刘文哲

语音增强初探

June 14, 2022

Springer Nature

Contents

Part I
基于深度学习的语音增强

语音增强算法引入深度学习大抵是一种必然，讨论基于深度学习的语音增强算法我们必须首先把目光放在Prof. DeLiang Wang的OSU-PNL身上。Prof. Wang 是用于语音分离的计算听觉场景分析(CASA)领域的代表任务，他提出用理想二值掩蔽(IBM)解决语音增强（分离）问题。理想二值掩蔽，其中掩蔽广泛用于图像处理之中，具体到语音增强领域是指对选定的时频区域进行遮挡以控制要处理的区域，二值则是对每个时频点量化的精度非0即1，理想二值掩蔽的含义则是根据纯净语音和噪声之间的能量关系，将语音能量占主导的时频点标记为1，噪声能量占主导的时频点标记为0，由此得到一个滤波器，对带噪特征进行滤波，在语音时频稀疏性假设下留下的即使语音成分。于是，熟悉机器学习的读者应该很容易发现，语音增强问题在IBM的定义下可以被看做一个分类问题，当时OSU-PNL的Y. Wang已经开展了支持向量机(SVM)估计IBM的研究。机器学习中特征提取和目标选择十分重要，作为机器学习模型的SVM当然也不例外。Y. Wang的研究当然也包括评估众多利用听觉特性构造的手工特征对SVM进行语音增强的性能影响，在深度学习大行其道的今天，利用深度学习提取更有效的特征的思维并不难理解，而Y. Wang在2013年也的确选择这样去做。如果你对机器学习有一定了解的话，应该了解机器学习任务可以分为分类任务和回归任务两种，基于回归模型的语音增强算法也于2014年被中科大USTC SPRAT的Y. Xu 等人提出。基于回归的语音增强模型直接利用神经网络建立从带噪语音谱特征到纯净语音谱特征的映射，利用网络直接生成增强的对数功率谱。如果说Y. Wang的工作是基于监督学习的CASA算法的延申的话，Y. Xu的这项工作则更具有基于统计模型的传统语音增强算法的风格。工作中沿用了MMSE-LSA中的对数谱特征以及传统语音增强方法中人耳对相位信息不敏感的假设。然而利用分类和回归去分类这两个极具代表性的工作并不利于对后续方法的研究，因为后续的理想比值掩蔽(IRM)、相位敏感掩蔽(PSM)等训练目标也明显受到了传统语音增强统计模型的影响，这类掩蔽估计由于其定义也不再能被视为分类问题(而同样是回归问题)。这类掩蔽本质上是在设计某种意义上最优的滤波器，毕竟，回顾IBM的介绍“IBM的含义正是根据每个时频点上语音和噪声之间的能量关系，将语音能量占主导的时频点标记为1，噪声能量占主导的时频点标记为0”中“主导”二字充满了不安全感。一个极端的例子是，在极低信噪比情况下IBM会出现非常多的0值导致理想掩蔽的滤波结果的语音质量和可懂度都会十分糟糕。IRM将IBM的硬分类变成了一种软分类，每个时频点的滤波器系数定义为语音能量和语音噪声二者能量之和的比值。熟悉传统语音增强算法的读者不难发现，这个定义和频域维纳滤波算法尽管并不相同但还是十分相像。因此，Prof. Wang在综述中从训练目标角度将上述两大类工作分别定义为基于掩蔽(masking-based)和基于谱映射(mapping-based)的算法用于分类，这种分类方式得到了广泛地接受。

尽管上述内容足以概括基于深度学习的语音增强在早期阶段的历史，然而除了训练目标，模型架构的发展同样值得我们关注。不过在这一段中我们对文献中的训练目标使用掩蔽还是谱特征并不感兴趣，因此，如无必要在介绍中不会涉及映射目标。另一方面，虽然本节介绍的是深度学习语音增强的发展，然而为了保证叙述的连贯性，本段刻意忽略了算法的提出时间，只保证了相关联工作的顺序。由于语音具有时序性，因此时间建模是网络结构设计上不可忽视的问题。前面提及的方法都是利用全连接网络，考虑时序关系的方式也十分朴素——将当前帧前后几帧组合在一起形成上下文窗(context

window)后送入网络。很明显，这种方式并不足以建模长时信息。而循环神经网络(RNN)天然适合时序建模，因此2017年OSU-PNL的J. Chen和USTC-SPRAT的Sun分别开展了基于LSTM的语音增强模型的工作。这类网络和全连接网络唯一结构上的区别在于全连接网络的其中几层（通常是前几层）被替换成了LSTM层，但其在增强性能、未见说话人和噪声类型的泛化能力上都明显优于全连接网络。RNNs的提出使得目前基于深度学习的语音增强模型几乎全部放弃了全连接网络。另一个需要提及的内容在于卷积神经网络的应用，熟悉图像分割的读者应该已经感受到了语音增强与图像分割的相似之处，因此卷积神经网络的引入也十分自然。不过在介绍卷积神经网络在语音增强的应用之前，有必要至少提一点语音增强与图像分割的区别，尽管我们可以将语音特征看成一张图片，但是由于语音增强有很多实时应用场景，因此在这张“图片”时间维度的处理需要十分谨慎“实时陷阱”。卷积层由于“局部连接、权重共享”的特点相比全连接层和循环层拥有更少的参数，2015年清华 SATLab的L. Hui提出的maxout卷积网络首次在语音增强领域性能超过全连接网络。而后中央研究院Bio-ASP Lab的S. Fu和卡内基梅隆大学的S. R. Park分别将图像领域常用的普通卷积神经网络和卷积编解码器结构成功应用到了语音增强中。由于卷积编解码器结构对于时频点级预测任务的天然适配，基于卷积编解码器的语音增强模型逐渐成为主流。自然地，将卷积层的特征提取能力和循环层的时序建模能力结合的工作应运而生。2018年H. Zhao在微软亚研完成了卷积层特征提取、LSTM时序建模、最后由全连接层进行幅度谱估计的EHNet，同年OSU-PNL的K. Tan将LSTM插入卷积编解码器之间提出一种卷积循环网络(CRN)。除了RNN适合序列建模外，时序卷积网络 (TCN)利用一维空洞卷积实现了并行时序建模过程。2018年K. Tan引入门控(gating)TCN取代RNN构造了一种全卷积增强网络。

随着基于深度学习的幅度谱估计算法性能的快速发展，两个很有意思的现象出现了。一是相对简短可以介绍完的，是深度学习反求诸传统，深度学习作为一种参数估计器为传统语音增强中的MMSE、MMSE-LSA和卡尔曼滤波等算法提供依赖先验假设更少且更为准确参数估计。一般将这类深度学习与传统算法结合的方法称为混合式模型，然而混合式模型类工作相比性能提升更重要的贡献在于让基于深度学习的语音增强渐渐褪去纯机器学习或深度学习意味的工作，而逐渐带有语音信号处理的特色。当然，相比于上述这种“主动”结合语音特色，另一种相对“被动”的结合方式产生于另一个有意思的现象——基于深度学习语音增强的性能第一次遇到了瓶颈。当然人们很快意识到了问题——这一性能限制是只进行幅度谱估计带来的，相位恢复似乎对人耳感知是重要的。以至于解决该问题的方案出现得如此之快在今天看来似乎并没有瓶颈的意味在。在相位恢复重要性思潮下，用于同时进行幅度估计和相位恢复的方案大方向上可以分为时域的方案和谱域的方案。不论是哪种方案他们都是为了解决由于相位谱没有清晰的模式结构导致其不易优化的问题。首先介绍时域模型，原因依旧是因为介绍起来可以比较简略。时域模型的兴起主要由于其在语音分离领域大获成功，以一种波形到波形的映射方式提供了一种解决相位谱不易优化问题的方案——放弃建模相位。不过也许是由于噪声模式的复杂性，抑或是房间冲激响应的敏感多变，时域模型在噪声混响场景下的性能和鲁棒性使其在语音增强中并不能成为主流。网络结构上和之前的幅度谱架构基本还是类似的，有卷积编解码器结构也有卷积循环(或TCN)网络，只不过对应的卷积层大多用一维卷积代替了二

维卷积(当然也同样有使用二维卷积作为编解码器的架构)。相比之下，谱域的方案解决相位估计的方式则略显复杂。众所周知，复数可以由幅度相位表示也可以由实部虚部表示。其中实部虚部表示和之前幅度谱模型一样，主要分为基于掩蔽和基于谱映射的方式。2016年OHU-PNL的Williamson提出了一种复数比值掩蔽(cIRM)，分别估计实部和虚部的掩蔽；而2017年S. Fu提出了一种谱映射的CNN，2019年K. Tan将CRN应用到复数谱映射直接利用网络估计增强语音的实部和虚部。2019年首尔国立大学的H. S. Choi提出了一种基于极坐标系的复值掩蔽策略(pcRM)，通过网络估计的cRM得到有界的幅度掩蔽和无界的相位掩蔽。尽管这篇文章成功应用了复值网络并且西工大ASLP的Y. Hu继复值U-Net发展为复值CRN取得了令人瞩目的性能，然而本篇文章对于在不同损失函数情况下的几种典型复值掩蔽的分析可能更加有趣：尽管这类复数目标算法都宣称致力于相位的恢复，然而其带来的性能提升究竟是源于相位的恢复还是噪声成分幅值的抑制(相位可能调整的极少)值得探究，D. Yin等人同样通过实验发现了使用cRM虚部几乎为0的问题。而为了体现相位的修复，基于幅度相位表示的方案得到了考虑，其中包括采用双分支结构同时估计幅度和相位，通过多任务学习的方式达到相位恢复的方法；也有一些算法采用了更具结构性的相位目标与幅度谱同时和先后估计，比如OHU-PNL的Z. Wang提出的群时延(GD)以及西工大CIAIC的N. Zheng提出的瞬时频率倒数(IFD)。

尽管Z. Wang的分析要晚于接下来要介绍的一些工作，然而在此处提及也许是必要。尽管这篇分析的重点在于解释RI+Mag损失函数为何是有效的，但其提供的幅度估计和相位估计的补偿作用无疑为解耦风格工作有效性提供了一种合理解释。所谓幅度和相位估计的补偿作用，是指在迫使网络输出逼近理想复数谱时，幅度谱估计的准确度会大打折扣。而如Griffin-Lim相位重构的很多相位估计算法都在幅度谱估计的基础上进行。一个解决幅度相位估计补偿作用的想法是，将复数谱的优化问题解耦为幅度初估计和包括相位信息在内的“全信息”估计两部分。尽管这仍属于复数谱估计的一类方法，然而这类算法在近年来备受关注且性能优异，因此有必要单独作为成段介绍。2020年，奥尔登堡大学通信声学的N. L. Westhausen先后利用两次信号变换——首先利用STFT幅度谱上进行幅度估计，而后再利用一维卷积生成的可学习的分析/合成基函数直接将语音时域波形映射到高维空间进行语音和噪声成分的分离，利用时域模型的方式完成相位恢复。哈工大SPLab的Z. Du在Mel谱(一种符合人耳听觉特性的幅度谱变换)域进行降噪，而后利用语音合成声码器根据增强的Mel谱合成语音波形从而实现相位信息的引入。中科院声学所IACAS-Lab9的A. Li则在2021年提出一种完全在STFT域上先后完成幅度谱估计和复数谱残差修正的框架，利用复数谱残差修复相位信息。完全在STFT域上处理两部分任务的一个好处是，原本级联的两个子任务出现了并行优化的可能，2022年A. Li和西工大ASLP的Y. Fu分别提出了幅度估计和复数谱修正的两种并行架构。其实，从复数谱的角度考虑，这种解耦是由于幅度相位估计的补偿作用导致幅度谱估计不准确造成的，因此，除了首先约束幅度谱后再根据相位差微调增强结果，也可以先得到复数谱结果后补偿不准确的幅度信息。于是在2022年，北交大IIS的T. Wang则先进行复数谱估计再利用谐波特征进行幅度谱修正的方法。而解耦风格的工作也并非只由从训练目标解耦一种方式。从语音被噪声污染的过程角度，早在2016年USTC-SPRAT的T. Gao就提出以不同信噪比的被污染语音作为目标将一次降噪过程

分解为渐进地、多阶段地、逐信噪比地提升。从语谱高低频特性差异角度，搜狗的J. Li对语谱进行子带分解，多阶段地处理各个子带。而对语音特性进行分析，语音可以分解为包络和周期两部分，2018年，J.-M. Valin首先利用GRU估计Bark域(另一种符合人耳听觉特性的幅度谱变换)的增益因子，而后利用信号处理的方式进行基频滤波抑制谐波间噪声残留，而后在2020年，其又对模型中各部分进行进一步提升，以可观的复杂度达到超过众多复数谱模型的性能。而对该工作神经网络化最杰出的代表当属PAU模式识别实验室的H. Schröter提出的在估计包络增益因子外利用网络估计深度滤波器(deep filtering)系数的框架。 除此之外，语音增强任务也可以理解为噪声去除和语音恢复两部分，因为单通道语音增强具有噪声抑制和语音失真之间的折衷问题，彻底去除噪声势必会对语音造成严重损伤。因此，一种先进行激进地噪声抑制而后再恢复在降噪过程中的失真语音的框架首先由TU Braunschweig通信技术所信号与机器学习组的M. Strake于2019年提出，2020年内蒙古大学IMUAI的X. Hao在该思路的基础上引入计算机视觉的概念形象地将该过程命名为“masking and inpainting”即将所有噪声的时频点去除后根据语谱相关性修补被破坏的语音时频点。而传统的谱减法则从带噪语音的合成过程反推，将语音增强看做噪声估计和移除噪声成分的过程。2020年，哥伦比亚大学C2G2的R. Xu受传统谱减法的启发将降噪过程分解为噪声的“masking and inpainting”过程(利用VAD得到静音段而后通过静音段推测出完整的噪声谱)并将噪声信息和带噪语音信息一同作为输入完成降噪。2021年，IACAS-Lab9的W. Liu将基于训练目标解耦的思路与基于带噪语音合成的解耦思路结合首先利用多任务学习幅度域对语音降噪并直接估计噪声成分，而后修正复数谱恢复过度抑制的语音。需要注意的是，尽管本段从解耦角度介绍了以上算法，但是这些算法在提出时受到的启发性工作和动机可能并非如此。这种分类目前也并不作为一种共识，而更多的是一家之言。

需要注意的是，网络架构和学习目标虽然的研究虽然更多然而只是性能提升的一方面，更长时间范围的输入特征以及精心设计的损失函数对模型性能的提升可能更加有益，其中损失函数由于不会在推理过程中引入额外的时延和复杂度而备受关注。一个有力的证据是在ICASSP2021 DNS-Challenge中微软提供的结合一种幂律压缩的复数谱均方误差损失的复数掩蔽网络相比包括解耦风格框架在内的一众高参数量和计算量方案仍表现出最优的性能。损失函数包括基于距离的损失函数、基于能量比的损失函数和基于感知指标的损失函数等。常见的距离表征最常用的莫过于基于范数的损失函数L_p损失，即平均绝对误差(MAE)和均方误差(MSE)，这两类误差都被用于度量时域波形、掩蔽、频谱以及嵌入式特征。时域波形的L_p度量形式相对简单，表示为$\mathcal{L}_{time-L_p} = \|s - \widehat{s}\|_p$，有时会加入对噪声项$L_p$损失作为正则，以提升语音估计性能，有文章表示，对于时域样本而言MAE相比MSE更关注小信号并具有更强的噪声抑制能力。除了IBM有使用交叉熵损失进行度量外，其余掩蔽大多都会采用L_p损失直接计算掩蔽间的误差$\mathcal{L}_{mask} = \|M - \widehat{M}\|_p$，J. Valin提出了幂律压缩的掩蔽MSE损失$\mathcal{L}_{RNNoise} = \|M^c - \widehat{M^c}\|_2$，利用指数项$c$控制噪声抑制的激进程度。然而，目前普遍认为基于信号近似(SA)技术的损失相比直接度量掩蔽间差异对语音增强的性能更有益：$\mathcal{L}_{SA} = \|S - \widehat{M} \cdot Y\|_p$。可以看出，此时对掩蔽的范数损失已经和基于频谱的范数损失基本一致，只需将滤波项$\widehat{M} \cdot Y$改为网

络映射的谱$\widehat{S}$：$\mathcal{L}_{spec-L_p} = \|S - \widehat{S}\|_p$。当然$S$和$\widehat{S}$可以是幅度谱或其变换(如对数谱、Mel谱等)，也可以是复数谱。由于幅度谱和复数谱是如此常用，这里我们分别给出其具体形式：$\mathcal{L}_{Mag} = \mathbb{E}[|A - \widehat{A}|^p]$和$\mathcal{L}_{com} = \mathbb{E}[|\mathcal{R}(X) - \mathcal{X}(\widehat{X})|^p + |\mathcal{I}(X) - \mathcal{I}(\widehat{X})|^p]$(其中$A$和$X$分别代表幅度谱和复谱)。$\mathcal{L}_{spec-L_p}$还可能根据如AMR编码等心理声学机制进入加权，以期望其更符合人耳听觉，而复数谱范数损失又常与幅度损失结合以缓解幅度相位估计的补偿作用，该损失被称为RI+Mag损失：$\mathcal{L}_{RI+Mag} = \alpha\mathcal{L}_{Mag} + \beta\mathcal{L}_{com}$。在此基础上，与时域波形$L_p$损失结合的$\mathcal{L}_{time-spec-L_p} = \mathcal{L}_{RI+Mag} + \mathcal{L}_{time-L_p}$、基于幂律压缩的RI+Mag损失$\mathcal{L}_{cprs} = \alpha\mathbb{E}[|A^c - \widehat{A^c}|^2] + \beta\mathbb{E}[|\frac{X \cdot A^c}{A} - \frac{\widehat{X} \cdot \widehat{A^c}}{\widehat{A}}|^2]$以及多分辨率(假设有$I$种STFT点数对应的$I$个分辨率)的$\mathcal{L}_{cprs}$函数$\mathcal{L}_{MR-STFT} = \sum_i \mathcal{L}^i_{cprs}, i = 1, \cdots, I$都成为目前广泛使用的$L_p$损失。另外由于$\mathcal{L}_{cprs}$对噪声的抑制能力较强导致语音失真较多，一种非对称损失有时也被作为正则项$\mathcal{L}_{asym} = \mathbb{E}[|max(A^c - \widehat{A^c}, 0)|^2]$。可以看出，尽管有文献提及由于STFT频点分布更类似拉普拉斯分布因此建议使用$\mathcal{L}_1$范数损失，然而大多数工作更喜欢选择$\mathcal{L}_2$范数损失。L_p范数损失最后也见于对嵌入式特征的距离度量，利用wav2vec、PANN等预训练模型生成嵌入式特征之间的距离作为谱距离损失的正则项。此外，一些除了$\mathcal{L}_p$范数作为距离度量，KL散度也有一些工作将其与$\mathcal{L}_p$范数联合度量谱距离：$\mathcal{L}_{KL} = \mathbb{E}[\hat{X} \cdot log(\frac{\hat{X}}{X})]$。基于信号能量比的损失函数主要用于时域波形，常见的有SDR(及其Log形式)和SI-SDR损失，分别定义为$\mathcal{L}_{SDR} = -10log(\frac{|s|^2}{|\widehat{s}-s|^2})$和$\mathcal{L}_{SI-SDR} = -10log(\frac{|\xi \cdot s|^2}{|s-\xi \cdot \widehat{s}|^2})(\xi = \frac{s^T\widehat{s}}{s^Ts})$。尽管SI-SDR被广泛应用于语音分离当中，但是由于其过多的语音失真导致最近的工作常将其与L_p范数谱距离损失联合使用。上述特征常被诟病并不能反映人耳的听感，因此，一些感知指标，如PESQ、ESTOI和CD，一度被引入作为损失函数训练模型。然而这类损失函数仅能提升被优化指标的性能，并没有带来其余指标的提升。要知道，目前的客观指标与人类的主观听感并不完全一致，单纯提升某些客观指标大多也并未带来主观听感的明显提升。于是近年来，这类感知指标类损失更多是以一种联合形式作为验证集损失被用于选择最优的模型。

尽管生成对抗网络(GAN)常作为一种不同的网络架构，然而这里我们更希望将其作为一种训练手段，即对抗性训练。众所周知，GAN由生成器和鉴别器两部分，在GAN在语音增强的使用中，其生成器可以用前文提到的众多网络代替，而之前提到的各种损失同样可以作为生成器损失函数的其中一项或等价于生成器损失函数。因此，GAN在语音增强中目前表现的作用，更类似于一种类损失函数”用于帮助网络专注于那些原本损失函数难以强调的噪声成分和语音伪影(artifacts)部分。于是，一言以蔽之GAN在语音增强的整体发展：之前的语音增强模型(如LSTM、UNet等)代入生成器，各式GAN的发展技术(如LSGAN、 Wasserstein GAN 、Relativistic GAN、Conditional GAN和Geometric GAN等)代入鉴别器。如此导致GAN始终并未在语音增强领域至少在实时语音增强领域成为主流技术 ，这很可能与语音增强任务与图像生成、语音合成等生成类任务的差异过大有关，然而，在无监督或自监督情况下，包括GAN在内的生成式模型(还包括变分自动编码VAE等)也许是值得研究的方法。

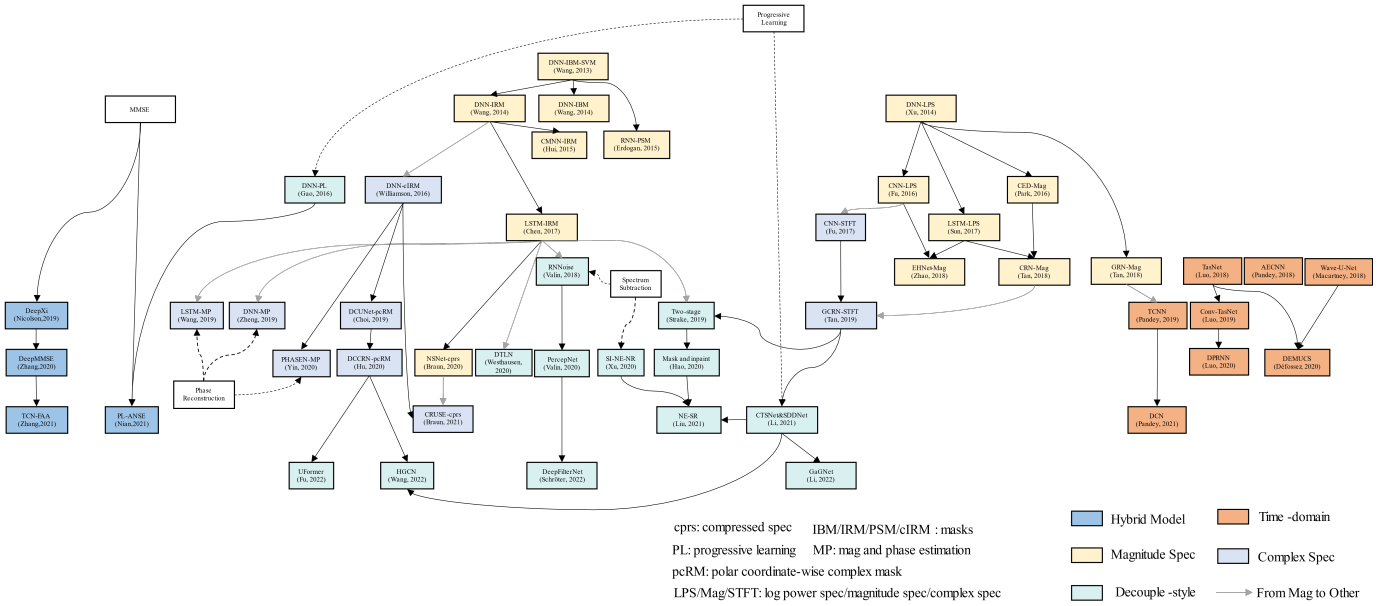

Fig. 0.1 深度学习语音增强算法发展

图0.1梳理了深度学习语音增强发展的部分脉络，然而对于目前处于发展之初的无监督语音增强、超宽带语音增强、多模态语音增强、个性化语音增强等并未涉及，去混响等问题也未讨论。

Chapter 1
基于幅度谱的语音增强

由于人耳对于幅度比相位更加敏感，传统的语音增强算法大部分都集中于在幅度谱上进行处理。于是，深度学习最早被引入语音增强时也在幅度谱上最先应用。在2013年和2014年，Y. Wang和Y. Xu分别成功利用神经网络估计滤波器和语谱，而后一段时间的工作都可看作这两类工作——基于掩蔽(masking-based)和基于谱映射(mapping-based)——的延续。基于掩蔽的语音增强方法利用网络预测出一个滤波器对幅度谱进行滤波，而后将滤波后的幅度谱与带噪相位耦合并重构回时域波形；基于谱映射的语音增强方法利用神经网络直接建立从带噪谱特征到纯净谱特征之间的映射，而后将谱特征与带噪相位耦合得到增强的语音。这两类方法的框图如图1.1所示，可以看出，二者的区别主要在于是否有显式的滤波过程(橙色部分)。

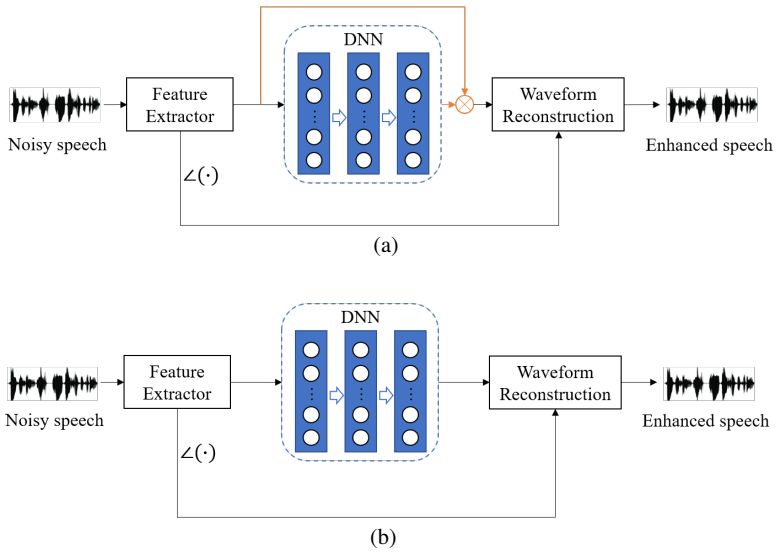

Fig. 1.1 (a) 基于掩蔽的语音增强处理流程， (b) 基于谱映射的语音增强处理过程。

由于基于掩蔽方法是估计滤波器并对带噪语谱滤波，早期这类算法估计的滤波器都是起抑制作用，不可避免地在语音能量弱但存在强噪声的谱上形成不连续的黑洞(black holes)，由于语音能量弱的地方可能是谐波之间(图1.2(a))、中高频谐波、以及轻音(图1.2(b))，这种不连续极其影响语音质量和可懂度；而基于谱映射的方法可以称之为脑补，算法的作用可以看成生成或者再生，凡是噪声存在的时频点均有可能完全抹去后重生出新的、结构类似的频谱，不过也正是由于谱映射的生成机制，早期的伪影(artifacts)问题(图1.2(b))、过平滑(over-smoothing)问题和鲁棒性问题常令人令人诟病。不过随着两类算法后来的发展，上述问题已经不再明显。

我们将图1.2(a)或(b)展示的语音的时频关系图称为语谱，语谱中的每个点被称为时频点(time-frequency bin)。

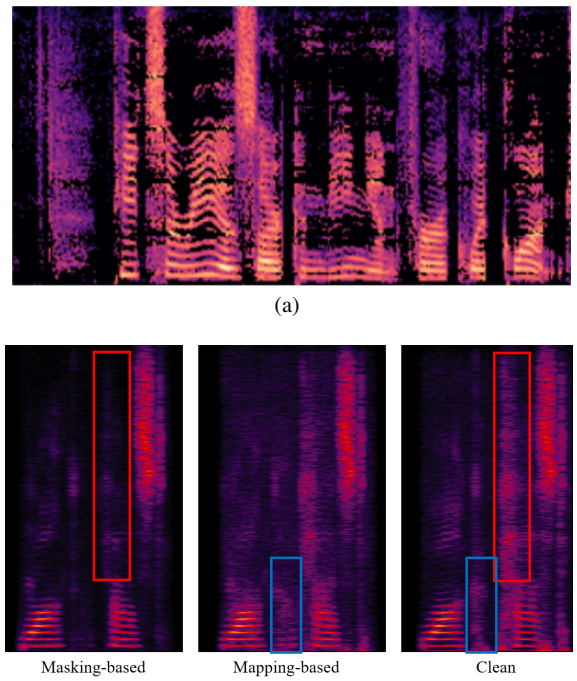

Fig. 1.2 (a) 频谱黑洞[2]， (b) 利用基于掩蔽和基于谱映射方法处理的语音，基于掩蔽的方法的清音被吞掉(见红框)，基于谱映射的方法在无谐波部分脑补出谐波(见蓝框)。

1.1 基于掩蔽的语音增强算法

1.1.1 处理流程

为了更好的解释基于幅度谱的语音增强算法，我们会选择一个或几个开源代码的一部分为例进行讲解。这个习惯将贯穿全书，这些开源代码及更多代码都被整理在awesome-speech-enhancement项目中。请注意：该仓库不承诺收录其中的仓库是完全满足论文声称的或复现正确的，但本书中选取的代码部分若与对应的论文不对应会进行提醒。我们选用NSNet作为基于掩蔽的语音增强算法的例子。

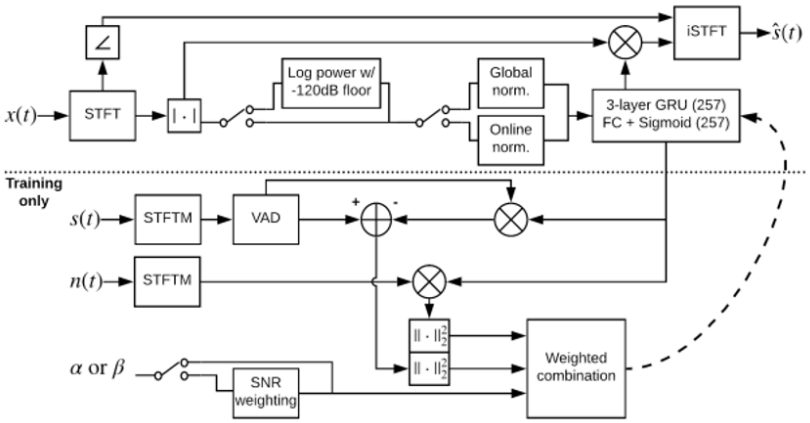

Fig. 1.3 NSNet框图[1]

NSNet的框图如图1.3所示，对于初窥门径者也许这个框图有些复杂，不过我也并不打算阐述所有部分。请将图1.1(a)和图1.3对应起来，图1.3虚线以上的部分，从左至右的，$x(t)$代表图1.1(a)中的带噪波形(noisy speech)，从STFT至代表网络的"3-layer GRU (257) FC + Sigmoid (257)"之间的那部分是特征提取器(feature extractor)，注意，图1.1(a)中的特征提取器有两个输出，一个送入网络(DNN)，另一个代表相位分量用于波形重构。请观察图1.3，同样可以发现相位从特征提取器中的STFT模块获得并传送给iSTFT模块，理所当然的，iSTFT对应的就是图1.1(a)中的波形重构(wavform reconstruction)。iSTFT的另一个输入自然是被网络估计的掩蔽滤波的特征。而$\hat{s}(t)$则对应了图1.1(a)中的增强语音(enhanced speech)。至于虚线以下的部分代表的是损失函数的计算过程，与虚线以上的联系是图中唯一的虚线，用于表示通过损失函数计算得到的梯度的反向传播，以更新网络参数。总结一下，基于掩蔽的语音增强方法首先将带噪语音经过特征提取得到幅度谱相关的特征以及相位，而后幅度谱相关特征经过网络得到时频掩蔽，时频掩蔽通过滤波幅度谱(或幅度谱相关特征)得到增强的幅度谱(或幅度谱相关特征)。而后增强

的幅度与带噪语音的相位结合重构得到增强的语音信号。NSNet采用了标准化的对数功率谱作为特征，而网络输出的是幅度谱的时频掩蔽。

> 输入特征不仅可以是对数功率谱，幅度谱、Mel谱和Bark域谱等都曾作为输入特征。输入特征的选取或多或少源于提出者的出身背景，比如对数谱、Mel谱和Bark域谱分别是传统语音增强MMSE-LSA、语音识别和音频编码领域常用的特征。

```
# 特征提取部分 wav_dataset.py lines 51-109
x_stft = torch.stft(noisy_waveform.view(-1), ...)
x_ps = x_stft.pow(2).sum(-1)
x_lps = LogTransform()(x_ps)
x_ms = x_ps.sqrt()
...
for frame_counter, frame_feature in enumerate(x_lps):
    ...
    norm_feature = (frame_feature - mu) / sigma
    frames.append(norm_feature)
    x_lps = torch.stack(frames, dim=0)
...
```

可以看到，NSNet的特征提取过程是将带噪语音经过STFT后，计算对数功率谱*x_lps*，将其进行标准化后作为网络的输入特征；幅度谱*x_ms*则用于与网络估计的时频掩蔽滤波得到增强的幅度谱。

网络部分则很简单，由三层GRU和一层全连接级联而成，最后的激活函数采用Sigmoid函数以保证网络的输出在[0,1]之间，这是由于IRM的范围即是如此(然而有很多工作表明将激活函数设为无界的ReLU可能带来性能的提升)。输入后的permute(·)操作是为了使变量维度变为(batch size, num frames, num bins)，使GRU和全连接对频率维进行操作。

```
# 网络部分 nsnet_model.py lines 37-53
def __build_model(self):
 self.gru = nn.GRU(...)
 self.dense = nn.Linear(...)

def forward(self, x):
 x = x.permute(0, 2, 1)
 x, _ = self.gru(x)
 x = torch.sigmoid(self.dense(x))
 x = x.permute(0, 2, 1)
 return x
```

在训练过程中图1.3(也就是虚线下面的部分)，是作者提出的一种损失函数设计方式，同时计算了语音失真和噪声抑制：

$$\mathcal{L} = \alpha\mathbb{E}(\|S - \widehat{M}\cdot S\|_2^2) + (1-\alpha)\mathbb{E}(\|\widehat{M}\cdot N\|_2^2), \tag{1.1}$$

可以看到，语音失真项损失是标准的信号近似(signal approximation, SA)形式，只用这一项作为损失函数的工作相对更多。要注意的是在训练阶段是使

用纯净语音幅度谱与掩蔽的哈达玛积作为估计项的，有的工作则采用带噪幅度谱与掩蔽的积作为估计项，即$\mathcal{L} = \mathbb{E}(\|S - \widehat{M} \cdot X\|_2^2)$。式1.1对应的代码为:

```
# 损失部分 nsnet_model.py lines 57-71
def loss(self, target, prediction):
    loss = F.mse_loss(prediction, target)
    return loss

def training_step(self, batch, batch_idx):
    ...
    y_hat = self.forward(x_lps)
    loss_speech = self.loss(y_ms[...], (y_hat * y_ms)[...])
    loss_noise = self.loss(torch.zeros_like(y_hat), y_hat *
        noise_ms)
    loss_val = self.alpha * loss_speech + (1 - self.alpha) *
        loss_noise
    ...
```

此外，只计算掩蔽之间距离的掩蔽近似(mask approximation, MA)形式同样是基于掩蔽方法常见的损失函数，比如nngev的损失函数就是计算语音和噪声掩蔽之间的交叉熵损失(阅读nngev源码时请注意论文采用的是非因果网络BLSTM):

```
# nn_models.py lines 17-22
def train_and_cv(self, Y, IBM_N, IBM_X, dropout=0.):
        N_mask_hat, X_mask_hat = self._propagate(Y, dropout)
        loss_X = binary_cross_entropy(X_mask_hat, IBM_X)
        loss_N = binary_cross_entropy(N_mask_hat, IBM_N)
        loss = (loss_X + loss_N) / 2
        return loss
```

> 由于采用SA形式的损失函数，因此时频掩蔽并不需要显式定义。而采用时频掩蔽距离作为损失函数则需要显式计算出目标时频掩蔽，掩蔽的相关内容将在下一小节介绍。

接下来关注NSNet项目的解码过程，也就是测试阶段的代码。

```
# test_nn.py lines 27-34
gain_mask = model(x_lps)
y_spectrogram_hat = x_ms * gain_mask
y_stft_hat = torch.stack([y_spectrogram_hat * torch.cos(angle(
    x_stft)), y_spectrogram_hat * torch.sin(angle(x_stft))], dim
    =-1)

y_waveform_hat = istft(y_stft_hat, ...)
```

可以看到，基于掩蔽的语音增强算法在测试阶段模型将提取的带噪对数功率谱特征*x_lps*作为输入并预测得到掩蔽*gain_mask*，而掩蔽与带噪幅度谱*x_ms*进行哈达玛积得到滤波后的幅度谱*y_spectrogram_hat*。根据$S_r = |S| \cdot \angle S$和$S_i = |S| \cdot \angle S$)的幅度相位-实部虚部关系将增强的幅度谱和带噪的相位耦合得到复数谱，并经过iSTFT重构回波形*y_waveform_hat*。

1.1.2 用于幅度谱的时频掩蔽

时频掩蔽的研究无疑是基于掩蔽的语音增强算法的重点。最早提出的时频掩蔽是理想二值掩蔽(Ideal Binary Mask, IBM)，根据语音时频分布的稀疏性，即语音成分在语谱中指分布于少数时频点，因此每个时频点上的语音和噪声之间的能量差异通常较大，于是可以将语音增强问题简单地看做将语音能量占主导的时频点筛选出来。二值掩蔽要做的，就是标记出那些语音占主导的时频点；而网络要做的，就是学习一个分类器，判断每个时频点是语音占主导还是噪声占主导。IBM被定义为：

$$M_{IBM}(t,f) = \begin{cases} 1, if \quad SNR(t,f) \geq T \\ 0, otherwise \end{cases} \tag{1.2}$$

IBM的代码来自speech-feature-extractor：

```
# speech_utils.py lines 114-120
noise_spect = stft_extractor(noisy_speech-clean_speech, ...)
clean_spect = stft_extractor(clean_speech, ...)
ibm = np.where(10.*np.log10(clean_spect/noise_spect)>=local_snr
    ,1.,0.)
```

理想比值掩蔽(Ideal Ratio Mask, IRM)用一个0到1的实值代替了IBM的非0即1，表征了时频点上语音能量和在语音噪声不相关假设下带噪语音能量的关系：

$$M_{IRM}(t,f) = \left(\frac{|S(t,f)|^2}{|S(t,f)|^2 + |N(t,f)|^2} \right)^{\beta} \tag{1.3}$$

式中β一般取0.5。可以看出，IRM的定义式形式上与维纳滤波器完全一致，但维纳滤波器是基于统计的线性滤波器，式中的功率值是求期望的结果。代码同样来自speech-feature-extractor：

```
# speech_utils.py lines 122-129
noise_spect = stft_extractor(noisy_speech-clean_speech, ...)
clean_spect = stft_extractor(clean_speech, ...)
mask = np.sqrt(np.square(np.abs(clean_spect)) / (np.square(np.abs
    (noise_spect)) + np.square(np.abs(clean_spect))))
```

此外，还有幅度谱掩蔽(Spectral Magnitude Mask, SMM)。相比IRM，SMM一方面在幅度上而非能量上进行定义，另一方面也没有进行语音和噪声不相关的假设。这是由于根据IRM的定义，在测试过程中的滤波过程$\widehat{M}_{IRM} \cdot |X| \neq |S|$。通过放弃语音和噪声不相关的假设，将SMM定义为$M_{SMM} = \frac{|S|}{|X|}$，其滤波后的结果是幅度谱上的最优滤波器估计，然而，这也导致了SMM的取值范围在$[0, +\infty)$之间。早期为了使训练目标有界，SMM的最大值常被截断为1或2(代码中截断至2)。

```
# speech_utils.py line 10 and lines 130-136
EPSILON = np.finfo(np.float32).eps
...
noisy_spect = stft_extractor(noisy_speech, ...)
```

```
clean_spect = stft_extractor(clean_speech, ...)
mask = np.abs(clean_spect) / (np.abs(noisy_spect) + EPSILON)
...
mask = np.where(mask > 2., 2., mask)
```

相位敏感滤波器(Phase-sensitive mask, PSM)则是复数域上的最优的实值滤波器，定义为：

$$M_{PSM} = \frac{|S|}{|X|}\cos(\angle S - \angle X), \tag{1.4}$$

可以看出，PSM的范围是$(-\infty, +\infty)$。和SMM一样，PSM也常被截断以方便训练，常用的截断范围为$[0, 1]$。

若采用SA的方式训练，损失函数定义为：

$$\mathcal{L} = \mathbb{E}\{(\widehat{M}_{PSM} \cdot |X| - |S|\cos(\angle S - \angle X))^2\}, \tag{1.5}$$

而解码时和上述掩蔽一样直接作用于带噪幅度谱后与带噪相位耦合得到增强的谱。

PSM的代码选自espnet，代码中的r是纯净语音的复谱，利用$\angle S = \frac{S}{|S|}$的方式得到相位，并利用三角函数关系式得到相位差的余弦值：

```
# espnet2/enh/loss/criterions/tf_domain.py lines 63-73
phase_r = r / (abs(r) + EPS)
phase_mix = mix_spec / (abs(mix_spec) + EPS)
# cos(a - b) = cos(a)*cos(b) + sin(a)*sin(b)
cos_theta = phase_r.real * phase_mix.real + phase_r.imag *
    phase_mix.imag
mask = (abs(r) / (abs(mix_spec) + EPS)) * cos_theta
mask = (
        mask.clamp(min=0, max=1)
        ...
    )
```

1.2 基于谱映射的语音增强算法

1.2.1 处理流程

基于谱映射的语音增强算法相比之下对前置知识的要求更低，与基于深度学习的图像分割任务类似，这类方法将语谱视为一张图片，网络接收带噪语谱图作为输入，直接预测得到增强的语谱图。本节将以CRN-causal为例展示基于谱映射的语音增强算法的处理过程。

尽管图1.4给出的是SEDNN的框图，但是我们将结合CRN-causal说明谱映射方法的流程。这张流程图看起来就简明得多，和图1.3相反，虚线以上是训练过程而虚线以下是解码过程。训练阶段的带噪语音和纯净语音首先经过特征提取，而后将特征送入DNN。经过损失函数训练后得到优化后的网络参数。测试阶段则将带噪语音Y^t经过特征提取得到幅度谱Y^1(而SEDNN的特征

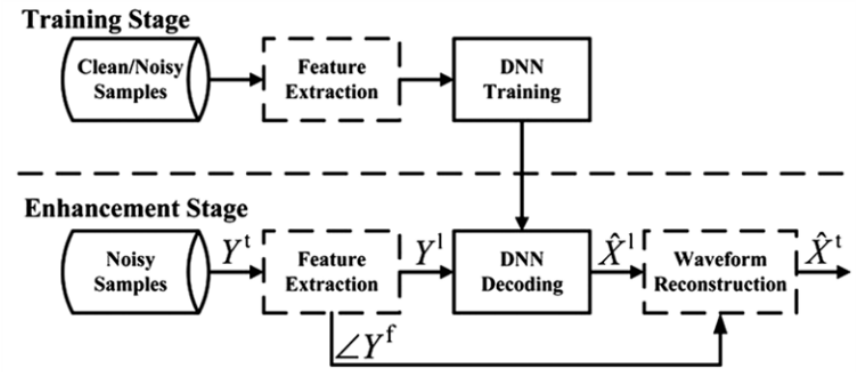

Fig. 1.4 基于谱映射的幅度谱算法框图[3]

是对数功率谱)和相位谱$\angle Y^f$。幅度谱Y^l送入训练好的DNN得到增强的幅度谱$\widehat{X}^l$。最后经过波形重构得到增强的时域波形$\widehat{X}^t$。

CRN-causal直接以STFT幅度谱作为输入特征和映射目标。为了方便转成C语言落地，这里的STFT采用以傅里叶基为卷积核的一维卷积的方式，而后利用$|X| = \sqrt{\mathcal{R}(X)^2 + \mathcal{I}(X)^2}$计算得到幅度谱，具体代码如下：

```
# scripts/utils/pipeline_modules.py lines 7-21
class NetFeeder(object):
    def __init__(self, device, win_size=320, hop_size=160):
        self.eps = torch.finfo(torch.float32).eps
        self.stft = STFT(win_size, hop_size).to(device)

    def __call__(self, mix, sph):
        real_mix, imag_mix = self.stft.stft(mix)
        mag_mix = torch.sqrt(real_mix**2 + imag_mix**2)
        feat = mag_mix

        real_sph, imag_sph = self.stft.stft(sph)
        mag_sph = torch.sqrt(real_sph**2 + imag_sph**2)
        lbl = mag_sph
        return feat, lbl
```

代码中$feat$和lbl分别作为网络的特征和训练目标用于对网络进行训练：

```
# scripts/utils/models.py lines 162-174
# forward + backward + optimize
optimizer.zero_grad()
with torch.enable_grad():
est = net(feat)
loss = criterion(est, lbl, loss_mask, n_frames)
loss.backward()
if self.clip_norm >= 0.0:
clip_grad_norm_(net.parameters(), self.clip_norm)
optimizer.step()
# calculate loss
running_loss = loss.data.item()
```

```
accu_tr_loss += running_loss * sum(n_frames)
accu_n_frames += sum(n_frames)
```

feat输入网络得到估计的谱est，而后计算est和lbl之间的损失反向传播优化网络参数。

解码过程同样采用NetFeeder类的实例feeder进行特征提取，将幅度谱特征feat送入训练好的网络得到估计的幅度谱est，最后经过波形合成过程resynthesizer即可得到增强语音：

```
# scripts/utils/models.py lines 337-348
feat, lbl = feeder(mix, sph)
with torch.no_grad():
    ...
    est = net(feat)
    ...
...
sph_est = resynthesizer(est, mix)
```

波形合成resynthesizer对应的类Resynthesizer按照

$$\begin{aligned} \mathcal{R}(\widehat{S}) &= |\widehat{S}| \cdot \cos(\angle X), \\ \mathcal{I}(\widehat{S}) &= |\widehat{S}| \cdot \sin(\angle X) \end{aligned} \tag{1.6}$$

得到增强的复谱，经过iSTFT生成增强的语音波形：

```
# scripts/utils/models.py lines 337-348
class Resynthesizer(object):
    def __init__(self, device, win_size=320, hop_size=160):
        self.stft = STFT(win_size, hop_size).to(device)

    def __call__(self, est, mix):
        real_mix, imag_mix = self.stft.stft(mix)
        pha_mix = torch.atan2(imag_mix.data, real_mix.data)
        real_est = est * torch.cos(pha_mix)
        imag_est = est * torch.sin(pha_mix)
        sph_est = self.stft.istft(torch.stack([real_est, imag_est
            ], dim=1))
        sph_est = F.pad(sph_est, [0, mix.shape[1]-sph_est.shape
            [1]])

        return sph_est
```

最后简单提一下网络结构，网络结构如图1.5所示，结合图1.5和代码可以看出，网络可以分为三部分：编码器、时序建模结构和解码器。编码器由5层二维卷积级联Batch Normalization和eLU激活函数组成；时序建模结构是两层LSTM；解码器由5层二维反卷积构成，除了最后一层反卷积层，其余反卷积层同样级联了Batch Normalization和eLU激活函数，最后一层反卷积层后则是Softplus激活函数以保证输出结果为非负数从而满足幅度谱的物理意义。幅度谱经过编码器被变换到高维嵌入式(embedding)空间，卷积核用于建模相邻帧和相邻频点之间的相关性。而后通道维和频率特征维被合并到一个维度，经过两层LSTM建模不同帧之间这组特征的时序关系，最后变回与编

码器输出相同的张量形状，经过解码器张成与输入幅度谱相同的尺寸，从而得到增强的幅度谱。编解码器之间采用concatenate形式的skip connection连接以缓解网络过深可能导致的梯度消失问题。可以看出，该网络架构与图像分割中的编解码器结构极其相似，这是不难理解的，因为以上两个任务都可以看做逐点(时频点/像素点)的滤波问题。然而二者仍有不同之处，相比于图片是一次快拍(借用阵列信号处理中的概念，可以认为是同一时刻)即可全部获得的，语谱图由于其中一个维度是时间帧，需要等待一句话结束后才能得到一张语谱图。在实时应用的推理阶段每次只能获得当前帧而未来帧的信息是未知的，因此需要一些技术来保证模型的因果性。这里的代码在编解码器中沿时间帧维度的截断和补零被使用以保证模型的因果性，而我们将在下一节对因果性进行更详细的讨论。

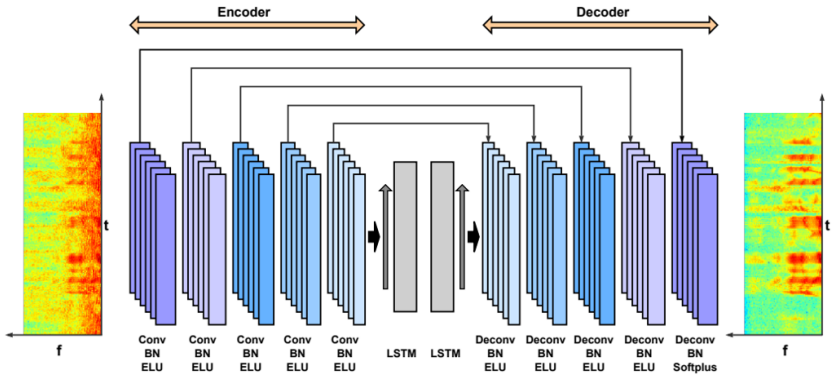

Fig. 1.5 CRN-causal框图[4]

```
# scripts/utils/networks.py
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()

        self.conv1 = nn.Conv2d(in_channels=1, out_channels=16,
            kernel_size=(2,3), stride=(1,2), padding=(1,0))
        self.conv2 = nn.Conv2d(in_channels=16, out_channels=32,
            kernel_size=(2,3), stride=(1,2), padding=(1,0))
        self.conv3 = nn.Conv2d(in_channels=32, out_channels=64,
            kernel_size=(2,3), stride=(1,2), padding=(1,0))
        self.conv4 = nn.Conv2d(in_channels=64, out_channels=128,
            kernel_size=(2,3), stride=(1,2), padding=(1,0))
        self.conv5 = nn.Conv2d(in_channels=128, out_channels=256,
             kernel_size=(2,3), stride=(1,2), padding=(1,0))

        self.lstm = nn.LSTM(256*4, 256*4, 2, batch_first=True)

```

```
14          self.conv5_t = nn.ConvTranspose2d(in_channels=512,
                out_channels=128, kernel_size=(2,3), stride=(1,2),
                padding=(1,0))
15          self.conv4_t = nn.ConvTranspose2d(in_channels=256,
                out_channels=64, kernel_size=(2,3), stride=(1,2),
                padding=(1,0))
16          self.conv3_t = nn.ConvTranspose2d(in_channels=128,
                out_channels=32, kernel_size=(2,3), stride=(1,2),
                padding=(1,0))
17          self.conv2_t = nn.ConvTranspose2d(in_channels=64,
                out_channels=16, kernel_size=(2,3), stride=(1,2),
                padding=(1,0), output_padding=(0,1))
18          self.conv1_t = nn.ConvTranspose2d(in_channels=32,
                out_channels=1, kernel_size=(2,3), stride=(1,2),
                padding=(1,0))
19
20          self.bn1 = nn.BatchNorm2d(16)
21          self.bn2 = nn.BatchNorm2d(32)
22          self.bn3 = nn.BatchNorm2d(64)
23          self.bn4 = nn.BatchNorm2d(128)
24          self.bn5 = nn.BatchNorm2d(256)
25
26          self.bn5_t = nn.BatchNorm2d(128)
27          self.bn4_t = nn.BatchNorm2d(64)
28          self.bn3_t = nn.BatchNorm2d(32)
29          self.bn2_t = nn.BatchNorm2d(16)
30          self.bn1_t = nn.BatchNorm2d(1)
31
32          self.elu = nn.ELU(inplace=True)
33          self.softplus = nn.Softplus()
34
35      def forward(self, x):
36
37          out = x.unsqueeze(dim=1)
38          e1 = self.elu(self.bn1(self.conv1(out)[:,:,:-1,:].
                contiguous()))
39          e2 = self.elu(self.bn2(self.conv2(e1)[:,:,:-1,:].
                contiguous()))
40          e3 = self.elu(self.bn3(self.conv3(e2)[:,:,:-1,:].
                contiguous()))
41          e4 = self.elu(self.bn4(self.conv4(e3)[:,:,:-1,:].
                contiguous()))
42          e5 = self.elu(self.bn5(self.conv5(e4)[:,:,:-1,:].
                contiguous()))
43
44          out = e5.contiguous().transpose(1, 2)
45          q1 = out.size(2)
46          q2 = out.size(3)
47          out = out.contiguous().view(out.size(0), out.size(1), -1)
48          out, _ = self.lstm(out)
49          out =out.contiguous().view(out.size(0),out.size(1),q1,q2)
50          out = out.contiguous().transpose(1, 2)
51
52          out = torch.cat([out, e5], dim=1)
```

```
53
54          d5 = self.elu(torch.cat([self.bn5_t(F.pad(self.conv5_t(
                out), [0,0,1,0]).contiguous()), e4], dim=1))
55          d4 = self.elu(torch.cat([self.bn4_t(F.pad(self.conv4_t(d5
                ), [0,0,1,0]).contiguous()), e3], dim=1))
56          d3 = self.elu(torch.cat([self.bn3_t(F.pad(self.conv3_t(d4
                ), [0,0,1,0]).contiguous()), e2], dim=1))
57          d2 = self.elu(torch.cat([self.bn2_t(F.pad(self.conv2_t(d3
                ), [0,0,1,0]).contiguous()), e1], dim=1))
58          d1 = self.softplus(self.bn1_t(F.pad(self.conv1_t(d2),
                [0,0,1,0]).contiguous()))
59
60          out = torch.squeeze(d1, dim=1)
61
62          return out
```

1.3 语音增强算法的因果性

语音增强技术根据其应用的不同可以分为实时系统和非实时系统两大类。助听辅听、耳机手机通话等是典型的实时应用场景，其中的语音增强算法必须在使用较少的未来信息甚至不使用未来信息的情况(即模型是因果的)下完成处理，同时，算法在相应的硬件设备上的计算时间也应满足实时要求。需要注意的是，这里的实时是一个相对的概念，绝大多数情况的语音增强并不要求逐采样点的实时处理，而是指在较短延时内完成处理，而对于不同的应用，能容忍的延时也各不相同。由于不同硬件平台的算力不同，更常见的是用模型的因果性代替对其实时性的评价，并使用参数量和计算量作为是否能够在某个硬件上实时处理的参考指标。这是由于语音增强的应用场景是如此广泛、模型压缩技术仍有对模型效率改进的可能、以及芯片等计算资源的迅猛发展，因此即使某个提出的语音增强模型无法应用在某个平台上，该模型可能仍不失其价值。

> 语音增强处理延时的定义并不止一种，这里给出一种常见的定义：对于采用重叠相加(overlap-add)方法重构的频域语音增强算法，时延由OLA带来的时延(合成窗长)、频域变换带来的时延(帧移)以及算法接收的未来信息带来的时延三部分组成，另外算法的处理时间应不大于帧移。

为了保证模型比较的公平性以确定模型中每个模块的有效性，模型因果与否是应该明确的。然而，很遗憾的是，社区内常出现工作有意无意地忽视了这一问题，从而误导了社区对于某种特征/模块/损失函数的认识。因此，作为一本语音增强导读，有必要将这部分单列一节进行说明。

这一话题无疑是必要但敏感的，因此这里不得不表明个人态度。其一，本节的立意是尽量澄清一些社区内的误区、尽量减少初学者所走的弯路、尽量提升初学者对文章和开源代码的判断能力，而非否定以往的和未来的任何工作。其二，尽管一些工作并不严格满足因果处理，但与因果处理保持一致的baselines比较时大约是能够保证其创新部分的有效性的。另一方面，尽管个人由于不希望看到语音前端陷入到和计算机视觉一样门槛大幅度降低、滥竽充数者比例渐高、工业界多方面压制学术界的状况，因此对开源并不持支持态度。但无疑开源者是对社区发展有着推动和帮助作用的，他们的贡献是巨大的。窃以为不宜以这种很可能无意的失误而使贡献者受到责难。我也反对在任何高速发展的学科因为小的技术失误而全面否定整个工作的行为，这对学科和个人的发展是有负面作用的(很不幸近年来这种趋势似乎是明显的)。更何况实时语音增强中大量工作并不因这一问题而失去其价值，而且语音增强在非实时场景同样也有广泛应用。

1.3.1 语音音量的归一化

由于说话人本身音量以及说话人距传声器距离远近的差异，语音音量大小的动态范围通常很大。为了保证语音增强模型能够处理不同音量的语音，对训练数据的处理通常是重要的。社区内常见的两种非因果操作是对训练和测试过程中的带噪语音和纯净语音进行能量均方根归一化(即假设语音经过了理想的自动增益控制(AGC))、以及利用整段带噪音频计算均值和方差后用作均值方差归一化(这种方式既有利用训练集的均值方差对测试语料做归一化的，又有直接测试语料计算均值和方差做归一化的，显然后者是非因果的)。以这种方式压缩了数据的动态范围，方便网络优化。替代这类非因果操作的方式有两种：一种是对输入特征进行在线归一化、另一种是在损失端进行归一化。

前者常见的有在线频率依赖的均值方差归一化，当前帧的均值方差均以指数衰减平滑的形式估计得到：

$$\begin{aligned} \mu(t,f) &= \alpha\mu(t-1,f) + (1-\alpha)|X(t,f)|, \\ \sigma^2(t,f) &= \alpha\sigma^2(t-1,f) + (1-\alpha)|X(t,f)|^2, \\ |X^{norm}(t,f)| &= \frac{|X(t,f)| - \mu(t,f)}{\sqrt{\sigma^2(t,f) - \mu^2(t,f)}}. \end{aligned} \tag{1.7}$$

上述关系式代码为：

```
# https://github.com/GuillaumeVW/NSNet/blob/master/dataloader/wav_dataset.py lines 85 - 102
# mean normalization
frames = []
x_lps = x_lps.transpose(0, 1)
```

```
n_init_frames = self.n_init_frames
alpha_feat_init = self.alpha_feat_init
alpha_feat = self.alpha_feat
for frame_counter, frame_feature in enumerate(x_lps):
    if frame_counter < n_init_frames:
        alpha = alpha_feat_init
    else:
        alpha = alpha_feat
    if frame_counter == 0:
        mu = frame_feature
        sigmasquare = frame_feature.pow(2)
    mu = alpha * mu + (1 - alpha) * frame_feature
    sigmasquare = alpha * sigmasquare + (1 - alpha) *
        frame_feature.pow(2)
    sigma = torch.sqrt(torch.clamp(sigmasquare - mu.pow(2), min=1
        e-12))  # limit for sqrt
    norm_feature = (frame_feature - mu) / sigma
    frames.append(norm_feature)
```

此外，其他累积归一化方式和可学习归一化方式也被探索。

后者则是将语音能量均方根归一化的操作转移到损失端，整句语音的能量仅在训练阶段被使用，因此模型是因果的。这类方法被用于复数域语音增强模型，其损失函数被定义为[5]:

$$\mathcal{L} = \frac{1}{\sigma}(\alpha \sum_{t,f} |S - \widehat{S}|^2 + (1-\alpha) \sum_{t,f} ||S| - |\widehat{S}||^2), \tag{1.8}$$

其中σ是指经过语音活动检测(VAD)后的语音能量。一般同时还会在训练阶段对语料音量以某种随机分布随机缩放，如在训练阶段将纯净语音和噪声的音量都服从均值为-26 dBFS，方差为10 dB的高斯分布进行随机缩放[5]，类似操作的代码见于DPCRN3:

```
# data_loader.py lines 163 - 192
gain = np.random.normal(loc=-5,scale=10)
gain = 10**(gain/10)
gain = min(gain,3)
gain = max(gain,0.01)
...
batch_clean[i,:] = clean_s * gain
batch_noisy[i,:] = noisy_s * gain
```

References

1. Y. Xia, et. al. *Weighted Speech Distortion Losses for Neural-network-based Real-time Speech Enhancement*, ICASSP 2020.
2. X. Hao, et. al. *Masking and Inpainting: A Two-Stage Speech Enhancement Approach for Low SNR and Non-Stationary Noise*, ICASSP 2020, pp. 6959–6963. doi: 10.1109/ICASSP40776.2020.9053188.

3. Y. Xu, et. al. *An Experimental Study on Speech Enhancement BasedonDeepNeuralNetworks*, IEEE Signal Processing Letters, 21(1), pp. 65-68, 2014.
4. K. Tan, et. al. *A convolutional recurrent neural network for real-time speech enhancement*, INTERSPEECH 2018, pp. 3229-3233.
5. S. Braun, et. al. *Effect of noise suppression losses on speech distortion and ASR performance*, arXiv:2111.11606.