad()
  #lr_scheduler.batch_step()
  #print('lr:', lr_scheduler.get_lr())
  pred.backward()
  optimier.step()

  grad_x_sets.append(x_0.tolist()[0])
  grad_y_sets.append(x_0.tolist()[1])
  grad_z_sets.append(pred.tolist())

  if step % 2==1:
    print('step {}: x = {}, f(x) = {}'.format(step, x_0.tolist(), pred.item
    ()))

ax.plot(grad_x_sets, grad_y_sets, grad_z_sets,c='black', lw=4, label='
    gradient decent curve')
plt.show()
```

关于 himmelblau 函数，容易算出其四个极小值点为 (3,2), (-3.7794,-3.2832), (3.5845, -1.8481), (-2.8050, 3.1313) 四极值点的中心点几乎是 $(0,0)$ 点，明显 (0, 0) 点于 (3, 2) 点最近，于是初始化值若为 (0, 0)，则优化到的极值点一般就是 (3, 2)(图例 himmelblau 中红色线条为带动量的 SGD 优化器优化路线), 但是当你学习率较大，比如 lr=10, 使用 adam，则结果为第二个极值点，lr=20 时，结果为第三个极值点等。对于 complex_peaks，观察其在 $[2,6]\times[2,6]$ 区域内的图形，容易看出极值点密集，选取不同初始值并使用不同的优化器，你会发现很难找到全局的极值点，几乎都停留在初始值附近的波谷之中。也许神经网络的极值点也大概如此，只是波峰波谷之间的差距没有 complex_peaks 函数来得大，但我们仍然要适当跳出局部极值点，向全局极值点逼近，关于这个问题，我想下一节的动态调整学习率，会有一定的帮助 (事实上对鞍点的帮助更大，这点我将利用下一节的策略实验上面给出的 monkey_saddle 函数)。

7.1.2 关于学习率

Cyclical Learning Rates for Training Neural Networks 最佳初始学习率，学习率的变化方式: 分步衰减，余弦式衰减，周期性变化寻找最佳初始化学习率，参考：find_lr 将其翻译成 pytorch1.0 版本如下：

```
def find_lr(train_loader, optimizer, criterion, net, init_value = 1e-8,
    final_value=10., beta = 0.98):
  num = len(train_loader)-1
  mult = (final_value / init_value) ** (1/num)
  lr = init_value
  optimizer.param_groups[0]['lr'] = lr
  avg_loss = 0.
  best_loss = 0.
  batch_num = 0
  losses = []
  log_lrs = []
  for data in train_loader:
    batch_num += 1
    #As before, get the loss for this mini-batch of inputs/outputs
    inputs,labels = data
    inputs, labels = inputs.to(device), labels.to(device)
    optimizer.zero_grad()
    outputs = net(inputs)
    loss = criterion(outputs, labels)
    #Compute the smoothed loss
    avg_loss = beta * avg_loss + (1-beta) *loss.item()
    smoothed_loss = avg_loss / (1 - beta**batch_num)
    #Stop if the loss is exploding
    if batch_num > 1 and smoothed_loss > 4 * best_loss:
      return log_lrs, losses
    #Record the best loss
    if smoothed_loss < best_loss or batch_num==1:
      best_loss = smoothed_loss
    #Store the values
    losses.append(smoothed_loss)
    log_lrs.append(math.log10(lr))
    #Do the SGD step
    loss.backward()
    optimizer.step()
    #Update the lr for the next step
    lr *= mult
    optimizer.param_groups[0]['lr'] = lr
  return log_lrs, losses
```

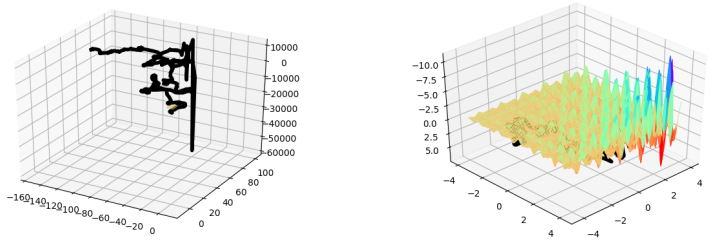
关于学习率的更新问题，很明显，如果鞍点太多，局部极值点太多，很容易陷入局部极值点或者鞍点 (后期学习率太小)，所以让学习率以梯田式的周期变化，也许是一个办法 (后面做了实验，感觉对梯度方向的周期性突变，并整合信息，进行更新，或许才是达到全局最优的有效策略之一)。这

方面 tf 有更为完整的策略 api，参考：tf 学习率更新策略。这方面，pytorch 显得单薄一些，其学习率更新方式大概如下集中 (较为有用)：

- torch.optim.lr_scheduler.ReduceLROnPlateau
- torch.optim.lr_scheduler.ExponentialLR
- torch.optim.lr_scheduler.CosineAnnealingLR

其中第一条和 keras 中相同，从提升模型在数据上的性能的来说似乎很有用，其参数 mode 有 min 和 max 两种模式，min 表示当指标不再降低 (如监测 loss)，max 表示当指标不再升高 (如监测 accuracy)；threshold_mode(str)-选择判断指标是否达最优的模式，也有两种模式，rel 和 abs，其和 mode 模式有关系，具体关系查源码文档，其他参数意义明显。要在 pytorch 中实现其他类似 tf 中的更新策略参考其源码即可。比如循环更新学习率的方法类可参考:CycleLR。

现在用 CycleLR，来测试一下上节所给的函数 (代码见上节)，实验参数略，实验结果总结起来大致有：针对 complex_peak 函数，当学习率 < 1 时，除 SGD 外各个优化器都很难跳出局部最优；学习率过小且带动量，则连局部极值点都很难收敛到；学习率大于一定阈值，各优化器均可以跳出去... 对于 himmelblau 函数，当其收敛到某一极值点后，各优化器在循环学习率更新策略下均很难跳出去。以下为梯度更新方向示例图。



(a) rmsprop_momentum (b) sgd_mometum

图 7.4: decent_curve

7.1.3 设计损失函数

损失函数的设计是一个难点。基本上使用 pytorch 官方文档中的函数，这里给出一些根据框架写新的损失函数需要注意的一些细节。主要以 pytorch 官方版本 maskrcnn 为基础，进行说明。

7.1.4 设计正则化

理论参考说明:L1,L2 正则化 pytorch 的实现，keras 的实现 pytorch 的正则化是很简单的，只需将 optimizer 中的参数 weight_decay 设置一个非负的系数即可。keras 中可参考 Matterport 的 mask-rcnn 中的代码，具体摘录如下:

```
import keras
import tensorflow as tf

def compile(self, learning_rate, momentum):
  """Gets the model ready for training. Adds losses, regularization, and
  metrics. Then calls the Keras compile() function.
  """
  # Optimizer object
  optimizer = keras.optimizers.SGD(
  lr=learning_rate, momentum=momentum,
  clipnorm=self.config.GRADIENT_CLIP_NORM)
  # Add Losses
  # First, clear previously set losses to avoid duplication
  self.keras_model._losses = []
  self.keras_model._per_input_losses = {}
  loss_names = [
  "rpn_class_loss",  "rpn_bbox_loss",
  "mrcnn_class_loss", "mrcnn_bbox_loss", "mrcnn_mask_loss"]
  for name in loss_names:
    layer = self.keras_model.get_layer(name)
    if layer.output in self.keras_model.losses:
      continue
    loss = (
    tf.reduce_mean(layer.output, keepdims=True)
    * self.config.LOSS_WEIGHTS.get(name, 1.))
    self.keras_model.add_loss(loss)

  # Add L2 Regularization
  # Skip gamma and beta weights of batch normalization layers.
  reg_losses = [
  keras.regularizers.l2(self.config.WEIGHT_DECAY)(w) / tf.cast(tf.size(w),
    tf.float32)
  for w in self.keras_model.trainable_weights if 'gamma' not in w.name and '
    beta' not in w.name]
  self.keras_model.add_loss(tf.add_n(reg_losses))

  # Compile
  self.keras_model.compile(
  optimizer=optimizer,
  loss=[None] * len(self.keras_model.outputs))

  # Add metrics for losses
  for name in loss_names:
```

```
42      if name in self.keras_model.metrics_names:
43        continue
44      layer = self.keras_model.get_layer(name)
45      self.keras_model.metrics_names.append(name)
46      loss = (
47      tf.reduce_mean(layer.output, keepdims=True)
48      * self.config.LOSS_WEIGHTS.get(name, 1.))
49      self.keras_model.metrics_tensors.append(loss)
```

或者在构建网络时，增加参数 kernel_regularizer=regularizers.l2(0.01)，activity_regularizer，这点参考 keras 官方文档，此处 kernel_regularizer 接收一个将权重映射成一个常量的函数，因此可以自行设计。

7.1.5 经典网络结构的微调

主要参考李沐等人的论文:Bag of Tricks for Image Classification 代码示例:

7.1.6 性能评估

- AUC 曲线：主要调用 sklearn.
- 生成模型的评估: inception, FID 等, 参考:Gan-evaluate
- 物体检测:mAP 等，参考 pytorch 官方版本的 maskrcnn.
  物体检测的性能指标 mAP: 中文版参考:mAP 理解 主要点:
  1 查全率:$\frac{TP}{TP+FN}$ 查准率:$\frac{TP}{TP+FP}$
  2 物体置信度作为样本类别正负判断标准，IOU 分数作为位置正误检测的标准
  3 最终的 TP,FP,FN 数值是类别分数和 IOU 分数的联合效果，类似于分别将其看做 x, y 轴
  4 平均就是平均各个类别的 AP, 单类 AP 就是算 precision-recall 曲线先的面积

其中算 p-r 曲线下的面积的方式是灵活的，主要有 VOC07，VOC10，COCO 以及标准方式。代码可参考 Detertron 以及后来的 maskrcnn，COCO 计算方式核心在：cocoeval，该计算方法分别统计了不同尺度样本的 mAP，详解可参考 cocoeval 解析。

生成模型的评估方式很多，有些合理与不合理，并不好说明，合理性分析可参考尹相楠博士的博文:Inception Score 的原理和局限性,FID 的分析，完整的性能实验见上面的链接 Gan-evaluate. 这里简单说说数学性较强的 FID 指标：FID 卷积的理解：Conv-Matrix

7.2 加速

训练

单卡多 gpu，以及多卡多 gpu 的分布式，主要集中在 keras 和 pytorch 的代码经验上。Pytorch 优化加速

最大化单卡训练的 batch-size(延迟更新参数):

```
model.zero_grad()                                    # Reset gradients
    tensors
for i, (inputs, labels) in enumerate(training_set):
  predictions = model(inputs)                        # Forward pass
  loss = loss_function(predictions, labels)          # Compute loss function
  loss = loss / accumulation_steps                   # Normalize our loss (if
    averaged)
  loss.backward()                                    # Backward pass
  if (i+1) % accumulation_steps == 0:                # Wait for several
    backward steps
    optimizer.step()                                 # Now we can do an optimizer
     step
    model.zero_grad()                                # Reset gradients tensors
    if (i+1) % evaluation_steps == 0:                # Evaluate the model when we
    ...
      evaluate_model() # ...have no gradients accumulated
```

单机多卡 DataParallel 模式，缺点是 gpu 利用不平衡。

```
parallel_model = torch.nn.DataParallel(model) # Encapsulate the model

predictions = parallel_model(inputs)           # Forward pass on multi-GPUs
loss = loss_function(predictions, labels)      # Compute loss function
loss.mean().backward()                         # Average GPU-losses +
    backward pass
optimizer.step()                               # Optimizer step
predictions = parallel_model(inputs) # Forward pass with new parameters
```

具体过程见下图:

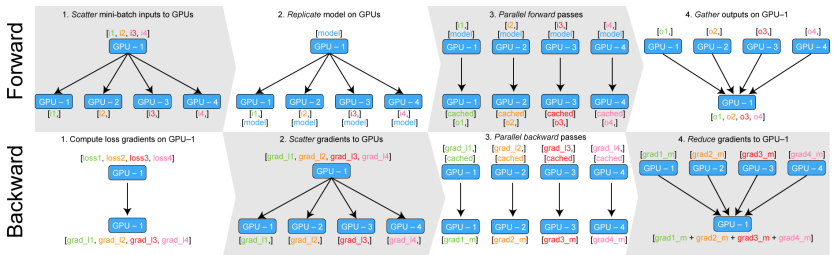


图 7.5: gpu not banalance

单机多卡均匀利用模式：分布式计算损失函数

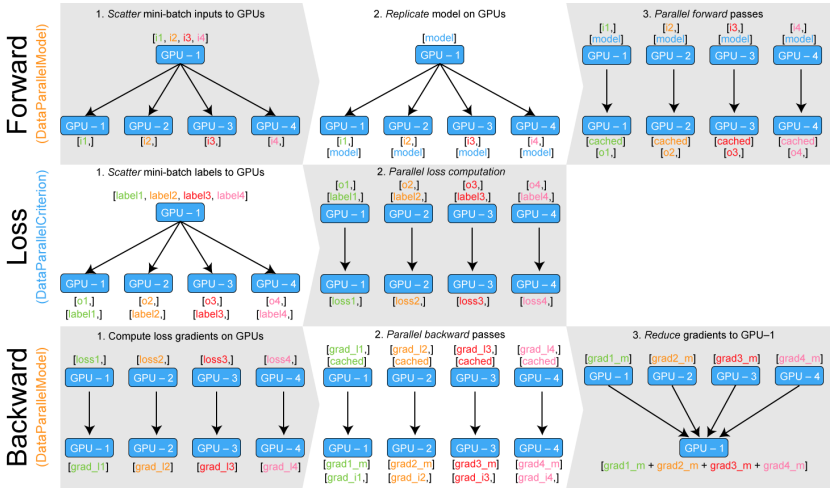


图 7.6: gpu banalance

比较以上两图，即可看出区别。代码实现：

```
from parallel import DataParallelModel, DataParallelCriterion

' '' parallel.py 来源：
https://gist.github.com/thomwolf/7e2407fbd5945f07821adae3d9fd1312
 '' '

parallel_model = DataParallelModel(model)              # Encapsulate the
    model
parallel_loss  = DataParallelCriterion(loss_function) # Encapsulate the loss
     function

predictions = parallel_model(inputs)       # Parallel forward pass
# "predictions" is a tuple of n_gpu tensors
loss = parallel_loss(predictions, labels) # Compute loss function in
    parallel
loss.backward()                            # Backward pass
optimizer.step()                           # Optimizer step
predictions = parallel_model(inputs) # Parallel forward pass with new
    parameters
```

那么这个和官方的分布式有什么区别呢？下次再说。

预测

一些时间分析模块：

7.3 如何让模型表现更好

主要思路包括：图像预处理，自动增强，初始化，结构微调整，损失函数的设计，优化器的选择，学习率的动态更新，模型集成，超参数调节，结合强化学习自动搜索模型结构。

完成这一节，主要还是将之前的小节不断完善即可，这一节主要就是在项目中去完成，以上是一些比较典型的思路，比较慢才能完成的思路包括，理论的升级，对模型的改进，提出更好的算法等。具体例子比如，物体检测提升框的性能，先后是 guided-anchor,meta-anchor,iou-net 等文章来提升框的准确度；超分辨率生成例子，比如 pggan 到 stylegan，比如 srgan 到 esrgan 等都是从模型结构的功能设计上重新设计和组装；人脸识别损失函数的改进系列等从理论上进行分析，这些都是较难的，我们能做的就是将学术界此类文章快速消化，并集成到已有的模块中，达到模型性能提升的效果。

第八章　智慧城市

瞎想