

# 包教包会！从零实现基于Transformer的语音识别(ASR)模型🤖



晚安汤姆布利多  
做纯粹的知识分享者

199 人赞同了该文章

关注他

## 关于作者



晚安汤姆布利多  
做纯粹的知识分享者

回答

9

文章

10

关注者

1,996

关注他

发私信

起

## 前言

在文章[三万字最全解析！从零实现Transformer（小白必会版）](#)中，我们从零开始搭建了一个Transformer模型，详细讲解了Transformer的输入输出格式、注意力机制、位置编码、掩码机制等等，并基于上述知识从零搭建了Transformer的编码器和解码器，但是那篇文章重点在于拆解Transformer的实现方式，而不是其具体应用。

在本文中，我们尝试基于Transformer去搭建一个端到端<sup>[1]</sup>的语音识别 (Automatic Speech Recognition, ASR) 模型，详细拆解数据预处理、模型前向计算、损失函数计算和模型参数更新、模型推理等各个阶段的具体做法。通过本文，相信读者可以对Transformer如何用于序列转换任务<sup>[2]</sup> (e.g. 语音序列到文本序列) 有一个更清晰的认识。

本文所有代码已开源，包含数据预处理的代码、训练和测试的代码、训练log、测试的文本结果、模型checkpoint等，可以直接训练以及推理。链接如下， Have fun!

<https://github.com/xiabingquan/Automatic-Speech-Recognition-from-Scratch>  
[github.com/xiabingquan/Automatic-Speech-Recogniti...](#)

该github仓库相比于本博客做了部分改进，最终代码以该github链接为准。

码字不易，文章的图基本都是自己手画的，内容也是原创，大家记得一键三连！

文章较长，加之博主水平有限，如有谬误欢迎指正。

相关前置知识：Python、PyTorch、Transformer基础、深度学习相关基础。

本文共约2.9w字，阅读需约72分钟，建议PC端阅读。

本文的基本结构如下：（PC端点击左侧目录可直接跳转到对应章节）

- 语音识别任务简介：简单介绍语音识别任务
- 语音识别模型整体框架：介绍模型的整体框架
- 数据预处理：介绍如何对音频和文本数据进行预处理
- 前向传播：介绍模型的前向传播过程，包括损失函数的计算
- 加载数据：介绍训练时如何高效加载数据
- 训练代码：上述部分的整合，给出训练阶段的主要代码
- Greedy Search简介：简单介绍Greedy Search解码技巧
- 推理代码：推理阶段的主要代码
- 案例：LRS2

## 语音识别任务简介

赞同 199

62 条评论

分享

喜欢

收藏

申请转



我们从介绍语音识别任务 (Automatic Speech Recognition, ASR) 开始。

语音识别，顾名思义，就是指**通过说话人的语音来判断其说话内容**。语音识别目前已经进入大规模商用阶段，相信大家都不陌生，比如手机端常用的AI助手、语音输入等功能都离不开语音识别。



图一. 语音识别简图 [https://developer-blogs.nvidia.com/wp-content/uploads/2019/12/automatic-speech-recognition\\_updated.png](https://developer-blogs.nvidia.com/wp-content/uploads/2019/12/automatic-speech-recognition_updated.png)

语音识别技术具有非常悠久的历史，可以追溯到20世纪50年代<sup>[3]</sup>，不过当时主要是一些早期的探索。

20世纪80年代以来，研究人员引入隐马尔可夫链模型 (Hidden Markov Model, HMM)<sup>[4]</sup> 来完成语音识别任务，取得了非常显著的进展。基于HMM的语音识别系统主要采用级联式的建模方式，整个系统包括特征预处理、声学模型、语言模型等多个部分，由于这部分并不是本文的主体内容，本文不做详细介绍，有兴趣的读者可以参考论文

<https://www.cs.ubc.ca/~murphyk/Software/HMM/E6820-L10-ASR-seq.pdf>  
🔗 [www.cs.ubc.ca/~murphyk/Software/HMM/E6820-L10-...](http://www.cs.ubc.ca/~murphyk/Software/HMM/E6820-L10-...)

一作Kai-Fu Lee就是大家所熟知的李开复

深度学习时代以来，基于神经网络的端到端语音识别技术逐渐取代了原有的基于HMM的级联式技术<sup>[5]</sup>。相比于级联式而言，端到端的建模方式更加简单直观，同时也可以减少级联式存在的误差累积等问题，因此已经成为了深度学习时代主流的语音识别建模范式，本文介绍的也是**端到端**的基于交叉熵损失函数 (Cross Entropy) 的语音识别模型。除了本文用的Cross Entropy以外，语音识别中也有一些其他常用的损失函数，比如RNN-T、CTC等。

注：关于“端到端”，简单来说就是**用一个模型完成语音识别的整个过程**，即直接以语音为输入、文本为输出。而级联式的做法是，先用一个模型A完成语音 $\rightarrow x$ ，再用模型B完成 $x \rightarrow y$ ，最后再用模型C完成 $z \rightarrow$  文本，级联式的好处是A、B、C可以分开单独训练，但是坏处在于存在误差累积的问题，模型整体的准确率等于A、B、C三部分准确率的乘积，而端到端则不存在误差累积的问题，可以直接对整个系统进行优化。

## 语音识别模型整体框架

端到端的语音识别模型的整体框架图如下图所示。

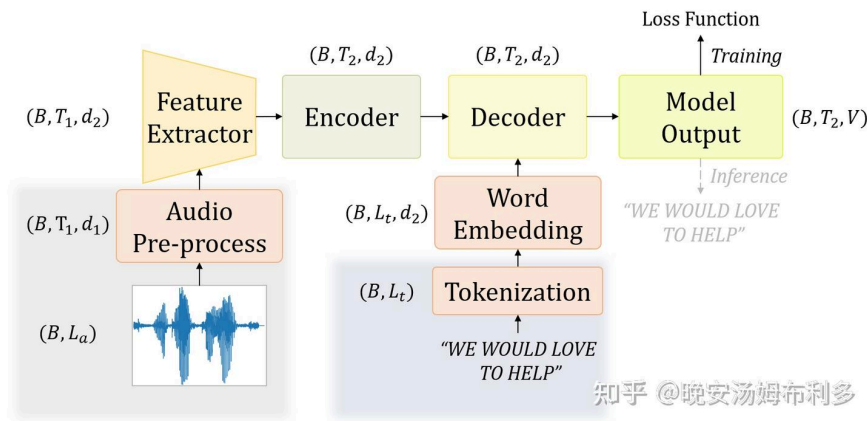


图2. 语音识别模型整体框架图

其中，左下角的浅灰色部分为音频预处理，右下角的浅蓝色部分为文本预处理，而上面的部分则为模型的主干以及输出的后处理（训练时计算loss，测试时输出文本）。为便于读者理解，每一部分旁边还标出了当前部分对应的数据的维度

训练时的整个过程大致可以分为以下几个步骤：

1. 对音频进行预处理，然后将预处理之后的音频作为模型输入。模型输入经过一个特征提取器提取基础特征，然后经过encoder进一步抽取特征；
2. 对文本进行预处理，得到离散的token，离散的token经过word embedding，得到数值连续的张量，并将该张量和第1步中encoder的输出一起送入decoder；
3. encoder前向传播；
4. decoder前向传播，计算损失函数；
5. 反向传播计算梯度，更新模型参数。

接下来上述每一部分的细节进行讲述。

## 数据预处理

语音识别模型的主体部分为Transformer的encoder和decoder，但是在前文中我们提到，Transformer的encoder和decoder的输入都为  $(B, T, d)$  形式的三维张量，其中  $B$  是 batch-size，即一个batch的样本个数， $T$  为序列的长度， $d$  为特征的维度。然而，我们音频的原始数据格式为一些一维的离散的采样点，文本的原始数据格式为一串字符。显然，模型无法直接处理这些原始形式的音频和文本。因此在将音频和文本送入encoder/decoder之前，我们需要先分别对音频和文本进行预处理。

注：关于音频的说明，假设音频的采样率为16000Hz，时长2s，那么该音频读进来之后就是(32000,)的一维张量。

### 音频预处理：提取FBank特征<sup>+</sup>

首先是音频的预处理，我们基于torchaudio进行音频数据的读取和预处理，torchaudio是和PyTorch配套的一个处理音频的第三方Python库。

前面提到，原始音频是  $(L_a,)$  形式的一维张量（其中  $L_a$  为音频中采样点的总个数），而Transformer模型的输入尺寸为  $(T, dim)$ （不考虑batch维度），也就是说需要对原始音频进行维度上的转换。目前主流的有两种方法。

- 先提取原始音频的比较底层的人工特征，比如MFCC、FBank等（这一步可直接计算，不需要过神经网络），再经过一个特征提取器再次提取特征，最后送入Transformer的encoder；
- 直接用一个特征提取器对原始音频进行特征提取，然后送入Transformer encoder。

这两种方法无所谓好坏，本文用第一种。给定一个原始音频，首先用torchaudio读取音频，然后计算其FBank特征，FBank特征是一种人为定义的特征，详情请参考wiki，当然也可以用其他特征，比如MFCC等。代码如下：

```
import torchaudio

# wav: (L_a,)
wav, sr = torchaudio.load(wav_path)
wav = wav * (1 << 15) # rescale to int16 for kaldi compatibility
# feats: (T_1, 80)
feats = torchaudio.compliance.kaldi.fbank(wav, num_mel_bins=80)
feats = feats.unsqueeze(0) # (1, seq_len, 80)
```

上述feats就是最后送入feature extractor的张量，feature extractor的细节在后文的前向传播一节进行阐述。

文本预处理：tokenization

音频预处理结束之后，再来看文本的预处理。

首先我们介绍tokenization的概念。在文本相关的序列任务中，tokenization简单来说就是将字符形式的文本转变为一个个离散的数字，比如"WE WOULD LOVE TO HELP" -> [1, 2, 3, 4, 5]，其中"WE"对应于1，"WOULD"对应于2，"LOVE"对应于3，"TO"对应于4，"HELP"对应于5，我们把[1, 2, 3, 4, 5]称为tokens。

实际的tokenization过程通常要比上面的例子稍微复杂点，但是基本思路是一样的，都是建立文本和数字之间的——映射关系，从而将字符串形式的文本变成计算机可以处理的数字。

根据建模单元（即最小的建模单位）的不同，语音中常用的tokenization方式大致有三类：char-based, word-based和subword-based，三者之间的区别如下图所示。

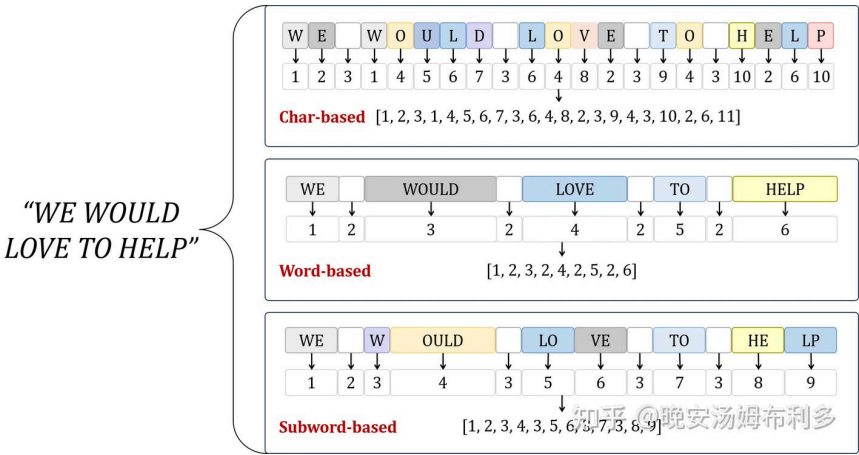


图3. 不同tokenization方式的对比

其中，

**char-based:** 如上图最上面的框所示。即将单个字母映射为一个数字，比如'W' -> 1, 'E' -> '2', ' ' (空格) -> 3等等。这样建模的好处是我们用简单的26个字母（不考虑大小写）和空格、标点等特殊符号就可以表示任意英语内容，词表非常小，但是坏处也显而易见，那就是一句话对应的token数会特别多，比如"WE WOULD LOVE TO HELP"这句话，总共有21个字母（算上空格），那么也会得到21个token。当文本较长时，以char为建模单元的话token序列会特别长，而Transformer的计算复杂度又是随着序列长度平方级上涨，计算成本会非常高。

**word-based:** 如上图中间的框所示。考虑到char-based的token序列长的弊端，一个自然而然的替代办法是以word为建模单元，比如"WE" -> 1, "WOULD" -> 2, "LOVE" -> 3，那么上面那句话用9个token（5个单词，4个空格）就可以表示。这样做以后，token序列倒是短了，但是又引来了一些其他问题。一个最直观的问题就是英语单词每一个单词我们都必须用一个不同的数字来表示。但是，很多单词

如LOVE, LOVER, LOVED这三个词, 明明只差了一个字母, 我们却必须用三个完全不同的token来表示它们, 这显然是不划算的。同时, 以word为建模单位也很容易出现OOV (Out of Vocabulary) 的问题, 比如某些专业名词很可能不在词表里面 (点名批评pneumonoultramicroscopicsilicovolcanoconiosis), 那么这个时候这个单词就完全没法用现有词表来表示, 这显然不是我们所希望的。

**subword-based**<sup>[6]</sup>: 如上图最下面的框所示。简单来说, char-based的缺点在于token序列长, word-based的缺点在于词表大, 而且容易OOV。那么, 有没有什么办法可以兼顾两者的优点呢? 这就是这里要介绍的subword-based, subword直译为子词, 即单词中的某个片段, 是介于字符和单词之间的一个建模单位。根据子词的构建规则, subword可以分为WordPiece、Byte-Pair Encoding (BPE)、Unigram等等。

比如transition这个单词, 该单词含有trans和tion两个子词, 而trans和tion都是英语中非常常见的组合, 比如transparent, transport, transgender, station, information等等, 于是, 我们在对transition做tokenization时, 完全可以将其拆分为[trans, i, tion], 这样, 我们用3个token就可以表示该单词, token数比char-based更少, 而且trans和tion这两个subword在其他词中也经常出现, 可以复用, 因此词表大小会比word-based小。而对于"WE WOULD LOVE TO HELP"这句话, 当以subword作为建模单位时, 我们将其拆分成["WE", ' ', "W", "OULD", ' ', "LO", "VE", ' ', "TO", ' ', "HE", "LP"] (这个拆分仅用作示例, 不代表实际情况)。

总结起来, subword有以下优点:

- 相比于char而言, token数更少, 更节约计算成本;
- 相比于word而言, 词表更小, 不容易出现OOV的问题。词表更小这一点我们在上面的例子已经解释过了, 简单来说就是我们不用像word那样每一个单词都用一个唯一的token来表示。而不容易出现OOV是因为, subword对应词表通常会包含不同粒度的子词, 从最小粒度的a-z字母, 到最大粒度的单词。在极端情况下, 如果碰到无法用词表中的子词来表示的单词, 那么至少还可以用a-z的字母来表示, 比如生僻词pneumonoultramicroscopicsilicovolcanoconiosis -> [p, n, e, u, ., s, i, s]。
- 可以通过人为控制词表大小来平衡计算成本和OOV。比如, 当我们允许词表非常非常大时, 那么此时完全可以将单个word作为一个token, 此时和word-based等价; 当词表非常非常小时, 比如仅包含字母a-z和一些特殊符号, 由于词表过小, 在做tokenization的时候需要把整个词拆成单个单个的字母, 此时就和char-based等价。

得益于subword的上述优点, 目前subword基本是众多语音或者自然语言处理任务中处理文本时最常用的建模单位。不过为了简单起见, 本文直接以char作为建模单位。subword的更多内容请读者自行查阅: [A comprehensive guide to subword tokenisers](#)、[SentencePiece](#)等。

下文给出了char-based tokenizer的完整代码, 本文只考虑a-z和空格这27个字符。对于词表之外的字符, 我们选择直接跳过 (实际操作时不建议这样粗暴地处理)。

下述代码中<sos>和<eos>为两个特殊token, 后文会讲解其用途。

```
# coding=utf-8
# Contact: bingquanxia@qq.com

class CharTokenizer(object):
    def __init__(self, do_lower_case=True):
        self.do_lower_case = do_lower_case
        self.char2idx = {
            "<sos>": 0, "<eos>": 1,
            'a': 2, 'b': 3, 'c': 4, 'd': 5, 'e': 6, 'f': 7, 'g': 8,
            'h': 9, 'i': 10, 'j': 11, 'k': 12, 'l': 13, 'm': 14, 'n': 15,
            'o': 16, 'p': 17, 'q': 18, 'r': 19, 's': 20, 't': 21,
            'u': 22, 'v': 23, 'w': 24, 'x': 25, 'y': 26, 'z': 27, ' ': 28
        }
        self.idx2char = {v: k for k, v in self.char2idx.items()}
        self.vocab = len(self.char2idx)
        self.sos_id = self.char2idx['<sos>']
        self.eos_id = self.char2idx['<eos>']
        self.skipped = set()
```

```

def tokenize(self, text):
    if self.do_lower_case:
        text = text.lower()
    remained = [char for char in text if char in self.char2idx]
    skipped = [char for char in text if char not in self.char2idx]
    if len(skipped) > 0:
        for s in skipped:
            if s not in self.skipped:
                print(f"Skipped character: {s}")
                self.skipped.add(s)
    return [self.char2idx[char] for char in remained]

def detokenize(self, token_ids):
    remained = [d for d in token_ids if d in self.idx2char]
    skipped = [d for d in token_ids if d not in self.idx2char]
    if len(skipped) > 0:
        print(f"Skipped token ids: {skipped}")
    return ''.join([self.idx2char[d] for d in remained])

if __name__ == '__main__':
    tokenizer = CharTokenizer()
    print(tokenizer.tokenize('hello world'))
    # output: [9, 6, 13, 13, 16, 28, 24, 16, 19, 13, 5]
    print(tokenizer.detokenize([9, 6, 13, 13, 16, 28, 24, 16, 19, 13, 5]))
    # output: 'hello world'

```

## 前向传播

在对音频和文本进行预处理之后，音频的数据形式为 (B,T<sub>1</sub>,d<sub>1</sub>)（其中 d<sub>1</sub>=80，可以不同调整），文本的数据形式为 (L,t)，为原始文本经过tokenization之后的token序列。

在本节中，我们将详细拆解图2中的整个模型的前向传播部分。该模型的前向传播大体可以分为三部分：特征embedding、encoder和decoder。接下来分别讲述这三部分的细节。

## 特征embedding

特征embedding并不是一个专有名词，本文用来代指从数据预处理到输入encoder/decoder之前的这部分流程，即音频的Feature Extractor和文本的Word Embedding\*这两部分。

首先来看音频，得到FBank特征之后，我们用一个基于卷积的特征提取器来进一步提取特征。特征提取器的选取多种多样，比如可以用一维的ResNet，也可以直接把多个卷积堆叠起来（比如wav2vec2.0）等等，简单起见，本文直接用两个线性层。

```

class LinearFeatureExtractionModel(nn.Module):
    def __init__(self, in_dim: int, out_dim: int):
        super().__init__()
        self.linear = nn.Sequential(
            nn.Linear(in_dim, out_dim),
            nn.ReLU(inplace=True),
            nn.Linear(out_dim, out_dim)
        )

    def forward(self, x, x_lens):
        return self.linear(x), x_lens

fbank_dim = 80
enc_dim = 256
feature_extractor = LinearFeatureExtractionModel(in_dim=fbank_dim, out_dim=enc_dim)

```



```
# feats: (B, T_1, 80)
output = feature_extractor(feats)
# output: (B, T_2, 256)
```

上述output就是encoder的输入。

接下来看文本的特征embedding。关于文本，我们通过对原始文本进行tokenization，已经得到了token序列，比如[1, 2, 3, 4, 5, 6...], 那么，怎么将这些离散的token变为连续的特征呢？或者换个角度问，怎么将一串输入文本映射为可以代表文本的语音信息的向量表征？关于这一点，NLP领域有不少探索，比如Word2vec、GloVe等等。当前，通常直接用一个随机初始化的embedding来表示一个token，然后在训练整个模型的过程中端到端地优化每个token对应的embedding，代码如下：

```
import torch
import torch.nn as nn

# Define the vocabulary size and embedding dimension
vocab = 29 # the size of vocabulary
embedding_dim = 512 # the dimension of each embedding

# Create an instance of the Embedding module
embedding = nn.Embedding(vocab, embedding_dim)

# Input tensor with token indices
input_tensor = torch.LongTensor([[1, 2, 3, 4, 5], [6, 7, 8, 9, 10]])

# Perform word embedding
embedded_tensor = embedding(input_tensor)

# Print the embedded tensor
print(embedded_tensor.shape)
# (2, 5, 512)
```

## encoder

到目前未知，我们已经完成了文本和音频数据各自的预处理和特征的embedding，下面到了encoder的部分，encoder的作用是，对输入的特征进行进一步抽象，得到较高层的、具有更加明显的语义信息的特征。这么描述可能有点抽象，简单来说音频和特征提取器和encoder没有本质不同，目的都是为了对原始音频进行特征提取，只是前文的音频特征提取器更加侧重于局部特征的提取，而encoder更加侧重于具有上下文信息的、全局特征的提取，功能更加强（因为encoder的主要构成模块为注意力层）。

接下来给出整个encoder的代码。encoder的代码仍然基于之前介绍Transformer的文章

[晚安汤姆布利多：三万字最全解析！从零实现Transformer（小白必会版）](#)

此处只给出调用代码，不再介绍模型细节，如下所示：

```
import torch
import torch.nn as nn

dropout_emb = 0.1 # dropout rate of the feature embedding
dropout_posffn = 0.1 # dropout rate of position-wise feed-forward
dropout_attn = 0. # dropout rate of attention
num_layers = 6 # number of encoder layers
enc_dim = 256 # dimension of the encoder
num_heads = enc_dim // 64 # number of attention heads
dff = 2048 # dimension of feed-forward network
max_seq_len = 4096 # maximum length of the target sequence

encoder = Encoder(
    dropout_emb=dropout_emb, dropout_posffn=dropout_posffn,
    num_layers=num_layers, enc_dim=enc_dim, num_heads=num_heads,
```

```

    dff=dff, tgt_len=max_seq_len,
)

# audio_features: (batch_size, seq_len, feature_dim)
b, max_feat_len, feat_dim = 2, 100, 256
# batched audio feature sequence
audio_features = torch.randn(b, max_feat_len, feat_dim)
# the length of each audio feature sequence
audio_lens = torch.tensor([90, 100])
device = audio_features.device
# the mask for audio feature sequence
enc_mask = get_len_mask(b, max_feat_len, audio_lens, device)

# enc_output: (batch_size, seq_len, enc_dim), the same size as audio_features
encoder_output = encoder(audio_features, audio_lens, enc_mask)
print(encoder_output.shape)          # torch.Size([2, 100, 512])

```

encoder\_output 就是encoder的输出，之后会送入decoder。

## decoder

encoder部分较为简单，直接将音频特征作为输入然后计算即可，但是decoder部分稍微复杂点。

关于decoder的具体实现请参考之前介绍Transformer的文章，这里主要是讲解模型decoder如何处理文本。原理解之后，代码反而比较简单。

在介绍decoder之前，我们先对整个**语音识别任务的形式化**进行简单介绍。

语音识别：给定音频  $X$ ，我们希望计算  $\mathop{\text{argmax}}\limits_{Y \in \mathcal{Y}} P(Y|X)$ ，其中  $\mathcal{Y}$  表示所有可能的文本内容的集合， $Y$  表示使得条件概率  $P(Y|X)$  最大的那个文本句子。简而言之，语音识别的目标就是**在所有可能的文本中找到音频对应的那个文本**。

但是，通过上述式子，我们暂时还没法显式地构建起对应的数学模型，更没法进行模型的训练，接下来对上面的式子进行进一步拆解，我们将其分为训练和测试两个阶段。

**训练阶段：**音频  $X$  和文本  $Y$  都已知。给定音频  $X$  和对应的文本内容  $Y=[y_1, y_2, \dots, y_n]$ ，根据**最大似然准则**，训练阶段的目标是最大化条件概率  $P(Y|X)$ 。根据链式法则，该条件概率可以拆分为如下形式：

$$\begin{aligned} P(Y|X) &= P(y_1, \dots, y_n|X) \quad \&= P(y_1|X)P(y_2|X, y_1)P(y_3|X, y_1, y_2) \\ &\quad \cdots P(y_n|X, y_1, \dots, y_{n-1}) \quad \&= P(y_1|X) P(y_n|X, y_1, \dots, y_{n-1}) \\ &= \prod_{i=1}^n P(y_i|X, y_1, \dots, y_{i-1}) \end{aligned}$$

仍然以句子"WE WOULD LOVE TO HELP"举例。简单起见，仍然用  $X$  表示这句话对应的音频，

那么，在以word作为token的建模单位时，上述概率  $P(Y|X)$  可以表示为

$$\begin{aligned} p_1 &= P(\text{``WE''}|X) \quad p_2 = P(\text{``WOULD''}|X, \text{``WE''}) \\ p_3 &= P(\text{``LOVE''}|X, \text{``WE''}, \text{``WOULD''}) \quad p_4 = P(\text{``TO''}|X, \text{``WE''}, \\ &\quad \text{``WOULD''}, \text{``LOVE''}) \quad p_5 = P(\text{``HELP''}|X, \text{``WE''}, \text{``WOULD''}, \text{``LOVE''}, \text{``TO''}) \\ P(Y|X) &= \prod_{i=1}^5 p_i \end{aligned}$$

也就是说，在文本序列的每一个位置，我们**根据历史token（包括当前token）以及音频，预测下一个token**，然后来计算损失函数。

**测试阶段：**测试阶段和训练阶段最大的不同在于测试阶段我们只有音频，完全没有文本，而测试阶段的任务也正是需要从音频中解码 (decoding) 出对应的文本。其实，测试阶段模型的行为和训练阶段有类似之处。具体来讲，训练阶段模型是根据历史token和音频来预测下一个token，从而计算损失函数，而测试阶段模型是根据历史token和音频来预测下一个token，再将预测的token纳入历史token，再重复预测的过程，直至解



"从音频中解码出文本"本身是一个很大的话题，对应非常多的解码技巧，比如Viterbi algorithm, Beam Search, Greedy Search, 加权有限状态转换器 (Weighted Finite-State Transducer, WFST) 等等。简单起见，本文使用最简单的Greedy Search，上文关于测试阶段的行为描述也仅适用于Greedy Search。

虽然我们给出了语音识别任务的形式化定义，但是细心的读者可能会发现，上面的内容还存在一些没有解决的问题，主要有下面两点：

1. 在训练阶段，"根据历史token以及音频预测下一个token"对应的数学公式是  $P(y_i | x, y_1, \dots, y_{i-1})$ ，这个很好理解。但是，假设当前token是第一个token呢？即  $P(y_1 | x)$  这一项，这个时候应该怎么计算？
2. 在测试阶段，由于我们不知道真实文本是什么，在根据历史token以及音频预测下一个token时，我们怎么知道什么时候停止解码？

为了回答上述两个问题，我们需要对公式  $P(Y|X)$  进行以下两点改动，分别对应于上面的1和2两点问题

1. 在第一个token  $y_1$  前面加入一个特殊token  $\text{< sos >}$
2. 在最后一个token  $y_n$  前面加入一个特殊token  $\text{< eos >}$

其中， $\text{< sos >}$  为"start of sentence"的缩写，表示输入文本的起始位置，而  $\text{< eos >}$  表示"end of sentence"的缩写，表示文本的结束位置。加入  $\text{< sos >}$  和  $\text{< eos >}$  之后，训练阶段的示意图可以用下图图4表示。

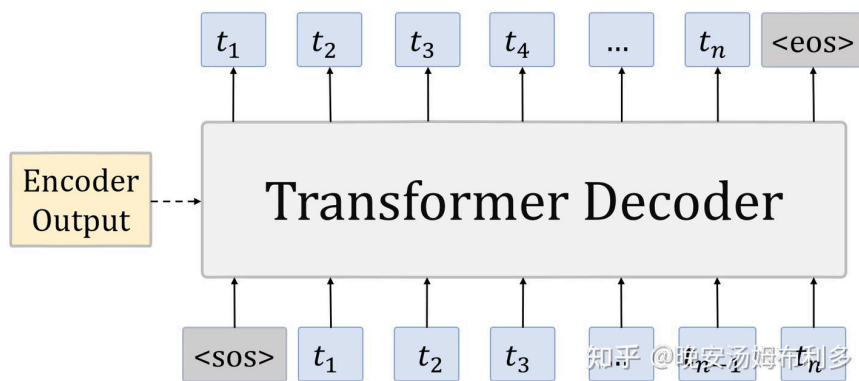


图4. 加入sos和eos的decoder训练阶段示意图

在公式  $P(Y|X) = P(y_1 | x) P(y_2 | x, y_1) \dots P(y_n | x, y_1, \dots, y_{n-1})$  中，如果我们将  $\text{< sos >}$  和  $\text{< eos >}$  分别记为  $y_0$  和  $y_{n+1}$ ，并在上述公式中加入起始和结束位置的预测，那么上述公式可以简化为如下形式：

$$\begin{aligned} P(Y|X) &= P(y_1 | x, y_0) P(y_{n+1} | x, y_0, y_1, \dots, y_{n-1}) \\ &\prod_{i=2}^{n-1} P(y_i | x, y_0, y_1, \dots, y_{i-1}) \end{aligned}$$

得到  $P(Y|X)$  之后，我们可以计算负对数似然 (Negative Log Likelihood, NLL) 损失函数：

$$\text{NLL} = -\log P(Y|X) = -\sum_{i=1}^n \log P(y_{i+1} | x, y_0, \dots, y_i)$$

最小化上述 NLL 就是模型训练阶段的优化目标。值得注意的是，虽然 NLL 的形式里有  $\sum$ ，看起来需要串行执行，但是Transformer可以通过其掩码机制实现并行计算，一次forward pass即可得到所有用于计算 NLL 的 P，这也是Transformer相比于RNN家族的一个巨大优势。

在测试阶段，由于在训练时模型已经可以根据  $[\text{< sos >}, y_1, \dots, y_n]$  来预测  $\text{< eos >}$ ，因此，当模型解码出  $\text{< eos >}$  时，我们就停止解码。

从图4中可以看出，文本的输入为  $[\text{< sos >}, y_1, \dots, y_n]$ ，而输出的预测目标为  $[y_1, \dots, y_n, \text{< eos >}]$ ，两者有所不同，我们分别用  $y_{\text{input}}$  和  $y_{\text{target}}$  表示。

```
tokens = tokenizer.tokenize(sentence)
ys_in = [sos] + tokens
ys_out = tokens + [eos]
ys_out = torch.tensor(ys_out).long()
ys_in = torch.tensor(ys_in).long()
```

接下来, 终于给出forward部分的完整代码了

## 完整代码

各模块的定义省略, 如LinearFeatureExtractionModel、Encoder、Decoder等

```
# -----Define constants-----
dropout_emb = 0.1          # dropout rate of the feature embedding
dropout_posffn = 0.1       # dropout rate of position-wise feed-forward
dropout_attn = 0.          # dropout rate of attention
num_layers = 6             # number of encoder layers
enc_dim = 256              # dimension of the encoder
num_heads = enc_dim // 64  # number of attention heads
dff = 2048                 # dimension of feed-forward network
fbank_dim = 80             # dimension of the FBank features
max_seq_len = 4096        # maximum sequence length

# -----Define the tokenizer-----
# tokenizer
tokenizer = CharTokenizer()
sos, eos, vocab = tokenizer.sos_id, tokenizer.eos_id, tokenizer.vocab
print(f"sos: {sos}, eos: {eos}, vocab: {vocab}")

def tokenization(tokenizer: CharTokenizer, sos: int, eos: int, sentence: str):
    tokens = tokenizer.tokenize(sentence)
    ys_in = [sos] + tokens
    ys_out = tokens + [eos]
    ys_out = torch.tensor(ys_out).long()
    ys_in = torch.tensor(ys_in).long()
    return ys_in, ys_out

# -----Define the data-----
# path to the input wav file
wav_path = "path/to/wav/file.wav"
# ground truth sentence
sentence = "transcription of audio file"

# -----Define the model-----
# feature extractor for extracting features from the input audio
feature_extractor = LinearFeatureExtractionModel(in_dim=fbank_dim, out_dim=enc_dim)
# encoder
encoder = Encoder(
    dropout_emb=0.1, dropout_posffn=0.1, dropout_attn=0.,
    num_layers=num_layers, enc_dim=enc_dim, num_heads=num_heads, dff=2048, tgt_dim=enc_dim
)
# decoder
decoder = Decoder(
    dropout_emb=0.1, dropout_posffn=0.1, dropout_attn=0.,
    num_layers=num_layers, dec_dim=enc_dim, num_heads=num_heads, dff=2048, tgt_dim=enc_dim
)
# seq2seq model
```

```

transformer = Transformer(feature_extractor, encoder, decoder, enc_dim, vocab)
if torch.cuda.is_available():
    transformer.cuda()

# -----Data pre-processing-----
# Load audio
wav, sr = torchaudio.load(wav_path)
wav = wav * (1 << 15)      # rescale the wav data to the range of int16 for kald
# extract FBank features
feats = torchaudio.compliance.kaldi.fbank(wav, num_mel_bins=fbank_dim)    # (seq
feats = feats.unsqueeze(0)          # (1, seq_len,
feat_lens = torch.tensor([feats.shape[1]])    # (1,), the L
# tokenize the sentence
ys_in, ys_out = tokenization(tokenizer, sos, eos, sentence)    # (seq_len,),
ys_in = torch.tensor(ys_in).unsqueeze(0)    # (1, seq_len,
ys_out = torch.tensor(ys_out).unsqueeze(0)    # (1, seq_len,
print(f"feats: {feats.shape}, feat_lens: {feat_lens.shape}, ys_in: {ys_in.shape},

# -----Forward pass-----
if torch.cuda.is_available():
    feats = feats.cuda()
    feat_lens = feat_lens.cuda()
    ys_in = ys_in.cuda()
    ys_out = ys_out.cuda()
logits = transformer(feats, feat_lens, ys_in)    # (batch_size,

# -----Loss computation-----
# compute CrossEntropyLoss with logits and the ground truth
criterion = nn.CrossEntropyLoss(ignore_index=-1, label_smoothing=0.1)
logits = logits.view(-1, logits.size(-1))
ys_out_pad = ys_out.view(-1).long()
loss = criterion(logits, ys_out_pad)
print(f"loss: {loss.item()}")

```

在上述代码中，我们从接收原始的音频和对应的文本，定义模型、预处理模块等组件，完成了一次完整的前向传播，并且成功计算交叉熵损失函数。上述代码已经完整阐述了语音识别任务中Transformer模型的前向传播细节。通过定义optimizer、Dataset、Dataloader等组件，我们就可以进行模型的训练了。

## 加载数据

### Dataset: 如何处理变长数据

在上节中，我们已经明白了Transformer前向传播的细节。但是，上述代码仅仅以一个样本为例，而在实际训练过程中，为了充分利用计算资源，我们通常会多个样本，即一个batch，送入模型中进行并行运算，因此我们需要将多个样本凑成一个batch。这一点在传统的图像分类等任务中很容易实现，因为不同图片的尺寸往往都是相同的，但是在语音中却不那么容易，原因在于语音样本彼此之间、文本样本彼此之间的长度存在差异，不能直接将不同样本的tensor沿着batch维度叠起来。

因此，在凑batch时，需要对样本进行额外处理，具体处理办法也很简单：在一个batch中，我们将较短的样本填充(padding)到当前batch中最长样本的长度即可。这种做法和CNN中卷积核的padding类似，只是这里是对样本在时间维度上进行padding，而不是在空间维度。

但是，上述做法还有两个待解决的问题：

1. 怎么确保padding的部分不影响模型的训练？或者具体点，怎么确保padding的部分不纳入loss的计算？

关于这一点，音频方面，我们通过Transformer中encoder的掩码机制来确保音频的padding部分不会纳入attention的计算，进而也不会影响前向传播以及loss的计算；文本方面，对于padding的部分，我们可以直接用`torch.nn.functional.cross_entropy`函数中的`ignore_index`参数来忽略padding部分的loss

有关encoder的掩码机制，请参考讲解Transformer的前文。

2. 怎么确保不同batch要求的计算资源变化不会太大？即既保证不会爆显存，又保证计算资源被充分利用？

具体来说，不同的样本长度差异比较大，如果仅仅指定batch-size，那么有可能不同batch之间的特征总帧数差异较大。比如，同样是十个样本，batch A中的十个音频样本的长度都为1，而batch B中的十个音频样本的长度都为10000。在对batch A进行forward pass时，显然计算资源没有得到充分利用，而在对batch B进行forward pass时，又容易出现OOM (Out-Of-Memory) 等问题。针对这点，一个比较简单的解决方案是，我们不指定batch-size，而是根据实际情况指定单张GPU上可以处理的最大帧数（在语音识别中，我们也可以指定音频的累计时长），然后将多个帧数之和不超过最大帧数的样本凑成一个batch，参考下面代码的`init_mini_batch`方法；

下面给出Dataset的完整代码：

```
# coding=utf-8
# Contact: bingquanxia@qq.com

class ASRDataset(Dataset):
    def __init__(
        self, wav_paths: List[str], transcripts: List[str], wav_lengths: List
        tokenizer: CharTokenizer, batch_size: int, batch_seconds: int, shuffle
    ):
        """
        wav_paths: list of paths to wav files
        wav_lengths: list of lengths (in seconds) of wav files
        transcripts: list of texts
        tokenizer: tokenizer to convert char to tokens
        sos: start of sentence token
        eos: end of sentence token
        batch_size: batch size
        batch_seconds: batch length in seconds
        shuffle: whether to shuffle the dataset
        """
        assert len(wav_paths) == len(transcripts) == len(wav_lengths)
        self.wav_paths = wav_paths
        self.transcripts = transcripts
        self.wav_lengths = wav_lengths
        self.samples = [(wav_paths[i], transcripts[i], wav_lengths[i]) for i in range(len(wav_paths))]
        self.sr = 16000
        # self.sr = 8000

        self.tokenizer = tokenizer
        self.sos = tokenizer.sos_id
        self.eos = tokenizer.eos_id

        self.batch_size = batch_size
        self.batch_seconds = batch_seconds
        self.is_shuffle = shuffle
        self.minibatches = []
        self.shuffle()

        print(f"ASRDataset: {len(self.samples)} samples, {len(self.minibatches)} minibatches, "
              f"batch_size: {self.batch_size}, batch_seconds: {self.batch_seconds}, "
              f"mean batch-size: {len(self.samples) / len(self.minibatches):.2f}")

    def shuffle(self):
        if self.is_shuffle:
            random.shuffle(self.samples)
```

```

self.init_mini_batches()
if self.is_shuffle:
    random.shuffle(self.minibatches)

def init_mini_batches(self):
    """
    initialize mini-batches
    Code generated by Github Copilot
    """
    self.minibatches = []
    # sort samples by length
    self.samples = sorted(self.samples, key=lambda x: x[2])
    # initialize mini-batches
    minibatch = []
    frames = 0
    for sample in self.samples:
        path, transcript, length = sample
        frames += length # length is in seconds
        if frames > self.batch_seconds or len(minibatch) >= self.batch_size:
            self.minibatches.append(minibatch)
            minibatch = [sample]
            frames = length
        else:
            minibatch.append(sample)
    if minibatch:
        self.minibatches.append(minibatch)

def __len__(self) -> int:
    return len(self.minibatches)

def __getitem__(self, index: int) -> Tuple[torch.Tensor, torch.Tensor, torch.Tensor]:
    """
    fetch a batch of data

    index: batch index

    returns:
        fbank_feat: (batch_size, seq_len, 80)
        feat_lens: (batch_size,)
        ys_in_pad: (batch_size, max_token_len)
        ys_out_pad: (batch_size, max_token_len)

    """
    # Load fbanks
    fbanks = []
    for sample in self.minibatches[index]:
        path, transcript, length = sample
        wav, sr = torchaudio.load(path)
        assert sr == self.sr, f"sample rate mismatch: {sr} != {self.sr}"
        wav = wav * (1 << 15) # rescale to int16 for kaldi compatib
        fb = torchaudio.compliance.kaldi.fbank(wav, num_mel_bins=80) # (seq_
        fbanks.append(fb)
    # Load tokens
    ys_in = []
    ys_out = []
    for sample in self.minibatches[index]:
        path, transcript, length = sample
        tokens = self.tokenizer.tokenize(transcript)
        ys_in.append([self.sos] + tokens)
        ys_out.append(tokens + [self.eos])
    # padding fbanks
    max_fbank_len = max([fb.shape[0] for fb in fbanks])
    fbank_feat = torch.zeros(len(fbanks), max_fbank_len, 80)
    for i, fb in enumerate(fbanks):
        fbank_feat[i, :fb.shape[0]] = fb
    feat_lens = torch.tensor([fb.shape[0] for fb in fbanks]).long()
    # padding tokens
    pad_token_for_ys_in = self.eos

```

```

pad_token_for_ys_out = -1 # when calculating loss, ignore padded tokens
max_token_len = max([len(tokens) for tokens in ys_in])
ys_in_pad = torch.ones(len(ys_in), max_token_len)
ys_in_pad.fill_(pad_token_for_ys_in)
ys_out_pad = torch.ones(len(ys_out), max_token_len)
ys_out_pad.fill_(pad_token_for_ys_out)
for i, tokens in enumerate(ys_in):
    ys_in_pad[i, :len(tokens)] = torch.tensor(tokens)
for i, tokens in enumerate(ys_out):
    ys_out_pad[i, :len(tokens)] = torch.tensor(tokens)

return fbank_feat, feat_lens, ys_in_pad, ys_out_pad

```

值得一提的是，在Dataset类中，`__getitem__` 方法通常指返回单个样本，并且不指定 `batch_size`，然后在Dataloader中指定 `batch_size`，并且通过Dataloader默认的 `collate_fn` 来将同个worker进程返回的多个单独的样本凑成一整个batch（[官方文档链接](#)）。

但是，在此处，我们是直接在Dataset的 `__getitem__` 方法中就已经凑成了一整个batch，因此在初始化Dataloader的实例时我们需要将 `batch_size` 设置为 `None`，没有使用PyTorch默认的 `Automatic batching`。

## Dataloader

Dataloader的完整代码如下：

```

def get_dataloader(
    wav_paths: List[str], transcripts: List[str], wav_lengths: List[float],
    tokenizer: CharTokenizer,
    batch_size: int, batch_seconds: int,
    shuffle: bool, num_workers: int = 4
) -> DataLoader:
    dataset = ASRDataset(
        wav_paths, transcripts, wav_lengths, tokenizer, batch_size, batch_seconds
    )
    dataloader = DataLoader(dataset, batch_size=None, num_workers=num_workers)
    return dataloader

```

## 补充说明

上述代码展示了如何加载变长的音频和文本数据，但是，在实际训练过程中，还需要考虑其他因素。这里简单进行说明，有兴趣的读者可以参考目前主流的Seq2Seq框架中的做法，比如 `transformers`、`espnet` 和 `fairseq` 等

- 拼凑batch的时机：上述做法是直接在Dataset类中就凑成一整个可以直接输入模型的batch，但是也有其他做法。比如在dataset类中仍然返回单个样本的输入，然后再自定义Dataloader的 `collate_fn` 来进行样本的padding并凑成batch；
- dataset的shuffle：可以直接在dataset中定义shuffle方法然后在每个epoch之初调用以实现shuffle，也可以定义Dataloader时传入 `batch_sampler` 或者 `sampler`，或者定义Dataloader时指定 `shuffle` 参数为 `True`。做法比较多，视个人喜好而定。

## 训练代码

综合上述代码，接下来，我们给出模型训练的代码：



为了保证代码简洁, 省略部分较冗长的重复代码, 完整的、可直接运行的代码见文章首部的GitHub链接。

```
# define_tokenizer
tokenizer = CharTokenizer()
sos_id = tokenizer.sos_id
eos_id = tokenizer.eos_id
vocab_size = tokenizer.vocab

# define_data_loader
with open("./data/train/wav.scp") as f:
    wav_paths = f.read().splitlines()
with open("./data/train/text") as f:
    transcripts = f.read().splitlines()
with open("./data/train/wav.lengths") as f:
    wav_lengths = f.read().splitlines()
batch_size = 32
batch_seconds = 256 # depends on your GPU memory
data_loader = get_data_loader(wav_paths, transcripts, tokenizer, sos_id, eos_id, batch_size)

# define_model
feature_extractor = ...
encoder = ...
decoder = ...
enc_dim = ...
model = Transformer(feature_extractor, encoder, decoder, enc_dim, vocab_size)
model.train()

# define_optimizer_and_scheduler
optimizer = torch.optim.Adam(model.parameters(), lr=4e-4)
scheduler = torch.optim.lr_scheduler.OneCycleLR(optimizer, max_lr=4e-4, steps_per_epoch=...)

# define_loss_criterion
criterion = nn.CrossEntropyLoss(ignore_index=-1) # -1: ignore padding

# main_loop
num_epochs = 10
for epoch in range(num_epochs):
    for i, (batch) in enumerate(data_loader):
        # get_batch_data
        fbank_feat, ys_in_pad, ys_out_pad = batch

        # forward
        logits = model(fbank_feat, feat_lens, ys_in_pad)

        # calculate_loss
        loss = criterion(logits, targets)

        # backward
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
        scheduler.step()

    # print_loss
    if (i+1) % 10 == 0:
        print(f"Epoch: {epoch+1}/{num_epochs}, Step: {i+1}/{len(data_loader)}.")

    # save_model
    torch.save(model.state_dict(), f"./checkpoints/model_{epoch+1:04d}.pth")
```

至此, 模型的训练部分圆满收工。

接下来, 让我们看推理部分。

## Greedy Search简介

简单来说, Greedy Search就是不停地将当前的预测结果输入模型, 让模型预测下一个token的分布, 然后再从该分布中选取概率最大的那个token, 拼接到目前预测结果中。重复该步骤, 直至解码结束。

Greedy Search算法的伪代码如图5所示

### Algorithm Greedy Search

**Input:** encoder's output X, decoder D

**Output:** transcription Y

```
1. Y ← [sos]
2. While True
3.     logits ← D(Y, X)
4.     logp ← log_softmax(logits[-1])
5.     next_token ← logp.argmax()
6.     If next_token = eos
7.         break
8.     end if
9.     Y ← [Y, next_token]
10. end while
```

知乎 @晚安汤姆布利多

图5. Greedy Search伪代码

## 推理代码

前面已经做了足够多的铺垫, 这里直接给出推理阶段的代码, 如下所示:

```
# define_tokenizer
tokenizer = CharTokenizer()
tokenizer = add_sos_eos_to_tokenizer(tokenizer)
sos_id = tokenizer.sos_id
eos_id = tokenizer.eos_id
vocab_size = tokenizer.vocab

# define_dataloader
with open("./data/val/wav.scp") as f:
    wav_paths = f.read().splitlines()
with open("./data/val/text") as f:
    transcripts = f.read().splitlines()
with open("./data/val/wav.lengths") as f:
    wav_lengths = f.read().splitlines()
batch_size = 1
batch_seconds = 1000000 # unlimited
data_loader = get_dataloader(wav_paths, transcripts, wav_lengths, tokenizer, batch_size)

# define_model
feature_extractor = ...
encoder = ...
decoder = ...
enc_dim = ...
model = model(feature_extractor, encoder, decoder, enc_dim, vocab_size)
model.eval() # switch to evaluation mode
```

```
# main loop
max_decode_len = 100
with torch.no_grad():
    for i, data in enumerate(data_loader):
        fbank_feat, feat_lens, _, gt = data
        ys_in_pad = torch.tensor([[sos_id]]).long() # (1, 1)
        # greedy search
        while True:
            logits = model(fbank_feat, feat_lens, ys_in_pad)
            logits = logits[:, -1]
            logp = F.log_softmax(logits, dim=-1)
            y_hat = logits.argmax(-1)
            if y_hat == eos_id:
                break
            ys_in_pad = torch.cat([ys_in_pad, y_hat.view(1, -1)], dim=-1)
            if len(ys_in_pad[0]) > max_decode_len:
                break
        ys_in_pad = ys_in_pad[:, 1:] # remove sos_id
        pred = tokenizer.detokenize(ys_in_pad[0].tolist())
        print(f"{i:05d}\t{gt}\t{pred}")
```

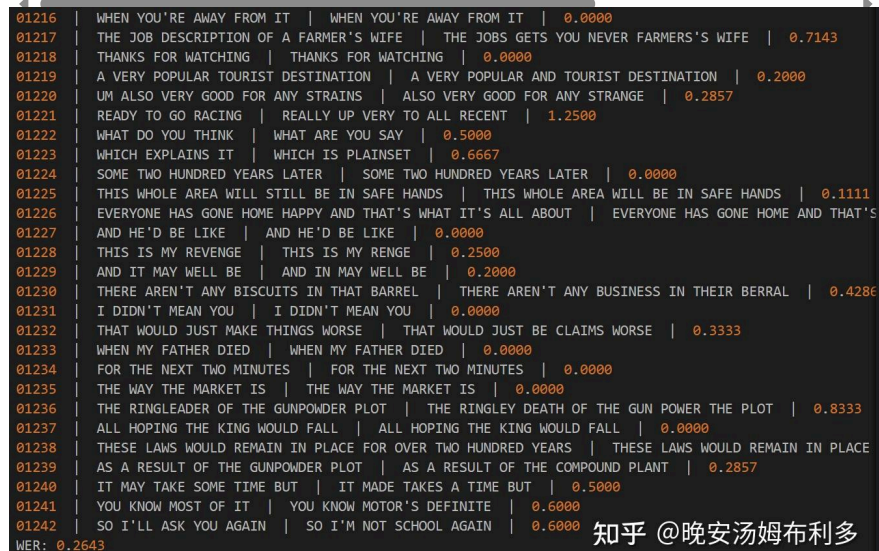
## 案例: LRS2

接下来介绍一个基于数据集数据集LRS2的简单案例。

LRS2是一个200多小时的包含说话人视频、音频和文本的唇语识别数据集，我们使用该数据集的音频和文本进行语音识别模型的训练。该数据集的训练部分包含pretrain和train两个子集，这里只使用train子集用于训练，然后在测试集上进行测试。完整的训练代码和训练参数设置参考下面的链接。

由于模型尺寸小、训练时间短、数据集小等因素，模型性能并不算很好，不过此处主要为了验证，而不是冲性能，有兴趣的读者可以自行调参。

部分测试结果如下，其中每一行为“样本序号 | ground truth | prediction | wer”。wer表示词错误率 (Word Error Rate, WER)，越低越好。



```
01216 | WHEN YOU'RE AWAY FROM IT | WHEN YOU'RE AWAY FROM IT | 0.0000
01217 | THE JOB DESCRIPTION OF A FARMER'S WIFE | THE JOBS GETS YOU NEVER FARMERS'S WIFE | 0.7143
01218 | THANKS FOR WATCHING | THANKS FOR WATCHING | 0.0000
01219 | A VERY POPULAR TOURIST DESTINATION | A VERY POPULAR AND TOURIST DESTINATION | 0.2000
01220 | UM ALSO VERY GOOD FOR ANY STRAINS | ALSO VERY GOOD FOR ANY STRANGE | 0.2857
01221 | READY TO GO RACING | REALLY UP VERY TO ALL RECENT | 1.2500
01222 | WHAT DO YOU THINK | WHAT ARE YOU SAY | 0.5000
01223 | WHICH EXPLAINS IT | WHICH IS PLAINSET | 0.6667
01224 | SOME TWO HUNDRED YEARS LATER | SOME TWO HUNDRED YEARS LATER | 0.0000
01225 | THIS WHOLE AREA WILL STILL BE IN SAFE HANDS | THIS WHOLE AREA WILL BE IN SAFE HANDS | 0.1111
01226 | EVERYONE HAS GONE HOME HAPPY AND THAT'S WHAT IT'S ALL ABOUT | EVERYONE HAS GONE HOME AND THAT'S
01227 | AND HE'D BE LIKE | AND HE'D BE LIKE | 0.0000
01228 | THIS IS MY REVENGE | THIS IS MY RENGE | 0.2500
01229 | AND IT MAY WELL BE | AND IN MAY WELL BE | 0.2000
01230 | THERE AREN'T ANY BISCUITS IN THAT BARREL | THERE AREN'T ANY BUSINESS IN THEIR BERRAL | 0.4286
01231 | I DIDN'T MEAN YOU | I DIDN'T MEAN YOU | 0.0000
01232 | THAT WOULD JUST MAKE THINGS WORSE | THAT WOULD JUST BE CLAIMS WORSE | 0.3333
01233 | WHEN MY FATHER DIED | WHEN MY FATHER DIED | 0.0000
01234 | FOR THE NEXT TWO MINUTES | FOR THE NEXT TWO MINUTES | 0.0000
01235 | THE WAY THE MARKET IS | THE WAY THE MARKET IS | 0.0000
01236 | THE RINGLEADER OF THE GUNPOWDER PLOT | THE RINGLEY DEATH OF THE GUN POWER THE PLOT | 0.8333
01237 | ALL HOPING THE KING WOULD FALL | ALL HOPING THE KING WOULD FALL | 0.0000
01238 | THESE LAWS WOULD REMAIN IN PLACE FOR OVER TWO HUNDRED YEARS | THESE LAWS WOULD REMAIN IN PLACE
01239 | AS A RESULT OF THE GUNPOWDER PLOT | AS A RESULT OF THE COMPOUND PLANT | 0.2857
01240 | IT MAY TAKE SOME TIME BUT | IT MADE TAKES A TIME BUT | 0.5000
01241 | YOU KNOW MOST OF IT | YOU KNOW MOTOR'S DEFINITE | 0.6000
01242 | SO I'LL ASK YOU AGAIN | SO I'M NOT SCHOOL AGAIN | 0.6000
WER: 0.2643
```

知乎 @晚安汤姆布利多

图6. LRS2测试集部分推理结果

参考

1. ^ <https://www.quora.com/What-does-end-to-end-mean-in-deep-learning-methods>

2. ^ <https://machinelearningmastery.com/transduction-in-machine-learning/>

3. ^ <https://medium.com/descript/a-brief-history-of-asr-automatic-speech-recognition-b8f338d4c0e5>

4. ^ <http://shelf2.library.cmu.edu/Tech/18073818.pdf>

5. ^ <http://proceedings.mlr.press/v32/graves14.pdf>

6. ^ <https://arxiv.org/pdf/1508.07909.pdf>

编辑于 2024-04-26 09:51 · 北京

语音识别    Transformer    深度学习（Deep Learning）



理性发言，友善互动

62 条评论

默认    最新



monster

这个数据集下载好像有点麻烦，有没有可以替代的更小的数据集呢？

2024-02-05 · 安徽

回复    1



晚安汤姆布利多 作者 · 是王王王啊

都可以，不挑数据集，但是训练策略等可能要根据数据集做一定调整

2024-04-01 · 北京

回复    1



Insistence

大哥 我也搞不来lsr2数据集，就是好麻烦，请问您最后怎么解决的呢？

02-11 · 陕西

回复    喜欢

展开其他 2 条回复 >



益达abcd

博主你好，请问这个算法可以用于船舶声音的分类吗，同样是音频，处理方式应该是一样的把🤔

2024-10-12 · 湖北

回复    喜欢



晚安汤姆布利多 作者 · 益达abcd

这个看你自己，没有统一答案

2024-10-14 · 湖北

回复    喜欢



益达abcd · 晚安汤姆布利多

感谢大大🤖。请问在序列分类中decoder输入的内容就是分类的标签吗？

2024-10-14 · 湖北

回复    喜欢

展开其他 1 条回复 >



华洛

可以求一份LSR2数据集吗？一直申请不了，用学生邮箱发了都不行🤔

04-24 · 广东

回复    喜欢



晚安汤姆布利多 作者

- 

抱歉🙏我已经毕业了

04-24 · 北京

回复

喜欢
- 

华洛 · 晚安汤姆布利多

...

没事没事，谢谢回复！这个只是我的一个选修课的复现代码需要的数据集，找不到就算了🤔

04-24 · 广东

回复

喜欢
- 

墨心

...

博主，你好，通过fbank特征提取后，是一个帧数✖特征系数的矩阵，请问文章中说的（btd）怎么进行对应？

03-24 · 江西

回复

喜欢
- 

晚安汤姆布利多 作者

...

b是batch size，表示多个样本。这个你看一下代码就知道了

03-24 · 北京

回复

喜欢
- 

一杆梅子酒

...

feats的seq\_len不一定等于 ys\_out的seq\_len吧？ys\_out还多了一个eos

02-20 · 浙江

回复

喜欢
- 

Insistence

...

大佬 可以发一份这个lsr2数据集嘛，萌新一直搞不来这个数据集🤔

02-11 · 陕西

回复

喜欢
- 

华洛

...

可以求一份LSR2数据集吗？一直申请不了，用学生邮箱发了都不行🤔

04-24 · 广东

回复

喜欢
- 

包之

...

博主大大，我想在老家录一个方言语音识别数据集，请问这个数据集大概是什么格式比较好呀？（比如语音长度、标签、样本个数）谢谢啦🤔，如果我的问题不合理也请斧正。

01-19 · 湖北

回复

喜欢
- 

晚安汤姆布利多 作者

...

数据格式这个自己规定应该就行，只要能读取

01-19 · 北京

回复

喜欢
- 

Mr.Rabbit

...

出现了这样的问题： Couldn't find appropriate backend to handle uri ./data/LibriSpeech/train-clean-100/1963/142393/1963-142393-0051.flac and format None.

2024-12-04 · 青海

回复

喜欢
- 

fine

...

博主你好，请问推理部分可以支持batch吗

2024-10-20 · 北京

回复

喜欢
- 

晚安汤姆布利多 作者

...

支持，自己修改即可

2024-10-20 · 湖北

回复

喜欢
- 

知乎用户jyYMdi

...

请教博主，都是一样的设置，为啥log里的loss掉的那么快，我的掉的好慢啊🤔🤔🤔🤔🤔

2024-07-19 · 四川

回复

喜欢
- 

知乎用户jyYMdi · 晚安汤姆布利多

...

好的，我再看看，谢谢博主

2024-07-21 · 四川

回复

喜欢
- 

晚安汤姆布利多 作者 · 知乎用户jyYMdi

...

LRS2有完整log、设置以及checkpoint，可以先debug看看

2024-07-21 · 四川

回复

喜欢

展开其他 2 条回复 >

点击查看全部评论 >



理性发言，友善互动

推荐阅读

Neural Speech Synthesis with Transformer Network

Saihan Li<sup>1,4</sup>, Shujie Liu<sup>2</sup>, Yangting Liu<sup>3</sup>, Sheng Zhao<sup>3</sup>, Ming Liu<sup>1,4</sup>, Ming Zhou<sup>1</sup>

<sup>1</sup>University of Electronic Science and Technology of China  
<sup>2</sup>Microsoft Research Asia  
<sup>3</sup>Microsoft STC Asia  
<sup>4</sup>CETC Big Data Research Institute Co., Ltd, Guizhou, China  
{shuajie.liu, yangting.liu, sheng.zhao, mingzhou}@microsoft.com  
saihanli@uestc.edu.cn

基于Transformer的语音合成系统

星辰漫游者



一文看懂Transformer

学术FUN      发表于助考笔记

RetNe Transf

我认为这  
is all yo  
文了。这  
Transfo  
架构的语  
音到图像  
山河动ノ