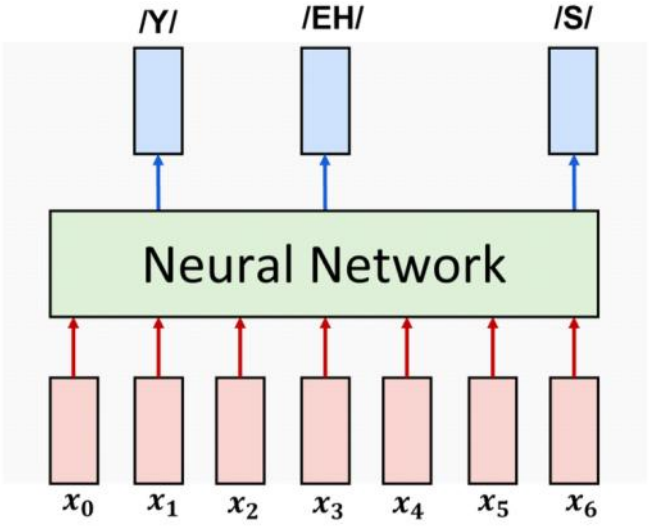
HW3-P2（S24）

2024年8月22日 10:50

1 简介

- **前言：**数据集用的S24版，讲义参考的S23版（完整看完并浓缩到笔记中了），代码参考了S24和S23两CMU人的。没有做消融实验。
- **主要任务**：RNN方案来实现音素识别任务（不能使用attention技术）。如下图，语音MFCC序列（橙色方框）对应单词 "yes"，发音为音素序列"/Y/ /EH/ /S/"。
  - 

- **任务难点**：解决输出序列与输入序列之间的顺序对齐但时间不同步的问题。输出是零星的，而不是时间同步的。输出中出现哪些音素、输出序列的长度，甚至何时输出录音中的音素，都是未知的。
- **项目官方PPT**：
  - https://docs.google.com/presentation/d/1eMc4M6ABAwZmFnlo2VByizsy2_Hgtmmk
  - https://docs.google.com/presentation/d/1Og0ovDwT7-a8_HUILKUEbF-jdgYZSAuHFwSJuZ23kt0
- **官网项目视频**：https://www.youtube.com/watch?v=T87ZBvIKwlM
- **kaggle主页**：https://www.kaggle.com/competitions/hw3p2asr-s24

作业目标

完成本次作业后，理想情况下你将学到：

- **解决使用序列模型的序列到序列问题**
  - 如何在 PyTorch 上设置基于 GRU/LSTM 的模型
  - 如何利用 CNN 作为特征提取器
  - 如何处理序列数据
  - 如何对长度可变的数据批次进行填充/打包
  - 如何使用 CTC 损失来训练模型
  - 如何优化模型
  - 如何实现和利用解码器，如贪婪解码器和束搜索解码器
- **探索架构和超参数以找到最佳解决方案**
  - 如何识别和列举所有影响解决方案的设计/架构选择、参数和超参数
  - 如何制定策略，在这些选项空间中搜索以找到最佳解决方案
- **探索的阶段性过程**
  - 如何最初设置一个简单的解决方案，使其易于实施和优化
  - 如何分阶段处理数据以有效地搜索解决方案空间
  - 如何挑选有前景的配置/设置，并对其进行调整以获得更高的性能
- **使用你的工具来工程化解决方案**
  - 如何使用 PyTorch 框架中的对象来构建基于 GRU/LSTM 的模型
  - 如何处理数据加载、内存使用、算术精度等问题，以最大化训练和推理的时间效率

2 数据集介绍

本项目的数据集几乎跟HW1P2 音素分类一模一样，除了一点，那就是本项目音素的标签与对应的mfcc数据并不是一一对应的了。

具体的说，在HW1P2项目中，“19-198-0000.npy”MFCC数据对应的标签维度为(194,)，而这个MFCC维度为(192, 28)，之所以多了2帧，是因为转录文件数据中首尾分别加了 [SOS] 和 [EOS] ，分别代表“Start of Speech”和“End of Speech”，表示语音片段的开始和结束。完整的标签内容如下： [SIL] 表示“silence”（静默帧，即本帧内没有在说话）

算。

```
(194,)
['[SOS]' '[SIL]' '[SIL]' '[SIL]' '[SIL]' '[SIL]' '[SIL]' '[SIL]' '[SIL]'
 '[SIL]' '[SIL]' '[SIL]' '[SIL]' '[SIL]' '[SIL]' '[SIL]' '[SIL]' '[SIL]'
 '[SIL]' '[SIL]' '[SIL]' '[SIL]' '[SIL]' '[SIL]' '[SIL]' '[SIL]' '[SIL]'
 '[SIL]' '[SIL]' '[SIL]' '[SIL]' '[SIL]' '[SIL]' '[SIL]' '[SIL]' '[SIL]'
 '[SIL]' '[SIL]' '[SIL]' '[SIL]' '[SIL]' '[SIL]' '[SIL]' '[SIL]' '[SIL]'
 '[SIL]' '[SIL]' '[SIL]' '[SIL]' '[SIL]' '[SIL]' '[SIL]' '[SIL]' '[SIL]'
 'N' 'N' 'N' 'N' 'N' 'N' 'N' 'N' 'N' 'N' 'N' 'N' 'N' 'N' 'AO' 'AO' 'AO'
 'AO' 'AO' 'AO' 'AO' 'R' 'R' 'R' 'R' 'R' 'R' 'R' 'R' 'R' 'R' 'TH' 'TH'
 'TH' 'TH' 'TH' 'TH' 'TH' 'TH' 'TH' 'TH' 'TH' 'AH' 'AH' 'AH' 'AH' 'AH'
 'AH' 'AH' 'N' 'N' 'N' 'N' 'N' 'N' 'JH' 'ER' 'ER' 'ER' 'ER' 'ER' 'ER' 'ER'
 'ER' 'ER' 'ER' 'ER' 'ER' 'ER' 'ER' 'ER' 'ER' 'ER' 'AE' 'AE' 'AE' 'AE'
 'AE' 'AE' 'AE' 'AE' 'AE' 'AE' 'AE' 'AE' 'AE' 'AE' 'AE' 'AE' 'AE' 'AE' 'B'
 'B' 'B' 'B' 'B' 'B' 'B' 'B' 'IY' 'IY' 'IY' 'IY' 'IY' 'IY' 'IY' 'IY' 'IY'
 'IY' 'IY' 'IY' 'IY' 'IY' 'IY' 'IY' 'IY' 'IY' 'IY' 'IY' 'IY' 'IY' 'IY'
 'IY' 'IY' 'IY' 'IY' '[SIL]' '[SIL]' '[SIL]' '[SIL]' '[SIL]' '[SIL]'
 '[SIL]' '[SIL]' '[SIL]' '[SIL]' '[SIL]' '[SIL]' '[SIL]' '[EOS]']
```

即，在HW1P2项目中，每帧MFCC都有一个对应的音素标签。

但是在本项目中，“19-198-0000.npy”MFCC数据对应的音素标签变成如下：

```
音素标签内容：
['[SOS]' '[SIL]' 'N' 'AO' 'R' 'TH' 'AH' 'N' 'JH' 'ER' 'AE' 'B' 'IY'
 '[SIL]' '[EOS]']
```

相当于现在给你一段音频数据，你用模型直接预测出这段音频的文字内容（文字内容用音素表示）。

S24训练集：28539个.npy（每个文件对应一句话信息，每句话长度不一样）

S24验证集：2703个

S24测试集：2620个

S24版模型的预测输出类别41个：39个音素 + [SIL]静默标签 + BLANK（音素分隔符）。

S24版标签中字符总类别40个：39个音素 + [SIL]静默标签 。

备注：S23版讲义中说是42个应该是笔误或年复一年换皮数据集时讲义有些地方没注意修改。

关于BLANK作用：假设语音信息中有连续的2个R音素，预测输出应该是RR，但是根据合并规则，连续出现的音素符号需要合并，即变为R，这就与真实情况不一样，所以需要有个间隔符BLANK来分隔它们，表示两个R不是同一个音素。具体在CTC解码中，会对41个类别进行合并，最终输出40个字符类别的字符串去跟标签进行编辑距离计算。

下图是音素标签定义，网络模型的输出是字典中的“值”，带空白符“-”和静音符“|”总共是41个标记。

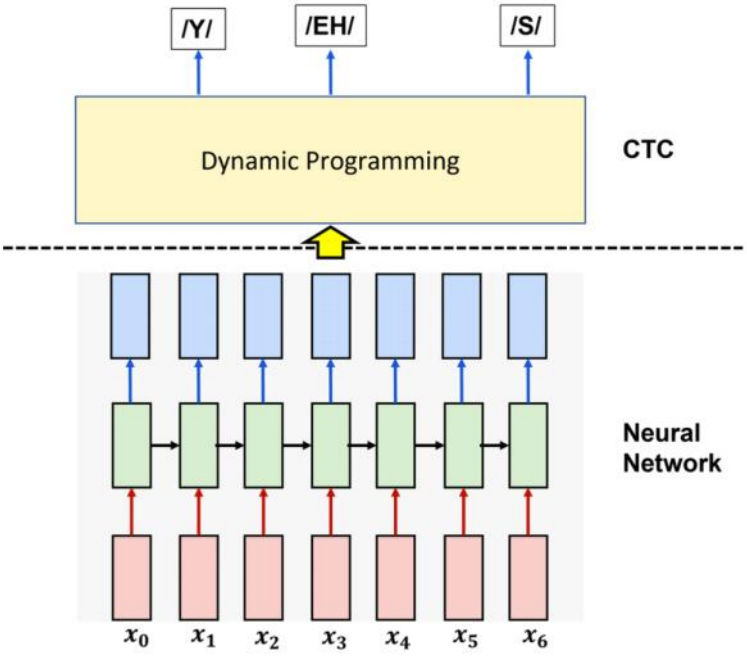
```
# CMU启动笔记本默认的标签设计：
# CMUdict_ARPAbet = {
#     "" : " ",
#     "[SIL]": "-", "NG": "G", "F" : "f", "M" : "m", "AE": "@",
#     "R"    : "r", "UW": "u", "N" : "n", "IY": "i", "AW": "W",
#     "V"    : "v", "UH": "U", "OW": "o", "AA": "a", "ER": "R",
#     "HH"   : "h", "Z" : "z", "K" : "k", "CH": "C", "W" : "w",
#     "EY"   : "e", "ZH": "Z", "T" : "t", "EH": "E", "Y" : "y",
#     "AH"   : "A", "B" : "b", "P" : "p", "TH": "T", "DH": "D",
#     "AO"   : "c", "G" : "g", "L" : "l", "JH": "j", "OY": "O",
#     "SH"   : "S", "D" : "d", "AY": "Y", "S" : "s", "IH": "I",
#     "[SOS]": "[SOS]", "[EOS]": "[EOS]"
# }

# 我改造了第前2个字符的表示。
CMUdict_ARPAbet = {
    "" : "_",
    "[SIL]": "|", "NG": "G", "F" : "f", "M" : "m", "AE": "@",
    "R"    : "r", "UW": "u", "N" : "n", "IY": "i", "AW": "W",
    "V"    : "v", "UH": "U", "OW": "o", "AA": "a", "ER": "R",
    "HH"   : "h", "Z" : "z", "K" : "k", "CH": "C", "W" : "w",
    "EY"   : "e", "ZH": "Z", "T" : "t", "EH": "E", "Y" : "y",
    "AH"   : "A", "B" : "b", "P" : "p", "TH": "T", "DH": "D",
    "AO"   : "c", "G" : "g", "L" : "l", "JH": "j", "OY": "O",
    "SH"   : "S", "D" : "d", "AY": "Y", "S" : "s", "IH": "I",
    "[SOS]": "[SOS]", "[EOS]": "[EOS]"
}

CMUdict = list(CMUdict_ARPAbet.keys())
ARPAbet = list(CMUdict_ARPAbet.values())

PHONEMES = CMUdict[:-2]
LABELS = ARPAbet[:-2]
```

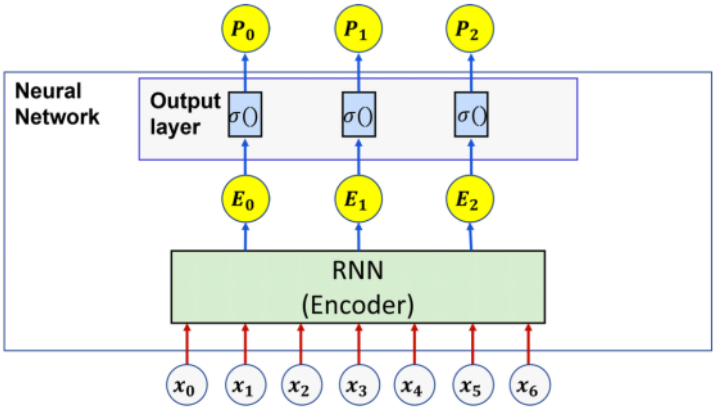
3 网络架构

模型结构如下：

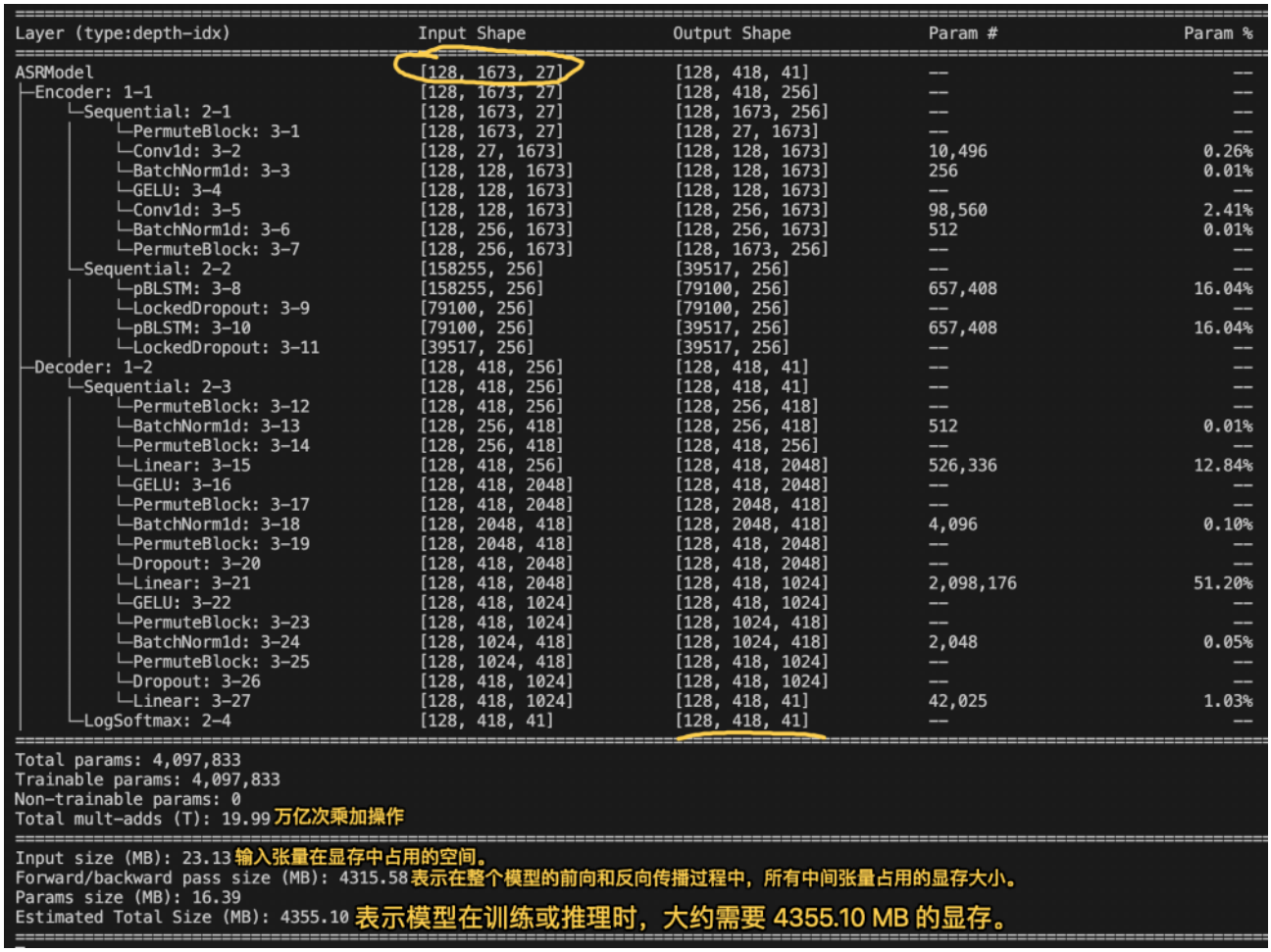
- **encoder层：**通过卷积层和双向 LSTM（pBLSTM）层提取并编码输入序列中的时序信息。
- **decoder层：**通过多层线性变换和激活处理，将编码器输出的高维特征转化为预测的类别概率。
- **输出层**：通过 LogSoftmax 将预测结果转换为对数概率，以进行更稳定的损失计算。

- **损失函数**：CTC loss





```
ASRModel(
  (encoder): Encoder(
    (embed): Sequential(
      (0): PermuteBlock()
      (1): Conv1d(27, 128, kernel_size=(3,), stride=(1,), padding=(1,))
      (2): BatchNorm1d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (3): GELU(approximate='none')
      (4): Conv1d(128, 256, kernel_size=(3,), stride=(1,), padding=(1,))
      (5): BatchNorm1d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (6): PermuteBlock()
    )
    (pBLSTMs): Sequential(
      (0): pBLSTM(
        (blstm): LSTM(512, 256, batch_first=True, dropout=0.2, bidirectional=True)
      )
      (1): LockedDropout()
      (2): pBLSTM(
        (blstm): LSTM(1024, 256, batch_first=True, dropout=0.2, bidirectional=True)
      )
      (3): LockedDropout()
    )
  )
  (decoder): Decoder(
    (mlp): Sequential(
      (0): PermuteBlock()
      (1): BatchNorm1d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (2): PermuteBlock()
      (3): Linear(in_features=512, out_features=2048, bias=True)
      (4): GELU(approximate='none')
      (5): PermuteBlock()
      (6): BatchNorm1d(2048, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (7): PermuteBlock()
      (8): Dropout(p=0.2, inplace=False)
      (9): Linear(in_features=2048, out_features=1024, bias=True)
      (10): GELU(approximate='none')
      (11): PermuteBlock()
      (12): BatchNorm1d(1024, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (13): PermuteBlock()
      (14): Dropout(p=0.2, inplace=False)
      (15): Linear(in_features=1024, out_features=41, bias=True)
    )
    (softmax): LogSoftmax(dim=2)
  )
)
```



备注：

- **输入维度[128, 1673, 27]**：
  - 第一维：batchsize
  - 第2维：128个样本中，最长帧数为1673。其他127个样本进行0填充帧到1673帧对齐。
  - 第3维：每一帧有27个特征。
- **PermuteBlock层**：就是自定义的一个维度转换层，将第1维和第2维数据调换位置。（0维是batchsize）
- **进入pBiLSTM前的158255**：是一个batch中所有样本的非0填充总帧数。即进来之前有128*1673=214144帧，删除非0填充帧后还剩158255帧。

问题1：为什么要加1D卷积扩充特征数?

使用 1D 卷积对 MFCC 的原始 27 维特征进行特征提取是有意义的，尤其是在音频处理和语音识别任务中。这种做法的意义和优势可以从以下几个方面来理解：

a. **局部特征提取**：
   - **MFCC** 特征表示了音频在时间和频率上的分布，27 维通常对应于不同频带的能量分布。
   - 通过 1D 卷积，模型可以学习到跨时间步的局部模式（例如，某些音素或音调的变化模式）。这些模式可能对应于某些频率范围内特定的发音特征。

b. **频率之间的相关性**：

- 在传统的 MFCC 特征中，不同频带的能量通常是独立计算的，但实际上在语音信号中，频率之间存在一定的相关性。
- 使用 1D 卷积可以帮助模型捕捉到这些相关性，通过在时间轴上滑动的卷积核，能够提取频率之间的局部依赖关系，从而更好地理解音频信号。

c. **空间不变性**：
- 卷积操作本身具有平移不变性（translation invariance）的特点，意味着模型可以更好地处理音频信号中的噪声和变动，例如不同的讲话速度或音高。
- 这种空间不变性帮助模型对音素的特征进行更鲁棒的提取，即使在输入信号发生小的变化时，卷积核仍然可以捕捉到相似的特征。

d. **减少特征维度，增加模型的表达能力**：
- 卷积可以通过对输入特征进行下采样（例如，通过 stride 或 pooling）来减少数据的时间维度，从而减少计算量。
- 通过多层卷积网络，可以逐渐提取更高级的特征，使模型对音频信号有更深层次的理解。

e. **提升模型的表现力**：
- 简单的 MFCC 特征仅捕捉到了基本的频率信息，而卷积层可以提取更复杂、更抽象的特征。这些特征在后续的层（如 LSTM 或全连接层）中能更好地帮助模型进行分类或预测任务。

问题2：pBLSTM层原理

本质就是【下采样+BiLSTM】，下采样通过将输入序列相邻元素两两拼接起来实现（序列长度降为50%，每个序列元素特征数翻倍）。

比如：输入序列长度是100（即100个时间步），每个时间步有512维度特征，那么输入BiLSTM前，维度会转换为（50, 1024）。

如果输入序列是奇数，要么就删除最后一个，要么就填充个0值帧，这样就能两两拼接。

1. 简介

pBLSTM（Pyramidal Bi-directional Long Short-Term Memory）是Bi-LSTM的一种变体，旨在通过减少序列长度来降低计算复杂度，同时保留时间序列的依赖性。它通过在运行Bi-LSTM之前进行下采样，即将相邻的时间步向量拼接在一起来减少序列长度。

2. 基本原理

- **输入序列**：输入数据通常是一个时间序列，其中每个时间步都是一个向量。假设输入序列为 $X = [X_0, X_1, X_2, \ldots, X_{N-1}]$，其中 X_i 是第 i 个时间步的特征向量。
- **下采样**：pBLSTM通过将相邻的时间步向量拼接在一起进行下采样，将序列长度减少一半。例如，输入序列中的

相邻向量 X_0 和 X_1 拼接成一个新的向量 $[X_0; X_1]$，依此类推。因此，下采样后的序列为：

$$X' = \left[[X_0; X_1],\ [X_2; X_3] \ldots [X_{N-2}; X_{N-1}]\right]$$

其中 $[X_0; X_1]$ 表示将向量 X_0 和 X_1 拼接后的新向量。

假设：

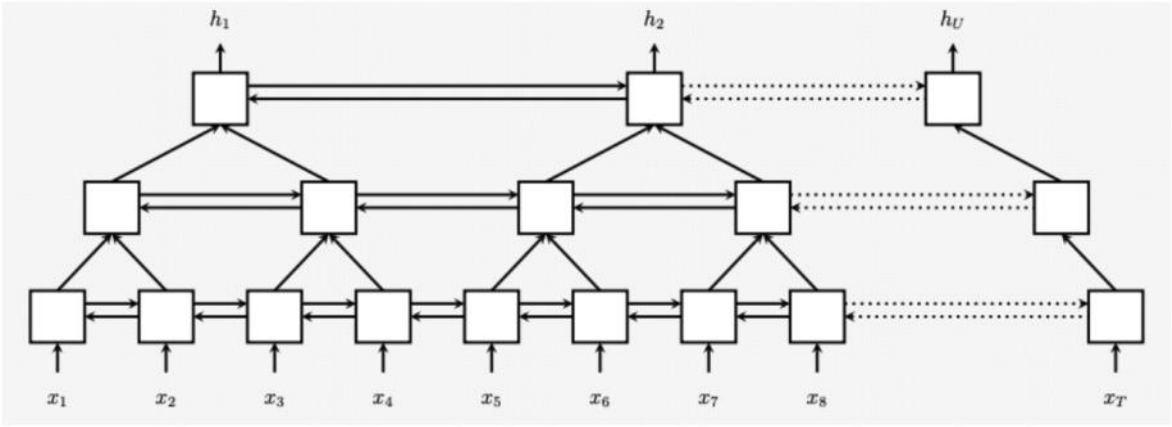
- X_0 是一个25维度的向量，表示为 $X_0 = [x_{0,1}, x_{0,2}, \ldots, x_{0,25}]$
- X_1 是另一个25维度的向量，表示为 $X_1 = [x_{1,1}, x_{1,2}, \ldots, x_{1,25}]$

在pBLSTM中，这两个向量的拼接操作如下：

$$[X_0; X_1] = [x_{0,1}, x_{0,2}, \ldots, x_{0,25}, x_{1,1}, x_{1,2}, \ldots, x_{1,25}]$$

拼接后的结果是一个50维度的向量，其中前25维是 X_0 的内容，后25维是 X_1 的内容。这个新的50维向量会被传递到下一层的Bi-LSTM中进行处理。

- **Bi-LSTM处理**：将下采样后的序列 X' 作为输入传递给常规的Bi-LSTM层。Bi-LSTM会并行处理从前向和后向的时间依赖性，生成双向的隐藏状态输出。
- **多层pBLSTM**：在多层pBLSTM结构中，第一层pBLSTM的输出将作为下一层pBLSTM的输入，继续进行下采样和Bi-LSTM处理。每一层的输出序列长度都会进一步减半。



3. 优点

- **减少计算复杂度**：通过下采样，序列长度在每一层都被减少一半，这大大降低了后续层的计算量，尤其是在处理长序列时，这一特性尤为重要。
- **保持时间依赖性**：虽然序列长度减少，但通过拼接相邻时间步的向量，仍然保留了时间序列中的关键依赖关系。

4. 适用场景

- **语音识别**：pBLSTM常用于语音识别系统中，因为语音信号往往是长时间序列，pBLSTM的下采样可以有效减少计算资源的消耗。
- **自然语言处理**：在处理长文本序列时，pBLSTM也可以应用来减少计算量，同时维持对上下文的理解。

问题3：BN层出来后的数据如何重组进入pBLSTM层?

BN出来后，维度是：[batchsize, 最大帧数， 256]。后续要进入BILSTM了，但是这里还有个问题，即batchsize中每个

样本，除了帧数最长的那个样本，其它每一个后面都有0填充帧。

我们知道BILSTM是自循环计算的，即上一个时间步的LSTM结果会和本次时间步的输入进行拼接然后进行LSTM计算，这些填充的0值帧（即本帧的256个特征全是0值），也会参与LSTM计算并有非0值结果传到下一个时间步。这对后续反向传播、精度是有不良影响的，并且也加大了LSTM的无效计算。

如何解决这个问题？我们要对数据进行重组打包，让LSTM不对填充值进行计算。具体示意图如下：

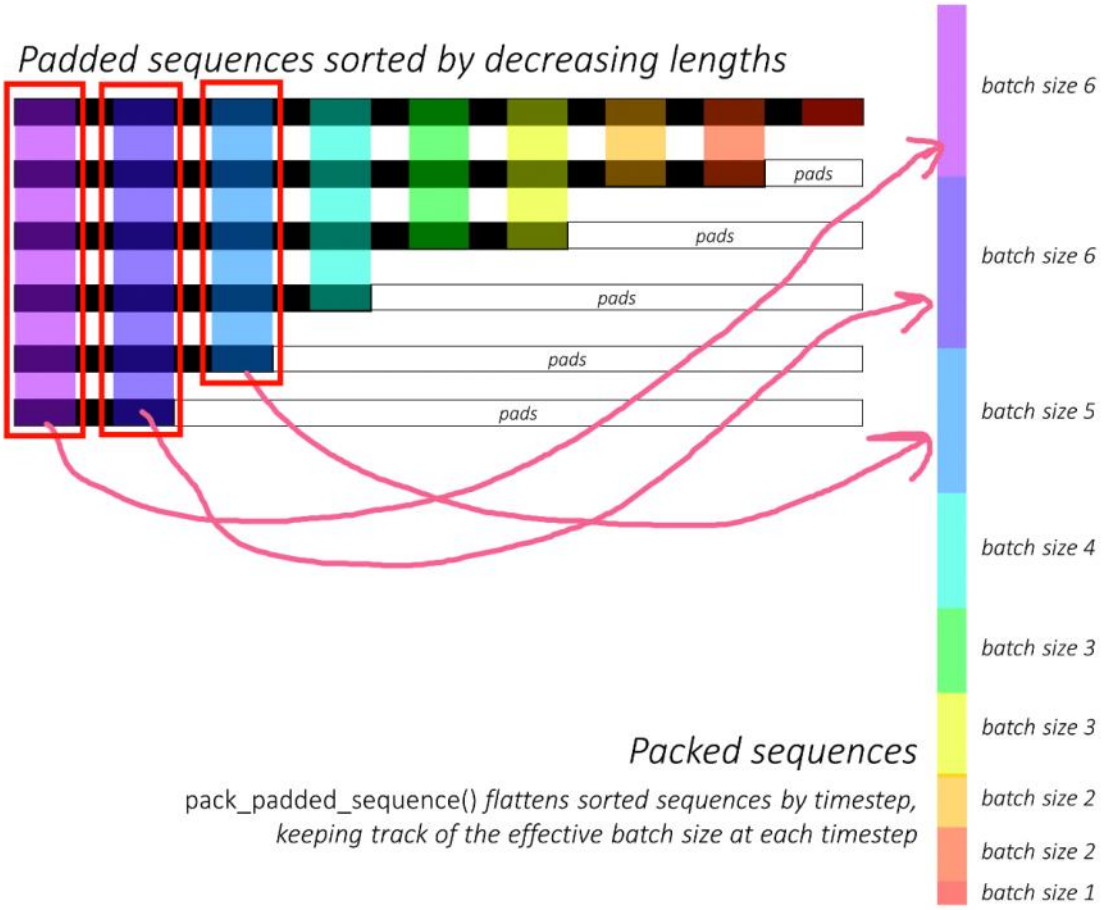


Figure 11: Packed padded sequence

假设输入维度是[6, 9, 特征数]，即batchsize=6，最大帧数=9，然后把这6个样本按照序列真实（非填充）长度从高到低排序，变成上图所示。它们中有5个样本有填充帧，我们希望它们进入LSTM后：

- 在第一个时间步，batch=6，即LSTM批处理上图【粉紫色】6个数据。
- 在第二个时间步，batch=6，同上。
- 在第三个时间步，batch=5，即只有批处理5个数据，最下面那个样本不进行计算了。
- 以此类推。

如果是BiLSTM，其实也一样，反向计算时，根据上图，第一个step，batch=1；第2个stepbatch=2，依次类推最后一个step，batch=6，完成整个推理。

上述数据操作和告诉LSTM的信息，用torch.nn.utils.rnn.pack_padded_sequence()实现。具体原理如下：

1. 批处理的基本思路

在 BiLSTM 中，批处理意味着同时处理多个序列，每个序列都可能具有不同的长度。为了能够在一个批次中处理这些不同长度的序列，我们通常会使用填充（padding）将所有序列填充到相同的长度。然后，通过 `pack_padded_sequence` 函数，我们可以将填充后的序列打包成一个紧凑的数据结构 `PackedSequence`，这样 BiLSTM 就可以高效地处理这些序列，而忽略填充部分。

2. BiLSTM 的批处理步骤

- **填充序列**：将所有序列填充到同样的长度，以便在同一个批次中处理。
- **打包序列**：使用 `pack_padded_sequence` 函数将填充后的序列打包，使 BiLSTM 只处理有效时间步，而不处理填充部分。
- **处理序列**：BiLSTM 使用打包的序列，在每个时间步上处理有效的输入。
- **解包序列**：使用 `pad_packed_sequence` 函数将 BiLSTM 的输出解包，恢复为填充的形式，以便后续处理。

3. 具体例子

假设我们有以下三个序列：

- 序列 A: 长度 5，数据为 `[a1, a2, a3, a4, a5]`
- 序列 B: 长度 3，数据为 `[b1, b2, b3]`
- 序列 C: 长度 2，数据为 `[c1, c2]`

为了批处理这些序列，首先我们需要将它们填充到相同的长度。假设最长的序列长度是 5，我们将所有序列填充为长度 5：

```
序列 A: [a1, a2, a3, a4, a5]
序列 B: [b1, b2, b3, 0, 0]  # 0 表示填充值
序列 C: [c1, c2, 0, 0, 0]
```

接着，我们使用 `pack_padded_sequence` 将这些序列打包：

```
import torch
from torch.nn.utils.rnn import pack_padded_sequence, pad_packed_sequence

# 序列的填充形式
sequences = torch.tensor([
    [a1, a2, a3, a4, a5],
    [b1, b2, b3, 0,  0 ],
    [c1, c2, 0,  0,  0 ]
], dtype=torch.float32)

# 每个序列的实际长度
lengths = torch.tensor([5, 3, 2])
```

```
# 打包序列
packed_sequences = pack_padded_sequence(sequences, lengths, batch_first=True,
enforce_sorted=False)
```

在 `PackedSequence` 中，数据会被打包成一个紧凑的形式：

```
packed_sequences.data: [a1, b1, c1, a2, b2, c2, a3, b3, a4, a5]
packed_sequences.batch_sizes: [3, 2, 2, 1, 1]
```

其中：

- `data` 包含了所有有效的序列数据，按时间步展平。
- `batch_sizes` 是一个张量，表示在每个时间步上参与计算的序列数量。例如，第一时间步上有 3 个序列（A, B, C）；第二时间步上有 2 个序列（A, B）；依次类推。

接着，打包后的数据会传递给 BiLSTM：

```
import torch.nn as nn

# 定义BiLSTM
bilstm = nn.LSTM(input_size=input_size, hidden_size=hidden_size, batch_first=True,
bidirectional=True)

# 将打包的序列传入BiLSTM
output_packed, (hn, cn) = bilstm(packed_sequences)
```

BiLSTM 会按照 `batch_sizes` 的信息逐时间步处理数据，只对有效的时间步进行计算，跳过填充部分。

最后，我们可以使用 `pad_packed_sequence` 将输出解包，恢复到填充形式：

```
output, _ = pad_packed_sequence(output_packed, batch_first=True)
```

解包后的输出 `output` 将恢复为 `[batch_size, max_seq_length, 2 * hidden_size]` 的形状，其中 `2 * hidden_size` 是因为 BiLSTM 是双向的。

问题4：LockedDropout作用?

1. **作用对象**：在 BiLSTM 层中，神经元在每个时间步上都会接收到输入数据并进行计算。`LockedDropout` 应用于这些神经元上，即对 BiLSTM 层的隐藏状态或输出进行 dropout。
2. **锁定时间步之间的 Dropout**：与传统的 dropout 不同，`LockedDropout` 不会在每个时间步上随机丢弃神经元，**而是在整个序列的所有时间步上应用相同的 dropout 掩码**。换句话说，如果某个神经元在第一个时间步被丢弃了，那么它在该序列中的其他时间步上也会被丢弃。
3. **BiLSTM中的效果**：在 BiLSTM 这样的序列模型中，`LockedDropout` 可以保持时间步之间的一致性。这对于 BiLSTM 来说尤为重要，因为 BiLSTM 需要在前向和后向传递中保持信息的连贯性，避免因为时间步之间的随机性导致的模型不稳定。

为什么在 BiLSTM 中使用 `LockedDropout`？

- **提高模型稳定性**：由于 BiLSTM 是处理序列数据的，保持每个时间步上网络结构的一致性有助于模型更好地捕捉序列中的模式。
- **避免随机性导致的信息丢失**：在普通 dropout 中，不同时间步随机丢弃神经元可能导致序列中的某些信息丢失，而 `LockedDropout` 通过锁定掩码来减少这种不一致性。

问题5：关于最后输出层的LogSoftmax

LogSoftmax是什么：

1. **输入向量**：
   - 假设你有一个模型的输出向量，长度为41（对应于41个音素类别）。这个向量中的每个元素是模型预测的原始分数（logit），通常在分类任务中，logit 可以看作是模型对于每个类别的“信心”值，但它们不是概率。
- **Softmax 操作**：
  - 如果我们先使用 `Softmax` 操作，这些分数会被转换为概率分布。`Softmax` 会将这些分数标准化，使它们的和等于1，每个值代表该类别的预测概率。
  - 计算公式为：

$$\text{Softmax}(x_i) = \frac{exp(x_i)}{\sum_{j=1}^{41} exp(x_j)}$$

- **LogSoftmax 操作**：
  - `LogSoftmax` 对 `Softmax` 的结果取对数，计算公式为：

$$\text{LogSoftmax}(x_i) = log\left(\frac{exp(x_i)}{\sum_{j=1}^{41} exp(x_j)}\right) = x_i - log\left(\sum_{j=1}^{41} exp(x_j)\right)$$

  - 这个对数操作的结果是将每个音素类别的概率转换为对数概率。
- **结果解释**：
  - 对41个音素类别进行 `LogSoftmax` 后，得到的每个元素值是一个负数，表示该类别的对数概率（log-probability）。这些值越接近0，表示模型对该类别的预测越有信心（即该类别的概率越高）。
  - 由于是对数概率，这些值的和不会等于1，而是所有类别概率的对数和为负数。

举例说明

假设经过 `LogSoftmax` 操作后，你得到一个长度为41的向量，如下：

$$\text{LogSoftmax Output} = [-3.0, -2.5, -4.0, \ldots, -3.2]$$

- 这里的 `-2.5` 表示模型预测的第2个音素类别的对数概率。
- 对应的概率可以通过取指数得到：

$$\text{Probability} = exp(-2.5)$$

- `LogSoftmax` 值越接近于0（对数概率越大），该类别的预测概率越高。

为什么一定要用LogSoftmax？

因为损失函数用的torch.nn.CTCLoss，所以最后一层的预测要用LogSoftmax。原因是：

难道用softmax输出不能直接用torch.nn.CTCLoss吗

不能直接使用 `Softmax` 输出来计算 `torch.nn.CTCLoss`，必须先将 `Softmax` 输出转换为 `LogSoftmax` 输出。这是因为 `torch.nn.CTCLoss` 的实现依赖于对数概率（log probabilities）而不是直接的概率值。

原因解释：

1. **对数概率计算的要求**：
   - `CTCLoss` 计算过程中涉及大量的对数运算。直接使用 `Softmax` 输出的概率值会导致数值稳定性问题，因为在计算对数时，概率值接近0时会产生非常大的负值，概率值接近1时会产生接近零的值。
   - 通过 `LogSoftmax`，我们直接获得了对数概率，这样可以避免计算中可能出现的数值下溢或上溢问题。
2. **数值稳定性**：
   - 使用 `LogSoftmax` 代替 `Softmax` 然后再取对数，能够简化计算并提高数值稳定性。`LogSoftmax` 通过结合了 `Softmax` 和对数运算，避免了可能的数值精度问题。
   - 如果直接使用 `Softmax` 输出然后在 `CTCLoss` 中计算对数，可能会引入额外的数值不稳定性，尤其是在处理非常长的序列或具有极端概率值时。
3. **CTCLoss 的预期输入**：
   - `torch.nn.CTCLoss` 的实现假设输入的是对数概率，因此它不会在内部再对输入进行 `LogSoftmax` 操作。如果输入的是 `Softmax` 概率值，而不是 `LogSoftmax` 结果，计算出的损失值会不正确。

4 CTC解码

本项目踩的坑：不要用ctcdecode和pyctcdecode 库用来CTC解码

CMU S23版启动笔记本中默认用的import ctcdecode，但是这个库年代久远几乎废弃了，自己也装不上放弃了。也尝试了S24版某CMU学生的代码使用的from pyctcdecode import build_ctcdecoder，发现也是巨坑。一方面S24版某学生的代码，感觉上传GitHub前进行了大幅修改，代码前半段和后半段根本不搭，有点像自己代码复原到启动代码感觉，根本使用不了。另一方面，build_ctcdecoder文档也很差，主要是传参label时会自动新增一个BLANK，但是我label已经有代表BLANK的符号了，它还是会新增一个。如果我删除我自己label中的BLANK符号，但我又不知道它内

部到底会新增一个什么字符串作为BLANK含义，因为我后续需要用模型预测的音素字符串和真实label 音素字符串计算距离。

上面两个库搞了很久，接口就是没打通。后面详细研究了一下pytorch的from torchaudio.models.decoder import cuda_ctc_decoder。好用多了，库的设计和文档也比上述两个好的多~

这里不得不说一句，pytorch做的真好（哭死），没有pytorch太多东西会搞醉你。

使用pytorch的cuda_ctc_decoder函数解码

文档、接口、测试用例可以去看下。只是解码后的数据结构文档中没说，需要单步调试时自己查看一下。具体看我的代码。

5 Levenshtein距离（编辑距离）

Levenshtein距离，也称为编辑距离（Edit Distance），是衡量两个字符串之间相似程度的一种方法。其定义是将一个字符串转换成另一个字符串所需的最少编辑操作次数。编辑操作包括以下三种：

1. **插入一个字符**（Insert a character）：在字符串的任意位置插入一个字符。
2. **删除一个字符**（Delete a character）：从字符串中删除一个字符。
3. **替换一个字符**（Replace a character）：将字符串中的某个字符替换为另一个字符。

Levenshtein距离的计算过程

动态规划算法

Levenshtein距离通常使用动态规划算法来计算。设两个字符串分别为 `A` 和 `B`，长度分别为 `m` 和 `n`。定义一个 `(m+1) x (n+1)` 的二维矩阵 `D`，其中 `D[i][j]` 表示将字符串 `A[0:i]` 转换成字符串 `B[0:j]` 所需的最小编辑距离。矩阵 `D` 的计算方式如下：

1. **初始化边界条件**：
   - `D[0][0] = 0`，表示将空字符串转换为空字符串的编辑距离为0。
   - `D[i][0] = i`，表示将字符串 `A[0:i]` 转换为空字符串需要 `i` 次删除操作。
   - `D[0][j] = j`，表示将空字符串转换为字符串 `B[0:j]` 需要 `j` 次插入操作。

- **递推公式**：

  对于 `1 ≤ i ≤ m` 和 `1 ≤ j ≤ n`，有以下递推公式：

  - 如果 `A[i-1] == B[j-1]`，则 `D[i][j] = D[i-1][j-1]`。
  - 如果 `A[i-1] != B[j-1]`，则 `D[i][j] = min(D[i-1][j-1] + 1, D[i-1][j] + 1, D[i][j-1] + 1)`。

  其中：

- `D[i-1][j-1] + 1` 对应替换操作。
- `D[i-1][j] + 1` 对应删除操作。
- `D[i][j-1] + 1` 对应插入操作。

- **最终结果**：

  矩阵 `D` 的右下角元素 `D[m][n]` 即为字符串 `A` 和 `B` 的Levenshtein距离。

计算示例

假设 `A = "kitten"`，`B = "sitting"`，我们来计算这两个字符串的Levenshtein距离：

- 初始矩阵：

```
   ""  s  i  t  t  i  n  g
""  0  1  2  3  4  5  6  7
 k  1
 i  2
 t  3
 t  4
 e  5
 n  6
```

- 填充矩阵的过程：

```
   ""  s  i  t  t  i  n  g
""  0  1  2  3  4  5  6  7
 k  1  1  2  3  4  5  6  7
 i  2  2  1  2  3  4  5  6
 t  3  3  2  1  2  3  4  5
 t  4  4  3  2  1  2  3  4
 e  5  5  4  3  2  2  3  4
 n  6  6  5  4  3  3  2  3
```

- 最终结果：`D[6][7] = 3`。

因此，`kitten` 和 `sitting` 的Levenshtein距离为 `3`，即最少需要3次编辑操作将 `kitten` 转换为 `sitting`（替换 `k` 为 `s`，替换 `e` 为 `i`，插入 `g`）。

时间复杂度和空间复杂度

- **时间复杂度**：O(m*n)，其中 `m` 和 `n` 分别是两个字符串的长度。
- **空间复杂度**：O(m*n)，因为需要存储一个大小为 `(m+1) x (n+1)` 的矩阵。

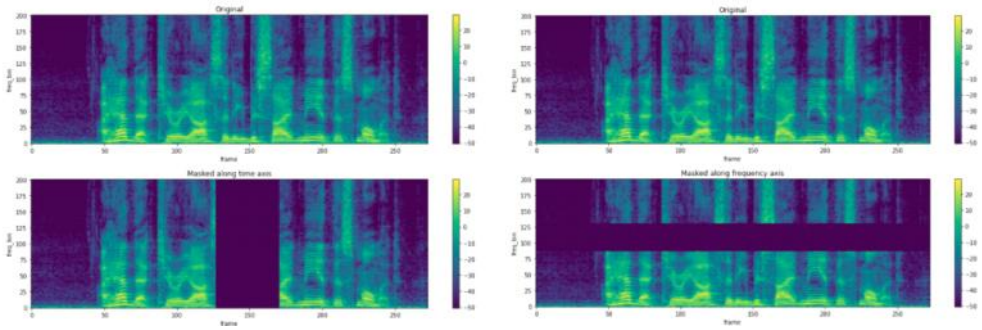
可以通过空间优化将空间复杂度降低到O(min(m, n))，但这是以只存储当前行和上一行的方式来实现的。

Levenshtein距离广泛应用于字符串匹配、拼写纠错、DNA序列比对等领域。

6 调参建议

以下源自课程项目辅导PPT的摘录：

1. 对【输入数据】进行标准化
2. 对【卷积】和【全连接层】使用不同的权重初始化方法
3. 学习器（step LR、ReduceLROnPlateau）
4. 优化器（AdamW, Adam）
5. 初始学习率：建议1e-3
6. dropout：很关键
    a. 在Embedding层, sequence层，classification层都能使用。
    b. 可以从小的 Dropout 率开始，并在模型训练后逐步增加。
7. LSTM 层的 Locked Dropout：
    a. 可以使用 Locked Dropout 在每个时间步中应用相同的 Dropout 掩码。
    b. 参考 PyTorch NLP 的 Locked Dropout 实现。
8. 对数据进行mask（时间维度、频率维度）：
    - Torch Audio Transforms [docs]
        - Time Masking
        - Frequency Masking

    a. 
9. 关于beam width：
    a. 较大的 Beam Width 可能会提供更好的结果，但建议在测试时将 Beam Width 保持在 50 以下，以节省计算资源。
    b. 有时设置 bw = 1（贪婪搜索）也能获得不错的结果。
    c. 建议：在每个 epoch 的验证过程中，不要使用过高的 Beam Width，因为这会增加每个 epoch 的时间。

7 训练结果

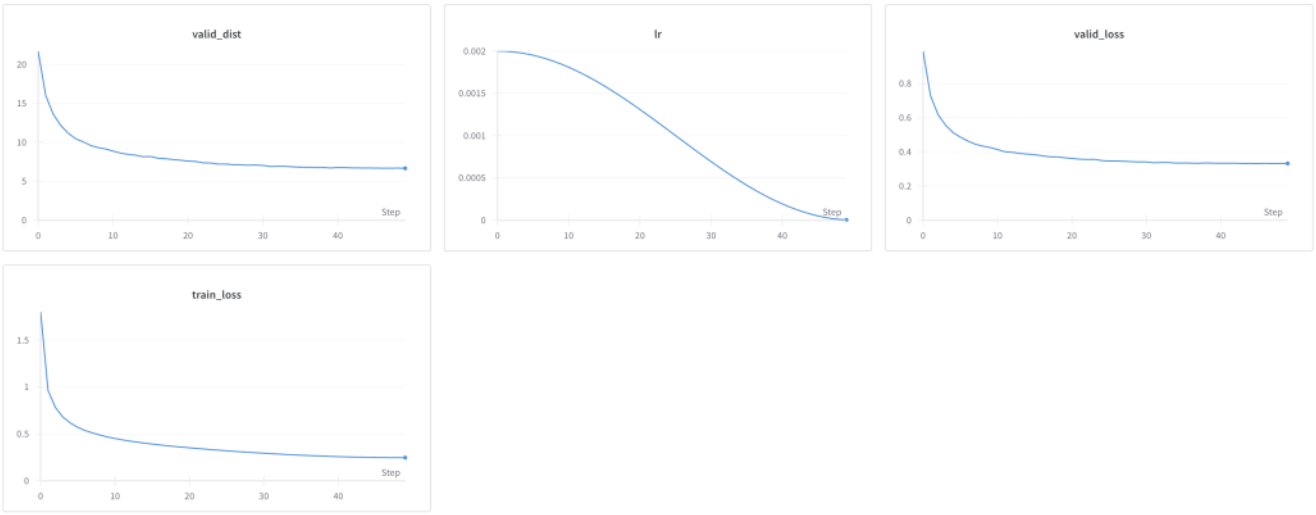
没有做消融实验（HW4更重要，也差不多，后面项目再做做），网络结构就是用的项目讲义中建议的架构，数据预处理只用了归一化，没使用写好的mask代码。关于其他人的调参，可查看GitHub上历届11785人分享的代码。

我的训练结果：

Epoch: 50/50

Train Loss 0.2462 Learning Rate 0.0000030

Val Dist **6.6595** Val Loss 0.3319



S24界：

- top1精度 = 4.00572
- top50精度 = 5.40362

本项目训练完后模型的输出形态：

```
pred_string = '|SiprEsthIzh@ndjEntliAndgr@tAtyud|wRAntyuh@piDEn@tcl|'
label_string = '|SiprEsthIzh@ndjEntliIngr@tAtud|wRAntyuh@piDEn@tcl|'
```