Project 2：LVCSR 系统搭建

518021910698 薛春宇

1. Baseline (70 分)

本章首先在 1.1 节中介绍 Kaldi 的安装及环境配置，并在 1.2 节中介绍官方 recipe 在实际训练前的数据处理与特征提取步骤；完成后，本章将在 1.3 节的模型训练中介绍官方 recipe 训练出 SAT+GMM-HMM 模型的完整流程，并在 1.4 节中给出不同阶段训练系统对应的评分结果，以及相应的性能分析。

1.1. 环境配置

本项目的实验平台是 *Linux 20.04 with kernel 5.8.0-55-generic*，通过执行如下指令流完成 Kaldi 环境的安装和配置：

```
git clone https://github.com/kaldi-asr/kaldi.git
cd kaldi/tools/extras
./check_dependencies.sh      # 检查依赖
#（安装缺少的依赖软件包）
cd ..
make -j 6                    # 并行编译，6位CPU数量
cd ../src
./configure                  # 配置
make depend
make                         # 编译安装
```

上述指令流执行完成后，可使用独立词语音识别 yesno 例程来进行验证。分别执行如下指令：

```
cd kaldi/egs/yesno/s5
./run.sh
```

执行完成后系统评分结果输出如图1所示。可以看到，系统对 yesno 数据集中共 232 个词语的预测结构全部正确，字错误率 WER 为 0。该实验结果说明 Kaldi 已正确安装和配置。

```
decode.sh: feature type is delta
steps/diagnostic/analyze_lats.sh --cmd utils/run.pl exp/mono0a/graph_tgpr exp/mono0a/decode_test_yesno
run.pl: job failed, log is in exp/mono0a/decode_test_yesno/log/analyze_alignments.log
local/score.sh --cmd utils/run.pl data/test_yesno exp/mono0a/graph_tgpr exp/mono0a/decode_test_yesno
local/score.sh: scoring with word insertion penalty=0.0,0.5,1.0
%WER 0.00 [ 0 / 232, 0 ins, 0 del, 0 sub ] exp/mono0a/decode_test_yesno/wer_10_0.0
root@ubuntu:/home/dicardo/Kaldi/kaldi/egs/yesno/s5#
```

图 1: *yesno* 例程评分结果

对于本项目所使用的 AIShell 例程，需要在运行前进行一定的配置。除了实验指导中给出的数据准备，脚本替换和任务运行方式修改等操作之外，还需要根据主机配置修改并行进程数和内存大小等参数。具体来说，我们在 *run.sh* 脚本里将所有并行进程参数 *nj* 设置为 *$(nproc)*，该数值等于实验主机的 CPU 数目；同时，我们在 *cmd.sh* 中设置脚本执行内存为 *12G*，该数值等于实验主机的总内存大小。

1.2. 数据处理及特征提取

本节中，我们将分析 AIShell 官方 recipe 在实际训练前的数据处理与特征提取模块，并给出模块运行的结果。

在 *run.sh* 脚本的第 43-48 行中，例程对三个数据集分别依次调用了 *make_mfcc_pitch.sh*，*compute_cmvn_stats.sh* 和 *fix_data_dir.sh* 三个脚本，实现 MFCC，基频 pitch 和倒谱特征 CMVN 的特征提取，以及数据的规整化。注意，在执行上述指令之前，需要保证相应 AIShell 语料库已经准备完毕。

```
mfccdir=mfcc
for x in train dev test; do
  steps/make_mfcc_pitch.sh --cmd "$train_cmd" --nj
      $(nproc) data/$x exp/make_mfcc/$x $mfccdir
    || exit 1;
  steps/compute_cmvn_stats.sh data/$x exp/
    make_mfcc/$x $mfccdir || exit 1;
  utils/fix_data_dir.sh data/$x || exit 1;
done
```

在 *make_mfcc_pitch.sh* 脚本的第 11-14 行中有如下字段：

```
nj=4
cmd=run.pl
mfcc_config=conf/mfcc.conf
pitch_config=conf/pitch.conf
```

可以看出，上述脚本默认并行进程数为 4，且根据 *mfcc.conf* 和 *pitch.conf* 两个配置文件来设置 MFCC 和基频提取的参数细节。分别打开两个配置文件，发现仅配置了采样率一个参数，值为

16kHz，因此猜测如截断频率值等参数均使用脚本自带的默认大小，也可以通过命令行参数进行设置。该脚本分别提取说话人的 MFCC 和基频特征，并进行组合。此外，*compute_cmvn_stats.sh* 脚本被用来计算说话人的倒谱均值和方差统计量，以用作后续的语音识别。

最后，在完成上述特征生成后，会调用 *fix_data_dir.sh* 脚本来对数据进行格式规整。

本模块的运行结果见图2，提取的特征保存在*./kaldi/egs/aishell/s5/mfcc* 目录下，文件被保存为*.scp* 和*.ark* 格式。

图 2: 数据处理和特征提取运行结果

1.3. 模型训练

本节将在 1.2 节数据处理和特征提取的基础上，通过 1.3.1 节分别介绍语音技术中语言模型和声学模型的相关理论知识，给出二者在逻辑上的联系，并介绍说话人自适应 SAT 的相关知识，为模型训练提供理论基础；同时，本节将在 1.3.2 节中依次给出 AIShell 例程中语言与声学模型的实现过程，以及最终共同构成 SAT+GMM-HMM 模型的完整流程。

1.3.1. 理论基础

语言模型表示字或音素序列的发生概率，通过链式法则将整句的概率表示为句中每个词的概率之积。N-gram 语言模型则近似表示了一个较短历史情况的条件概率，当前词的概率仅依赖于前 $n-1$ 个前继词。假设词序列为 $W_1^N = [w_1, ..., w_N]$，若已知前 $k-1$ 个词序列中的词和词表 $V = \{v_1, ..., v_M\}$，则对第 k 个词的概率有：

$$P(w_k|W_{k-n+1}^{k-1}), w \in V \tag{1}$$

逐词获取上述条件概率后，按照前述的链式法则即可计算整句的概率。因此，语言模型的功能可以概括为将字词以概率的形式进行选择和组合，从若干可能的结果中保留最符合语言习惯的句子。

不同的是，声学模型的功能则可以概括为将语音的声学特征解码映射为音素或字词，是对说话人声学及语音学特征，以及环境实时变量的规范化表示，大多基于 HMM 模型进行建模，利用 LSTM+CTC 进行训练。声学模型的输入是离散或连续的多维语音特征向量，常采用混合高斯模型 GMM 对语音信号的分布进行拟合，其可以被表达为如下数学形式：

$$G(x) = \prod_{i=1}^{n} w_i \cdot G_i(x) \tag{2}$$

其中 $G_i(x)$ 是均值为 μ_i，方差为 σ_i 的高斯分布。当 n 足够大时，任何分布都可以通过这 n 个高斯分布的加权平均来逼近，对于 D 维随机矢量 x，其概率密度函数可表示为：

$$p(x|\lambda) = \sum_{i=1}^{M} w_i p_i(x) \tag{3}$$

其中 $p_i(x)$ 表示第 i 个概率密度分量，w_i 表示其权重。基于上述事实，声学模型通常被组织为 GMM-HMM 的模型形式。

综上所述，我们给出语言模型与声学模型在逻辑上的关系。首先，语言模型基于 lexicon 词典进行训练，获得从字词到音素的映射关系；随后，声学模型基于从音频文件得到的特征向量进行训练，得到预测到的音素，再和语言模型一起进行解码，将音素和字词预测为句子。

最后介绍说话人自适应技术 SAT，其作用是缓解实际数据与训练得到的三音素模型在声学条件中不匹配的问题，包括说话人的口音等声学特性，以及环境背景音等。SAT 利用特定说话人的数据对说话人无关的码本进行改造，进而得到对

说话人具有泛化性和自适应特性的码本，以更好地适配当前说话人的声学特性。

1.3.2. 实现流程

本小节将给出 AIShell 例程里的 *run.sh* 从无到有地构建 SAT+GMM-HMM 模型的完整流程，模型整体架构如图 3 所示。

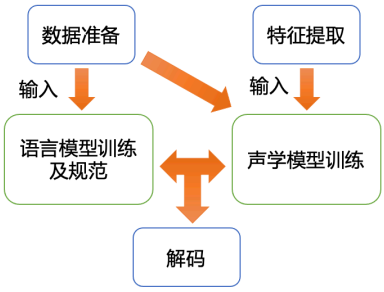

图 3: *AIShell 例程的模型架构*

首先，基于准备好的 lexicon 词典及语料标注，recipe 依次调用如下脚本来生成语言模型：

```
# LM training
local/aishell_train_lms.sh || exit 1;
# G compilation, check LG composition
utils/format_lm.sh data/lang data/local/lm/3gram-mincount/lm_unpruned.gz \
    data/local/dict/lexicon.txt data/lang_test || exit 1;
```

上述 recipe 涉及到两个重要的脚本，分别是 *aishell_train_lms.sh* 和 *format_lm.sh*。前者无参数输入，基于先前准备好的语料集进行训练，结果如图 4 所示；后者以预先生成好的 *data/lang* 为语料库，*lexicon.txt* 为词典集，生成规范的语言模型并保存在 *data/lang_test* 目录下。

```
Computing final perplexity
Building ARPA LM (perplexity computation is in background)
interpolate_ngrams: 137074 words in wordslist
interpolate_ngrams: 137074 words in wordslist
Perplexity over 15844.000000 words is 1581.623712
Perplexity over 15844.000000 words (excluding 0.000000 OOVs) is 1581.623712
1581.623712
Done training LM of type 3gram-mincount
```

图 4: 基于小语料库的语言模型训练

声学模型的构建则更为复杂，共分为单音素训练，三音素模型，线性判别分析和最大似然线性转换，以及最后的说话人自适应，每个模块都需依次进行训练、构图、解码和对齐等步骤，每步均以前面某步的输出为输入，其逻辑关系见图 7。

单音素训练需要共进行 40 轮迭代，每两次迭代执行一次对齐操作；三音素模型模块以单音素模型为输入，对上下文相关的三音素模型进行训练；接下来对输出进行线性判别分析，以及最大似然线性转换，并在最后通过训练说话人自适应 SAT 模块，实现最终的 SAT+GMM-HMM 模型。

在进行模型实现的过程中，我遇到了部分问题，并在查阅资料以及和同学讨论后成功解决。例如，在开始运行 *run.sh* 时，我得到了如下报错：

```
local/aishell_train_lms.sh: train_lm.sh is not found. That might mean it's not installed
```

容易发现 error 的原因是未找到 *train_lm.sh*。查阅相关资料后，进入*./kaldi/tools/extras* 分别运行如下指令，即可成功解决。

```
./install_kaldi_lm.sh
source env.sh
```

此外，在执行 *decode.sh* 脚本时得到报错：

```
run.pl: job failed, log is in exp/mono/decode_dev/log/analyze_alignments.log
```

查看上述 log 日志文件后发现：

```
# gunzip -c exp/mono/decode_dev/phone_stats.*.gz | steps/diagnostic/analyze_phone_length_stats.py exp/mono/graph
# Started at Fri Jun 25 13:39:51 CST 2021
#
/usr/bin/env: 'python': No such file or directory
# Accounting: time=0 threads=1
# Ended (code 127) at Fri Jun 25 13:39:51 CST 2021, elapsed time 0 seconds
```

推测是已有 python 环境的路径问题。因此卸载已安装的 python，使用 *apt install python* 指令重新安装缺失的 python 软件包，问题解决。

1.4. 实验结果

本节首先给出 recipe 不同训练阶段对应的 CER 和 WER 评分结果在目录中的相对路径，并给出获取各阶段最优结果的指令；随后，本节将给出各阶段对应的最优评分标准，并在此基础上进行必要的分析。注意，本节目录的默认路径为 *kaldi/egs/aishell/s5*。

各阶段对应的评分结果均保存在*./exp* 目录下，如单音素训练阶段对应的即为 *mono* 子目录。

每个子目录下均保存不同训练轮次的 WER/CER 得分，因此需要分别取两种评分标准的最优结果。

我们使用如下指令来分别获取字错误率 CER 和词错误率 WER 各阶段的最优结果：

```
cd kaldi/egs/aishell/s5
# CER
for x in exp/*/decode_test; do [ -d $x ] && grep
    WER $x/cer_* | utils/best_wer.sh; done 2>/dev
    /null
# WER
for x in exp/*/decode_test; do [ -d $x ] && grep
    WER $x/wer_* | utils/best_wer.sh; done 2>/dev
    /null
```

运行上述指令可以分别得到如下结果：

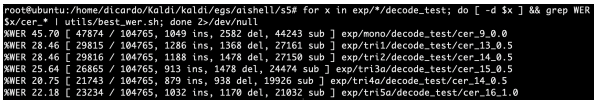

图 5: 小语料库模型的 *CER* 结果

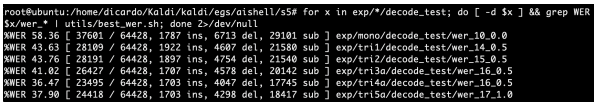

图 6: 小语料库模型的 *WER* 结果

将上述实验结果通过表 1 进行表示，并基于实验结果进行相应的对比分析，两行评估指标分别从上而下为 CER 和 WER。

表 1: 小语料库 *GMM-HMM* 模型的评分情况

Mono	Tri1	Tri2	Tri3a	Tri4a	Tri5a
45.70%	28.46%	28.46%	25.64%	**20.75%**	22.18%
58.36%	43.63%	43.76%	41.02%	**36.47%**	37.90%

通过上述结果，我们可以发现如下现象：

- 字错误率 CER 恒小于等于词错误率 WER；
- 除大 SAT 系统外，CER/WER 大体都呈逐训练阶段递减趋势。

现象一的原因显然，而现象二则表明从单音素训练，到三音素模型，到线性判别分析和最大似然线性转换，再到最后的说话人自适应，模型性能是逐步优化的，这也证明了该部分实验的正确性。大 SAT 系统性能较差的原因可能是系统欠拟合。

2. 语料数量对模型性能的影响（15 分）

本章首先在 2.1 节中给出将语言模型解码时使用的语料库替换为原 AIShell-1 训练集的修改方法，以此提供更优的解码效果；其次，在 2.2 节中给出了不同大小的语料库在模型解码时的性能对比，以此说明语料数量对模型训练的影响。

2.1. 修改方法

为了进行外部语料库的链接，我们需要对相应脚本内的语料库相对地址进行修改。具体来说，该修改针对 *aishell_train_lms.sh* 的如下字段：

```
# Line 7
text=data/local/train/text
```

该链接的作用是将 AIShell 的部分语料库在经过子集划分后得到的 train 训练集，链接到语言模型训练的语料输入。相应的数据集划分在模型训练前的数据处理步骤中完成。

我们将上述字段修改为大数据集中 *train_large.txt* 的绝对地址，并删除 *data/local/lm* 的内容，以实现外部大语料库链接到语言模型的训练中。

```
text=/home/dicardo/Kaldi/kaldi/dataset/
    data_aishell/transcript/train_large.txt
```

完成链接后，我们还需要重新执行 *run.sh* 中的部分步骤，以获得新语言模型的解码结果。分析各脚本的输入及输出目录可知，*run.sh* 中某模块（如单音素训练）实现的逻辑架构如图 7 所示。其中，语言模型的训练参照 1.3.2 节中的描述；构造译码图时以语言模型和模块训练的结果为输入，并基于译码图进行解码；对齐仅针对模块训练的结果，且被用作下一个模块的训练。

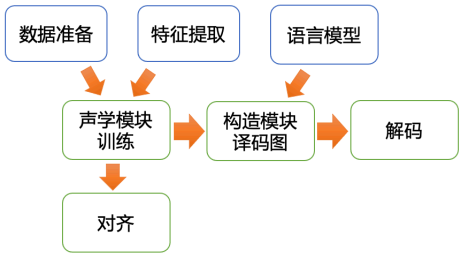

图 7: 模块的逻辑架构

基于上述分析，我们可以确定在对 *aishell_train_lms.sh* 内的 *text* 链接进行修改，替换为大语料库 *train_large.txt* 之后，需要重新执行的步骤包括：

- 语言模型的训练和规范；
- 声学模型中各模块的译码图构造及解码。

重新训练后基于大语料库的模型结果如图 8 所示，与图 4 所示的小语料库语言模型相比，在更大的字词集合上取得了更小的困惑度 Perplexity。

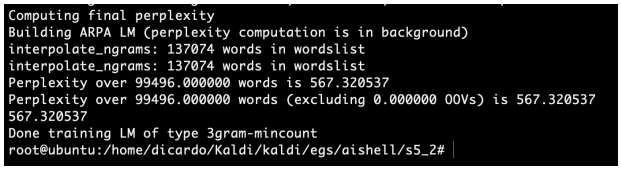

图 8: 基于大语料库的语言模型训练

2.2. 性能对比

按照 2.1 节中的方法对大语料库语言模型进行训练，并分别与各阶段声学模型进行重新解码，并执行 1.4 节给出的查询命令，得到的评估结果如图 9 和 10 所示：

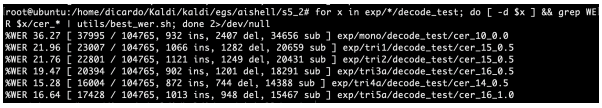

图 9: 大语料库模型的 *CER* 结果

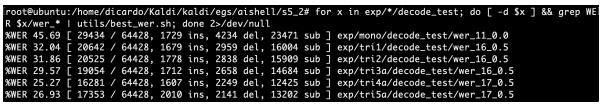

图 10: 大语料库模型的 *WER* 结果

表 2 给出了替换语料库后，新旧语言模型各自的最优评分结果的对比。可以看出，当采用规模更大的语料库对语言模型进行训练后，GMM-HMM 模型的性能显著增强。

下面分析上述现象的原因：增加语料数量能够很好地缓解模型过拟合的现象，同时提高模型的泛化性，因此对模型性能有着相对显著的提升效果。

表 2: （修改语料数量后）新旧模型性能对比

	Mini	Large
CER	20.75%	**15.28%**
WER	36.47%	**25.27%**

3. 训练 DNN-HMM 系统 (15 分)

本章将在中介绍 DNN_HMM 的相关理论知识，并在 3.2 节中给出模型的完整训练流程。最后，本章将在 3.3 节中以表格的形式给出 GMM_HMM 和 DNN_HMM 模型的性能对比。

3.1. 理论知识

GMM 和 DNN 都是对观测序列概率分布的拟合，并将其作为 HMM 的观测状态概率矩阵。二者主要的差别在于 DNN 能够根据观测值通过反向传播去求状态值，属于监督学习，其经过 softmax 层后即得到后验概率。

GMM 可以表征状态，输出为似然概率，且相邻的 GMM 之间相关性较弱；作为判别模型的 DNN 直接对给定的观察序列 Y 后状态的分布进行建模，输出则为经贝叶斯转换后得到的后验概率。DNN 为多帧输入，因此在建模时具有较强的上下文信息，并同时引入非线性的效果。

3.2. 训练流程

为了训练 DNN-HMM 模型，我们首先需要训练 GMM-HMM 模型，得到音素（状态）的后验。注意，以上步骤包括数据对齐等数据准备操作。

完成上述准备操作后，我们调用*./local/nnet3/run_tdnn.sh* 脚本，来对 DNN-HMM 模型进行训练和使用。该脚本共设置 4 轮迭代，初始学习率为 0.0015，最终学习率为 0.00015，且并行任务数始终为 3。在该脚本中，又分别调用了 *steps/nnet3/train_dnn.py* 和 *steps/nnet3/decode.sh* 脚本，下面将分别进行分析。

train_dnn.py 脚本的主要函数为 *train()* 函数，该函数根据传入的相应参数对 DNN 模型进

行相关配置，并进行 *num_iters* 次的迭代，来对模型进行参数更新。

decode.sh 同 GMM 模型中的类似，目的是对 DNN 网络中输出的音素（最大概率对应的）进行解码，进而得到最终预测的词句。

3.3. 性能对比

由于 2.2 节中已经证明基于大语料库的语言模型具有更优的性能，因此本节 DNN-HMM 的实现选择使用 2.1 节中描述的语言模型。

本项目可以通过如下命令来获取 DNN 模型的评分结果：

```
# CER
for x in exp/*/*/decode_test; do [ -d $x ] && grep WER $x/cer_* | utils/best_wer.sh; done 2>/dev/null
# WER
for x in exp/*/*/decode_test; do [ -d $x ] && grep WER $x/wer_* | utils/best_wer.sh; done 2>/dev/null
```

上述命令相应的输出如图 11 所示：

图 11: *DNN-HMM* 模型的评估结果

通过表格的形式与 2.1 节中的 GMM-HMM 模型做性能对比，结果如表 3 所示。从上述实验结果可以得出结论，将 GMM 替换为 DNN 网络对整个模型的性能提升较为明显。

表 3: *DNN* 与 *GMM* 模型的性能对比

	GMM	DNN
CER	15.28%	**13.28%**
WER	25.27%	**22.75%**

4. 优化系统 (20 分)

本章在 4.1 节中给出基于 kaldi 中 AIShell 例程现有 recipe 方案进行的一些改进，包括模型训练速度和模型性能两方面的优化；随后，本章在 4.2 节中给出与 3.2 节中原始 DNN-GMM 模型方案的性能对比。

4.1. 方法

在模型训练速度方面，我们可以选择将 *run.sh* 脚本中的译码图构建和解码步骤全部注释，以大幅提高模型训练的效率。这样做的原因是，如 2.1 节所分析，AIShell 例程中的解码步骤与声学模块训练是单向独立的，解码步骤并不会对模型训练产生影响，其作用仅是测各阶段的模型性能。

在模型性能方面，我们选择针对 DNN 训练时的学习率变化曲线以及 batchsize 进行优化。我们知道，更大的 batchsize 会带来更快的训练速度和更稳定的性能，但会造成模型泛化性的下降。因此，为了进一步提高模型的性能，我们选择将 batchsize 从原来的 128 降低为 64。同时，为了缓解由于 batchsize 减小带来的不稳定影响，我们将学习率的变化曲线整体降低并拉伸，改为从 0.01 动态变化为 0.00005，同时加倍迭代次数。

此外，为了保证模型的正常训练，我们还需在 *cmd.sh* 中修改运行内存为 *8G*，并在 *run_tdnn.sh* 中设置*–egs.opts "–nj 1"*。

4.2. 性能对比

系统优化后的模型评分输出如图 12 所示：

图 12: 优化后的 *DNN-HMM* 模型评估结果

经过 4.1节阐述的优化方案后，模型性能小幅提升，见表 4。

表 4: 系统优化前后的模型性能对比

	DNN1	DNN2
CER	13.28%	**13.18%**
WER	22.75%	**22.64%**

5. 最佳性能

CER=**13.18%**， WER=**22.64%**