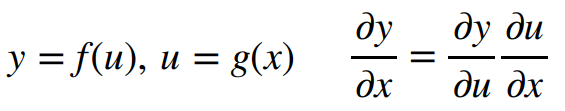
求导：

求导使用链式法则。下即标量对标量求导的链式法则：



trick：

1.想要加速训练，不用32位float而是16位float。

2.验证bug存在：对于分类模型，当output为未经softmax的评估分数时：未训练时，Hinge Loss应该=C-1，C是总类别数；无论在哪一个阶段，Hinge Loss最小为0。否则你的代码应该有bug。

3. 验证bug存在：对于分类模型，当output为softmax后的概率是：未训练时，cross-entropy损失函数值应该是log(C)，C是总类别数。否则你的代码有bug。

4．如果损失函数曲线抖动很大的话，可以：1.降低学习率；2.增大批量大小(吴恩达有讲)。

5.CNN里，卷积层数越深(即便为了计算复杂度服务会把kernel做小一点)，输出通道数越多，往往模型越精准。

6.**bacth\_size不宜太过大**，因为重复的样本会变多，影响训练数据的多样性，模型学到的都是些重复的没用的grad，这**导致收敛变慢**。举个极端的例子就是：训练集的样本全是同一张图片，你选batch\_size为1学得的grad等同于batch\_size为10，但要最终达到同样的acc，你后者要学的轮数是前者的10倍——所以这里我们说**收敛速度，单位常常是epoch**。若batch\_size过大导致收敛变慢，如何解决？这时候可以提高lr，或者你运气好，数据集中类别本身就多，样本多样性丰富，那batch\_size大一点也没关系。但理论上来说，batch\_size变大会提高训练过程中的数值稳定性，而且batch\_size过小也会降低收敛速度。

7.使用GPU要避免：频繁在CPU和GPU之间搬运数据；在GPU上有太多if-else等控制语句。

8.对权重w做更新，据说宜用w-=lr\*grad，而非w=w-lr\*grad，因为后者可能会导致requires\_grad==False，因为后者求得w-lr\*grad得到新值存于一个tensor中，再将此tensor值存入w中，从而丧失了requires\_grad原有的值。但我觉得这样也说不通啊？

9.在train的过程中，越上层的w往往学得越快。

10.train过程中常用的一种手法是，lr随着epoch递减。这便要用到torch.optim.lr\_scheduler.

11.看懂这样的操作：net=nn.Sequential() net.myfeature=nn.Linear(256,1000) net.myoutput=nn.Linear(1000,1)。其中.myfeature和.myoutput相当于是在net中添加的块的名字，名字是自定义的。

12.当模型是多变量预测时，尽量使得train set每个样本的label值足够分散(李沐课上那个案例是单样本的labels均值为0，方差较大)，这样更便于模型预测/收敛。我觉得其本质在于提高样本多样性。

13.提高分类/回归准确率的方法是TTA，即在测试时增广，使得net对增广出来的不同数据分别输出结果，多个结果取平均。其实，你还可以不在测试时对数据做增广，你可以训练多个不同的模型，对同一个输入数据输出多种结果，结果取平均即可。

14.学习率在训练时的变动不仅仅有cosine曲线之类的，你还可以在valid acc曲线趋于不变时把学习率往下调。

15.其实，有时候把valid acc刷太高也没有太多意义，因为这相当于把valid dataset也拿来训练了。

16.当一张图片很大时，若是想针对每一个像素都生成若干锚框，则计算开销太大，可以利用自编码器的思想，先用conv把图片的大小降一层，然后再生成锚框。也可以像yolo一样，不管你图片有多大，它都是按照一定的规则按h\*w生成一些网格，而不是每个像素生成几个锚框，这样不论如何生成的网格/锚框都不会很多。

17.我们要微调在imagenet上训练的resnet模型，常常对输入的图片要做一个torchvision.transforms.Normalize处理，处理的参数是rgb\_mean=torch.tensor([0.485,0.456,0.406])，rgb\_std=torch.tensor([0.229,0.224,0.225])，这是resnet(pretrained=True)对输入图片的像素值(0-1之间)的硬性规定，因为resnet(pretrained=True)自己在训练时就对图片的像素做了这样的处理。但我听说现在pytorch上的resnet(pretrained=True)的输入好像也不需要这样的归一化了啊，不过做一下也没错。

求导：

深度学习里面我们用得最多的是对标量的求导。因为我们求导常常是对loss变量求导，loss变量是一个标量。假如loss变量是一个向量就麻烦了，想象一下一个向量的loss对一个向量x求导，得出一个2维张量，然后再对x求导，得到一个3维张量……以此类推，张量越来越大，还怎么算呢？

人类已知的全部求导方式大概分为3种：符号求导、数值求导、自动求导。

自动求导是深度学习中常用的求导方式。

自动求导要用到一种叫计算图的东西，可以说，**计算图的存在就是为了方便自动求导的**。准确来说，计算图更像是有一种便于你理解自动求导的工具——因为李沐说：“学习pytorch不需要深究计算图，但tensorflow要用到”。

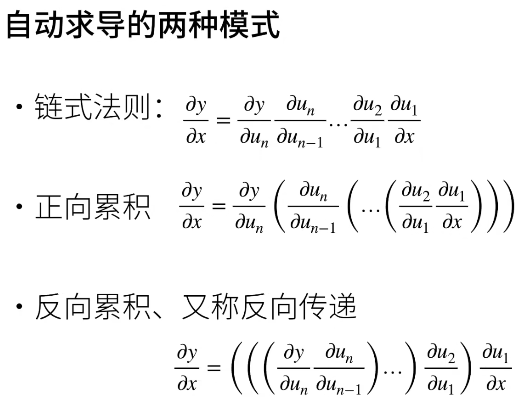
计算图是什么？拿求导举例子，计算图相当于链式法则的图形化体现。其实我觉得，一个算式的计算图很像存储这个算式中缀表达式的二叉树。

计算图是如何构造的？有显式的构造(tensorflow/Theano/MXNet)和隐式的构造(PyTorch/MXNet)。前者对应符号式编程；后者对应命令式编程。前者是先构造出一个符号化的数学表达式，然后最后你要计算这个算式的话，就自己再输入变量，计算结果；后者是边构造数学表达式，边告诉计算机你此时键入的未知数符号的具体数值是什么，让计算机记住，当数学表达式构造完时，算式的结果也就被计算出来了。而**在pytorch深度学习的计算中，我们常常使用隐式构造计算图而非显式构造**，也就是说在做一个计算时，常用命令式编程。

(下面我们考虑的都是pytorch框架里用到的知识，在讲述深度学习时，指的也是pytorch框架下做的深度学习。毕竟李沐的全部课讲的就是pytorch。)

利用计算图，自动求导也可以分为两种模式：正向累积、反向累积/反向传递。

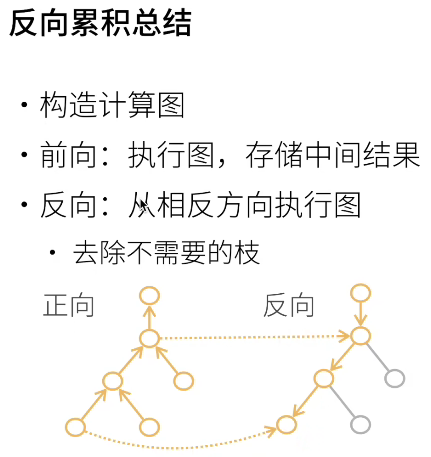
而**在深度学习中，我们用的一般是反向累积/传递。**



反向累积/传递：

反向累积是一种自动求导模式，y(x)对x求导，且y(x)的构造方法是隐式构造。反向累积的过程是什么？代码层面上先定义好x，再由x构造起y(x)——这也就相当于构造起了y(x)的计算图，此方式乃隐式构造。于是当你构造完y(x)时，你就拿到了一张y(x)的正向计算图，且由于是隐式构造，所以计算图构造的过程也是计算图被执行的过程，计算过程的中间和最终结果也被存储在了计算图中。接下来才是反向传递的正戏：你拿到了一张正向计算函数且存储了计算的中间和最终结果的计算图，然后你就可顺着这张计算图依照链式法则反向求导了。

(上面可以补充说明的一点是：你在代码上由x步步计算得到y的过程，pytorch框架会在背后隐式帮你构造起计算图，但假如你在写某段x参与运算的代码时不想要这段代码也被pytorch用来构造计算图怎么办？你给这段代码总体加上个“**with torch.no\_grad()**”即可。如何理解“no\_grad”？grad是求导得到的梯度，no\_grad就是不求导，而计算图的构造就是为求导服务的，既然你不求导了，那我pytorch就不构造计算图了。)



Pytorch中，反向累积的函数是backward，用法是y(x).backward()。理论上来说，y(x)可以是自定义的任何函数，故而构造出来的y(x)的计算图不一定是一棵树，有可能是一个带环的图，具体视你这个函数的正向计算图而定。

但使用y(x).backward()函数有一个前提，就是在构造y(x)之前，你要事先激活x的gard成员变量，方法是x.requires\_grad\_(True)，等价于x=torch.arrange(4.0, requires\_grad=True)#假设你想要个x=[0., 1., 2., 3.]。注意：x.grad被激活时初始值为NoneType。

激活x的grad成员变量有多个好处：1.李沐说的，grad存储最终y(x).forward()求出来的导数值；2.我猜的，给x激活grad变量让y(x,z)调用forward方法时，知道y是从x计算过来的，forward求导是对x求导而非z。y调用forward时怎么知道自己是由x计算过来的？貌似该信息被存储在y的成员变量grad\_fn中。

Grad有个奇怪的特性：当你调用一次y(x).forward()求出对x导数存储在x.grad中时(y一般为标量，x为长度为n的一维张量，则求出来的grad也是长度为n的一维张量)，若是再调用一次forward，则x.grad中的旧值不会被清除，而是会留在那，跟这一次求导出来的是做累积，产生新值。Pytorch这样设计grad的目的在于：便于存储对loss连续求导产生的累积梯度。你要是像重置x.grad也行，调用x.zero\_()即可——pytorch中的方法后带有“\_”一般都是用于重写调用该方法的对象的内容。

优化算法：

优化算法，优化的是一个深度学习模型的参数，得到这些参数的最优值。

我们要训练出一个模型，模型的大概样子我们已经知道，例如是最简单的线性回归模型y=wTx，但这个模型里有很多未知参数，即为w=(w1, w2, w3, ……, wn)，我们训练模型，最终要得到的是w的最优值。

一开始，我们甩给计算机的是一个带有w随机初值的y=wTx，w的随机初值肯定不是我们最终想要的最优值，我们就要用一些方法来训练y=wTx模型，使得w中的某个wi越来越接近我们想要的最优值，这些方法就叫做模型的“优化方法”。

模型的优化方法里有一种最常见的，叫做“**梯度下降法GD(Gradient Descent)**”，它每次对所有样本的损失函数的加和求导，并迭代优化。它就要用到我们上面讲过的求导，求导是对y=wTx模型拟合实际数据的损失函数loss求导，grad=d(loss)/d(wi)。求导的结果用于不断更新wi的值，怎么用？wi=wi” – grad \* n，n在此处被称为学习率，是一个超参数。

GD太贵了，我们对其做改进，每次只从全部样本中取一个固定大小的批量来求导，这叫做“**批梯度下降法BGD(Batch Gradient Descent)**”。

但是“bacth” GD算起来好贵，所以我们对它做改进，每次进来一个样本就求一次导，而不是对一批样本求导，你看一次求导变便宜了。这种叫做“**随机梯度下降法SGD(Stochastic Gradient Descent)**”。

但是BGD求一次导的bacth太大，而SGD求一次导只有一个样本，一轮epoch求导次数过多，所以我们二者取折衷，让BGD的每个批量的大小随着当前的求导情况而变化(怎么变化我也不知道)，得到“**小批量随机梯度下降法MSGD(minibatch SGD)**”。需要注意的是，若是采用小批量随机梯度下降法，则在优化模型参数的过程中可能需要逐渐减小lr的值，特别是当batch\_size=1的极端情况，因为你的最终目的是得到一组模型参数使得所有样本的Loss最小，但你每次更新模型参数使用的仅仅是一个或几个样本，这可能使得你模型参数“下山的路径”并不是总朝着使得所有样本Loss最小的方向，看上去就是你的“下山路径”在震荡，特别是模型参数接近Loss最低点时，震荡最明显，总是无法收敛到最优解，你则需要降低你的lr了，以减小路径震荡，收敛到所有样本Loss最低点。

我认为，任何**优化算法都有两要素**：1.模型的样子；2.训练所用超参数。以线性回归模型的训练举例子，线性回归模型实际上也就相当于单层神经网络，这个神经网络的参数包括权值w和阈值b。于是，对于线性回归模型来说，模型的样子可以有两种表示方法，传统的是y=xT\*w+b，但我们也可以统一规范地用单层神经网络图来表示。训练所用超参数是学习率lr。

我们在实现一个实现某优化算法(例如SGD)的函数时，首先要明确这个函数的作用是不断根据损失函数对模型参数的梯度值来对模型参数进行更新。于是你要给这个函数传入的是梯度值(在pytorch中，用于w和b变量有数据成员grad，你只需要在优化算法函数外调用y.forward()，再往优化算法函数中传w和b即可)；于是这个函数需要实现知道优化算法两要素：模型的样子+训练用的超参数。

优化算法的实现还需要注意两点：1.若是使用的SGD等需要用到梯度的算法，则每次对模型参数做完优化后，要对所有grad作一次清零；2.时常注意在对模型参数进行优化的过程中，不应当对已有计算图进行更新，不进行更新的方法有：1. with torch.no\_grad():#回车 ； 2. detach()。 detach函数貌似是tensor变量存储的模型参数w、b的方法，但具体用法我忘了。

损失函数：

不论是训练/测试/验证集，数据都是以样本为基本单位的，而每个样本中会有多个指标和一个标签label，所以**一个样本的数据就是用一个行向量**+一个标量(对于训练集和测试集来说才有这个标量)来表示，这个标量就是标签label，在讨论训练出的模型的语境下，我们也叫它真实值。

在实际应用中，我们关注的主要不是损失函数的形式，也就是说我们工作的重心不在于损失函数如何通过output与target的计算出损失值，而是主要关注“损失函数值”，因为我们在反向传播对模型参数求导是，不是对损失函数本身调用backward，而是对loss(output, target)计算出来的值调用backward，才能计算出模型参数的grad。

损失函数值本身也可以看做一个以模型参数为自变量的函数，它具备三要素：1.真实值；2.样本指标；3.模型参数。也可以说：损失函数值由真实值和预测值构成。其中样本指标和模型参数共同运算出预测值，运算的法则即是模型本身。对于损失函数值而言，模型参数相当于自变量，真实值+样本指标相当于系数，故而损失函数值求导是对模型参数求导，不是对预测值求导。

理论上来说，仅仅采纳一个样本，用模型依据其指标的预测值和样本真实值的偏差也可以表示损失函数，但我们却往往不仅采纳一个样本，而是多个样本。理论上你训练模型的过程就是一轮轮不断更新模型参数的过程，每一轮更新采用的方法就是优化算法，例如SGD算法，每一轮更新你都要由损失函数对每个模型参数求导计算一组梯度，梯度的值往往用系数——即真实值+样本指标就能表示出来(eg线性回归模型)。而前面我们说过，求损失函数往往采用多个样本，你甚至可以使用训练集中的全部样本，但那样代价太高了，所以才有了“S”GD。

损失函数若是仅仅采纳一个样本，则预测值和真实值的偏差肯定是一个标量，因为真实值label本身是一个标量，而指标和模型参数运算出来的预测值也是一个标量。但若是损失函数采纳n个样本，则偏差的直接结果肯定是一个长度为n的向量，这时就要对这个向量做一定处理，例如对这个向量取范数值，得到损失函数最终的结果。

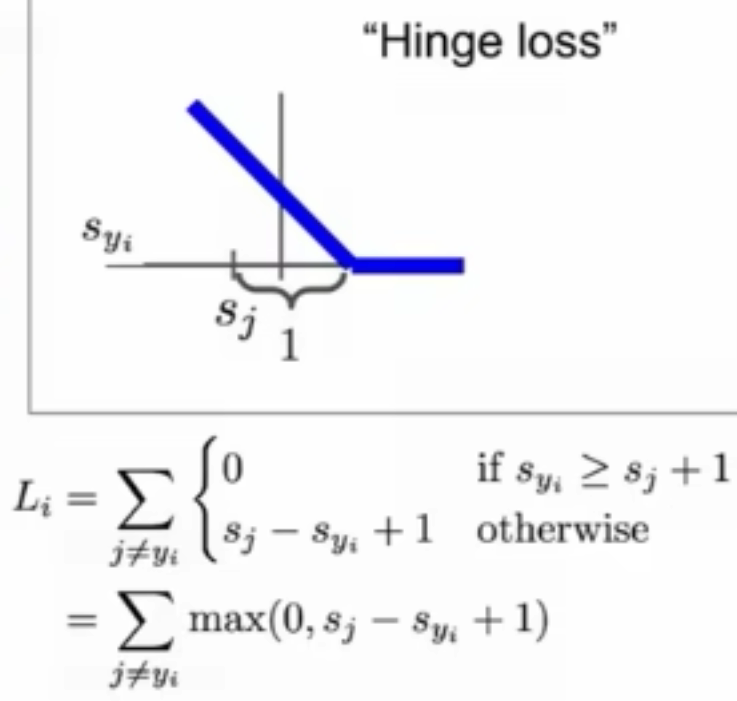
“损失函数”本身也可以表示预测值和真实值的“误差”，损失函数不同的形式则称作不同的误差，例如形如1/2\*(y-y’)^2称作MSE**均方误差损失函数**­——但这个公式针对的是采纳单样本的损失函数，对于一般情况下采纳多样本的损失函数，y和y’有下标i，要做加和然后取平均，“取平均”，故而叫“均方”误差损失函数。

对均方误差损失函数还有一个改进：log\_rmse，先对pred和label做log，再做rmse。

我们知道，sgd每次对param做更新时要原param-lr\*grad，这里的grad是损失函数对param的导数，若是损失函数对批量样本的损失值仅仅是求加和而未取平均，则sgd公式适修改为param-lr\*grad/batch\_size。

分类问题中常用到的损失函数：

假如分类模型中输出的结果没有经过softmax变换，则可以直接对评估分数上Hinge Loss。对于Hinge Loss中为什么不正确分类的评估分数要+1，这是为了确保正确分类的评估分数远大于不正确的，事实上，你不一定要选用1，你可以选用其他数字。



Syi表示正确分类的评估分数，Si表示错误分类的评估分数。

假如你拿到的是结果softmax变换后得到的结果，可以用Cross-entropy Loss。

Softmax模型：

Softmax没有隐藏层。

Softmax回归模型虽然叫回归，但它其实是用于处理分类问题。

Softmax模型的input个数和output个数都不为1，softmax模型的结构本质上可以视作是多个线性回归模型的组合，有多少个线性回归模型，就有多少个output。

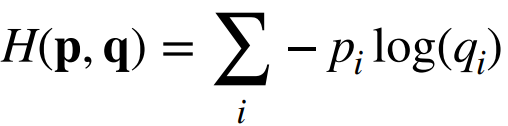
Softmax模型的输出是一个一维数组，每个元素是对相应分类的评估分数(也叫做**置信度，只有分类问题才有置信度输出，回归问题是没有的**)，而我们想得到的理想输出是每个分类的概率，如何将评估分数转换为概率？就要用到softmax函数，对每个output做一个变换——所以，实际上多个线性回归模型的简单叠加是不足以称为softmax模型的，还要在这些线性回归模型的output加一个softmax函数做变换，得到每种分类的概率，这才可以被称作softmax模型。

我们对softmax模型做训练，是要将它的输出结果尽量逼近[0,…1,…0]的形式，即是，若output有n个，则target的输出结果是一堆对应分类的概率，这些概率只有某一个为1，其他全为0。Output和target的损失函数我们一般用交叉熵。

这里我们可以看出来了，softmax模型是：线性回归模型的叠加+softmax变换，想要最终结果逼近target，损失函数一般用交叉熵，交叉熵函数的输入是softmax的结果和taget。但有时候，一些库(pytorch)提供的交叉熵损失函数的输入应当是线性回归模型叠加后直接输出的output，即是评估分数，交叉熵损失函数自动帮你对评估函数做softmax变换。

由于softmax变换的特殊性质，output变换后的结果永远无法得到理想的target，只能逼近。假如你想达到target的话，可以使用一个trick，不将target设为[0,…1,…0]，而是设为”Softlabel”，即是类似于[0.01, …0.9, …0.01]，这样output做softmax变化后绝对能直接完全等于softlabel，即为交叉熵损失函数为0。

关于交叉熵损失函数，我还想补充一些很有趣的信息论的知识。交叉熵损失函数其实源于信息论，你看交叉熵损失函数的形式是：



而信息论中，一条信息的信息量是-log(p)，p代表目标信息是正确的概率，比如说我说楚杰CV课不及格，那么这基本是不可能，它的概率p很小，信息量就很大。

引出另一个问题就是，log的底该是多少呢？我们取几都没关系，但我们约定俗成取2，因为取2后，算出来的结果可以以bit为单位，这样信息量就在实际应用中有意义了，一条信息的信息量=在通信中能用来表示这条信息的最短二进制位数。你想想是不是这个道理，一个硬币抛正面的概率为1/2，信息量=1，我们要表示这样一个结果，至少得用1bit的数据对吧。

那么，交叉熵损失函数与信息量的联系是什么呢？你看它是不是对于softmax模型输出的概率值的加权平均信息量！其中对于正确分类的权值大一点，错误分类的权值小一点，最终训练的效果就是让整个输出的信息量最小，而由于输出的概率值加和为1，所以为了达到这一点，正确分类的信息量会越来越小(输出概率值变大)，错误分类的信息量会越来越大(输出概率值变小)。

感知机：

多层感知机(MLP,multilayer perceptron)。

感知机是用来解决分类问题的，一般分为单层感知机和多层感知机。

单层感知机一般是单输出的，解决二分类问题。且只能解决线性可分问题，局限性较大。于是有了多层感知机，多层感知机可以解决二分类多分类问题，且能解决线性不可分问题，例如做异或运算(二分类，true和false)。

单层单输出感知机本身就是一个线性回归模型，靠激活函数才引入了非线性的部分，其他感知机以此类推。

对于常常出现的多输出的多层感知机，若是将其隐含层全部去掉，其将成为softmax模型。所以，按道理来说，softmax模型是一种特殊的单层多输出的感知机。而是事实上，多层多输出感知机输出的评估分数也要做一次softmax变换后才求交叉熵损失函数。当然，softmax变换nn.CrossEntropyLoss函数就已经帮你集成好了。

线性分类器：

这是对感知机的进一步理解：

1.用于分类的单层感知机即为线性分类器，它不会用到激活函数。线性分类器常用于图片分类，图片要输出，要先flatten成一维张量。

2.一般情况下，类别有多少个，线性分类器的输出就有多少个。但对于二分类而言，即有特殊情况，你可以有两个输出，但也可以仅有一个输出，也能实现二分类。

3.从“**伪分隔面**”角度理解线性分类器：对于一般情况下的多输出的线性分类器，其每一个输出都可以用output=w\*x+b表示，其中w是权重行向量，b是偏移值。而w\*x+b=0(位于该面上的样本output均为0)可以视作n维样本空间(n维取决于线性分类器有多少个输入)的n-1维“伪分隔面”，为什么叫做“伪分隔面”？因为它不是“真分隔面”，它并没有真正实现该n-1维分隔面将其分隔的一边的样本严格划分为一类，另一边划分为别的一种或多种类。每一个output对应一个伪分隔面，每一个伪分隔面让其一边越远的样本更有可能隶属于该分隔面对应的分类，另一边越远的样本更有可能隶属于其他类，只是“有可能”，而非绝对划分，所以最终的分类结果还要看每一个output的评估分数比较相对大小。而其实，上面说的都是多输出多分类的一般情况，还有用单输出实现二分类的特殊情况，在这种情况下，伪分隔面就能完全实现w\*x+b=0严格划分n维样本空间为两类，因为一边的样本w\*x+b>0，另一边<0——这里值得注意的是，不同的权重w可能在n维空间中作出同一个伪分隔面w\*x+b=0，但同一个伪分隔面的一边的样本，对于不同的权重w可能是w\*x+b>0，也可能是w\*x+b，所以一组权重可以唯一确定一个伪分隔面，一个伪分隔面不可以唯一确定一组权重。

4.从“**模式匹配**”角度理解线性分类器：

W\*x+b=OUTPUT，W的每一个行向量对应于一个output，即对应于一个分类。要让某类的图片被成功分到i类，则相较于其他类的行向量，该类的行向量w应当与input最为相近，也就是说，某类对应的w是某类图片的一个“模式匹配”。我们将w行向量reshape(size(原图片))，一定会发现w在reshape后形成的图片与该类别的图像有相似之处。

5.线性分类器无法实现在n维空间中的非线性分隔(例如异或运算。分类问题是离散问题，所以异或运算是分类问题而非回归问题)。

6.在线性分类器利用权重行向量w给图片特征打出评估分数之前，我们常常做这一步操作：对图片像素做一定转换，提取出特征向量，输入线性分类器，而非直接用图片原像素作为特征向量。为什么要这么做？有两个角度的理解。角度一：模式匹配。线性分类器的输入若直接是图片像素，则对应类别的权重行向量w是对该类图片的一个模式匹配，但这种情况无法**处理多模态**，例如“马”类的图片，马头可能朝左，可能朝右，这就是二模态，w做reshape产生的模式匹配图像看上去就像是有两个头的马，这会降低线性分类器的精度。于是我们对图像原像素做一定转换，让多模态的某类图像们通通变成单模态，这样w就只用兼顾这一种模态就可以了。角度二：单层线性分类器无法处理复杂分类。一个特征空间中有两类样本，但要划分开这两类样本需要用到非线性的真分隔面，于是单层线性分类器无法处理它。于是我们不使用原特征空间的特征，而是对样本的特征做转化，成为一组新的特征，在新特征空间中的两类样本能够被线性分类器划分开。这既是将不可分的样本**线性可分化**。

激活函数：

激活函数的本质在于引入非线性性。

对输出层往往不需要添加激活函数。

训练/验证/测试数据集：

在训练集上的误差是训练误差，在验证集上的误差是泛化误差。我们关注的是泛化误差。

训练数据集是用来训练模型参数的；验证数据集是用来验证训练的结果，查看训练过程中设置的超参数训练模型的效果，几次训练下来，通过模型在验证数据集上表现效果的对比，来选取合适的超参数——验证数据集是用来选择超参数的；测试数据集一般用得很少，它相当于最后的高考，训练数据集相当于平时做的练习题，验证数据集相当于模拟考试，我们千万不要把测试数据集和验证数据集混用。

一般给你一个数据集，你要将其划分为训练数据集和验证数据集，如何划分出验证数据集？当你的总数据集过小时，划分出验证数据集的比例太高是一种浪费，我们最好将更大部分的数据集划分出来作为训练集，所以我们常常使用一种K则交叉验证的方式，即是将整个数据集划分为k端，每次将第i段数据用来做验证集，将剩下的k-1段数据用来做训练集，将最终一共k段验证集验证得出的loss或accuracy的平均结果作为验证集的验证效果。K则交叉验证让你最大程度地利用了总数据集的更大部分来训练。其实还有一种更极端的情况，就是假设总数据集共有n个数据，你采用n则交叉验证，这样最大长度利用了数据集，但算起来太贵了，因为你要训练+验证n/1=n轮。

当总数据集足够大时，我们也用不着做k则交叉验证。

正则项：

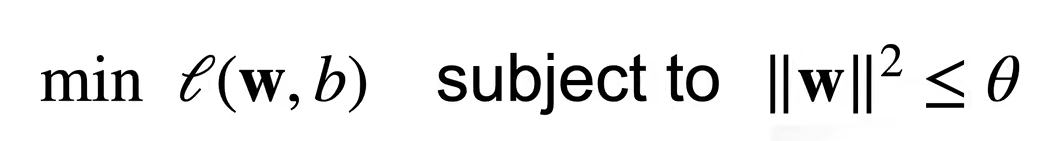
模型容量本意是拟合一个函数的能力，也常用以描述一个模型的复杂程度。

在统计学习知识范畴内，有一个VC维的概念，它一般用来描述一个分类模型的容量，一个分类模型的VC维为：对于容量小于等于n的数据集，给该数据集中的样本任意标label，该模型都能完成其分类，则称该模型的VC维为n。

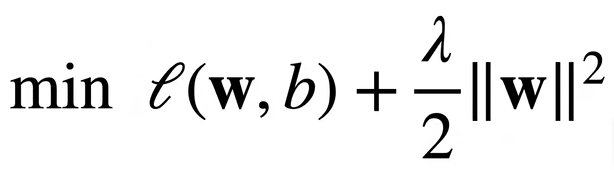
对于一般的神经网络模型，我们衡量模型容量往往不是用VC维，太难了，而是参考的两个指标：权值w(和偏移b)的数量和取值范围。对于简单的数据集而言，我们训练的神经网络不宜过复杂，否则容易出现过拟合。限制模型复杂度的角度也就是w的数量和取值范围，w的数量你根据神经网络的层数和宽度即可限定，而w的取值范围就能牵扯出一个“权重衰退”的知识。

首先我们要明白，为什么权重的数量和取值范围会影响模型的容量，权重的数量好理解，那取值范围又是为什么呢？假如权重的取值范围过大的话，则神经网络可以拟合出一个不平滑且复杂的曲线，反之拟合出来的曲线一般都很平滑。

如何限制w(模型参数)的取值范围？写成数学表达式，有两种写法，一种叫“刚性限制”：

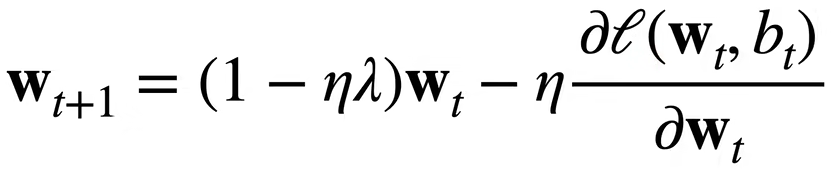


一种叫“软性限制”：



其中拉姆达是个正数，拉姆达越大，对w的限制越大，当拉姆达==0时，对w没有限制。

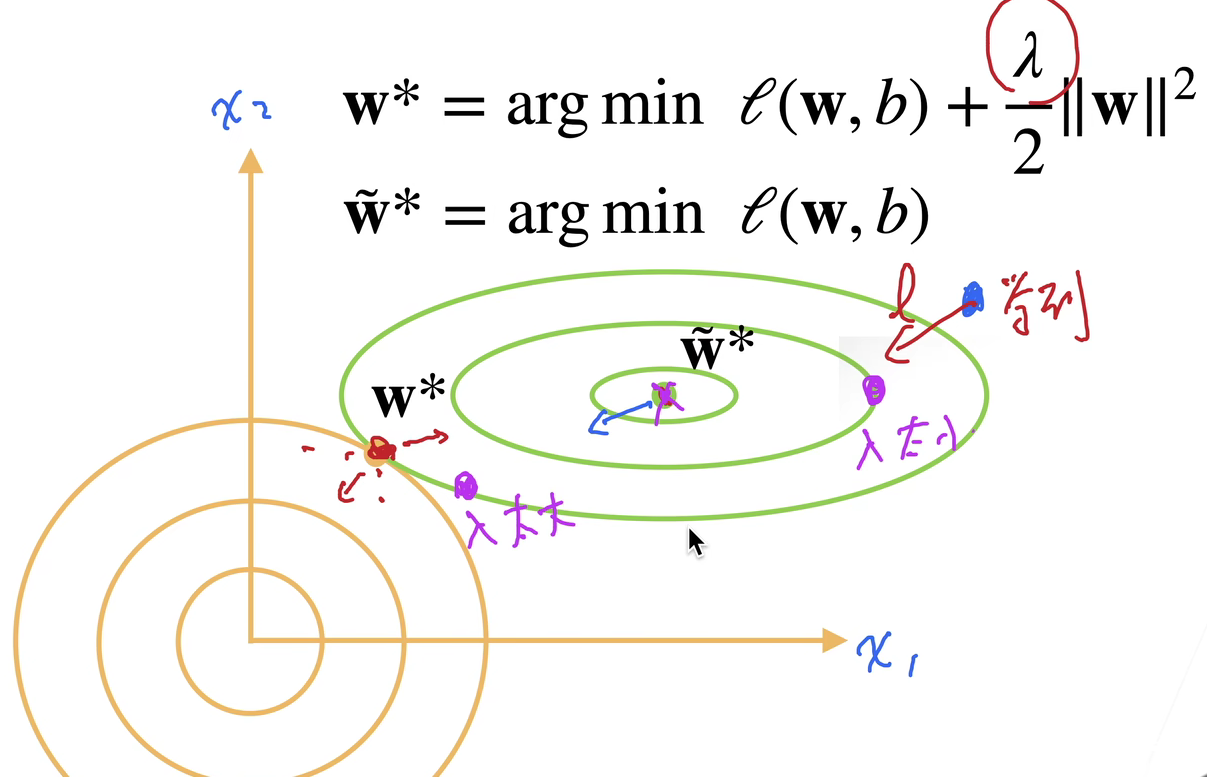
软性限制的形式更容易被我们的机器学习训练过程所体现，所以我们一般也采用软性限制的写法来描述模型参数的取值范围限定。体现在机器学习训练过程中，我们相当于将软性限制整个作为一个新的损失函数，其中的lambda/2\*||w||^2又被称作“**惩罚函数**”。当采用SGD来优化模型参数时，更新参数的表达式变成了：



你可以看到正规的SGD模型参数更新公式里wt的系数为1，这里却乘了个小于1的系数，所以我们叫上述限制模型参数取值范围的方法为“权重衰退”。但在实际的训练机器学习模型的过程中，我们实现权重衰退的方法一般不是改动损失函数成软性限制的形式，因为这会让自动求导变贵，我们一般是给SGD优化器一个weight\_decay参数，即是lambda，直接给wt乘系数，便宜一点。

要注意的是：我们管||w||叫L2范数，其标准写法应该是||w||2。那这里L2范数作为损失函数的一个项，我们为什么叫它L2“正则项”呢？其实，**在损失函数中任何项，只要它起到的作用是防止w取值范围过大，我们都叫它正则项**。推而广之，若是一种手法能使得模型参数w取值范围不过大，以起到防止过拟合的作用，我们就称这种手法为一个**正则**，例如权重衰退就是一种“正则”。其实这里的正则项，不一定要用L2，也可以用L1，只是L1对模型容量的衡量是W矩阵有多稀疏(0元素的多少)，L2则更关注W所有元素的整体分布。这里还有一个trick，如果你只想限制W中某几个参数的大小，也可以针对这几个参数制定惩罚项。

其实，模型训练产生过拟合的根本原因在于训练样本中存在噪音(噪音可能导致模型学习到错误的信息，从而影响模型在未见过的数据上的泛化能力)，这些噪音会导致**训练出来的模型参数w过大(绝对是过大，而非过小或别的什么，这个可以数学证明)**，即是离最优解w变远，所以需要给损失函数加上一个L2正则项将训练出来的模型参数w往小了拉(权重衰退)。但假如你的训练样本中没有噪音，那你训练出来的模型参数w将不会受到干扰，训练出来的模型参数只会倾向于逼近最优解，这个时候就不需要权重衰退了。也就是说：样本噪音是导致过拟合的根本原因，有噪音的训练集越小+模型容量越大🡪过拟合越严重，权重衰退本质上就是解决样本噪音导致的过拟合问题。



限制模型容量以防止过拟合的方法除了权重衰退之外，还有dropout，也叫丢弃法。不同的是，dropout的参数更直观好调，但只能应用于全连接层，但权重衰退应用范围却更加广泛。

有一种防止过拟合的思路叫做Tikhonov正则，基本思路是训练模型时在数据样本中加入随机噪音(其实我不知道这里的加入噪音是在一大堆样本中有几个样本是噪音，还是一个样本中有几个特征是噪音)，使模型具备良好的鲁棒性，也能起到限制模型容量的效果(达成的效果相当于限制模型参数取值范围)。Dropout正则就是一种常用的实现Tikhonov正则的方法。但dropout和Tikhonov原意的不同点是：dropout正则是在神经网络的隐藏层(仅限于全连接层)的输出加入噪音，而非是在输入层的样本中加入噪音。Dropout正则要遵守一个原则是：一层神经网络输出的数据在加入噪音后，这些数据的期望值不会改变，也就是E(x’)=x，其中x’是加入噪音后的输出数据。

Hinton在提出dropout方法时，本意是在训练过程中dropout正则每轮batch能将整个神经网络划分为多个子网络，对这些子网络分别进行训练，最终平均得到的总神经网络模型效果会更好。为什么说是划分为多个子网络？因为dropout方法**将隐藏层中的某些神经元的输出设为了0，那么loss对对应wi(某个神经元输出所连接的wi可能有多个)求导得到的结果为0**，按照SGD优化算法，对应神经元输出连接的wi就不会被更新，所以相当于该神经元输出所连接到的wi没有被训练，相当于缺失了这个神经元，于是dropout从总神经网络中删除了一部分，留下的就是子神经网络(子神经网络神经元输出的结果被做了简单的乘法加法数值运算，以使得输出值的期望跟未dropout的一样)。但是后来的人研究dropout方法时发现，与其说dropout达到了hinton想要的目的，不如说dropout起到了：通过在模型训练过程中加入噪音，限制模型参数取值范围从而避免过拟合的作用。所以dropout应该是一种正则，一种Tikhonov正则。

用pytorch实现dropout可以在神经网络中加入nn.Dropout层。我记得是：dropout层只在训练时起作用。所以，在训练net时，你要用net.train()语句，在推理时，你只需要用net.eval()就能注销掉dropout层了。

有一个很能反映Tikhonov正则思想的正则项，叫做“数据增广”，其中对图像数据进行增广的方法在torchvision.transforms里有提供。图片数据增广里有一个很奇怪的增广叫mix-up增广(好像也叫跨图片增广)，它不仅要将两张图片的像素进行叠加，还要将相应label进行叠加——而除了将像素进行叠加，还可以把两张图片拼成一张图片，同时label也加权求和。

数值稳定性：

数值稳定性在神经网络训练过程中表现为模型训练时不会因为输入数据的微小扰动而性能大幅震荡，甚至无法收敛。

事实上，数值不稳定的原因中，梯度清零的情况多于梯度爆炸。

不一定参数多，或者层数深的模型数值就越不稳定，因为若是一个模型层数较深，但它每一层运算的output值都不大不小，然后运算的时候一层一层累积起来也是靠的加法而非乘法(虽然我也不知道靠加法如何累积其层来)，那么它也是不一定会出现严重的梯度爆炸或消失的。

一个模型要保持“数值稳定性”，体现在两个方面(主要是前者)：1.训练过程中不要出现梯度消失(导致w更新速度过慢)或梯度爆炸(导致产生inf或nan)，也就是说梯度维持在一定范围；2.模型每一层的输出值不要过大或过小，维持在一定范围(事实上**我认为这一点也是为了满足第1点，每一层output范围合适，求导结果也能缓解梯度爆炸/消失。想想对(WX)^2求导**)(后来我才学习到，原来要是每一层的输出值太大了，也可能会导致16/32bit浮点数溢出，模型完蛋)。

其中最重要的是第1点。

说具体点：正向角度看，我们希望每一层的输出h期望都为0，方差都相等；反向角度看，我们希望Loss对每一层的输出h(为什么不是权重w？因为数学推导出来是这样。其实都一样。)的导数期望都为0，方差都相等。

而上述两个方面的要求我们都无法完全满足任意其一，我们所做的一切都只是缓解，或者说，尽量防止数值不稳定。我们如何缓解数值不稳定？可以从模型参数初始化和超参数设置(激活函数)两个角度入手。

经过数学推导(具体过程不提及)，尽量满足数值稳定性，我们需要：1.模型参数w(视作随机变量)**初始化时**，每个参数的期望应当为0；2.模型参数w**初始化时**，每个参数的方差应当满足上一层的宽度\*这一层的方差=1，以及这一层的宽度\*这一层的方差=1；3.激活函数在x=0附近应当尽量接近于y=x。

上述3个条件中的第一个很好满足。

第二个条件的两个子条件难以同时满足，怎么办？我们采用“Xavier初始”，即是我们折中一下，将两个子条件合并为“这一层的方差=2/(上一层宽度+这一层宽度)”。

第3个条件要满足，根据泰勒展开，我们选用tanh或relu激活函数皆可，sigmoid不太合适，除非你对它做一点变换：f=4\*sigmoid-2。

有一个疑问在这里：反向来看，为了维持grad不消失/爆炸，我们让dLoss/dh维持在一个小范围内，这是不得已而为之，可以理解。但为什么要求正向来看(每一层)输出h也维持在一个小范围内？不怕损失模型的表达能力吗？其实不怕，因为模型的最终输出值在什么范围都无所谓，关键是要看你如何描述这个输出值，比如输出的像素值在0~1，你可以描述为0~255的像素值。

问题：为什么要把w初始化时期望设为0？不能是其他值吗？我也不知道，数学推导时，有个假设前提，假设的就是w初始化时期望为0。而且我怀疑，w初始化期望为0，方差符合Xavier初始，激活函数也满足要求，能保证数值稳定性——但在后面对w更新的过程中，w的期望和方差还能不能维持初始化的状态？假如不能，岂不是不能保证输出稳定性了？我想这也行就是为什么说我们所做的只是**尽量**保持数值稳定性，缓解数值不稳定吧。

**防止梯度爆炸还可以在一个地方做优化，即是使得输入数据不至于太大(取log，减mean()等)，然后学习率不能太大**，虽然我觉得这种优化是治标不治本。

处理在模型参数初始化和超参数设置等角度缓解数值不稳定外，还可以用梯度剪裁、批量归一化等手段缓解数值不稳定。

在深度学习中，遇到的最多的数值不稳定的问题是梯度消失，而解决梯度消失的根本性方法是residual connection，其源于ResNet。其防止梯度消失的方法可概括为深层梯度的计算“乘法变加法”。

梯度剪裁：

梯度剪裁很简单，梯度剪裁专攻梯度爆炸。

梯度剪裁就是将模型训练时反向传播求导得出的每个模型参数的梯度拉成一条向量，然后你求这个向量的L2范数，看它是否大于theta，小于就不管，大于就让它的L2范数压成theta，theta值是你认为设定的。

批量归一化：

Batch Normalization，normalization一般翻译成归一化，有别于regularization的正则化。

BN一开始提出来是为了：1.解决梯度清零，提高收敛速度；2.但它也附带了正则化的功用。

BN常置于conv或linear后面，激活函数或pooling前面。

前面我们讲过从模型参数初始化和超参数设置角度保持数值稳定性的3大条件，但它们是从模型参数初始化入手缓解梯度爆炸和梯度清零的。这里的BN是专攻梯度清零的，且它的解决方案不是针对模型参数，而是针对每一层的输出数据X，它相当于给神经网络新加了一种层，让X符合一种归一化/标准化分布，从而使得模型深层的grad和浅层大小相当，缓解梯度清零——具体怎么样的分布为什么能缓解梯度清零？这是数学上的问题，我们先不讨论，会用就行。需要注意的是，这里的BN层也有需要学习的参数，所以BN层不仅仅在训练时发挥作用。

BN是如何通过缓解梯度清零来实现提高收敛速度的？梯度清零带来一个问题：靠近输出层的grad比较大，靠近输入层的grad太小，于是lr\*grad导致浅层w收敛快，深层grad收敛慢(有一个很有意思的点，浅层的w很快就学习完了，但等到深层的w慢悠悠稍微改变了一点点，浅层w又得全部重新学习了)，而决定一个舰队航行速度是最最慢的那艘船，于是模型收敛速度慢了。同时，你也不能通过提高lr来提高收敛速度，因为lr大了，浅层受不了，lr小了，深层受不了。于是你用了BN，BN让浅层和深层的grad大小基本均衡，深层grad\*lr变大了；同时你可以放心地提高学习率，不用担心grad过大导致浅层受不了了——于是收敛速度变快了。

BN是如何附带了正则化的功能？还记得Tikhonov正则化吗？BN相当于给模型的每层输出X加入噪音，从而像dropout一样实现了Tikhonov正则化。但要注意的是，BN和dropout不能混用，否则模型效果很不好，这是前人通过实验得出的。

对BN公式的理解，看不看都行：BN按道理来说，只用将每层输出的X减均值除方差实现归一化即可，为什么还要给已经实现归一化的结果做一次线性变换呢？(线性变换的两个系数即是BN层需要学习的参数)这是因为单单X减均值除方差的话，X的期望就是0了，而对于很多激活函数而言，其在0点近似于y=x，这样激活函数的作用基本没有了，它没有引入非线性性，为了防止这种现象，对已经归一化的X在做一次线性变换，使其期望不为0，照顾激活函数。

实际使用时，torch.nn里有现成的BN层，BatchNorm1d用于全连接层的一维输出，BatchNorm2d用于卷积层的2维输出。

在学习BN的过程中，发现了一个很有意思也很重要的点：神经网络中，不论是全连接层还是卷积层，输入输出的张量至少是2维，而且，**第一维必定是批量维(代表样本个数)，第二维必定为特征维**。对于卷积层(4d)来说，第二维还是通道维，卷积层的输入一个样本，有多少个通道表示该样本有多少个特征。顺着来，还有一个很有意思的点：kernel=1x1的卷积层相当于一个全连接层，对于这个全连接层而言，若一次输入一个样本，该样本有n个channel，每个channel有m个像素，且输出有p个channel，则相当于给全连接层的的输入有m个样本，每个样本n个特征——本来对于卷积层而言，应该是1个样本，n个特征的。

卷积层：

下面说的卷积层都是二维卷积层，二维卷积又称作二维相关操作。

卷积层时怎么来的？卷积层是由全连接层推导来的，它本身是一个特殊的全连接层(这个全连接层的权重大多数都是0)。我们希望有这样一个全连接层，它能提取出一个图片的特征。

提取出一个图片的特征，这个提取器必须遵守两个原则：1.平移不变性；2.局部性。对于全连接层来说，这个提取器就是层的权重。

对于输入是一个矩阵，输出也是一个矩阵的全连接层来说，其权重张量是4维的，好理解。

于是，我们先让这个权重张量遵守局部性，于是对于输出的神经元，我们只需要输入矩阵中某一小块的数值，这便使得权重张量产生了大量的0，虽然非0部分大小和值不一定相等；然后让这个权重张量遵守平移不变性，于是对于输出的每个元素而言，其对应的权重张量非0部分的大小和值是一样的，便唯一确立了卷积核。

上面的特殊全连接层是卷积层，那卷积核是什么？是输入输出都是矩阵的全连接层的每个输出神经元对应的权重张量的非0部分。本质上，全连接层的权重张量是4维的，但只考虑卷积核，我们常说该卷积层的输入输出和卷积核都是二维的，我们做的是二维卷积。

其实严格来说，我们用的应该叫做二维交叉相关，但由于不考虑数学上的表示，在实际使用过程中，二维卷积和二维交叉相关是一样的，所以我们直接叫二维卷积了。

卷积核的大小我们又称感受野。

卷积层有两个超参数，padding缓解减少量，stride倍增减少量。

卷积输出的结果叫做输入的feature map。

卷积层卷积一次输出一个像素值，该像素值的计算也可以有偏移量bias，一般是有的，你可以给pytorch的conv函数传bias=False让它没有。

卷积层等价于一个受限的全连接层(kernel=1x1相当于一个全连接层了，常用于改变输出通道数而不改变feature map大小，要改变也可以，stride变一变)，相比于全连接层它的**模型容量要小很多，所以过拟合现象不那么严重**。

设计一个CNN时，往往卷积核的大小越大，卷积层数越浅，这是为了计算复杂度服务，不然卷积层数也做多了，计算太贵，而且也容易过拟合。有时候一个kernel size太大了，我们不仅要降低模型深度，还要减少输入输出的channel数。

一个CNN，每层conv在提取特征输出feature map，越下层的feature map每个像素提取到的特征越局部，越上层则越整体；越下层的feature map越还原图像的原貌，越上层的feature map越抽象。

**有些CNN在某些情况下对输入图片的大小有要求，这些CNN可能具备以下特点**：1.CNN每个stage使得feature map的size减半，假如input的size太小，可能使得未达到预定深度时，feature map的size就小得无法减半了；2.CNN最后部分有flatten操作，然后接全连接层，全连接层的input有数量要求，假如输入图片的size不对，则feature map到达flatten后，得到的一维向量的size可能就不符合全连接层的input要求；3.……

池化层：

卷积层对输入图片各特征的位置十分敏感，例如同一张图片，你稍微让它左移一个像素，那后序的神经网络可能就认为卷积层提取出来的特征就完全不一样了。

为了缓解这种现象，我们引入了池化层，相当于对提取出来的特征做一下模糊，让卷积层对图片各特征的位置不那么细致入微地敏感，提高模型的泛化能力。

其实，池化层还有一个作用是将feature map的size变小，减少计算量。

Torch.nn默认的池化层函数的stride等于池化核的大小。

但现在池化层用得很少，主要有两个操作淡化了池化层降低卷积层敏感性的功能：1.池化层的操作可以被卷积层学出来；2.输入模型的样本本身做了很多旋转，平移等操作，卷积层能看到不同位置的特征，也就不会认为两个特征位置稍微不同的图片特征不同了。

我认为，**降低卷积层对特征位置的敏感度，是不是等效于防止过拟合？**

经典卷积神经网络：

CNN的设计从VGG开始引入了“块”的设计思想，VGG、NiN等都是设计CNN的思想。

**我们描述一个CNN的结构，常以stage为单位，feature map的size每减少一半，就是一个stage**。而且，我们在设计时，常常feature map size减少一半，channel数就增加一倍，这是为什么呢？因为卷积往往会使得feature map的size不断减小，虽然是在summarize information，但终究会导致信息的丢失，不过我们要是使得channel变多，则虽然一张feature map蕴含的信息随着size变小了，但是feature map却变多了呀，保留了很多信息。

LeNet

AlexNet

VGG：VGG更像是一种架构CNN的思想，它是对AlexNet的改进，它的基本构造单元是若干VGG块+3个全连接层。全连接层有特定的模样，它和AlexNet最后的3个全连接层一样。VGG块也有特定的构造细节，对于一个VGG块来说，它的超参数只有其每个卷积层的通道数和卷积层的层数。VGG网络的写法常有：VGG-16、VGG-19……后面的数字表示该VGG网络中含可训练参数的层数，也就是说，池化层、dropout层和激活函数层不算。

NiN：基于VGG改进，从此，去除了CNN末端的全连接层。NiN有属于自己的NiN块，但它将最后的全连接层取缔，改为一个全局池化层，池化得到的结果即为评估分数。去掉了强大的学习能力强的全局池化层，虽然减缓了模型收敛的速度，但却缓解了过拟合，提高了模型的泛化能力与预测精度。

GoogLeNet：inception块。Inception块相比于同样层数的3x3或5x5卷积，模型容量都要小，你看，它有很多层都是1x1的卷积层专门用来降低通道数，从而降低模型容量——这样做的目的也是为了减少后面3x3或5x5卷积的计算负担。

ResNet：残差(residual)块。残差块的residual connection使得真正的“深度”学习成为了可能，能有效防止深度学习中常见的梯度消失。

并行计算实现高性能：

在训练过程中，用并行计算实现高性能，往往会被卡在设备通讯的瓶颈上(包括读取内存)。并行计算中，计算/通讯比越高，计算的性能就越高。计算即为FLOPS，通讯可以理解成每做一次flo，要从其他设备那里读取多少bit数据(实际上，通讯时间的影响因素还有带宽)。我也不知道为什么，通讯的耗费可以等效为模型的大小，模型越小，通讯耗费越少，越适合做并行计算，例如模型容量：inception<ResNet<Alexnet。

多GPU训练：

为了将一批量的数据拆分，torch.nn提供nn.parallel库里的scatter函数，但只有在手动实现多GPU训练时你才会用到这个库手动拆分数据。

手动实现多GPU训练是很困难的事情，若要直接调包则很简单，直接将net通过nn.DataParallel复制到不同GPU上就行，在用net(X)时，net即会自动帮你拆分X并多GPU运算。

我们实现多GPU训练时，看模型训练的速度，衡量标准往往是多久训练完1epoch，若是你发现多GPU的速度不比但GPU快多少时，原因往往有：1.计算量本身就小，没把GPU的core调动起来；2.大部分时间花在了内存访问而非数值计算上……处理方法往往是，提高batch\_size的大小，但batch\_size越大，收敛速度越慢，看你如何权衡了。

分布式训练：

这和多GPU训练其实没有本质区别。

进行分布式训练要注意的一些点是：尽量避免时间浪费在设备时间的通讯上，如同多GPU训练中让数值计算次数多于内存访问次数一样。如何做到？可以提高batch\_size，尽量使得数值计算的时间大于设备通讯的时间，且较好地调动GPU的多核。

微调：

微调是迁移学习领域的一种算法。

这里我们先只讨论分类器的微调，我们要先了解分类器的本质：任何分类器都可以视作两部分：最后的线性分类器(+softmax)，以及前面的所有特征提取层。特征提取层将输入数据的所有特征提取到一个线性可分的语义空间上来，然后在该语义空间上做线性划分。但在有些CNN(例如NiN)里也可能干脆隐去线性分类器，因为最后只是一个池化层来提取最后的特征，池化完也就分类完了——但你要知道**分类器的本质在于将特征提取到线性可分的语义空间中**。

由上述可以知道，对于一个CNN分类器而言，越深的layer越通用(低层次的特征)，越浅的layer参数越专有化(特定数据集相关特征)。

微调的通用做法是，我们拿到一个pretrained的模型，将其原本线性分类器换成一个新的，然后在目标数据集上对“旧特征提取层+新线性分类器”做训练。我们可以在此基础上改进，下面几种方法是步步递进的关系：1.整体的学习率改小，这是一种正则项。因为原模型已经pretrained了，特征提取层已经很成熟了；2.线性分类器的学习率调大，比特征提取层大得多，因为新的线性分类器权重要重新训练要它更快点。虽然根据数值稳定性部分的知识，线性分类器的grad本身应该比特征提取层的大；3.把较深的layer的w固定不动，不train了；4.若是原数据集的label有一部分与新数据集重合的话，可以选择不完全删去原模型线性分类器的w，可以保留重合label分类对应的w，置于新模型中；5.在一些极端的情况下，若是就数据集的label完全包含新数据集的label，则可以直接在旧模型的输出层加上一个线性分类器，旧模型的参数不变(param.requires\_grad=False for param in net.parameters()即可)，只训练新线性分类器的参数。

微调带来的好处是：更少的epoch与时间可以带来更高的模型精度。

锚框用于目标检测：

一般的流程为：

1.导入一张图片(图片本身有真实框的labels，每个**真实框有标号，标号往往对应着某种类别**，狗或猫)，按照某个**锚框生成算法**生成大量锚框，每个锚框为一个训练样本。锚框生成算法我们假设用最简单的，就是每个锚框生成一组由参数size和ratio决定的不同大小的锚框，s和r怎么确定呢？一种方法是随机确定，一种方法是统计图像中groud truth/bounding box的大小，然后依照这些bounding box的大小来生成决定s和r，为什么这样做？因为万一你要检测的物体是电线杆，那么你这个s和r的参数会选得很极端，你怎么知道s和r的理想选取值会怎么极端啊？；

2.由每个锚框与每个真实框的IOU值，根据某个**前景框筛选算法**选出部分锚框作为前景框，分别和某一个真实框的标号相对应。一般我们针对一个真实框，你可以选出多个前景框来跟它对应，用以训练(反正模型推理得到了该真实框的多个bouding box，你还要做nms筛选)；

3.根据某个**偏移量算法**计算出每个前景框到对应真实框的偏移值；

4.一张图片的每个锚框为一个训练样本，上述过程你已经打好了每个训练样本的锚框信息，**锚框信息=(1.该锚框(1)是否与(2)哪一个真实框标号匹配？2.该锚框到对应真实框的偏移值？)**；

5.你可以拿打好的样本的锚框信息去训练模型了；

6.模型做预测。模型不是直接预测和真实框相匹配的预测框，而是预测出每个锚框的锚框信息。后面还要通过一定手法根据这些锚框的锚框信息🡪前景框🡪候选预测框🡪预测框。照样导入一张图片，图片按锚框生成算法生成大量锚框，每个锚框即样本，模型对每个锚框预测出label。根据label，从这些锚框中挑出前景框，再根据偏移量算法将前景框换算成候选预测框。每个真实框标号对应多个候选预测框，你下面还要对每个真实框标号对应的多个候选预测框中选取出最终的预测框。

7.接下来你就该对每个真实框标号对应的多个候选预测框做选择，选择常用NMS(none maximal suppress)算法。算法大概为，对每一种真实框标号对应的多个候选预测框做以下操作：(1)对每种类别的候选预测框crop的图片部分做softmax分类，对应类别/真实框标号的softmax输出值为该锚框的针对该类别的置信度(第一次计算出置信度后，可能还要设定一个不用太高的阈值，对这些候选预测框做一次预筛选)；(2)将这些锚框的置信度排序，(3)选出置信度最高的锚框，剔除出序列，作为最终预测框之一(事实上,nms算法选出来的每一个真实框标号对应的预测框一般有多个)；(4)计算出序列中剩下的候选预测框和第(2)步选出来的预测框的IoU值；(5)根据预先确定的IoU阈值，剔除掉IoU值大于该阈值的候选预测框，序列中剩下的候选预测框作为新的已排好序的序列；(6)若是序列中仅剩下一个锚框，则nms完成，否则回到第(3)步继续。

7.第6步中你已经用nms从模型预测出来的大量候选预测框中选出了每个真实框标号对应的预测框。由于NMS算法的流程，你选出来的的bounding box的数量不确定，刚好就对应了一张输入图片中被检测物体数不确定的情况。还有就是，你做完nms后，得到的bounding box只是局部非极大相似锚框被消去了，局部置信度极大锚框被保留了，但你局部置信度极大，就一定达到够用吗？不一定，你可能还要设定一个置信度阈值，对nms筛选出来的局部置信度极大的锚框做一轮**二次筛选**。

总结：根据锚框进行目标检测和我们之前学的东西(例如图片分类)都不一样，有以下几个区别：1.以前样本是大量图片，现在是一张图片的大量锚框；2.以前训练模型用的label是最终真实值，现在是与真实框相关的锚框信息(当然，最终你想要的真实值还是真实框)；3.以前模型预测出来的值就是你想要的东西，现在模型预测出来的值是锚框信息，你还要🡪前景框🡪候选预测框🡪(nms)🡪预测框；4.以前训练模型用的label都是数据集原原本本给你了的，现在label都是你要根据真实框手动给每个锚框打上锚框信息。

也就是说，**数据是**(图片，真实框)🡪(锚框(前景框)，锚框信息)🡪训练+预测🡪(锚框，锚框信息)🡪(图片，真实框)。**算法是**：锚框生成算法🡪前景框筛选算法🡪偏移量算法🡪【模型训练】🡪锚框生成算法🡪【模型预测】🡪前景框筛选(一个步骤，不是算法)🡪偏移量算法🡪NMS。

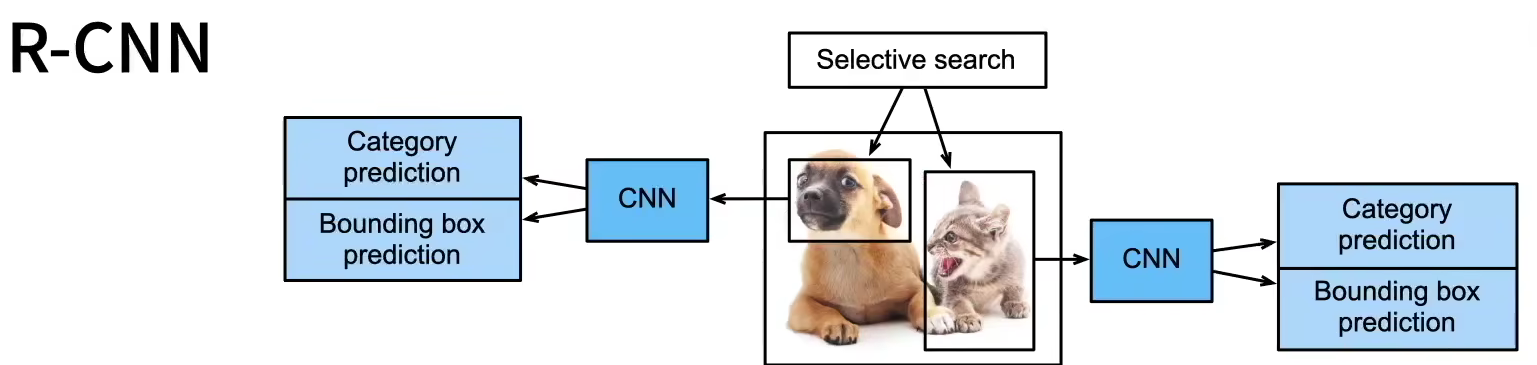
锚框目标检测实例一：R-CNN

区域卷积神经网络，常常用来做锚框目标检测，我们这里就拿它来作为我们的“锚框用于目标检测”的一个实例。

一般R-CNN的算法流程是：1.启发式算法搜索选择锚框；2.使用预训练模型(常常是CNN)来对每个锚框抽取特征，抽取出来怎样的特征怎么用？一种常见的做法是CNN输出一个矩阵，把这个矩阵拉成向量，即为feature map；3.训练一个模型(早年间常常是SVM)来根据特征对每个锚框crop的区域分类；4.训练一个模型(早年间常常是线性回归模型)来预测锚框对ground truth的偏移。

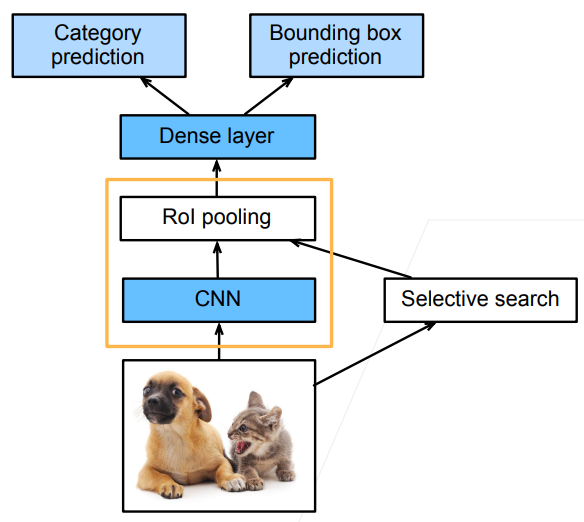
第1步锚框选择好后，第2步训练模型了，训练模型要解决的一个首要问题是把每个锚框crop下的样本batch化，但每个锚框大小不一，怎么batch？有种方法，把锚框均匀分割成n\*m块，输出每块的最大值——这叫n\*m**ROI(兴趣区域)池化层，它的作用就是帮助批量化不同大小的图像**。

RCNN流程再总结就是：启发式🡪ROI pooling🡪CNN锚框🡪SVM+LinearClassify



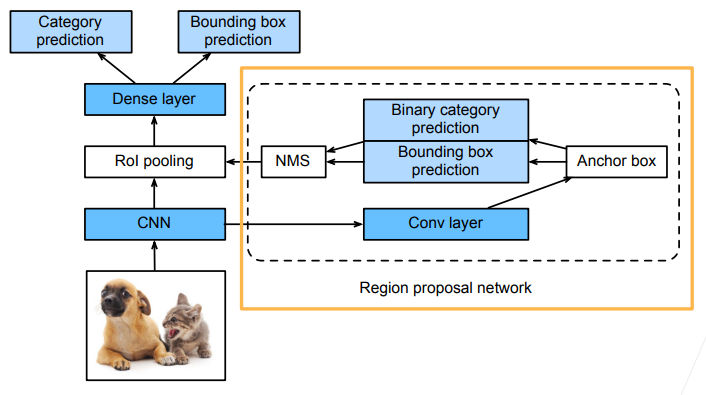
但上面的流程有个缺点，第2步要对每个锚框抽特征，计算量太大，于是我们提出了fast RCNN。改进是：1.CNN先对整张图片抽取特征，抽取出来的特征是一个矩阵；2.启发式算法选出锚框；3.锚框在原图上的区域映射到CNN抽取出来的特征上，等比例地crop下对应特征；4.锚框crop下的特征矩阵必然不能被batch化，这个时候要对特征矩阵做ROI池化，然后拉成向量；5.把向量送入SVM或线性分类器预测类别，送入线性分类器预测偏移量即可。

Fast RCNN流程再总结就是：启发式🡪卷一下原图得到fetaure\_map🡪锚框crop掉fetaure\_map🡪ROI pooling🡪SVM+LinearClassify。



但有人还是觉得Fast RCNN不够快，因为启发式算法选锚框太慢了，所以提出来了Faster RCNN，具体不多讲，就是相当于用一个RPN区域提议网络替换了启发式算法，而RPN就是相当于一个糙一点的目标检测啦，我只解释一下下面这张图的RPN：

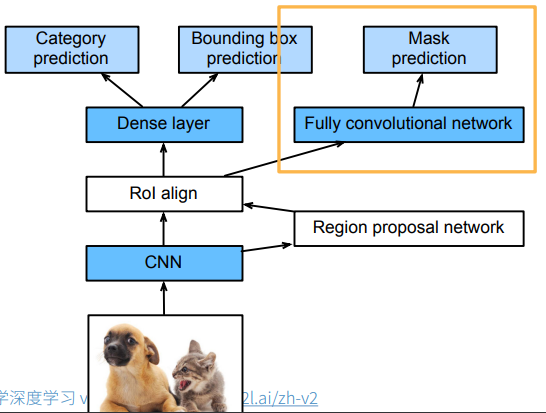
RPN中对CNN提取出来的feature map做了一次conv后，再次得到一个feature map，然后管你用启发式还是暴力生成锚框，反正接下来你得到了Anchor box一堆，然后再训练一个模型，预测你的每个锚框的锚框信息：是否框住了某个类别+离ground truth的偏移值，然后以框住了某个类别的锚框做NMS得到最终你要的锚框，离开RPN就是了，剩下的流程就是和Fast RCNN一样啦，映射锚框……ROI pooling……



类似这种Faster RCNN的，先做一个糙一点的预测再做一个精准预测的一般被称作2-stage模型，即两步预测。

还有一种针对性能而非针对速度的改进，叫做Mask RCNN，我还是针对下面这种图讲解一下，它基本上是由Faster RCNN改进而来的，区别主要在：1.该网络需要ground truth是mask像素级分类而非简单anchor锚框分类的数据集；2.在ROL后面加了一个FCN卷积网络，卷积出来的结果做预测，预测原图mask/每个像素的分类；3.预测mask并不是真的要它这个结果，因为这里通过CNN和ROI已经丢失掉很多原图信息了，不可能再得到精确的mask预测，这里预测mask只是为了让CNN和RPN表现得更好——**这是一种重要的思想，有时候对一个模型的某个功能的训练只是为了使得它在其他功能处表现得更好**。4.ROI pooling改成了ROI align，因为原来的pooling对锚框做n\*m分割，像素是离散的值，例如对3\*3的图像做2\*2分割，分割出来的区域必然出现像素偏移，也就是说4个分割出来的区域的像素来源数量不均，分别是4,2,2,1，这对于mask prediction不好，因为mask prediction是对像素级别做预测，你这里产生像素偏移了，会影响mask预测结果，所以我们这里把pooling改成align pooling，对3\*3的图像区域，对半切割它的第2列和第2行，切割出来的每小部分离散单像素按照一定权重分配给2\*2的4个子区域，即得到2\*2的ROI align pooling的结果。

很有意思，**池化本来是想要通过模糊原图像素的方式来消除后层神经网络对像素抖动的敏感性，而这里用于要做精确的mask prediction，人家后层神经网络就需要这种对像素抖动的敏感性**，所以你这里的池化不能像一般池化持有很糙的模糊掉原图的思想，而是在模糊的过程中也要尽量保持像素间的相对关系，支持后层对像素抖动的敏感性。



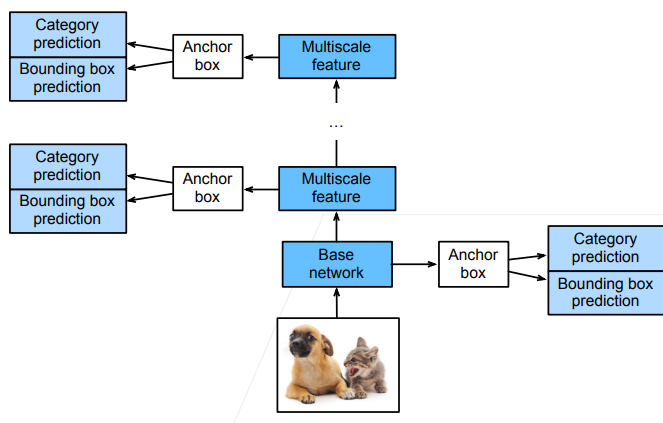
锚框目标检测实例二：SSD

Single Shot Detection。可以译作单发检测，意思是最常用的RCNN都是2-stage，即一个主网络+一个RPN，麻烦，我SSD一发入魂。它的基本思想是通过多层的分辨率不同的网络叠加最终得到良好的效果。

SSD网络的整体架构分为多步stage(这里的stage指的是我们常常描述CNN用的那种使得feature map长宽减半的stage)，每个stage上对于输入的feature map的锚框生成算法都是每个像素为中心生成多个锚框(但在实操时，你其实多半会采用均匀采样的方式在feature map的部分像素上生成锚框)，但每个stage生成的锚框大小都差不多，这就导致了：每个stage输入进来的feature map大小不一，但锚框大小又差不多，于是下面的stage锚框框住的区域较小，适合检测小尺度的物体；上面stage锚框框住的区域较大，适合检测大尺度的物体。

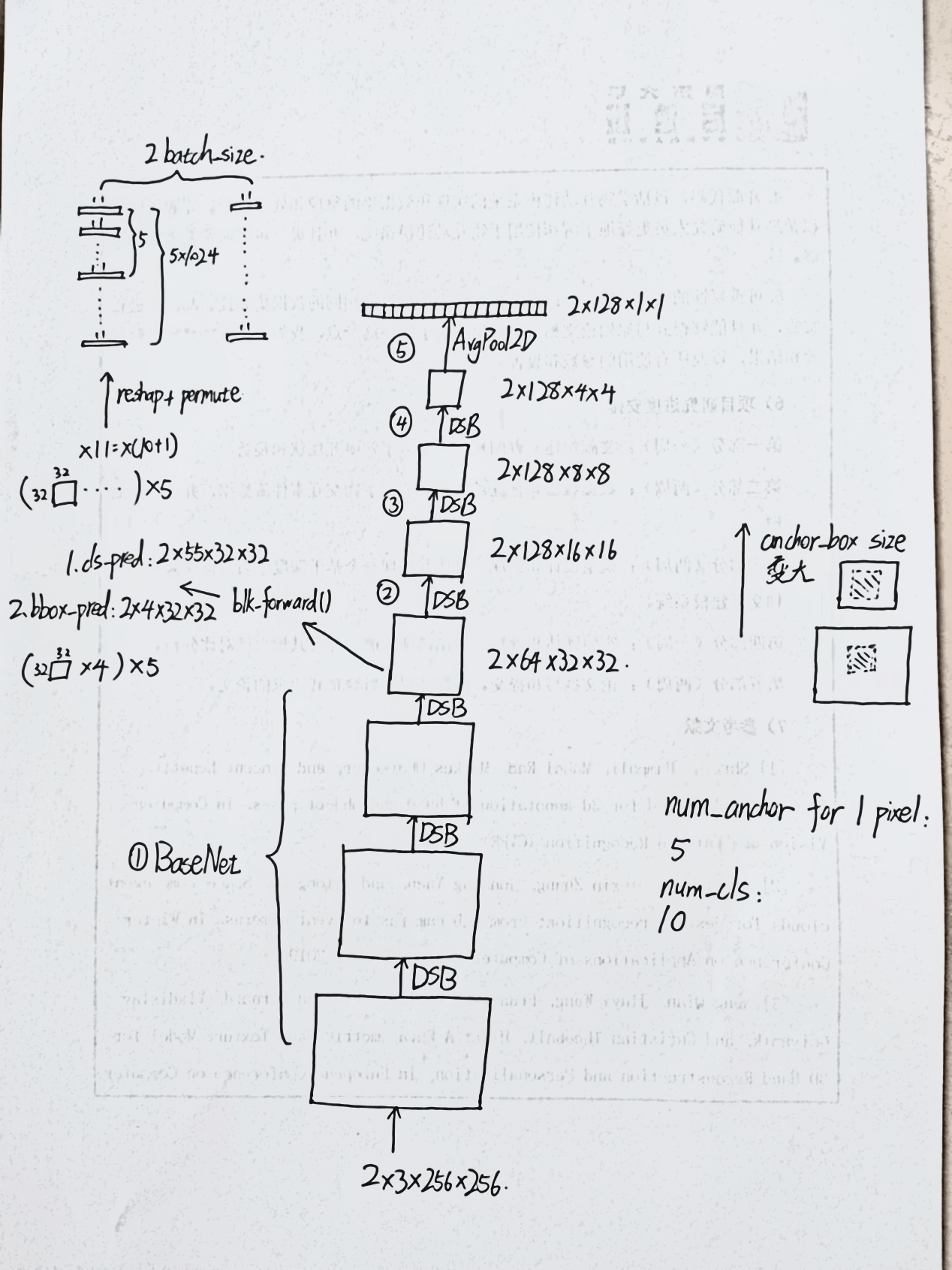
总结起来SSD3大思想就是：1.单神经网络检测模型；2.每个像素为中心产生锚框；3.多个stage的输出上进行多尺度的检测。

【由于是分层多尺度检测，假如你图像中的物体特别小的话，就把输入图片的size做大一点吧】【SSD，还有很多锚框目标检测算法的finetune有两种做法，一是用图片分类的pretrained\_net来加上cls\_pred和bbox\_pred，直接利用图片分类的卷积层，而是把另一个ssd拿过来，但cls\_pred和bbox\_pred要大改，因为num\_anchor和num\_class都不一样了】



YOLO(you only look(live) once)是比SSD效果好，且更快，因为SSD中每个像素生成若干锚框，必然导致锚框大量重叠，浪费掉了很多计算，而YOLO将图片均匀分为S\*S个锚框，每个锚框再预测多个ground truth边缘框。

我已看完SSD pipeline李沐代码实现，接下来我将就自己对其代码的理解讲解一下SSD的详细结构，帮助你自己手写代码。有些细节处可能和李沐的代码不一样，但道理相通。下面是一张基本的SSD结构图：



它有5个stage(不是使长宽减半的stage)，其中第一个stage称作basenet，就是从feature map刚输入到第一次输出的feature map被用于做锚框信息预测的stage它就是普通的卷积，由3个DSB构成；第2、3、4个stage都是DSB，DSB就是down\_sample\_block的缩写，它就是一个普通的conv，第5个stage是一个池化层，将第4stage的output flatten成一个一维张量。

锚框是事先就生成好的，而不是在模型推理过程中根据feature map的大小生成好的，也就是说，你在构造模型的时候就该知道每一层的feature map长宽是多少。生成锚框的算法有大小形状两个重要参数，size数组和ratio数组，s是0~1之间，ratio反映长宽比例，它们都无量纲。对于每个stage输出的feature map，都会有不同的size和ratio参数，ratio参数其实可以不同stage都一样，但size数组从下到上就要逐渐增大，这才符合SSD多尺度目标检测的原则。注意对于每个feature map来说，最终根据s和r生成的每个锚框有4个参数，分别是左上和右下角的坐标。

每个stage输出的feature map要加上两个CNN，一个是class\_pred，一个是bbox\_pred，分别预测每个锚框的类别信息和到真实框的偏移量信息。它们输入都是feature map，输出的话，cls\_pred的通道数应该是“每个像素生成的锚框数”\*(“预测的总类别数”+1)，”+1”是代表对背景框的预测；bbox\_pred的通道数应该是4，代表到真实框的4个坐标的偏移量。要注意cls\_pred和bbox\_pred输出的也是feature map，大小必须和输入的feature mao一样！为什么要怎么干？拿第1个stage输出的feature map的cls\_pred举例，它输出了5\*(10+1)=num\_anchor\*(num\_cls+1)个feature map，num\_anchor是每个像素要生成的锚框数量，feature map32\*32，32\*32是输入的feature map的像素个数，也就是说，cls\_pred要针对32\*32个像素，每个像素生成5个锚框，每个锚框预测11个类别的评估分数，所以一共应该有32\*32\*55=32\*32\*5\*11个单个输出数据，这32\*32\*5\*11个单个输出数据如何表示？如何表示决定了其该如何从输入的feature map中计算出来，我们采用了CNN中最方便的输入输出数据表示方式，即是也用feature map表示，32\*32天然可以对应于输出的feature map的大小，55可以分为5组，代表该输入的feature mao根据s和r计算出来的应该生成的5类锚框(每个像素生成的5个不同大小形状的锚框)，每组有11个feature map，也就代表了32\*32个像素预测的5种中的一种锚框所预测的11个类别值，5个分组也就代表了32\*32个像素预测的5种锚框的没种11个类别预测值。bbox\_pred的输出也是一样的道理，只不过它的每个像素预测的不是11个类别，而是4个偏移量。那么你就要问了，上述cls\_pre和bbox\_pred好像对于一个feature map预测出的类别信息和偏移量信息跟生成的anchor没有任何关系？也就是说，它们作为卷积层，输出的整个feature map看到的视野是整个输入的featrue map(或者严格细致一点来讲，每个输出的feature map的一个像素代表的是一个锚框预测的某一个类别评估分数，或者某一个偏移量，但它们的预测数据来源所覆盖的输入feature map的范围是blk\_forward卷积核的大小，或许是3\*3，但肯定不是对应输入像素的对应锚框大小)，而不是5组输出分别对应的锚框的视野，跟生成的预先anchor没有如何关系，这是怎么回事呢？其实啊，我们在infer时确实没有让预测的class和bbox与看输入的feature map的锚框视野，而是稀里糊涂地去卷积就完了——但我们**在train时会用根据预生成的anchor所打出来的groud truth去训练CNN的参数，强迫cls\_pred和bbox\_pred去看对应锚框的视野**。那么接上另一个问题，就算是train时强迫cls/bbox\_pred CNN去看对应锚框的视野，但在卷积时，输出的feature map的单个值也绝对绝对只能覆盖掉3\*3的范围(假设卷积核大小是3\*3)，不可能覆盖掉锚框的视野啊？这里，就体现出BaseNet先用DSB做了多层卷积的重要性了，你cls/bbos\_pred名义上是说看的是每个stage输出的feature map的锚框视野，但说到底是要看最开始输入的image的锚框视野(虽然没有针对它生成锚框)，所以BaseNet的3个DSB已经把image的像素信息都给你混匀了，你kernel\_size=3\*3，但看到的仍然是更大的视野——针对每个stage输出的feature map生成的锚框的本质其实是多尺度地看image中的信息。

于是上述还引出一个问题，给你一个数据集，里面的每张图片打好了bounding box的位置和class，bounding box的数量不固定，位置不固定，class不固定——所以，你就要根据每个像素生成的大量锚框运用前景框筛选算法+偏移量计算算法，给每个锚框打上锚框信息，这才是你最终训练时用的groud truth的数值——预先生成的anchor最终也就在这里起到了训练作用当然。你模型训练完做推理时也要用同样的size和ratio生成大量锚框，比较cls\_pred和bbox\_pred生成的只是锚框类别和偏移量，你还要根据类别筛选出锚框，用偏移量+锚框计算出真实框，在用nms得到最终的真实框。**你训练模型时既然是用了这一组size和ratio生成的anchor，那你完了预测时，也要用同一组anchor——这也就是为什么基于锚框的目标检测算法的模型微调不容易的原因，一个pretrained\_net基于的锚框都跟你的不一样，所以微调还不如干脆用图像分类CNN中的卷积层拿来微调，甚至pretrained\_SSD的主干上的BaseNet和DSB都不适合拿来微调，因为它们提取特征是为了让后面的cls/bbox\_pred看到更大的锚框视野而非kernel\_size，pretrained\_SSD铺开的锚框视野是每个像素10个锚框，锚框形状稀奇古怪，但假如你想要的是每个像素3个锚框呢？对吧。**

**所以，SSD模型超参数不仅仅是kernel\_size+padding+stride+层数+输入输出通道数，还有1.锚框是针对每个像素生成还是等距采样，2.size，3.ratio。你微调，大不了迁移过来改一下层数，其他的你什么也改不了，关键是后3种超参数还对SSD十分十分重要，你怎么兼容pretrained-traing？不像kernel\_size和输入输出通道数这些，前者反正是提取特征，后者也能用数据预处理解决pretrained-training不兼容问题。**

说到groud truth的数值，还要说说它的表示。对于class，每个像素的每个锚框会有一个真实值，而不是11个评估分数，所以，class\_label应该是长为sum\_anchor\*1的一维向量(sum\_anchor=32\*32\*5+16\*16\*5+……)，bbox\_label应该是(sum\_anchor \*4)\*1的一维向量(就不用sum\_anchor\*4的二维矩阵了，虽然语义上好理解，但是这种数据表示的话，pytorch提供的损失函数不好算它)。所以，你就要对你cls\_pred和bbox\_pred预测的55\*32\*32和4\*32\*32(以第1个stage为例而已是32\*32)做.permute函数维度变换，变换成(5\*11)\*1024🡪11\*5120🡪5120\*11(把55分割成5\*11没问题的，训练时会强迫你5个里面每个11都是预测的一个锚框的11个类别的评估分数，让**5\*11的分割不是胡乱瞎分割乃由train过程来诱导实现**。分割的过程由.reshape函数实现)，其他stage上的clas\_pred结果照例，最终把全部stage上的cls\_pred预测值都cat成sum\_anchor\*11，与class\_label做交叉熵，得出cls\_pred的loss1。bbox\_pred则是将4\*32\*32直接转换成4096\*1，与bbox\_label求出loss2。加权求和得到最终的loss。

你训练时算损失函数是把cls\_pred和bbox\_pred的结果做了reshape、permute和concat(concat的是不同stage经reshape和permute后的结果)，但是你最终模型在预测用是，的出来的结果还是不同stage feature\_map的55\*w\*h与4\*w\*h，你应该写一个函数predict对它们做进一步的提取得到最终的bounding box的类别与坐标。

要注意在对bbox\_pred和bbox\_label求loss2的时候，要对两个数据都乘以一个bbox\_mask(里面的值只有0和1)，因为cls\_pred预测出来的类别最终决定了选取哪个锚框作为前景框，只有前景框的bbox\_pred才是有效的，背景框的bbox\_pred与bbox\_label全部置为0，不参与有效的损失函数计算。但是你是在训练啊，你cls\_pred根本没训练出来，你怎么知道那些锚框是前景框背景框那些bbox\_pred值是有效的？也就是说，你bbox\_mask是怎么确定下来哪些值为0哪些为1的？对了，我们不是有groud truth吗，你就把groud truth里的前景框对应的bbox\_mask的值置为1就行了，就用真实值对应的bbox去训练bbox\_pred就行了，不用管训练时的cls\_pred。我打比方：对于锚框a，你最后训练出来，cls\_pred认为它类m的前景框，但groud truth不是，所以bbox\_pred训练用bbox\_mask把对应锚框的bbox\_pred预测值置为了0，没参与训练。所以你对a上偏移，得到所谓的真实框，但这个真实框的框住的是一个乱七八糟的东西，对它框住的内容做softmax分类，对m的置信度极低，nms一上去就先预筛选把它给剔除掉了。所以，你训练时，cls\_pred不影响bbox\_pred的label又怎样？So What？

我还想补充两点：

1.我们前面说SSD是多尺度多stage生成锚框的目标检测，我们又说过目标检测中每个锚框是一个样本。我们这里的例子中的样本是一张image进来后，每个stage所有的锚框，毕竟你最终训练时也是将所有锚框的预测值concat到一起来训练的。也就是说SSD最终的输出是每个stage的所有锚框在一块儿，而不是选出几个stage的锚框作为输出。**其实我想到一个idea**，就是我下面的size比较小嘛，我检测电线杆，我上面的size检测大象，也挺好，反正你的cls\_label是由groud truth和anchor计算出来的，你每一层都会由不同的size和ratio生成不同的anchor，不同stage的anchor根据本stage负责检测大象还是电线杆去针对性地打上label。当然你这样干的话，stage电线杆cls\_pred可能是(5\*(3+1))\*32\*32，stage大象cls\_pred可能是(5\*(5+1))\*16\*16，类别变了嘛，类别数也可能变——于是在计算损失函数时你就要解决一个问题：原本要把cls\_pred的结果reshape+permute+concat成sum\_anchor\*4，这下可好，你一个stage可以拼成sum\_anchor\*4，另一个只能拼成sum\_anchor\*6，你怎么把不同stage再拼在一起算loss？可能只能不同stage分别算loss然后加起来了。

2.每个stage可能针对每个像素的锚框生成的个数都不一样，例如stage1 cls\_pred出来是(5\*11)\*32\*32，stage2则是(3\*11)\*16\*16，但这不影响，在算损失函数时你保证11==11就能让不同stage的全部锚框拼接起来就行。

3.往往为了简化计算，我们对一张feature\_map可能不是每个像素都生成锚框，可能给一张32\*32大小的图片，我们告诉算法说，诶，我这张图片的大小是20\*20，那就让算法给你生成20\*20\*num\_anchor个锚框吧。于是，cls\_pred出的结果可能就不是55\*32\*32了，而是55\*20\*20，但由于都是预测11个分类，所以55==55🡪11==11，最后训练时算loss能reshape+permute+concat成标准的sum\_anchor\*11的形式。

4.综上，cls\_pred和bbox\_pred的数据维度表示影响的只是损失函数的计算形式，其实只要你仅仅保证每个stage的所有锚框预测的类别数是一样的，那就没有任何问题，最终算损失函数不同stage全部锚框预测值拼在一起算，不会出现拼接时维度冲突的问题。

5.注意stage5的一维向量，似乎没办法用cls\_pred和bbox\_pred做卷积了，但其实不然，你用padding=1、kernel\_size=3\*3照样可以做卷积。

语义分割：

对于语义分割数据集而言，有feature和label，feature是模糊点的.jpg都没有问题，label必须是无损模糊少的至少.png。一份语义分割样本，feature和label的大小是一样的，若是你想改变图像的大小，你只能crop不能reshape，因为reshape涉及像素间的插值，两个像素，背景是黑色，mask是红色，你怎么插值，你难道要把它插成粉红色吗？

语义分割使用转置卷积神经网络，早期的网络叫做FCN(全连接卷积神经网络，我也不知道它跟全连接层有什么关系)，FCN核心思想就是先用CNN对input抽取特征，size压缩变小，然后加上转置卷积层把feature map放大到原来的input大小，此时output有n个channel，像素在每个channel的值代表一个类别的评估分数，损失函数宜用交叉熵损失函数。用于语义分割的网络必须满足像素到像素的映射，假如我们使用样式迁移的思想来做语义分割，原图是feature，样式图是label，则不能用于语义分割，因为它不满足像素级别的映射，只有转置卷积满足像素到像素的映射。

转置卷积：

首先明确：转置卷积不是对卷积的还原，虽然你可以说它是还原了size，但绝对不是为了还原value。

我认为转置卷积不如叫做逆卷积(事实上转置卷积在DL中也常被叫作反卷积，但反卷积其实是数学上的概念，它同时逆变换了size和value)，首先你通过矩阵乘法的角度来看卷积，早期为了兼顾计算资源，把卷积运算改造成矩阵运算，即是将输入输出都拉成向量，Ym\*1=Vm\*nXn\*1，V是卷积核，其中输出Y的m必然大于等于输入n。但转置卷积则不一样，它是Yn\*1=Vn\*mXm\*1，输出m小于等于输入n，有人认为就说Vn\*m相当于把Vm\*n转置了，但我认为不如说是把Vm\*n乘到左边去，作为一个逆运算，所以不如叫逆卷积。

而且从非矩阵运算的原始直观卷积运算来看，转置卷积也更像是对一般卷积的一次你运算，你画个图试试，比较一下二者的stride的形式，是不是就很好理解。

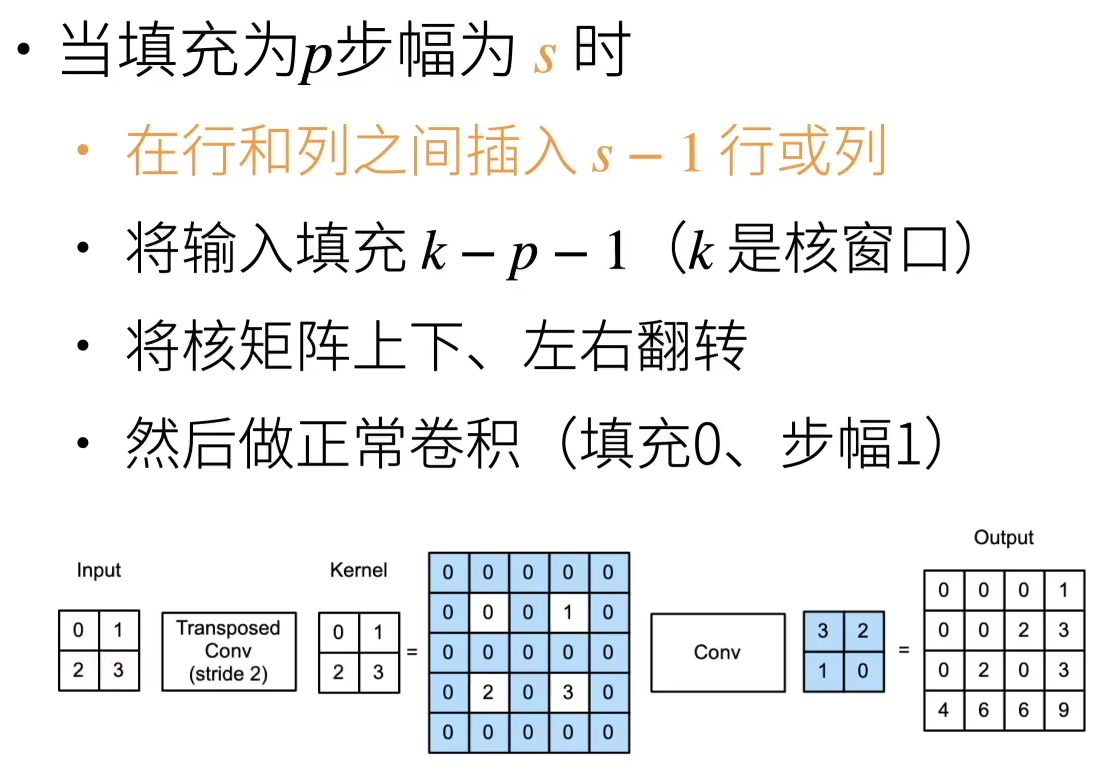
那么问题来了，computer scientist为什么一开始就是要把反卷积叫做“转置卷积”呢？原因如是：我们说了卷积操作可以用矩阵和向量的乘法得到向量来表示，设定卷积核为K，当它用来做正常卷积时(我们默认padding=0,stride=1)，它的矩阵形式是W，那么，当它用来做转置卷积时(此时输入输出向量的大小刚好和正常卷积时相反)，它的矩阵形式是WT，而不是W-1，所以叫装置卷积——所以我们还知道了，**当我们用正常卷积核K把向量X变成向量Y时，再用同样的卷积核对做转置卷积，并不能把向量Y通过逆阵逆变换成X，所以用K做的反卷积叫做“转置卷积”而非“逆卷积”**。所以我们有了一个**转置卷积核对应的矩阵的计算方式(这个计算方式是我想出来的，现实中基本不会用到，但在下面我对自编码和自解码器理解中能起到帮助)**：当给你一个转置卷积核时，1.你肯定(默认padding=0,stride=1)能得到其输入feature map Y和输出feature map X的size(这里我们只关心size，好计算W的size应该是怎样，才好在该W size中填入K的元素值。WT在W非正则情况下是不可能把X再还原为Y的)；2.然后你先求它对X做正常卷积所对应的矩阵W，好求；3.接着把W转置，就是X=WTY中转置卷积对应的卷积核WT了。

要注意对于一个卷积核K来说，若是其**正常卷积矩阵乘法表示有Y=W\*X，那么K做转置卷积必然有Z=WTX，Z!=Y，Z!=Y，Z!=Y！，除非W是正则的**。

你还要注意转置卷积的padding，当ConvTranspose2d函数的padding参数为0，stride参数1，kernel\_size=2\*2时，且input size=2\*2时，output会是3\*3（你可以理解为3\*3的output被2\*2正常卷积后会成为2\*2的input），但若是把padding设为1，则output会变成1\*1，则是为什么呢？明明正常卷积的话，这种情况output应该会是5\*5啊——错了，你还是要按照output正常卷积为input的思路去理解，想象怎样一个output在加了padding之后会被2\*2的正常卷积kernel卷成2\*2？那必然是1\*1的卷积啊——后面再实际计算过程中，你遇到ConvTranspose2d padding=n，直接把转置卷积的output size外围去掉一圈padding就行了。

如果说卷积的输出是模式匹配产生的响应，那么转置卷积的输出就是这些响应在空间上更细粒度的分配/具象表达。这种再分配/表达不是按某种规则恢复出来的，而是学习得到的。转置卷积也可以看做一种上采样的方式。**在转置卷积的时候，kernel\_size最好还是大于stride，这样能使得每从输入卷一个像素得到的map之间能够有一定的重合**——为什么要这么做？有一种不太恰当的理解(毕竟转置卷积逆变换的只不过是size)：假如我们把转置卷积的inputY看作正常卷积的输出，output X看作正常卷积的输入，则对于正常卷积而言，X的1个像素为Y中多个像素提供了信息，那么当转置卷积要从Y逆变换为X时，就要对X中一个像素从Y中多个像素抽取信息来供养，则必须使得转置卷积核对Y中相邻两个像素卷积出来的map映射到X上有所重叠，所以应该stride<kernel\_size。

其实你早就理解了转置卷积的计算方式，不就是卷积核对每个像素做计算吗，然后把计算结果按照步长来堆叠吗，然后把输出去掉外围一圈padding吗，但是这对于计算机而言，代码实现可能会变难，所以有更简便通用虽然人类直观不太好理解的方式：



所以我觉得，你对一个feature map做卷积后再做转置卷积(两个卷积核不一样)，相当于自编码器+自解码器。而且显而易见，你要是想让自编码器的结果完美还原为自编码器的输入，则自解码器的卷积核对应的矩阵必然得是自编码其卷积核对应的矩阵的逆阵(应该是伪逆)，所以自解码器的卷积核对应矩阵必然会在训练过程中无限逼近自编码器的逆阵，若是此时恰好自编码器的卷积核对应的矩阵是正则的(逆阵==转置)，则自解码器的卷积核对应的矩阵将既是其逆阵，又是其转置——结合我们上面讲的为什么转置卷积会叫做“转置”卷积即转置卷积核对应的矩阵的计算方式，我们可以知道转置卷积核必然与自编码器卷积核完全相同——因为如果一个**卷积核K的对应矩阵W**能将Xsize的向量转化为Ysize的向量，那么其转置WT必然能将Ysize的向量转化为Xsize的向量(**我们这里只关心size**)，且**WT对应的转置卷积核必然也是K**！

我想，我们最后在**不断训练这个自编码器+自解码器的卷积核转置卷积核时，训练的结果应该也是让1.自编码器卷积核对应的乘法矩阵越来越正则；2.转置卷积核则相应地和卷积核越来越一致**。

转置卷积核初始化：

转置卷积核像一个上采样，或者我们可以说，它把一个图像放大了，把图像放大，我们常常使用双线性插值算法。在实际使用转置卷积层时，为了缩短训练时间，我们常常会预先初始化转置卷积层，我们让它的初始值就表现得像一个专门用于双线性插值的。为此我们会写一个函数，专门用于初始化转置卷积核使其具备双线性插值的功能，这个函数具体怎么实现就不细说了。

样式迁移：

样式迁移是个很简单的东西，按照李沐讲的代码pipeline，首先你要有一张原始图片，一张样式图片，一张待训练的目标图片——没错，你训练的不是网络的权重，你训练的是目标图片的像素值，目标图片的初始像素值可以随机生成，也可以直接取原始图片像素值。

接着你找来一个resnet，这个resnet是现成的pretrain好了的，它对原始图片和样式图片提取特征，你把它对原始图片提取到的某些卷积层的feature map作为目标图片的content label，你把它对样式图片提取到的某些卷积层的feature map作为目标图片的style label，然后在把这个resnet作用到目标图片上，提取出对应conv层输出的content feature和style feature，接着你就可以计算出content loss和style loss了。

关于损失函数，content loss好说，直接比较每个像素的相似性即可，L2范数。对于style loss，显然你比较每个像素相似性并不合适，你应该比较style feature和style label的统计信息间的相似性，统计信息常用gram matrix表示，计算出feature和label的gram matrix，如何计算很简单，不讲。损失函数还应该包含第3个部分，就是平滑损失，反映了目标图片平滑程度，也就是不要噪点，计算平滑损失直接在目标图片上计算即可，不用resnet提取feature map。总损失函数包含3个部分就是了。

上述是原始的样式迁移算法，你每要生成一张目标图片，都要重新训练一次目标图片，这不好，所以现在的样式迁移算法都会额外训练一个CNN来infer出目标图片了，具体实现不讲。

自编码器(AE):

自编码器不是NLP中的编码器-解码器架构。

AE是一个无监督学习模型，简单的AE只有3层，其中隐藏层的大小往往小于输入和输出层(输入层大小等于输出层)，AE起到的是一个数据压缩，或者说，提取输入关键特征的作用，提取出的关键特征被隐藏层输出，然后给了输出层。所以在实际使用中，我们训练好了一个自编码器，往往把它的输出层去掉，输出层只是单纯用来无监督训练用的，我们把自编码器的输入层+隐藏层保留，给其他神经网络使用。

很多AE的隐藏层大小都小于输入输出层，但稀疏自编码器SAE是个例外，它的隐藏层大小大于输入输出层，为什么这样干？不是要数据压缩/提取关键特征吗？这样干明显不合理。所以这样干的原因是，人们想要模拟大脑机理，人们发现大脑在收到外界刺激后，大多数神经元是被抑制状态，不会传递出激活的信号——于是人们就想让SAE的隐藏层大一点，模拟大脑容量，但在训练过程中按照一定法则诱导隐藏层大多数神经元的活性处于抑制状态，训练出来的模型参数也就**模拟了大脑机理**，使得输出的张量大多数值为抑制/非激活状态，也就是0。所以叫“稀疏”自编码器。而由于编码结果虽然比输入的张量大，但**由于其稀疏，所以起到了和AE一样的数据压缩的作用**。

是怎样的法则使得SAE训练时能诱导大多数神经元活性被抑制，输出接近0的数值？这需要我们对神经元激活/抑制做进一步理解——神经元的活性的评判规则是它对于不同输入产生的输出值的分布，**神经元输出值越容易是0，则活性越低，越容易不是0，则活性越高。稀疏性就是指神经网络所有神经元的活性**。举例就是：当神经元的的输出接近激活函数上限时(例如对于Sigmoid为1)称该神经元状态为激活,反之当神经元的输出接近激活函数的下限时称该神经元的状态为抑制，那么当某个约束或规则使得神经网络中大部分的神经元的状态为抑制时(也就是我们想要的SAE的效果)，称该约束为“稀疏性限制”，**稀疏性限制，是对神经网络模型参数的限制，使得神经网络稀疏性变强**。我们对SAE的**稀疏性限制的手段是：在训练SAE时，给损失函数加入一个惩罚项**，诱导模型参数达到一个使得神经元活性在大多数输入情况下被抑制的状态。这个惩罚项是一个函数，该函数的输入有两个：1.对**神经元活性的度量(反映神经元输出容易为0的程度**，由神经元输出值计算来)；2.神经元活性的理想期望。惩罚项计算出两个输入的偏差即可。惩罚项的细节涉及数学知识，这里不再赘述。

图神经网络：

即是输入时一张图，输出也是一张图。

现实中的几乎所有数据都可以抽象成图。

我们使用图神经网络首要解决输入输出的图的数据表示的问题，理想的图表示方法应该满足： 1.每个点/边/全局有一个属性向量来描述之。2.GNN更新图的每个属性向量，但图的形状，或者说是点和边的连接关系不会被改变(这一点是GNN的基本假设，叫做“对称性”，如同RNN基本假设是时间延续性，CNN基本假设是平移不变性+局部性)。

关于第2点，普通的邻接矩阵就绝对满足不了，因为你要是交换两个点之间的下标顺序，就会得到一个完全不同的邻接矩阵，GNN会识别成不同的图，这不合理。

为了满足上述两点，且保证存储图(往往邻接矩阵稀疏)的空间不会过于庞大，我们把图的数据结构划分为4块，即顶点列表、边列表、邻接列表、全局信息列表。邻接列表中每个元素是一个二元组，有两个序号，序号是对应顶点在在顶点列表中的下标，每个二元组在邻接列表中的下标与对应边在边列表中的下标一致。于是：当我们改变顶点在顶点列表中的下标顺序时，只需要修改邻接列表中每个二元组的元素值即可；当我们该表边在边列表中的下标顺序时，只需要修改邻接列表中每个二元组的下标顺序即可。我们还要注意的是：顶点列表、边列表、全局信息列表中每个元素存储的值是对应顶点、边、图的属性向量(不同中数据的属性向量的长度不一定相同)，所以，这些列表，本质是一个二维张量，但全局信息列表往往只有一个属性向量，表示本图的属性。

最基础最简单的GNN是，把点列表、边列表、全局列表(邻接列表往往不直接作为网络的输入，它只是起到确定并维系点和边关系的作用)分别输入各自的MLP，各自输出值，每个MLP互不干扰。这是最简单的。

但是这种最简单的GNN没有利用好图的空间信息。我们如何在GNN中利用图的空间信息？我们要把几个MLP整合起来，每个MLP每层layer处理各自列表前，各自列表的元素的属性向量会先受到其他列表的元素的属性向量的影响(带有这种性质的GNN又被称为GCN)，怎么个影响法呢？首先明确几个术语(这几个术语是我暂时总结的，可能是偏狭错漏的)，这几个术语描述的功能都是针对顶点/边/全局列表来操纵的，但却要利用邻接列表：

1. 汇聚。举例，当你不知道一个边的属性时，你把与该边相邻的点的属性加和起来，作为该边的属性向量。“汇聚”的定义比下面几种定义都要明确而可信，它强调弥补缺失的属性向量。
2. 信息传递。举例，当你不知道一个点的属性向量时，你就把与该点相邻的所有点的属性向量加和起来，作为该点的属性。
3. 更新。举例，你可能知道一个边的属性向量，但是你还是通过某种方式将与该边相邻的顶点的属性向量整合进该边的属性向量。

上述几种操作的本质都是：一个元素的属性向量通过某种方式(常常是简单加和，也可以是求平均、求max)被另外的元素(通常是多个元素，且可能来自被更新元素不同的列表)的属性向量更新(可以是叠加到原属性向量上，也可以是直接替换)。

要额外注意的是，我们上面解释的这几种术语(可能是我编造的，更新、汇聚、信息传递等名词我们常常混用，反正它们本质一样)，在GNN中用的最多的是“更新”。“更新”举的例子是相邻的点和边的信息相互更新，但实际上，若每层GNN的layer仅仅靠邻接列表表示的“相邻”关系来更新点和边的属性向量的话，未免有点局部了，我们每层GNN更新属性向量想考虑全局的图，如何考虑全局呢？注意我们上面的更新没有利用全局列表，现在用它：每层GNN分别处理各列表的元素的属性向量前，要对属性向量的值做更新，更新的边的数据来源是点和全局；更新点的数据来源是边和全局；同时我们还要用点和边的数据去更新全局——如此，便实现每层GNN了图的顶点和边列表的属性向量更新受到了全局的影响而非仅仅是相邻顶点和边了。

GNN和CNN有异曲同工之妙的地方在于，就算不传递全局列表的信息，只要你每层有对各列表属性向量的汇聚/更新，那么你的层数足够多的话，最后网络输出的值看到的数据就是整个图的数据，而非仅仅某个顶点与其相邻的顶点或边（这就跟卷积核的感受野一个道理，所以GNN的汇聚也被称作“图卷积”。其实当汇聚的方式是简单求和或平均时，这种图卷积也相当于对邻接矩阵做矩阵乘法，你自己想是不是这个道理）。那么，output的数据来源变多了，最后反向传播训练网络时的求导开销就会很大，所以我们在推理的时候不应该对每个列表的每个属性向量都做汇聚/更新，而是对各列表的部分元素做采样，采出来的属性向量做一下汇聚/更新即可。但令我疑惑的是，采样的方式李沐只讲了随机采样，而且随机采样也可以玩很花，例如随机游走、宽度遍历。汇聚时随机采样一些点做汇聚在训练时很合理，如同drop\_out一样，但在训练完推理时呢？也是每层GNN对各列表的属性向量随机汇聚吗？这恐怕不合理，我不懂。

图神经网络的门槛很高，它对超参数特别敏感，而且训练起来非常贵，而且很难做到批量训练，因为你一个batch的图要是长不同的形状，那么它们的各列表的长度就是不一样的，你要如何统一各张图的表示，让它们成批量地输入GNN呢？这很麻烦。

GNN，或者说GCN的基本推理过程，总结起来就是：纵向来看GCN可以看做3个分开又彼此相连(通过邻接列表辅助的汇聚操作确定它们的相连关系)的的MLP；横向来看每一层GCN先对上一层输出的各列表的属性向量做一次汇聚(这里说汇聚不一定准确，它强调弥补缺失的属性向量，应该还要加上更新，或者笼统来说，信息传递)，然后在上神经元做W\*X。

下面要开始讲NLP了。

序列模型：

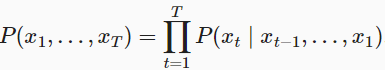
我觉得序列模型分两个task，一个task是没有给定输入，直接预测一段序列的出现概率(常常是基于统计的method)；一个task是给定一段输入序列，预测下一个或多个元素(常常是基于learning的method)。

序列模型就是要预测的数据xt与过去的数据xi相关，也就是说，你能用过去的数据xi预测当前的数据xt(这么说好像是接近于一个回归模型了，但实际上序列模型也可以做分类)。并且，理论上来说，当前数据xt应当与过去已知的全部数据xi相关。像这个样子的，要预测的label数据和用来预测的feature数据是同一类数据的模型，被称为“**自回归模型**”。

由于上面说的序列模型当前要预测的数据和过去所有数据相关的特性，在数学上，我们常常可以用条件概率来表示一个序列模型，即：y=p(xt|x1x2x3…xt-1)。我们说，我们会训练出一个模型f(x1x2x3…xt-1)= p(xt’|x1x2x3…xt-1)，式子中的xt’是一个包含xt所有可能取值的向量，f()也会输出一个概率向量，我们挑选其中概率最大的标量，即为最终预测的xt。这么看来，**序列模型是不是有点像一个softmax模型**？虽然叫做“**自回归模型**”，但是不是看上去**更像是一个分类模型**？

模型f()输出值是一个概率向量的特性，反映了“用条件概率表示序列模型”的本质。

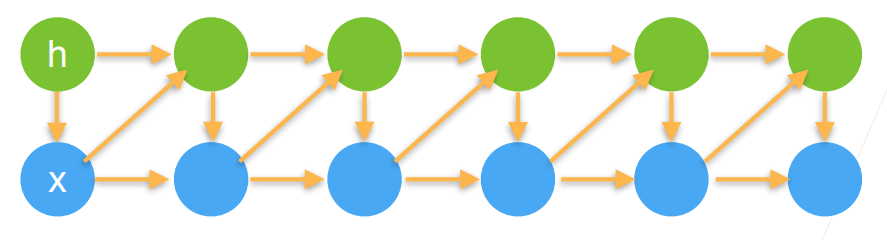
将上述序列模型泛化成更一般的情况，当我们要预测序列x1,x2,…,xt时，或者是说，我们在已知x1,x2,…,xt-1要预测xt时，实际上是要找到这样一个序列x1,x2,…,xt，使得下述P最大：



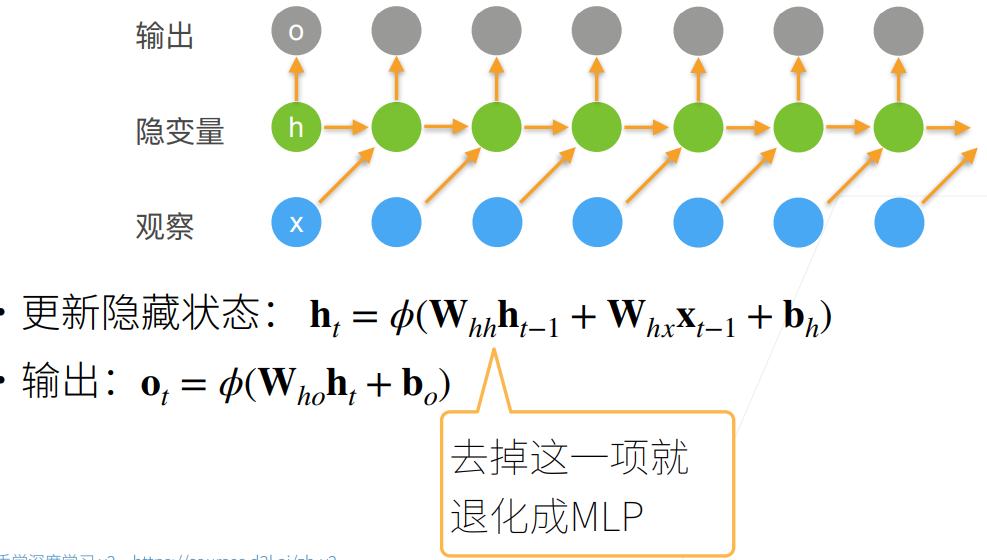
理想的由过去的全部数据预测现在的数据是否太复杂？其实，常用的序列模型有两种，它们都不是由过去全部数据预测现在的数据。

1.其中一种是基于马尔科夫假设的序列模型，它假设现在的数据xt只和过去tau个数据相关，语言模型中的n元语法就是一种典型的基于马尔科夫假设的序列模型，很简单，不基于learning，而是基于统计，由于它基于统计，所以它有更好的数学可解释性，能完美地用条件概率p(x1,x2,x3)=p(x1)\*p(x2|x1)\*p(x3|x1,x2)来表达和解释。当然也有基于learning的马尔科夫模型，它很无脑，它只能处理这样的情景：你已知(x1,x2,x3)，你要预测x4，你于是构建这样的model，输入x1,x2,x3，输出x4，数据集你也会搭建，但我感觉它和“条件概率”什么的都没啥关系了，可能一涉及到machine learning，就没什么数学可解释性了吧，你充其量就是说，让model去学习x1,x2,x3,x4之间的概率关系。虽然看上去没什么数学可解释性，但我们也常常用条件概率来表示它，p(x4|x1,x2,x3)表示x4和x1,x2,x3有关，然后model已知(x1,x2,x3)做分类预测，输出字典中不同可能取值的x4的softmax值。

2.还有一种更高级的就是潜(latent)变量模型(又称隐(hidden)变量)，它假设对于当前数据来说有一个隐变量ht(后序**我们常常用state来表示隐变量h，毕竟其实这里的“隐变量”并不是真正严格数学意义上的隐变量，而是hidden state**)，该潜变量由上一个数据xt-1和上一个隐变量ht-1预测而来，然后当前xt由当前隐变量ht预测而来，于是通过条件概率表示就有y=p(xt|ht)，但在实际应用中，我们预测xt有时不仅仅根据ht，还会根据xt-1，图形化表示就是：



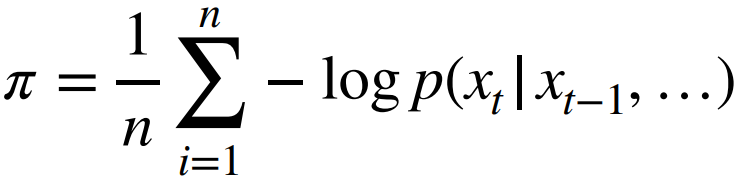
更准确一点的表示应该是(此时预测xt仅根据ht)：



事实上，隐变量模型是一种广泛的概念，它包括了所有涉及隐状态或潜在变量的模型，而且它更像是一种数学上的概念，上述隐变量模型其实应该是特指的RNN，RNN是一种可以被视作隐变量模型的神经网络模型，但它不能严格地被分为隐变量模型中的某一类，因为严格隐变量模型涉及到很多概率论上的东西，例如p(xt|ht)这种形式的条件概率计算，但RNN并没有严格的数学解释，所以**RNN**只能“被视作一种隐变量模型”。隐马尔科夫模型就能被算作严格的隐变量模型中的一种，它是**基于隐变量条件概率p(xt|ht)**和**马尔科夫过程(tau=1**的马尔科夫模型)的，我们这里不多讲。

(浅层)RNN：

RNN就是基于隐变量假设的。它的损失函数叫做“困惑度”，即

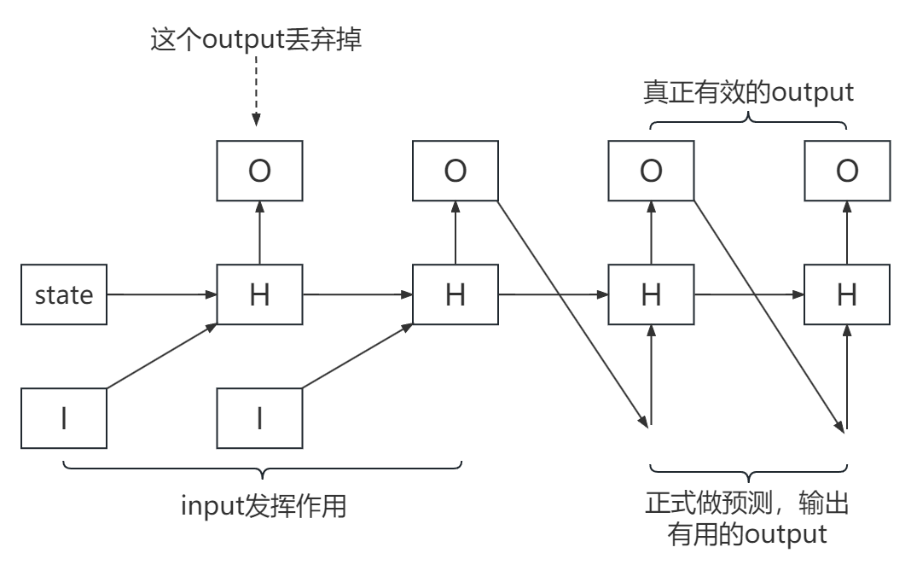


其中p(xt|xt-1,…)表示model输出的对应groud truth的softmax概率值，-log()是在计算交叉熵，n是指一共输出了长度为n的序列。由于历史原因，我们常常把最终的困惑度取作exp(pi)，大概是为了使pi更大一点。其实你能看到，exp(pi)的结果大致就等于p^-1，所以对于人的理解很直观的就是，当困惑度为1的时候，表示完美预测，当困惑度为2的时候，或许我们可以说，p=1/2，有两个预测值其实都差不多，都可以作为最终的预测值，当困惑度为3的时候，或许我们可以说，p=1/3，有三个预测值其实都差不多，都可以作为最终的预测值，困惑度inf就最差了，谁都有可能——这种想法很感性。

RNN是一种隐变量模型，它的网络结构是以隐变量为核心的。这个模型接受两个输入，一个输入时上一个时间步的隐变量h，一个输入时上一个时间步的input x，它的输出就是当前时间步的output o，o在计算的时候只看得到当前时间步的隐变量h。其实不论是隐变量h还是input x还是output o都往往是向量而非标量。

这个模型的参数有param=[Whh、Wxh、Whq、b\_h、b\_q]，Whh.size=num\_hidden\*num\_hidden，Wxh.size= num\_input\*num\_hidden，Whq.size=num\_hidden\*num\_output，b\_q=num\_output，b\_h=num\_hidden。

我们这里举一个具体的具有代表性的，典型的，很有普适性的RNN例子，它大致能做到这样的功能：给定任意时间步长任意批量大小的input，它都能指定生成任意时间步长的output。具体流程是：1.先自动生成一个初始化的隐变量h(称为state，state.size=batch\_size\*num\_hidden，要根据input的batch\_size自动调整state.size。state的初值在写代码时就给RNN模型固定死了，通常取全为0)；2.然后先用这些input更新隐变量h，计算h时用的损失函数通常是tanh，所以h的取值往往是(-1,1)，我也不知道为什么，可能是为了数值稳定性和减小模型容量提高泛化性吧。在更新的过程中的output扔掉不管，除了最后一个Input时间步输出的output on；3.在h更新完成后拿新的h和最后一个input时间步的输出on去预测指定时间步长的output，预测方法是拿上一个时间步的output和h作为当前时间步的RNN的输入，即得到当前时间步的输出o，这是一个自回归的推理过程，自回归体现在上一个时间步的output是当前时间步的input。然后进入下一个时间步，继续上述过程；4.再次强调，输入input主要是用来更新隐变量h，在真正预测想要的输出output时，你从输入input推理过程中索取的，只有**更新完成的隐变量h**和**最后一个时间步的输出on**。



但其实，不同任务下的不同结构的RNN的很多具体实现细节可以不同，比如说，我一定要保留前期feature输入过程中最后一个时间步得到的output吗？一定要用它作为正式做预测时用的第一个input吗？不一定吧，比如在很多做文本翻译的RNN中，feature输入完后，正式做预测，**第一个input是人为设定的vocab.token2item(“<bos>”)**，即一个代表begin-of-sentence的reserved\_token。**这种情况下**，我们就能说，**模型对于输入的feature推理得到的output可以全部弃之不用了**！很有意思的是，用pytorch搭建的类RNN网络时默认没有output输出的，它们默认自己的输出只有H，你要有output输出，就要自行在输出的最后一层H上搭建一个全连接层。那你说，既然如此，我在真正推理时后期用<bos>做起始input，那我在前期输入feature时，岂不是可以完全不搭建全连接层，反正由feature得来output也没有作用。这是不对的，因为你不论是train时，还是真实推理时前期输入feature，还是后期循环用上一个时间步的output做当前时间步的input，你用的都是同一个网络，你肯定是要始终保持全连接层的，output丢不丢掉那是另一回事。当然对于有些结构的RNN来说，它的前期feature输入部分和后期预测label部分不是同一个结构(但train部分肯定要用两部分结构的总和)，例如做文本翻译的编码器-解码器结构，这时候，前期输入feature部分你搭建的RNN确实可以干脆不要全连接层算ouput了，后期预测部分你用<bos>做起始时间步输入即可。

要注意的是，上述RNN在做完一次指定时间步长的infer之后，都会返回该指定时间步长的output以及刚被更新的隐变量h。为什么还要隐变量h呢？这是因为此时的隐变量h保留了已经保存了一开始的input的时序信息和之后所有自回归推理过程中output的时序信息，你可能会需要利用这个时序信息做下一轮指定时间步长的自回归推理。

进一步剖析上述的RNN的结构，你可以发现，假如去掉模型参数Whh，也就是不让模型从上一个时间步长得到隐变量h的信息了，也就是说模型做一次推理不再是自回归了，不再利用上一个时间步长的时序信息了，那么RNN就退化成一个双层感知机，就算你说你RNN做一次时间步的推理还是把上一个时间步的output做了当前时间步的input，但**RNN的精髓就在于隐变量的更新，这是其与MLP最大的不同**。

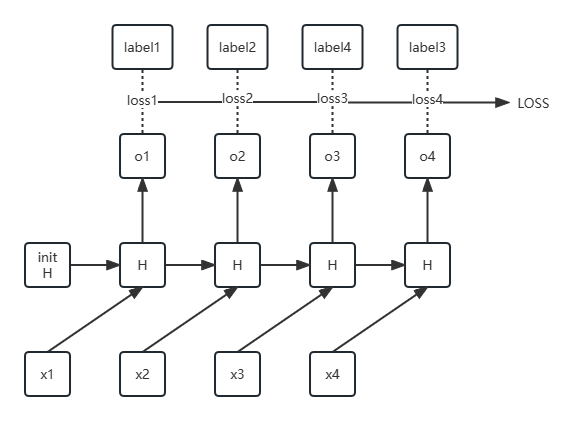
上述RNN结构可以用作一个灵活的语言模型，任务就是：给你一段话，你接上下一段话。

首先你要经过文本预处理得到vocab类和取样本迭代器Seqdataloader，训练之前，bacth\_size和时间步长T都要固定好，传给Seqdataloader。我们这里假设每一个token都是一个char而非一个word，所以算上空格和<unk>，一共有28种token。

然后我们在前面早已说过，语言模型归根结底是一个分类模型，所以对于输出而言，我们肯定是输出长为28的向量，代表28中token的概率的评估分数，你看它像不像个独热编码？这个时候对于每个输入输出的token而言，也很自然而然地做成一个长度为28的独热编码了。但假如有些情况下，我们以word而非char为token呢？这样独热编码也太长了吧，而且独热编码有个弊端，best和good的距离与best和bad的距离一样远。于是我们可以**在输入token独热编码之后加有一个embedding层**，embeding层也是一个可训练的层，它输出embedding编码，它的参数是一个vocab\_size\*num\_embedding的矩阵+bias，vocab\_size是独热编码的长度，num\_embedding是embedding编码的长度，这相当于一个全连接层，它能压缩原本onehot的维度，而且结果训练之后，它能学习到best编码的距离离good很近。embedding编码的输出再给出去做Wxh得到hidden state。

需要注意的是，训练时，每个epoch是Seqdataloader刚好从corpus中把全部样本取完为止，每个bacth是一个shape为batch\_size\*T的item编码序列——当然后面要做成独热编码。并且，在train时，有几个input token就该有几个output token，在infer时就不一定了，你只需要给一串任意长度的input，然后指定RNN输出任意长度的output，都行。

训练策略图形化为(图中label1=x2,label2=x3,label3=x4,label4=…)：



我们这里描述的RNN例子(也就是如上图的RNN)要实现的功能是

还要注意的一点是，在pytorch中，输出层要自己定义，我们这里把RNN当语言模型，输入输出的token都是长为28的向量，隐藏层的隐变量h不用说了也是向量，输出层被默认为了Linear层，但在pytorch中，你要自己定义输出层为Linear层，因为pytorch默认的RNN和GRU等都默认输出为state。

我再继续补充几点。

RNN实际上可以处理如何时序问题，但是它用来处理NLP的情况比较多，所以在很多情况下，当我谈起RNN时，常默认它是一个语言模型，处理文本。

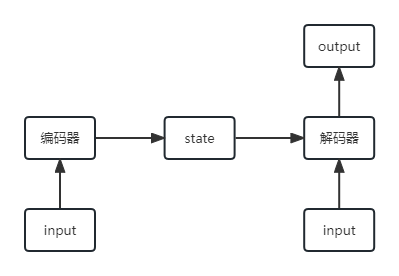
RNN可以粗略分为以下几类：seq2seq、seq2token、token2seq、token2token，对它们分类的标准是按照任务来分的，而非其详细结构或训练方式。其实，这种所谓“按照任务分类”的说法很模糊，你看，我上文对RNN的总体描述，其实是一种具有代表性的，而且是其他一切RNN架构基础的，一个典型实例。我说这个RNN给定任意指定的input和output的时间步长，都能输出相应结果，那么这个RNN，要是我指定不同长度的input和output数据，它是不是就能从上述4大分类中反复横跳？对吧，而且要是我指定时间步长都为1，那岂不就是一个MLP了。

那这么说，我们为什么不干脆就开发一种对输入输出时间步长具备普适性的RNN，或者干脆就不要上述4中分类，以普适的视角看待RNN呢？因为在特定情况下的特定任务中，我们开发的RNN确实是符合seq2seq、seq2token、token2seq、token2token中的某种类型的特征的，而且在这种任务，它是seq2seq就是seq2seq，不大可能因为任务的微小变动而变成token2token。

NLP中的自编码-自解码器架构：

我在简单介绍后，会举一个例子。

太简单，下图就是介绍：



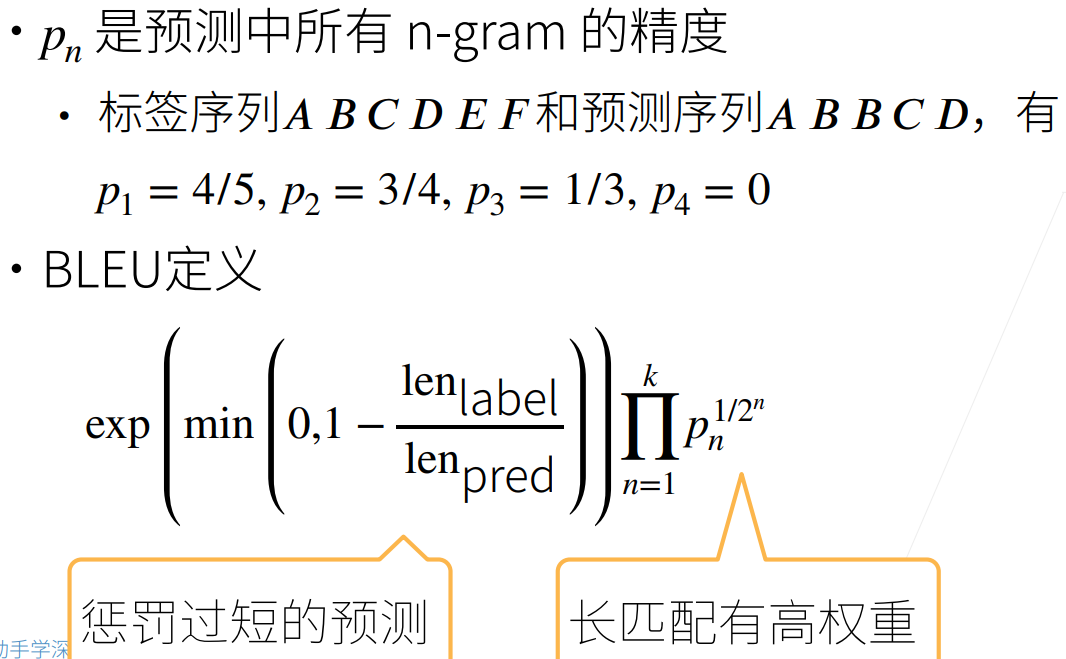
下面举一个特定任务的seq2seq型RNN的例子。

这是一个做文本翻译的RNN，它将指定长度num\_step的英文翻译成指定长度num\_step的法文。下图是该模型在valid/test上正式推理时的结构图(注意不是train时的结构图，train时不可能一遇到<eos>就break)，能看懂大部分这个RNN的结构，但还是有少部分需要额外注明的：

1. 编码器-解码器输入的token都是onehot编码，然后结果embedding层得到embedding编码，然后给出去算hidden state，而hidden state在解码器部分输出的却是onehot编码。
2. 编码器部分完全不用output，但它最后一个时间步得到的hidden state却要完整保留下来，整个H给解码器做初始时间步隐藏状态，hidden state的最后一层H[-1]作为重要的编码结果(事实上，我认为它就是这里就是编码器-解码器结构不同于其他RNN的本质地方，也是它叫做“编/解码”的根本原因。你说编码器不要output，那其他RNN在输入feature时也可以不要FCN算output啊，大不了把预测部分的初始时间步输入设为<bos>即可)，H[-1]会被拼接到解码器每个时间步的输入部分。我想，H[-1]就是编码器编的码，解码器要解的码。
3. 我们在训练的时候，feature是英文，label是法文，feature在编码器部分输入，label也要参加运算！很有意思吧？训练时，label要作为解码器的输入部分参与运算，输出的token序列再进一步和label做损失函数。这里的损失函数很有意思，下图是模型在valid/test上的结构图，遇到<eos>就break，但是在train时，可是要满打满算预测完num\_step个时间步的token的。假如token序列长度为num\_step的label句子的有效长度只有valid\_len(例如label为(Jie Chu Shen <eos> <pad> <pad> <pad>)的num\_step=7，valid\_len=4)，那我们预测出的长度为num\_step的token序列就只掐出来前valid\_len的部分(这一步叫mask)，和(Jie Chu Shen <eos>)做损失函数。

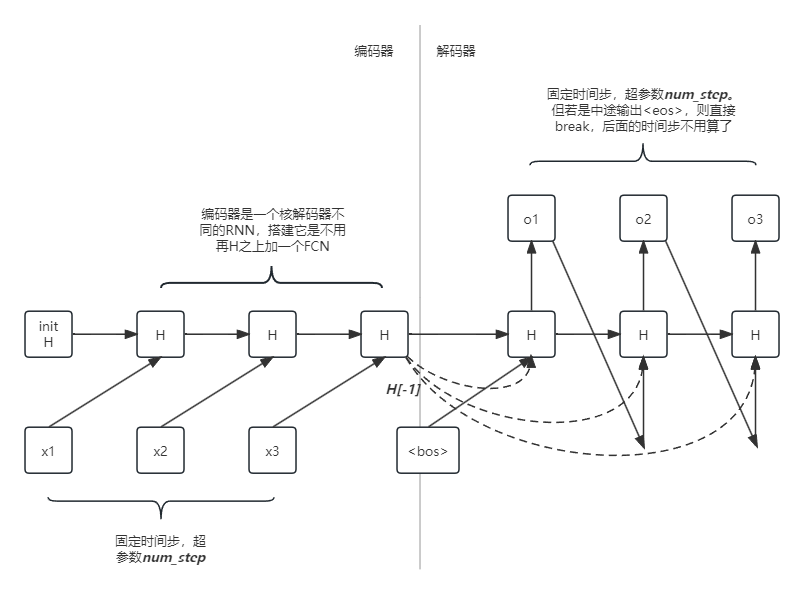
这里的valid\_len特指label的valid\_len，是在文本预处理中，得到每个feature-label样本token序列的过程算出来的，实际上在文本预处理是还应该算出feature的valid\_len，但是这里用不上，feature valid\_len只有在引入注意力机制时才用得上。这里直接把整个feature token序列包含<pad>一并交给编码器-解码器架构的RNN做训练了。

1. 训练时用的损失函数是一种对交叉熵的扩充，具体我忘了，自己查表。但是它却不太适合用来反映翻译的准确度，因为实际翻译时，谁给做mask？真正翻译时，我们没哟mask，我们预测出<eos>就break，要么取<eos>前的token序列做翻译结果，要么取最长长度num\_step做预测结果，然后用一种叫做BLEU的评判式子计算与label的匹配度，BLEU式子很看重预测的token序列中和label匹配到的较长的序列：



也就是说，我们train时表示误差用的是一种可微的损失函数，valid/test时表示误差用的是一种叫BLEU的评判标准。

1. 编码器-解码器架构远远可以比这里举的例子复杂，例如编码器你完全可以用双向RNN。
2. 还有不懂的，可以看“文本预处理”部分对该处RNN例子的提及。

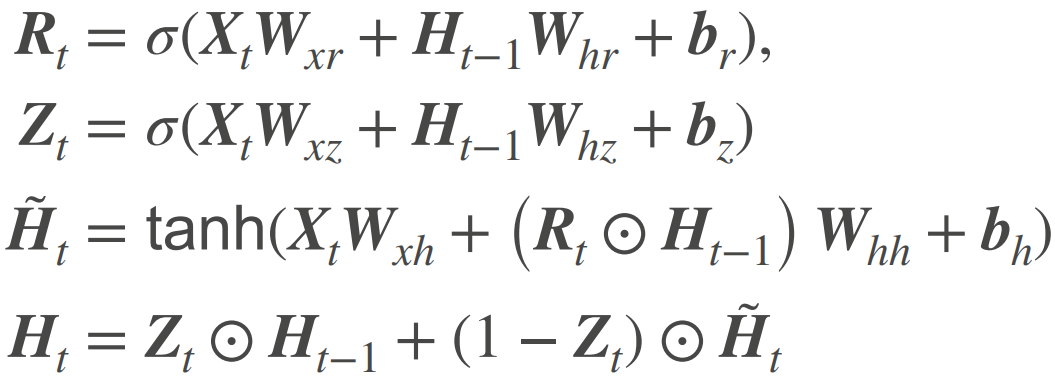


GRU：

门控循环单元，Gated Recurrent Unit。

类RNN网络。它是对RNN的改进，引入了**门控机制**，它有点类似于注意力机制，具体我不多讲。GRU和RNN一样，也是以隐变量H为核心，输入Xt-1和Ht-1，输出Yt。

具体来说，是除了隐变量H外，还加了个Rt参数和Zt参数。其中R是reset gate，Z是update gate，H~是候选隐变量，如下式子(Xt应该写成Xt-1，这里有误)：



可以看得出来，重置门的作用是选择要不要让当前隐变量少看点儿Xt-1，更新门的作用是选择要不要让当前隐变量多看点Ht-1。

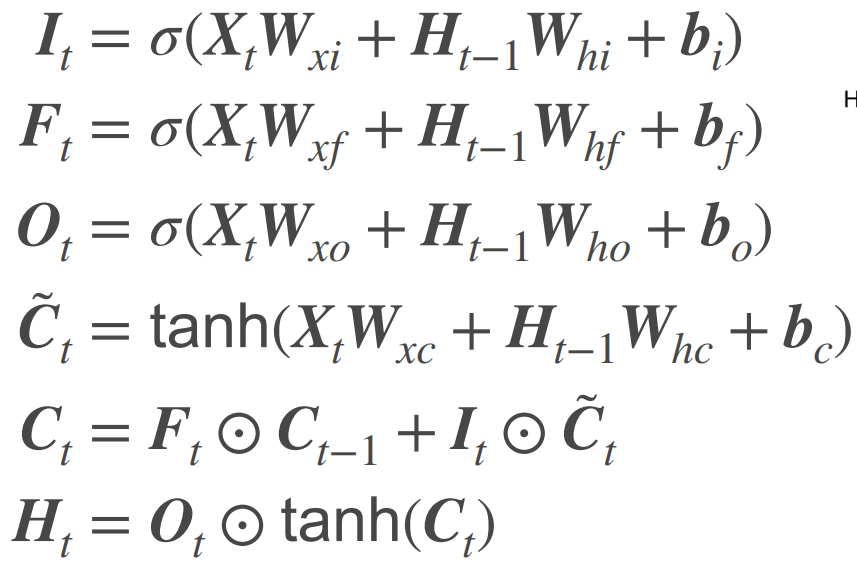
要注意的是，Rt和Zt的激活函数必须是sigmoid，因为sigmoid的输出值是(0,1)，这符合我们对R和Z的语义定义。选择sigmoid当激活函数或许还有一个意外的好处(虽然未经一定线性变换的sigmoid可能不利于在超参数初始化时缓解数值不稳定)，就是它让R和Z的值较小，在反向传播求导时求得的导数也较小，在一定程度上可以说是缓解梯度爆炸吧。

LSTM：

长短期记忆神经网络是对RNN的改进(GRU是在LSTM之后出来的)。

类RNN网络。LSTM我感觉就是一个RNN多了一个“隐变量”C，当然这里叫记忆单元，它的作用美其名曰“**记忆机制**”。然后记忆机制的计算就美其名曰“**门控机制**”。

非常简单：



C~叫做候选记忆单元，I叫做输入门，控制是否要忽略掉输入数据，F叫做忘记门，觉得是否忽略掉上一个时间步的记忆，O叫做输出门，控制输出。

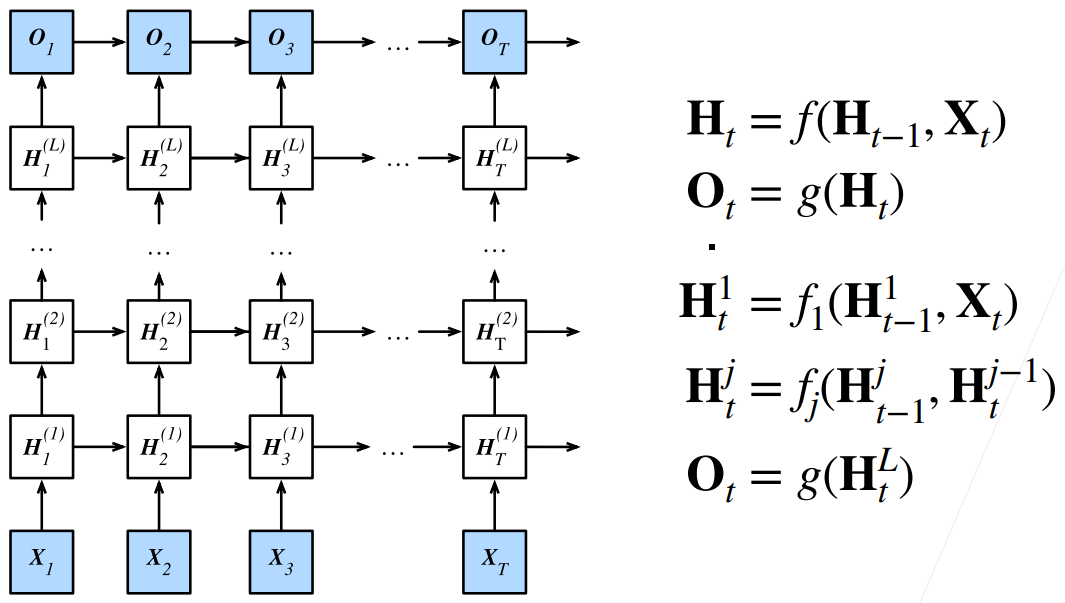
要注意的是，门的取值范围都应该在(0,1)，所以激活函数得用sigmoid，而Ht=Ot\*tanh(Ct)，所以H的取值在(-1,1)。C的初始化和H一样，通常全取0。

LSTM把区分C和H有个好处，你有没有看到，我们RNN中的隐变量H好像都是用激活函数tanh算出来的，取值范围都在(-1,1)，而**C的取值范围可以随着时间步变得越来越大，它能表征H所不足以表征的更多信息**。那为什么H取值要限制在(-1,1)呢？可能是为了数值稳定性和限制模型容量提高泛化性吧，那么C的取值范围为什么可以那么大呢？大概是在反向传播求导C的过程中由于门控机制的存在能保证C的梯度不会太大吧，而且同样由于门控机制的存在导致C不直接参与output的计算，所以模型容量也不会太大吧。那么这么总结说来，**LSTM中，引入了以C为代表的记忆机制，是为了存储更多时序信息**，而**引入了以FIO为代表门控制机制，是为了防止引入C记忆单元顺带来的数值不稳定和模型容量提升**吧。当然，**门控机制也控制以H的计算**。归根结底，**记忆单元C都是为了辅助H的计算——H还是类RNN神经网络的核心**。

(深层)RNN：

前面讲的是浅层RNN和基于浅层RNN的GRU和LSTM，浅层RNN实际上就相当于一个两层MLP，就算是多了个记忆单元C的LSTM也不能算是多了一层神经网络，因为对于前面讲的LSTM而言，C和H实际上是处于一个层上。

对于深层RNN而言，其相当于一个n层MLP，n>=3，实现起来也是很简单的，但它模型容量更大，相较于浅层RNN而言引入了更多非线性性。



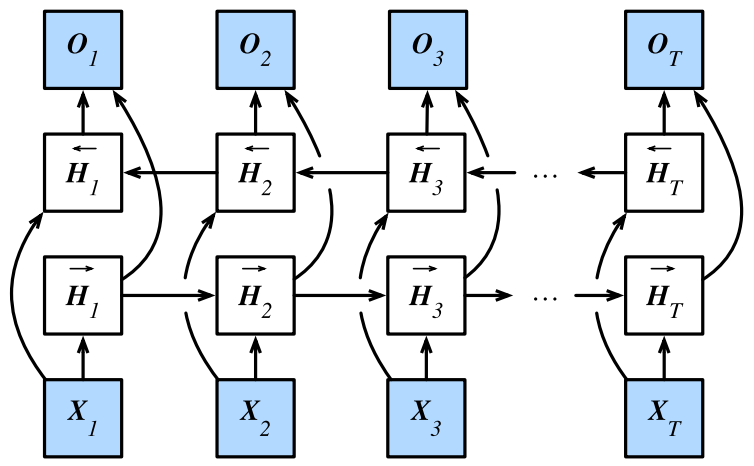
对于多层RNN而言，每层RNN都相当于对当前及之前的token序列提取出了特征，其中以hidden state最后一层H[-1]所代表的特征信息量最大，最为重要。所以你能看到H[-1]在做文本翻译RNN中的妙用，以及H[-1]在双向RNN做特征提取器时的妙用。

双向RNN：

bidirectional RNN。

双向RNN没办法处理给定一组序列，预测下一组序列的功能，但它可以用来做完形填空或提取文本特征。

双向RNN的本质还是一个以H为核心的RNN，只不过它把H分为了两组，即前向隐变量H->和反向隐变量H<-，在浅层双向RNN中，H->和H<-在同一层，它们各自接受前一个时间步H->t-1和后一个时间步H<-t+1的隐变量信息输入以更新自身，当然，它们都统一接受前一个时间步的input Xt-1的数据输入以更新自身(**好像在很多地方，都相关性地统一将Xt-1表示成了Xt**，可能是为了不言自明的方便吧)。双向RNN结构如下：

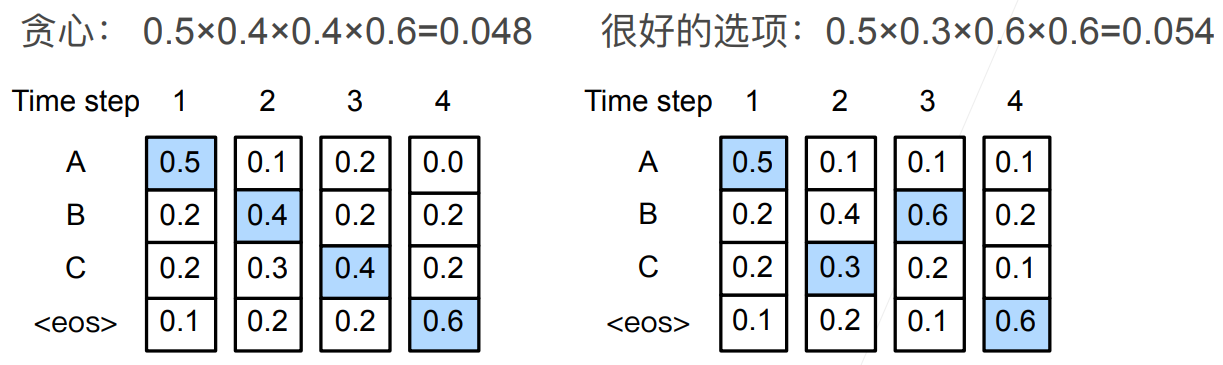


双向RNN如何训练？你给定一个文本序列“abcdefg”，你要预测abc和efg中间的d ，你训练的时候，先把abc给BRNN，让它更新H->和H<-，当然此时只有H->在为下一个时间步提供信息，然后把efg倒过来，把gfe给BRNN，让它更新H->和H<-，当然此时只有H<-在为上一个时间步提供信息。最后，你那得到的H<-和H->去预测最终的d。

BRNN的工作模式注定了其不能给定一组input序列然后预测下一组序列，它更适合做完形填空，或者用来抽取一段文本的特征，抽取出来的特征就是中间变量H，是不是和CNN抽取feature map很像？

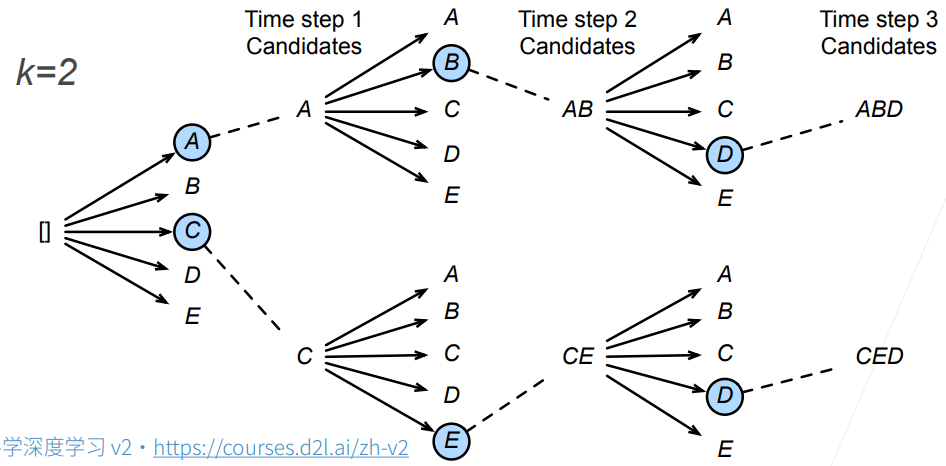
束约束：

上述我们介绍的RNN中，我们默认用了贪心的策略来得到最终预测的模型。这是指：当我们把RNN当做一个语言模型时，我们预测得到的每个时间步理想都是一个token的独热编码，但实际上它是一个softmax概率向量，我们选出这个概率向量中最大的那个作为当前时间步的预测token值，钉死。这是一种贪心的策略，实际上最终得到的结序列果可能不是最优序列：



为了防止贪心的弊端，得到更好的选项，我们可以选择遍历所有可能的序列，但是这样太贵了。

于是我们有了折中方案，束约束，每个时间步保存当前最好的前K个预测token，作为下一个时间步的输入，继续预测，重复上述步骤。最终得到k个token序列，计算哪个序列总概率最大，即是最终结果。



文本预处理：

在做nlp模型训练的时候，你肯定不能直接拿自然语言文本来做处理，比如给你一段话，你肯定要用数字对这段话做编码，得到一个tensor后然后拿去做训练。如何编码？我这里提供一个基本pipline，主要包含四步：

1.下载一个英文.txt文件，把这个文件里的非26个英文字母的字符全部变成空格，然后把全部字母都变成小写。然后，我们做一个列表，这个列表的每个元素也是一个列表，子列表的内容是原文每一行的所有单词字符串，我们暂且管这个二维列表叫做tokens。至于为什么要把txt中的内容做成这样一个二维列表tokens的结构，纯粹是为了后序好处理，你要是高兴，直接把txt的内容做成一维列表tokens也完全没问题啊，你vocabulary类的init函数能处理好这个一维列表的数据类型就行。

总之，我们这里就当做是你得到一个二维列表tokens了，每一个元素word被称为一个**token**(nlp中常用术语，译作令牌。但它其实是个通用概念，表示数据的最小单位)——其实这里，你也可以把每个字母当做一个token，这样的话，你最终的元素种类就只有28种了，也行，但是我们一般不这样做，我们一般还是把一个word当做一个token，我们下面按照一个word一个token来处理。

2.然后你要建立一个vocabulary类，这个类的对象存储一个编码表，编码表逻辑上来说是一个字典，键是token，值是编码item，但为了方便vocabulary类方法的使用，我们在代码实现上还增添了一个列表，列表的每个元素都是token，下标为对应item，主要是为了方便用item去得到token，然后字典方便用token去得到item。故而，这个类具备两个主要方法：token2item传入一个token，返回它的item，常在train时用；item2token传入一个item，返回它的token，常在infer时用。还有另外一个重要的方法：len返回编码表的长度。

如何编码？这一般在vocabulary类的\_\_init\_\_函数中完成，我们给它传入3个参数：tokens二维列表、min\_freq、reserved\_token。我们先统计tokens中所有独立的单词及其在文本中的出现次数，然后按照它们的出现频次从大到小排序，然后把出现频次小于min\_freq的token给剔除掉，剩下的单词加入编码表。但是这些tokens里的单词并不是直接就覆盖掉刚初始化的编码表，而是衔接在编码表的末尾，编码表初始时里面就有内容了，编码表第一个键值对是我们在nlp中常常使用的：”<unk>”:0，unk是nlp中另一个常用术语，表示未知的word，它的编码是0，因为vacabulary不可能覆盖掉所有可能存在的token，它只存储原本.txt文件中出现频次大于min\_freq的word——你总得在vocabulary中预留一个编码应付可能的所有情况，即用”<unk>”:0来解决。接着”<unk>”:0，下面的几个键值对也不是tokens中被min\_freq筛选出来的token，而是reserved\_token，reserved\_token代表你人为想要格外关注的token，不管tokens二维列表里有没有出现，你都想把它加入编码表。

注意，在制作编码表的过程中，有一个中间成员变量self. token\_freq，它是一个字典，键是在原txt中出现的token，值是对应token在原txt中出现的频次freq，这个self. token\_freq十分有用，能被外界访问。

综上二点，**总的来说，你实现一个编码表类(字典)，编码表包含3块：”<unk>”:0+reserved\_token+被min\_freq筛选过的tokens**。

然后**你主要从这个class中用3大功能：1.用item得到token；2.用token得到item；3.通过访问self.token\_freq得到每个token在原txt中的出现频次**。

(补充一点：为什么我们偏偏要把原始tokens中的token按出现频次从大到小来排序分配item？一方面是为了min\_freaq好筛选；另一方面是我们要频繁从vocabulary字典里取token/item，我们把出现频次高的放在相近的一块区域，对计算机性能来说有一定好处。)

3.这个时候你把txt中的内容tokens传给vocabulary类即**得到对象vocab**，它能实现item2token()和token2item()两大功能，并提供self.token\_freq供外界访问；然后你还要做一件事，就是利用vocab.token2item()将txt中内容tokens全部从字符转换为item编码序列corpus，这里的序列通常应该是一维的，这是你后序要训练语言模型用的**文本序列corpus，也可称作语料库**。

4.接下来你还要写个迭代器，这个迭代器是用来训练RNN语言模型时用的，相当于一个DataLoader。你在初始化这个迭代器的时候告知它batch\_size和每个批次的文本序列的长度(时间步长)T，该迭代器就能每次返回X、Y，它们的shape都是batch\_size\*T，X和Y都是取自corpus，但X是feature，Y是相对于X滞后一个时间步的label，X-Y即是一个样本。其实DataLoader从corpus中取X有两种取法，corpus不是一个一连串很长的文本item编码序列吗，一种取法是从头到尾，顺序取X，一种是先按照T把corpus分组然后打乱，然后按照被打乱的分组随机从corpus中取X，至于取Y就完全依据X来取了。这两种从corpus中取X-Y的取法是很关键的，它会导致你最终训练RNN时，每个batch训练时要不要先把隐变量h重新初始化，假如迭代器取corpus中的X-Y是顺序取的，那么**一个epoch(每轮从迭代器中把corpus中样本全部取完称作一个epoch)**只初始化一次h，因为每个batch在训练时可以利用上一个bacth训练结束时留下的隐变量h保存的时序信息，当然你一定一定一定要记住，**在一个batch利用上一个batch留下的h来做这一轮bacth的训练时，一定要用detach函数删除掉上一轮h留下的计算图，不然在train时会有很大的求导计算开销**。然而，假如迭代器取corpus中的X-Y是乱序随机的，那么每个batch训练时都要重新初始化一遍h，因为上一个bacth的X和这一轮batch的X不是连续的，你用不了人家h保存的时序信息。

其实我觉得，乱序从corpus中取样本时还是乱序取比较好，因为你顺序取，train时每轮batch固然可以利用上一轮batch的时序信息隐变量h，但你在真正用这个RNN的时候，你的场景应该是用户给一个input序列，然后你的RNN生成一个初始化h，然后利用这个h和input，去生成output，这种场景下，你上哪去找所谓“上一轮时序信息h”？你每一次推理都是第一轮，每次推理都得重新初始化隐变量h——诶好像也不一定？不管了，反正我觉得顺序样本和乱序样本两种训练策略中，后者训练出来的RNN泛化能力更好。

你最终得到的是能做到item2token和token2item的vocab，以及返回feature和label的迭代器，还有vocab的属性——语料库corpus。

其实上述所说，不过是最基本的文本预处理，面向不同的任务或模型有不同的更复杂的文本预处理的方式，但上述可以说是那些更复杂文本预处理的基础。我下面举一个更复杂的文本预处理方式，它面向的是文本翻译任务，这里举例做文本翻译任务所使用的模型是 seq2seq型的RNN(GRU\LSTM都是类RNN网络)，它很简单，它的任务是将固定长度num\_step的英文文本翻译成固定长度num\_step的法文文本，以word为token，这样的RNN我在上文总体介绍RNN时有详细解释——所以我们的文本预处理也要面向这样的模型结构+任务。

假如我们有这样一个数据集，数据集的feature是英文，数据集的label是法文，每个feature-label样本是英文一句话，法文一句话，还包含标点符号，所以最后我们的token也要包含标点符号。一句话的长度不一定是num\_step。我们要训练一个RNN把英文翻译成法文，我们就要做这样的文本预处理：

首先我们要分别对英文文本和法文文本做一个上述的vocab类vocab\_E、vocab\_F，构造vocab是传的reserved\_token有<bos>、<eos>、<pad>，其中<eos>代表end-of-sentence，<pad>代表padding，<bos>代表begin-of-sentence，前两者主要用在这里的文本预处理，后者主要用在最后的模型推理阶段。然后利用vocab，对于一个样本，我们固定把feature和label分别做成长为num\_step的向量，怎么做？我们先将一句话(不论是法文还是英文)用vocab转换为item编码序列，然后给这段话的末尾加上<eos>的item，然后对这个新的编码序列向量做长为num\_step的截断——若是本来这个编码序列长度就不足num\_step，那我们就在不足的部分补上<pad>，反正你模型最后推理的时候会做num\_step个时间步推理(看，这里我们就说文本预处理方式面向了模型+任务)，若是某个时间步推理得到了<eos>，表明翻译结束，下面的时间步直接break就是了；若是这个编码序列长度超过num\_step，那就直接截断，你说，把<eos>截断了，那RNN推理的时候就找不到句子什么时候翻译结束了啊？错了，我说这里的seq2seqRNN固定推理num\_step个时间步长。所以说，最后你的feature-label虽然都是长度为num\_step的向量，但是其中有效的内容长度valid\_len(包括<eos>)却不一定是num\_step，你要做一个分别针对feature和label的向量，分别表示每个样本的valid\_len，这个valid\_len很重要，因为最后你在这个RNN的模型训练过程中，计算损失函数是，只需要计算长为num\_step(用来valid-test的时候遇到推理结果<eos>立马break，但在训练过程中要老老实实推理num\_step个时间步)推理结果序列中的前valid\_len的item和对应label前valid\_len个item的差异即可。诶，那这么说来，你好像只用到了label的valid\_len啊？你是对的，对于这个RNN而言，不用feature的valid\_len，只有对于引入注意力机制的RNN，我们才会用到feature的valid\_len。

以上，你面向“一个seq2seqRNN对num\_step长度的英文翻译成num\_step长度的法文”的模型+任务的文本预处理就结束了，你先得到了两个不同的vocab，然后利用这个vocab将将原本的feature-label做成了长度固定为num\_step的编码向量，以及附带的feature-label valid\_len。你能看出，它是以vocab为基础了，虽然没用到corpus，但是用到了token2item，最后模型推理时也必然用到item2token。

语言模型：

语言模型是序列模型的一种，它是用来估计一段文本序列出现的概率。

例如估计文本序列x1,x2,x3出现的概率，可以这样算：

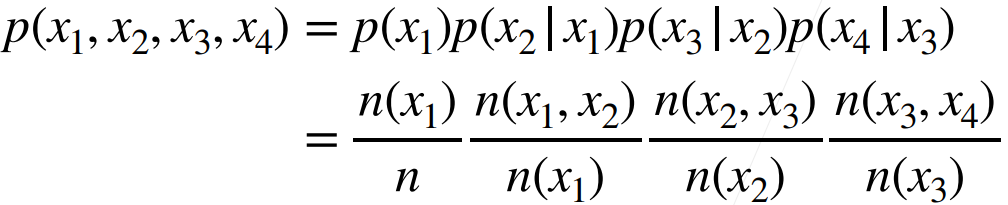
p(x1,x2,x3)=p(x1)\*p(x2|x1)\*p(x3|x1,x2)=n(x1)/n\*n(x1,x2)/n(x1)\*n(x1,x2,x3)/n(x1,x2)。

所以计算p(x1,x2,x3)时，你只需要统计出给定原始txt中的n、n(x1)、n(x1,x2)、n(x1,x2,x3)即可，其中n和n(xi)你在文本预处理阶段构造vocabulary类的时候就已经统计过了。小小提一嘴，实际上你估计p(x1,x2,x3)，你观察上面的通式，实际上你只要知道n(x1,x2,x3)/n即可了。

但有时候，若是估计一段很长的文本序列的概率p(x1,x2,…,xm)，由于原txt不够大，你很有可能在原txt中找不到这样一个序列，你就没法估计p(x1,x2,x3)=n(x1,x2,x3)/n了，怎么办？像是在一般的序列模型中的一样，我们引入马尔科夫假设，假设当前长度token出现的概率只跟过去N-1(即tau，我们这里取N为2)个token有关，则p(x1,x2,x3)=p(x1)\*p(x2|x1)\*p(x3|x1,x2)=n(x1)/n\*n(x1,x2)/n(x1)\*n(x1,x2,x3)/n(x1,x2)可以退化成p(x1,x2,x3)=p(x1)\*p(x2|x1)\*p(x3|x2)=n(x1)/n\*n(x1,x2)/n(x1)\*n(x2,x3)/n(x2)，这样，就算原txt中未出现过序列(x1,x2,x3)，你也可以估计它出现的概率，只需要统计n,n(x1),n(x2)，n(x1,x2),n(x2,x3)即可，像上述这种基于马尔科夫假设的语言模型，我们称之为“N元语法”(N-gram，我们这里把N取作了2。若是n取1、2、3就叫uni-gram、bia-gram、tri-gram)。

总结N元语法的实现就是：**你要估计任意长度token序列的出现概率，只要token在原txt中出现过，你都能实现，你只需要统计原txt中长度为0到N的所有token序列的出现频次n(x1,…,xm)即可**。

这里再举个二元语法估计(x1,x2,x3,x4)出现概率的例子：



你只需要统计原txt中的n、长度为1的token序列的频次、长度为2的token序列出现的频次即可，即能实现二元语法语言模型。**若是你已知了x1,x2,x3，要预测x4是什么，你只需要看看原文本中哪个word作为x4能使得n(x3,x4)最大就行了——这，就是二元语法**。顺带提一下，你这里统计长度为2的token的序列出现的频次，实际就相当于把原txt中连续的2个word当做一个token了嘛，你还是可以不加改动地复用文本预处理阶段就准备好了的vocabulary类，然后它就能帮你统计好长度为2的token的出现频次了嘛。

上述这种N元语法语言模型不是基于learning的，它基于统计，它也是马尔科夫序列模型的一种。其实，n-gram的概念有时还可以泛化为关注长度为n的序列的文本预测模型，而不仅限于这里tau=n-1的马尔科夫假设的统计语言模型，例如在机器翻译模型中，BLEU中的pn项就是衡量预测结果中长度为n的序列的命中率

当然也有基于learning的马尔科夫模型，它不能处理预测整个(x1,x2,x3,x4)的出现概率，它只能处理已知(x1,x2,x3)预测x4的功能，它就没什么“条件概率”的数学可解释性了，充其量是让model去学习x1,x2,x3,x4之间的概率关系。