##

调参经验

- 学习率 3e-4
- 数据归一化
- L1 L2正则化 / weight decany
- DropOut2d / DropPath / DropBlock(Dropout对卷积层的效果不好；block size控制大小最好在7x7，；keep prob在整个训练过程中从1逐渐衰减到指定阈值比较好)
- Batch Normalization / Group Normalization (每组channel为16)
- BatchSize(大batchsize对小物体好，梯度累积)
- OneCycleLR + SGD / Adam (torch.optim.lr_scheduler.ReduceLROnPlateau)
- Warm Up / Early stopping
- Multi Scale Training
- ResNet / DenseNet(DenseNet训练样本比较少的情况下收敛的比ResNet更好)
- 3x3卷积（有利于保持图像性质）
- 卷积核权重初始化使用xavier（Tanh,Zé wéi'ěr）或者He normal（ReLU）
- cv2读取图片速度快比Pillow快
- 加速训练pin_memory=true work_numbers=x(卡的数量x4) data.to(device, no_blocking=True),设置为True后，数据直接保存在锁页内存中，后续直接传入cuda；否则需要先从虚拟内存中传入锁页内存中，再传入cuda
- ReLU可使用inplace操作减少显存消耗
- Focal Loss：对CE loss增加了一个调制系数来降低容易样本的权重值，使得训练过程更加关注困难样本
- With Flooding:当training loss大于一个阈值时，进行正常的梯度下降；当training loss低于阈值时，会反过来进行梯度上升，让training loss保持在一个阈值附近，让模型持续进行"random walk"
- Label Smoothing：使得原本的hard-target变为soft-target，让标签分布的熵增大,使网络优化更加平滑,通常用于减少训练的过拟合问题并进一步提高分类性能

数据增强

- Mix up / Cutout / Mosaic
- Label Smoothing：使得原本的hard-target变为soft-target，让标签分布的熵增大
- 物体的复制粘贴（小物体）
- 随机剪裁，翻转，缩放，亮度，色调，饱和度
- 对普通数码照片进行归一化，可以简单的将0-255线性映射到0-1；而医学图像、遥感图像则不能简单的利用最小最大像元值归一化到0-1；

拼接增广指随机找几张图各取一部分或者缩小之后拼起来作为一幅图用，拼接出来的图有强烈的拼接痕迹；

抠洞指随机的将目标的一部分区域扣掉填充0值；

- **拼接、抠洞属于人为制造的伪显著区域，不符合实际情况，对工程应用来说没有意义，白白增加训练量；**
- **训练过程随机缩放也是没必要的，缩放之后的图像可能会导致特征图和输入图像映射错位；**

网络CheckList

1. 从最简单的数据/模型开始，全流程走通

模型简单：解决一个深度学习任务，最好是先自己搭建一个最简单的神经网络
数据简单： 一般来说少于**10个样本**做调试足够了，一定要**做过拟合测试**(如果你的模型无法在7、8个样本上过拟合，要么模型参数实在太少，要么有模型有bug，要么数据有bug),多选几个有代表性的输入数据有助于直接测试出非法数据格式。但数据太多模型就很难轻松过拟合了

2.选择合理的loss/评价指标，检查一下loss是否符合预期

- **初始loss期望值和实际值误差是否过大：多分类例子**：CIFAR-10用Softmax Classifier进行10分类，那么一开始每个类别预测对的概率是0.1（随机预测），Softmax loss使用的是negative log probability，所以正确的loss大概是：-ln（0.1）= 2.303左右；**二分类例子**：假设数据中有20%是标签是0，80%的标签是1，那么一开始的loss大概应该是-0.2ln（0.5）-0.8ln（0.5）=0.69左右，如果一开始loss比1还大，那么可能是**模型初始化不均匀**或者**数据输入没有归一化；**
- **多任务学习**的时候，**多个loss相加，那这些loss的数值是否在同一个范围；**
- **数据不均衡的时候尝试Focal Loss；**

3.网络中间输出检查、网络连接检查

- 确认所有子网络的输入输出**shape对齐**，并确认全部都连接上了，可能有时候定义一个子网络，但放一边忘记连入主网络；
- **梯度更新是否正确？** 如果某个参数没有梯度，那么是不是没有连入主网络； 有时候我们会通过参数名字来设置哪些梯度更新，哪些不更新，是否有误操作；

4. 时刻关注着模型参数

所谓模型参数也就是一堆矩阵/或者说大量的数值，如果这些数值中有些数值异常大/小，那么模型效果一般也会出现异常；

5. 详细记录实验过程

- MarkDown记录；

有效阅读PyTorch源码

- **项目背景调研 + Paper**
- 阅读项目说明文档 + **README**
- 通过文件命名分析：**数据处理、数据加载**部分，通常命名可能有xxx_dataloader.py等；**网络模型**构建部分，通常命名可能为 resnet20.py model.py等；**训练部分**脚本，通常命名可能为train.py等；**测试部分**脚本，通常命名可能为test.py eval.py 等；**工具库**，通常命名为utils文件夹；
- **找到项目运行的主入口**，比如train.py运行：3类BUG：环境不兼容；深度学习框架；项目本身相关的BUG，这类bug最好是在github的issue区域进行查找，如果无法解决可以在issue部分详细描述自己的问题，等待项目库作者的解答；
- **用IDE打开项目**：阅读入口文件的逻辑，查看调用到了哪些 - 通过IDE的功能跳转到对应类或者函数进行继续阅读，配合代码注释进行分析 - 分析过程可能会需要软件工程的知识，比如画一个类图来表述项目的关系 - 一开始可以泛读，大概了解整体流程，做一些代码注释。而后可以精读，找到文章的核心，反复理解核心实现；

##

PyTorch

默认梯度累积

- 机器显存小，可以变相增大batchsize；
- weight在不同模型之间交互时候有好处；（动手学习深度学习v2）

```
accumulation_steps = batch_size // opt.batch_size

loss = loss / accumulation_steps
running_loss += loss.item()
loss.backward()

if ((i + 1) % accumulation_steps) == 0:
    optimizer.step()
    scheduler.step()
    optimizer.zero_grad()
```

PyTorch提速

- **图片解码**：cv2要比Pillow读取图片速度快
- 加速训练**pin_memory=true / work_numbers=x(卡的数量x4) / prefetch_factor=2 / data.to(device，no_blocking=True)**

- **DALI库**在GPU端完成这部分**数据增强**，而不是**transform**做图片分类任务的数据增强
- OneCycleLR + SGD / AdamW
- `torch.nn.Conv2d(..., bias=False, ...)`
- DP & DDP
- 不要频繁在CPU和GPU之间转移数据
- `torch.backends.cudnn.benchmark = True`
- `from torch.cuda import amp`使用`FP16`

Module & Functional

nn.Module实现的layer是由class Layer(nn.Module)定义的特殊类，**会自动提取可学习参数 nn.Parameter**

nn.Functional中的函数更像是**纯函数**，由def function(input)定义，一般只定义一个操作，因为其无法保存参数

Function需要定义三个方法：**init, forward, backward**（需要自己写求导公式）

Module只需定义 **init**和**forward**，而backward的计算由自动求导机制

对于激活函数和池化层，由于没有可学习参数，一般使用nn.functional完成，其他的有学习参数的部分则使用nn.Module

但是**Droupout**由于在训练和测试时操作不同，所以**建议使用nn.Module实现**，它能够通过**model.eval**加以区分

- [x] https://blog.csdn.net/wzy_zju/article/details/81262472?utm_medium=distribute.pc_relevant.none-task-blog-baidujs_title-0&spm=1001.2101.3001.4242
- [x] https://blog.csdn.net/andyjkt/article/details/107428618

Sequential & ModuleList

区别：

- **nn.Sequential内部实现了forward函数**，因此可以不用写forward函数（或者继承nn.Module类的话，就要写出forward函数）；而**nn.ModuleList则没有实现内部forward函数**；
- **nn.Sequential可以使用OrderedDict对每层进行命名**；
- **nn.Sequential里面的模块按照顺序进行排列的**，所以必须确保前一个模块的输出大小和下一个模块的输入大小是一致的。而**nn.ModuleList 并没有定义一个网络，它只是将不同的模块储存在一**

起，这些模块之间并没有什么先后顺序可言。

nn.Sequential

- nn.Sequential里面的模块按照顺序进行排列的，所以必须确保前一个模块的输出大小和下一个模块的输入大小是一致的；
- nn.Sequential中可以使用OrderedDict来指定每个module的名字，而不是采用默认的命名方式；

```
import torch
import torch.nn as nn
import torch.nn.functional as F
class net_seq(nn.Module):
    def __init__(self):
        super(net2, self).__init__()
        self.seq = nn.Sequential(
                        nn.Conv2d(1,20,5),
                         nn.ReLU(),
                          nn.Conv2d(20,64,5),
                        nn.ReLU()
                        )
    def forward(self, x):
        return self.seq(x)
net_seq = net_seq()
```

```
from collections import OrderedDict

class net_seq(nn.Module):
    def __init__(self):
        super(net_seq, self).__init__()
        self.seq = nn.Sequential(OrderedDict([
                         ('conv1', nn.Conv2d(1,20,5)),
                          ('relu1', nn.ReLU()),
                           ('conv2', nn.Conv2d(20,64,5)),
                        ('relu2', nn.ReLU())
                        ]))
    def forward(self, x):
        return self.seq(x)
net_seq = net_seq()
```

nn.ModuleList

- nn.ModuleList，它是一个储存不同 module，并自动将每个 module 的 parameters 添加到网络之中的容器。你可以把任意 nn.Module 的子类 (比如 nn.Conv2d, nn.Linear 之类的) 加到这个 list 里面，方法和 Python 自带的 list 一样，无非是 extend，append 等操作。但不同于一般的 list，加入到 nn.ModuleList 里面的 module 是会自动注册到整个网络上的，同时 module 的 parameters 也会自动添加到整个网络中。使用 Python 的 list 添加的卷积层和它们的 parameters 并没有自动注册到我们的网络中；
-

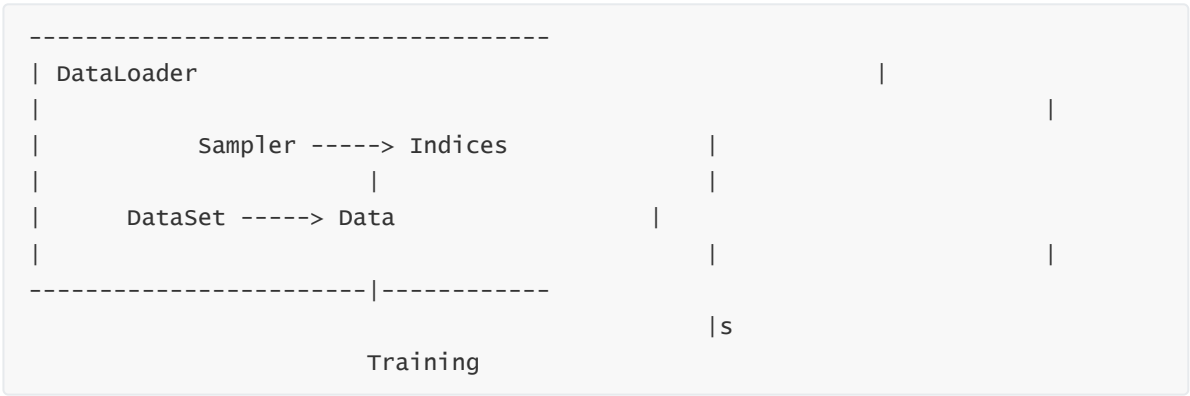
```
class net_modlist(nn.Module):
    def __init__(self):
        super(net_modlist, self).__init__()
```

```
        self.modlist = nn.ModuleList([
                        nn.Conv2d(1, 20, 5),
                        nn.ReLU(),
                         nn.Conv2d(20, 64, 5),
                         nn.ReLU()
                         ])

    def forward(self, x):
        for m in self.modlist:
            x = m(x)
        return x

net_modlist = net_modlist()
```

DataLoader & Sampler & DataSet

假设我们的数据是一组图像，每一张图像对应一个index，那么如果我们要读取数据就只需要对应的index即可，即上面代码中的`indices`，而选取index的方式有多种，有按顺序的，也有乱序的，所以这个工作需要`Sampler`完成，`DataLoader`和`Sampler`在这里产生关系

我们已经拿到了indices，那么下一步我们只需要根据index对数据进行读取即可了，这时`Dataset`和`DataLoader`产生关系



DataLoader 的源代码初始化参数里有两种sampler：`sampler`和`batch_sampler`，都默认为`None`；前者的作用是生成一系列的index，而batch_sampler则是将sampler生成的indices打包分组，得到batch的index

Pytorch中已经实现的`Sampler`有如下几种：`SequentialSampler` `RandomSampler` `WeightedSampler` `SubsetRandomSampler`

- 如果你自定义了`batch_sampler`,那么这些参数都必须使用默认值：`batch_size`,`shuffle`,`sampler`,`drop_last`.
- 如果你自定义了`sampler`，那么`shuffle`需要设置为`False`

- 如果`sampler`和`batch_sampler`都为`None`,那么`batch_sampler`使用Pytorch已经实现好的`BatchSampler`,而`sampler`分两种情况:
  - 若`shuffle=True`,则`sampler=RandomSampler(dataset)`
  - 若`shuffle=False`,则`sampler=SequentialSampler(dataset)`

Dataset定义方式如下:

```
class Dataset(object):
    def __init__(self):
        ...

    def __getitem__(self, index): # 能让该类可以像list一样通过索引值对数据进行访问
        return ...

    def __len__(self):
        return ...
```

- [x] https://www.cnblogs.com/marsggbo/p/11308889.html

Model.Eval & Torch.No_Grad

两者都在Inference时候使用，但是作用不相同:

```
model.eval() 负责改变batchnorm、dropout的工作方式，如在eval()模式下，dropout是不工作的;
torch.no_grad() 负责关掉梯度计算，节省eval的时间;
```

只进行Inference时，`model.eval()`是必须使用的，否则会影响结果准确性； 而`torch.no_grad()`并不是强制的，只影响运行效率;

##

深度学习

ResNet & DenseNet

NiN: Conv + 1x1 Conv + 1x1 Conv

ResNet和DenseNet比较

ResNet	稀疏连接	Add	训练速度快	参数量相对较多
DenseNet	密集连接	Concat	训练速度慢（concat需要频繁读取内存）	参数量相对较少

- **DenseNet比传统的卷积网络所需要的参数更少：1.密集连接带来了特征重用**，不需要重新学习冗余的特征图，**2.维度拼接的操作，带来了丰富的特征信息**，利用更少的卷积就能获得很多的特征图;
- **DenseNet提升了整个网络中信息和梯度的流动，对训练十分有利：**密集连接使得**每一层都可以直接从损失函数和原始输入信号获得梯度**，对于训练更深的网络十分有利；
- **密集连接的网络结构有正则化的效果，能够减少过拟合风险**；
- **对显存需求**

ResNet解决了什么问题

- **网络性能退化能力：**单纯的堆积网络正确率不升反降：按理说，当我们堆叠一个模型时，理所当然的会认为效果会越堆越好。因为，假设一个比较浅的网络已经可以达到不错的效果，那么即使之后堆上去的网络什么也不做，模型的效果也不会变差。然而事实上，这却是问题所在。**“什么都不做”恰好是当前神经网络最难做到的东西之一； -> 恒等映射**
- **有效减少梯度相关性的衰减**：即使BN过后梯度的模稳定在了正常范围内，但梯度的相关性实际上是随着层数增加持续衰减的；
  对于L层的网络来说，没有残差表示的Plain Net梯度相关性的衰减在1 / 2^L ，而ResNet的衰减却只有 1 / sqrt(L)
- **稳定梯度**：在输出引入一个输入x的恒等映射，则梯度也会对应地引入一个常数1，这样的网络的确不容易出现梯度值异常，在某种意义上，起到了**稳定梯度**的作用；
- **shortcut相加可以实现不同层级特征的组合：**因为浅层容易有高分辨率但是低级语义的特征，而深层的特征有高级语义，但分辨率就很低了；

ResNet两种结构实现，BottleNeck作用

- **两个3x3卷积和一个shortcut**
- **两个1x1卷积中间加一个3x3卷积，然后再加一个shortcut**
- **BottleNeck作用：降低维度，模型压缩，减少计算量**

DenseNet和ResNet哪个比较好

在小数据集，DenseNet比ResNet要好，因为小数据集的时候容易产生过拟合，但是DenseNet能够很好的解决过拟合的问题。DenseNet 具有非常好的抗过拟合性能，尤其适合于训练数据相对匮乏的应用。这一点从论文中 DenseNet 在不做数据增强的 CIFAR 数据集上的表现就能看出来。**对于 DenseNet 抗过拟合的原因有一个比较直观的解释：神经网络每一层提取的特征都相当于对输入数据的一个非线性变换，而随着深度的增加，变换的复杂度也逐渐增加（更多非线性函数的复合）。相比于一般神经网络的分类器直接依赖于网络最后一层（复杂度最高）的特征，DenseNet 可以综合利用浅层复杂度低的特征，因而更容易得到一个光滑的具有更好泛化性能的决策函数；**

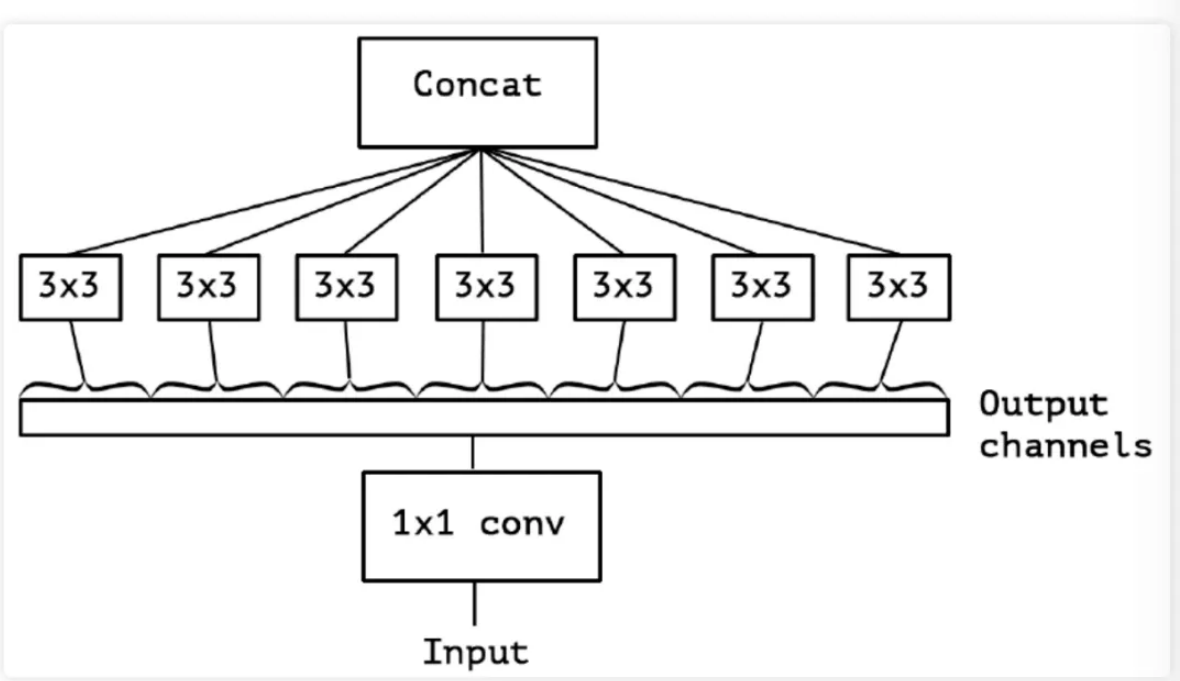
Inception

Inception使用split-transform-merge策略把multi-scale filter生成的不同感受野的特征融合到一起，有利于识别不同尺度的对象；

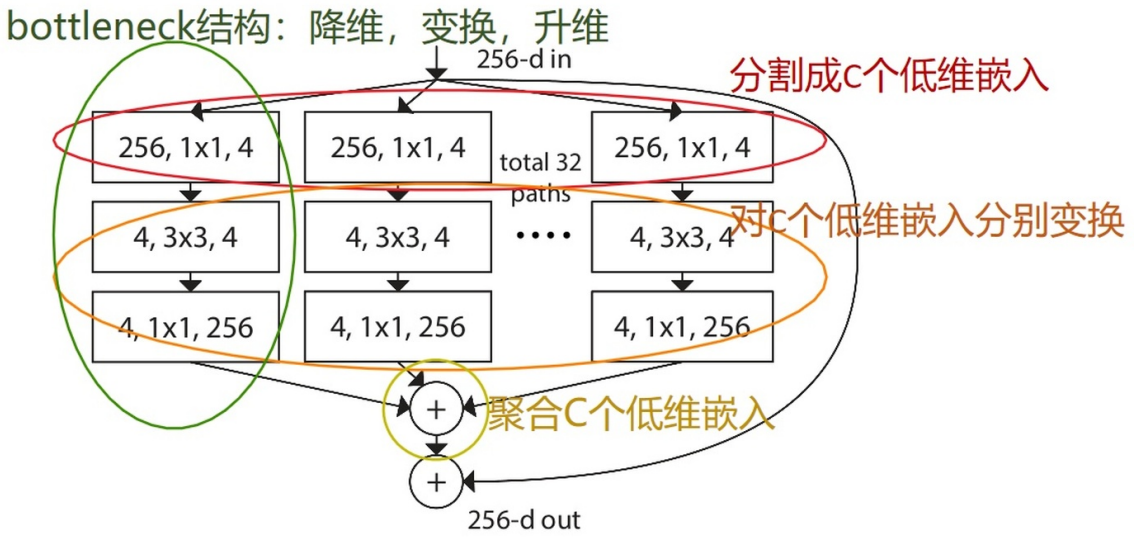
- v1：1x1 3x3 5x5 不同感受野；
- v2：使用BN；
- v3：两个3x3代替一个5x5，3x3=1x3 + 3x1；
- v4：残差连接；
- **Xception**

  “简述Xception中的深度可分离卷积”： -> 在**使用了残差结构**和**去掉 Xception 模块中两个操作之间的RELU激活函数**下训练**收敛的速度更快，精度更高**

  - Xception 模块为：$\mathrm{Conv}(1\times1)+BN+\mathrm{Re}\,LU+\mathrm{Depth}\,\mathrm{conv}(3\times3)+BN+ReLU$
  - 普通的深度可分离卷积结构为：$\mathrm{Depthconv}(3\times3)+B\mathbf{N}+\mathrm{Conv}(1\times1)+BN+\mathrm{ReL}\,U$



- ResNext



MobileNet

- v1：
  - **深度可分离卷积 + ReLU6**：3x3深度卷积， 1x1升维；
- v2：
  - 深度卷积训出来的卷积核有不少是空的 -> 在低维度ReLU使得信息丢失，所以**Inverted residuals**：1x1先升维，3x3深度卷积， 1x1降维，最后的ReLU6替换为Add；
- v3：
  - 使用NAS
  - v2的Inverted residuals
  - SE模块
  - h-swish激活函数

ShuffleNet

- v1：
  - **pointwise group convolution**（降低1x1卷积的计算量）
  - **channel shuffle**（解决不同组之间的特征图不通信）
- v2：
  - **平衡输入输出通道**(in = out)
  - **谨慎使用组卷积**
  - **避免网络碎片化**(一些操作可以合并，conv+BN)
  - **减少元素级运算**(Add，ReLU)

梯度消失 & 梯度爆炸

目前优化神经网络的方法都是基于BP，即根据损失函数计算的误差通过梯度反向传播的方式，指导深度网络权值的更新优化。其中将误差从末层往前传递的过程需要**链式法则（Chain Rule）**的帮助，因此反向传播算法可以说是梯度下降在链式法则中的应用。

而链式法则是一个**连乘的形式**，所以当层数越深的时候，梯度将以指数形式传播。梯度消失问题和梯度爆炸问题一般随着网络层数的增加会变得越来越明显。在根据损失函数计算的误差通过梯度**反向传播**的方式对深度网络权值进行更新时，得到的**梯度值接近0**或**特别大**，也就是**梯度消失**或**爆炸**。梯度消失或梯度爆炸在本质原理上其实是一样的。

产生原因：

```
0 梯度消失：1.深层网络，雅各比矩阵最大特征值小于1  2.激活函数（Sigmoid）

1 梯度爆炸：1.深层网络，雅各比矩阵最大特征值大于1  2.Weights初始化值太大
```

解决方法：

```
- 深层网络：
    0 残差结构（乘法改成加法，固定梯度1） / RNN门机制（LSTM，GRU）

- 损失函数、权重：
    0 合理的激活函数：ReLU
    合理的参数初始化（He,Xavier）-> 让每层均值和方差保持一致
    权重衰减

- 梯度：
    1 梯度归一化 / 梯度剪切 -> 让梯度值在合理范围内[1e-6, 1e3]
    Batch Normalization

预训练 + 微调
```

(W, b) 初始化

一种比较简单有效的方法是： (W, b) 初始化从区间 $\left(-\frac{1}{\sqrt{d}}, \frac{1}{\sqrt{d}}\right)$ 均匀随机取值。**其中 d 为（ W, b ） 所在层的神经元个数；**

可以证明，如果X服从正态分布, 均值0, 方差1，且各个维度无关, 而 (W, b) 是 $\left(-\frac{1}{\sqrt{d}}, \frac{1}{\sqrt{d}}\right)$ 的均匀分布, 则 $W^T X+b$ 是均值为0, 方差为1/3的正态分布；

Kaiming初始化

- **前向传播**的时候, 每一层的**卷积计算的方差为1**;
- **反向传播**的时候, 每一层的继续往**前传的梯度方差为1**(因为每层会有两个梯度的计算, 一个用来更新当前层的权重, 一个继续传播,用于前面层的梯度的计算);
- `torch.nn.init.kaiming_normal_(layer.weight,mode='fan_out', nonlinearity='relu')`

全连接层的作用

- 全连接层在整个卷积神经网络中**起到“分类器”的作用**，如果说卷积层、池化层和激活函数层等操作是将原始数据映射到隐层特征空间的话，**全连接层则起到将学到的“分布式特征表示”映射到样本标记空间的作用**，在实际使用中，全连接层可由卷积操作实现：对前层是全连接的全连接层可以转化为卷积核为1x1的卷积；而前层是卷积层的全连接层可以转化为卷积核为hxw的全局卷积，h和w分别为前层卷积结果的高和宽；
- 全连接层参数冗余（仅全连接层参数就可占整个网络参数80%左右），用全局平均池化取代FC来融合学到的深度特征，**用GAP替代FC的网络通常有较好的预测性能**；
- **FC可在模型表示能力迁移过程中充当“防火墙”的作用。**不含FC的网络微调后的结果要差于含FC的网络。因此FC可视作模型表示能力的“防火墙”，特别是在源域与目标域差异较大的情况下，FC可保持较大的模型capacity从而保证模型表示能力的迁移。（冗余的参数并不一无是处。）

网络不收敛&训练失败

在训练过程中，loss并不是一直在下降，准确率一直在提升的，会有一些震荡存在，只要总体趋势是在收敛就行

```
train loss 不断下降，test loss不断下降，说明网络仍在学习;

train loss 不断下降，test loss趋于不变，说明网络过拟合;

train loss 趋于不变，test loss不断下降，说明数据集100%有问题;

train loss 趋于不变，test loss趋于不变，说明学习遇到瓶颈，需要减小学习率或批量数目;

train loss 不断上升，test loss不断上升，说明网络结构设计不当，训练超参数设置不当，数据集经过清洗等问题;
```

数据与标签

- 没有对数据进行**预处理**，**数据是否干净**
- **对数据进行归一化**：由于不同评价指标往往具有不同的量纲和量纲单位，这样的情况会影响到数据分析的结果，为了消除指标之间的量纲影响，需要进行数据标准化处理，以解决数据指标之间的可比性
- 是否存在**过拟合**
- **标签是否正确**

模型

- **网络设定不合理**：任务难度和模型不匹配
- **Learning rate**不合适：如果太大，会造成不收敛，如果太小，会造成收敛速度非常慢
- 错误**初始化网络参数**：如果没有正确初始化网络权重，那么网络将不能训练
- **没有正则化**：正则化典型的就是BN、加噪声等
- **网络存在坏梯度：**如果你训练了几个epoch误差没有改变,那可能是你使用了Relu，可以尝试将激活函数换成leaky Relu。因为Relu激活函数对正值的梯度为1，负值的梯度为0。因此会出现某些网络权值的成本函数的斜率为0，在这种情况下我们说网络是“dead”,因为网络已经不能更新

NAN & INF

INF：数值太大、权重初始值太大 、Learning rate太大

NAN：除数为0产生

INF解决方案：

- 激活函数
- 权重初始均值为0，方差小
- learning不断减小

L1 L2正则化

正则化之所以能够**降低过拟合**的原因在于，**正则化是结构风险最小化的一种策略实现**

- **权重衰减**通过控制**L2正则**项使得模型参数不会过大，从而控制**模型复杂度**；
- **正则项权重**是控制模型复杂的**超参数**；
- **正则化其他方法：1. EMA of Weights 2. Label Smoothing 3. RandAugment 4. Dropout on FC 5.BN**
- 给loss function加上正则化项，能使得新得到的优化目标函数**h = f(w, b)+normal(w)**，需要在f和normal中做一个权衡，如果还像原来只优化f的情况下，那可能得到一组解比较复杂，使得正则项normal比较大，那么h就不是最优的，normal引入使得最优解向原点移动，因此可以看出**加正则项能实现参数的稀疏，让解更加简单，通过降低模型复杂度防止过拟合，提升模型的泛化能力**；

		特点1	特点2		作用	
L1正则化	在loss function后边所加**正则项为L1范数**	容易得到**稀疏解**	容易产生**稀疏的权重**	趋向于产生**少量的特征**，而**其他的特征都是0**	特征选择	**对异常值更鲁棒**；在0点不可导，计算不方便；**没有唯一解**；输出稀疏，会将不重要的特征直接置0；
L2正则化	loss function后边所加**正则项为L2范数的平方**	容易得到**平滑解**	容易产生**分散的权重**	会选择**更多的特征**，这些**特征都会接近于0**	防止过拟合	计算方便；**对异常值敏感**；**有唯一解**；**抗干扰能力强**

L0 范数：向量中**非0元素的个数**； 果用L0规则化一个参数矩阵W，就是**希望W中大部分元素是零，实现稀疏**，*稀疏化可以去掉这些无用特征，将特征对应的权重置为零*，但是L0范数很难优化求解（NP难问题）

L1 范数：向量中**各个元素绝对值的和；**

L2 范数：向量中**各元素平方和**再**求平方根；**

BN & Dropout

B x H x W，不涉及Channel，数据归一化方法；

BN的精髓在于归一之后，使用γ, β作为还原参数，在一定程度上保留原数据的分布；

机器学习领域有个很重要的假设：**IID独立同分布假设**，就是假设训练数据和测试数据是满足相同分布的，这是通过训练数据获得的模型能够在测试集获得好的效果的一个基本保障

```
批规范化（Batch Normalization，BN）：在 minibatch维度 上在每次训练iteration时对隐藏层进行归一化
标准化（Standardization）：对输入 数据 进行归一化
正则化（Regularization）：通常是指对 参数 在量级和尺度上做约束，缓和过拟合情况，L1 L2正则化
```

提出原因 & 作用 & 缺点

- **解决Internal Covariate Shift（内部协变量偏移）**：随着网络深度加深或者在训练过程中，其分布逐渐发生偏移或者变动，之所以训练收敛慢，一般是整体分布逐渐往非线性函数的取值区间的上下限两端靠近，所以这**导致反向传播时低层神经网络的梯度消失**，这是训练深层神经网络收敛越来越慢的**本质原因，而BN就是通过一定的规范化手段，把每层神经网络任意神经元这个输入值的分布强行拉回到均值为0方差为1的标准正态分布**
- **缓解过拟合**：

作用

- 将数据规整到统一区间，减少数据的发散程度，
- 加快模型训练时的收敛速度，使得模型训练过程更加稳定；
- 避免梯度爆炸或者梯度消失；
- 起到一定的正则化作用，防止过拟合；
- 用了之后，对学习率、参数更新策略等不敏感；

$Input:B=\{x_{1...m}\};\gamma,\beta($ parameters to be learned $)$

$\text{Output}:\{y_i=BN_{\gamma,\beta}(x_i)\}$

$$\begin{aligned}\mu_B &\leftarrow \frac{1}{m}\sum_{i=1}^{m}x_i\\ \sigma_B^2 &\leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i-\mu_B)^2\\ \tilde{x}_i &\leftarrow \frac{x_i-\mu_B}{\sqrt{\sigma_B^2+\epsilon}}\\ y_i &\leftarrow \gamma\tilde{x}_i+\beta\end{aligned}$$

均值的计算，就是在一个批次内，将每个通道中的数字单独加起来，再除以 $\boldsymbol{N\times H\times W}$；举个例子：该批次内有10张图片，每张图片有三个通道RBG，每张图片的高、宽是H、W，那么均值就是计算**10张图片R通道的像素数值总和**除以$10\times\boldsymbol{H}\times\boldsymbol{W}$，再计算B通道全部像素值总和除以$10\times\boldsymbol{H}\times\boldsymbol{W}$，最后计算G通道的像素值总和除以$10\times\boldsymbol{H}\times\boldsymbol{W}$。方差的计算类似；

可训练参数 $\boldsymbol{\gamma}$、 $\boldsymbol{\beta}$ 的维度等于**张量的通道数**，在上述例子中，RBG三个通道分别需要一个 $\boldsymbol{\gamma}$ 和一个 $\boldsymbol{\beta}$，所以 $\vec{\boldsymbol{\gamma}}$、 $\vec{\boldsymbol{\beta}}$ 的维度等于3；

缺陷

1. **依赖Batch size** -> GN
2. 2.**对于RNN这样的动态网络效果不明显**，且当推理序列长度超过训练的所有序列长度时，容易出问题 -> LN
3. 当**mini-batch中的样本非独立同分布时**，性能比较差 -> BRN

PyTorch中BN

在PyTorch中**将gamma和beta改叫weight、bias**，使得打印网络参数时候只会打印出weight和bias（PyTorch中只有可学习的参数才称为Parameter）,但是`Net.state_dict()`是有running_mean和running_var的，**因为running_mean和running_var不是可以学习的变量，只是训练过程对很多batch的数据统计;**

BN层的**输出Y与输入X之间的关系：Y = (X - running_mean) / sqrt(running_var + eps) * gamma + beta**，其中**gamma、beta为可学习参数（在PyTorch中分别改叫weight和bias），训练时通过反向传播更新**；而**running_mean、running_var则是在前向时先由X计算出mean和var，再由mean和var以动量momentum来更新running_mean和running_var**，所以**在训练阶段，running_mean和running_var在每次前向时更新一次**；在**测试阶段，则通过**`net.eval()`**固定该BN层的running_mean和running_var，此时这两个值即为训练阶段最后一次前向时确定的值，并在整个测试阶段保持不变；**

```
训练时：
running_mean = (1 - momentum) * running_mean + momentum * mean_cur
running_var = (1 - momentum) * running_var + momentum * var_cur
```

```
测试时：
running_mean = running_mean
running_var = running_var
```

先更新running_mean和running_var，再计算BN；

Conv和BN的融合

BN层最酷的地方是它可以用一个1x1卷积等效替换，更进一步地，我们可以将BN层合并到前面的卷积层中；

$$
\begin{aligned}
y_{\text{conv}} &= w \cdot x + b \\
y_{bn} &= \gamma \cdot \left(\frac{y_{\text{conv}} - E[x]}{\sqrt{\text{Var}[x] + \epsilon}} \right) + \beta \\
&= \gamma \cdot \left(\frac{wx + b - E[x]}{\sqrt{\text{Var}[x] + \epsilon}} \right) + \beta \\
\hat{w} &= \frac{\gamma}{\sqrt{\text{Var}[x] + \epsilon}} \cdot w \\
\hat{b} &= \frac{\gamma}{\sqrt{\text{Var}[x] + \epsilon}} \cdot (b - E[x]) + \beta \\
y_{bn} &= \hat{w} \cdot x + \hat{b}
\end{aligned}
$$

Dropout在训练和测试时区别

`torch.nn.Dropout2d(p=0.5, inplace=False)`：input shape: (N, C, H, W)， output shape: (N, C, H, W)

Dropout在层与层之间加噪声，是一种正则；在全连接使用，CNN用BN

Dropout 是**在训练过程中以一定的概率的使神经元失活控制模型复杂度，提高模型的泛化能力，减少过拟合**；

Dropout 在训练时采用，是为了减少神经元对部分上层神经元的依赖，类似将多个不同网络结构的模型集成起来，减少过拟合的风险；而在测试时，应该用整个训练好的模型，因此**测试时不需要dropout**；

=> **在测试时如果丢弃一些神经元，这会带来结果不稳定的问题，**是给定一个测试数据，有时候输出a有时候输出b，结果不稳定，用户可能认为模型预测不准。那么**一种"补偿"的方案就是每个神经元的权重都乘以一个p，这样在"总体上"使得测试数据和训练数据是大致一样的。**比如一个神经元的输出是x，那么在训练的时候它有p的概率参与训练，(1-p)的概率丢弃，那么它输出的期望是px+(1-p)0=px。因此测试的时候把这个神经元的权重乘以p可以得到同样的期望；

BN和Dropdout同时使用

方差偏移现象

Dropout 与 BN 之间冲突的关键是**网络状态切换过程中存在神经方差的不一致行为。**试想若有神经响应 X，当网络从训练转为测试时，Dropout 可以通过其随机失活保留率（即 p）来缩放响应，并在学习中改变神经元的方差，而 BN 仍然维持 X 的统计滑动方差。这种方差不匹配可能导致数值不稳定。而随着网络越来越深，最终预测的数值偏差可能会累计，从而降低系统的性能。事实上，如果没有 Dropout，那么实际前馈中的神经元方差将与 BN 所累计的滑动方差非常接近，这也保证了其较高的测试准确率。

解决方案

1.在所有 BN 层后使用 Dropout；

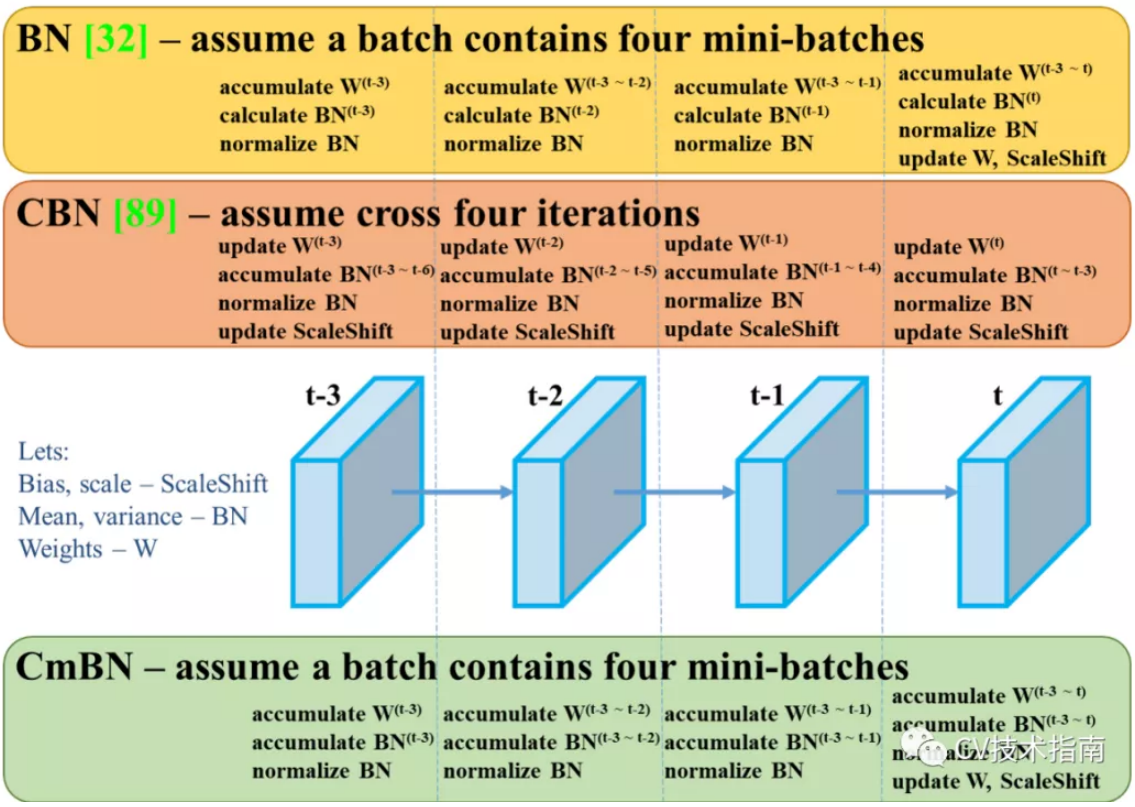
2.修改 Dropout 的公式让它对方差并不那么敏感，就是高斯Dropout、均匀分布Dropout；

BN / LN / IN / GN / CmBN

对特征图做归一化，对于 `[B,C,W,H]` 这样的训练数据而言

- **BN** 是在 `[B,W,H]` 维度求均值方差进行规范化 -> CNN
- **LN** 是对 `[C,W,H]` 维度求均值方差进行规范化 -> RNN，Transformer
- **IN** 是对 `[W,H]` 维度求均值方差进行规范化 -> 图像风格化：GAN，style transfer
- **GN** 先对通道进行分组，每个组内的所有[C_i,W,H]维度求均值方差进行规范化，与BatchSize无关 -> 在目标检测，语义分割等要求尽可能大的分辨率的任务,**但GN有两个缺陷，其中一个是在batchsize大时略低于BN，另一个是由于它是在通道上分组，因此它要求通道数是分组数g的倍数**
- **BRN** 提出在训练过程中就不断学习修正整个数据集的均值和方差，使其尽可能逼近整个数据集的均值和方差，并最终用于推理阶段
- **Cross-GPU BN** 例如batchsize=32，用四张卡训练，实际上只在32/4=8个样本上做归一化；
- **CBN** 将前k-1个iteration的样本参与当前均值和方差的计算：但由于前k-1次iteration的数据更新，因此无法直接拿来使用，论文提出了一个处理方式是通过泰勒多项式来近似计算出前k-1次iteration的数据；

- **CmBN**



数据集划分

验证集要和训练集来自于同一个分布（shuffle数据集然后开始划分），测试集尽可能贴近真实数据

- 通常80%为训练集，20%为测试集
- 当**数据量较小**时（万级别）的时候将训练集、验证集以及测试集划分为**6：2：2**；若是**数据量很大**，可以将训练集、验证集、测试集比例调整为**98：1：1**
- 当数据量很小时，可以采用**K折交叉验证**
- 刚开始的时候，用训练集训练，验证集验证，确定超参数和一些细节；在验证集调到最优后，再把验证集丢进来训练，在测试集上测试
- 划分数据集时可采用随机划分法（当样本比较均衡时），分层采样法（当样本分布极度不均衡时）

卷积 & 互相关 & 参数量计算

卷积是透过两个函数f和g生成第三个函数的一种数学算子，表征函数f与**经过翻转和平移的**g的乘积函数**所围成曲边梯形的面积**；

互相关是两个函数之间的**滑动点积**或滑动内积，互相关中的过滤不经过反转，而是直接滑过函数f，f与g之间的**交叉区域**即是互相关；

严格意义上来说，深度学习中的"卷积"是互相关(Cross-correlation)运算，本质上执行逐元素乘法和加法。但在之所以习惯性上将其称为卷积，是因为过滤器的权值是在训练过程中学习得到的；

卷积特点

```
1.局部连接 2.权值共享 3.层次结构
```

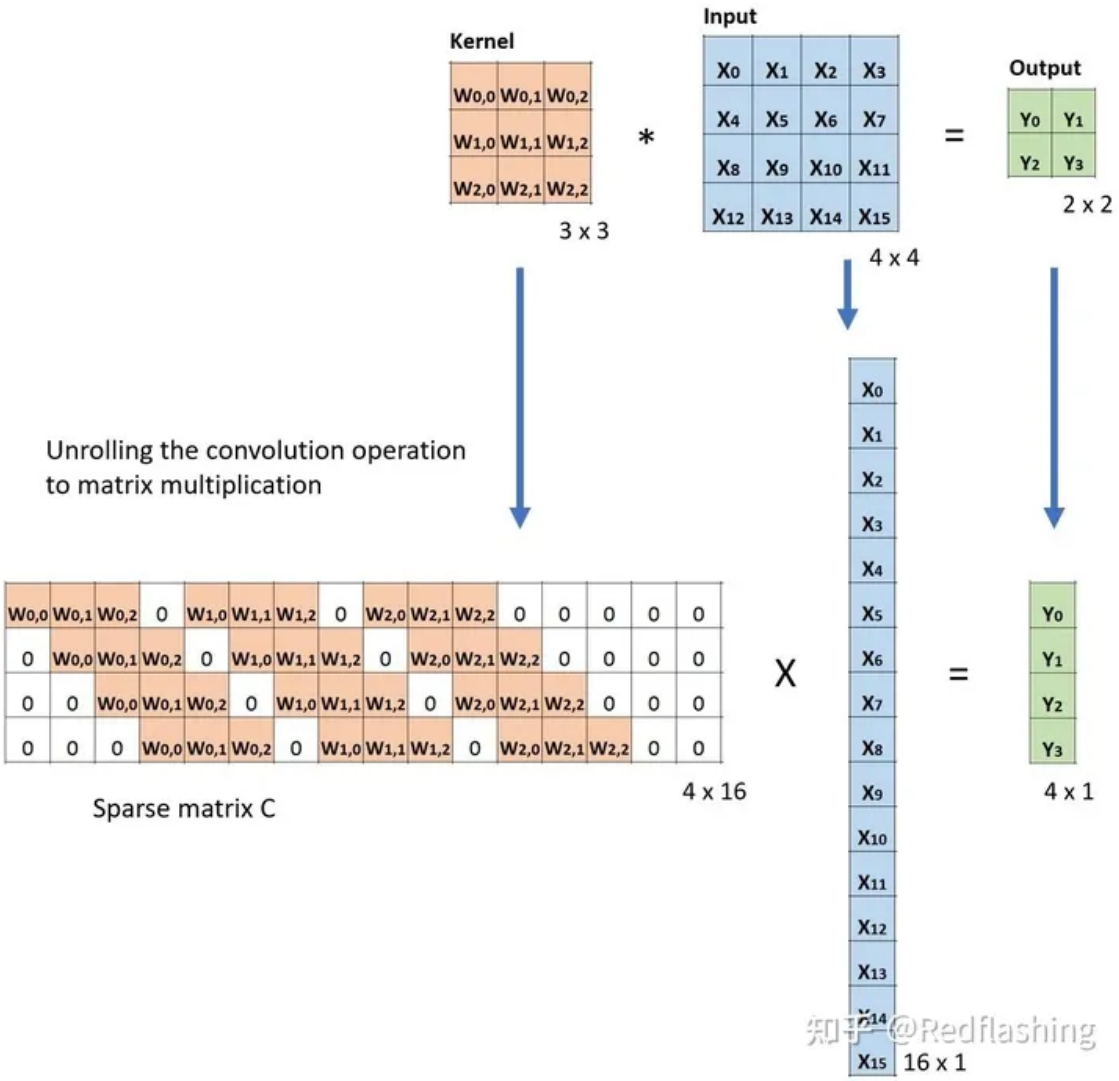
- **每个通道之间不共享参数，希望每个通道学到不同的模式**
- **不具有旋转不变性**：要不是中心对称的像素团，旋转之后的卷积值肯定是不一样的；
- **不具有平移不变性**：同一个像素团，只要卷积核对齐了，卷积值都是一样的，但是**加了padding的卷积网络平移不变性也是不存在的**，不带padding的网络每一层都必须进行严密的设计，如不带padding的UNet，通常为了网络设计简单，对训练样本做**平移增广**是很有必要的；
- 如果边长减去1后不能被stride整除，**卷积的降采样过程会丢弃边缘的像素**，**特征图像素与输入图像位置映射会产生偏移**，目前所有的深度学习框架都没有考虑这里的映射错位关系，训练时输入样本图像的大小和检测时切块的大小只能用最终特征图的尺寸反推回去，保证在卷积过程中不丢弃边缘；

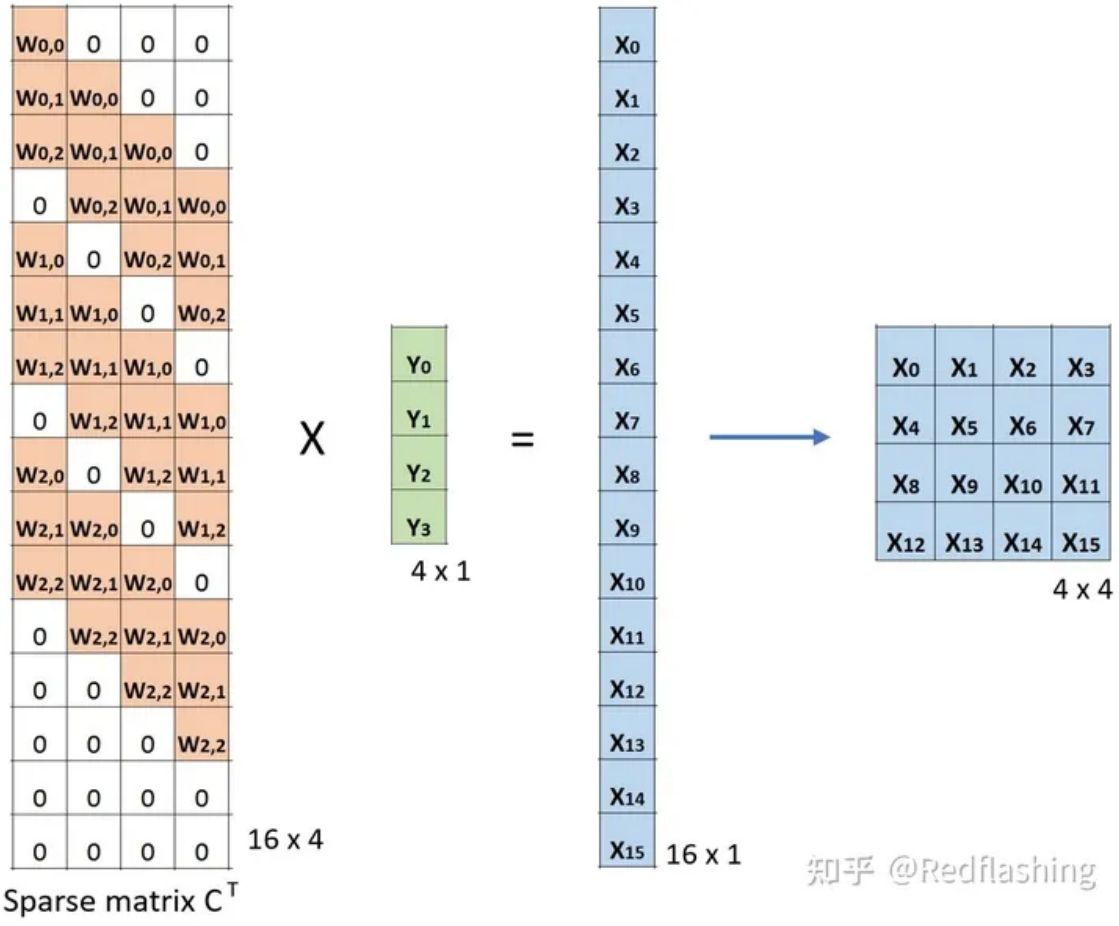
卷机类型

- **3x3卷积**
  - 底层专门做过优化，适合提取图像特征，已经成为主流组件；
- **1x1卷积**
  - 升维降维；
  - 减少参数；
  - 通道融合；
  - 增加非线性（利用后接的非线性激活函数如ReLU）；
- **空洞卷积**
  - kernal之间增加空洞，增加感受野，图森组针对空洞卷组专门做过研究：[1, 3, 5, 1, 3, 5]这样的空洞率；
  - 虽然增大了感受野，但是使得特征更加稀疏；
  - **解决了网格效应**；
  - k - > k + (r - 1) (k - 1);
- **转置卷积**

  会出现棋盘效应：由于转置卷积的"不均匀重叠" -> **1.采取可以被步长整除的卷积核长度 2.插值**；

  对于同一个卷积核（因非其稀疏矩阵不是正交矩阵），结果转置操作之后并不能恢复到原始的数值，而仅仅保留原始的形状，**上采样常用双线性插值；**


Kernel
Input
Output
3 x 3
4 x 4
2 x 2
Unrolling the convolution operation
to matrix multiplication
Sparse matrix C
4 x 16
4 x 1
16 x 1
知乎 @Redflashing


16 x 4
Sparse matrix C^T
4 x 1
16 x 1
4 x 4
知乎 @Redflashing

- **深度可分离卷积**

  计算成本仅仅是2D卷积的12%左右：对于规模较小的模型，如果将普通卷积替换为深度可分卷积，其模型大小可能会显著降低，模型的能力可能会变得不太理想，因此**得到的模型可能是次优的**；但如果使用得当，深度可分卷积能在不牺牲模型性能的前提下显著提高效率；

  MobileNet v1中提出：深度卷积 + 1x1卷积

参数量：卷积核的尺寸是Dk×Dk×M，一共有N个

> **标准卷积**的参数量是：Dk x Dk x M x N
> **深度卷积**的卷积核尺寸Dk×Dk×M；逐点卷积的卷积核尺寸为1×1×M，一共有N个，所以深度可分离卷积的参数量是：Dk x Dk x M + M x N

计算量：普通卷积核的尺寸是Dk×Dk×M，一共有N个，每一个都要进行Dw×Dh次运算

> **标准卷积**的计算量是：Dk x Dk x M x N x Dw x Dh
> **深度卷积**的卷积核尺寸Dk×Dk×M，一共要做Dw×Dh次乘加运算；逐点卷积的卷积核尺寸为1×1×M，有N个，一共要做Dw×Dh次乘加运算，所以深度可分离卷积的计算量是：Dk x Dk x M x Dw x Dh + M x N x Dw x Dh

参数量和运算量均下降为原来的：$\frac{1}{N}+\frac{1}{D_k^2}$

感受野

感受野大小计算公式：

$$r_l = r_{l-1} + (k_l - 1) * \prod_{i=0}^{l-1} s_i$$

其中 r_{l-1} 为第 $l-1$ 层的感受野大小, k_l 为第l层的卷积核大小（也可以是Pooling），s_i 为第 i 层的卷 积步长。一般来说 $r_0 = 1, s_0 = 1$；

CNN中感受野定义及性质

> 在深度神经网络中，每个神经元节点都对应着输入图像的某个确定区域，仅该区域的图像内容能对相应神经元的激活产生影响，那么这个区域称为该神经元的感受野；
>
> - 越靠近感受野中心的区域越重要
> - 各向同性
> - 由中心向周围的重要性衰减速度可以通过网络结构控制

感受野是直接或者间接参与计算特征图像素值的输入图像像素的范围，直接感受野就是卷积核大小，随着卷积层数的加深之前层次的感受野会叠加进去。**感受野小了缺乏环境信息，感受野大了引入太多环境干扰**，所以**一个网络能够检测的目标框范围与特征图像素或者特征向量的感受野有关**，通常能够检测的目标框边长范围是感受野边长的0.1-0.5倍；

拿到了一个网络**要做感受野分析，然后确定它能够检测多少像素的目标。**实际目标检测任务需要综合网络结构设计和图像分辨率选择。如果目标框的像素范围超过了网络的感受野，就需要将原始图像缩小后再检测；

Padding

- **保持边界信息**：如果没有加padding的话，输入图片最边缘的像素点信息只会被卷积核操作一次，但是图像中间的像素点会被扫描到很多遍，那么就会在一定程度上降低边界信息的参考程度，但是在加入padding之后，在实际处理过程中就会从新的边界进行操作，就从一定程度上解决了这个问题；
- 可以利用padding对输入尺寸有差异图片进行补齐，**使得输入图片尺寸一致**；
- 在卷积神经网络的卷积层加入Padding，可以使得**卷积层的输入维度和输出维度一致**；

池化Pooling

Pooling层是模仿人的视觉系统对数据进行降维

类型：

```
最大/平均池化 Max/Average Pooling
全局池化Global Pooling
随机池化Stochastic pooling

ROI Pooling
```

作用：

```
1. 抑制噪声，降低信息冗余
2. 提升模型的鲁棒性，防止过拟合
3. 降低模型计算量
```

缺点：

```
造成梯度稀疏、丢失信息
```

最大池化作用：保留主要特征，突出前景

平均池化作用：保留背景信息，突出背景

Mean Pooling

在forward的时候，就是在前面卷积完的输出上依次不重合的取2x2的窗平均，得到一个值就是当前mean pooling之后的值；**backward的时候，把一个值分成四等分放到前面2x2的格子里面就好了；**（假设pooling的窗大小是2x2）

```
forward: [1 3; 2 2] -> [2]
backward: [2] -> [0.5 0.5; 0.5 0.5]
```

平均池化取每个块的平均值，提取特征图中所有特征的信息进入下一层。因此**当特征中所有信息都比较有用时，使用平均池化。如网络最后几层，最常见的是进入分类部分的全连接层前，常常都使用平均池化。这是因为最后几层都包含了比较丰富的语义信息，使用最大池化会丢失很多重要信息；**

Max Pooling

在forward的时候你只需要把2x2窗子里面那个最大的拿走就好了，**backward的时候你要把当前的值放到之前那个最大的位置，其他的三个位置都弄成0；**

```
forward: [1 3; 2 2] -> [3]
backward: [3] -> [0 3; 0 0]
```

最大池化的操作，取每个块中的最大值，而其他元素将不会进入下一层。CNN卷积核可以理解为在提取特征，对于最大池化取最大值，可以理解为提取特征图中响应最强烈的部分进入下一层，而其他特征进入待定状态；

一般而言，前景的亮度会高于背景，因此，最大池化具有提取主要特征、突出前景的作用。但在个别场合，前景暗于背景时，最大池化就不具备突出前景的作用了；

当特征中只有部分信息比较有用时，使用最大池化。如网络前面的层，图像存在噪声和很多无用的背景信息，常使用最大池化；

- [x] https://mp.weixin.qq.com/s/2mEhUuHOeT4Y4NZZ3NlktQ

过采样 & 欠采样

原始数据大小为$\mathfrak{R}^{1831\times21}$，1831条数据，每条数据有21个特征：其中正例176个（9.6122%），反例1655个（90.3878%），类别不平衡;

欠采样：从反例中随机选择176个数据，与正例合并（ $\mathfrak{R}^{352\times21}$ ）

过采样：从正例中反复抽取并生成1655个数据（势必会重复），并与反例合并（ $\mathfrak{R}^{3310\times21}$ ）

- 使用采样方法一般可以**提升模型的泛化能力**，但有一定的**过拟合的风险**，应搭配使用正则化模型；
- **过采样的结果较为稳定**，过采样大部分时候比欠采样的效果好；

过采样带来**更大的运算开销**，当数据中噪音过大时，结果反而可能会更差因为**噪音也被重复使用**；

尝试**半监督学习**的方法；注意积累样本；数据增强；可以训练多个模型，最后进行集成；

- [x] https://www.zhihu.com/question/269698662/answer/352279936

过拟合 & 欠拟合

过拟合指的是在训练集error越来越低，但是在验证集和测试集error不变或越来越高，模型拟合了训练样本中的噪声，导致泛化能力差；

```
数据增强
缩减模型表达能力
正则化（Weight Decay， L1，L2）
Early Stopping
Dropout / BN
```

欠拟合指的是训练集提取特征较少，导致模型不能很好拟合训练集；

```
增加模型复杂度  eg.ResNet-50 -> resNet-101;
减少正则化
错误分析：（训练集和测试集的分布偏差）测试时候出现问题进行分析，训练集缺少哪些情况导致错误，后续将在训练集中加入此类数据纠正偏差;
加入更多特征
```

SGD / Adam

一阶方法：随机梯度下降（SGD）、动量（Momentum）、牛顿动量法（Nesterov动量）、AdaGrad（自适应梯度）、RMSProp（均方差传播）、Adam、Nadam

二阶方法：牛顿法、拟牛顿法、共轭梯度法（CG）、BFGS、L-BFGS

自适应优化算法：Adagrad（累积梯度平方）、RMSProp（累积梯度平方的滑动平均）、Adam（带动量的RMSProp，即同时使用梯度的一、二阶矩）

梯度下降陷入局部最优有什么解决办法?

OncycleLR+SGD / Adam，每隔一段时间重启学习率，可以在单位时间收敛到多个局部最小值

- 二阶导是向量，学术上比较好，但难算，一阶实用；
- 只关注收敛在哪个地方，SGD一步一步来，二阶收敛效果不一定比一阶收敛效果好；

马鞍状的最优化地形，其中对于不同维度它的曲率不同（一个维度下降另一个维度上升）

- **基于动量**的方法使得最优化过程看起来像是一个球滚下山的样子
- **SGD**很难突破对称性，一直卡在顶部
- **RMSProp之类**的方法能够看到马鞍方向有很低的梯度（因为在RMSProp更新方法中的分母项，算法提高了在该方向的有效学习率，使得RMSProp能够继续前进）

梯度下降与拟牛顿法的异同

- **参数更新模式相同**
- **梯度下降法利用误差的梯度**来更新参数，**拟牛顿法利用海塞矩阵**的近似来更新参数
- **梯度下降**是泰**勒级数的一阶展开**，而**拟牛顿法是泰勒级数的二阶展开**
- **SGD能保证收敛**，但是L-BFGS在非凸时不收敛

一个框架来梳理所有的优化算法

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 $\boldsymbol{t}$：

1. 计算目标函数关于当前参数的梯度： $g_t = \nabla f\left(w_t\right)$
2. 根据历史梯度计算一阶动量和二阶动量:
   $m_t = \phi\left(g_1, g_2, \cdots, g_t\right); V_t = \psi\left(g_1, g_2, \cdots, g_t\right)$
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： $w_{t+1} = w_t - \eta_t$

SGD（普通更新）

最简单的沿着负梯度方向改变参数；假设有一个**参数向量x**及其**梯度dx**，那么最简单的更新的形式是：

```
# 普通更新
x += - learning_rate * dx
```

SGD最大的缺点是下降速度慢，而且可能会在沟壑的两边持续震荡，停留在一个局部最优点

- (1) (W, b) 的每一个分量获得的梯度绝对值有大有小, 一些情况下， 将会迫使优化路径变成Z字形状；
- (2) SGD求梯度的策略过于随机, 由于上一次和下一次用的是 完全不同的BATCH数据, 将会出现优化的方向随机的情况；

SGDM（动量更新，解决梯度随机性）

该方法从**物理角度**上对于最优化问题得到的启发：

从本质上说，动量法，就像我们从山上推下一个球，球在滚下来的过程中累积动量，变得越来越快（直到达到终极速度，如果有空气阻力的存在，则mu<1）；同样的事情也发生在参数的更新过程中：**对于在梯度点处具有相同的方向的维度，其动量项增大，对于在梯度点处改变方向的维度，其动量项减小。**因此，我们可以得到更快的收敛速度，同时可以减少摇摆

也就是说，t 时刻的下降方向，不仅由当前点的梯度方向决定，而且由此前累积的下降方向决定。 mu的经验值为0.9，这就意味着下降方向主要是此前累积的下降方向，并略微偏向当前时刻的下降方向

在SGD中，梯度影响**位置**；

而在这个版本的更新中，物理观点建议**梯度只是影响速度**，然后**速度再影响位置**：

```
# 动量更新
    v = mu * v - (1 - mu) * dx # 与速度融合，mu其物理意义与摩擦系数更一致
    x += v # 与位置融合
```

mu通常取值为0.9，这就意味着下降方向主要是此前累积的下降方向，并略微偏向当前时刻的下降方向

NAG（Nesterov动量）

SGD 还有一个问题是困在局部最优的沟壑里面震荡。想象一下你走到一个盆地，四周都是略高的小山，你觉得没有下坡的方向，那就只能待在这里了。可是如果你爬上高地，就会发现外面的世界还很广阔。因此，我们不能停留在当前位置去观察未来的方向，而要向前一步、多看一步、看远一些。

当参数向量位于某个位置 *x* 时，观察上面的动量更新公式，动量部分会通过啊$mu * v$改变参数向量；

因此，如要计算梯度，那么可以将**未来的近似位置**$x + mu * v$看做是“**向前看**”，这个点在我们一会儿要停止的位置附近。因此，**计算** $x + mu * v$**的梯度**而不是“旧”位置 *x* 的梯度，使用Nesterov动量，我们就在这个“向前看”的地方计算梯度

```
x_ahead = x + mu * v
计算dx_ahead(在x_ahead处的梯度，而不是在x处的梯度)
v = mu * v - learning_rate * dx_ahead
x += v
```

上面的程序还得计算dx_ahead，通过对 x_ahead = x + mu * v 使用变量变换进行改写，然后用x_ahead而不是x来表示上面的更新，即：实际**存储**的参数向量总是**向前一步版本**。x_ahead 的公式（将其**重新命名为x**）就变成了：

```
v_prev = v # 存储备份
v = mu * v - learning_rate * dx # 速度更新保持不变
x += -mu * v_prev + (1 + mu) * v # 位置更新变了形式
```

mu=0.9

RMSprop

引用自Geoff Hinton的Coursera课程，具体说来，就是它使用了一个**梯度平方的滑动平均**：

```
cache = decay_rate * cache + (1 - decay_rate) * dx**2
x += - learning_rate * dx / (np.sqrt(cache) + eps)
```

decay_rate=0.9，learning_rate=0.001，RMSProp仍然是基于梯度的大小来对每个权重的学习率进行修改，但**其更新不会让学习率单调变小**；

- 不累积全部历史梯度，而**只关注过去一段时间窗口的下降梯度**，而指数移动平均值大约就是过去一段时间的平均值，因此我们用这一方法来计算二阶累积动量：

Adam

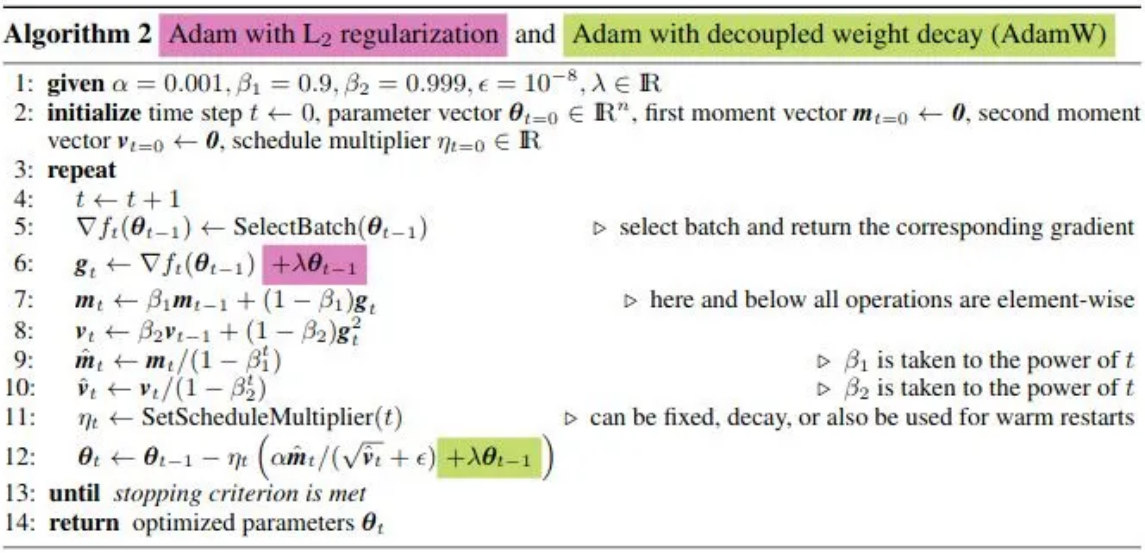
Adam本质上实际是RMSProp+动量：Adam对每一个参数都计算自适应的学习率。除了像RMSprop一样存储一个历史梯度平方的滑动平均vt，Adam同时还保存一个历史梯度的滑动平均mt，类似于动量：

```
# 根据历史梯度计算一阶动量和二阶动量
m_t = beta1*m + (1-beta1)*dx
v_t = beta2*v + (1-beta2)*(dx**2)

# 当mt和vt初始化为0向量时，发现它们都偏向于0，尤其是在初始化的步骤和当衰减率很小的时候（例如beta1和beta2趋向于1）,通过计算偏差校正的一阶矩和二阶矩估计来抵消偏差
m_hat = m_t / 1 - (beta1 ** t)
v_hat = v_t / 1 - (beta2 ** t)

x += - learning_rate * m_hat / (np.sqrt(v_hat) + eps)
```

eps=1e-8, beta1=0.9, beta2=0.999

AdamW

简单来说，**AdamW就是Adam优化器加上L2正则**，来限制参数值不可太大，以往的L2正则是直接加在损失函数上，所以在计算梯度`g_t`时要加上粉色的这一项，但AdamW稍有不同，如下图所示，将正则加在了绿色位置：



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

given $\alpha = 0.001, \beta_1 = 0.9, \beta_2 = 0.999, \epsilon = 10^{-8}, \lambda \in \mathbb{R}$
initialize time step $t \leftarrow 0$, parameter vector $\boldsymbol{\theta}_{t=0} \in \mathbb{R}^n$, 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
  $t \leftarrow t + 1$
  $\nabla f_t(\boldsymbol{\theta}_{t-1}) \leftarrow \text{SelectBatch}(\boldsymbol{\theta}_{t-1})$ ▷ select batch and return the corresponding gradient
  $\boldsymbol{g}_t \leftarrow \nabla f_t(\boldsymbol{\theta}_{t-1}) + \lambda \boldsymbol{\theta}_{t-1}$
  $\boldsymbol{m}_t \leftarrow \beta_1 \boldsymbol{m}_{t-1} + (1 - \beta_1)\boldsymbol{g}_t$ ▷ 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
  $\boldsymbol{\theta}_t \leftarrow \boldsymbol{\theta}_{t-1} - \eta_t \left(\alpha \hat{\boldsymbol{m}}_t / (\sqrt{\hat{\boldsymbol{v}}_t} + \epsilon) + \lambda \boldsymbol{\theta}_{t-1} \right)$
until *stopping criterion is met*
return optimized parameters $\boldsymbol{\theta}_t$

至于为何这么做？直接摘录BERT里面的原话：如果直接将L2正则加到loss上去，由于Adam优化器的后序操作，该正则项将会与`m_t`和`v_t`产生奇怪的作用。因而，AdamW选择将`L2`正则项加在了Adam的`m_t`和`v_t`等参数被计算完之后、在与学习率相乘之前，所**以这也表明了weight_decay和正则虽目的一致、公式一致，但用法还是不同，二者有着明显的差别；**

- [x] https://blog.csdn.net/google19890102/article/details/69942970
- [x] https://www.zhihu.com/question/323747423/answer/790457991

激活函数

- **Sigmoid**

  更倾向于更新靠近输出层的参数

  导数为$f(x) * (1 - (f(x)))$

  导数取值范围【0，0.25】

  左右两侧都是**近似饱和区**，导数太小，容易造成梯度消失

  涉及指数运算，容易溢出

  输出值不以零为中心，会导致模型收敛速度慢

  激活函数的**偏移现象**

- **ReLU**

  Dead ReLU：当 $x < 0$ 时，ReLU 输出恒为零。反向传播时，梯度恒为零，参数永远不会更新；

激活部分神经元，增加稀疏性；

计算简单，收敛速度快；

- **ReLu6**

  1. ReLU6比ReLU能更早学习到稀疏特征；
  2. 增强浮点数的小数位表达能力（整数位最大是6，所以只占3个bit，其他bit全部用来表达小数位）

- **Leaky ReLU**

  $LeakyReLU(x) = max(0.01x, x)$

  解决ReLU Dead；

- **Swish**

  $f(x) = x * sogmoid(\beta x)$

  可以看做是**介于线性函数与ReLU函数之间的平滑函数。**β是个常数或可训练的参数，Swish 具备**无上界有下界、平滑、非单调**的特性;

- **Hard-Swish**（Searching for MobileNetV3） YOLO v5使用后会有10%的推理速度损失；

$$\mathrm{h}-\operatorname{swish}(x)=x \frac{\operatorname{ReLU} 6(x+3)}{6}$$

- **SiLU**

$$f(x) = x \cdot \sigma(x)$$

- **GELU**

  $f(x) = x * sogmoid(1.702x)$

  与 Swish 激活函数$x * sogmoid(\beta x)$的函数形式和性质非常相像，一个是固定系数 1.702，另一个是可变系数 β

- **Mish**

  Mish是一个光滑非单调的激活函数，在Backbone使用后内存会增大:
  $f(x) = x \cdot \tanh(\ln(1 + e^x))$

- Loss function，即**损失函数**：用于定义单个训练样本与真实值之间的误差；
- Cost function，即**代价函数**：用于定义单个批次/整个训练集样本与真实值之间的误差；、
- Objective function，即**目标函数**：泛指任意可以被优化的函数；
- 通常，我们都会**最小化目标函数**，最常用的算法便是"**梯度下降法**"。损失函数大致可分为两种：回归损失（针对**连续型**变量）和分类损失（针对**离散型**变量）

分类Loss

CE:

$$\mathrm{CE}(p, y) = \begin{cases} -\log(p) & \text{if } y = 1 \\ -\log(1-p) & \text{otherwise} \end{cases}$$

其中$\boldsymbol{y} \in \{-1, 1\}$为真实标签，1表示为正例，-1表示为负例；而$\boldsymbol{p} \in [0, 1]$为模型预测为正例的概率值；

BCE Loss:

$$-\frac{1}{n}\sum\left(y_n \times \ln x_n + (1-y_n) \times \ln(1-x_n)\right)$$

BCEWithLogitsLoss = Sigmoid(将X_n转化到0～1) + BCELoss

Focal Loss

与抽样方法不同，Focal Loss从另外的视角来解决样本不平衡问题，那就是**根据置信度动态调整交叉熵loss**，当预测正确的置信度增加时，loss的权重系数会逐渐衰减至0，这样模型训练的loss更关注难例，而大量容易的例子其loss贡献很低

解决了one-stage算法中**正负样本的比例失衡**：在CE基础上增加了一个调节因子$(1-p_t)^\gamma$

$$FL\left(p_t\right) = -(1-p_t)^\gamma \log p_t$$

$\gamma = 2$最好，FL相比CE可以大大降低简单例子的loss，使模型训练更关注于难例；

Tversky loss

$$T(A, B) = \frac{|A \cap B|}{|A \cap B| + \alpha|A - B| + \beta|B - A|}$$

它是结合了Dice系数（F1-score）以及Jaccard系数（IoU）的一种广义形式，如：

- 当 α = β = 0.5时，此时Tversky loss便退化为Dice系数（分子分母同乘于2）
- 当 α = β = 1时，此时Tversky loss便退化为Jaccard系数（交并比）

因此，我们只需控制 α 和 β 便可以控制**假阴性**和**假阳性**之间的平衡，比如在医学领域我们要检测肿瘤时，更多时候我们是希望Recall值更高。因此，我们可以通过增大 β 的取值，来提高网络对肿瘤检测的灵敏度。其中，α + β 的取值我们一般会令其1；

回归Loss

Smooth L1 Loss⟶CDIoU loss IoU Loss⟶ GIoU Loss ⟶ DIoU Loss ⟶ CIoU Loss⟶CDIoU loss

---

一、Smooth L1 Loss

$$\text{smooth}_{L_1}(x) = \begin{cases} 0.5x^2 & if|x| < 1 \\ |x| - 0.5 & \text{otherswise} \end{cases}$$

从损失函数对x的导数可知：

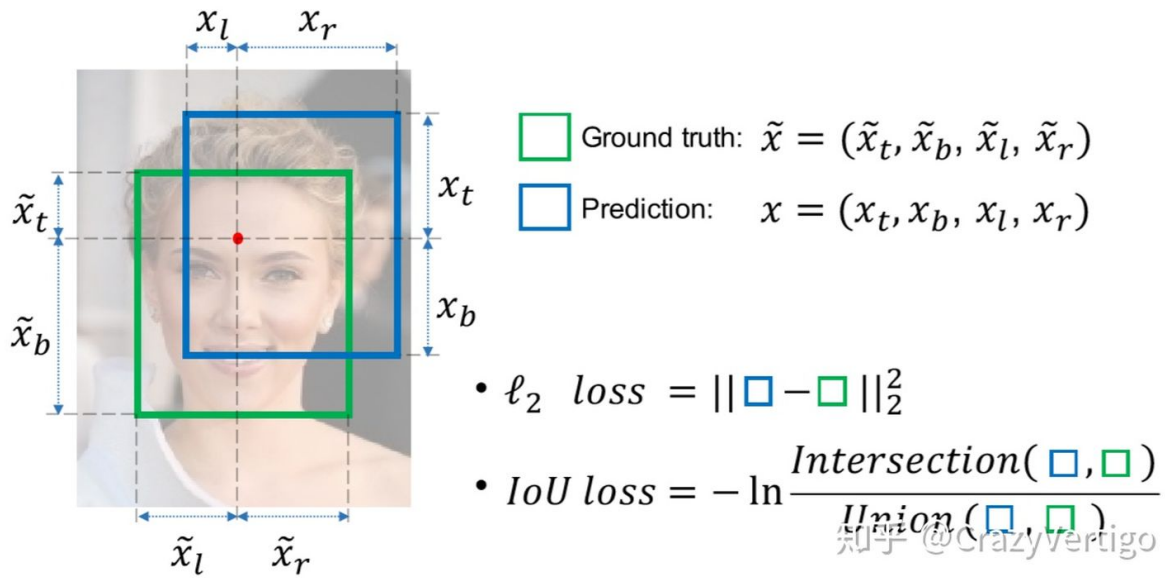
L_1 损失函数对x的导数为常数，在训练后期，x很小时，如果学习率不变，损失函数会在稳定值附近波动，很难收敛到更高的精度；

L_2 损失函数对x的导数在x值很大时，其导数也非常大，在训练初期不稳定；

$smooth_{L_1}(x)$ **完美的避开了** L_1 **和** L_2 **损失的缺点**

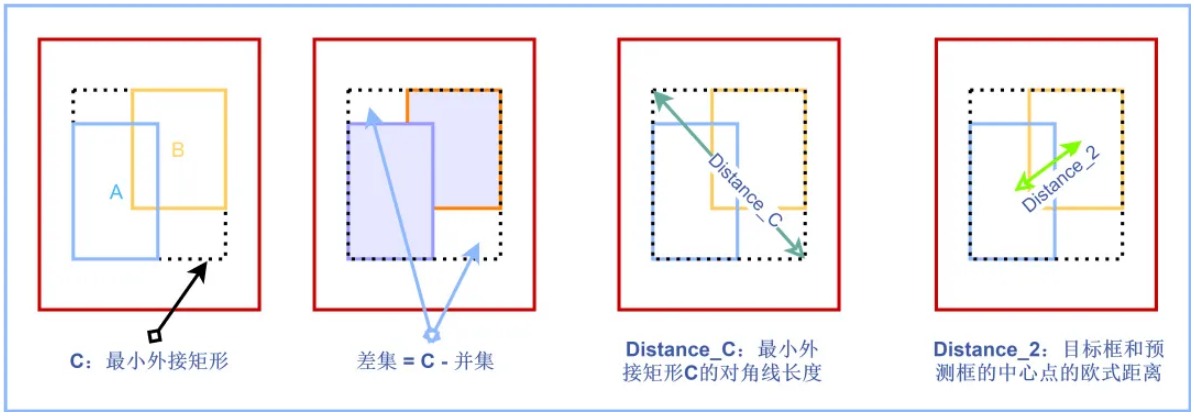
上面的三种Loss用于计算目标检测的Bounding Box Loss时，独立的求出4个点的Loss，然后进行相加得到最终的Bounding Box Loss，这种做法的假设是4个点是相互独立的，实际是有一定相关性的；

二、IoU Loss



三、GIoU Loss（同时考虑了重叠区域和非重叠区域，并没有考虑RP和GT之间的"**差异评估**"，这个差异评估包括中心点之间的距离和长宽比）

- **当预测框和目标框不相交时**，IoU(A,B)=0，不能反映A,B距离的远近，此时损失函数不可导，IoU Loss 无法优化两个框不相交的情况；
- 假设预测框和目标框的大小都确定，只要两个框的相交值是确定的，**其IoU值是相同时，IoU值不能反映两个框是如何相交的**；



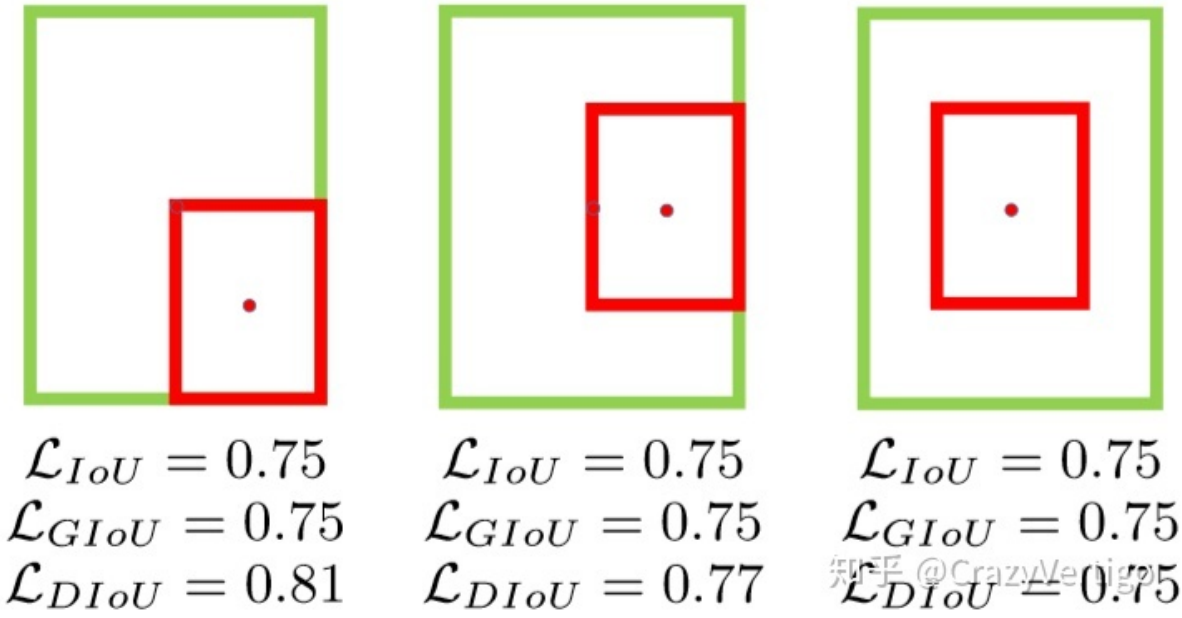
$$GIoU = IoU - \frac{|C \backslash (A \cup B)|}{|C|},$$

$$\mathcal{L}_{GIoU} = 1 - GIoU$$

$$GIoU = IoU - \frac{|C - (A \cup B)|}{|C|}$$

- GIoU具有尺度不变性；
- 当 $A \rightarrow B$ 时，两者相同都等于1，此时 $GIoU$ 等于1；当 A和 B 不相交时，$GIoU\left(A, B\right) = -1$

四、DIoU Loss



- **当目标框完全包裹预测框的时候，IoU和GIoU的值都一样，此时GIoU退化为IoU，无法区分其相对位置关系；**
- **GIoU损失一般会增加预测框的大小使其能和目标框重叠，而DIoU损失则直接使目标框和预测框之间的中心点归一化距离最小，**即让预测框的中心快速的向目标中心收敛；

好的目标框回归损失应该考虑三个重要的几何因素：**重叠面积，中心点距离，长宽比；**

DIoU Loss，相对于GIoU Loss**收敛速度更快**，该Loss考虑了**重叠面积和中心点距离**，但**没有考虑到长宽比**；

CIoU Loss，**其收敛的精度更高**，以上**三个因素都考虑到了**；

$$\mathcal{L}_{DIoU} = 1 - IoU + \frac{\rho^2\left(\mathbf{b}, \mathbf{b}^{gt}\right)}{c^2}$$

$$L_{DIoU} = 1 - IoU + \frac{\rho^2\left(b, b^{gt}\right)}{c^2}$$

其中 b和b^{gt} 分别表示 B和B^{gt} 的中心点， $\rho\left(\cdot\right)$ 表示欧式距离， c 表示 B和B^{gt} 的最小外界矩形的对角线距离；

- 尺度不变性；
- 当两个框完全重合时，$L_{IoU} = L_{GIoU} = L_{DIoU} = 0$,当2个框不相交时 $L_{GIoU} = L_{DIoU} \to 2$；
- DIoU Loss可以直接优化2个框直接的距离，比GIoU Loss收敛速度更快；

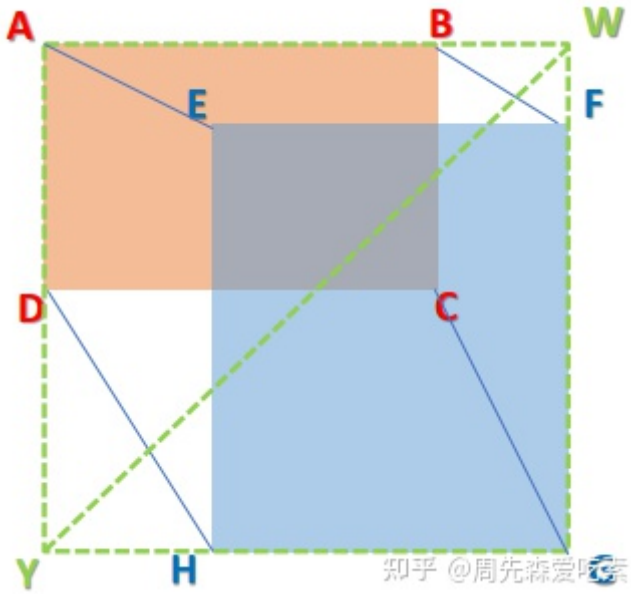
五、CIoU Loss

CIoU的惩罚项是在DIoU的惩罚项基础上加了一个影响因子 αv，这个因子把**预测框长宽比拟合目标框的长宽比**考虑进去；

$$L_{CIoU} = 1 - IoU + \frac{\rho^2\left(b, b^{gt}\right)}{c^2} + \alpha v$$

六、CDIoU Loss

应当是高效且参数较少的



在反向传播之后，深度学习模型倾向于将RP的四个顶点拉向GT的四个顶点，直到它们重叠为止，具体算法如下图所示：

Algorithm 1 CDIoU and CDIoU loss function

Input: RP for region proposal; GT for ground truth;

Output: CDIoU and CDIoU loss;

1: For RP and GT, find MBR;

2: compute $IoU = \frac{|RP \cap GT|}{|RP \cup GT|}$, $\text{diou} = \frac{\|RP - GT\|_2}{4\text{MBR's diagonal}}$;

3: compute $CDIoU = IoU + \lambda(1 - \text{diou})$;

4: compute $\mathcal{L}_{CDIoU} = \mathcal{L}_{IoU_s} + \text{diou}$, $\mathcal{L}_{IoU_s}$ could be $\mathcal{L}_{IoU} = -\ln(IoU)$, $\mathcal{L}_{IoU} = 1 - IoU$, $\mathcal{L}_{IoU} = 1 - IoU$ or $\mathcal{L}_{DIoU}$, $\mathcal{L}_{CIoU}$;

提高模型速度

- TensorRT、Int8
- 将训练时候的3x3conv， 1x1conv， Identity在推理时融合为一个3x3conv（RepVGG）
- Conv和BN进行融合

降低网络复杂度但不影响精度

- 模型压缩：通道剪枝 / 权重剪枝
- 重新设计卷积代替普通卷积：深度可分离卷积，RepVGG

Attention

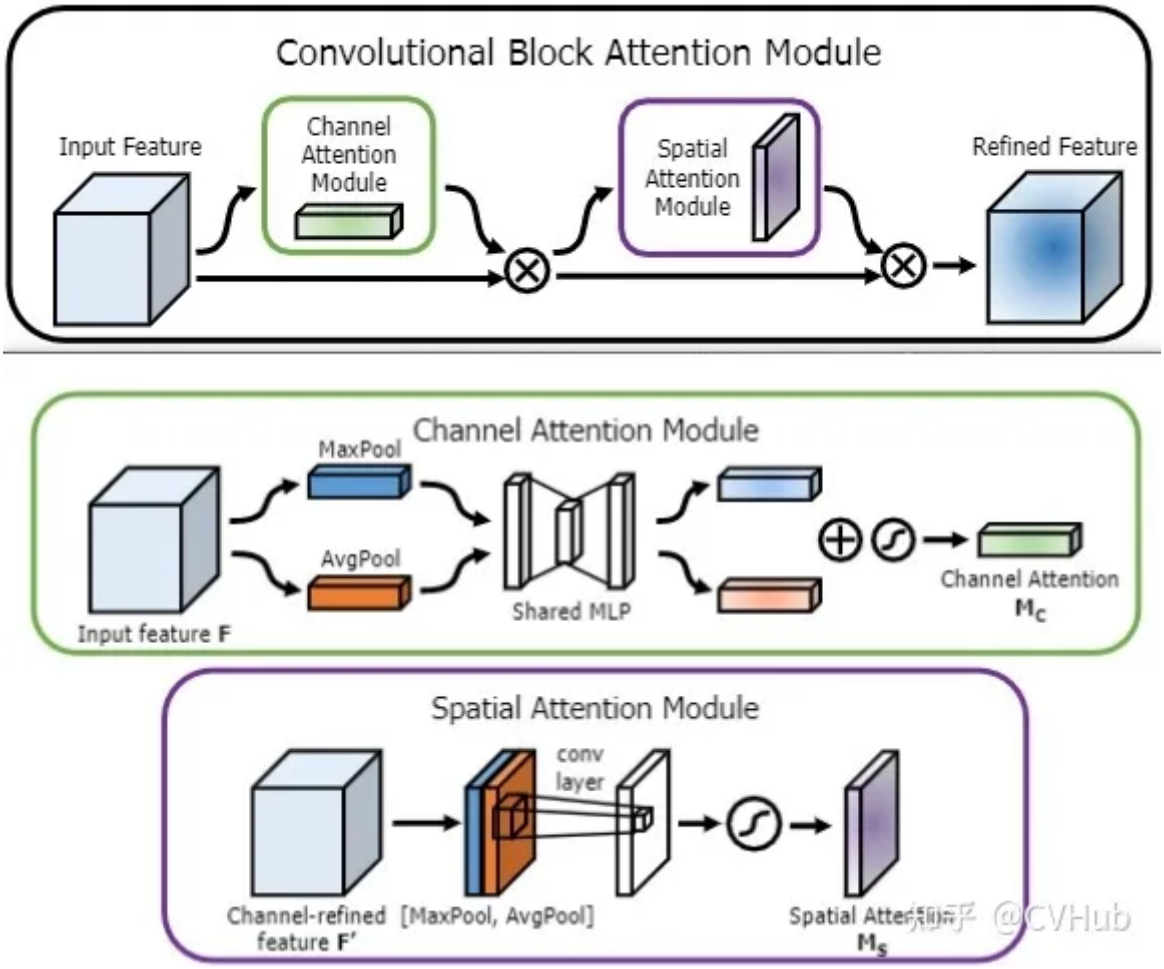
- 一般来说，人类在观察外界环境时会迅速的扫描全景，然后根据大脑信号的处理快速的锁定重点关注的目标区域，最终形成**注意力焦点。**该机制可以帮助人类在有限的资源下，从大量无关背景区域中筛选出具有重要价值信息的目标区域，帮助人类更加高效的处理视觉信息；
- 注意力机制在计算机视觉领域的应用主要使用于捕捉图像上的respective field，而在自然语言处理领域中的应用主要使用于定位关键的token；
- 本质作用是增强重要特征，抑制非重要特征。注意力机制的特点是参数少-速度快-效果好；

通道 & 空间 & 混合 & 自注意力

SENet：GAP + 2个FC + Sigmoid

SKNet：引入多个带有不同感受野的并行卷积核分支来学习不同尺度下的特征图权重，使网络能够挑选出更加合适的多尺度特征表示，不仅解决了SE-Net中单一尺度的问题，而且也结合了多分枝结构的思想从丰富的语义信息中筛选出重要的特征，Split(不同大小卷积核) + Fuse(add + GAP + FC) + Select(Softmax)

CBAM：Channel Attention Module + Spatial Attention Module



- **Self-Attention：**

Attention有什么缺点

Attention模块的参数都是通过label和预测值的loss反向传播进行更新，没有引入其他监督信息，因而其**受到的监督有局限，容易对label过拟合**

模型如何处理大小可变的输入

只处理局部的信息

CNN中的卷积层通过若干个kernel来获取输入的特征，每个kernel只通过一个小窗口在整体的输入上滑动，所以不管输入大小怎么变化，对于卷积层来说都是一样的

- 一般的CNN里都会有Dense层，**Dense层连接的是全部的输入，一张图片，经过卷积层、池化层的处理后，要把全部的单元都“压扁（flatten）”然后输入给Dense层，所以图片的大小，是影响到输入给Dense层的维数的，因此CNN不能直接处理**；
- **FCNN**是一种没有Dense层的卷积网络，那么它就**可以处理大小变化的输入**了；
- CNN处理大小可变的输入的另一种方案是使用特殊的池化层——**SSP**，这种池化层，不使用固定大小的窗口，而是有固定大小的输出。比方不管你输入的网格是多大，一个固定输出2×2的SSP池

化，都将这个输入网络分成2×2的区域，然后执行average或者max的操作，得到2×2的输出；
（ROI Pooling是特殊SPP）

##

Python

迭代器 & 生成器

- **迭代器协议**：对象需要提供next方法，它要么返回迭代中的下一项，要么就引起一个StopIteration异常，以终止迭代
- **迭代器：**实现了**iter**和**next**方法的对象都称为迭代器。迭代器是一个有状态的对象，在调用next()的时候返回下一个值，如果容器中没有更多元素了，则抛出StopIteration异常；
- **生成器：**其实是一种特殊的迭代器，但是不需要像迭代器一样实现**iter**和**next**方法，**自动实现迭代器协议**；Python有两种不同的方式提供生成器：
  - **生成器函数：**常规函数定义，但是，使用yield语句而不是return语句返回结果。yield语句一次返回一个结果，在每个结果中间，挂起函数的状态，以便下次重它离开的地方继续执行
  - **生成器表达式：**类似于列表推导，但是，生成器返回按需产生结果的一个对象，而不是一次构建一个结果列表

生成器的好处：

1.**延迟计算：**一次返回一个结果，它不会一次生成所有的结果，这对于大数据量处理，将会非常有用；

2.**提高代码可读性**；

生成器注意事项：

生成器只能遍历一次

类实例方法 & 类方法 & 类静态方法

采用 @classmethod 修饰的方法为类方法；

采用 @staticmethod 修饰的方法为类静态方法；

不用任何修改的方法为实例方法；

其中 @classmethod 和 @staticmethod 都是函数装饰器

装饰器

- **使得代码更具有Python简洁的风格**；
- **它是一种函数的函数**，因为装饰器传入的参数就是一个函数，然后通过实现各种功能来对这个函数的功能进行增强；
- 在每个**函数上方加一个@**就可以对这个函数进行增强；
- 装饰器最大的优势是**用于解决重复性的操作**；
- 通过闭包来实现装饰器，函数作为外层函数的传入参数，然后在内层函数中运行附加功能，随后把内层函数作为结果返回；

保留元信息的装饰器：使用Python自带模块**functools**中的**wraps**来保留函数的元信息（**func.name**，**func.doc**）

##