Chinese Doc of Kaldi

shiweipku

Published with GitBook

目錄

Kaldi中文手册

翻译自Kaldi官方文档，并后附一些译者认为很有价值的资料。注意，本中文翻译文档与原英文文档不一定完全同步。要想获取最新的Kaldi文档，请访问Kaldi项目的官方文档，地址：http://kaldi.sourceforge.net/index.html 或者 http://www.kaldi-asr.org/doc。

关于Kaldi项目

下载和安装Kaldi

安装和运行**Kaldi**所必须的软件

法律相关

Kaldi教程

数据准备

翻译：V(shiwei@sz.pku.edu.cn)

时间：2014年5月

介绍

在运行完示例脚本后（见Kaldi教程），你可能会想用自己的数据在Kaldi上跑一下。本节主要讲述如何准备相关数据。本页的读者应该使用最新版本的实例脚本（即在脚本目录下被命名为s5的那些，例如 `egs/rm/s5` ）。 另外，除了阅读本页所述内容外，你还可以阅读脚本目录下的那些数据准备相关的脚本。（译者：结合起来看更易理解。） 在顶层的 `run.sh` 脚本（例如 `egs/rm/s5/run.sh` ）中，最前面的几行命令都是和数据准备相关的，代表数据准备的不同步骤。子目录 `local/` 下的脚本都是和数据集相关的。例如，Resource Management（RM）数据集相应的脚本就是 `local/rm_data_prep.sh` 。对RM数据集来说，这几行数据准备的命令为：

```
local/rm_data_prep.sh /export/corpora5/LDC/LDC93S3A/rm_comp || exit 1;

utils/prepare_lang.sh data/local/dict '!SIL' data/local/lang data/lang || exit 1;

local/rm_prepare_grammar.sh || exit 1;
```

而对WSJ来说，命令为：

```
wsj0=/export/corpora5/LDC/LDC93S6B
wsj1=/export/corpora5/LDC/LDC94S13B

local/wsj_data_prep.sh $wsj0/??-{?,??}.? $wsj1/??-{?,??}.?  || exit 1;

local/wsj_prepare_dict.sh || exit 1;

utils/prepare_lang.sh data/local/dict "<SPOKEN_NOISE>" data/local/lang_tmp data/lang || e

local/wsj_format_data.sh || exit 1;
```

在WSJ的示例脚本中，上述命令之后还有一些训练语言模型的命令（根据标注重新训练，而不是使用LDC提供的）， 但是上述几条命令是最重要的。

数据准备阶段的输出包含两部分。一部分与“数据”相关（保存在诸如 `data/train/` 之类的目录下），另一部分 则与“语言”相关（保存在诸如 `data/lang/` 之类的目录下）。“数据”部分与数据集的录音相关，而“语言”部分 则与语言本身更相关的内容，例如发音字典、音素集合以及其他很多Kaldi需要的关于音素的额外信息。 如果你想用已有的识别系统和语言模型对你的数据进行解码，那么你只需要重写“数据”部分。

数据准备-- 数据部分.

举个数据准备阶段关于“数据”部分的例子，请查看任何一个示例脚本目录下的 `data/train` 目录（假设你已经运行过一遍这些脚本了）。注意：目录名字 `data/train` 本身没有什么特别的。一些被命名为其他名字的目录，如 `data/eval2000` （为一个测试集建立的），有几乎差不多的目录结构和文件格式（说“几乎”是因为在测试集的目录下，可能含有“sgm”和“glm”文件，用于sclite评分）。我们以Switchboard数据为例，对应脚本在 `egs/swbd/s5` 下。

```
s5# ls data/train
cmvn.scp  feats.scp  reco2file_and_channel  segments  spk2utt  text  utt2spk  wav.scp
```

不是所有的文件都同等重要。如果要设置简单点，分段（segmentation）信息是不必要的（即一个文件里只有一段发音），你只需要自己创建 `utt2spk` 、 `text` 和 `wav.scp` ， `segments` 和 `reco2file_and_channel` 是可选的，根据实际需要决定是否创建。剩下的就都交给标准脚本。

下面我们会详细描述该目录下的这些文件。首先从那些需要你手动创建的文件开始。

需要手动创建的文件

文件“text”包含每段发音的标注。

```
s5# head -3 data/train/text
sw02001-A_000098-001156 HI UM YEAH I'D LIKE TO TALK ABOUT HOW YOU DRESS FOR WORK AND
sw02001-A_001980-002131 UM-HUM
sw02001-A_002736-002893 AND IS
```

每行的第一项是发音编号（utterance-id），可以是任意的文本字符串，但是如果在你的设置中还包含说话人信息， 你应该把说话人编号（speaker-id）作为发音编号的前缀。这对于音频文件的排序非常重要。发音编号后面跟着的 是每段发音的标注。你不用保证这里出现的每一个字都出现在你的词汇表中。词汇表之外的词会被映射到 `data/lang/oov.txt` 中。 注意：尽管在这个特别的例子中，我们用下划线分割了发音编号中的“说话人”和“发音”部分，但是通常用破折号（“-”）会更安全 一点。这是因为破折号的ASCII值更小。有人向我指出说，如果使用

下划线，并且说话人编号的长度不一，在某些特殊的情况下， 如果使用标准"C"语言风格对字符串进行排序，说话人编号和对应的发音编号会被排成不同的顺序。 另外一个很重要的文件是**wav.scp**。在Switchboard例子中，

```
s5# head -3 data/train/wav.scp
sw02001-A /home/dpovey/kaldi-trunk/tools/sph2pipe_v2.5/sph2pipe -f wav -p -c 1 /export/co
sw02001-B /home/dpovey/kaldi-trunk/tools/sph2pipe_v2.5/sph2pipe -f wav -p -c 2 /export/co
```

这个文件的格式是

```
<recording-id> <extended-filename>
```

其中，“extended-filename”可能是一个实际的文件名，或者就像本例中所述那样，是一段提取wav格式文件的命令。 extended-filename末尾的管道符号表明，整个命令应该被解释为一个管道。等会我们会解释什么是“recording-id”， 但是首先，我们需要指出，如果“segments”文件不存在，“wav.scp”每一行的第一项就是发音编号。 在Switchboard设置中，我们有“segments”文件，所以下面我们就讨论一下这个文件。

```
s5# head -3 data/train/segments
sw02001-A_000098-001156 sw02001-A 0.98 11.56
sw02001-A_001980-002131 sw02001-A 19.8 21.31
sw02001-A_002736-002893 sw02001-A 27.36 28.93
```

"segments" 文件的格式是:

```
<utterance-id> <recording-id> <segment-begin> <segment-end>
```

其中，segment-begin和segment-end以秒为单位。它们指明了一段发音在一段录音中的时间偏移量。“recording-id” 和在“wav.scp”中使用的是同一个标识字符串。再次声明一下，这只是一个任意的标识字符串，你可以随便指定。 文件"reco2file_and_channel"只是在你用NIST的sclite工具对结果进行评分（计算错误率）的时候使用：

```
s5# head -3 data/train/reco2file_and_channel
sw02001-A sw02001 A
sw02001-B sw02001 B
sw02005-A sw02005 A
```

格式为：

```
<recording-id> <filename> <recording-side (A or B)>
```

filename通常是.sph文件的名字，当然需要去掉sph这个后缀；但是也可以是任何其他你在“stm”文件中使用的标识字符串。“录音方”（recording side）则是一个电话对话中两方通话（A或者B）的概念。如果不是两方通话，那么为保险起见最好 使用“A”。如果你并没有“stm”文件，或者你根本不知道这些都是什么东西，那么你可能就不需要reco2file_and_channel"文件。 最后一个需要你手动创建的文件是“utt2spk”。该文件指明某一段发音是哪一个说话人发出的。

```
s5# head -3 data/train/utt2spk
sw02001-A_000098-001156 2001-A
sw02001-A_001980-002131 2001-A
sw02001-A_002736-002893 2001-A
```

文件格式是：

```
<utterance-id> <speaker-id>
```

注意一点，说话人编号并不需要与说话人实际的名字完全一致——只需要大概能够猜出来就行。 在这种情况下，我们假定每一个说话方（电话对话的每一方）对应一个说话人。这可能不完全正确—— 有时一个说话人会把电话交给另外一个说话人，或者同一个说话人会在不同的对话中出现——但是上述假定 对我们来说也足够用了。如果你完全没有关于说话人的信息，你可以把发音编号当做说话人编号。那么 对应的文件格式就变为```

```。 在一些实例脚本中还出现了另外一个文件，它在Kaldi的识别系统的建立过程中只是偶尔使用。在Resource Management （RM）设置中该文件是这样的：

```
s5# head -3 ../../rm/s5/data/train/spk2gender
adg0 f
ahh0 m
ajp0 m
```

这个文件根据说话人的性别，将每个说话人编号映射为“m”或者“f”。 上述所有文件都应该被排序。如果没有排序，你在运行脚本的时候就会出现错误。在 \ref io_sec_tables 中我们解释了为什么需要这样。这与（Kaldi的）I/O框架有关，归根到底是因为排序后的文件可以在一些不支持 fseek()的流中——例如，含有管道的命令——提供类似于随机存取查找的功能。需要Kaldi程序都会从其他Kaldi命令 中读取多个管道流，读入各种不同类型的对象，然后对不同输入做一些类似于“合并然后排序”的处理。既然要合并排序， 当然需要输入是经过排序的。小心确保你的shell环境变量LC_ALL定义为“C”。例如，在bash中，你需要这样做：

```
export LC_ALL=C
```

如果你不这样做，这些文件的排序方式会与C++排序字符串的方式不一样，Kaldi就会崩溃。这一点我已经再三强调过了！
```

如果你的数据集中包含NIST提供的测试集，其中有“stm”和“glm”文件可以用作计算WER，那么你可以直接把这些文件拷贝 到数据目录下，并分别命名为“stm”和“glm”。注意，我们把评分脚本 `score.sh` （可以计算WER）放到 `local/` 下， 这意味着该脚本是与数据集相关的。不是所有的示例设置下的评分脚本都能识别stm和glm文件。能够使用这些文件的一个 例子在Switchboard设置里，即 `egs/swbd/s5/local/score_sclite.sh` 。如果检测到你有stm和glm文件该脚本会被顶层的评分脚本 `egs/swbd/s5/local/score.sh` 调用。

不需要手动创建的文件

数据目录下的其他文件可以由前述你提供的文件所生成。你可以用如下的一条命令创建“spk2utt”文件（ 这是一条从 `egs/rm/s5/local/rm_data_prep.sh` 中摘取的命令）：

```
utils/utt2spk_to_spk2utt.pl data/train/utt2spk > data/train/spk2utt
```

这是因为utt2spk和spk2utt文件中包含的信息是一样的。spk2utt文件的格式是：

```
<speaker-id> <utterance-id1> <utterance-id2> ....
```

下面我们讲一讲 `feats.scp` 文件.

```
s5# head -3 data/train/feats.scp
sw02001-A_000098-001156 /home/dpovey/kaldi-trunk/egs/swbd/s5/mfcc/raw_mfcc_train.1.ark:24
sw02001-A_001980-002131 /home/dpovey/kaldi-trunk/egs/swbd/s5/mfcc/raw_mfcc_train.1.ark:54
sw02001-A_002736-002893 /home/dpovey/kaldi-trunk/egs/swbd/s5/mfcc/raw_mfcc_train.1.ark:62
```

这个文件指向已经提取好的特征——在这个例子中所使用的是MFCC。 `feats.scp` 文件的格式是：

```
<utterance-id> <extended-filename-of-features>
```

每一个特征文件保存的都是Kaldi格式的矩阵。在这个例子中，矩阵的维度是13（译者注：即列数；行数则 和你的文件长度有关，标准情况下帧长20ms，帧移10ms，所以一行特征数据对应10ms的音频数据）。第一行的“extended filename”， 即 `/home/dpovey/kaldi-trunk/egs/swbd/s5/mfcc/raw_mfcc_train.1.ark:24` ，意思是，打开存档（archive）文件 `/home/dpovey/kaldi-trunk/egs/swbd/s5/mfcc/raw_mfcc_train.1.ark` ， fseek()定位到24（字节），然后开始读数据。

`feats.scp` 文件由如下命令创建：

```
steps/make_mfcc.sh --nj 20 --cmd "$train_cmd" data/train exp/make_mfcc/train $mfccdir
```

该句被底层的 `run.sh` 脚本调用。命令中一些shell变量的定义，请查阅对应run.sh。**$mfccdir** 是.ark文件将被写入的目录，由用户自定义。

data/train下最后一个未讲到的文件是 `cmvn.scp` 。该文件包含了倒谱均值和方差归一化的统计量，以说话人编号为索引。 每个统计量几何都是一个矩阵，在本例中是2乘以14维。在我们的例子中，有：

```
s5# head -3 data/train/cmvn.scp
2001-A /home/dpovey/kaldi-trunk/egs/swbd/s5/mfcc/cmvn_train.ark:7
2001-B /home/dpovey/kaldi-trunk/egs/swbd/s5/mfcc/cmvn_train.ark:253
2005-A /home/dpovey/kaldi-trunk/egs/swbd/s5/mfcc/cmvn_train.ark:499
```

与feats.scp不同，这个scp文件是以说话人编号为索引，而不是发音编号。该文件由如下的命令创建：

```
steps/compute_cmvn_stats.sh data/train exp/make_mfcc/train $mfccdir
```

(这个例子来自 `egs/swbd/s5/run.sh`).

因为数据准备阶段的错误会影响后续脚本的运行，所以我们有一个脚本数据目录的文件格式是否 正确。运行：

```
utils/validate_data_dir.sh data/train
```

你可能会发现下面这个命令也很有用：

```
utils/fix_data_dir.sh data/train
```

（当然可对任何数据目录使用该命令，而不只是 `data/train` ）。该脚本会修复排序错误，并会移除那些被指明需要特征数据或标注，但是却找不到被需要的数据的那些发音（utterances）。

数据准备-- “lang”目录

现在我们关注一下数据准备的“lang”这个目录。

```
s5# ls data/lang
L.fst  L_disambig.fst  oov.int    oov.txt  phones  phones.txt  topo  words.txt
```

除data/lang，可能还有其他目录拥有相似的文件格式：例如有个目录被命名为“data/lang_test”，其中包含和data/lang完全一样的信息，但是要多一个G.fst文件。该文件是一个FST形式的语言模型：

```
s5# ls data/lang_test
G.fst  L.fst  L_disambig.fst  oov.int  oov.txt    phones    phones.txt  topo  words.txt
```

注意，lang_test/由拷贝lang/目录而来，并加入了G.fst。每个这样的目录都似乎只包含为数不多的几个文件。但事实上不止如此，因为其中phones是一个目录而不是文件：

```
s5# ls data/lang/phones
context_indep.csl  disambig.txt         nonsilence.txt        roots.txt    silence.txt
context_indep.int  extra_questions.int  optional_silence.csl  sets.int     word_boundary.
context_indep.txt  extra_questions.txt  optional_silence.int  sets.txt     word_boundary.
disambig.csl       nonsilence.csl       optional_silence.txt  silence.csl
```

phones目录下有许多关于音素集的信息。同一类信息可能有三种不同的格式，分别以.csl、.int和.txt结尾。幸运的是，作为一个Kaldi用户，你没有必要去一一手动创建所有这些文件，因为我们有一个脚本 `utils/prepare_lang.sh` 能够根据更简单的输入为你创建所有这些文件。在讲述该脚本和所谓更简单的输入之前，有必要先解释一下 `lang` 目录下到底有些什么内容。之后我们将解释如何轻松创建该目录。如果用户不需要理解Kaldi是如何工作的，而是秉着快速建立识别系统的目的，那么可以跳过下面的 `“lang”目录下的内` 容这一节。

“lang”目录下的内容

首先是有文件**phones.txt**和**words.txt**。这些都是符号表（symbol-table）文件，符合OpenFst的格式定义。其中每一行首先是一个文本项，接着是一个数字项：

```
s5# head -3 data/lang/phones.txt
<eps> 0
SIL 1
SIL_B 2
s5# head -3 data/lang/words.txt
<eps> 0
!SIL 1
-'S 2
```

在Kaldi中，这些文件被用于在这些音素符号的文本形式和数字形式之间进行转换。 大多数情况下，只有脚本utils/int2sym.pl、utils/sym2int.pl和OpenFst中的程序fstcompile和fstprint会读取这些文件。

文件**L.fst**是FST形式的发音字典（L，见 "Speech Recognition with Weighted Finite-State Transducers" by Mohri, Pereira and Riley, in Springer Handbook on SpeechProcessing and Speech Communication, 2008），其中，输入是音素，输出是词。文件 `L_disambig.fst` 也是发音字典，但是还包含了为消歧而引入的符号，诸如#1、#2 之类，以及为自环（self-loop）而引入的 #0 。#0 能让消岐符号“通过”（pass through）整个语法（译者注：n元语法，即我们的语言模型。另外前面这句话我是在不知道该怎么翻译）。更多解释见 \ref graph_disambig 。但是不管 明白与否，你其实不用自己手动去引入这些符号。

文件 `data/lang/oov.txt` 仅仅只有一行：

```
s5# cat data/lang/oov.txt
<UNK>
```

在训练过程中，所有词汇表以外的词都会被映射为这个词（译者注：UNK即unknown）。“”本身并没有特殊的地方，也不一定非要用这个词。重要的是需要保证这个词 的发音只包含一个被指定为“垃圾音素”（garbage phone）的音素。该音素会与各种 口语噪声对齐。在我们的这个特别设置中，该音素被称为 `<SPN>`，就是“spoken noise”的缩写：

```
s5# grep -w UNK data/local/dict/lexicon.txt
<UNK> SPN
```

文件 `oov.int` 则是SPN的数字形式（从 `words.txt` 中提取的），在本设置中是221。你可能已经注意到了，在Resource Management设置中，oov.txt里有一个 静音词，在RM设置中被称为“!SIL”。在这种情况下，我们从词汇表中任意选一个词（放入oov.txt）—— 因为训练集中没有oov词，所以选哪个都不起作用。

文件data/lang/topo则含有如下数据：

```
s5# cat data/lang/topo
<Topology>
<TopologyEntry>
<ForPhones>
21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50
</ForPhones>
<State> 0 <PdfClass> 0 <Transition> 0 0.75 <Transition> 1 0.25 </State>
<State> 1 <PdfClass> 1 <Transition> 1 0.75 <Transition> 2 0.25 </State>
<State> 2 <PdfClass> 2 <Transition> 2 0.75 <Transition> 3 0.25 </State>
<State> 3 </State>
</TopologyEntry>
<TopologyEntry>
<ForPhones>
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
</ForPhones>
<State> 0 <PdfClass> 0 <Transition> 0 0.25 <Transition> 1 0.25 <Transition> 2 0.25 <Trans
<State> 1 <PdfClass> 1 <Transition> 1 0.25 <Transition> 2 0.25 <Transition> 3 0.25 <Trans
<State> 2 <PdfClass> 2 <Transition> 1 0.25 <Transition> 2 0.25 <Transition> 3 0.25 <Trans
<State> 3 <PdfClass> 3 <Transition> 1 0.25 <Transition> 2 0.25 <Transition> 3 0.25 <Trans
<State> 4 <PdfClass> 4 <Transition> 4 0.75 <Transition> 5 0.25 </State>
<State> 5 </State>
</TopologyEntry>
</Topology>
```

这个文件指明了我们所用HMM模型的拓扑结构。在这个例子中，一个“真正”的音素内含3个发射状态，呈标准 的三状态从左到右拓扑结构——即“Bakis”模型。发射状态即能“发射”特征矢量的状态，与之对应的就是那些 “假”的仅用于连接其他状态的非发射状态。音素1到20是各种静音和噪声。之所以会有这么多， 是因为低词中的不同位置相同音素进行了区分。这种情况下，实际上这里的静音和噪声音素大多数都用不上。 不考虑在词中的位置的话，应该只有5个代表静音和噪声的音素。所谓静音音素（silence phones）有更复杂的 拓扑结构。每个静音音素都有一个起始发射状态和一个结束发射状态，中间还有另外三个发射状态。 你不用手动创建 `data/lang/topo` 。

`data/lang/phones/` 下有一系列的文件，指明了音素集合的各种信息。这些文件大多数有三个不同版本：一个 “.txt”形式，如：

```
s5# head -3 data/lang/phones/context_indep.txt
SIL
SIL_B
SIL_E
```

一个“.int”形式，如:

```
s5# head -3 data/lang/phones/context_indep.int
1
2
3
```

以及一个“.csl”形式，内含一个冒号分割的列表:

```
s5# cat data/lang/phones/context_indep.csl
1:2:3:4:5:6:7:8:9:10:11:12:13:14:15:16:17:18:19:20
```

三种形式的文件包含的是相同的信息，所以我们只关注人们更易阅读的“.txt”形式。文件 `context_indep.txt` 包含一个音素列表，用于建立文本无关的模型。也就是说， 对这些音素。 我们不会建立需要参考左右音素上下文的决策树。实际上，我们建立的是更小的决策树，只参考中心音素和HMM状态。这依赖于 `roots.txt` ， 下面将会介绍到。关于决策树的更深入讨论见 \ref tree_externals 。

文件 `context_indep.txt` 包含所有音素，包括那些所谓“假”音素：例如静音 `SIL` ，口语噪声 `SPN` ， 非口语噪声 `NSN` 和笑声 `LAU` ：

```
# cat data/lang/phones/context_indep.txt
SIL
SIL_B
SIL_E
SIL_I
SIL_S
SPN
SPN_B
SPN_E
SPN_I
SPN_S
NSN
NSN_B
NSN_E
NSN_I
NSN_S
LAU
LAU_B
LAU_E
LAU_I
LAU_S
```

因为考虑了在词中的位置，这些音素都有许多变体。不是所有的变体都会被实际使用。这里， `SIL` 代表静音词， 会被插入到发音字典中（是一个词而不是一个词的一部分，可选的）； `SIL_B` 则代表一个静音音素，应该出现 在一个词的开端（这种情况应该永不出

现）；`SIL_I` 代表词内静音（也很少存在）；`SIL_E` 代表 词末静音（不应该存在）；
而 `SIL_S` 则表示一种被视为“单独词”（singleton word）的静音，意指这个音素 只对应一个词——当你的发音字典中有“静音词”且标注中有明确的静音段时会有用。

`silence.txt` 和 `nonsilence.txt` 分别包含静音音素列表和非静音音素列表。 这两个集合是互斥的，且如果合并在一起，应该是音素的总集。在本例
中， `silence.txt` 与 `context_indep.txt` 的内容完全一致。 我们说“非静音”音素，是指我们将要估计各种线性变换的音素。所谓线性变换是指全局变换，如LDA 和MLLT，以及说话人自适应变换，如fMLLR。 根据之前的实验，我们相信，加入静音对这些变换没有 影响。我们的经验是，把噪声和发生噪声都列为“静音”音素，而其他传统的音素则是“非静音”音素。 在Kaldi中我们还没有通过实验找到一个最佳的方法来这样做。

```
s5# head -3 data/lang/phones/silence.txt
SIL
SIL_B
SIL_E
s5# head -3 data/lang/phones/nonsilence.txt
IY_B
IY_E
IY_I
```

`disambig.txt` 包含一个“消岐符号”列表“（见 \ref graph_disambig）：

```
s5# head -3 data/lang/phones/disambig.txt
#0
#1
#2
```

这些符号会出现在 `phones.txt` 中，被当做音素使用。

`optional_silence.txt` 只含有一个音素。该音素可在需要的时候出现在词之间：

```
s5# cat data/lang/phones/optional_silence.txt
SIL
```

你可以自行选择是否让该音素出现在词之间，方法就是在发音字典的FST中，可选地让该音素出现在每个词的词尾（以及每段发音的前面）。该因素必须在 `phones/` 中 指明而不是仅仅出现在 `L.fst` 中。这个原因比较复杂，这里就不讲了。

文件 `sets.txt` 包含一系列的音素集，在聚类音素时被分组（被当做同一个音素），以便建立文本相关问题集（在Kaldi中，建立决策树时使用自动生成的问题集，而不是具有语言语义 的问题集）。 在本设置中， `sets.txt` 将每个音素的所有与在词中位置相关的变体 组合为一行：

```
s5# head -3 data/lang/phones/sets.txt
SIL SIL_B SIL_E SIL_I SIL_S
SPN SPN_B SPN_E SPN_I SPN_S
NSN NSN_B NSN_E NSN_I NSN_S
```

文件 `extra_questions.txt` 包含那些自动产生的问题集之外的一些问题：

```
s5# cat data/lang/phones/extra_questions.txt
IY_B B_B D_B F_B G_B K_B SH_B L_B M_B N_B OW_B AA_B TH_B P_B OY_B R_B UH_B AE_B S_B T_B A
IY_E B_E D_E F_E G_E K_E SH_E L_E M_E N_E OW_E AA_E TH_E P_E OY_E R_E UH_E AE_E S_E T_E A
IY_I B_I D_I F_I G_I K_I SH_I L_I M_I N_I OW_I AA_I TH_I P_I OY_I R_I UH_I AE_I S_I T_I A
IY_S B_S D_S F_S G_S K_S SH_S L_S M_S N_S OW_S AA_S TH_S P_S OY_S R_S UH_S AE_S S_S T_S A
SIL SPN NSN LAU
SIL_B SPN_B NSN_B LAU_B
SIL_E SPN_E NSN_E LAU_E
SIL_I SPN_I NSN_I LAU_I
SIL_S SPN_S NSN_S LAU_S
```

你可以看到，所谓一个问题就是一组音素。 前4个问题是关于普通音素的词位信息，后面五个则是关于”静音音素“的。 ”静音“音素也可能没有像 `_B` 这样的后缀，比如 `SIL` 。这些可被作为发音字典中可选的静音词的表示，即不会出现在某个词中，而是 单独成词。 在具有语调和语气的设置中， `extra_questions.txt` 可以包含与之相关的问题集。

`word_boundary.txt` 解释了这些音素与词位的关联情况：

```
s5# head  data/lang/phones/word_boundary.txt
SIL nonword
SIL_B begin
SIL_E end
SIL_I internal
SIL_S singleton
SPN nonword
SPN_B begin
```

这和音素中的后缀是相同的信息，但我们并不想给音素的文本形式附加这样的强制限制——记住一件事， Kaldi的可执行程序从不使用音素的文本形式，而是整数形式。）所以我们使用文件 `word_boundary.txt` 来指明各音素与词位间的对应关系。建立这种对应关系的原因是因为我们需要这些信息从音素网格中恢复词的边界（例如，lattice-align-words 需要读取 `word_boundary.txt` 的整数版本 `word_boundary.int` ）。找出词的边界是有 用的，其中之一是用作NIST的sclite评分，该工具需要词的时间标记。还有其他的后续处理需要这些信息。

`roots.txt` 包含如何建立音素上下文决策树的信息：

```
head data/lang/phones/roots.txt
shared split SIL SIL_B SIL_E SIL_I SIL_S
shared split SPN SPN_B SPN_E SPN_I SPN_S
shared split NSN NSN_B NSN_E NSN_I NSN_S
shared split LAU LAU_B LAU_E LAU_I LAU_S
...
shared split B_B B_E B_I B_S
```

暂时你可以忽略”shared“和”split“——这些与我们建立决策树时的具体选项有关（更多信息见\ref tree_externals）。 像 `SIL SIL_B SIL_E SIL_I SIL_S` 这样，几个音素出现在同一行的意义是， 在决策树中它们都有同一个”共享根“（shared root）， 因此状态可在这些音素间共享。对于带语气语调的系统，通常所有与语气和语调相关的音素变体都会出现在同一行。此外，一个HMM中的3个状态（对静音来说有5个状态）共享一个根， 且决策树的建立过程需要知道状态（的共享情况）。 HMM状态间共享决策树根节点，这就是roots文件中”shared“代表的意思。

建立"lang"目录

`data/lang/` 目录下有很多不同文件，所以我们提供了一个脚本为你创建这个目录，你只需要提供一些相对简单的输入信息：

```
utils/prepare_lang.sh data/local/dict "<UNK>" data/local/lang data/lang
```

这里，输入目录是 `data/local/dict/` ， `<UNK>` 需在字典中，是标注中 所有OOV词的映射词（映射情况会写入 `data/lang/oov.txt` 中）。 `data/local/lang/` 只是脚本使用的一个临时目录， `data/lang/` 才是输出文件将会写入的地方。

作为数据准备者，你需要做的事就是创建 `data/local/dict/` 这个目录。该 目录包含以下信息：

```
s5# ls data/local/dict
extra_questions.txt  lexicon.txt nonsilence_phones.txt  optional_silence.txt  silence_pho
```

（实际上还有一些文件我们没有列出来，但那都是在创建目录时所遗留下的临时文件，可以忽略）。下面的这些命令可以让你知道这些文件中大概都有些什么：

```
s5# head -3 data/local/dict/nonsilence_phones.txt
IY
B
D
s5# cat data/local/dict/silence_phones.txt
SIL
SPN
NSN
LAU
s5# cat data/local/dict/extra_questions.txt
s5# head -5 data/local/dict/lexicon.txt
!SIL SIL
-'S S
-'S Z
-'T K UH D EN T
-1K W AH N K EY
```

正如你看到的，本设置（Switchboard）中，这个目录下的内容都非常简单。 我们只是分别列出了“真正”的音素和“静音”音素，一个叫 `extra_questions.txt` 的空文件，以及一个有如下格式的 `lexicon.txt` ：

```
<word> <phone1> <phone2> ...
```

注意： `lexicon.txt` 中，如果一个词有不同发音，则会在不同行中出现多次。如果你想使用发音概率，你需要建立 `lexiconp.txt` 而不是 `lexicon.txt` 。

`lexiconp.txt` 中第二域就是概率值。 注意，一个通常的作法是，对发音概率进行归一化，使最大的那个概率值为1，而不是使同一个 词的所有发音概率加起来等于1。 这样可能会得到更好的结果。 如果想在顶层脚本中找一个 与发音概率相关的脚本，请在 `egs/wsj/s5/run.sh` 目录下搜索 `pp` 。

需要注意的是，在这些输入中，没有词位信息，即没有像 `_B` 和 `_E` 这样的后缀。 这是因为脚本 `prepare_lang.sh` 添加这些后缀。

从空的 `extra_questions.txt` 文件中你会发现，可能还有些潜在的功能我们没有利用。 这其中就包括重音和语调标记。对具有不同重音和语调的同一音素，你可能会想用不同的标记去表示。 为展示如何这样做，我们看看在另外一个设置 `egs/wsj/s5/` 下的这些文件。结果如下：

```
s5# cat data/local/dict/silence_phones.txt
SIL
SPN
NSN
s5# head data/local/dict/nonsilence_phones.txt
S
UW UW0 UW1 UW2
T
N
K
Y
Z
AO AO0 AO1 AO2
AY AY0 AY1 AY2
SH
s5# head -6 data/local/dict/lexicon.txt
!SIL SIL
<SPOKEN_NOISE> SPN
<UNK> SPN
<NOISE> NSN
!EXCLAMATION-POINT  EH2 K S K L AH0 M EY1 SH AH0 N P OY2 N T
"CLOSE-QUOTE  K L OW1 Z K W OW1 T
s5# cat data/local/dict/extra_questions.txt
SIL SPN NSN
S UW T N K Y Z AO AY SH W NG EY B CH OY JH D ZH G UH F V ER AA IH M DH L AH P OW AW HH AE
UW1 AO1 AY1 EY1 OY1 UH1 ER1 AA1 IH1 AH1 OW1 AW1 AE1 IY1 EH1
UW0 AO0 AY0 EY0 OY0 UH0 ER0 AA0 IH0 AH0 OW0 AW0 AE0 IY0 EH0
UW2 AO2 AY2 EY2 OY2 UH2 ER2 AA2 IH2 AH2 OW2 AW2 AE2 IY2 EH2
s5#
```

你可能已经注意到了， `nonsilence_phones.txt` 中的某些行，一行中 有多个音素。这些是同一元音的与重音相关的不同表示。 注意，在CMU版的字典中， 每个音素有4种表示：例如， `UW UW0 UW1 UW2` 。 基于某些原因,其中 一种表示没有数字后缀。行中音素的顺序没有关系。通常，我们建议用于将每个“真实音素” 的不同形式都组织在单独的一行中。 我们使用CMU字典中的重音标记。文件 `extra_questions.txt` 中只有一个问题 包含所有的“静音”音素（实际上这是不必要的，只是脚本 `prepare_lang.sh` 会添加这么一个问题），以及一个涉及不同重音标记的问题。 这些问题对利用重音标记信息来说是必要的，因为在 `nonsilence_phones.txt` 中每个音素的不同重音表示都在同一行，这确保了他们在 `data/lang/phones/roots.txt` 和 `data/lang/phones/sets.txt` 也属同一行，这又反过来确保了它们共享同一个（决策）树 根，并且不会有决策问题弄混它们。因此，我们需要提供一个特别的问题，能为决策树的建立过程 提供一种区分音素的方法。 注意：我们在 `sets.txt` 和 `roots.txt` 中 将音素分组放在一起的原因是，这些同一音素的不同重音变体可能缺乏足够的数据去稳健地估计 一个单独的决策树，或者是产生问题集时需要的聚类信息。 像这样把它们组合在一起，我们 可以确保当数据不足以对它们分别估计决策树时，这些变体能在决策树的建立过程中“聚集在一起”（stay together）。

写到这里我们需要提一点，脚本 `utils/prepare_lang.sh` 支持很多选项。下面是该脚本 的用法，可让你们了解这些选项都有哪些：

```
usage: utils/prepare_lang.sh <dict-src-dir> <oov-dict-entry> <tmp-dir> <lang-dir>
e.g.: utils/prepare_lang.sh data/local/dict <SPOKEN_NOISE> data/local/lang data/lang
options:
     --num-sil-states <number of states>             # default: 5, #states in silence mod
     --num-nonsil-states <number of states>          # default: 3, #states in non-silence
     --position-dependent-phones (true|false)        # default: true; if true, use _B, _E
                                                     # markers on phones to indicate word
     --share-silence-phones (true|false)             # default: false; if true, share pdf
                                                     # all non-silence phones.
     --sil-prob <probability of silence>             # default: 0.5 [must have 0 < silpro
```

一个可能的重要选项是 `--share-silence-phones` 。 该选项默认是false。 如果该选项被设为true， 所有静音音素——如静音、发生噪声、 噪声和笑声——的概率密度函数（PDF，高斯混合模型）都会共享，只有模型中的转移概率不同。 现在还不清楚为什么这样做有用，但我们发现者对IARPA的BABEL项目中的广东话数据集非常有效。 该数据集非常乱，其中有很长的未标注的部分，我们试着将其与一个特别标记的音素对齐。 我们怀疑训练数据可能没能成功正确对齐，而且基于某些不明原因，将上述选项设置为true则 改变了结果。

另外一个可能的重要选项是 `--sil-prob` 。 通常，对这些选项我们所作的实验都不同，所以对具体如何设置也不能给出非常详细的建议。

创建语言模型或者语法文件

前面的关于如何创建 `lang/` 目录的教程没有涉及如何产生 `G.fst` 文件。该文件 是语言模型——或者称为语法——的有限状态转换器格式的表示，我们解码时需要它。 实际上，在一些设置 中，为做不同的测试，我们可能会有许多“lang”目录。这些目录中有不同的语言模型和字典。以 华尔街日报（WSJ）的设置为例：

```
s5# echo data/lang*
data/lang data/lang_test_bd_fg data/lang_test_bd_tg data/lang_test_bd_tgpr data/lang_test
 data/lang_test_bg_5k data/lang_test_tg data/lang_test_tg_5k data/lang_test_tgpr data/lan
```

根据我们使用的语言模型的不同——是统计语言模型还是别的种类的语法形式——生成 `G.fst` 的 步骤会不同。 在RM设置中，使用的是二元语法，只允许某些词对。我们将总概率值1分配给所有向外 的弧，以确保每个语法状态的概率和为1。在 `local/rm_data_prep.sh` 中有这样一句代码：

```
local/make_rm_lm.pl $RMROOT/rm1_audio1/rm1/doc/wp_gram.txt  > $tmpdir/G.txt || exit 1;
```

脚本 `local/make_rm_lm.pl` 会建立一个FST格式的语法文件（文本格式，不是二进制格式）。该文件包含如下形式的行：

```
s5# head data/local/tmp/G.txt
0    1    ADD    ADD    5.19849703126583
0    2    AJAX+S    AJAX+S    5.19849703126583
0    3    APALACHICOLA+S    APALACHICOLA+S    5.19849703126583
```

到 www.openfst.org 上查阅更多关于OpenFst的信息（他们有一个很详细的教程）。 脚本 `local/rm_prepare_grammar.sh` 会将文本格式的 语法文件转换为二进制文件 `G.fst` 。所用命令如下：

```
fstcompile --isymbols=data/lang/words.txt --osymbols=data/lang/words.txt --keep_isymbols=
    --keep_osymbols=false $tmpdir/G.txt > data/lang/G.fst
```

如果你要建立自己的语法问，你也应做类似的事。 注意：这种过程只适用于一类语法：用上述方法你不能创建上下文无法的语法，因为这类语法不 能被表示为OpenFst格式。 在WFST框架下还是有办法这么做（见Mike Riley最近关于push down transducers 的研究工作），但是在Kaldi中我们还没实现这些功能。

在WSJ设置中，我们使用了一个统计语言模型。 脚本 `local/wsj_format_data.sh` 将WSJ数据库提供的ARPA格式的语言模型文件转换为OpenFst格式的。 脚本中关键的命令如下：

```
  gunzip -c $lmdir/lm_${lm_suffix}.arpa.gz | \
   utils/find_arpa_oovs.pl $test/words.txt  > $tmpdir/oovs_${lm_suffix}.txt
...
  gunzip -c $lmdir/lm_${lm_suffix}.arpa.gz | \
    grep -v '<s> <s>' | \
    grep -v '</s> <s>' | \
    grep -v '</s> </s>' | \
    arpa2fst - | fstprint | \
    utils/remove_oovs.pl $tmpdir/oovs_${lm_suffix}.txt | \
    utils/eps2disambig.pl | utils/s2eps.pl | fstcompile --isymbols=$test/words.txt \
      --osymbols=$test/words.txt  --keep_isymbols=false --keep_osymbols=false | \
     fstrmepsilon > $test/G.fst
```

这里，变量 `$test` 的值形如 `data/lang_test_tg` 。最重要的一条命令是 `arpa2fst` ， 这是一个Kaldi程序。该程序将ARPA格式的语言模型转换为一个 加权有限状态转换器（实际上是一个接收器）。

`grep` 命令移除语言模型中“不可用”的N元语法。 程序 `remove_oovs.pl` 移除包含集外词的N元语法（如果不移除会引起 `fstcompile` 崩溃）。 `eps2disambig.pl` 将回退弧上的 `<eps>` （ϵ）符号转换为一个特殊的符号 `#0` ，以 保证语法文件是确定的（determinizable），见 \ref graph_disambig 。 如果你知道“回退弧”是什么， 你可以参考关于回退N元语法的文献，例如

Goodman的 *A bit of progress in language modeling* ，以及我们前面引用的Mohri的论文。命令 s2eps.pl 将句首和句末 符号 <s> 和 </s> 转换为epsilon（ <eps> ）， 意即“没有符号”。 fstcompile 是一个OpenFst命令，可将文本形式的FST转换为二进制形式的。 fstrmepsilon 也是一个OpenFst命令，可将FST中由 <s> 和 </s> 替换而来的 少量的 <eps> 符号移除掉。

编译过程（Kaldi如何编译）

Kaldi的编码风格

Kaldi项目的历史

Kaldi的矩阵运算库

外部矩阵库

CUDA矩阵库

Kaldi的I/O机制

翻译：izy

时间：2015年7月

本页提供了 Kaldi 输入输出机制的概述。

这部分文档主要面向I/O的代码层机制分析，从命令行角度分析的说明文档，参看 Kaldi I/O from a command-line perspective

The input/output style of Kaldi classes

Kaldi 定义的类有统一的I/O接口。标准接口如下所示：

```
class SomeKaldiClass {
 public:
   void Read(std::istream &is, bool binary);
   void Write(std::ostream &os, bool binary) const;
};
```

注意函数返回 void；而错误会通过异常处理来显示（见 Kaldi logging and error-reporting）. 布尔变量“binary”标识对象是以二进制数据还是文本数据被写（或读）。调用代码必须知道被读写的对象是二进制还是文本形式（关于在读的过程中它是如何知道，见 How Kaldi objects are stored in files）.注意“binary”变量的值没有必要和文件打开的方式（在 Windows下），即二进制还是文本模式，保持一致。更多注释请参照 How the binary/text mode relates to the file open mode。

`Read` 和 `Write` 函数可以有额外的可选参数。一个常见的 `Read` 函数例子有如下形式：

```
class SomeKaldiClass {
 public:
  void Read(std::istream &is, bool binary, bool add = false);
};
```

如果 `add==true`，`Read` 函数会添加所有在磁盘上的内容（e.g.statistics）到当前类中，前提该类不为空。

Input/output mechanisms for fundamental types and STL types

参照 Low-level I/O functions 中相关的函数列表。我们提供这些函数，使读写基本类型变得更加容易；他们大多被Kaldi类中的 `Read` 和 `Write` 函数调用。Kaldi 的类可以任意使用这些函数，只要它的 `Read` 函数可以正确地读取 `Write` 函数所写入的数据。

这一类中最重要的函数是 `ReadBasicType()` 和 `WriteBasicType()` ;这些是类模板，涵盖了bool, float, double和integer 类型。它们在 `Read` 和 `Write` 函数中应用的例子如下：

```
// we suppose that class_member_ is of type int32.
void SomeKaldiClass::Read(std::istream &is, bool binary) {
  ReadBasicType(binary, &class_member_);
}
void SomeKaldiClass::Write(std::ostream &is, bool binary) const {
  WriteBasicType(binary, class_member_);
}
```

我们假定 `class_member_` 是 int32类型，一种已知大小的类型。这些函数中用 int类型并不安全。在二进制模式中，函数写字符时会编码类型的大小和符号，如果不匹配就会读取失败。我们本可以尝试对它们进行自动转换，但并没有这么做；目前来说，在 I/O中你必须用已知大小的整型类型（一般建议采用 int32）。另一方面，浮点类型会被自动转换。这是为了调试方便，所以你可以以 `-DKALDI_DOUBLE_PRECISION` 方式编译，同时读取没有设定该选项而写成的二进制文件。I/O惯例中没有 byte swapping；如果这对你是个问题，请换用文本格式。

还有 `WriteIntegerVector()` 和 `ReadIntegerVector()` 模板函数。它们和 `ReadBasicType()` ，`WriteBasicType()` 是同样的代码风格，不过是针对 `std::vector<I>` ，其中 I是某种整型类型（当然，它的大小在编译时应该是已知的，e.g. int32）

其他一些重要的底层I/O函数有：

```
void ReadToken(std::istream &is, bool binary, std::string *token);
void WriteToken(std::ostream &os, bool binary, const std::string & token);
```

一个记号(token)必须是不带空格的非空字符串，实际中一般是类似于 XML形式的字符串，如 `<SomeKaldiClass>` ，`<SomeClassMemberName>` 或者 `</SomeKaldiClass>` 。这些函数的功能正如它们被定义的那样。方便起见，我们提供 `ExpectToken()` ，类似 `ReadToken()` ，只是你要指定想要的字符串（如果没有指定，就会产生异常）。典型的调用代码如下：

```
// in writing code:
WriteToken(os, binary, "<MyClassName>");
// in reading code:
ExpectToken(is, binary, "<MyClassName>");
// or, if a class has multiple forms:
std::string token;
ReadToken(is, binary, &token);
if(token == "<OptionA>") { ... }
else if(token == "<OptionB>") { ... }
...
```

还有 `WritePretty()` 和 `ExpectPretty()` 函数，一般较少使用。除了是以文本方式读写，它们的作用和相应的 Token函数相似，而且可以接受任意字符串（i.e.允许空格）. `ReadPretty` 函数可以接受与期望输入仅仅有空格差异的输入。Kaldi 类中的 `Read` 函数不会检查文件是否结束，而默认读到 `Write` 函数写完的地方（在文本模式下，有几个空格没被读取也没有关系）。这是多个 Kaldi对象可以放入同一文件， 也是允许 archive概念（见 The Kaldi archive format）的原因。

How Kaldi objects are stored in files

如上所示，Kaldi 代码读取数据时需要知道读模式是文本还是二进制，但是我们不希望由用户来跟踪一个文件是文本还是二进制。所以，包含 Kaldi对象的文件需要声明它们存储的是二进制还是文本数据。二进制文件以字符串“\0B”开始，同时由于文本文件不能包含“\0”，也就不需要头。如果你用标准的 C++机制来打开文件（一般不会这么做，见 How to open files in Kaldi）,你需要在采取其他操作前先处理这个文件头。你可以用 `InitKaldiOutputStream()` （它同时设定了数据流的精度）和 `InitKaldiInputStream()`

How to open files in Kaldi

假定需要从/向磁盘载入/储存一个Kaldi对象，同时对象是类似声学模型那样的（而不像声学特征那样有很多，针对这个请参考 The Table concept）；你一般会用到 `Input` 和 `Output` 类。一个例子：

```
{ // input.
  bool binary_in;
  Input ki(some_rxfilename, &binary_in);
  my_object.Read(ki.Stream(), binary_in);
  // you can have more than one object in a file:
  my_other_object.Read(ki.Stream(), binary_in);
}
// output.  note, "binary" is probably a command-line option.
{
  Output ko(some_wxfilename, binary);
  my_object.Write(ko.Stream(), binary);
```

花括号的作用是使 `Input` 和 `Output` 的对象在结束时就离开作用域，因此文件会被立刻关闭。这也许没什么用处（为什么不用标准的C++流呢？）。但是这样我们可以支持各种扩展的文件名，同时处理错误也更容易些（ `Input` 和 `Output` 类在遇到错误时会打印提示性信息，并抛出异常）。注意文件名包括“rxfilename”和“wxfilename”。我们经常使用这种类型的文件名，它们意在提示编程人员它们是扩展的文件名，这会在下一部分进行描述。

`Input` 和 `Output` 类的接口比上面例子中的要稍微丰富一些。你可以调用 `Open()` 来打开，调用 `Close()` 来关闭，而不仅仅是让他们离开作用域。这些函数会返回布尔类型的状态值，而不像构造函数和析构函数在遇到错误时抛出异常。 `Open()` 函数（包括构造函数）也可以在被调用时不处理 Kaldi的二进制文件头，如果你是想读写非Kaldi对象。估计你不会用到这一额外的功能。

参考 Classes for opening streams 中与 `Input` 和 `Output` 相关的类和函数，和rxfilenames, wxfilenames (下一部分)

Extended filenames:rxfilename and wxfilename

“rxfilename”和“wxfilename”不是类，他们是对变量名的描述符，他们表示：

- rxfilename 是一个可以被 Input类 当做扩展的文件名来读取的字符串
- wxfilename 是一个可以被 Output类 当做扩展的文件名来写入的字符串

rxfilename的类型如下：

- “-”或“” 表示标准输入
- “some command |” 表示一个输入管道命令，i.e.我们去掉管道符“|”，把剩下的字符串通过popen()传入shell
- “/some/filename:12345” 表示文件的偏置，i.e.我们打开文件并定位至12345
- “/some/filename”... 与以上不匹配的模式都会被当做普通的文件名（当然，一些明显的错误会被检测出来，在它们被打开之前）

你可以用 `ClassifyRxfilename()` 来获得 rxfilename的类型，不过这一般没有必要。

wxfilename的类型如下：

- “-”或“” 表示标准输入
- “| some command” 表示一个输出管道命令，i.e.我们去掉管道符“|”，把剩下的字符串通过popen()传入shell
- “/some/filename”... 与以上不匹配的模式都会被当做普通的文件名（当然，会检测并过滤掉明显的错误）

同样地， `ClassifyWxfilename()` 可以告诉你一个文件名的类型。

The Table concept

表(Table)是一个概念而不是一个实际的C++类。它是一个已知类型的对象的集合，并且用字符串(strings)索引。字符串必须是 tokens（不包含空格的非空字符串）。表的典型实例包括：

- 特征文件（表示为 `Matrix<float>` ）的集合，由语句id (utterance id)索引
- 转码文本（表示为 `std::vector<int32>` ）的集合，由语句id索引
- 约束的MLLR变换（表示为 `Matrix<float>` ）的集合，由说话人id索引

在 Types of data that we write as tables, 我们对这些类型的表有更详细的处理;在这里我们只是解释一般原则和内部机理。表可以存在磁盘（或实际上，在管道里）以两种可能的格式：脚本文件(script file)或存档文件(archive)（见下面的 The Kaldi script-file format 和 The Kaldi archive format）。表相关的类和类型，见 Table types and related functions。

Table可以通过三种方式访问： `TableWriter` , `SequentialTableReader` 和 `RandomAccessTableReader` （还有 `RandomAccessTableReaderMapped` 是一个特例，后面会讲到）。这些都是类模板；它们不是基于 table对象的模板，而是基于Holder类型（见下面， Holders as helpers to Table classes），可以告诉代码如何读写该类型的对象。为了打开 Table类型，你必须提供一个 wspecifier或 rspecifier的字符串（见下面，Specifying Table formats:wspecifiers and rspecifiers）来告诉代码表在磁盘上是如何存储的，同时提供各种其他指令。我们用一个示例代码来解释，这个代码读取特征，对其进行线性变换然后再写出：

```
std::string feature_rspecifier = "scp:/tmp/my_orig_features.scp",
   transform_rspecifier = "ark:/tmp/transforms.ark",
   feature_wspecifier = "ark,t:/tmp/new_features.ark";
// there are actually more convenient typedefs for the types below,
// e.g. BaseFloatMatrixWriter, SequentialBaseFloatMatrixReader, etc.
TableWriter<BaseFloatMatrixHolder> feature_writer(feature_wspecifier);
SequentialTableReader<BaseFloatMatrixHolder> feature_reader(feature_rspecifier);
RandomAccessTableReader<BaseFloatMatrixHolder> transform_reader(transform_rspecifier);
for(; !feature_reader.Done(); feature_reader.Next()) {
   std::string utt = feature_reader.Key();
   if(transform_reader.HasKey(utt)) {
      Matrix<BaseFloat> new_feats(feature_reader.Value());
      ApplyFmllrTransform(new_feats, transform_reader.Value(utt));
      feature_writer.Write(utt, new_feats);
   }
}
```

这个设定的优点是访问表的代码可以把表中数据看做 generic maps或 lists。数据类型和读过程中的参数（e.g.容错率）可以通过 rspecifiers和 wspecifiers中的选项进行控制，而不必由调用代码进行处理；上面的例子中，“,t”选项表示以文本形式写数据。

一个理想的表可能是一个字符串到对象的映射。然而，只要我们不在表上做随机访问，即使同一字符串有多个入口（i.e.在写或顺序访问时，它更像是a list of pairs），代码也不会出现问题。

关于 Table类型读写特定类型的类型定义，参考 Specific Table types。

The Kaldi script-file format

脚本文件(script file)（名字可能不太合适）是一个文本文件，每一行一般包括：

```
some_string_identifier /some/filename
```

另外一种有效的行可能是这样：

```
utt_id_01002 gunzip -c /usr/data/file_010001.wav.gz |
```

当读取 script file的一行时，Kaldi 会去除开头和结尾的空格，再以空格为分隔符进行拆分。第一部分成为表的 key（例如发声id，在上面例子里是“utt_id_01001”），而第二部分成为 xfilename（即 wxfilename或 rxfilename，在上面例子里是“gunzip -c /usr/data/file_010001.wav.gz |”）。 空行或空 xfilename是不允许的。 script file用于读或写或同时读写都是可以的，这取决于 xfilenames是否是有效的 rxfilenames，或 wxfilenames，或两者兼而有之。

假设一个 script file是读取有效的，包含一些 Kaldi类中的对象。通常可以读出其中的一行，用 `Input` 对象打开（见 How to open files in Kaldi）。如果是二进制的，文件流会包括二进制文件头“\0B”（即使是在文件的中间部分，如 archive）

The Kaldi archive format

Kaldi的存档文件(archive)格式很简单。再次明确 token定义为不含空格的非空字符串。archive格式描述如下：

```
token1 [something]token2 [something]token3 [something] ....
```

我们可以把这看成零个或多个重复（token;空格;调用Holder的 `Write` 函数的结果）。回想Holder是讲述代码如何读写数据的一个对象。

当写 Kaldi对象时，Holder写的 `[something]` 会包括二进制模式的文件头（如果是二进制模式），然后是调用该对象 `Write` 函数返回的结果。当写的非Kaldi对象是简单的（像int32, float或vector）,Holder类一般会确保文本模式下 `[something]` 是以换行符结尾的字符串。这样一来，archive是每行一个条目，很像 script file，比如：

```
    utt_id_1 5
    utt_id_2 7
    ...
```

是我们用于存储整数的文本存档格式。

archive 可以合并起来，仍是有效的（假定它们包含同样类型的对象）。这种格式被设计为管道友好的，i.e.你可以把 archive放入管道而读它的程序不用等到管道末尾就可以处理这些数据。为了有效的访问 archive，可以同时写 archive和 script file，script file存放的是地址的偏移。为此，请参阅下一节。

Specifying Table formats: wspecifiers and rspecifiers

Table类需要传递字符窜给构造函数或 `Open` 函数。这个字符串叫做 wspecifier如果它是传递给 `TableWriter` 类，或 rspecifier如果它是传递给 `RandomAccessTableReader` 或 `SequentialTableReader` 类。有效的 rspecifiers和 wspecifiers的例子包括：

```
std::string rspecifier1 = "scp:data/train.scp"; // script file.
std::string rspecifier2 = "ark:-"; // archive read from stdin.
// write to a gzipped text archive.
std::string wspecifier1 = "ark,t:| gzip -c > /some/dir/foo.ark.gz";
std::string wspecifier2 = "ark,scp:data/my.ark,data/my.ark";
```

通常，rspecifier或 wspecifier包括由逗号分隔的，无序的一个或两个字母的定义的选项和“ark”“scp”标识。后面跟着冒号和一个 rxfilename或 wxfilename。冒号前面选项的顺序并不重要。

Writing an archive and a script file simutaneously

wspecifiers的一个特例是：冒号前是“ark,scp”；冒号后是一个用于写 archive的 wxfilename，一个逗号，然后是一个 wxfilename（用于script file）。例如：

```
"ark,scp:/some/dir/foo.ark,/some/dir/foo.scp"
```

这会以像"utt_id /somedir/foo.ark:1234"这样的方式来写入archive和script file，指定了 archive的偏置，来达到更有效率的随机访问。接着你就可以随意处理script file，比如分解成多个片段，它仍像其他 script file一样正常工作。注意，虽然冒号前面的选项顺序一般不重要，这个例子中“ark”必须放在“scp”前面；这是为了避免冒号后面的两个 wxfilename产生混淆（archive始终在第一个）。指定 archive的 wxfilename必须是一个正常的文件名，否则写入的 script file不能被 Kaldi直接读取，但是代码中并没有限定这一点。

Valid options for wspecifiers

允许的wspecifier选项有：

- “b”(binary) 以二进制方式写（目前这是不必要，因为它是默认方式）
- “t”(text) 以文本方式写
- “f”(flush) 每次写操作后都刷新数据流
- “nf”(non-flush) 每次写操作后不刷新数据流（目前这是无意义的，但是调用代码可以更改这个默认值）
- “p” 指许可模式，影响“scp:”wspecifiers在scp文件丢失一部分条目时，“p”选项会导致它不写这些内容，同时不会报告错误。

用了很多选项的 wspecifiers的例子是：

```
    "ark,t,f:data/my.ark"
    "ark,scp,t,f:data/my.ark,|gzip -c > data/my.scp.gz"
```

Valid options for rspecifiers

读下面的选项时，请记住读 archives的代码不能在 archive中进行搜索，因为 archive可能是一个管道（经常是这样）。如果 `RandomAccessTableReader` 在读 archive，读代码可能要在内存中存储很多对象，以防后面再次被请求到，或代码需要搜寻到 archive的末尾，当它想找到一个 key而 archive中实际并不存在这样的 key时。下面的一些选项就是用来避免这种情况。

重要的rspecifier选项是：

- “o”(once) 是用户声明给 `RandomAccessTableReader` 每个 key只被请求一次。这样就不必把已经读过的对象储存在内存中以防万一它们会再被用到。
- “p”(pemissive) 指示代码忽略掉错误，只提供有效数据；无效数据会被视为不存在。在scp文件中，一个查询 `Haskey()` 强制加载相应的文件，而如果该文件已损坏，代码知道返回错误。在 archive中，这个选项避免了由 archive损坏或截断所引起的异常（它只是不会读那样的点）
- “s”(sorted) 指示代码 archive中的 key是按字符顺序排序的。
  对 `RandomAccessTableReader` 来说，这意味着当 `Haskey()` 查询某个不存在 archive中的 key时，它在遇到一个“更高”的 key时就会立刻返回错误；而不必读到文件末尾
- “cs”(called-sorted) 指示代码调用 `Haskey()` 和 `Value()` 时是按照字符顺序的。因此，如果其中一个函数被调用于某个字符串时，读代码就会忽视字符串序低的那些对象。这节省了内存。实际上，“cs”代表用户声明了程序可能在遍历的其他 archive是排好序的。

如果用户错误地提供了上面的选项，e.g.对实际上没有排序的 archive给定“s”选项， `RandomAccessTableReader` 代码会尽可能地检测出此错误并停止工作。

为了对称和方便，也提供下面的一些选项，但是目前并不起作用。

- “no”(not-once) 是“o”的对立选项（当前代码中，它没有任何作用）
- “np”(not-permissive) 是“p”的对立选项（当前代码中，它没有任何作用）
- “ns”(not-sorted) 是“s”的对立选项（当前代码中，它没有任何作用）
- “ncs”(not-called-sorted) 是“cs”的对立选项（当前代码中，它没有任何作用）
- “b”(binary) 什么都不做，仅为了写脚本方便
- “t”(text) 什么都不做，仅为了写脚本方便

使用了很多选项的rspecifiers的典型例子是：

```
    "ark:o,s,cs:-"
    "scp,p:data/my.scp"
```

Holders as helpers to Table classes

如之前所提到的，Table类i.e. `TableWriter`，`RandomAccessTableReader` 和 `SequentialTableReader`,是基于 Holder类的模板。Holder不是类或基类，而是一系列的类，它们的命名以 Holder结尾，e.g. `TokenHolder` 或 `KaldiObjectHolder`。（`KaldiObjectHolder` 是一个通用的 Holder,可以作为任意符合如 The input/output style of Kaldi classes 中 Kaldi I/O风格的类的模板）。我们写好了模板类 `GenericHolder`，不为实际使用，只为说明 Holder类必须满足的特性。

Holder类所包含的类是 typedef Holder::T（此处 Holder是实际 Holder类的类名）。可用的 Holder类型列表见 Holder types。

How the binary/text mode relates to the file open mode

本节仅与 Windows平台相关。通用的规则是，当写入时，文件模式要与 `Write` 函数中的“binary”变量值一致；当读二进制数据时，文件模式永远是二进制，但是读文本数据时，文件模式可以使二进制或文本（因此文本模式的读函数必须接受 Windows插入的额外“\r”字符）。这是因为直到打开一个文件，我们永远也不知道它的内容是二进制还是文本，所以当不确定的时候，我们以二进制方式打开。

Avoiding memory bloat when reading

当 Table代码随机访问大的 archive时，可能发生内存膨胀(memory bloat)。这可能发生在 `RandomAccessTableReader<SomeHolder>` 的一个对象读取 archive时。Table代码在编写时被优先保证正确性，所以当以随机访问方式读 archive时，除非你给定了额外的信息（也是下面要讨论的），它不会丢掉它已经读过的对象，以防后面还会被再次访问。一个明显的问题是：为什么 Table代码不直接跟踪对象在文件中的起点，然后用 fseek()定位到那个位置呢？我们还没实现这一点，原因如下：你能用 fseek()的前提是被读的 archive是一个实际的文件（i.e. 不是一个管道命令或标准输入）。如果 archive是一个磁盘上的实际文件，你在写它时很可能已经附含了 scp文件来标识偏置信息（用“ark,scp”前缀，见 Writing an archive and a script file simutaneously）,并提供了这个 scp文件给程序来帮助读 archive。这几乎和直接读 archive一样高效，因为代码读入 scp文件后可以避免重复打开不需要的文件或不必要地调用 fseek()。所以把 archive文件看做一个特列并在文件中附加偏置信息并不会解决这一问题。

当以随机访问模式读 archive时会产生两个独立的问题，如果你用“ark:”前缀并不提供额外选项，它们可能同时产生。

- 如果你请求一个不存在 archive中的 key时，读代码会被强制读到文档结束来确保它真的不存在。
- 每次代码读一个对象时，它都被强制保留在内存中以防后面还会被请求到。

关于第一个问题（要一直读到文件末尾），避免它的方式是确保 archive按照 key排序（按照通用的“C”字符串顺序排序，即“sort”代码所用的那样，如果指定“export LC_ALL=C”）。你可以在读 archives时用“s”选项来实现：例如，rspecifier“ark,s:-”指示代码读标准输入时将其看做archive并期望它是有序的。Table代码会检测你声明的是否是真，如果不是将会停止运行。当然，你应该设置你的脚本，保证 archives实际上是按 key排序好的（通常这在特征提取阶段就会被完成）。

关于第二个问题（强制在内存中保留已经读过的内容），有两个解决方案。

- 第一个解决方案，是一个不太稳定的方案，即提供“once”选项；例如，rspecifier“ark,o:-”从标准输入中读数据并认为每个对象你只会请求一次。为了确保这一点你必须知道在处理问题的程序是如何工作的，而且你应该知道提供给程序的其他 Table不包含重复的 keys（是的，Tables可以包含重复的 keys只要它是被顺序访问的）

  如果你提供“o”选项，Table可以再访问对象后释放它们。然而，这只在你的 archives完全同步对齐，且不存在空隙(gaps)或缺少元素时才能有效。例如，假设你执行以下命令：

  ```
  some-program ark:somedir/some.ark "ark,o:some command|"
  ```

  程序“some-program”首先会顺序遍历 archive“somedir/some.ark”，然后对于它遇到的每一个 key，随机访问第二个 archive。注意命令行中变量的顺序不是任意的：我们采用了这样的惯例，被顺序访问的 rspecifiers出现在被随机访问的 rspecifiers之前。

  假设两个 archives大部分都对齐了，但是有很多空隙（i.e.丢失的 keys，e.g.由于特征提取、数据对齐等产生的错误）每当第一个 archive中有空隙时，程序就需要缓存第二个 archive中的相关对象，因为它不知道它们在后面会不会被访问（它只能丢弃已经读过了的对象）。第二个 archive中有空隙时，问题会更严重，因为哪怕只有一个元素有空隙，当程序请求这 个key时，它就要一直读到第二个 archive的末尾来寻找，并且缓存这一过程中遇到的所有对象。

- 第二个解决方案比较鲁棒，即用“called-sorted”(cs)选项。这确保对象会被顺序地请求，同样地，这也需要知道程序是如何工作的，而且所有被顺序访问的 archives都是排好序的。“cs”选项一般和“s”选项一起用时效用最高。假设我们执行以下命令：

  ```
  some-program ark:somedir/some.ark "ark,s,cs:some command|"
  ```

  我们假设两个 archives都是排好序的，程序会顺序访问第一个 archive，随机访问第二个 archive。这样就对空隙鲁棒了。想象第一个 archive中存在一个空隙（e.g.它的 keys是 001, 002, 003, 081, 082, ...）。当在第二个 archive，搜索完 key 003后搜索 key 081 时，代码会遇到 keys 004, 005, ...，但是它能忽视这些对象，因为它知道081之前的 key 都不会再被请求（感谢“cs”选项）。如果第二个 archive中存在空隙，因为是排好序的，它能避免要搜索到文件末尾（这是“s”选项的作用）

io_sec_mapped

为了压缩很多程序中反复出现的特定代码模式，我们已经介绍了模板类 `RandomAccessTableReaderMapped` 。不像 `RandomAccessTableReader` ，这需要两个初始化参数，比如：

```
std::string rspecifier, utt2spk_map_rspecifier; // get these from somewhere.
   RandomAccessTableReaderMapped<BaseFloatMatrixHolder> transform_reader(rspecifier,
                                                                         utt2spk_map_rspe
```

如果 `utt2spk_map_rspecifier` 是空字符串，它和普通的 `RandomAccessTableReader` 表现得一样。如果不是空，e.g.ark:/data/train/utt2spk,它会读一个 utterance-to-speaker 的映射，并在任何查询到特定字符串 e.g.utt1的地方，将这个 utterance-id 映射到一个 speaker-id（e.g.spk1）然后用这个作为key来查询从 rspecifier读到的 table。这个 utterance-to-speaker 映射也是一个 archive，因为这碰巧是 Table代码最容易读的方式。

从命令行角度理解Kaldi的I/O机制

本页从用户命令行的角度来描述Kaldi的I/O机制。

代码层的分析见 Kaldi I/O mechanisms

Overview

Non-Table I/O

首先描述“non-table” I/O。这指的是只包含一个或两个对象的文件或数据流（e.g.声学模型文件；变换矩阵），而不是一系列由字符串索引的对象。

- Kaldi 默认文件格式是二进制，但是程序可以输出非二进制格式，如果你设定标志 `--binary=false`
- 很多对象都有相应的“copy”程序，e.g. copy-matrix或 gmm-copy，可以用标志 `--binary=false` 来转换成文本形式输出，e.g.“copy-matrix --binary=false foo.mat -”
- 通常磁盘上的一个文件和内存中的一个C++对象有一一对应关系，e.g.一个浮点型矩阵，尽管有些文件会包含多于一个的对象（典型的例子：声学模型文件中，通常是一个 `TransitionModel` 对象，接着是一个声学模型）
- Kaldi 程序一般知道它要读的对象的类型，而不是试图从数据流中去判断
- 和 perl类似，一个文件名可以用-（代表标准输入输出）替代，或者是一个字符串，比如“|gzip -c >foo.gz”或“gunzip -c foo.gz|”
- 读文件时，我们支持这样的定义，如 foo:1045，表示 foo文件内的1045字符偏移
- 为了指示我们扩展文件名的概念，我们一般用 rxfilename来表示一个被读的文件流（i.e.一个文件，流或标准的输入），用 wxfilename来表述一个输出流。更多细节见 Extended filenames: rxfilenames and wxfilenames

为了说明上述概念，确保 $KALDI_ROOT/src/bin 在你的工作路径中，其中 $KALDI_ROOT是Kaldi 库的路径，键入以下内容：

```
echo '[ 0 1 ]' | copy-matrix - -
```

它会打印出日志信息和对应于该矩阵的二进制数据。现在试试：

```
echo '[ 0 1 ]' | copy-matrix --binary=false - -
```

输出将是这样：

```
# copy-matrix --binary=false - -
copy-matrix --binary=false - -
 [
  0 1 ]
LOG (copy-matrix:main():copy-matrix.cc:68) Copied matrix to -
```

虽然看起来矩阵和日志信息混在一起了，但是日志信息是在标准错误流(stderr)中，并不会被传递给管道；为避免看到日志信息你可以重定向标准错误到 /dev/null，只要在命令行加入 2>/dev/null。

Kaldi 程序可以通过管道连接起来，利用 Kaldi I/O中流就是文件的机制。这里是一个管道的例子：

```
echo '[ 0 1 ]' | copy-matrix - - | copy-matrix --binary=false - -
```

这里以文本形式输出矩阵（第一个 copy-matrix命令转换到二进制形式，第二个转换到文本形式）。你可以以更加复杂的方式来完成同样的事情：

```
copy-matrix 'echo [ 0 1 ]|' '|copy-matrix --binary=false - -'
```

这里没必要这么做，但是当程序中有多个输入输出时它可能有用，因为这时 stdin或 stdout已经被占用。对 tables 这一点特别有用（见下节）。

Table I/O

Kaldi 在处理由字符串索引的对象集合时有特殊的 I/O机制。这方面的例子有，由 utterance-ids 索引的特征矩阵，或由 speaker-ids 索引的说话人自适应变换矩阵。索引的字符串必须是非空的且不包含空格。见 The Table concept 中更深入的讨论。

表(Table)可以有两种形式：“archive”和“script file”。区别在于，archive包含实际的数据，而 script file存储数据的地址。

从 Tables读数据的程序期望这一个“rspecifier”的字符串，来告诉如何读取索引的数据，写 Tables的程序也需要一个“wspecifier”的字符串来告诉如何写。这些字符串指明了得到的是 script file还是 archive，文件在磁盘的存储位置和一些其他选项。典型的 rspecifier有“ark:-”，意思是从标准输入读入 archive的数据，或“scp:foo.scp”，意思是指定从 foo.scp读数据。要记住几点：

- 冒号后面的部分被解释成 wxfilename或 rxfilename（同 Non-table I/O），意味着支持管道和标准输入/输出
- Table 总是只包含一类对象（e.g.一个浮点型矩阵）

- rspecifiers和 wspecifiers主要的选项有
  - rspecifiers中 ark,s,cs:- 意味着读的时候（这里是从标准输入）我们认为 keys是排好序的(,s)，而且声明它们会被顺序访问，(,cs)意味着我们知道程序会顺序访问它们（如果这些条件不满足，程序会报错）。这让 Kaldi 可以模拟随机访问，而无需占用大量内存。
  - 对占空间不大的或是不确定是排好序的（e.g.说话人自适应变换）数据,省略,s,cs也可以
  - 有多个 rspecifiers的程序一般会遍历第一个 rspecifier中的对象（顺序访问），并随机访问后面的。所以“,s,cs”对第一个 rspecifier没有必要
  - 在 scp,p:foo.scp中，,p 意味着即使部分搜索的文件不存在，也不会报错（对 archives，,p阻止了程序崩溃，即使 archive 发生损坏或截断）
  - 写数据时，,t 表示文本模式，e.g. ark,t:-。这里命令行选项 `-binary` 对 archives无效
- script file的格式是“ ”，e.g.“utt1/foo/bar/utt1.mat”。rspecifier 或 wspecifier 可以包含空格，e.g.“utt1 gunzip -c /foo/bar/utt1.mat.gz|”
- archive 的格式是 ...
- archives 可以被组合起来，仍然是有效的。但是要注意组合使得顺序。e.g.如果你需要排序的文件，请避免“cat a/b/*.ark”
- 尽管不太常用，但 script files可以作为输出文件，e.g.如果我们想写入 wspecifier scp:foo.scp，当程序想写入 key utt1时，它会在 foo.scp中寻找“utt1 some_file”这样的行，并把数据写入“some_file”。如果不存在这样的一行，就会报错。
- 我们可以同时写 archive 和 script。e.g.ark,scp:foo.ark,foo.scp。写入 script file中的行像“utt1 foo.ark:1016”（i.e.它指出了字节偏移）。这在数据被随机访问或分开存储时很有用，而且你并不希望产生很多小文件。
- 可以利用 archive的机制来操作单个文件。例如，

  ```
  echo '[ 0 1 ]' | copy-matrix 'scp:echo foo -|' 'scp,t:echo foo -|'
  ```

  这里需要解释下，首先，rspecifier“scp:echo foo -|”等效于“scp:bar.scp”，如果 bar.scp中只包含一行“foo -”。意思是从标准输入中读由“foo”索引的对象。同样，对于 wspecifier“scp,t:echo foo-|”，它把“foo”的数据写到标准输出。这一招不应该被过多使用。在这个例子里，这样做并没有必要，因为 copy-matrix 可以直接支持 non-table I/O，所以你也可以写成“copy-matrix - -”。如果你不得不用这样一个技巧，你最好还是去修改相关的程序代码。
- 如果你只想取出 archive中的一个元素，你可以在 wspecifier“scp:”加入“,p”选项来只写出这一个元素而忽略掉 scp文件中其他元素。假设你想要的“foo_bar”，在一个包含矩阵的 some_archive.ark中，你可以这样用来提取这一元素：

  ```
  copy-matrix 'ark:some_archive.ark' 'scp,t,p:echo foo_bar -|'
  ```

- 在特定情况下，读 archive的代码允许某些类型转换，如矩阵中的 float和 double类型，网格中的 Lattice和 CompactLattice。

Utterance-to-speaker and speaker-to-utterance maps

很多 Kaldi的程序都会有“utt2spk”或“spk2utt”这样的 utterance-to-speaker 和 speaker-to-utterance 映射文件。这些一般用命令行选项 `-uut2spk` 和 `-spk2utt` 指定。utt2spk 格式是这样

```
utt1 spk_of_utt1
utt2 spk_of_utt2
..
```

spk2utt 格式是这样

```
spk1 utt1_of_spk1 utt2_of_spk1 utt3_of_spk1
spk2 utt1_of_spk2 utt2_of_spk2
...
```

这些文件用于说话人自适应，e.g.用于查找某个 utterance属于哪个说话人，或遍历说话人。根据大部分 Kaldi示例脚本的设定和我们分割数据的方式，我们要确保 utterance-to-speaker 映射中的说话人事排好序的（见 Data preparation）。无论如何，这些文件都被当做 archives，所以你可以看到这样的命令行选项 `-utt2spk=ark:data/train/utt2spk` 。你可以看到这些文件符合通用的 archive 格式，即“ ”，这里的数据时文本格式的。在代码层，utt2spk 文件被当做一个包含一个字符串的 table，而 spk2utt 文件被当做包含了一系列字符串的 table。

Kaldi中的日志和错误报告

Overview

Kaldi 程序中的包含日志信息，警告信息和错误信息的输出都被定向到标准错误流。这也是我们的程序可以直接用在管道里，而这些信息不回和程序的输出所混淆的原因。最常用的生成日志，警告或错误信息输出的方式是通过宏 KALDI_LOG, KALDI_WARN 和 KALDI_ERR。宏 KALDI_ERR 被激活时一般程序都会被终止（除非异常被捕获）。一个代码片段的例子如下：

```
KALDI_LOG << "On iteration " << iter
          << ", objective function change was " << delta;
if (delta < 0.0) {
  KALDI_WARN << "Negative objf change " << delta;
  if (delta < -0.1)
    KALDI_ERR << "Convergence failure in EM";
}
```

注意这些例子中不包含换行符（这个是自动添加的）。它产生的消息的一个典型的例子是：

```
WARNING (copy-feats:Next():util/kaldi-table-inl.h:381) Invalid archive file format
```

对于不太重要的日志信息（或太冗余的），以至不必在正常的日志中出现的，你可以用 KALDI_VLOG，例如

```
KALDI_VLOG(2) << "This message is not important enough to use KALDI_LOG for.";
```

如果 `-verbose` 选项设置的比括号中的数字大或者相等，它会被打印出来。e.g.如果程序的参数 `-verbose=2` 或更大，上面的信息就会被打印。更多内容见 Implicit command-line arguments

部分代码直接打印日志信息到标准错误流；这种做法不被提倡。

Assertions in Kaldi

断言最好用宏 KALDI_ASSERT 来完成。这比通常的 assert() 会给出更多信息；KALDI_ASSERT 打印出堆栈信息，同时也更容易配置。

一个断言的例子是：

```
KALDI_ASSERT(i < M.NumRows());
```

一个获得 assert-failure 更多信息的技巧是在断言语句后加上“&&[some string]”，例如

```
KALDI_ASSERT(ApproxEqual(delta, objf_change) && "Probable coding error in optimization");
```

正常编译时断言都会被检查，除非你定义了 NDEBUG。对占用大量CPU的内环断言，我们用下面的模式：

```
#ifdef KALDI_PARANOID
  KALDI_ASSERT(i>=0);
#endif
```

在当前的设定中，宏 KALDI_PARANOID 默认是定义了的。

Exceptions thrown by KALDI_ERR

宏 KALDI_ERR 被调用时，错误信息被输出到标准错误流，然后抛出 std::runtime_error 类型的异常。异常信息的字符串包含了错误信息和堆栈信息（如果OS支持）。目前 Kaldi 程序通过 `main()` 中的 `try...catch` 模块来捕捉异常，即把异常信息打印至标准错误流并推出。这通常会导致错误信息被打印了两遍。

某些情况下，错误信息会被 Kaldi 代码捕捉到而不会重新抛出。这发生在 Holder 代码（见 Holders as helpers to Table classes），并被 Table 代码调用（见 The Table concept）。这里 Kaldi 对象的 `Read` 函数抛出的异常会被捕捉并作为一个布尔型返回值传递给 Table 代码（e.g.见 KaldiObjectHolder::Read()）。根据选项，例如 Table 代码中的“p”(permissive)选项，和 Table 代码被调用的方式，这可能会也可能不会导致另一个异常。

正常情况下，Kaldi代码中除了 std::runtime_error，另一类应该被抛出的错误是 std::alloc_error。

部分 Kaldi 代码中，目前会直接抛出 std::runtime_error 或直接调用 assert()，这在以后会更新成标准的宏 KALDI_ERR 和 KALDI_ASSERT。

Compile-time assertions in Kaldi

可以在编译时设置断言（如果失败，它们会产生编译错误）。这通过 `kaldi-utils.h` 中定义的一些宏来实现。特别有用的是，确保模板在实例化时有正确的类型。编译断言的例子有：

```
KALDI_COMPILE_TIME_ASSERT(kSomeConstant < 0);
...
template<class T> class foo {
   foo() { KALDI_ASSERT_IS_INTEGER_TYPE(T);
};
...
template<class T> class bar {
   bar() { KALDI_ASSERT_IS_FLOATING_TYPE(T);
}
```

解析命令行选项

Introduction

`ParseOptions` 类负责解析通过 argc 和 argv 传递给main()的命令行选项。首先我们给出一个从命令行调用 Kaldi 程序的例子：

```
gmm-align --transition-scale=10.0 --beam=75 \
       exp/mono/tree exp/mono/30.mdl data/L.fst \
       'ark:add-deltas --print-args=false scp:data/train.scp ark:- |' \
        ark:data/train.tra ark:exp/tri/0.ali
```

命令行选项只有长形式（没有单字符选项），必须出现在位置参数前面。在这里例子里有六个位置参数，从“exp/mono/tree”开始；注意以“ark:add-delta”开头的是一个带空格的单个字符串；单引号里的内容会被 shell 解释；这个参数作为管道被调用。

Example of parsing command-line options

我们解释下这些选项是如何被 C++ 处理的，下面是 `gmm-align.cc` 中的部分代码（为了表达更清晰，略微改动了下）：

```
int main(int argc, char *argv[])
{
  try { // try-catch block is standard and relates to handling of errors.
    using namespace kaldi;
    const char *usage =
        "Align features given [GMM-based] models.\n"
        "Usage: align-gmm [options] tree-in model-in lexicon-fst-in feature-rspecifier "
        "transcriptions-rspecifier alignments-wspecifier\n";
    // Initialize the ParseOptions object with the usage string.
    ParseOptions po(usage);
    // Declare options and set default values.
    bool binary = false;
    BaseFloat beam = 200.0;
    // Below is a structure containing options; its initializer sets defaults.
    TrainingGraphCompilerOptions gopts;
    // Register the options with the ParseOptions object.
    po.Register("binary", &binary, "Write output in binary mode");
    po.Register("beam", &beam, "Decoding beam");
    gopts.Register(&po);
    // The command-line options get parsed here.
    po.Read(argc, argv);
    // Check that there are a valid number of positional arguments.
    if(po.NumArgs() != 6) {
      po.PrintUsage();
      exit(1);
    }
    // The positional arguments get read here (they can only be obtained
    // from ParseOptions as strings).
    std::string tree_in_filename = po.GetArg(1);
    ...
    std::string alignment_wspecifier = po.GetArg(6);
    ...
  } catch(const std::exception& e) {
    std::cerr << e.what();
    return -1;
  }
}
```

读上面的代码基本就明白它的功能。在通常的 Kaldi 程序中，处理流程是这样的：

- 用包含使用说明的字符串初始化 `ParseOptions` 对象
- 声明并给出可选参数（选项结构(options structure)）的初始值
- 用 `ParseOptions` 对象注册命令行选项（选项结构有它们自己的注册函数，执行同样的功能）
- 执行 “po.Read(argc, argv);”[如果存在无效选项，此处会退出程序]
- 检查 `po.NumArgs()` 在有效范围内
- 利用 `po.GetArg(1)` 等获取位置参数；对可能超出有效数目范围的可选位置参数来说，可用 `po.GetOptArg(n)` 来获取第n个参数，如果n超出范围则返回空字符串

一般来说，在 Kaldi 程序中写一个新命令行时，最简单的方法就是拷贝一个现成的然后修改它。

Implicit command-line arguments

特定的命令行选项会被 `ParseOptions` 对象自动注册。这些包括：

- `-config` 这个选项会从配置文件中导入命令行选项。e.g.如果我们设定 `-config=configs/my.conf`， `my.conf` 文件可能包含：

```
--first-option=15  # This is the first option
--second-option=false # This is the second option
```

- `-print-args` 这个布尔型选项控制程序是否将命令行参数打印到标准错误流（默认是真）；`-print-args=false` 会将它关掉
- `-help` 这个布尔型参数，如果是真，程序会打印出使用说明（像其他布尔型参数一样，只用 `-help` 就认为是真）。一般你不输入任何命令行参数时就可以获得使用说明，因为大部分程序都需要有位置参数。
- `-verbose` 它控制冗余度，进而决定用 KALDI_VLOG 输出的日志信息是否被打印。值越大打印的信息越多（e.g.典型值是 `-verbose=2`）

其他Kaldi组件

本页给出了 Kaldi 代码中用到的各种类型的组件函数。

这里不包括已经在其他章节分析了的重要组件，如 matrix library, I/O, logging and error reporting, and command-line parsing。

Text utilities

`text-utils.h` 中是各种操作字符串的函数，多用于解析命令行。重要的几个包括模板函数 `CnoverStringToInteger()` 和为 float 和 double 重载后的 `ConvertStringToReal()` 。还有模板 `SplitStringToIntegers()` 输出时整型向量，和 `SplitStringToVectors()` 将字符串切分成字符串向量。

STL utilities

`stl-utils.h` 中是操作STL类型的模板函数。常用的一个是 `SortAndUniq()` ，给向量排序并去重（对任意类型）。函数 `CopySetToVector()` 复制 set 中元素到 vector 中，属于在sets, vectors 和 maps 之间移动数据的一大类函数中的一个（见 stl-utils.h 中的列表）。还有哈希函数类 `VectorHasher` （用于整型向量）和 `StringHasher` （用于字符串）；它们和STL的 unordered_map 和 unordered_set 模板一起使用。另一个常用函数是 `DeletePointers()` ，删除 std::vector 类型的指针，将其设置为 NULL。

Math utilities

`kaldi-math.h` 中除了提供一些标准 #defines 以防它们不在系统头文件 math.h 中，还有其他的数学通用函数。这些包括比较重要的：

- 随机数生成函数： `RandInt()` , `RandGuass()` , `RandPoisson()`
- `LogAdd()` 和 `LogSub()`
- 测试和判断数学上近似相等的函数，i.e. `ApproxEqual()` , `AssertEqual()` , `AssertGeq()` , `AssertLeq()`

Other utilities

`const-integer-set.h` 包含了一个存储整型集合的，允许快速查询的 `ConstIntegerSet` 类。需要说明的是该对象在初始化后不能再被改动。这用在了e.g.决策树代码中。根据集合中的整数类型，它们作为 vector 或整型向量被存储。

一个平台无关的程序计时类 `Timer` 在 `timer.h` 中。

其他组件类函数有 `simple-io-funcs.h` 和 `hash-list.h`，有更专门的用途。一些额外矩阵代码依赖的的组件函数和宏，多数是有专门用途的，在 `kaldi-utils.h` 中；这些包括字节交换，内存对齐和编译时的断言（对模板很有用）。

Kaldi中的聚类机制

本页解释了Kaldi中使用的通用的聚类机制和接口。

相关的类和函数参见Classes and functions related to clustering 。本页没有讲述音素的决策树聚类（见内部决策树和Kaldi中如何使用决策树），并且本页中介绍的类和函数在音素聚类的低层代码中有使用。

Clusterable接口

Clusterable类是一个纯虚类，被GaussClusterable类继承（GaussClusterable类表示高斯统计量）。以后我们会添加其他继承Clusterable类的聚类对象。建立Clusterable类的原因是它可以允许我们使用通用的聚类算法。

Clusterable接口的核心概念是统计量相加和目标函数的测量。两个Clusterable对象之间的距离概念是指首先分别测量两个对象的目标函数，然后将其相加并再次测量目标函数；目标函数减值的负值即给出了我们的距离测度。

我们打算加入Clusterable类的地方包括从固定、共享或混合高斯模型的后验概率导出的混合高斯统计量，以及离散观察矢量的计数集合（）

HMM的拓扑结构和状态转移的建模

介绍

在这里我们将介绍在kaldi用如何表示HMM topologies和我们如何让建模和训练HMM 转移概率的。我们将简要的说下它是如何跟决策树联系的；决策树你可以在How decision trees are used in Kaldi和Decision tree internals这些地方看到更详细的； 对于这个里面的类和函数，你可以看Classes and functions related to HMM topology and transition modeling。

HMM topologies

HmmTopology类是为用户指定到HMM音素的拓扑结构。在平常的样例里，脚本创建一个HmmTopology 对象的文本格式的文件，然后给命令行使用。为了说明这个对象包含什么，接下来的就是一个HmmTopology 对象的文本格式的文件(the 3-state Bakis model)：

```
<Topology>
<TopologyEntry>
 <ForPhones> 1 2 3 4 5 6 7 8
 </ForPhones>
 <State> 0 <PdfClass> 0
<Transition> 0 0.5
<Transition> 1 0.5
</State>
 <State> 1 <PdfClass> 1
<Transition> 1 0.5
<Transition> 2 0.5
 </State>
 <State> 2 <PdfClass> 2
<Transition> 2 0.5
<Transition> 3 0.5
 </State>
<State> 3
 </State>
</TopologyEntry>
 </Topology>
```

这里在这个特定的HmmTopology对象里有一个TopologyEntry，它覆盖音素1到8(所以在这个例子里有8个音素和他们共享相同的topology)。他们有3个发射态；每一个有个自环和一个转移到下一个状态的。这里也有一个第四个非发射态，状态3 (对于它没有 entry) 没有任何的转移。这是一个这些topology entries的标准特征；Kaldi 把第一个状态(state zero) 看做开始状态。最后一个状态，一般都是非发射状态和没有任何的转移，但会有最终的概率的那一个。在一个HMM中，你可以把转移概率看做最后一个状态的最终概率。在这个特定的例子里所有

的发射状态他们有不同的pdf's (因为PdfClass数目是不同的)。我们可以强制数目一样。在对象HmmTopology里概率是用来初始化训练的；这个训练概率对于上下文相关的HMM来说是特定的和存储在TransitionModel对象。TransitionModel用一个类名来存储HmmTopology对象，但是要注意，在初始化TransitionModel后，一般不使用在HmmTopology 对象的转移概率。这里有个特例，对于那些不是最终的非发射状态(比如：他们有转移但是没有 entry)，Kaldi不训练这些转移概率，而是用HmmTopology对象里给的概率。对于不支持非发射状态的转移概率的训练，实际上简化了我们的训练机制，和由于非转移状态有转移是不正常的，所以我们就觉得这些没有很大的损失。

Pdf-classes

pdf-class是一个与HmmTopology对象有关的概念。HmmTopology对象为每个音素指定了一个HMM拓扑结构。HMM拓扑结构的每一个状态数有一个"pdf_class"变量。如果两个状态有相同的pdf_class变量，他们将共享相同的概率分布函数(p.d.f.)，如果他们在相同的音素上下文。这是因为决策树代码不能够直接“看到”HMM状态，它仅仅可以看到pdf-class。在正常的情况下，pdf-class和HMM状态的索引(e.g. 0, 1 or 2)一样，但是pdf classes为用户提供一种强制共享的方式。如果你想丰富转移模型但是想离开声学模型，这个是非常有用的，要不然就变成一样的。pdf-class的一个功能就是指定非发射状态。如果一些HMM状态的pdf-class 设定常量 kNoPdf = -1，然后HMM状态是非发射的(它与pdf没有联系)。这个可以通过删除标签和相联系的数字来简化对象的文本格式。 一个特定的HMM拓扑结构的pdf-classes的设置期望是从0开始和与其邻近的(e.g. 0, 1, 2)一些值。这些是为了图建立代码的方便和不会导致任何的损失。

Transition models (the TransitionModel object)

TransitionModel对象存储转移概率和关于HMM拓扑结构的信息(它包括一个HmmTopology 对象)。图建立代码根据TransitionModel对象来获得topology 和转移概率(它也需要一个ContextDependencyInterface对象来获得与特定音素上下文的pdf-ids)。

How we model transition probabilities in Kaldi

我们跟据下面的4件事情决定一个上下文相关的HMM状态的转移概率(你可以把他们看做4-tuple):

- 音素(HMM里的)
- 源HMM-state (我们解释成HmmTopology对象, i.e. normally 0, 1 or 2)
- pdf-id (i.e.与状态相关的pdf索引)
- 在HmmTopology对象的转移的索引

这4项可以看做在HmmTopology对象里的目标HMM状态的编码。根绝这些事情来求转移概率的原因是，这是最细粒度（the most fine-grained way）的方式，我们在不增加编译解码图的大小来对转移建模。对于训练转移概率也是非常方便的。事实上，用传统的方法来对转移尽

可能精确的建模可能没有任何区别，和在单音素层面上的共享转移的HTK方式可能更加有效。

The reason for transition-ids etc.

TransitionModel对象当被初始化时，就被设置成许多整数的映射，也被其他部分的代码用作这些映射。除了上面提到的数量，也有转移标识符(transition-ids), 转移索引(这个与转移标识符不是一样的), 和转移状态。之所以要介绍这些标识符和相联系的映射，是因为我们可以用一个完整的基本FST的训练脚本。最自然的基本FST的设置是输入标签是pdf-ids。然而，考虑到我们的树建立的方法，它不是总可能从pdf-id唯一的映射到一个音素，所以那样很难从输入的标签序列映射到音素序列，那样就不是很方便了；一般很难仅仅用FST的信息来训练转移概率。基于这个原因，我们把转移标识符transition-ids作为FST的输入标签，和把这些映射到pdf-id，也可以映射到音素，也可以映射到一个prototype HMM (在HmmTopology对象里)具体的转移概率。

Integer identifiers used by TransitionModel

以下类型的标识符都用于TransitionModel接口，它们所有的都用int32类型。注意其中的一些量是从1开始索引的，也有的是从0开始索引的。我们尽可能的避免从1开始索引的，因为与C++数组索引不兼容，但是因为OpenFst把0看做特定的情况(代表特殊符号epsilon), 我们决定把那些频繁在FST中用作输入符号的从1开始索引，我们就允许从1开始。更重要的是，transition-ids 是从1开始的。因为我们没有想到作为FST的标签，pdf-ids出现很频繁。因为我们经常使用他们作为C++数组索引，我们使他们从0开始，但当pdf-ids作为FST的输入标签，我们就给pdf-ids加1。当我们阅读TransitionModel代码时，我们应该对索引从1开始的量考虑，如果是这种情况我们就减去1，如果不是就不用减去。我们记录这些声明的成员变量。在任何情况，这些代码不是公开接口的，因为那会引起很多困惑。在TransitionModel中用的许多整数量如下：

- 音素(从1开始): 标识符的类型贯穿整个工具箱；用OpenFst符号表可以把pdf-ids转换成一个音素。这类的id不一定是邻近的(工具箱允许跳过音素索引)。
- Hmm状态(从0开始)：这是HmmTopology::TopologyEntry类型的索引，一般取值为{0, 1, 2}里的一个。
- Pdf或者pdf-id (从0开始)：这是p.d.f.的索引，由决策树的聚类来初始化(看PDF identifiers)。在一个系统里通常由几千个pdf-ids。
- 转移状态，或者trans_state (从0开始)：这是由TransitionModel 自己定义的。每个可能的三(phone, hmm-state, pdf) 映射到一个自己的转移状态。转移状态可以认为是HMM状态最细化的分割尺度，转移概率都是分别评估的。
- 转移索引或者trans_index (从0开始)：这是HmmTopology::HmmState类型中的转移数组的索引，这个索引记录了转移出某个转移状态的次数。
- 转移标识符或者trans_id (从0开始)：其中的每一个都对应转移模型中的唯一的一个转移概率。用从(transition-state, transition-index)到transition-id的映射，反之亦然。

在转移模型代码中也参考如下的概念：

- A triple意思就是a triple (phone, hmm-state, pdf)从转移状态中映射。
- A pair意思就是a pair (transition-state, transition-index)从转移id中映射。

Training the transition model

转移模型的训练步骤是非常简单的，我们创建的FST(训练和测试中)将transition-ids作为他们的输入标签。在训练阶段，我们做维特比解码，生成输入标签的序列，就是每一个transition-ids的序列(每个特征向量一个，意思也就是对应每帧输出有限个transition-id的序列)。训练转移概率时我们积累的统计量就是数每个transition-id在训练阶段出现了多少次(代码本身用浮点数来数，但在正常的训练时我们仅仅用整数)。函数Transition::Update()对每一个transition-state做最大似然更新。这个明显是非常有用的。这里也有一些与概率下限有关的小的问题，和如果一个特定的transition-state不知道，我们怎么做？更多的细节，请看代码。

Alignments in Kaldi

这里我们将介绍对齐的概念。说到对齐，我们意思就是类型的向量，含有transition-ids的序列，(c.f. Integer identifiers used by TransitionModel)它的长度跟需要对齐的那句话一样长。transition-ids序列可以通过对输入标签序列解码获得。对齐通常用在训练阶段的维特比训练和在测试时间的自适应阶段。因为transition-ids编码了音素的信息，所以从对齐中计算出音素的序列是可行的 (c.f. SplitToPhones() and ali-to-phones.cc)。 我们经常需要处理以句子id做索引的一系列对齐。为了方便，我们用表来读和写“对齐”，更多的信息可以看 I/O with alignments 。 函数ConvertAlignment() (c.f.命令行convert-ali) 把对齐从一个转移模型转换成其他的。这种典型的情况就是你使用一个转移模型(由特定的决策树来创建)来做对齐，和希望将其转换为不同的树的其他的转移模型。最好的就是从最原始的音素映射到新的音素集；这个特征通常不是一定要的，但是我们使用它来处理一些由减少的音素集的简化的模型。 在对齐的程序里通常使用一个后缀"-ali"。

State-level posteriors

状态级的后验概率是“对齐”这个概念的扩展，区别于每帧只有一个transition-ids，这里每帧可以有任意多的transition-ids，每个transition-ids都有权重，存储在下面的结构体里：

```
typedef std::vector<std::vector<std::pair<int32, BaseFloat> > > Posterior;
```

如果我们有个Posterior类型的对象"post"，post.size()就等于一句话的帧数，和post[i]是一串pair (用向量存储), 每一个pair有一个(transition-id, posterior)构成。 在当前的程序里，有两个方式创建后验概率：

- 用程序ali-to-post来讲对齐转换为后验，但是生成的Posterior对象很细小，每帧都有一个

单元后验的transition-id

- 用程序weight-silence-post来修改后验，通常用于减小静音的权重。

未来，当加入lattice generation，我们将会从lattice生成后验。 与"-ali"相比,读入的后验程序没有后缀。这是为了简洁；读取状态层的后验要考虑对齐信息的这类程序的默认值。

Gaussian-level posteriors

一个句子的高斯级别的后验概率用以下的形式存储：

```
typedef std::vector<std::vector<std::pair<int32, Vector<BaseFloat> > > > GauPost;
```

这个和状态级的后验概率结构体非常相似，除了原来是一个浮点数，现在是一个浮点数向量(代表状态的后验)，每个值就是状态里的每个高斯分量。向量的大小跟pdf里的高斯分量的数目一样，pdf对应于转移标识符transition-id，也就是每个pair的第一个元素。 程序post-to-gpost把后验概率的结构体转换为高斯后验的结构体，再用模型和特征来计算高斯级别的后验概率。当我们需要用不同的模型或特征计算高斯级别的后验时，这个就有用了。读取高斯后验程序有一个后缀"-gpost"。

Functions for converting HMMs to FSTs

把HMMs转换为FSTs的整个函数和类的列表可以在here这里找到。 GetHTransducer() 最重要的函数就是GetHTranducer(), 声明如下：

```
fst::VectorFst<fst::StdArc>*
GetHTransducer(const std::vector<std::vector<int32> > &ilabel_info,
constContextDependencyInterface &ctx_dep,
constTransitionModel &trans_model,
constHTransducerConfig &config,
std::vector<int32> *disambig_syms_left);
```

如果没有介绍ilabel_info对象，ContextDependencyInterface接口，和fst在语音识别里的一些应用基础，这个函数的很多方面大家会很难理解。这个函数返回一个FST，输入标签是transition-ids，输出标签是代表上下文相关的音素(他们有对象ilabel_info的索引)。FST返回一个既有初始状态又有最终状态，所有的转移都有输出标签(CLG合理利用)。每一个转移都是一个3状态的HMM结构，和循环返回最初状态。FST返回GetHTransducer()仅仅对表示ilabel_info的上下文相关音素有效，我们称为特定。这是非常有用的，因为对于宽的上下文系统，有很多数量的上下文，大多是没有被使用。ilabel_info对象可以从ContextFst (代表C)对象获得，在合并它和一些东西，和它含有已经使用过的上下文。我们提供相同的ilabel_info对象给GetHTransducer()来获得覆盖我们需要的一个H转换器。 注意GetHTransducer()函数不包括自环。这必须通过函数AddSelfLoops()来添加；当所有的图解码优化后，仅仅添加自环是很方便的。

The HTransducerConfig configuration class

The HTransducerConfig configuration class控制了GetHTransducer行为。

- 变量trans_prob_scale是一个转移概率尺度。当转移概率包含在图里，他们就包含了这个尺度。命令行就是–transition-scale。对尺度的合理使用和讨论，可以看Scaling of transition and acoustic probabilities。
- 一个变量reverse和二个其他变量是可以修改的，如果"reverse"选项是true。"reverse"选项是为后向解码创建一个time-reversed FSTs。为了使这个有用，我们需要添加功能到Arpa语言模型里，我们将在以后去做。

The function GetHmmAsFst()

函数GetHmmAsFst() 需要一个音素上下文窗和返回一个用transition-ids作为符号的对应的有限状态接受器。这个在GetHTransducer()使用。函数GetHmmAsFstSimple() 有很少的选择被提供，是为了表示在原理上是怎么工作的。

AddSelfLoops()

AddSelfLoops() 函数把自环添加到一个没有自环的图中。一个典型的就是创建一个没有自环的H转换器，和CLG一起，做确定性和最小化，和然后添加自环。这样会使确定性和最小化更有效。AddSelfLoops() 函数哟选项来重新对转移排序，更多的细节可以看Reordering transitions。它也有一个转移概率尺度，"self_loop_scale",这个跟转移概率尺度不是一样的，可以看Scaling of transition and acoustic probabilities。

Adding transition probabilities to FSTs

AddTransitionProbs() 函数添加转移概率到FST。这样做时有用的，原因是我们创造了一个没有转移概率的图(i.e. without the component of the weights that arises from the HMM transitions), 和他们将在以后被添加；这样就使得训练模型的不同的迭代使用相同的图成为可能，和保持图中的转移概率更新。创造一个没有转移概率的图是使用trans_prob_scale (command-line option: –transition-scale)为0来实现的。在训练的时候，我们的脚本开始存储没有转移概率的图，然后每次我们重新调整时，我们就增加当前有效的转移概率。

Reordering transitions

AddSelfLoops()函数有一个布尔选项"reorder"，这个选项表明重新对转移概率排序，the self-loop comes after the transition out of the state。当应用变成一个布尔命令行，比如，你可以用 –reorder=true 在创建图时重新排序。这个选项使得解码更加简单，更快和更有效(看Decoders used in the Kaldi toolkit)，尽管它与kaldi的解码不兼容。 重排序的想法是，我们切换自环弧与所有其他状态出来的弧，所以自环维语每一个互相连接的弧的目标状态（The idea

of reordering is that we switch the order of the self-loop arcs with all the other arcs that come out of a state, so the self-loop is located at the destination state of each of the other arcs）。为了这个，我们必须保证FST有某些特性，即一个特定状态的所有弧必须引导相同的自环(也，一个有自环的状态不能够有一个静音的输入弧，或者是开始的状态)。AddSelfLoops() 函数修改图，以确保他们有这个特性。一个相同的特性也是需要的，即使"reorder"选项设置为false。创建一个有"reorder"选项的图准确的说就相对于你解码一个句子得到的声学和转移模型概率而言，是一个非重新排序的图 。得到的对齐的transition-ids 是一个不同的顺序，但是这个不影响你使用这些对齐。

Scaling of transition and acoustic probabilities

这里有三种在kaldi中使用的尺度类型：

```
Name in code    Name in command-line arguments    Example value (train)    Example value
acoustic_scale    –acoustic-scale=?    0.1    0.08333
self_loop_scale    –self-loop-scale=?    0.1    0.1
transition_scale    –transition-scale=?    1.0    1.0
```

你也许注意到这里没有用语言模型的尺度；相对于语言模型来说，任何事情都是尺度。我们不支持插入一个词的惩罚，一般来说(景观kaldi的解码不支持这个)。语言模型表示的真正的概率，一切相对于他们的尺度都是有意义的人。在训练阶段的尺度是我们在通过Viterbi 对齐解码得到的。一般而言，我们用0.1，当一个参数不是很关键和期望它是小的。当测试是很重要的和这个任务是调整的，声学尺度将被使用。现在我们来解释这三个尺度：

- 声学尺度是应用到声学上的尺度(比如：给定一个声学状态的一帧的log似然比).
- 转移尺度是转移概率上的尺度，但是这个仅仅适合多个转换的HMM状态，它应用到这些转移中的相对的权重。它对典型的没有影响。
- 自环尺度是那些应用到自环的尺度。特别的是，当我们添加自环，让给定自环的概率为p，而剩下的为(1-p)。我们添加一个自环，对数概率是self_loop_scale *log(p),* 和对所有其他不在这个状态的对数转移概率添加*(self_loop_scale* log(1-p))。在典型的topologies，自环尺度仅仅是那个有关系的尺度。

我们觉得对自环应用一个不同的概率尺度，而不是正常的转移尺度有意义的原因是我们认为他们可以决定不同时间上事件的概率。稍微更加微妙的说法是，所有的转移概率可以被看成真正的概率(相对于LM 概率), 因为声学概率的相关性的问题与转移没有关系。然而，一个问题出现了，因为我们在测试时使用Viterbi 算法(在我们的例子里，训练也是)。转移概率仅仅表示当累加起来的真正的概率，在前向后向算法里。我们期望这个是自环的问题，而不是由于HMM的完全不同的路径的概率。因为后面的情况里，声学分布通常不是连接的，在前向后向算法和 Viterbi 算法里的差别很小。

内部决策树

这页将描述音素决策树代码的内部机制，这个是用通用的数据结构和算法来实现的。从更高层的解释整个算法是如何实现的和怎么在kaldi使用的，可以看How decision trees are used in Kaldi。

Event maps

决策树建立代码的核心概念就是the event map,用类EventMap来描述。不要被event这个词误认为这是发生在特定事件的。event仅仅代表一个(key,value)对的一个集合，没有key是重复的。从概念上讲，它是可以被类型std::map `<int32,int32>` 来描述。事实上，为了更加有效的使用，我们把它表示为a sorted vector of pairs，通过typedef：

```
Typedef std::vector<std::pair<EventKeyType,EventValueType> > EventType;
```

这里，EventKeyType和EventValueType仅仅定义为int32，但是如果我们可以从他们的名字里区分，这个代码是非常容易理解的。想象一下一个Event（类型EventType）当做一个向量的集合，而且变量的名字和值都为整数。也有一个类型EventAnswerType，和它们也是int32，它是由EventMap返回的类型；事实上它是一个pdf标识符（声学状态索引）。获取EventType的函数期望你可以首先对它进行排序（比如：通过调用std::sort(e.begin(),e.end())）。

EventMap类的用意可以通过一个例子很好的阐述。假设我们的上下文音素是a/b/c；假设我们有一个标准的三音素拓扑结构；和假设在这个上下文中我们想问音素"b"的中心状态的pdf对应的索引是多少。这个问题里这个状态将是状态1，因为我们使用从0开始的索引。当我们说到"state"，我们将跳过一些细节；更多的细节你可以看Pdf-classes。假设音素a，b和c的整数索引分别为10，11和12。"event"对应的映射为：

```
(0  -> 10), (1  -> 11), (2  -> 12), (-1 -> 1)
```

在三音素窗"a/b/c"的0，1和2是位置，和-1是一个特殊的索引，我们用这个来编码state-id(请参照：常量kPdfClass = -1)的一个特殊的索引。用这种方式的话，我们的一个排好序的向量对是：

```
// caution: not valid C++.
EventType e = { {-1,1}, {0,10}, {1,11}, {2,12} };
```

假设相对应的声学状态的索引(pdf-id)是1000。然后如果我们有一个表示tree的EventMap"emap"，然后我们期望接下来的设置不会失效：

```
EventAnswerTypeans;
boolret = emap.Map(e, &ans); // emap is of type EventMap; e is EventType
KALDI_ASSERT(ret == true && ans == 1000);
```

因此当我们指出EventMap是从EventType映射到EventAnswerType，你可以认为这大概是一个从上下文音素到整数索引的一个映射。音素上下文是由(key-value)对的集合表示，和原则上我们可以添加新的keys和投入更多的信息。注意函数EventMap::Map()返回bool。这是因为它可能对于某些events来说，不映射到其他任何集(比如：考虑一个无效的音素，或者想象一下，当EventType不包含所有的EventMap查询的keys)。

EventMap是一个很被动的对象。它没有任何学习决策树的能力，它仅仅就是存储决策树。可以把它看成表示一个从EventType到EventAnswerType的函数的一个结构。EventMap是一个多态，纯虚类(比如： 它不能被实例化，因为它具有不实施虚拟功能)。实现EventMap接口需要三个具体的类：

- ConstantEventMap: 认为它是一个决策树叶子结点。这个类存储类型EventAnswerType的一个整数和它的映射函数常常返回这个值。
- SplitEventMap: 认为它是一个决策树的非叶子结点，它查询一个特定的key，和根据问题的答案去"yes"或者"no"孩子结点。它的映射函数调用合适的孩子结点的映射函数。它存储对应"yes"孩子结点的类型kAnswerType的整数集(其他任何事情将对应到"no")。
- TableEventMap: 这个在一个特定的key上做一个完整的分裂。一个典型的例子就是：你也许首先完全分裂中心音素，然后你对这个音素的每一值有一个分离的决策树。内部地它存储EventMap*指针的一个向量。它查找对应分裂的key的值，和调用向量对应位置的EventMap的映射函数。

除了从EventType映射到EventAnswerType，EventMap的确不能够做很多其他事情。它的接口不能提供允许你遍历树的函数，无论是向上还是向下。这里仅仅有一个函数允许你修改EventMap,和这个函数就是EventMap::Copy()函数，这样的声明(作为一个类成员)：

```
virtualEventMap *Copy(const std::vector<EventMap*> &new_leaves) const;
```

这个与函数组合有一样的功能。如果你用一个空的向量"new_leaves"调用Copy()，它将仅仅返回整个对象的一个深度拷贝，拷贝它沿着树往下的所有的指针。然而，如果new_leaves不为空，每次函数Copy()到达一个叶子，如果这个叶子1是在(0, new_leaves.size()-1)范围内的和new_leaves[l]不为空， 函数Copy()将返回调用new_leaves[l]->Copy()的结果，而不是Copy()函数的叶子本身(将返回一个新的ConstantEventMap)。一个典型的例子就是当你决定对一个特定的叶子来分裂，比如叶子852。你将创建一个vector类型的一个对象，它的非空数字是在位置852那里。它包含一个指针指向一个真正类型为SplitEventMap的对象，和"yes"和"no" 指针是有叶子值为852和1234的ConstantEventMap类型。(我们为新的叶子重新私用旧的叶子结点)。真正的树建立代码不是这么低效的。

Statistics for building the tree

用来建立音素决策树的统计量如下：

```
typedefstd::vector<std::pair<EventType, Clusterable*> > BuildTreeStatsType;
```

这种类型的一个对象的参考被传递到所有决策树建立过程中。这些统计量被期望包含相同的EventType成员的no duplicates，比如它们从概念上表示一个从EventType到Clusterable的映射。 在我们目前的代码中Clusterable对象是一个真正的类型GaussClusterable，但是树建立代码不知道这个，积累这些统计量的程序是acc-tree-stats.cc。

Classes and functions involved in tree-building

Questions (config class)

类Questions是一个与树建立代码相交互的类，行为有点像类"configuration"。它是一个从"key" 值(类型EventKeyType)到一个类型QuestionsForKey的configuration类的真正映射。

QuestionsForKey类有一个"questions"集，每一个是类型EventValueType的整数集他们大对数对应于音素集，或者如果key是-1，他们就是对应HMM状态索引集的一个典型例子，比如{0,1,2}的子集。QuestionsForKey 包含一个类型RefineClustersOptions的一个configuration类。它在树建立代码中控制树建立的行为将试图在分裂的两边迭代的移动这些值(e.g. phones)来最大化似然比(as in K-means with K=2)。然而，这些可以通过设定"refining clusters"的迭代数目为0来关闭，相当于仅仅从一个固定列表的问题集中选择。这看起来效果要好点。

Lowest-level functions

一个完整的列表，可以看Low-level functions for manipulating statistics and event-maps；我们这里将总结一些重要的。

这些函数大多是涉及到类型BuildTreeStatsType的一些操作，作为我们下面看到的一个(EventType,Clusterable*)对的一个向量。最简单的一个是DeleteBuildTreeStats(), WriteBuildTreeStats()和ReadBuildTreeStats()。

函数PossibleValues()寻找在一个统计量集中一个特定的key对应的值(和通知用户这个值是否经常被定义)；SplitStatsByKey()将通过一个特定的key对应的值来分裂类型BuildTreeStatsType的一个对象为一个向量(比如：在中心音素上分裂)；SplitStatsByMap()做同样的事情，但是索引不是这个key的值，但是由EventMap返回的答案被提供给这个函数。

SumStats()在一个BuildTreeStatsType对象里求统计量(i.e对象Clusterable)的和返回对应的Clusterable*对象。SumStatsVec()采用一个类型vector的对象和输出为 vector类型的量，比如它像SumStats()但是是一个向量；处理SplitStatsByKey()和SplitStatsByMap()的输出是非常有

用的。

给定一些统计量和EventMap，ObjfGivenMap()评估这个目标函数：它在每一个聚类中对所有的统计量求和(相对应EventMap::Map()函数的每一个不同的答案)，通过这些聚类来添加目标函数，和返回整个。

FindAllKeys() 将找到所有定义在统计量集合里的值，和根据不同的参数，它可以找到被定义为所有的event的所有keys，或者为其他一些event定义的所有的keys(比如采用定义的keys的交集或者并集)。

Intermediate-level functions

涉及到树建立的下一组函数被列在here 和对应树建立的不同阶段。我们现在将提一些有代表的。

首先，我们指出许多这些函数有一个参数int32 *num_leaves。这个指向的整数作为一个计数器来分配新的叶子。在树建立的开始阶段，这个caller被设定为0。每次一个新节点被需要时，树建立的代码采用当前的数字来指向作为新的leaf-id，和然后增加它。

一个重要的函数是GetStubMap()。这个函数返回一个没有分裂的树，比如pdfs 不根据上下文。这个函数的输入控制所有的音素要不是不同的要不他们的一些是共享决策树的根节点，和或者在一个特定的音素里所有的状态共享相同的决策树结点。

SplitDecisionTree()函数采用输入一个EventMap，对应有GetStubMap()创建的类型的一个分裂的"stub"决策树，和做决策树分裂，知道无论是叶子的最大数目达到了，还是分裂一个叶子的增益比特定的门限小。

函数ClusterEventMap()采用一个EventMap和一个门限，和只要这样做的成本小于阈值，合并EventMap的叶子。在SplitDecisionTree()后，这个函数正常的被调用。这里有一些操作限制的函数的不同版本 (e.g.从不同的音素里去避免合并叶子)。

Top-level tree-building functions

高层次树建立的函数被列在here。这些函数可以从命令行里直接调用。最重要的一个就是BuildTree()。它被赋予了一个Questionsconfiguration类，和音素集的一些信息，这些音素他们的根节点是共享的和(对于音素的每一个集合)无论是在单决策树的根节点共享还是对每一个pdf-class有一个分离的根节点。它去掉关于音素的长度的信息，加上不同的门限。它建立树和返回一个用来构建ContextDependency对象EventMap对象。

在这组里有一个重要的事是AutomaticallyObtainQuestions()，它用来获得通过自动对音素聚类的问题。它把音素聚类成树，和对树的每一个节点，从这个节点的所有叶子组成一个问题(问题等价于音素集)。这个(从cluster-phones.cc调用)使我们的脚本独立于人产生的音素集。

给定特征和transition-ids的一个序列的对齐，函数AccumulateTreeStats()为训练树来积累统计量(看Integer identifiers used by TransitionModel)。这个函数被定义在一个与树建立相关函数的不同的目录(hmm/ not tree/)，它依赖于更多的代码(e.g.it knows about the theTransitionModel class),和我们更愿意保持对核心的树建立代码的依赖性更小。

Kaldi中如何使用决策树

介绍

这部分将介绍音素决策树在kaldi中是如何建立和使用的，以及是如何将训练和图建立交互的。对于决策树的构建的代码，可以看Decision tree internals; 对于建立图解码的更多细节，可以看Decoding graph construction in Kaldi。

最基本的实现方法就是自顶向下贪婪的分裂，这里我们有很多的方法来分裂我们的数据，比如我们可以用左音素、右音素、中间音素，和我们所在的状态等等。我们实现的方法与标准的方法很相似，具体的你可以去看Young，Odell 和Woodland的论文"Tree-based State Tying for High Accuracy Acoustic Modeling"。在这个方法里，我们通过局部最优问题来分裂数据，比如假设我们通过一个单高斯来分裂我们需要建模的数据，通常我们会用最大似然函数的增加来作为分裂的依据。与标准实现方法不同的有：对怎么计算树的根节点来添加flexibility；对hmm状态和中间音素决策的能力；但是在kaldi脚本中默认的是，决策是通过对数据自顶向下的自动实现二叉树聚类，意思就是我们不需要手工产生的这个决策。不考虑树的根节点的构造：它可能用在一个单共享组中的所有音素的所有统计量来进行来分裂(包括中心音素和HMM状态的问题), 或者用一个单一的音素，或者HMM状态里的音素，来作为分裂树的根节点，或者音素组作为树的根节点。对于在标准的脚本里如何构建，你可以看Data preparation。在实际中，我们让每一个树的根节点对应一个真正的音素，意思就是我们重组了所有word-position-dependent, 每一个音素的tone-dependent 或者stress-dependent组成一个组，成为树的根节点。 这页的剩下部分大多数介绍代码层的一些细节。

上下文相关音素窗（Phonetic context windows）

这里我们将解释在我们的代码里是如何描述上下文相关音素。一个特定的树含有两个值，他们分别描述上下文音素窗的宽度和中心位置。下表是总结这些值：

Name in code	Name in command-line arguments	Value (triphone)	Value (monophone)
N	–context-width=?	3	1
P	–central-position=?	1	0

N代表上下文相关音素窗的宽度，P表示指定中心音素。正常的P是窗的中心（因此名字叫中心位置）；举个例子，当N=3，中心位置就是P=1，但是你可以自由的选择0到N-1中的任何值；比如，P=2和N=3表示有2个音素在左边的上下文，而右边没有上下文。(译者注：因为是从0开始数，所以0,1在2的左边，即没有右边的上下文)。在代码中，当我们讨论中心音素时，我们的意思是第P个音素，不一定是上下文相关音素窗的中心音素。

一个整型的vector表示一个典型的三音素上下文窗也许就是：

```
// probably not valid C++
vector<int32> ctx_window = { 12, 15, 21 };
```

假设N=3和P=1，这个就代表音素15有一个右边的上下文21和左边的上下文12。我们用这种方式处理句子的结尾时就用0（表示不是一个有效的音素，因为在OpenFst里的epsilon表示没有符号），所以举个例子：

```
vector<int32> ctx_window = { 12, 15, 0 };
```

表示音素15有一个左上下文和没有右上下文，因为是一句话的结尾处。尤其在一个句子的结尾处，这种用0的方式也许有一点出乎意料，因为最后一个音素事实上是后续符号"$" (看Making the context transducer)，但是在决策树代码里为了方便，我们不把后续符号放在这些上下文窗中，我们直接给其赋0。注意如果我们有N=3和P=2，上述的上下文窗将无效，因为第P个元素是0，它不是一个真正的音素；当然，如果我们用一个树的N=1, 所有的窗都将无效，因为他们是错误的大小。在单因素的情况下，如果的窗就像如下表示：

```
vector<int32> ctx_window = { 15 };
```

所以单音素系统是上下文相关系统的一个特殊情况，窗的大小N=1和一个不做任何事情的树。

树建立的过程（The tree building process）

在这部分我们将介绍在kaldi中的树建立的过程。

如果我们的一个单音素系统有一个决策树，那也应该是比较简单。看函数MonophoneContextDependency()和MonophoneContextDependencyShared()，他们将返回这个简单的树。这个在命令行里叫gmm-init-mono; 他们主要的输入是HmmTopology类和输出是树，通常把作为类ContextDependency写到一个名字叫树的文件（tree），和它们的模型文件(模型文件含有一个TransitionModel类和一个AmDiagGmm类)。如果程序gmm-init-mono接受一个叫–shared-phones的选项，它将在音素特定集中共享概率密度函数；否则就把所有的音素分离。

在训练一个以flat start开始的单音素系统时，我们训练树是采用单音素对齐和使用函数AccumulateTreeStats() (称为acc-tree-stats)来累积统计量。这个程序不仅仅在单音素对齐中使用;也可以对上下文相关音素来对齐,所以我们可以根据三音素对齐来建立树。为树建立的统计量将写到磁盘里,作为类型 BuildTreeStatsType (看Statistics for building the tree)。函数AccumulateTreeStats() 采用值N和P，像我们在之前的那部分解释那样；命令行将默认的设定它们为3和1,但是这个可以用–context-width和–central-position选项来覆盖。程序acc-tree-stats采用上下文音素 (e.g. silence)的列表，但是如果是上下文音素，他们是不需要的。它仅仅是一个减少统计量的一个机制。对于上下文相关音素，程序将积累相对应的统计量without the keys corresponding to the left and right phones defined (c.f. Event maps)。

当统计量已经积累，我们将使用程序build-tree 来建立树。输出是一棵树。程序build-tree需要三个东西：

- 统计量(BuildTreeStatsType类的)
- 问题的构建(Questions 类)
- 根文件(看下面)

统计量是通过程序acc-tree-stats得到的；问题构建类是通过问题构建程序得到的，这需要在音素问题集的一个topology表中 (在我们的脚本里，他们通常从程序cluster-phones的树建立统计量自动获得)。根文件指定音素集，这些音素集是在决策树聚类处理中共享根节点，对于每一个音素需要设定2个东西：

- "shared" or "not-shared" 意思是对于每一个pdf-classes是否有单独的根节点(例如：典型的情况，HMM的状态), 或者根节点是否共享。如果我们打算去分裂(下面有个"split" 选项)，我们执行时，根节点是共享的。
- "split"或者"not-split"意思决策树分裂是否应该通过根节点的问题来决定(对于silence, 我们一般不split)。

要小心点，因为这些符号有些棘手。在根文件里的行"shared"是表示我们是否在一个单一的树的根节点上共享的所有的三个HMM状态。但是我们经常共享所有音素的根节点，这些音素在一个根文件的一行上。这个不是通过这些字符串来构建的，因为如果你不想共享它们，你会把它们分布在根文件的不同的行上。

下面是根文件的一个例子；这里假设音素1是silence和所有的其他有分离的根节点。

```
not-shared not-split 1
shared split 2
shared split 3
...
shared split 28
```

当我们有像position and stress-dependent音素时，在同一行上有许多音素是非常有用的。这种情况下，每一个"real"音素将对应整数音素ids的一个集合。在这种情况下，我们对一个特定标注的音素的所有情况将共享根节点（译者注：意思应该就是一个音素可能有不同的标注版本，我们用的时候就相当于一个，具体的可以看下面的根文件）。下面是wsj数据库的一个根文件，在egs/wsj/s5脚本里(这个在text里，不是整数形式；它将经过kaldi转为整数形式):

```
not-shared not-split SIL SIL_B SIL_E SIL_I SIL_S SPN SPN_B SPN_E SPN_I SPN_S NSN NSN_B NS
shared split AA_B AA_E AA_I AA_S AA0_B AA0_E AA0_I AA0_S AA1_B AA1_E AA1_I AA1_S AA2_B AA
shared split AE_B AE_E AE_I AE_S AE0_B AE0_E AE0_I AE0_S AE1_B AE1_E AE1_I AE1_S AE2_B AE
shared split AH_B AH_E AH_I AH_S AH0_B AH0_E AH0_I AH0_S AH1_B AH1_E AH1_I AH1_S AH2_B AH
shared split AO_B AO_E AO_I AO_S AO0_B AO0_E AO0_I AO0_S AO1_B AO1_E AO1_I AO1_S AO2_B AO
shared split AW_B AW_E AW_I AW_S AW0_B AW0_E AW0_I AW0_S AW1_B AW1_E AW1_I AW1_S AW2_B AW
shared split AY_B AY_E AY_I AY_S AY0_B AY0_E AY0_I AY0_S AY1_B AY1_E AY1_I AY1_S AY2_B AY
shared split B_B B_E B_I B_S
shared split CH_B CH_E CH_I CH_S
shared split D_B D_E D_I D_S
```

当创建根文件时，你应该确保每一行至少有一个音素。举例来说，在这种情况下，如果音素AY应该至少在stress和word-position有一些组成，我们就认为是可以的。

在这里例子里，我们有许多word-position-dependent静音的变异体等等。在这里例子里，他们将共享他们的pdf，因为他们在同一行，而且是"not-split"，但是他们有不同的转移参数。事实上，静音的大多数变异体从来不被用作词间出现的静音；这些都为了未来做准备，以防止在未来有人做了一些奇怪的变化。

我们初始阶段的高斯混合是用之前(比如：单音素)的建立来做对齐；对齐是通过程序convert-ali来把一个树转换成其他的。 PDF标识符 (identifiers) PDF标识符(pdf-id)是一个数字，从0开始，被用作概率分布函数(p.d.f.)的一个索引。在系统中的每一个p.d.f.有自己的pdf-id，和他们是邻近的(典型的就是在一个LVCSR有上千上万个)。当树一开始建立的时候，他们就被初始化了。这有可能取决于树是如何建立的，对于每一个pdf-id，都有一个音素与之对应。

上下文相关类（Context dependency objects）

类ContextDependencyInterface对于一个与图建立相交互的特定树来说，是一个虚拟的基类。这种交互仅仅需要4个函数：

- ContextWidth()返回树所需要的N(context-width)的值。
- CentralPosition()返回树所需要的P (central-position)值。
- NumPdfs()返回由树定义的pdfs的数量，他们的值是从0到NumPdfs()-1。
- Compute()是计算一个特定的上下文 (和pdf-class)的pdf-id的函数。

函数ContextDependencyInterface::Compute() Compute() 如下声明：

```
class ContextDependencyInterface {
...
virtualboolCompute(const std::vector<int32> &phoneseq, int32 pdf_class,
int32*pdf_id) const;
}
```

如果可以计算这个上下文和pdf-class的pdf-id就返回true。如果是false，就说明许多种类是错误的或者不匹配的。用这个函数的一个例子是：

```
ContextDependencyInterface *ctx_dep = ... ;
vector<int32> ctx_window = { 12, 15, 21 }; // not valid C++
int32pdf_class = 1; // probably central state of 3-state HMM.
int32pdf_id;
if(!ctx_dep->Compute(ctx_window, pdf_class, &pdf_id))
KALDI_ERR<< "Something went wrong!"
else
KALDI_LOG<< "Got pdf-id, it is " << pdf_id;
```

从类ContextDependencyInterface继承的唯一一个类就是类ContextDependency，将轻微的丰富这种交互；唯一一个重要的添加就是函数GetPdfInfo，它将在类TransitionModel中计算一个特定的pdf对应某一个音素(通过枚举所有的上下文，这个函数可以模拟特定的ContextDependencyInterface的交互)。

对象ContextDependency事实上是对象EventMap的一点点的改动；可以看Decision tree internals。我们想要尽最大可能地隐藏树的真正的实现来使以后我们需要重组代码的时候更加的简单。

决策树的一个例子(An example of a decision tree)

决策树文件的格式不是以可读性为第一的标准创建的，而是由于大家的需要，我们将尝试解释如何来翻译这个文件。可以看下面的例子，这是在wsj数据库下的一个三音素的决策树的例子。它由一个名字叫ContextDependency对象开始的，然后N (上下文窗的宽度)等于3，P (上下文窗的中心位置)等于1，比如这个音素上下文位置的中心，我们是从0开始编号的。文件的剩下部分包含一个单一的对象EventMap 。EventMap是一个多态类型，它包含指向其他EventMap对象的指针。更多的细节可以看Event maps；它是一个决策树或者决策树集合的表示，它把key-value对的集合(比如：left-phone=5, central-phone=10, right-phone=11, pdf-class=2)映射成一个pdf-id (比如158)。简单的说，它有三个类型：SplitEventMap (像决策树里的一个分裂)，ConstantEventMap (像决策树里的叶子，只包含一个数字，代表一个pdf-id)，和TableEventMap(像含有其他的EventMaps表中查找)。SplitEventMap和TableEventMap都有一个他们查询的"key"，在这种情况下可以为0，1或者2，与之对应的是left，central或者right context，或者-1代表"pdf-class"的标识。通常pdf-class的值与HMM状态的索引值一样，比如0，1或者2。尝试不要因为这样而迷惑：key的值是-1，但是value是0，1或者 2，和他们在上下文窗里的音素的keys的 0，1或者2 没有任何联系。SplitEventMap 有一组值，这组值将触发树的"yes"分支。下面是解释树文件格式的一种quasi-BNF标识。

```
EventMap := ConstantEventMap | SplitEventMap | TableEventMap | "NULL"
ConstantEventMap := "CE" <numeric pdf-id>
SplitEventMap := "SE" <key-to-split-on> "[" yes-value-list "]" "{" EventMap EventMap "}"
TableEventMap := "TE" <key-to-split-on> <table-size> "(" EventMapList ")"
```

在下面的这个例子里，树的顶层EventMap 是一个分裂的key为1，表示中心音素SplitEventMap (SE)。在方括号里的是phone-ids的邻近值。就像上面说的那样，他们不是代表一个问题，而仅仅是在音素上的分裂的方式，以至于我们可以得到真正的每个音素的决策树。关键就是这个树是通过"shared roots"来建立的，所以这里有许多的phone-ids，对应相同音素的不同版本的word-position-and-stress-marked，他们共享树节点。我们不可能在树的顶层使用 TableEventMap (TE)，或者我们不得不重复决策树很多次 (因为EventMap是一个纯树，不是一个普通的图，它没有任何的机制被指向"shared")。"SE" label 的接下来的几个参数也是这个"quasi-tree"的一部分，它一开始就用中心音素来分类 (当我们继续往下看这个文件，我们将更加对这个树深入；注意到括号"{" 是打开的而不是闭合的)。然后我们的字符串"TE -1 5 (CE 0 CE 1 CE 2 CE 3 CE 4)"，表示在pdf-class "-1"(effectively, the HMM-position)用TableEventMap来分裂，和通过4返回值0。这个值代表对静音和噪声音素SIL, NSN and SPN的5个pdf-ids；在我们的建立中，这个pdfs是在这些三个非语音音素(对于每一个非语音音素只有转移矩阵是特定的)是共享的。注意：对于这些音素，我们有5个状态而不是三个状态的HMM，因此有5个不同的pdf-ids。接下来是"SE -1 [0]"；和这些可以被认为在树中的第一个"real" 问题。当中心音素的值是5到19时，我们可以从上面提供的看出SE问题，这个是音素AA的不同版本；问题就是pdf-class (key -1)的值是否为0 (i.e. the leftmost HMM-state)。假如这个答案是"yes"，接下来的问题就是"SE 2 [220 221 222 223]"，这就问音素的右边是不是音素"M"的各种形式(一个相对不直观的问题被问，因为他们是leftmost HMM-state)；如果是yes，我们将问"SE 0 [104 105 106 107... 286 287]" 音素的右边的一个问题，如果是yes，然后pdf-id就是5 ("CE 5") 和如果是no,就是696 ("CE 696")。

```
s3# copy-tree --binary=false exp/tri1/tree - 2>/dev/null | head -100
ContextDependency 3 1 ToPdf SE 1 [ 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21
{ SE 1 [ 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30
{ SE 1 [ 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30
{ SE 1 [ 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 ]
{ SE 1 [ 1 2 3 ]
{ TE -1 5 ( CE 0 CE 1 CE 2 CE 3 CE 4 )
SE -1 [ 0 ]
{ SE 2 [ 220 221 222 223 ]
{ SE 0 [ 104 105 106 107 112 113 114 115 172 173 174 175 208 209 210 211 212 213 214 215
{ CE 5 CE 696 }
SE 2 [ 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 36 37 38 39 40 41 42 43 44 45 46 47 48 4
```

下面是一个简单的例子：从rm数据库的一个单音素tree。顶层EventMap是一个TableEventMap ("TE 0 49 ..."）。key "0"表示的就是中心音素，因为context width (N)为1。在这个表中项的数量是 49 (这种情况下，音素的数量加1)。在表(索引为0)中的第一个EventMap是NULL，因为这里没有索引为0的音素。接下来是一个有三个元素的TableEventMap，与第一个音素的三个HMM状态(technically, pdf-classes)相对应："TE -1 3 (CE 0 CE 1 CE 2)"。

```
s3# copy-tree --binary=false exp/mono/tree - 2>/dev/null| head -5
ContextDependency 1 0 ToPdf TE 0 49 ( NULL TE -1 3 ( CE 0 CE 1 CE 2 )
TE -1 3 ( CE 3 CE 4 CE 5 )
TE -1 3 ( CE 6 CE 7 CE 8 )
TE -1 3 ( CE 9 CE 10 CE 11 )
TE -1 3 ( CE 12 CE 13 CE 14 )
```

The i label_info object

CLG图(看Decoding graph construction in Kaldi) 在输入边有代表上下文音素的符号(也可能是消歧符号或者可能是静音符号)。在这中，这些会被表示成整数型的标签。我们在代码中用一个对象，在文件名我们通常称为ilabel_info。ilabel_info对象跟ContextFst对象有很强的联系，可以看The ContextFst object。像kaldi中许多其他的类型一样，ilabel_info是一个通用(STL)的类型，但是我们可以使用一致的变量名，以至于可以被识别出来。就是下面的类型：

```
std::vector<std::vector<int32> > ilabel_info;
```

它是一个vector，是通过FST 输入标签来索引的，它可以给每一个输入标签相对应的音素上下文窗(看上面的Phonetic context windows)。举例来说，假设符号1500是音素30，有一个12的右边的上下文和4的左边的上下文，我们将有：

```
// not valid C++
ilabel_info[1500] == { 4, 30, 12 };
In the monophone case, we would have things like:
ilabel_info[30] == { 28 };
```

这是对处理消歧符号的一个特殊处理(看Disambiguation symbols或者Springer Handbook paper referenced above for an explanation of what these are)。如果一个ilabel_info entry对应一个消歧符号，我们把它放在消歧符号的符号表的负部分(note that this is not the same as the number of the printed form of the disambiguation symbol as in #0, #1, #2 etc，这些数字对应符号表文件的消歧符号，这个在我们现在的脚本中叫phones_disambig.txt)。举例来说，

```
ilabel_info[5] == { -42 };
```

意思就是在HCLG中的符号数字5对应一个整数id为42的消歧符号。我们是为了脚本的方便，所以对于给定的消歧符号列表解释ilabel_info对象的程序不是很需要，在单音素的情况下，是为了可以从真正的音素中区分它们。这里是二个其他的特殊的例子，我们有：

```
ilabel_info[0] == { }; // epsilon
ilabel_info[1] == { 0 }; // disambig symbol #-1;
// we use symbol 1, but don't consider this hardwired.
```

第一个表示正常的静音符号，我们将给它一个空的vector作为它的ilabel_info entry。这个符号不会出现CLG的左边。第二个是一个特殊的消歧符号，它的印刷形式为"#-1"。在正常的脚本里，我们使用它，这里的静音被用作C转换器的输入；它是确保在空音素表示的词里对CLG确定性。

程序fstmakecontextsyms是可以创建一个与ilabel_info对象的打印形式相对应的符号表；这主要用来调试和诊断。

就像你看到的那样，ilabel_info对象不是很方便，因为它涉及到消歧符号，但是与它密切交互的代码不是很多：仅仅fst::ContextFst 类(一些相关的东西可以看"Classes and functions related to context expansion")，程序fstmakecontextsyms.cc，和一些在Classes and functions for creating FSTs from HMMs列出来的函数)。特别地，ContextDependency对象，仅仅可以看到代表音素上下文窗的长度N的有效的序列。

Kaldi中解码图的建立

首先，我们不详细介绍有限状态转换以及它在语音识别中的应用。

需要了解的，可以看"Speech Recognition with Weighted Finite-State Transducers" by Mohri, Pereira and Riley (in Springer Handbook on Speech Processing and Speech Communication, 2008)。我们要用的主要方法那里面都有介绍，但是一些细节，尤其是像怎样处理混淆符，怎样处理weight-pushing可能与那篇论文中的就不一样了。

Overview of graph creation

整个解码网络都是围绕HCLG = H o C o L o G的图来建立的。

- G是用来编码语法或者语言模型的接收器(i.e：它的输入和输出符号是一样的)
- L是发声词典；输出是词，输入是音素。
- C表示上下文关系，输出是音素，输入是上下文相关的音素例如N个音素组成的窗。具体看Phonetic context windows里面的介绍。
- H包含HMM的定义；输出符表示上下文相关的音素，输入是transitions-ids(转移id)，transitions-ids是编码pdf-id或者其他信息(自转或向后转)，具体看(Integer identifiers used by TransitionModel)

以上是标准方法，然而还有好多细节没有说明。我们想确保输出是确定化的和最小化的，为了让HCLG确定化，我们插入了消歧符。关于消歧符更多的介绍，看下面的Disambiguation symbols。

我们也想让HCLG尽可能的随机。传统的方法是通过"push-weights"来达到这种效果。我们确保随机性方法与传统的不同，是基于确保没有消除随机性的图构建步骤；具体看Preserving stochasticity and testing it。

如果我们用一行公式来总结我们的方法（很明显一行不能覆盖所有的细节），那一行应该是下面这样，asl=="add-self-loops" rds=="remove-disambiguation-symbols"， H' 是 H不带自环的：

HCLG = asl(min(rds(det(H' o min(det(C o min(det(L o G)))))))

Weight-pushing没有被我们采用。相反，我们的目标是只要G是随机的，这样就可以确保图的构建过程中没有消除结果随机性的。当然，G不会是绝对随机的，因为带补偿的Arpa语言模型是被用FSTs表示，但是至少我们的方法确保不随机性的部分不会增加，不会比开始变的更糟糕；这种方法避免了"push-weights"操作失败或者是变更坏的危险。

Disambiguation symbols

消歧符是在被插在词典中音素序列末尾的类似 #1, #2, #3的符号。在词典中当一个音素序列是另一个音素序列的前缀，或者是这个音素序列出现在多个词中，就需要把这些符号加在它的后面。这些符号用来确保 L o G 输出时是确定化的。我们也插入混淆符在两个其它的地方。我们把#0加在语言模型G的补偿弧上，当删除静音后(确定化的方法删除静音)，确保G是确定化的。我们也加#-1 在出现在上下文FST C左边的静音的地方，这种情况出现在句子的开始。这对于解决当有的词是用空音素()表示的问题是必要的。

下面是关于怎样证明图编译的中间过程 (e.g. LG, CLG, HCLG) 是确定化的一个概述。这对于确保我们的方法永远不失败是很重要的。我们这里讲的确定化是删除静音后的确定化。主要步骤是: 首先保证G必须是确定化的，这就是为什么需要#0 (这里G确实是确定化的)。然后对于任何确定化的G，我们想让L也是这样，那么L o G也是确定化的。[对于C来说也是一样，把右边的G换成L o G 即可]。这里还有很多理论的细节需要充实，但是我认为对于L有以下两点属性就够了：

•必须是函数形式的

```
相当于：对于任何的输入序列在L中必须有唯一的输出序列
相当于：对于任何线性接受器A，A o L是线性转换或是空。
```

• L具有双胞胎属性，比如：同一个输入符号序列不可能对应两个可达到的状态，也就是它们有相同输入序列但不同权重或者不同输出序列的自环。 这对转换器C同样适用。我们认为我们的脚本和代码创建的转换器都具有这些属性。

The ContextFst object

ContextFst类的对象 (C)是一个动态建立的FST对象，用来表示一个从上下文相关的音素到上下文独立的音素的转换器。这个对象是用来避免我们在上下文中罗列所有可能的音素，当上下文窗长 (N)或音素数量比平常大很多时罗列所有音素的方法将会很困难。
ContextFst::ContextFst转换器需要上下文窗宽(N)和中间音素位置(P)这些都会在之前的Phonetic context windows中有进一步的解释。对于三音素系统他们的常用值是3和1。特殊符号的整数id也是需要的，"subsequential symbol"(上文中提到的$)，当所有音素都被处理完后FST输出N-P-1次$(用来确保context FST的输出是确定化的)。除此之外也要整数id的音素和消歧符列表。ContextFst 输出端的词表包括音素，和加上subsequential symbol的消歧符。输入端的词表是自动生成的(除了静音，它未被使用), 相当于内容有上下文相关的音素，混淆符，以及一些特殊符号在传统方法中用 #-1 代替静音，然而这里我们把他们看成别的混淆符(这是用来确保确定化的, i.e.删除静音)。subsequential symbol '$'和输入端没有太多联系(对传统方法来说也一样)。输入端混淆符的id和输出端的相关符号的整数id不是绝对一样。ContextFst类对象有一个ILabelInfo()的函数，是用来对std::vector >型对象返回引用，使用户能求出输入端每个符号的"meaning"。相关的解释可以查找The ilabel_info object。

这是ContextMatcher类的特殊对象"Matcher"，它是在组合算法中被使用(Matcher 是组合算法用来做弧查找，更多细节请看the OpenFst documentation)。 ContextMatcher 让ContextFst的对象使用更高效通过避免分配更多不必要的状态((the issue is that with the normal matcher, every time we want any arc out of a state, we would have to allocate the destination states of all other arcs out of that state)。当组合的左边参数是ContxtFst型时，这是一个相关函数，ComposeContextFst()是用来执行FST组合操作的。也有一个函数很类似ComposeContext()，但是用来创建ContextFst对象。

Avoiding weight pushing

我们处理权重前推的整个问题上和传统方法上有微小的差别。 权重前推对于log半环在加速搜索是有效的(权重前推是一个FST操作确保每个状态出弧概率的和在合适的情况下相加为1)。然而在一些情况下 权重前推有负面影响。这个问题来自用FSTs表示统计语言模型，通常相加的结果大于1，因为一些词被记了两次（直接的或通过回退弧）。

我们决定避免可能的权重前推，所以使用了不同的方法来处理问题。1.定义"stochastic" FST，它的权重和为1，读者可以假设这里只考虑log半环，而不是tropical，所以"sum to one"就是加和的意思，不是取最大。要确保图创建的每一过程都具有那种属性：如果前一个过程是随机的，后一个过程也将是随机的。也就是说，如果G是随机的，LG也是随机的；如果LG是随机的，那么det(LG)也是随机的；如果det(LG)是随机的，那么min(det(LG))也是随机的，等等。意思就是，每个独立的操作必须在一定情况下保持随机性。现在通过一种琐碎但是很有用的方法可以实现：例如：we could just try the push-weights algorithm and if it seems to be failing because, say, the original G fst summed up to more than one, then we throw up our hands in horror and announce failure. This would not be very helpful.

我们想保持随机性在某方面，也就是说首先，对于G，所有状态中最小和最大的，以及这些状态的出弧概率与最后的概率的和。这是我们程序"fstisstochastic"能实现的。如果G是随机的，所有的这些数字就都是1（你也从程序中看到是0，因为我们在log空间中执行的操作；这里是log半环）。我们想保持随机性在一下方面：当最小或最大从来不变的最坏；换句话说，他们从来不会离1很远。事实上，这很自然会发生，当我们的算法用局部的方法维保随机性。还有好多算法需要保持随机性，包括：• Minimization最小化操作 • Determinization确定化操作 • Epsilon removal删除空 • Composition 组合操作(with particular FSTs on the left) There are also one or two minor algorithms that need to preserve stochasticity, like adding a subsequential-symbol loop. Minimization naturally preserves stochasticity, as long as we don't do any weight pushing as part of it (we use our program "fstminimizeencoded" which does minimization without weight pushing). Determinization preserves stochasticity as long as we do it in the same semiring that we want to preserve stochasticity in (this means the log semiring; this is why we use our program fstdeterminizestar with the option --determinize-in-log=true). Regarding epsilon removal: firstly, we have our own version of epsilon removal "RemoveEpsLocal()" (fstrmepslocal), which doesn't guarantee to remove all epsilons but does guarantee to never "blow up". This algorithm is unusual among FST algorithms in that,

to to what we need it to do and preserve stochasticity, it needs to "keep track of" two semirings at the same time. That is, if it is to preserve equivalence in the tropical semiring and stochasticity in the log semiring, which is what we need in practice, it actually has to "know about" both semirings simultaneously. This seems to be an edge case where the "semiring" concept somewhat breaks down. Composition on the left with the lexicon L, the context FST C and the H tranducer (which encodes the HMM structure) all have to preserve stochasticity. Let's discuss this the abstract: we ask, when composing A o B, what are sufficient properties that A must have so that A o B will be stochastic whenever B is stochastic? We believe these properties are: • For any symbol sequence that can appear on the input of B, the inverse of A must accept that sequence (i.e. it must be possible for A to output that sequence), and: • For any such symbol sequence (say, S), if we compose A with an unweighted linear FST with S on its input, the result will be stochastic. These properties are true of C, L and H, at least if everything is properly normalized (i.e. if the lexicon weights sum to one for any given word, and if the HMMs in H are properly normalized and we don't use a probability scale). However, in practice in our graph creation recipes we use a probability scale on the transition probabilities in the HMMs (similar to the acoustic scale). This means that the very last stages of graph creation typically don't preserve stochasticity. Also, if we are using a statistical language model, G will typically not be stochastic in the first place. What we do in this case is we measure at the start how much it "deviates from stochasticity" (using the program fstisstochastic), and during subsequent graph creation stages (except for the very last one) we verify that the non-stochasticity does not "get worse" than it was at the beginning. 至少如果•每个都能被恰当的归一的话(例如，如果对任何给定的词，词典权重之和都为1，如果H中的HMMs状态被恰当的归一，而没用概率尺度表示)，这些属性对C,L,H也都是真实存在的。然而我们在建图的实际应用中用概率尺度表示HMMs的转移概率（类似声学尺度）。这就是说图构建的最后的过程不能保持随机性。如果我们用了一个随机的语言模型，G首先不会是随机的。

解码图创建示例（测试阶段）

这里我们将解释怎样一步一步的建立我们正常的图模型，可能会涉及到一些数据准备阶段的知识。

这个方法的许多细节没有被编码到我们的工具里；我们仅仅解释现在我们是怎么做的。如果这部分你感到迷惑，最好的办法就是去读Mohri写的"Speech Recognition with Weighted Finite-State Transducers" 。警告你的是，这篇文章非常长，如果你对FST不熟悉可能需要花费几个小时。还有一个更好的资料就是OpenFst website ， 可以提供像符号表的更多的介绍。

Preparing the initial symbol tables

我们现在准备OpenFst符号表words.txt和phones.txt。在我们的系统里，他们用整数id来分配所有的词和音素。注意OpenFst为静音保留符号0。对于WSJ任务的一个符号表的例子：

```
## head words.txt
<eps> 0
!SIL 1
<s> 2
</s> 3
<SPOKEN_NOISE> 4
<UNK> 5
<NOISE> 6
!EXCLAMATION-POINT 7
"CLOSE-QUOTE 8
## tail -2 words.txt
}RIGHT-BRACE 123683
#0 123684
## head data/phones.txt
<eps> 0
SIL 1
SPN 2
NSN 3
AA 4
AA_B 5
```

Words.txt文件包含单个消歧符号"#0" (在G.fst的输入里用于静音)。在我们的脚本里这是最后数字的词。如果你的词典含有一个词"#0"，你就要小心点。phones.txt 文件不包含消歧符号，但是当我们创建L.fst 后，我们将创建一个含有消歧符号的phones_disambig.txt 文件(在调试时这是非常有用的)。

Preparing the lexicon L

首先我们创建一个文本格式的词典，初始化时没有消歧符号。我们的C++ 工具将不会跟它交互，它仅仅在创建词典FST是被使用。我们WSJ词典的一小部分是：

```
## head data/lexicon.txt
!SIL SIL
<s>
</s>
<SPOKEN_NOISE> SPN
<UNK> SPN
<NOISE> NSN
!EXCLAMATION-POINT EH2_B K S K L AH0 M EY1 SH AH0 N P OY2 N T_E
"CLOSE-QUOTE K_B L OW1 Z K W OW1 T_E
```

音素的开始，结束和应急标记物(e.g. T_E, or AH0) 在我们的WSJ脚本中是特定的，和至于我们的工具箱，它们被视为不同的音素(然而，对于这种设置，我们在树建立时特定的处理；可以看在The tree building process里的根文件)。

注意我们允许含有空音素表示的词。在训练时词典被用来创建L.fst(不含消歧符号)。我们也创建一个含消歧符号的词典，在图解码创建的时候使用。这个文件的一部分在这里：

```
# [from data/lexicon_disambig.txt]
!SIL    SIL
<s> #1
</s>    #2
<SPOKEN_NOISE>  SPN #3
<UNK>   SPN #4
<NOISE> NSN
...
{BRACE  B_B R EY1 S_E #4
{LEFT-BRACE L_B EH1 F T B R EY1 S_E #4
```

这个文件通过一个脚本创建；脚本的输出就是我们需要添加的消歧符号的数目，和这个用来创建符号表phones_disambig.txt。这个phones.txt一样，但是它也包含消歧符号#0, #1, #2等等的整数id (#0是从G.fst得到的一个特殊的消歧符号，但是将在L.fst由自环过滤掉)。phones_disambig.txt文件中间的一部分是：

```
ZH_E 338
ZH_S 339
#0  340
#1  341
#2  342
#3  343
```

这个数字是非常大，因为在WSJ脚本中我们为音素添加stress and position信息。注意用在空词的消歧符号(比如 `<s>` 和 `</s>)` 与用在正常词的消歧符号是不一样的，所以在这个例子里正常消歧符号是从#3开始的。

把没有消歧符号的词典转换为FST的命令是：

```
scripts/make_lexicon_fst.pl data/lexicon.txt 0.5 SIL | \
  fstcompile --isymbols=data/phones.txt --osymbols=data/words.txt \
  --keep_isymbols=false --keep_osymbols=false | \
   fstarcsort --sort_type=olabel > data/L.fst
```

这里，脚本make_lexicon_fst.pl 创建一个文本格式的FST。这个0.5是在句子的开头的静音概率(比如：在句子的开头和在每个词后，我们输出一个概率为0.5的静音；所以分配给没有静音的概率就是1.0 - 0.5 = 0.5。这个例子里命令的其他部分与把FST转换为编译的形式有关；fstarcsort 是很有必要的，因为我们稍后将组合它们。

词典的结构大约跟我们期待的一样。有一个最终的状态(环状态) 。存在具有二个转移成环状态的起始状态，一个是静音，一个没有静音。从环状态存在对应每个词的一个转移，词是在转移时的输出符号；输入符号是这个词的第一个音素。对于有效的组合和最小化来说，最重要的就是输出符号尽可能的早(i.e.在词的开始而不是末尾)。在每个词的末尾，来处理可选的静音，对于最后音素对应的转移有二种形式：一种对环状态和一种它具有与环状态转移的"silence state"。在静音词后，我们不必烦扰放置可选的静音，我们将定义这是一个静音音素的词。

创建一个带有消歧符号的词典是有点复杂的。问题就是我们不得不添加自环到词典里，以至于从G.fst 的消歧符号#0 可以被词典过滤。我们将通过程序fstaddselfloops来做(参考Adding and removing disambiguation symbols)，尽管我们可以很容易通过脚本make_lexicon_fst.pl自动完成。

```
phone_disambig_symbol=`grep \#0 data/phones_disambig.txt | awk '{print $2}'`
word_disambig_symbol=`grep \#0 data/words.txt | awk '{print $2}'`

scripts/make_lexicon_fst.pl data/lexicon_disambig.txt 0.5 SIL  | \
   fstcompile --isymbols=data/phones_disambig.txt --osymbols=data/words.txt \
   --keep_isymbols=false --keep_osymbols=false |   \
   fstaddselfloops  "echo $phone_disambig_symbol |" "echo $word_disambig_symbol |" | \
   fstarcsort --sort_type=olabel > data/L_disambig.fst
```

程序fstaddselfloops不是原始的 OpenFst 的命令行工具，是我们自己的工具(我们有许多这样的程序)。

Preparing the grammar G

语法G 是把词作为它的符号的接收器最重要的部分(i.e.输入和输出符号是在每个弧相同)。特殊的就是消歧符号#0 仅仅出现在输入这边。假设我们的输入是一个Arpa 文件，我们用kaldi程序arpa2fst 来转为为FST。程序arpa2fst的输出是一个有嵌入符号的FST。在kaldi中，我们一般使用没有嵌入符号的FSTs (i.e. 我们用分离的符号表)。除了仅仅运行arpa2fst，我们还需要做的步骤如下：

- 我们必须从FST中移除嵌入符号(和他们依赖于磁盘里的符号表)。
- 我们必须确保语言模型里没有词典以外的词
- 我们必须移除句子开始和结束读好的不合法的序列，比如 ~~followed by~~ ,因为这些导致 L o G 不是确定性。
- 我们必须在输入时用特殊的符号#0代替静音。

做这个工作的实际脚本的稍微简化版本如下：

```
gunzip -c data_prep/lm.arpa.gz | \
    scripts/find_arpa_oovs.pl data/words.txt  > data/oovs.txt

 gunzip -c data_prep/lm.arpa.gz | \
   grep -v '<s> <s>' | \
   grep -v '<s> </s>' | \
   grep -v '</s> </s>' | \
    ../src/lm/arpa2fst - | fstprint | \
   scripts/remove_oovs.pl data/oovs.txt | \
   scripts/eps2disambig.pl | \
    fstcompile --isymbols=data/words.txt --osymbols=data/words.txt \
    --keep_isymbols=false --keep_osymbols=false > data/G.fst
 fstisstochastic data/G.fst
```

最后一个命令(fstisstochastic)是一个诊断步骤(see Preserving stochasticity and testing it)。在一个典型的例子里，它将打印这些数字：

```
9.14233e-05 -0.259833
```

第一个数字是非常小的，所以它肯定在这个弧里没有一个状态的概率加上最后一个状态的概率小于1。第二次数字是有意义的，它意思就是存在有很大概率的状态(在FST里的权重的数值可以解释为log概率)。对于一个有反馈的语言模型的FST的表示，有大概率的一些状态是正常的。在后来的图建立步骤，我们将确认这些非随机性并没有比开始的时候变得更糟。

最后结果的FST G.fst 当然仅仅用在测试阶段。在训练阶段，我们使用从训练词序列得到的线性FST，但是这是在kaldi的内部做的，不是脚本做的。

Preparing LG

当组合L和G，我们坚持一个相对标准的脚本，比如我们计算min(det(L o G))。命令行如下：

```
    fsttablecompose data/L_disambig.fst data/G.fst | \
        fstdeterminizestar --use-log=true | \
        fstminimizeencoded  > somedir/LG.fst
```

这个与OpenFst's算法有些不一样。我们使用一个更加高效的组合算法(see Composition)的命令行工具"fsttablecompose"。我们的确定性也移除了静音的算法，有命令行fstdeterminizestar实现。选项–use-log=true是询问程序 是否先把FST投掷到log semiring；它保证随机性(在log semiring)；可以看Preserving stochasticity and testing it。 我们通过程序"fstminimizeencoded"做最小化。这个大部分跟运用到有权重的接收器的OpenFst's的最小化算法一样。仅仅的改变就是避免pushing weights，因为保证随机性(更多的看Minimization)。

Preparing CLG

为了得到一个输入是上下文相关的音素的转换器，我们需要准备一个叫CLG的FST, 它等于 C o L o G，这里的L和G 是词典和语法，C表示音素上下文。对于一个三音素系统，C的输入符号是a/b/c形式的(i.e. 音素的三元组), 和输出符号是一个单音素(e.g. a or b or c)。关于音素上下文窗可以看Phonetic context windows，和如何用到不同的上下文大小里。首先，我们描述如何来创建上下文FST C，如果我们去通过自身来做和正常的组合(为了有效和可扩展性，我们的脚本事实上不那么去做)。

Making the context transducer

这部分我们来解释如何让获得作为一个独自的FST C。 C的基本结构是它由所有可能音素窗大小N-1的状态(c.f. Phonetic context windows; N=3在三音素的情况里)。在第一个状态，意味着句子的开始，仅仅对应N-1静音。每一个状态有每个音素的转移(现在让我们忘记自环)。作为一个通用的例子，状态a/b在输出时有一个c的转移和输入是a/b/c，到状态b/c。在句子的输入和输出是特殊的情况。

在句子的开始，假设状态是/ 和输出符号是a。通常，输入符号是//a。但是这个不代表一个音素(假设P = 1)，中心元素是 ，它不是一个音素。在这种情况下，我们让弧的输入符号是一个特殊的符号#-1 ，这是我们介绍的主要原因。(作为标准的脚本，我们不使用静音这里，当有空词时会导致nondeterminizability)。

句子的结尾是有点复杂的。上下文FST在右边(它的输出边)会有一个在句子末尾的特殊的符号$。考虑三音素的情况。在句子的末尾，看完所有的符号后，我们需要过滤掉最后一个三音素(e.g. a/b/, where represents undefined context)。很自然的方式就是在输入是 a/b/ 和输出是之间有个传递，从状态a/b 到最后一个状态(e.g. b/或者一个特定的最终状态)。但是对于组合这不是有效的，因为如果它不是句子的末尾，我们不得不在删掉这些之前发现这些转移。我们在句子的末尾用符号$ 代替，和保证它出现在LG的每一条路径的末尾。然后我们在C的输出用

$代填。一般而言，$的重复数目是N-P-1。为了避免计算出有多少后续的符号添加到LG的麻烦，我们仅仅允许它在句子的末尾接受任何数量这样的符号。可以通过函数AddSubsequentialLoop()和命令行程序fstaddsubsequentialloop得到。

如果我们想要它自己的C，我们首先需要消歧符号的列表；和我们需要计算一个没有使用的符号id，这些符号我们将用在后续的符号里，如下：

```
grep '#' data/phones_disambig.txt | awk '{print $2}' > $dir/disambig_phones.list
 subseq_sym=`tail -1 data/phones_disambig.txt | awk '{print $2+1;}'`

We could then create C with the following command:
 fstmakecontextfst --read-disambig-syms=$dir/disambig_phones.list \
 --write-disambig-syms=$dir/disambig_ilabels.list data/phones.txt $subseq_sym \
   $dir/ilabels | fstarcsort --sort_type=olabel > $dir/C.fst
```

程序fstmakecontextfst需要音素的一个列表，消歧符号的一个列表和后续符号的标识。除了C.fst，它写出了解释C.fst左边的符号的"ilabels"文件(看The ilabel_info object)。LG的组合可以按下面的做：

```
fstaddsubsequentialloop $subseq_sym $dir/LG.fst | \
 fsttablecompose $dir/C.fst - > $dir/CLG.fst
```

用于打印C.fst 和使用相同的索引为"ilabels"的符号, 我们可以使用下面的命令来做一个合适的符号表：

```
 fstmakecontextsyms data/phones.txt $dir/ilabels > $dir/context_syms.txt
```

这个命令要知道"ilabels"格式(The ilabel_info object)。CLG fst的随机一条路径(for Resource Management)，打印它的符号表，接下来是：

```
## fstrandgen --select=log_prob $dir/CLG.fst | \
   fstprint --isymbols=$dir/context_syms.txt --osymbols=data/words.txt -
0   1   #-1 <eps>
1   2   <eps>/s/ax  SUPPLIES
2   3   s/ax/p  <eps>
3   4   ax/p/l  <eps>
4   5   p/l/ay  <eps>
5   6   l/ay/z  <eps>
6   7   ay/z/sil    <eps>
7   8   z/sil/<eps> <eps>
8
```

Composing with C dynamically

在正常的图建立脚本中，我们使用程序fstcomposecontext，可以动态的创建需要的状态和C的弧，而不不需要浪费的建立。命令行是：

```
fstcomposecontext  --read-disambig-syms=$dir/disambig_phones.list \
                   --write-disambig-syms=$dir/disambig_ilabels.list \
                   $dir/ilabels < $dir/LG.fst >$dir/CLG.fst
```

如果我们有不同的上下文参数N和P，相当于默认的(3 and 1)。我们将对这个程序使用其他的选项。这个程序写入文件"ilabels" (看The ilabel_info object)，可以解释为CLG.fst的输入符号。一个rm数据库的ilabels 文件的开始几行是：

```
65028 [ ]
[ 0 ]
[ -49 ]
[ -50 ]
[ -51 ]
[ 0 1 0 ]
[ 0 1 1 ]
[ 0 1 2 ]
...
```

65028是文件里元素的个数。像[-49] 行是为了消歧符号；像[0 1 2] 行代表声学上下文窗；一开始的2个entries是为了静音(从来不使用)的[ ]和and特定的消歧符号[0]，它的打印格式为#-1，这是在C的开始，为了替代静音，为了确保确定性。

Reducing the number of context-dependent input symbols

当创建CLG.fst后，就有一个减小其大小的图建立选项。,我们使用程序make-ilabel-transducer，它可以形成决策树和得到HMM拓扑信息，上下文相关音素的子集相对于相同的图编译和将合并(我们选择每个自己的任意元素和把所有的上下文窗转为为这个上下文窗)。这个跟HTK's logical-to-physical mapping相似。命令行是：

```
make-ilabel-transducer --write-disambig-syms=$dir/disambig_ilabels_remapped.list \
 $dir/ilabels $tree $model $dir/ilabels.remapped > $dir/ilabel_map.fst
```

这个程序需要tree和model；它的输入是一个新的ilabel_info 对象，称为"ilabels.remapped"；这个跟原始的"ilabels"文件有相同的格式，但是有很少的行。FST "ilabel_map.fst" 是由CLG.fst 构成的和重新映射标签的。当我们做了这个后，我们将确定性和最小化，所以我们可以马上实现任意大小的减少：

```
fstcompose $dir/ilabel_map.fst $dir/CLG.fst  | \
  fstdeterminizestar --use-log=true | \
  fstminimizeencoded > $dir/CLG2.fst
```

这个阶段为了典型的建立事实上我们不会减少图大小很多(一般减少5% to 20%)，和在任何情况下，它是中间图建立阶段的大小，我们通过这个机制来减少。但是这个减少对宽上下文的系统是有意义的。

Making the H transducer

在传统的FST脚本里，H转换器是有它的上下相关音素的输出和输入是代表声学状态的符号。在我们的情况下，H (or HCLG)的输入的符号不是声学状态(在我们的术语里是pdf-id)，但是我们用transition-id来代替(看Integer identifiers used by TransitionModel)。transition-id是对pdf-id加上包含音素的一些其他的信息的编码。每一个transition-id可以映射一个pdf-id。我们创建的H转换器没有把自环编码进来。他们稍后将通过一个单独的程序添加进来。H转换器 具有开始和最终的状态，和从这个状态的每个entry有一个转移除了在ilabel_info 对象(the ilabels file, see above)里的第0个。上下文相关音素的转移到相对应的HMM(缺少自环)的结构中，和然后到开始的状态。对于正常的拓扑结构，HMM的这些结构仅仅是三个弧的线性序列。每一个消歧符号(#-1, #0, #1, #2, #3 等等)的开始状态，H都有自环。 脚本的这个部分是生成H 转换器(我们称为Ha 因为这里缺少自环)是：

```
make-h-transducer --disambig-syms-out=$dir/disambig_tstate.list \
   --transition-scale=1.0  $dir/ilabels.remapped \
   $tree $model  > $dir/Ha.fst
```

这是一个设置转移尺度的选项；在我们现在训练的脚本里，这个尺度是1.0。这个尺度仅仅影响那些与自环概率无关的转移部分。和正常的拓扑(Bakis model)没有任何影响；更多的解释可以看Scaling of transition and acoustic probabilities。除了FST，程序也写消歧符号的列表，这些符号稍后将被删除。

Making HCLG

生成最后一个图HCLG的第一部就是生成缺少自环的HCLG。命令行是

```
  fsttablecompose $dir/Ha.fst $dir/CLG2.fst | \
   fstdeterminizestar --use-log=true | \
   fstrmsymbols $dir/disambig_tstate.list | \
   fstrmepslocal  | fstminimizeencoded > $dir/HCLGa.fst
```

这里，CLG2.fst是一个减少符号集版本的CLG ("logical" triphones, in HTK terminology)。我们删除了消歧符号和任何容易删除的静音(看Removing epsilons), 在最小化之前；我们最小化算法是避免pushing symbols and weights (因此保证随机性), 和接受非确定行的输入(看Minimization)。

Adding self-loops to HCLG

添加自环到HCLG 是用下面的命令完成的：

```
add-self-loops --self-loop-scale=0.1 \
  --reorder=true $model < $dir/HCLGa.fst > $dir/HCLG.fst
```

怎么把self-loop-scale为0.1使用上的更多细节可以看Scaling of transition and acoustic probabilities(注意这会影响non-self-loop概率)。"reorder"选项的更多解释，可以看Reordering transitions；"reorder"选项增加了解码的速度，但是它与kaldi的解码不兼容。add-self-loops程序不是仅仅添加自环，它也许会复制状态和添加静音转移来确保自环可以以一致的方式加入。这个事情的更多细节可以看Reordering transitions。这是图建立中唯一一步不需要保证随机性；它不需要保证它，是因为self-loop-scale不为1。所以程序fstisstochastic可以给所有的G.fst, LG.fst, CLG.fst和HCLGa.fst相同的输出，但是不能是HCLG.fst。在add-self-loops之后，我们不需要再做确定性；这个会无效，因为我们已经移除了消歧符号。所以，这个会很慢，和我们相信没有什么可以从确定性和最小化那里得到。

解码图创建示例（训练阶段）

在训练阶段的图建立脚本是比在测试阶段的简单，主要是因为我们没有不需要消歧符号。

我们假设你已经读过了测试阶段的recipe。

在训练阶段我们用测试阶段相同的HCLG形式，除了G是相当于训练脚本构成的一个线性接受器(当然，这个建立是很容易扩展到在那些脚本的不确定)。

Command-line programs involved in decoding-graph creation

这里我们将解释在训练脚本时发生了什么；下面我们在看程序里发生了什么。假设我们已经建立了一个树和一个model。接下里的命令建立一个包含对应每个训练transcripts的图HCLG(c.f. The Table concept)。

```
compile-train-graphs $dir/tree $dir/1.mdl data/L.fst ark:data/train.tra \
  ark:$dir/graphs.fsts
```

输入文件train.tra 是一个包含训练 transcripts的整数版本，比如典型的命令行就是：

```
011c0201 110906 96419 79214 110906 52026 55810 82385 79214 51250 106907 111943 99519 7922
```

第一个token就是utterance id。程序的输出是一个archive graphs.fsts；它包含在train.tra的每一个句子的一个FST (in binary form)。对应HCLG的这个FST,除了没有转移概率(默认的, compile-train-graphs有–self-loop-scale=0和–transition-scale=0)。这是因为这些图是用来多阶段训练和转移概率将被改变，所以我们在后面加入他们。但是FSTs将有从静音概率(这些概率被编码在L.fst),上升的概率，和如果我们使用发音概率，这些也将被显示出来。 一个读取这些archives和对训练数据解码的命令是，接下来将产生state-level alignments (c.f. Alignments in Kaldi)；我们将简单的惠顾这个命令，尽管这页我们的焦点是图建立本身。

```
gmm-align-compiled \
 --transition-scale=1.0 --acoustic-scale=0.1 --self-loop-scale=0.1 \
  --beam=8 --retry-beam=40 \   $dir/$x.mdl ark:$dir/graphs.fsts \
  "ark:add-deltas --print-args=false scp:data/train.scp ark:- |" \
   ark:$dir/cur.ali
```

前三个参数就是概率尺度(c.f. Scaling of transition and acoustic probabilities)。transition-scale 和self-loop-scale 选择在这里的原因是在解码之前，程序需要添加转移概率到图上(c.f. Adding transition probabilities to FSTs)。接下来的选项是beams；我们使用一个初始化的beam，和然后如果对齐没能达到最后的状态，我们使用其他的beams。因为我们使用一个声学的尺度0.1，相对于声学似然比，我们将不得不对这些beams乘以10来得到figures。程序需要model；通过迭代，$x.mdl 可以扩展为1.mdl 或者2.mdl。它通过archive (graphs.fsts)来读图。在引号里的参数被认为是一个管道(minus the "ark:" and "|")，和被解释成utterance-id的archive索引的，包含特征。输出就是cur.ali，和如果写成txt格式，它看来像是上面描述的.tra文件，尽管整数现在对应的不是word-ids而是transition-ids (c.f. Integer identifiers used by TransitionModel)。

我们注意到这两个阶段(graph creation and decoding)可以通过一个命令行程序完成：gmm-align，可以编译你想要的图。因为图建立需要花费相对长的时间，当把他们写到磁盘里，任何图都需要不止一次写入。

Internals of graph creation

图建立都是在训练时间里完成的，无论是程序compile-train-graphs还是gmm-align，都是有相同的代码来完成的： 类TrainingGraphCompiler。当一个接着一个编译图，他们的行为如下：

在初始化：

- 添加子序列自环(c.f. Making the context transducer) 到词典L.fst，确保按输出标签排序和存储。

当编译一个transcript, 它执行以下的步骤：

- 生成一个相对于词序列的线性接收器
- 和词典一起编译它 (using TableCompose) 得到一个含有输入是音素和输出是词的FST (LG) ；这个FST包含发音和静音的选项。
- Create a new context FST "C" that will be expanded on demand.
- Compose C with LG to get CLG; the composition uses a special Matcher that expands C on demand.
- Call the function GetHTransducer to get the transducer H (this covers just the context-dependent phones that were seen).
- Compose H with CLG to get HCLG (this will have transition-ids on the input and words on the output, but no self-loops)
- Determinize HCLG with the function DeterminizeStarInLog; this casts to the log semiring (to preserve stochasticity) before determinizing with epsilon removal.
- Minimize HCLG with MinimizeEncoded (this does transducer minimization without weight-pushing, to preserve stochasticity).
- 添加自环。

TrainingGraphCompiler类有一个函数CompileGraphs()，这个函数是在一个batch里联合一些图。这个被用在工具compile-train-graphs来加速图的编译。主要的原因是可以帮助在一个batch里，当创建H转换器时，我们仅仅需要处理每一个看到的上下文窗，即使它被用在许多的CLG例子里。 没用第一个添加消歧符号做确定性的原因是，在这个情况下HCLG是函数(因为任何一个输入标签序列转换为同一个string)和非周期的；任何一个非周期的FST有两个特性，和任何函数性的有两个特性的FST是确定性的。

有限状态转换器算法

Kaldi套件中使用的解码器

Kaldi中的网格

翻译：izy

时间：2015年7月

Introduction

网格(lattice)是某个特定语句的最可能的可选词序列的表示。为了更好地理解网格，你应该了解 WFST框架下的解码图（见 Decoding graph construction in Kaldi）。本节中我们概述Kaldi网格的相关问题，并在本页后面的部分详细地解释它们。读者可能也想读这一篇文献“Generating exact lattices in the WFST framework”, D.Povey, M.Hannemann et.al, ICASSP 2012 (submitted)，其中给出了更多网格是如何与以前的算法相联系的内容。本页更侧重于编程的方面。

Kaldi 中的网格有两种表示。第一种是 Lattice。这是一个FST，权重包含两个浮点权值（图代价(the graph cost)和声学代价(the acoustic cost)），输入符号是 transition-ids （近似于上下文无关的 HMM状态），输出符号是词（一般来讲，它们表示解码图的输出符号）。

第二种是 CompactLattice。它本质上包含了和 Lattice相同的信息，但是形式不同：它是一个acceptor（意味着输出输出符号始终相同），输出输出符号代表词，而 transition-ids序列作为了权重的一部分（注意 OpenFst有一个很通用的权重概念；满足半环公理的都可以作为权值）。CompactLattice 中的权重包括一对浮点数和一个整数序列（代表 transition-ids）。我们会在后面描述半环。

因为 Lattice和 CompactLattice都代表同样的信息，它们在 I/O意义上是等价的，即读 Lattice的代码也可以读 CompactLattice，反之亦然。举例来说，如果用 `SequentialLatticeReader` 来读由 utterance-id索引的 lattice集合，不论 archive中实际上是 Lattice还是 CompactLattice，处理上都是一样的。然而，我们通常把 lattices写出为 CompactLattice。对于需要在两者之间进行转换的程序，可以使用 ConvertLattice()。

Lattice和 CompactLattice类型中的权重包括两个浮点数，即 graph cost和 acoustic cost。graph cost是 LM代价，（加权）转移概率和任何发音代价的和。它们被混在了一起，所以在做 LM重打分时，我们先要减去旧的 LM代价再加上新的 LM代价。如果我们把这两个浮点数叫做 (a,b)，Lattice中的半环像字典半环 (a+b,a-b)（见 OpenFst文档中的解释）。 也就是说，当我们“加”（在半环中）这样的两对儿数时，我们得到具有最低 (graph + acoustic) cost的一对，如果它们相等，选用 (graph - acoustic)来打破平局。这在“取最优路径”的意义上是合理的。实际上，它只有在 acoustic weights适当缩放时才是合理的。我们的惯例是在外部存储（e.g. 磁盘）时，acoustic costs不缩放，在算法比较权重时应用缩放（在写出 lattice时再缩放回来）。

Lattice 和 CompactLattice类型被当做数据结构来表示传统的网格，也用于表示 N-best列表。生成网格的算法有很精确的特性。设定 lattice-delta为一个数值（通常为10左右）。网格生成算法可保证（modulo beam pruning）每个词序列的最优对齐的代价，在当前网格中最优路径的 lattice-delta范围内，而且是具有最可能的对齐和权值。同时，每个词序列只在网格中出现一次。要达到这一点，首先创建一个状态级的网格然后适当剪枝(pruning)，然后用一个特殊的半环 determinize（或者，如果你愿意，可以认为这是一个总是选择最可能路径的 non-functional FSTs的特殊 determinization算法）。下面我们会解释这是如何实现的。

我们始终确保写出到磁盘的网格，每个词序列只有一条路径。这对某些算法（如 LM重打分）是必要的。Being determinized (on the word labels, in the CompactLattice format) is a sufficient condition for this.

我们从不在 Kaldi代码中附加符号表给 FSTs。为了查看包含实际词的最自然的形式的网格的 archives，我们用脚本来完成这一转换（例子见 egs/rm/s1/steps/decode_tri1_latgen.sh）

The Lattice type

Lattice类型只是一个基于特定半环模板化的 FST。在 kaldi-lattice.h，我们创建一个 typedef:

```
typedef fst::VectorFst<LatticeArc> Lattice;
```

这里 LatticeArc是基于 LatticeWeight模板化的 arc类型：

```
typedef fst::ArcTpl<LatticeWeight> LatticeArc
```

LatticeWeight又是 LatticeWeightTpl模板在 BaseFloat浮点类型的一个实例化。

```
typedef fst::LatticeWeightTpl<BaseFloat> LatticeWeight;
```

LatticeWeightTpl模板是文件 fstext/lattice-weight.h中的命名空间 fst中定义的。它和字典半环有一些相似处，i.e. 它类似于 OpenFst类型。

```
LexicographicWeight<TropicalWeight, TropicalWeight>
```

两种情况下，加法（带运算的半环）定义为取最大值，但是“最大值”的定义不同。LexicographicWeight 首先比较第一个元素并用第二个来打破平局。LatticeWeight首先比较和值；用差值打破平局。所以一个 LatticeWeight (a,b)和一个 LexigraphicWeight (a+b,a-b)等价。LatticeWeight的意图是取代价最小的（总代价是 graph + acoustic cost），同时分别记住

acoustic and graph cost。在理想设定中，我们可能想保留更多的信息：例如，LM cost和transition-model cost和 pronumciation-probability cost。但这不切实际，因为这些信息在解码图中混在了一起，在解码图中把它们分开会导致解码图尺寸的显著扩张。

上面提到了，Lattice的输入符号表示 transition-ids，输出符号表示 words（或者解码图输出的任何符号）。

设计 Kaldi时，我们考虑过用 LexicographicWeight类型替代 LatticeWeight，第一个元素是(graph + acoustic) cost第二个元素是 acoustic cost。最终没有这样做，因为尽管它在某些时候更有效，我们觉得它定义的有点混乱。

网格上的很多算法（如取最优路径，或剪枝）用 Lattice类型比 CompactLattice类型效率更高。这是因为 CompactLattice类型中的权重包括了 transition-ids序列，像取最优路径的操作，会将权重相乘，对应着序列的相加。对很多算法来说，时间复杂度是词网格长度的平方。上面提到，可以读 archive中的 Lattice，即便它实际上是 CompactLattice，因为 Holder类型（LatticeHolder）会自动转换。所以代码：

```
SequentialLatticeReader lattice_reader(lats_rspecifier);
for (; !lattice_reader.Done(); lattice_reader.Next()) {
   std::string key = lattice_reader.Key();
   Lattice lat = lattice_reader.Value();
   ...
}
```

是合法的即使“lats_rspecifier”对应的 archive或 script file包含的是 CompactLattice格式-通常会是这样，因为网格都是以这种格式写出的。网格类型转换的例子：

```
Lattice lat;
// initialize lat.
CompactLattice compact_lat;
ConvertLattice(lat, &compact_lat);
```

转变为 CompactLattice类型包含一个“factoring”操作，函数 fst::Factor（定义在 fstext/facor.h），标识可以组合成一个 CompactLattice arc的状态链。把 OpenFst算法用于网格的一个典型例子如下（代码改动自 latbin/lattice-best-path.cc，寻找网格中的最优路径）：

```
Lattice lat;
// initialize lat.
Lattice best_path;
fst::ShortestPath(lat, &best_path);
```

Compact lattices (the CompactLattice type)

和 Lattice类型类似，CompactLattice是一个典型 OpenFst模板的一个 typedef:

```
typedef fst::VectorFst<CompactLatticeArc> CompactLattice;
```

CompactLatticeArc 类型定义如下：

```
typedef fst::ArcTpl<CompactLatticeWeight> CompactLatticeArc;
```

CompactLatticeWeight 类型定义如下：

```
typedef fst::CompactLatticeWeightTpl<LatticeWeight, int32> CompactLatticeWeight;
```

模板参数是基础权重（LatticeWeight）和一个储存 transition-ids的整数类型（int32）。注意在 Kaldi代码中（比如上面），我们倾向于定义复合类型，然而在 OpenFst中是定义模板，当我们在命名空间 fst内写代码时，我们会遵循这一惯例。所以，我们在命名空间 fst定义了一个模板，然后把它变成命名空间 kaldi中的一个特定的类型(CompactLatticeWeight)。

CompactLatticeWeightTpl的模板变量是基础权重（LatticeWeight）和整数类型（int32）。它包含两个数据成员：一个权值和一个整数序列：

```
template<class WeightType, class IntType>
class CompactLatticeWeightTpl {
   ...
 private:
  WeightType weight_;
  vector<IntType> string_;
};
```

它们可以通过成员函数 Weight()和 String()来访问。据我们所知，CompactLatticeWeightTpl用到的半环和 OpenFst中的并不对应。乘法对应于权值相乘和字符串相连。加两个 CompactLatticeWeghts时，我们首先比较权值，选取其中权值大的和它对应的字符串。如果权值相等，我么用字符串的字符顺序来打破平局。这个字符顺序的规则是，如果长度不同选短的，否则用字典序（我们不能只用字典序作规则，否则会与半环公理中的乘法分配律冲突）

我们对权重比较大小或判断相等，前提是半环上是有次序的，这是模板权重的一个要求（LatticeWeight和 TropicalWeight是满足了的）。OpenFst访问这个次序的方式是通过模板"NaturelLess"，它对给定的 a, b，判断 (a+b)是否与 a或 b相等。为了提高效率，我们根据 weight类型定义了一个 `Compare()` 函数，返回值是-1,0 or 1。我们为 LatticeWeightTpl模板定义了这样的函数。同时为了测试的目的，我们也给 CompactLatticeWeightTpl在 TropicalWeight的实例定义了合适的 Compare函数。

我们定义这样一个看起来很奇怪的半环是有原因的，这和我们的网格概念有关。网格包含了对任意给定的词序列的最优路径。假定有一个网格（e.g. 可能是一个 raw state-level lattice），我们用 CompactLattice表示，即一个词序列的 acceptor，同时权重包含 transition-id序列。WFSTs的含义是对任意的（输入符号序列，输出符号序列）对，WFST分配的权重是所有包含该对符号序列的路径的权重的和。目前的情况是，我们处理的 acceptor只有符号序列（词序列）。当我们在半环中对（weight,string）求和时，我们得到的是最优的（即最小代价）的权重对。如果我们要在半环中 remove epsilons and determinize,我们会得到一个对每个词序列只有一个路径的网格，并且这个路径是最优的。这样效率不高，所以实际上我们有一个专门的 determinization算法来 remove epsilon，同时它的作用和调用 OpenFst 的 `RemoveEps()` 和 `Determinize()` 是一样的。

Getting the raw lattice, and converting it into the final form

目前，生成网格的唯一的解码器是定义在 decoder/lattice-simple-decoder.h中的类 LatticeSimpleDecoder，它被 gmm-latgen-simple.cc调用。顾名思义，LatticeSimpleDecoder 是由 SimpleDecoder修改得到的。SimpleDecoder是 Viterbi beam search算法的一个直接实现，只有一个可调参数：the pruning beam（见 SimpleDecoder:the simplest possible decoder）。LatticeSimpleDecoder有一个更重要的可调参数：the lattice beam (or lattice-delta)，一般应该比 the pruning beam 小。基本框架是，LatticeSimpleDecoder先生成一个状态级的网格，然后用 lattice-delta剪枝，最后执行 determinization算法，对每个词序列仅保留最优路径。

在 SimpleDecoder中，有引用计数的回溯(reference-counted tracebacks)。在 LatticeSimpleDecoder中，单个回溯是不够的，因为网格具有更复杂的结构。实际上，存储前向链接比后向链接更为方便。为了释放 lattice-delta pruning时不需要的链接，我们需要做的比引用计数更多；实际上也没做引用计数。

The final objective of the pruning algorithm is as follows (call this the "naive" approach. Firstly, suppose that for each frame, for each state that was active on that frame we created a structure or token of some kind, and for each arc out of it (to a state that was not pruned), we create some kind of link record. Let's use the word "token" for the structure that we create for a particular graph state at a particular frame, and "link" for the forward link from one token to another token (note: these links can be within a frame as well as across frames, due to epsilon arcs in the decoding graph). The simplest possible version of the pruning algorithm is: firstly, we decode up till the end of the file, keeping all the tokens and links that were active during decoding. When we have come to the end of the file we want to apply the lattice-delta beam, such that any state or arc that is not on a path that is within lattice-delta cost of the best path, will be pruned away. This is fairly easy to do: for example, we could turn the state-level traceback into an FST and use OpenFst's Prune() algorithm.

Unfortunately this wouldn't be a very practical algorithm because we would soon run out of memory: on each frame there are typically many thousands of states active, and we can't afford to wait till the end of the utterance to prune them.

Fortunately there is an efficient way to prune away these tokens without waiting until the end of the utterance. Periodically (every 25 frames by default), we do a backward sweep from the current frame to the beginning. In this backward sweep, we work out a "delta" quantity for each token. The "delta" is the difference between the cost of the best path and the cost of the best path the token is on, and it can be computed in the backward sweep (the "forward" information that we need it already inside the token, and is static). We can define an analogous "delta" quantity for the forward links. Any time this "delta" exceeds the "lattice-delta" beam, we prune away the associated token or link. For the tokens at the currently active frame, we set the "delta" to zero; we can prove that this is the right thing to do. That is, with this algorithm we will only be pruning away things that we eventually would have pruned away anyway using the "naive" algorithm.

It might seem on the face of it that our pruning algorithm would take time quadratic in the length of the utterance (because each 25 frames we visit the whole utterance up to the current point), but in fact it is more like linear, because we can detect when all the "delta" values of tokens for a particular frame have not changed, and we can stop going backward in time. We expect that for very long utterances, we will normally prune backward for only a second or two. Even if it were not for this feature, the algorithm would be dominated by a linear-time component in practical cases, because the vast majority of tokens are deleted the first time they are visited by the algorithm.

We note that there are a couple of complexities in our implementation of this. One is that because we have forward links, while the pruning algorithm goes backward in time, there is a danger of "dangling" forward links being created when we delete() tokens, so we have to avoid delete()ing tokens until we have pruned the forward links from the previous frame. The other issue is that we have to treat the final frame specially, because we need to take into account the final probabilities.

Getting the raw lattice, and converting it into the final form

When we have reached the end of the utterance, we produce the lattice in two phases. The first phase is to create the "raw lattice". This is a state-level FST that is pruned using the lattice-delta but is not determinized, so it has many alternative paths corresponding to each word sequence. Creation of the "raw lattice" is very trivial, and consists of taking our token and link structures and translating them into OpenFst structures. If we are asked for the best path (e.g. for getting the transcript), we would just run OpenFst's ShortestPath algorithm on this raw lattice. If we are asked for the lattice, there are two options.

The first option is: the decoder supports outputting the raw state-level lattice directly (set –determinize-lattice=false); this is mostly useful if you are going to do acoustic rescoring with a system built with the same tree. Afterwards you would probably want to determinize the lattice using the lattice-determinize program, before storing the lattices on disk for an extended time (as the raw state-level lattices are quite large).

Let's assume instead that we're outputting the determinized lattice (the second option). The determinized lattice is supposed to have only one path through it for every word sequence that is allowed in the original raw lattice. We can acheive this by creating a CompactLattice, removing epsilons and determinizing, but this would be very inefficient. Instead, we run a special determinization algorithm called DeterminizeLattice(). This algorithm does epsilon removal and determinization together. In this respect it is similar to our algorithm DeterminizeStar(), but the DeterminizeLattice() algorithm uses data structures that are specifically optimized for this case. The issue is the cost overhead of appending long strings of input symbols together (the sequences of symbols are only about as long as the maximum number of frames in a word, but this is quite long). That is, if represented in the most naive way as vectors, adding a symbol to a list of N symbols and storing the result separately take time O(N). Instead we use data structures that make it take constant time. DeterminizeLattice() takes as input the Lattice format, since this is the most natural form for the input, and outputs in the CompactLattice format. However, it should really be viewed as determinization of the CompactLattice format: that is, it preserves equivalence only when viewing it as an operation on a CompactLattice.

Lattices in archives

我们一般用 archives来存储网格。关于 archive和 Kaldi I/O机制的信息，见 Kaldi I/O mechanisms。用命令行生成网格的一个具体例子如下：

```
gmm-latgen-simple --beam=13.0 --acoustic-scale=0.0625 exp/tri/1.mdl \
     exp/graph_tri/HCLG.fst ark:test_feats.ark "ark,t:|gzip -c > exp/lats_tri/1.lats.gz"
```

这之后，我们可以看到如下的网格：

```
gunzip -c exp/decode_tri2a_bg_latgen_eval92/7.lats.gz | \
  scripts/int2sym.pl --field 3 data/words.txt | head
444c0410
0 1 <s> 5.27832,0,
1 2 <UNK> 8.08116,1099.84,
1 3 A 12.8737,8342.93,3_1_1_1_1_1_1_1_1_1_12_18_17_17_17_17_17_17_17_17_3\
464_3596_3632_3_1_1_1_1_1_1_1_1_1_1_1_1_1_1_1_1_1_1_1_1_1_1_1_1_1_1_1_\
1_1_1_1_1_1_1_1_1_1_1_1_1_1_1_1_1_1_1_1_1_1_1_1_1_12_18_17_17_18562_18561_18604_18\
603_18603_18618_604_603_603_638
1 4 IF 10.2262,8096.64,3_1_1_1_1_1_1_1_1_1_12_18_17_17_17_17_17_13708_137\
28_13806_12654_12670_12676_3_1_1_1_1_1_1_1_1_1_1_1_1_1_1_1_1_1_1_1_1_1_1_1_1\
_1_1_1_1_1_1_1_1_1_1_1_1_1_1_1_1_1_1_1_1_1_1_1_1_1_1_1_1_1_1_1_12_18_17_17_17_1856\
2 5 NOT 12.4568,10548.7,3_1_1_1_1_1_1_1_1_1_12_18_17_17_17_17_17_17_20_26\
...
```

标签 `<UNK>` 具有 graph和 acoustic scores但是没有输入标签（如果有，会在最后一个逗号后面出现），在这里似乎不太合适。必须理解的是，graph/acoustic scores和输入序列只有在通过 FST的完整路径上叠加（或连接）后才是有效的。并不需要保证，它们彼此之间对齐或和词标签对齐。

Lattices通常以 CompactLattice的形式存储在 archive，而且惯例是 acoustic weights不采用缩放，所以对于对 acoustic weight敏感的运算（如剪枝），对应的命令行会有 -acoustic-scale 选项，并在进行运算前缩放 acoustic weights（运算结束后缩放回来）。

Operations on lattices

下面我们讨论了一些网格上的运算操作和它们对应的程序。

Pruning lattices

网格可以用一个设定的 pruning beam来剪枝。这会去掉和网格中最优路径的代价相差不够小的那部分 arcs和 states。对每个网格，`lattice-prune` 首先根据设定的尺度缩放 acoustic weight，然后调用 OpenFst的 `Prune()` 函数，最后对 acoustic weight进行逆缩放。一个命令行示例：

```
lattice-prune --acoustic-scale=0.1 --beam=5 ark:in.lats ark:out.lats
```

注意：为了提高效率，剪枝运算是在 Lattice格式下完成的，但是会在写出时转换为 CompactLattice。

Computing the best path through a lattice

`lattice-best-path` 计算通过网格的最优路径，并输出输入符号序列（alignment）和输出符号序列（transcription）。通常输入和输出都是 archive。一个命令行示例：

```
lattice-best-path --acoustic-scale=0.1 ark:in.lats ark:out.tra ark:out.ali
```

Computing the N-best hypotheses

`lattice-nbest` 计算通过网格的 N-best路径（利用 OpenFst的 `ShortestPath()` 函数），输出是一个有特定结构的网格（a CompactLattice）。正如在 OpenFst的 `ShortestPath()` 的文档中说明的，开始状态会有（至多）n个 arcs，每一个都是一条单独路径的开始。但是这些路径可以共享后缀。一个命令行示例：

```
lattice-nbest --n=10 --acoustic-scale=0.1 ark:in.lats ark:out.nbest
```

Language model rescoring

因为 LatticeWeight的“graph part”（第一部分）包含了语言模型得分和转换模型(transition-model)得分，所有的发音或静音概率，我们不能只用新的语言模型得分替换它，否则会丢失后两个。所以，我们要先减去旧的 LM概率再加上新的 LM概率。这两个阶段的核心操作是组合（还有一些权值的缩放，等等）。相应的命令行是：首先，一处旧的 LM概率

```
lattice-lmrescore --lm-scale=-1.0 ark:in.lats G_old.fst ark:nolm.lats
```

然后加上新的 LM概率

```
lattice-lmrescore --lm-scale=1.0 ark:nolm.lats G_new.fst ark:out.lats
```

也有其他的方法可以做到这一点，见下面 `lattice-compose` 的文档。要解释程序 lattice-lmrescore做了些什么，我们先描述一个简化的版本。假设给定 LM-scale S和 LM G.fst。我们首先把 G.fst中的 costs都乘以 S。然后对每个网格，把它组合到 G的右边，进行 lattice-determinize（仅保留每个词序列的最优路径），然后写出。如果 S是正值，这没有问题，但如果 S是负值，这相当于对每个词序列取了 G的最差路径。为了解决这一问题，我们采用如下做法。对每个输入的网格，首先用 S的逆缩放网格的 graph (or LM) costs；然后组合到 G.fst的右边；执行 lattice-determinization，保留每个词序列的最优路径；然后用 S缩放网格的 graph/LM scores。这样 S为负值时也是对的。这个方法只有在输入网格的每个词序列都只有一条路径时（e.g. 它经过了 lattice determinization）才有效。我们假定所有由程序处理的网格都有这个特点（所以直接写 “raw” state-level网格不是一个好主意，除非你知道你要做什么）。

注意为了让组合生效，程序需要把 G.fst从 tropical semiring映射到 LatticeWeight semiring。这通过把 G.fst的权值赋给权重的第一部分（the "graph" part），然后权重的第二部分置零（在半环中，它是1）。在 C++层，这个映射通过 OpenFst的 MapFst完成，其中定义了一个 Mapper类来映射 StdArc到 LatticeArc，然后根据需求创建一个 MapFst类型的对象，完成 G.fst到 LatticeWeight的权重部分的转换。

Probability scaling

可以通过一个 2x2的矩阵来对网格权重的概率进行比例缩放。相应的命令行程序是 `lattice-scale` 。一个示例：

```
lattice-scale --acoustic-scale=0.1 --lm-scale=0.8 ark:in.lats ark:out.lats
```

它实际上不会被经常使用，因为我们希望 archive中的 acoustic weights是没有经过缩放的；需要进行缩放的程序可以接受一个 `-acoustic-scale` 选项。这些程序会应用缩放，执行运算（比如 determinization or pruning），然后逆缩放。对大多数操作，我们不会缩放 LM概率；然而，它有时也会被用到，e.g. 在区分性训练(discriminative training)时同时调整 LM和 acoustic尺度可能是有好处的。 `lattice-scale` 程序接受 `-lm2acoustic-scale` 和 `-acoustic2lm-scale` 选项。例如，程序会把 `-lm2acoustic-scale` 乘以权重的 LM部分然后加到新权重的 acoustic部分。我们引入这些主要是考虑到完整性。

Lattice union

`lattice-union` 计算两个网格的 union（和其他程序一样，程序遍历 archive中的网格集合）。主要设想的用途是在区分性训练中（特别是 MMI），为了保证分母网格中的 transcription是正确的。一个命令行示例是：

```
lattice-union ark:num_lats.ark ark:den_lats.ark ark:augmented_den_lats.ark
```

程序调用 OpenFst的 `Union()` 函数，进行 lattice-determinization来保证每个词序列仅有一条路径，然后写出。实际上 lattice union发挥作用的场景并不多（e.g. 在由不同特征或模型计算得到的网格上进行 union并没有什么意义）

Lattice composition

`lattice-compose` 有多种工作模式。一种是组合网格。这是在转换器(transducer)形式下完成的，i.e.把网格看做 transition-ids到 words的一个转换器。这一般会在映射一个网格集合到输出（来获得词到词的转换器）后完成。典型的用法是：

```
lattice-project ark:1.lats ark:-  | \
  lattice-compose ark:- ark:2.lats ark:3.lats
```

现在，3.lats的路径会具有 1.lats和 2.lats中得分的和。你可以用 `lattice-interp` （见下一节）来一步完成这一目的。

另一种模式是组合网格和一个固定的 FST。为了这个目的，FST被动态地转换为网格；FST的权重解释为网格权重的“graph part”。一个例子是，把网格转换成音素，再和字典组合转换回词，例如：

```
lattice-to-phone-lattice final.mdl ark:1.lats ark:- | \
  lattice-compose ark:- Ldet.fst ark:words.lats
```

这是一个简化了的例子；你可以进一步处理，去除网格中原来的“graph scores”然后加上grammar和 transition scores。同时，Ldet.fst是一个专门处理了的词典：带消歧符号的词典（但是没有 #0到 #0的环，来通过语言模型消歧符号），determinized and with the disambiguation symbols then removed.

上面的模式对语言模型重打分也很有用；然而它也有不同的版本，如下所示（这对把 LM得分加到网格里时很有用，假定我们已经移除了旧的 LM得分）：

```
lattice-compose --phi-label=13461 ark:1.lats G.fst ark:2.lats
```

这里，G.fst是在输入端包含特殊的“回退符号”(backoff symbol) #0的语法，13461是这个回退符的数字标号。这种组合形式把回退符看做一个失败的 transition（利用 OpenFst's PhiMatcher），当请求的标签不存在时被采用：所以 G.fst被看做一个回退语言模型。

Lattice interpolation

`lattice-interp` 根据一个缩放参数，对两个网格做内插。例子：

```
lattice-interp --alpha=0.4 ark:1.lats ark:2.lats ark:3.lats
```

意思是在 1.lats的得分上乘以0.4，在 2.lats上乘以0.6。它会维持第一个 archive中的对齐信息，如果有的话。如果两个网格没有共同路径，组合会为空，程序就不会输出任何网格，所以最后的 archive中可能有“gaps”。你需要建立某种机制来为这些 utterances构造输出，到解码脚本中（e.g. 见 egs/rm/s3/steps/decode_combine.sh）。这个程序的功能可被模拟为 `lattice-scale` , `lattice-project` , `lattice-compose` 的组合。

Conversion of lattices to phones

`lattice-to-phones-lattice` 在网格的输出端删除词标签，替换为音素标签。这些是通过输入的 transition-ids获得的。注意音素标签并不是和音素边界对齐的（FSTs 在输入和输出符号之间没有对齐的概念）。典型用法是：

```
lattice-to-phones final.mdl ark:1.lats ark:phones.lats
```

Lattice projection

Projection 是把 FST转变为 acceptor的一个 FST操作，在输入和输出符号间进行拷贝，让它们一致。 `lattice-project` 默认是把词标签拷贝至输入端（这在做网格内插时很有用）。例子是：

```
lattice-project ark:1.lats ark:- | \
   lattice-compose ark:- ark:2.lats ark:3.lats
```

Lattice equivalence testing

`lattice-equivalent` 作为一个调试工具非常有用。它测试网格是否相等，如果是，返回状态0。它通过利用一个随机化的相等测试算法来完成。一个例子：

```
lattice-equivalent ark:1.lats ark:2.lats || echo "Not equivalent!"
```

可选的参数包括 -num-paths（在随机化的测试算法中用到的路径数）和 -delta（在相等测试时允许的得分的差异值）。

Removing alignments from lattices

`lattice-rmali` 从网格的输入端移除对齐信息（i.e. the transition-ids）。这在你不需要某种信息时很有用（e.g. LM重打分时你只需要网格），而且可以节省储存空间。例子：

```
lattice-rmali ark:in.lats ark:word.lats
```

Error boosting in lattices

The program lattice-boost-ali is used in boosted MMI training. It reads in a lattice and an alignment (both as tables, e.g. archives), and outputs the lattices with the language model scores (i.e. the graph part of the scores) boosted by the parameter b times the number of frame-level phone errors on that path. Typical usage is:

```
lattice-boost-ali --silence-phones=1:2:3 --b=0.1 final.mdl ark:1.lats \
 ark:1.ali ark:boosted.lats
```

The silence phones are treated specially: wherever they appear in the lattices, they are assigned zero error, i.e. they are not counted as errors (this behavior can be controlled with the –max-silence option). Note that this special treatment of silence has been adopted from MPE training where it appeared to help; we have not investigated its exact effect on boosted MMI.

Computing posteriors from lattices

The program lattice-to-post computes, from a lattice, per-frame posterior probabilities of the transition-ids in the lattice. This is done by a standard forward-backward type of algorithm. Since, by convention, we store lattices without any acoustic scaling, it will normally be necessary to supply an acoustic scale to be used in the forward-backward algorithm. It also accepts a language model scale, but the default (1.0) will often be most appropriate. An example of using this program is:

```
lattice-to-post --acoustic-scale=0.1 ark:1.lats ark:- | \
  gmm-acc-stats 10.mdl "$feats" ark:- 1.acc
```

Determinization of lattices

The program lattice-determinize implements lattice-determinization, which essentially consists of keeping, for each word-sequence in the lattice, only the best-scoring sequence of transition-ids. In general this process is sensitive to the acoustic scale (because "best-scoring" depends on the scale), but if the lattice has previously been determinized and then has only been processed in a "word-level" way, i.e. each word-sequence still has only one transition-id sequence, then the acoustic scale doesn't matter. The only time the acoustic scale is likely to matter is if you generate state-level lattices, e.g. you do lattice generation with –determinize-lattice=false. This program has other options, that are mostly related to what to do if determinization "blows up", i.e. exhausts memory, but in general you can just leave these at their default values. These are mostly there because a previous version of the determinization tended to exhaust memory. You may want to set –prune=true if you want to do pruning as part of the same program, but be careful that this only makes sense if you have set the acoustic scale to an appropriate value; you might also want to set the –beam option in that case.

A typical usage is:

```
lattice-determinize ark:1.lats ark:det.lats
```

or:

```
lattice-determinize --acoustic-scale=0.08333 ark:state_level.lats ark:1.lats
```

Note that lattices produced by lattice generation programs will be default already be determinized, so it only makes sense to do determinization if you have just done an operation like composition that makes lattices nondeterministic, or if you have given –determinize-lattice=false to the lattice generation program to create state-level lattices.

Computing oracle WERs from lattices

`lattice-oracle` 输入是两个 tables：第一个是网格，第二个是 transcriptions（C++ 用 vector<>表示），输出是网格的词序列；在日志中会打印出相应的 WERs。这是通过构建一个“edit-distance FST”，把它和由网格和参考得到的 unweighted acceptors组合起来，来完成的。

一个例子（这个脚本片段来自 s3系列）：

```
cat $data/text | \
 sed 's:<NOISE>::g' |  sed 's:<SPOKEN_NOISE>::g'  | \
 scripts/sym2int.pl --ignore-first-field $lang/words.txt | \
 lattice-oracle --word-symbol-table=$lang/words.txt  \
   "ark:gunzip -c $dir/lats.pruned.gz|" ark:- ark,t:$dir/oracle.tra \
   2>$dir/oracle.log
```

Adding transition probabilities to lattices

`lattice-add-trans-probs` 把 transition概率加到网格的 costs中。这在做丢弃原始的 graph costs并从头重建的 lattice rescoring时很有用。一个典型用法：

```
lattice-add-trans-probs --transition-scale=1.0 --self-loop-scale=0.1 \
    final.mdl ark:1.lats ark:2.lats
```

关于这些缩放参数的更多解释，见 Scaling of transition and acoustic probabilities。

Converting lattices to FSTs

`lattice-to-fst` 将网格转换为 FSTs，和编译训练图(the training graphs)的代码用到的格式相同。得到的 FSTs就是 weighted word acceptors（尽管目前的程序中我们乘以了系数0，也就是没有 weights）。它们可以作为 `compile-train-graphs-fst` 的输入来生成只包含网格中存在的路径的解码图。它的主要用途是在不共享同一个树的模型间做 lattice rescoring。一个例子是：

```
lattice-to-fst --lm-scale=0.0 --acoustic-scale=0.0 ark:1.lats ark:1.words
```

Copying lattices

`lattice-copy` 作用就是拷贝网格，这在你想看二进制 FST的文本形式时很有用。例如：

```
lattice-copy ark:1.lats ark,t:- | head -50
```

N-best lists and best paths

有些时候（e.g. Viterbi training;用神经网络语言模型重打分）我们不需要网格结构而是需要最佳路径或 N-best路径。N-best列表的格式和网格一样，除了每个句子有多个 FSTs（最多 n 个，如果设定了 n）。假设 utterance ids是 uttA, uttB等。网格的一个 Table（e.g. an archive）会包含 uttA的网格，uttB的网格，等等。如果运行：

```
lattice-to-nbest --acoustic-scale=0.1 --n=10 ark:1.lats ark:1.nbest
```

那么 archive 1.nbest会包含网格 uttA-1, uttA-2,...uttA-10和 uttB-1...uttB-10等等。当然某些语句可能没有这么多的 N-best，因为网格没有那么多的不同的词序列。 `lattice-to-nbest` 需要 acoustic scale因为这影响到哪个路径得分最低。

某些情况下处理 FST还是不够方便，而我们需要一个线性结构，e.g. 一个词序列。这时，我们可以用：

```
nbest-to-linear ark:1.nbest ark:1.ali ark:1.words ark:1.lmscore ark:1.acscore
```

这个程序读网格的 archive，它必须是一个线性 FST，然后输出4个 archives，对应输入符号序列，输出符号序列，the acoustic and LM scores。格式会像下面一样（假定你以文本方式写）：

```
# head 1.words
utt-A-1 10243 432  436 10244
utt-A-2 7890  420 10244
...
```

Kaldi会处理后面的数字 id；你可以用符号表和 scripts/int2sym.pl把它们转换为词。上面的逆变换可以用 `linear-to-nbest` 完成。

另一个有用的程序是 `lattice-to-1best`，它和 `lattice-to-nbest --n=1` 不太一样，因为语句id后面不会有“-1”。你可以把输出送入到 `nbest-to-linear` 来得到词序列或类似的内容。 `lattice-best-path` 输入网格并输出两个 archives，包含词序列和 transition-id序列。这样用是有历史原因的； `lattice-to-1best` 和 `nbest-to-linear` 组合起来用更加清晰，和整体框架也更一致。

Times on lattices

如果你明确地想要网格的时间信息（而不是要去数 transition-ids），有 LatticeStateTimes函数（for Lattice）和 CompactLatticeStateTimes函数（for CompactLattice），会给出你每个状态的时间位置（从0到文件的帧数）。要注意的是，一般来讲，词和 transition-ids是没有对齐的，意味着同一个 arc上的 transition-ids不一定都属于 arc上的词标签。也就是说你从网格得到的时间（仅考虑到词标签）是不准确的。对权重来说也是这样，它们并没有和 arc上的词或 transition-ids对齐。如果你想要精确的时间（e.g. 用于转换为 HTK网格，或 sclite scoring），你需要运行 `lattice-align-words`。这个程序只有在你使用 word-position-dependent phones搭建系统时才能发挥作用，而且它需要有特定的命令行选项来告诉它音素在词中的位置。一个例子见 egs/wsj/s3/run.sh。如果你的系统没有 word-position-dependent phones，你可以用另一个程序 `lattice-align-words-lexicon`。

声学模型代码

翻譯: 老那(asr.naxingyu@gmail.com)

时间: 2014年4月

简介

我们先来简单说说Kaldi中用于建模的代码的思路，以及为什么这样设计。我们的目标是使Kaldi支持传统模型，也就是对角阵高斯混合模型（GMM）和子空间高斯混合模型（SGMM），同时也要便于扩展到新的模型。在上一轮代码设计过程中，我们设计了一个虚基类，从其中衍生出GMM类和SGMM类，然后编写了一个命令行工具用于操作这两种模型。但是根据我们的经验，基类并不像事先想象的那样能派上大用场，因为这两种模型之间的差异太大（例如，它们支持的自适应算法不同）。为了构建我们想象中具有“普适性”的代码，使它能够处理不同模型类别，我们不得不不断的扩展基类。最终，我们的命令行工具已经很难修改和维护了。

在重设计代码的时候，我们决定使用更“现代化”的软件工程方法，不为了增强普适性而过于强调类的层级结构，而是专注于创建简单的、可复用的组件。例如，我们的解码器代码是很通用的，因为它的需求很少，它只要我们从简单的基类DecodableInterface中 创造一个衍生的实例。这个实例操作起来类似于由一个句子的声学似然度组成的矩阵（参见Kaldi工具中的解码器）。同时，每个命令行工具更简单易用，例如 `gmm-align` 用于给定对角GMM获得一句话的状态级对齐结果。总的想法是，要实现某种新技术只要编写一个新的命令行程序，而不是增加已有的命令行程序的复杂度。

对角GMM

DiagGmm类表示一个对角协方差矩阵的高斯混合模型。一组DiagGmm的对象，由从0开始的 `pdf-ids` 索引组合在一起，组成声学模型。这个声学模型是由AmDiagGmm类实现的。尽管这个声学模型类的接口比单个GMM丰富，但你可以简单地把AmDiagGmm看作是DiagGmm的矢量。把声学模型表示为独立模型的集合，每个模型用于表示一个pdf，我们可以想象，这不是对所有模型都适用的构建方法。例如，SGMM就不能这样表示。而且，如果我们想实现不同状态之间GMM子成分的捆绑，就不能把每个pdf单独表示了。（译者注：pdf，科技文献中写作p.d.f.，是概率密度函数的缩写。）

独立*GMM*

理论上，DiagGmm类是一个简单的、被动的对象，负责存储高斯混合模型的参数，并且有一个用于计算似然度的成员函数。它完全不知道自己会被如何使用，它只提供获取其成员的接口。它不获取累计量（Accumulation），不更新参数（对于参数更新，参见 `MlEstimateDiagGmm` 类）。DiagGmm类存储参数的方式是：方差的倒数，以及均值乘以方差倒数的结果。这意味着，通过简单的内积操作就可以计算似然度。与HTK的 `gconst` 不同，Kaldi中的 `gconst` 是独立于均值的。

由于以这种方式调整高斯参数很不方便，我们提供了DiagGmmNormal类， 用于以简单直观的方式存储参数。同时，我们提供了DiagGmm和DiagGmmNormal实 例互相转换的函数。大部分的参数更新代码作用于DiagGmmNormal类。

基于*GMM*的声学模型

AmDiagGmm类 表示一组DiagGmm对象，由 `pdf-id` 索引。该类不表示HMM-GMM模型，仅仅是一组GMM的集合。将其与HMM组合在一起由其他代码负责，主要是负责拓扑结构和转移概率部分的代码以 及负责整合解码图的代码（参见HMM的拓扑结构和状态转移的建模）。在这里需要指出的是，在声学模型中，我们没有只存储一个AmDiagGmm对象，而是同时保存了一个TransitionModel对象和一个AmDiagGmm对象。这是为了避免在存储器上写入太多的独立文件。这样做的原因是，高斯模型和转移概率的更新往往是同时进行的。这样做的初衷是，在创建其他类型的 模型时，我们可以将一个TransitionModel和一个目标模型的对象保存在同一个文件中。这样，那些只需要读取转移模型的程序（如创建解码图）就可以直接读取模型文件，而不需要知道其中声学模型的类型。

AmDiagGmm类是一个相对简单的对象，并不负责模型估计（参见AccumAmDiagGmm）和变换矩阵估计等工作。在Kaldi中，有其他的代码来负责这些工作，参见特征域和模型域变换。

满协方差*GMM*

对于满协方差矩阵的GMM，我们创建了FullGmm类，其作用与DiagGmm相似，区别只在与协方差矩阵的形式。这个类主要是用来在SGMM建模中训练满协方差的广义背景模型（UBM）。唯一用于满方差GMM的命令行工具是用于训练全局混合模型的，也就是UBM。我们没有编写AmDiagGmm类的满方差版本以及对应的命令行工具。不过要实现这个功能也不是难事。

子空间高斯混合模型（SGMM）

子空间高斯混合模型（SGMM）由AmSgmm类实现。这个类从本质上实现了"The Subspace Gaussian Mixture Model – a Structured Model for Speech Recognition"（by D. Povey, Lukas Burget et. al, Computer Speech and Language, 2011.）这篇论文中的方法。AmSgmm类表

示一个完整的pdf的集合。Kaldi中没有用于表示一个SGMM的类（如用于GMM的DiagGmm）。SGMM参数的估计是由MleAmSgmmAccs类和MleAmSgmmUpdater类完成的。

对于如何训练一个基于SGMM系统的示例脚本，参见 `egs/rm/s1/steps/train_ubma.sh`， `egs/rm/s1/steps/train_sgmma.sh` 和 `egs/rm/s1/steps/decode_sgmma.sh` 。

特征提取

联系：wbgxx333@163.com

时间：2014年4月由@煮八戒翻译，由@wbgxx333校队和翻译，后由@wbgxx333搬移到这里。

简介

我们做特征提取和波形读取的这部分代码，其目的是为了得到标准的MFCC(译注：梅尔倒谱系数)和PLP(译注：感知线性预测系数)特征，设置合理的默认值但留了一部分用户最有可能想调整的选项（如梅尔滤波器的个数，最小和最大截止频率等等）。这部分代码只读取wav文件里的pcm(译注：脉冲编码调制)数据。这类文件通常带.wav或.pcm后缀（虽然有时.pcm后缀会用于sph文件；这种情况下必须转换该文件）。假如源数据不是wav类文件，则用户可自由选择命令行工具来转换，而我们提供的sph2pipe工具已能满足一般的情况。 命令行工具compute-mfcc-feats和compute-plp-feats计算特征；同其它Kaldi工具一样，不带参数地运行它们会给出一个选项列表。例子脚本里显示了这些工具的用法。

计算MFCC特征

这里我们介绍如何使用命令行工具compute-mfcc-feats计算MFCC参数。该程序需要两个命令行参数：rspecifier是用来读.wav数据（以发音为索引）和wspecifier是用来写特征（以发音为索引）；参见 The Table concept和Specifying Table formats: wspecifiers and rspecifiers获取更多关于这些术语的解释。典型的用法是，将数据写入一个大的“archive”文件，也写到一个“scp”文件以便随机存取；参见Writing an archive and a script file simultaneously解释。程序没有添加增量功能（如需添加，参见add-deltas）。它接收选项-channel来选择通道（如–channel=0, –channel=1），该选项在读取立体声数据时很有用。 计算MFCC特征由Mfcc类型的对象完成，它有Compute()函数可以根据波形计算特征。 完整的MFCC计算如下所示：

- 计算出一个文件中帧的数目（通常帧长25ms帧移10ms）。
- 对每一帧：
  - 提取数据，可选做dithering（注：直译为抖动，在这里可以理解为类似归一化的预处理），预加重和去除直流偏移，还可以和加窗函数相乘（此处支持多种选项，如汉明窗）
  - 计算该点能量（假如用对数能量则没有C0）
  - 做FFT（译注：快速傅里叶变换）并计算功率谱
  - 计算每个梅尔滤波器的能量；如23个部分重叠的三角滤波器，其中心在梅尔频域等间距

- 计算对数能量并作余弦变换，根据要求保留系数（如13个）
- 选做倒谱变换；它仅仅是比例变换，确保系数在合理范围。

上下截止频率根据三角滤波器界定，由选项–low-freq和–high-freq控制，通常分别设置为0Hz和奈奎斯特频率附近，如对16kHz采样的语音设置为–low-freq=20 和 –high-freq=7800。Kaldi的特征和HTK的特征在很多方面不同，但是几乎所有这些不同归结于有不同的默认值。用选项–htk-compat=true并正确设置参数，能得到同HTK非常接近的特征。一个可能重要的选项是我们不支持能量最大归一化。这是因为我们希望能把无状态方式应用到归一化方法，且希望从原理上计算一帧帧特征仍能给出相同结果。但是程序compute-mfcc-feats里有–subtract-mean选项来提取特征的均值。对每个语音做此操作；每个说话人可以有不同的方式来提取特征均值。（如在脚本里搜“cmvn”，表示倒谱均值和方差归一化）。

计算**PLP**特征

计算PLP特征的算法与MFCC的算法前期是一样的。稍后我们也许会在此部分增加些内容，但目前参见Hynek Hermansky《语音的感知线性预测（PLP）分析》，Journal of the Acoustical Society of America, vol. 87, no. 4, pages 1738–1752 (1990).

特征级声道长度归一化（**VTLN**）

程序compute-mfcc-feats和compute-plp-feats接收一个VTLN弯折因子选项。在目前的脚本中，这仅用作线性版的VTLN的初始化线性转换的一种方法。VTLN通过移动三角频率箱的中心频率的位置来实现。移动频率箱的弯折函数是一个在频域空间分段线性的函数。为理解它，记住以下数量关系：

```
0 <= low-freq <= vtln-low < vtln-high < high-freq <= nyquist
```

此处，low-freq和high-freq分别是用于标准MFCC或PLP计算的最低和最高频率（忽略更低和更高的频率）。vtln-low和vtln-high是用于VTLN的截止频率，它们的功能是确保所有梅尔滤波器有合适的宽度。 我们实现的VTLN弯折函数是一个分段线性函数，三个部分映射区间[low-freq, high-freq]至[low-freq, high-freq]。记弯折函数为W(f)，f是频率。中段映射f到f/scale，scale是VTLN弯折因子（通常范围为0.8到1.2）。x轴上低段和中段的连接点是满足min(f, W(f)) = vtln-low的f点。x轴上中段和高端的连接点是满足max(f, W(f)) = vtln-high的f点。要求低段和高段的斜率和偏移是连续的且W(low-freq)=low-freq， W(high-freq)=high-freq。这个弯折函数和HTK的不同；HTK的版本中，"vtln-low"和"vtln-high"的数量关系是x轴上可以不连续的点，这意味着变量"vtln-high"必须基于弯折因子的可能范围的先验知识谨慎选择（否则梅尔滤波器可能为空）。 一个合理的设置如下（以16kHz采样的语音为例）；注意这反映的是我们理解的合理值，并非任何非常细致的调试实验的结果。

low-freq	vtln-low	vtln-high	high-freq	nyquist
40	60	7200	7800	8000

特征域和模型域变换

Kaldi中的深度神经网络

翻译：wbgxx333@163.com

时间：2014年4月，2015年4月重新修改

介绍

深度神经网络已经是语音识别领域最热的话题了。从2010年开始，许多关于深度神经网络的文章在这个领域发表。许多大型科技公司（谷歌和微软）开始把DNN用到他们的产品系统里。

但是，没有一个工具箱像kaldi这样可以很好的提供支持。因为先进的技术无时无刻不在发展，这就意味着代码需要跟上先进技术的步伐和代码的架构需要重新去思考。

我们现在在kaldi里提供两套分离的关于深度神经网络代码。一个在代码目录下的nnet/和nnetbin/,这个是由 Karel Vesely提供。此外，还有一个在代码目录nnet-cpu/和nnet-cpubin/，这个是由 Daniel Povey提供（这个代码是从Karel早期版本修改，然后重新写的）。这些代码都是很官方的，这些在以后都会发展的。

在例子目录下，比如： egs/wsj/s5/, egs/rm/s5, egs/swbd/s5 and egs/hkust/s5b，神经网络的例子脚本都可以找到。 Karel的例子脚本可以在local/run_dnn.sh或者local/run_nnet.sh，而Dan的例子脚本在local/run_nnet_cpu.sh。在运行这些脚本前，为了调整系统，run.sh你必须首先被运行。

我们会很快的把这两个神经网络的详细文档公布。现在，我们总结下这两个的最重要的区别：

- Karel的代码(nnet1)支持用单卡GPU训练，这使得实现更加简单和相对容易修改。
- Dan的代码(nnet2)在训练方式上是灵活的：它支持多线程的多GPUs, 或者多CPUs。一般推荐多GPU。它们不必全部是相同的机器上。每一个都可以获得差不多的结果。

在具体的实现中，这两个代码有很多的不同点。举例来说，Karel的部分需要使用预训练，但是Dan的却不需要；Karel部分需要使用一个有效集来早点停止迭代，而Dan在最后几轮迭代训练中使用固定的迭代次数和平均化参数。更多的训练细节(非线性类型，学习率，网络类型，输入特征等) 也不同。

DNN部分最好的描述如下：

- Karel部分: Sequence-discriminative training of deep neural networks
- Dans部分: Parallel training of DNNs with natural gradient and parameter averaging

这两部分的DNN格式是不兼容的，这里有一个把Karel网络转换为 Dan的格式： Conversion of a DNN model between nnet1 -> nnet2。

Karel版本的文档在 Karel's DNN implementation 和Dan版本的文档在 Dan's DNN implementation。

karel的深度神经网络

翻译：wbgxx333@163.com

时间：2014年4月翻译，2015年4月重新修改翻译

综述

这个文档主要来说kaldi中Karel Vesely部分的深度神经网络代码。

如果想了解kaldi的全部深度神经网络代码，请Deep Neural Networks in Kaldi, 和Dan的版本, 请看Dan's DNN implementation。

这个文档的目标就是更加详细的介绍DNN部分，和简单介绍神经网络训练工具。我们将从Top-level script开始, 解释the Training script internals到底做了什么, 展示一些Advanced features, 和对The C++ code做了一些简单的介绍，和解释如何来扩展这些。

Top-level script

让我们来看一下脚本egs/wsj/s5/local/nnet/run_dnn.sh。这个脚本是使用单CUDA GPU，和使用CUDA编译过的kaldi(可以在 src/kaldi.mk中使用'CUDA = true'来检查)。我们也假设'cuda_cmd'在egs/wsj/s5/cmd.sh里设置是正确的，或者是使用'queue.pl'的GPU集群节点，或者是使用'run.pl'的本地机器。最后假设我们由egs/wsj/s5/run.sh得到了一个SAT GMM系统exp/tri4b和对应的fMLLR变换。注意其他数据库的 run_dnn.sh一般都会在s5/local/nnet/run_dnn.sh.

脚本 egs/wsj/s5/local/nnet/run_dnn.sh分下面这些步骤：

0.存储在本地的**40**维**fMLLR**特征, 使用**steps/nnet/make_fmllr_feats.sh**,这简化了训练脚本，40维的特征是使用CMN的MFCC-LDA-MLLT-fMLLR。

1. RBM 预训练**, steps/nnet/pretrain_dbn.sh**,是根据Geoff Hinton's tutorial paper来实现的。训练方法是使用1步马尔科夫链蒙特卡罗采样的对比散度算法(CD-1)。 第一层的RBM是Gaussian-Bernoulli，和接下来的RBMs是Bernoulli-Bernoulli。这里的超参数基准是在100h Switchboard subset数据集上调参得到的。如果数据集很小的话，迭代次数N就需要变为100h/set_size。训练是无监督的，所以可以提供足够多的输入特征数据目录。

当训练Gaussian-Bernoulli的RBM时，将有很大的风险面临权重爆炸，尤其是在很大的学习率和成千上万的隐层神经元上。为了避免权重爆炸，在实现时我们需要在一个minbatch上比较训练数据的方差和重构数据的方差。如果重构的方差是训练数据的2倍以上，权重将缩小和学

习率将暂时减小。

2. 帧交叉熵训练，**steps/nnet/train.sh**, 这个阶段是训练一个DNN来把帧分到对应的三音素状态(比如： PDFs)中。这是通过mini-batch随机梯度下降法来做的。（译者注：mini-batch指的就是分批处理，它的错误率表示的：每一次epoch中，所有的小batch的平均损失函数值；而full-batch是完整的数据集，它的错误率表示：一次epoch中，整个大batch的损失函数值。）默认的是使用Sigmoid隐层单元，Softmax输出单元和全连接层AffineTransform。学习率是0.008，minibatch的大小是256；这里我们未使用冲量和正则化(注: 最佳的学习率与不同的隐含层单元类型有关，sigmoid的值0.008，tanh是0.00001)。

输入变换和预训练DBN(比如：深度信念网络,RBMs块)是使用选项 '–input-transform'和'–dbn'传递给脚本的，这里仅仅输出层是随机初始化的。我们使用提早停止(early stopping)来防止过拟合。为了这个，我们需要在交叉验证集(比如: held-out set)上计算代价函数，因此两对特征对齐目录需要做有监督的训练。

对DNN训练有一个好的总结文章是http://research.google.com/pubs/archive/38131.pdf

3.,4.,5.,6. sMBR序列区分性训练，**steps/nnet/train_mpe.sh,** 这个阶段对所有的句子联合优化来训练神经网络，比帧层训练更接近一般的ASR目标。

- sMBR的目标是最大化从参考的对齐中得到的状态标签的期望正确率，然而这里使用一个词图框架是来表示这种竞争假设。
- 训练是使用每句迭代的随机梯度下降法，我们还使用一个低的固定的学习率1e-5 (sigmoids)和跑3-5轮。
- 当在第一轮迭代后重新生成词图，我们观察到快速收敛。我们支持MMI, BMMI, MPE 和 sMBR训练。所有的技术在Switchboard 100h集上是相同的，仅仅在sMBR好一点点。
- 在sMBR优化中，我们在计算近似正确率的时候忽略了静音帧。具体更加详细的描述见 http://www.danielpovey.com/files/2013_interspeech_dnn.pdf

其他一些有意思的**top-level scripts:**

除了DNN脚本，这里也有一些其他的脚本：

- DNN : egs/wsj/s5/local/nnet/run_dnn.sh , (main top-level script)
- CNN : egs/rm/s5/local/nnet/run_cnn.sh , (CNN = Convolutional Neural Network, see paper, we have 1D convolution on frequency axis)
- Autoencoder training : egs/timit/s5/local/nnet/run_autoencoder.sh
- Tandem system : egs/swbd/s5c/local/nnet/run_dnn_tandem_uc.sh , (uc = Universal context network, see paper)
- Multilingual/Multitask : egs/rm/s5/local/nnet/run_multisoftmax.sh, (Network with output trained on RM and WSJ, same C++ design as was used in SLT2012 paper)

Training script internals

主要的神经网络训练脚本steps/nnet/train.sh的调用如下：

```
steps/nnet/train.sh <data-train> <data-dev> <lang-dir> <ali-train> <ali-dev> <exp-dir>
```

神经网络的输入特征是从数据目录 `<data-train><data-dev>` 中获得的，训练的目标是从目录 `<ali-train> <ali-dev>` 得到的。目录 `<lang-dir>` 仅仅在使用LDA特征变换时才被使用，和从对齐中生成音素帧的统计量，这个对于训练不是很重要。输出(比如：训练得到的网络和log文件)都存到 `<exp-dir>` 。

在内部，脚本需要准备特征和目标基准，从而产生一个神经网络的原型和初始化，建立特征变换和使用调度脚本 steps/nnet/train_scheduler.sh，用来跑训练迭代次数和控制学习率。

当看**steps/nnet/train.sh**脚本内部时，我们将看到：

1. CUDA是需要的，如果没有检测到GPU或者CUDA没有被编译，脚本将退出。(你可以坚持使用'–skip-cuda-check true'来使用CPU运行，但是速度将慢10-20倍)
2. 对齐基准需要提前准备，训练工具需要的目标是以后验概率格式，因此ali-to-post.cc被使用：

```
labels_tr="ark:ali-to-pdf $alidir/final.mdl \"ark:gunzip -c $alidir/ali.*.gz |\" ark:
labels_cv="ark:ali-to-pdf $alidir/final.mdl \"ark:gunzip -c $alidir_cv/ali.*.gz |\" a
```

3. 重组的特征拷贝到/tmp/???/...，如果使用'–copy-feats false'，这个将失效。或者目录改为 `–copy-feats-tmproot <dir>` 。
    - 特征使用调用列表被重新保存到本地，这些显著地降低了在训练过程中磁盘的重要性，它防止了大量磁盘访问的操作。
4. 特征基准被准备：

```
# begins with copy-feats:
feats_tr="ark:copy-feats scp:$dir/train.scp ark:- |"
feats_cv="ark:copy-feats scp:$dir/cv.scp ark:- |"
# optionally apply-cmvn is appended:
feats_tr="$feats_tr apply-cmvn --print-args=false --norm-vars=$norm_vars --utt2spk=ar
feats_cv="$feats_cv apply-cmvn --print-args=false --norm-vars=$norm_vars --utt2spk=ar
# optionally add-deltas is appended:
feats_tr="$feats_tr add-deltas --delta-order=$delta_order ark:- ark:- |"
feats_cv="$feats_cv add-deltas --delta-order=$delta_order ark:- ark:- |"
```

5. 特征变换被准备：
    - 特征变换在DNN前端处理中是一个固定的函数，是通过GPU来计算的。一般来说，它会导致维度爆炸。这就要使得在磁盘上有低维的特征和DNN前端处理的高维特

征， 即节约了磁盘空间，由节约了读取吞吐量。
    - 大多数的nnet-binaries有选项'–feature-transform'
    - 它的产生依赖于选项'–feat-type'，它的值是(plain|traps|transf|lda)。
6. 网络的原型是由utils/nnet/make_nnet_proto.py产生的:

    - 每个成分在单独一行上，这里的维度和初始化的超参数是指定的；
    - 对于AffineTransform，偏移量的初始化是的均匀分布给定 `<BiasMean>` 和 `<BiasRange>` 的均匀分布，而权重的初始化是通过通过对 `<ParamStddev>` 拉伸的正态分布；
    - 注意：如果你喜欢使用外部准备的神经网络原型来实验，可以使用选项'–mlp-proto'；

```
$ cat exp/dnn5b_pretrain-dbn_dnn/nnet.proto
<NnetProto>
<AffineTransform> <InputDim> 2048 <OutputDim> 3370 <BiasMean> 0.000000 <BiasRange> 0.
<Softmax> <InputDim> 3370 <OutputDim> 3370
</NnetProto>
```

7. 神经网络是通过nnet-initialize.cc来初始化。下一步中， DBN是通过使用nnet-concat.cc得到的。

8. 最终训练是通过运行调度脚本steps/nnet/train_scheduler.sh来完成的。

    注：无论神经网络还是特征变换都可以使用nnet-info.cc来观看,或者用nnet-copy.cc来显示。

当具体看**steps/nnet/train_scheduler.sh**，我们可以看到：

一开始需要在交叉验证集上运行，和主函数需要根据$iter来运行迭代次数和控制学习率。典型的情况就是，train_scheduler.sh被train.sh调用

- 默认的学习率的变化是根据目标函数相对性的提高来决定的：
    - 如果提高大于'start_halving_impr=0.01'，初始化学习率保持常数
    - 然后学习率在每次迭代中乘以'halving_factor=0.5'来缩小
    - 最后，如果提高小于'end_halving_impr=0.001'，训练被终止。

神经网络被保存在$dir/nnet，log文件被保存在$dir/log:

1. 神经网络的名字包含迭代的次数，学习率和在训练和交叉验证集上的目标函数值
    - 我们可以看到从第五次迭代开始，学习率减半，这是一个普通的情况。

```
$ ls exp/dnn5b_pretrain-dbn_dnn/nnet
nnet_6.dbn_dnn_iter01_learnrate0.008_tr1.1919_cv1.5895
nnet_6.dbn_dnn_iter02_learnrate0.008_tr0.9566_cv1.5289
nnet_6.dbn_dnn_iter03_learnrate0.008_tr0.8819_cv1.4983
nnet_6.dbn_dnn_iter04_learnrate0.008_tr0.8347_cv1.5097_rejected
nnet_6.dbn_dnn_iter05_learnrate0.004_tr0.8255_cv1.3760
nnet_6.dbn_dnn_iter06_learnrate0.002_tr0.7920_cv1.2981
nnet_6.dbn_dnn_iter07_learnrate0.001_tr0.7803_cv1.2412
...
nnet_6.dbn_dnn_iter19_learnrate2.44141e-07_tr0.7770_cv1.1448
nnet_6.dbn_dnn_iter20_learnrate1.2207e-07_tr0.7769_cv1.1446
nnet_6.dbn_dnn_iter20_learnrate1.2207e-07_tr0.7769_cv1.1446_final_
```

2. 训练集和交叉验证集分别存储了对应的log文件。 每一个log文件命令行：

```
$ cat exp/dnn5b_pretrain-dbn_dnn/log/iter01.tr.log
nnet-train-frmshuff --learn-rate=0.008 --momentum=0 --l1-penalty=0 --l2-penalty=0 --r
```

gpu被使用的信息：

```
LOG (nnet-train-frmshuff:IsComputeExclusive():cu-device.cc:214) CUDA setup operating unde
LOG (nnet-train-frmshuff:FinalizeActiveGpu():cu-device.cc:174) The active GPU is [1]: GeF
```

从神经网络训练得到的内部统计量是通过函数 Nnet::InfoPropagate，Nnet::InfoBackPropagate 和 Nnet::InfoGradient来准备的。它们将在迭代的一开始打印和迭代的最后一次进行第二次打印。注意当我们实现新的特征来做网络训练时，每一个成分的统计量就尤其便利地计算，所以我们可以比较参考的值和期望的值：

```
VLOG[1] (nnet-train-frmshuff:main():nnet-train-frmshuff.cc:236) ### After 0 frames,
VLOG[1] (nnet-train-frmshuff:main():nnet-train-frmshuff.cc:237) ### Forward propagation b
[1] output of <Input> ( min -6.1832, max 7.46296, mean 0.00260791, variance 0.964268, ske
[2] output of <AffineTransform> ( min -18.087, max 11.6435, mean -3.37778, variance 3.280
[3] output of <Sigmoid> ( min 1.39614e-08, max 0.999991, mean 0.085897, variance 0.024987
[4] output of <AffineTransform> ( min -17.3738, max 14.4763, mean -2.69318, variance 2.08
[5] output of <Sigmoid> ( min 2.84888e-08, max 0.999999, mean 0.108987, variance 0.021520
[6] output of <AffineTransform> ( min -16.3061, max 10.9503, mean -3.65226, variance 2.49
[7] output of <Sigmoid> ( min 8.28647e-08, max 0.999982, mean 0.0657602, variance 0.02121
[8] output of <AffineTransform> ( min -19.9429, max 12.5567, mean -3.64982, variance 2.49
[9] output of <Sigmoid> ( min 2.1823e-09, max 0.999996, mean 0.0671024, variance 0.021642
[10] output of <AffineTransform> ( min -16.79, max 11.2748, mean -4.03986, variance 2.157
[11] output of <Sigmoid> ( min 5.10745e-08, max 0.999987, mean 0.0492051, variance 0.0194
[12] output of <AffineTransform> ( min -24.0731, max 13.8856, mean -4.00245, variance 2.1
[13] output of <Sigmoid> ( min 3.50889e-11, max 0.999999, mean 0.0501351, variance 0.0200
[14] output of <AffineTransform> ( min -2.53919, max 2.62531, mean -0.00363421, variance
[15] output of <Softmax> ( min 2.01032e-05, max 0.00347782, mean 0.000296736, variance 2.
```

```
VLOG[1] (nnet-train-frmshuff:main():nnet-train-frmshuff.cc:239) ### Backward propagation
[1] diff-output of <AffineTransform> ( min -0.0256142, max 0.0447016, mean 1.60589e-05, v
[2] diff-output of <Sigmoid> ( min -0.10395, max 0.20643, mean -2.03144e-05, variance 5.4
[3] diff-output of <AffineTransform> ( min -0.0246385, max 0.033782, mean 1.49055e-05, va
[4] diff-output of <Sigmoid> ( min -0.137561, max 0.177565, mean -4.91158e-05, variance 4
[5] diff-output of <AffineTransform> ( min -0.0311345, max 0.0366407, mean 1.38255e-05, v
[6] diff-output of <Sigmoid> ( min -0.154734, max 0.166145, mean -3.83602e-05, variance 5
[7] diff-output of <AffineTransform> ( min -0.0236995, max 0.0353677, mean 1.29041e-05, v
[8] diff-output of <Sigmoid> ( min -0.103117, max 0.146624, mean -3.74798e-05, variance 6
[9] diff-output of <AffineTransform> ( min -0.0249271, max 0.0315759, mean 1.0794e-05, va
[10] diff-output of <Sigmoid> ( min -0.147389, max 0.131032, mean -0.00014309, variance 0
[11] diff-output of <AffineTransform> ( min -0.057817, max 0.0662253, mean 2.12237e-05, v
[12] diff-output of <Sigmoid> ( min -0.311655, max 0.331862, mean 0.00031612, variance 0.
[13] diff-output of <AffineTransform> ( min -0.999905, max 0.00347782, mean -1.33212e-12,

VLOG[1] (nnet-train-frmshuff:main():nnet-train-frmshuff.cc:240) ### Gradient stats :
Component 1 : <AffineTransform>,
  linearity_grad ( min -0.204042, max 0.190719, mean 0.000166458, variance 0.000231224, s
  bias_grad ( min -0.101453, max 0.0885828, mean 0.00411107, variance 0.000271452, skewne
Component 2 : <Sigmoid>,
Component 3 : <AffineTransform>,
  linearity_grad ( min -0.108358, max 0.0843307, mean 0.000361943, variance 8.64557e-06,
  bias_grad ( min -0.0658942, max 0.0973828, mean 0.0038158, variance 0.000288088, skewne
Component 4 : <Sigmoid>,
Component 5 : <AffineTransform>,
  linearity_grad ( min -0.186918, max 0.141044, mean 0.000419367, variance 9.76016e-06, s
  bias_grad ( min -0.167046, max 0.136064, mean 0.00353932, variance 0.000322016, skewnes
Component 6 : <Sigmoid>,
Component 7 : <AffineTransform>,
  linearity_grad ( min -0.134063, max 0.149993, mean 0.000249893, variance 9.18434e-06, s
  bias_grad ( min -0.165298, max 0.131958, mean 0.00330344, variance 0.000438555, skewnes
Component 8 : <Sigmoid>,
Component 9 : <AffineTransform>,
  linearity_grad ( min -0.264095, max 0.27436, mean 0.000214027, variance 1.25338e-05, sk
  bias_grad ( min -0.28208, max 0.273459, mean 0.00276327, variance 0.00060129, skewness
Component 10 : <Sigmoid>,
Component 11 : <AffineTransform>,
  linearity_grad ( min -0.877651, max 0.811671, mean 0.000313385, variance 0.000122102, s
  bias_grad ( min -1.01687, max 0.640236, mean 0.00543326, variance 0.00977744, skewness
Component 12 : <Sigmoid>,
Component 13 : <AffineTransform>,
  linearity_grad ( min -22.7678, max 0.0922921, mean -5.66685e-11, variance 0.00451415, s
  bias_grad ( min -22.8996, max 0.170164, mean -8.6555e-10, variance 0.421778, skewness -
Component 14 : <Softmax>,
```

有一个全部数据集的目标函数值的总结log文件，它的progress vector是由第一步产生的，和帧正确率为：

```
LOG (nnet-train-frmshuff:main():nnet-train-frmshuff.cc:273) Done 34432 files, 21 with no
LOG (nnet-train-frmshuff:main():nnet-train-frmshuff.cc:282) AvgLoss: 1.19191 (Xent), [Avg
progress: [3.09478 1.92798 1.702 1.58763 1.49913 1.45936 1.40532 1.39672 1.355 1.34153 1.
FRAME_ACCURACY >> 65.6546% <<
```

log文件的结尾是CUDA的信息，CuMatrix::AddMatMat是矩阵乘法和大多数的花费时间如下：

```
[cudevice profile]
Destroy    23.0389s
AddVec    24.0874s
CuMatrixBase::CopyFromMat(from other CuMatrixBase)    29.5765s
AddVecToRows    29.7164s
CuVector::SetZero    37.7405s
DiffSigmoid    37.7669s
CuMatrix::Resize    41.8662s
FindRowMaxId    42.1923s
Sigmoid    48.6683s
CuVector::Resize    56.4445s
AddRowSumMat    75.0928s
CuMatrix::SetZero    86.5347s
CuMatrixBase::CopyFromMat(from CPU)    166.27s
AddMat    174.307s
AddMatMat    1922.11s
```

直接运行steps/nnet/train_scheduler.sh:

- 脚本train_scheduler.sh可以被train.sh调用，它允许覆盖默认的NN-input和NN-target streams，可以很便利的设置。
- 然而这个脚本假设所有的设置是正确的，仅仅对高级用户来说是合适的。
- 在直接调用前，我们非常建议去看脚本train_scheduler.sh是如何调用的。

Training tools

与nnet1相关的代码在目录src/nnetbin下，重要的工具如下：

- nnet-train-frmshuff.cc :最普遍使用的神经网络训练工具，执行一次迭代训练。
  - 过程如下：
    1. on-the-fly feature expansion by –feature-transform,
    2. per-frame shuffling of NN input-target pairs,
    3. mini-batch随机梯度下降法(SGD)训练,
  - 支持每一帧的目标函数(选项–objective-function):
    1. Xent :每一帧的交叉熵 $\mathcal{L}_{Xent}(\mathbf{t},\mathbf{y}) = -\sum_D t_d \log y_d$
    2. Mse: 每一帧的最小均方误差 $\mathcal{L}_{Mse}(\mathbf{t},\mathbf{y}) = \frac{1}{2}\sum_D (t_d - y_d)^2$ 这里的 t_d 表示目标向量 $\mathbf{t}$

的元素， y_d 是DNN输出向量 y 的元素，和D是DNN输出的维度。

- nnet-forward.cc : 通过神经网络计算前向数据，默认使用CPU
  - 看选项:
    - –apply-log :产生神经网络的对数输出(比如：得到对数后验概率)
    - –no-softmax :从模型中去掉soft-max层(decoding with pre-softmax values leads to the same lattices as with log-posteriors)
    - –class-frame-counts : counts to calculate log-priors, which get subtracted from the acoustic scores (在解码中的一个典型的技巧).
- rbm-train-cd1-frmshuff.cc :使用CD1来训练RBM，当内部调整学习率/冲量时需要训练数据好几次。
- nnet-train-mmi-sequential.cc : MMI / bMMI DNN training
- nnet-train-mpe-sequential.cc : MPE / sMBR DNN training

Other tools

- nnet-info.cc 打印关于神经网络的信息
- nnet-copy.cc 使用选项–binary=false把神经网络转换为ASCII格式，可以用来移除某些成分

Showing the network topology with nnet-info

接下来从nnet-info.cc里的打印信息显示"feature_transform"与steps/nnet/train.sh里的'–feat-type plain'相对应，它包含三个成分：

- `<Splice>` ：拼帧操作，对中间帧使用左右的上下文的帧 [-5 -4 -3 -2 -1 0 1 2 3 4 5]
- `<Addshift>` ：把特征变为零均值
- `<Rescale>` ：把特征变成单位方差
- 注意:我们从磁盘中读取低维特征，通过选项"feature_transform"扩展到高维特征，这样会节省磁盘空间和可读的吞吐量。

```
$ nnet-info exp/dnn5b_pretrain-dbn_dnn/final.feature_transform
num-components 3
input-dim 40
output-dim 440
number-of-parameters 0.00088 millions
component 1 : <Splice>, input-dim 40, output-dim 440,
frame_offsets [ -5 -4 -3 -2 -1 0 1 2 3 4 5 ]
component 2 : <AddShift>, input-dim 440, output-dim 440,
shift_data ( min -0.265986, max 0.387861, mean -0.00988686, variance 0.00884029, skew
component 3 : <Rescale>, input-dim 440, output-dim 440,
scale_data ( min 0.340899, max 1.04779, mean 0.838518, variance 0.0265105, skewness -
LOG (nnet-info:main():nnet-info.cc:57) Printed info about exp/dnn5b_pretrain-dbn_dnn/
```

接下来会打印6层的神经网络信息：

- 每一层是由2个成分构成，一般 `<AffineTransform>` 和一个非线性 `<Sigmoid>` 或者 `<Softmax>`
- 对于每一个 `<AffineTransform>` ，对于权重和偏移量来说，一些统计量将分开显示(min, max, mean, variance, ...)

```
$ nnet-info exp/dnn5b_pretrain-dbn_dnn/final.nnet
num-components 14
input-dim 440
output-dim 3370
number-of-parameters 28.7901 millions
component 1 : <AffineTransform>, input-dim 440, output-dim 2048,
linearity ( min -8.31865, max 12.6115, mean 6.19398e-05, variance 0.0480065, skewness
bias ( min -11.9908, max 3.94632, mean -5.23527, variance 1.52956, skewness 1.21429,
component 2 : <Sigmoid>, input-dim 2048, output-dim 2048,
component 3 : <AffineTransform>, input-dim 2048, output-dim 2048,
linearity ( min -2.85905, max 2.62576, mean -0.00995374, variance 0.0196688, skewness
bias ( min -18.4214, max 2.76041, mean -2.63403, variance 1.08654, skewness -1.94598,
component 4 : <Sigmoid>, input-dim 2048, output-dim 2048,
component 5 : <AffineTransform>, input-dim 2048, output-dim 2048,
linearity ( min -2.93331, max 3.39389, mean -0.00912637, variance 0.0164175, skewness
bias ( min -5.02961, max 2.63683, mean -3.36246, variance 0.861059, skewness 0.933722
component 6 : <Sigmoid>, input-dim 2048, output-dim 2048,
component 7 : <AffineTransform>, input-dim 2048, output-dim 2048,
linearity ( min -2.18591, max 2.53624, mean -0.00286483, variance 0.0120785, skewness
bias ( min -10.0615, max 3.87953, mean -3.52258, variance 1.25346, skewness 0.878727,
component 8 : <Sigmoid>, input-dim 2048, output-dim 2048,
component 9 : <AffineTransform>, input-dim 2048, output-dim 2048,
linearity ( min -2.3888, max 2.7677, mean -0.00210424, variance 0.0101205, skewness 0
bias ( min -5.40521, max 1.78146, mean -3.83588, variance 0.869442, skewness 1.60263,
component 10 : <Sigmoid>, input-dim 2048, output-dim 2048,
component 11 : <AffineTransform>, input-dim 2048, output-dim 2048,
linearity ( min -2.9244, max 3.0957, mean -0.00475199, variance 0.0112682, skewness 0
bias ( min -6.00325, max 1.89201, mean -3.96037, variance 0.847698, skewness 1.79783,
component 12 : <Sigmoid>, input-dim 2048, output-dim 2048,
component 13 : <AffineTransform>, input-dim 2048, output-dim 3370,
linearity ( min -2.0501, max 5.96146, mean 0.000392621, variance 0.0260072, skewness
bias ( min -0.563231, max 6.73992, mean 0.000585582, variance 0.095558, skewness 9.46
component 14 : <Softmax>, input-dim 3370, output-dim 3370,
LOG (nnet-info:main():nnet-info.cc:57) Printed info about exp/dnn5b_pretrain-dbn_dnn/
```

Advanced features

Frame-weighted training

调用带选项的teps/nnet/train.sh:

```
--frame-weights <weights-rspecifier>
```

这里的 `<weights-rspecifier>` 一般是表示每一帧权重的浮点型向量的ark文件。

- the weights are used to scale gradients computed on single frames, which is useful in

confidence-weighted semi-supervised training,
- or weights can be used to mask-out frames we don't want to train with by generating vectors composed of weights 0, 1

Training with external targets

调用带选项的steps/nnet/train.sh：

```
--labels <posterior-rspecifier> --num-tgt <dim-output>
```

这里的ali-dirs和lang-dir变成虚设的目录。"" 是一个典型的存储后验概率的ark文件，和""是神经网络输出的数目。这里后验概率没有概率的意思，它就是一个表示目标的数据类型，和这个目录值可以为任意的浮点数。

当每一帧使用一个单独的标签来训练时(比如：the 1-hot encoding)，你可以准备一个跟输入特征一样长度的整数向量的ark文件。 这个整数向量的元素对目标类的索引进行编码，如果神经网络的输出是这个索引，那么对应的目标值为1。这个整数向量可以使用ali-to-post.cc来转换为Posterior，和这个整数向量格式是非常简单的：

```
utt1 0 0 0 0 1 1 1 1 1 2 2 2 2 2 2 ... 9 9 9
utt2 0 0 0 0 0 3 3 3 3 3 3 2 2 2 2 ... 9 9 9
```

在多种非零目标的情况下，你可以使用ascii格式准备Posterior

- 每一个非零的目标值是由一对来编码的，这里的int32是神经网络输出的索引(从0开始)和float是目标值
- 每一帧(比如：datapoint)是由括号[...]里的值表示的，我们可以看到 数据对级联如下

```
utt1 [ 0 0.9991834 64 0.0008166544 ] [ 1 1 ] [ 0 1 ] [ 111 1 ] [ 0 1 ] [ 63 1 ] [ 0 1 ] [
```

在自编码的例子里egs/timit/s5/local/nnet/run_autoencoder.sh，外部的目标被使用。

Mean-Square-Error training

调用带选项的steps/nnet/train.sh：

```
--train-tool "nnet-train-frmshuff --objective-function=mse"
--proto-opts "--no-softmax --activation-type=<Tanh> --hid-bias-mean=0.0 --hid-bias-range=
```

最小均方误差训练是用在自编码的例子里，在脚本egs/timit/s5/local/nnet/run_autoencoder.sh

Training with tanh

调用带选项的steps/nnet/train.sh：

```
--proto-opts "--activation-type=<Tanh> --hid-bias-mean=0.0 --hid-bias-range=1.0"
```

tanh的最佳学习率一般小于sigmoid，通常最佳为0.00001。

Conversion of a DNN model between nnet1 -> nnet2

在Kaldi中，有二个DNN的例子，一个是Karel's (本页)和Dan's Dan's DNN implementation。这两个使用不兼容的DNN格式，这里是把Karel's DNN 转换为Dan的格式。

- 模型转换的样例脚本为：egs/rm/s5/local/run_dnn_convert_nnet2.sh,
- 模型转换的脚本为： steps/nnet2/convert_nnet1_to_nnet2.sh, 它是通过调用模型转换代码来实现的：nnet1-to-raw-nnet.cc
- 支持成分的列表可以看ConvertComponent。

The C++ code

nnet1的代码位于src/nnet，工具在src/nnetbin。它是根据 src/cudamatrix。

Neural network representation

神经网络是由称为成分的块构成的，其中一些简单的例子就是AffineTransform或者一个非线性Sigmoid, Softmax。一个单独的DNN层一般是由2个成分构成： AffineTransform和一个非线性。

表示神经网络的类：Nnet is holding a vector of Component pointers Nnet::components_. Nnet最重要的一些方法如下：

- Nnet::Propagate : 从输入传播到输出，这里需要保证为梯度计算准备的 per-component buffers
- Nnet::Backpropagate : 通过损失函数来后向传播，更新权重
- Nnet::Feedforward :传播，当使用两个翻动buffer来节省内存
- Nnet::SetTrainOptions : 设置训练的超参数(比如：学习率，冲量，L1, L2-cost)

为了调试，成分和buffers块是通过Nnet::GetComponent， Nnet::PropagateBuffer，Nnet::BackpropagateBuffer可以看到的。

Extending the network by a new component

当创建一个新成分，你需要使用下面2个接口中的一个：

1. Component : a building block, 不包含任何训练的参数 (可以看例子里的实现nnet-activation.h)
2. UpdatableComponent : Component的孩子, 一个带训练参数的 building block(可以看例子里的声线nnet-affine-transform.h)

最重要的一些虚函数的实现方法是(不是一个完整的列表):

- Component::PropagateFnc : 前向传播函数
- Component::BackpropagateFnc : 后向传播函数(使用一步的链式法则，由后向传播函数的导数乘以损失函数的导数)
- UpdatableComponent::Update :梯度计算和权重更新

使用一个新的成分来扩展神经网络的框架，你需要：

1. 定义一个新的成分入口Component::ComponentType
2. 在表Component::kMarkerMap定义新的一行
3. 添加一个"new Component"去调用像工厂一样的函数 Component::Read
4. 实现接口Component或者 UpdatableComponent的所有虚函数方法

dan的深度神经网络

翻译：wbgxx333@163.com

时间：2015年5月

Introduction

这个文档主要介绍在kaldi里Dan Povey版本的深度神经网络代码。对kaldi中所有的深度神经网络代码可以看Deep Neural Networks in Kaldi，对于Karel的版本，可以看Karel's DNN implementation.

这里(有些草草准备)关于DNN的介绍包括Looking at the scripts，Use of GPUs or CPUs，Tuning the neural network training和dnn2_preconditioning。

Looking at the scripts

如果你想理解神经网络训练的深层描述，最直接的方式就是去看脚本。在一些标准的例子里egs/rm/s5，egs/wsj/s5和 egs/swbd/s5b，最顶层的脚本是run.sh。这个脚本是调用脚本local/run_nnet2.sh。这对于Dan的代码来说是一个顶层的脚本。在local/run_nnet2.sh里，有许多不同的脚本来展示不同的例子，和之前最原始的那些脚本。除了运行脚本local/run_nnet2.sh,我们建议你应该花点时间去运行之前最原始的脚本。这里是一个p-norm网络(看this paper)。

Top-level training script

注意：之前最顶层的训练脚本steps/nnet2/train_pnorm.sh，现在我们已经放弃了这个脚本，和你现在应该使用脚本steps/nnet2/train_pnorm_fast.sh。这个脚本可以并行训练，接下来将解释这个。

Input features to the neural net.

神经网络的输入特征在某种程度上是可以配置的，但是默认的是采用通常的步骤，即在语音识别中GMM模型的自适应特征是MFCC(spliced)+LDA+MLLT+fMLLR，40维的特征（译者注：就是39维MFCC+1维Pitch）。神经网络使用这些特征的一个窗，默认中间帧的左右两边各3帧，总共7帧。由于神经网络很难从相关的特征中学习，这里需要使用一个固定的变换来去掉这种相关性，也就是这个变换乘以40*7维的特征。使用这个变换是由训练脚本来完成；可以去调用脚本steps/nnet2/get_lda.sh。主要根据这篇文章this paper来做的，但是这个变换

的真实代码不是真正的LDA：默认情况下，它更像LDA变换的一种非降维形式，由于类间方差很小，可以对输出特征的方差的维度做一个降维(这个没有被发表；可以去看代码)。这个脚本支持的其他类型的特征都是未处理的特征，比如MFCC特征；这些可以通过选–feat-type来设置，并通过选项–egs-opts和–lda-opts传递到脚本get_egs.sh和get_lda.sh。

既然在脚本里寻找选项，最好的方式就是仅仅搜索用下划线代替破折号的选项名：在这种情况下，就变为feat_type，egs_opts，和 lda_opts。脚本utils/parse_options.sh自动解释那些设置对应变量的命令行参数。

Dumping training examples to disk

假设最顶层的脚本(比如steps/nnet2/train_pnorm.sh)创建一个模型，在exp/nnet5d/。这个脚本做的第一件事情就是去调用steps/nnet2/get_egs.sh。这就放置了很多数据在exp/nnet5d/egs/。这个与输入的帧层随机化有关，这个对于随机梯度下降训练是需要的。我们仅仅做一次随机化，以至于在真正的训练中，我们可以顺序的访问这个数据。也就是说，每一轮迭代，我们有必要以相同的顺序访问数据；意味着对磁盘和网络是友好的。(事实上我们可以每次迭代时使用一个不同的种子和使用一小段内存来做随机化，但是这仅仅在本地上改变了顺序)。

如果你去看exp/nnet5d/egs/，你会看到很多文件名为egs.1.1.ark， egs.1.2.ark等等。这些archives文件含有一个叫NnetTrainingExample的许多例子。对于一个单独的帧，这个类含有标签信息，和对这帧来说，有足够多的特征窗来做神经网络计算。除了在神经网络外部做帧切分，神经网络训练代码有一个时间的概念和需要知道上下文特征是多少(看函数RightContext()和LeftContext())。在这个文件名中的2个整数索引分别是任务索引和迭代索引。任务索引对应我们有多少个并行的任务。例如如果我们使用CPUs运行，用16台机器并行(这里每个机器有很多不相关的线程)，然后任务索引就是从1到16，或者你使用GPUs，使用8个GPUs并行，然后任务索引就是从1到8。迭代索引的范围主要根据你有多少数据来决定的。默认情况下，每个archive有200,000个样本数。迭代索引的数量是由你有多少数据和你有多少任务来决定的。一般很多轮(比如：20)跑一次训练，和每一轮我们需要做很多次迭代(对于像rm这样的小数据量就是1，对于大的数据集可以为10)。

目录exp/nnet5d/egs/也包含一些其他的文件：iters_per_epoch， num_jobs_nnet和sample_per_iter，这些文件里包含一些数；在一个rm数据集上的例子里，它们分别为1，16和85493。它也包含valid_diagnostic.egs，这是用来诊断的，在held-out集上的一些小例子(看e.g. exp/nnet5d/log/compute_prob_valid.*.log)，和 *train_diagnostic.egs*，是除了*held-out*外的*valid_diagnostic.egs*；为了诊断，可以看*exp/nnet5d/log/compute_prob_valid.*.log。文件combine.egs是在训练的结束用来计算神经网络的联合权重的一个稍微大点的训练数据集。

Neural net initialization

我们使用一个单隐含层来初始化这个神经网络；我们将在稍后的训练中增加隐含层的数量到一个设定的数字(通常是2到5这个范围)。这个脚本创建了一个叫exp/nnet4d/nnet.config的config文件，这将传递给初始化模型的程序，名字叫nnet-am-init。对于rm数据集，对于p-norm部分的一些config文件的例子如下：

```
SpliceComponent input-dim=40 left-context=4 right-context=4 const-component-dim=0
FixedAffineComponent matrix=exp/nnet4d/lda.mat
AffineComponentPreconditionedOnline input-dim=360 output-dim=1000 alpha=4.0 num-samples-h
PnormComponent input-dim=1000 output-dim=200 p=2
NormalizeComponent dim=200
AffineComponentPreconditionedOnline input-dim=200 output-dim=1475 alpha=4.0 num-samples-h
SoftmaxComponent dim=1475
```

FixedAffineComponent是我们之前提到的像LDA去相关变换。AffineComponentPreconditionedOnline是AffineComponent的改良。AffineComponent是由标准的神经网络(权重矩阵和偏移项)构成的，在标准的随机梯度下降法训练中使用。AffineComponentPreconditionedOnline就是AffineComponent，但是训练步骤使用的不仅仅是一个单独的全局学习率，而是一个矩阵值学习率来预处理梯度下降。我们接下来将介绍更多(看dnn2_preconditioning)。PnormComponent是非线性的；对于一个更传统的神经网络，将使用TanhComponent来代替。 instead。NormalizeComponent是我们用来稳定p-norm网络训练而添加的。它也是固定的，非训练的非线性，但是它的作用不是一个单独的激活函数，而是对于一个单独的帧的整个向量，来重新归一化他们的单位标准偏差。SoftmaxComponent是最终的非线性，它是在输出上产生归一化的概率。

这个脚本也产生一个名叫hidden.config的config文件，这对应到我们添加的和我们介绍的一个新的隐含层；这个例子如下：

```
AffineComponentPreconditionedOnline input-dim=200 output-dim=1000 alpha=4.0 num-samples-h
PnormComponent input-dim=1000 output-dim=200 p=2
NormalizeComponent dim=200
```

直到第一对训练后，它才被使用。

脚本接下来做的就是调用nnet-train-transitions。它将计算在解码(似乎与神经网络本身无关)中使用的HMMs的转移概率，和计算这些目标(成千上万个上下文独立的状态)的先验概率。然后，当我们解码时，我们用神经网络得到的后验概率除以这些先验得到"pseudo-likelihoods"；这个比原来的后验概率更加与HMM框架兼容。

Neural net training

接下来我们将来到最主要的训练阶段。这是一个在迭代计算器x，范围从0到num_iters - 1的一个循环。迭代的次数num_iters是我们训练的轮数乘以每一轮我们迭代的次数。我们训练的轮数是num_epochs(default: 15)加上 num_epochs_extra(default:5)的和。这个与学习率进程有关：默认的，我们从最开始的学习率initial_learning_rate(default: 0.04)下降到15轮后的final_learning_rate (default: 0.004)和在后面的5轮保持为常数。每一轮的迭代次数存储在一个文件叫egs/nnet5d/egs/iters_per_epoch中；它取决于我们的数据量的大小和我们并行运行时的训练任务数，和一般是从0到几十范围内。

在每一次迭代中，我们做的第一件事情就是计算一些诊断：在训练和交叉集上的目标函数(对于第十次迭代，可以看egs/nnet5d/log/compute_prob_valid.10.log和egs/nnet5d/log/compute_prob_train.10.log)。在文件egs/nnet5d/log/progress.10.log中你将看到每一层参数是改变的，和训练集目标函数的改变对每一层的改变影响有多大。

下面是在某个特定目录中一个诊断的例子

```
grep LOG exp/nnet4d/log/compute_prob_*.10.log
exp/nnet4d/log/compute_prob_train.10.log:LOG<snip> Saw 4000 examples, average probability
exp/nnet4d/log/compute_prob_valid.10.log:LOG<snip> Saw 4000 examples, average probability
```

你可以看到在训练集(-0.89)的目标函数比交叉集(-1.16)上的要好。这是一个交叉熵，表示每一帧在这个正确类上的平均log概率。在训练集和交叉集上的目标函数相差很大是正常的，因为神经网络有一个很强大的学习能力：对于一个调好参的系统上仅仅几个小时的数据，它们会把它们一分为二(但是当你含有很多训练数据使就差点)。 如果你在训练目标函数上添加更多的参数将会改善，但是交叉集上的性能将下降。然而，对交叉集目标函数上调参不是一个好主意，因为它会导致系统含有很少的参数。如果对Word Error Rates添加参数，即使它在一定程度上降低了交叉集上的性能。

在这样的一个文件exp/nnet4d/log/progress.10.log里，你将寻找更多的诊断：

```
LOG <snip> Total diff per component is  [ 0.00133411 0.0020857 0.00218908 ]
LOG <snip> Parameter differences per layer are  [ 0.925833 1.03782 0.877185 ]
LOG <snip> Relative parameter differences per layer are  [ 0.016644 0.0175719 0.00496279
```

第一行"Total diff per component"在训练集上目标函数的下降是由其他层的共同影响，和其他行中说明的是不同层的参数改变有多大。

主要训练任务的log文件可以在exp/nnet5a/log/train...log被找到。第一个索引就是迭代的次数和第二个索引就是我们运行的并行任务数，4或者16(这个数字是由–num-jobs-nnet参数传递到脚本的)。接下来是其中一个训练任务的例子：

```
#> cat exp/nnet4d/log/train.10.1.log
# Running on a11
# Started at Sat Mar 15 16:32:08 EDT 2014
# nnet-shuffle-egs --buffer-size=5000 --srand=10 ark:exp/nnet4d/egs/egs.1.0.ark ark:- | \
  nnet-train-parallel --num-threads=16 --minibatch-size=128 --srand=10 exp/nnet4d/10.mdl
<snip>
LOG (nnet-shuffle-egs:main():nnet-shuffle-egs.cc:100) Shuffled order of 79100 neural-netw
 training examples using a buffer (partial randomization)
LOG (nnet-train-parallel:DoBackpropParallel():nnet-update-parallel.cc:256) Did backprop o
  79100 examples, average log-prob per frame is -1.4309
LOG (nnet-train-parallel:main():nnet-train-parallel.cc:104) Finished training, processed
   79100 training examples (weighted).  Wrote model to exp/nnet4d/11.1.mdl
# Accounting: time=18 threads=16
# Finished at Sat Mar 15 16:32:26 EDT 2014 with status 0
```

这个特定的任务是没用使用GPU，而是使用16 CPU并行运行的，和仅仅使用18秒完成了。这里最主要的任务就是nnet-train-parallel，它是做随机梯度下降的，与Hogwild(例如：没有锁)有点相似的并行化, 每个线程使用的块(minibatch)大小为128。这个模型的输出是11.1.mdl。在exp/nnet4d/log/average.10.log中，你可以看到一个程序叫nnet-am-average的输出log文件，这个程序为这次迭代对所有的SGD训练的模型做平均化。 它通过我们的学习率进度来修改学习率，这个是指数级下降的(具体可以看论文"An Empirical study of learning rates in deep neural networks for speech recognition" by Andrew Senior et. al.，你会发现对于语音识别这个非常有效)。注意：在我们例子里的最后二层中，我们使用tanh这个函数，学习率需要减半；看脚本train_tanh.sh里的选项–final-learning-rate-factor。

对于几十万的样本来说，最基本的并行方法就是，在不同的任务中使用不同的数据来使用随机梯度下降法来训练，然后对这些模型进行平均化。在这个参数上目标函数是非凸的，你也许会惊讶这个是可行的，但是从经验上看，凸函数这个性质在这里似乎不是一个问题。注意：我们接下来描述的去做"preconditioned update"，这个看起来更重要点；我们通过大量的实验证明，这个对于我们并行化方法的成功至关重要。同时也要注意在最近的训练脚本(train_norm_fast.sh和train_tanh_fast.sh)，我们没有做并行化和对迭代做平均，因为这里或者仅仅是初始化模型或者仅仅添加新的一层。这是因为在这些情况下，由于缺少凸性，做平均化有时候一点好处都没有(例如，在给定平均化模型的目标函数比单独目标函数的平均化要)。

Final model combination

如果你继续往里面看，举例来说，exp/nnet4d/log/combine.log，你会看到一个最终的神经网络被创建，名字叫"final.mdl"。这是联合最后N次迭代的模型的参数得到的，这里的N对应脚本里的参数–num-iters-final(默认地：20)。最基本的想法就是通过对若干次迭代上做平均，这样可以减少估计的偏差。我们不能很容易的证明这个就比仅仅取最终的模型好(因为这是一个非凸问题)，但是在实践中就是这样的。事实上，"combine.log"不仅仅是对这些参数取平均。这里使用训练数据样本(在这种情况下，从exp/nnet4d/egs/combine.egs得到)的一个子集来优化这些权重， 这个没有约束为正的。这里的目标函数是在这个数据集上的正常的目录函数(log-

probability)，和优化方法是L-BFGS，这里我们就不赘述这个特殊的预处理方法。对每个成分和每次迭代都有单独的权重，所以在这种情况下我们有学习(20*3=60)的权重。在这种方法的最初始版本，我们使用交叉集数据来估计这个参数，但是我们发现为了这个目的使用训练数据的一个随机子集，它可能表现的很好。

```
#> cat exp/nnet4d/log/combine.log
<snip>
    Scale parameters are  [
  -0.109349 -0.365521 -0.760345
  0.124764 -0.142875 -1.02651
  0.117608 0.334453 -0.762045
  -0.186654 -0.286753 -0.522608
  -0.697463 0.0842729 -0.274787
  -0.0995975 -0.102453 -0.154562
  -0.141524 -0.445594 -0.134846
  -0.429088 -1.86144 -0.165885
  0.152729 0.380491 0.212379
  0.178501 -0.0663124 0.183646
  0.111049 0.223023 0.51741
  0.34404 0.437391 0.666507
  0.710299 0.737166 1.0455
  0.859282 1.9126 1.97164 ]

LOG <snip> Combining nnets, objf per frame changed from -1.05681 to -0.989872
LOG <snip> Finished combining neural nets, wrote model to exp/nnet4a2/final.mdl
```

联合权重作为一个矩阵被打印出来，它的行索引对应迭代的次数，列索引对应层数。你将看到，联合权重在后面的迭代中是正的，在之前的迭代中是负的，我们可以解释为尝试对模型在这个方向上做更深的研究。我们使用训练数据集，而不是交叉数据集，因为我们发现训练数据集效果更好，尽管使用交叉数据集是一个更自然的想法；我们认为这个原因可能与在语音识别中的一个不准确的"dividing-by-the prior"归一化有关。

Mixing-up

如果使用程nnet-am-info来打印关于exp/nnet4d/final.mdl的信息，你将看到在输出层之前有一个大小为4000的层，这个输出层的大小是1483，因为决策树有1483个分支：

```
#> nnet-am-info exp/nnet4d/final.mdl
num-components 11
num-updatable-components 3
left-context 4
right-context 4
input-dim 40
output-dim 1483
parameter-dim 1366000
component 0 : SpliceComponent, input-dim=40, output-dim=360, context=4/4
component 1 : FixedAffineComponent, input-dim=360, output-dim=360, linear-params-stddev=0
component 2 : AffineComponentPreconditioned, input-dim=360, output-dim=1000, linear-param
component 3 : PnormComponent, input-dim = 1000, output-dim = 200, p = 2
component 4 : NormalizeComponent, input-dim=200, output-dim=200
component 5 : AffineComponentPreconditioned, input-dim=200, output-dim=1000, linear-param
component 6 : PnormComponent, input-dim = 1000, output-dim = 200, p = 2
component 7 : NormalizeComponent, input-dim=200, output-dim=200
component 8 : AffineComponentPreconditioned, input-dim=200, output-dim=4000, linear-param
component 9 : SoftmaxComponent, input-dim=4000, output-dim=4000
component 10 : SumGroupComponent, input-dim=4000, output-dim=1483
prior dimension: 1483, prior sum: 1, prior min: 7.96841e-05
LOG (nnet-am-info:main():nnet-am-info.cc:60) Printed info about baseline/exp/nnet4d/final
```

softmax层的维度是4000，然后由SumGroupComponent减少到1483.你使用命令nnet-am-copy把它转化为文本格式，然后你可以看到一些信息：

```
#> nnet-am-copy --binary=false baseline/exp/nnet4d/final.mdl - | grep SumGroup
nnet-am-copy --binary=false baseline/exp/nnet4d/final.mdl -
<SumGroupComponent> <Sizes> [ 6 3 3 3 2 3 3 3 2 3 2 2 3 3 3 3 2 3 3 3 3 \
3 3 4 2 1 2 3 3 3 2 2 2 3 2 2 3 3 3 3 2 4 2 3 2 3 3 3 4 2 2 3 3 2 4 3 3 \
<snip>
4 3 3 2 3 3 2 2 2 3 3 3 3 3 1 2 3 1 3 2 ]
```

softmax成分产生比我们需要更多的后验概率(4000 instead of 1483)，这些小组的后验概率求和产生输出的维度是1483，其中小组的大小范围在这里例子里是1到6。我们把"mixing up"和语音识别里的混合高斯模型训练中的mix up做类比，这里我们把高斯分成2份和影响均值。这种情况下，我们把最终权重矩阵的行分成2份和影响它们。这种额外的目标在训练过程中将增加一半。相关的log文件如下：

```
cat exp/nnet4d/log/mix_up.31.log
# Running on a11
# Started at Sat Mar 15 15:00:23 EDT 2014
# nnet-am-mixup --min-count=10 --num-mixtures=4000 exp/nnet4d/32.mdl exp/nnet4d/32.mdl
nnet-am-mixup --min-count=10 --num-mixtures=4000 exp/nnet4d/32.mdl exp/nnet4d/32.mdl
LOG (nnet-am-mixup:GiveNnetCorrectTopology():mixup-nnet.cc:46) Adding SumGroupComponent t
LOG (nnet-am-mixup:MixUp():mixup-nnet.cc:214) Mixed up from dimension of 1483 to 4000 in
LOG (nnet-am-mixup:main():nnet-am-mixup.cc:77) Mixed up neural net from exp/nnet4d/32.mdl
# Accounting: time=0 threads=1
# Finished at Sat Mar 15 15:00:23 EDT 2014 with status 0
```

Model "shrinking" and "fixing"

事实上没有在最原始例子里使用的p-norm网络上使用"Shrinking"和"fixing"，但是在使用脚本steps/nnet2/train_tanh.sh训练的神经网络中需要使用"Shrinking"和"fixing"，或者任何含有sigmoid激活函数的其他网络中也要使用。我们尝试解决的是发生在这些类型激活函数里的问题，这些神经元在过多的训练数据上将过饱和(也就是说，it gets outside the part of the activation that has a substantial slope)和训练将变的很慢。

对于shrinking，我们来看其中的一个log文件，首先：

```
#> cat exp/nnet4c/log/shrink.10.log
# Running on a14
# Started at Sat Mar 15 14:25:43 EDT 2014
# nnet-subset-egs --n=2000 --randomize-order=true --srand=10 ark:exp/nnet4c/egs/train_dia
  nnet-combine-fast --use-gpu=no --num-threads=16 --verbose=3 --minibatch-size=125 exp/nn
  ark:- exp/nnet4c/11.mdl
<snip>
LOG <snip> Scale parameters are  [
  0.976785 1.044 1.1043 ]
LOG <snip> Combining nnets, objf per frame changed from -1.01129 to -1.00195
LOG <snip> Finished combining neural nets, wrote model to exp/nnet4c/11.mdl
```

当使用nnet-combine-fast时，但是仅仅把它作为神经网络的输入，所以我们可以优化的唯一一件事情就是神经网络在不同层数的参数的尺度。这次尺度都接近于1，和有些大于1，所以在这种情况下， *shrinking*也许是用词不当。那些"shrinking"是有用的情况里，但是可能在这种情况下也没有多大区别。

接下来，我们看一下"fixing"的log文件，当我们不做"shrinking"时，每次迭代都需要做'fixing'：

```
#> cat exp/nnet4c/log/fix.1.log
nnet-am-fix exp/nnet4c/2.mdl exp/nnet4c/2.mdl
LOG (nnet-am-fix:FixNnet():nnet-fix.cc:94) For layer 2, decreased parameters for 0 indexe
   and increased them for 0 out of a total of 375
LOG (nnet-am-fix:FixNnet():nnet-fix.cc:94) For layer 4, decreased parameters for 1 indexe
   and increased them for 0 out of a total of 375
LOG (nnet-am-fix:main():nnet-am-fix.cc:82) Copied neural net from exp/nnet4c/2.mdl to exp
```

这里做的就是根据训练数据，计算tanh激活函数的导数的平均值。对于tanh来说，在任何数据点上这个导数都不能超过1.0。对于某个特定的神经元来说，如果它的平均值比这个值小很多(默认的我们设置门限值为0.1)，也就是我们过饱和了和我们通过对这个输入的神经元降低2倍来补偿权重和偏移项。就像你在log文件里看到的那样，这个仅仅在这次迭代某个神经元上发生，意思也就是说对于这次特定的运行，这不是一个很大的问(如果你使用很大的学习率，它将经常发生)。

Use of GPUs or CPUs

无论是GPU还是CPU，这个代码都可以保证平等透明的训练。注意如果你想使用GPU来运行，你必须使用GPU支持的编译，也就是说在src/下，你需要在一个含有 NVidia CUDA toolkit 的机器上运行"configure"和"make"(也就是，在这台机器上，命令"nvcc"可以被执行)。如果kaldi是使用GPU支持的编译，神经网络训练代码是可以使用GPU来训练的。你可使用命令"ldd"来确定kaldi是否使用GPU编译的，和检查libcublas是否编译过，例如：

```
src#> ldd nnet2bin/nnet-train-simple | grep cu
libcublas.so.4 => /home/dpovey/libs/libcublas.so.4 (0x00007f1fa135e000)
libcudart.so.4 => /home/dpovey/libs/libcudart.so.4 (0x00007f1fa1100000)
```

当使用GPU来训练时，你将看到一些像 train...log的文件:

```
LOG (nnet-train-simple:IsComputeExclusive():cu-device.cc:209) CUDA setup operating \
    under Compute Exclusive Mode.
LOG (nnet-train-simple:FinalizeActiveGpu():cu-device.cc:174) The active GPU is [0]: \
 Tesla K10.G2.8GB    free:3516M, used:66M, total:3583M, free/total:0.981389 version 3.0
```

一些命令行程序有一个选项–use-gpu，这个选项可以取的值为"yes"， "no"或者"optional"，和意思就是你是否使用GPU(如果你设定为"optional"，如果仅仅有一个GPU可用，那么将使用GPU)。 但是事实上我们在脚本里不使用这种机制，因为对于GPU和CPU训练来说，我们有两种不同的代码。CPU版本的代码是nnet-train-parallel，和因为它支持多线程，所以它这么命名。当我们使用一个CPU时，我们将使用16个线程。这就是去做没有任何锁的多核随机梯度下降，有时候我们可以把它看成是Hogwild的一种形式。顺便，当使用多线程更新时，我们建议minibatch的大小不超过128，因为这个会导致不稳定性。 We consider that "effective

minibatch size" as equal to the minibatch size times the number of threads, and if this gets too large the updates can diverge. Note that we have formulated the stochastic gradient descent so that the gradients get summed over the members of the minibatch, not averaged. Also note that the only reason why we can't just use nnet-train-parallel with one thread for GPU-based training is that nnet-train-parallel uses two threads even if configured with –num-threads=1 (because one thread is dedicated to I/O), and CUDA does not work easily with multi-threaded programs because the GPU context is tied to a single thread.

Switching between GPU and CPU use

当你借助像train_tanh.sh和train_pnorm.sh脚本来在使用CPU和GPU之间切换时，有一些你不得不改的步骤(也许这不是理想的)。这些程序有一个选项–parallel-opts， 它是由传递到queue.pl(或者类似的脚本)里的额外的标示构成的。这里假设queue.pl是借助GridEngine和参数将传递到GridEngine。 –parallel-opts的默认值是使用一个16线程的CPU，和就是"-pe smp 16 -l ram_free=1G,mem_free=1G"。这仅仅影响我们从queue里获取哪些资源，和不影响真实运行的那些脚本；我们将不得不告诉脚本事实上使用的是16个线程，是通过选项–num-threads(默认的是16)。 选项"ram_free=1G" 也许全部的queue无关，为了内存使用，我们人工把它作为一个资源添加到我们的queue里；如果你的电脑没有这些资源，你可以删除它。默认的设置是使用16个线程的CPU；如果你想使用一个GPU，你就得借助脚本的选项像

```
--num-threads 1 --parallel-opts "-l gpu=1"
```

这里我们强调的是，选项"gpu=1"仅仅表示我们在一个特定的集群上借助GPU，和其他的集群将不同，因为一个GPU不能够认为GridEngine– queues将被使用者以不同的方式编译。 Basically the string needs to be whatever options you need to give to "qstat" so that it will request a GPU. 如果所有的都使用一个没有GridEngine的单机来运行，你仅仅使用 run.pl来运行任务，然后parallel-opts仅仅是一个空的字符串。如果你借助脚本并使用–num-threads=1，它会调用nnet-train-simple，当你使用GPUs编译时，默认地将使用一个GPU。如果 –num-threads超过1，它将调用nnet-train-parallel，而它不是使用一个GPU。

Tuning the number of jobs

接下来我们描述如何在CPU训练和GPU训练中切换的一些重要点。你也许注意到一些样例脚本中(比如：对这一对脚本做对比local/nnet2/run_4c.sh和local/nnet2/run_4c_gpu.sh)，选项–num-jobs-nnet的值在GPU脚本和CPU脚本中是不一样的，比如在CPU版本是8，GPU版本是4。和–minibatch-size有时在这两个版本中也不同，举例来说， 在GPU中是512，在CPU中是128，学习率有时也不同。

这里将解释这些不同的原因。首先，不考虑minibatch的大小。你应该知道我们的SGD中梯度的计算不是通过对minbatch平均相加的。 在我们看来，当minbatch大小改变时，就需要去最大限度地改变学习率。一般而言，矩阵相乘在大的minbatch，比如512时比较快(每个样本)。

我们使用的预处理方法，就是接下来讨论的Preconditioned Stochastic GradientDescent，当使用大点的minbatch效果会好点。所以当minbatch大小是512或者1024时，训练会更快收敛。然而，这里有个限制，minbatch的大小与SGD更新的不稳定相关(和参数起伏不定和不可控)。如果minbatch太大，更新会不稳定。一旦这种不稳定性变大，它将受到选项–max-change的限制，对于每个minbatch，它会影响参数允许的改变范围，所以一般不会使得训练集的概率一直到负无穷大，但是它们也许会降得特别快。如果你在compute_prob_train.*.log里看到目标函数低于我们系统里叶子节点数目的负自然对数(一般大约为-7，你也可以在compute_prob_train.0.log看到这个值)，意思就是神经网络比chance要差，这是因为不稳定性导致的一些设置。这个问题的解决方法一般是降低学习率或者minibatch大小。

接下来我们讨论多线程更新时的不稳定的问题。当我们使用多线程更新时，为了稳定性这个目的，对minbatch的大小乘以线程数，所以我们让minbatch的大小低于设定的值。当我们使用CPU多线程训练时，一般minbatch的大小为128。(我们应该注意到，考虑到多线程CPU更新，我们尝试做单线程训练和允许使用BLAS来实现使用多线程，但是我们发现在相同的参数上，它比单线程独立地做SGD要快)。

接下来，不考虑选项–num-jobs-nnet：相对于GPU的例子，CPU的例子一般使用更多的任务数(8或者16)。原因很简单，因为在运行样例时，我们不想有CPU那么多的GPU。GPU训练一般比CPU快20%-50%。我们感觉我们使用更少的任务来达到相同的训练时间。但是一般情况任务数是独立的，无论我们使用CPU或者GPU。

最后一个修改就是学习率(选项–initial-learning-rate和 –final-learning-rate)，和这个与任务数有关(–num-jobs-nnet)。 一般而言，如果我们增加任务数，我们也应该以相同的因子来增加学习率。 因为这里的并行方法是对并行SGD跑出来的神经网络求平均得到的，我们把整个学习过程中的有效学习率等价的看成学习率除以任务数。所以当把任务数加倍时，如果我们也加倍学习率，但我们保持的有效学习率不变。但是这里有一些限制。如果学习率变高将导致不稳定，参数的更新将会发散。因此当最初的学习率变高，我们警惕它会增加很快。增加多少的依赖于我们的例子。

Tuning the neural network training

一般而言，当调整神经网络训练参数时，你应该从样例目录egs//local/nnet2/里的一些脚本开始，以某个方式来改变参数。这里假设你已经运行了train_tanh.sh或者train_pnorm.sh。

Number of parameters (hidden layers and hidden layer size)

最重要的参数就是隐含层的层数(–num-hidden-layers)。 对于tanh网络来说，它一般在2到5之间(如果数据越多，它的值越大)，和对于p-norm网络来说，它的值在2和3或者4之间。当修改隐含层的层数时，我们一般把隐含层节点数固定(512，或者1024，或者其他)。

对于tanh网络，你也可以改变隐含层的维度–hidden-layer-dim ；它是隐含层节点的数目。如果有更多的数据，这个值一般会越大，但是需要注意的是当它增大时，参数的数目将以2倍增长，所以你添加更多的数据时，它的增长应该小于0.5次方(比如：如果你的数据增加了10倍，你隐含层大小增加1倍将是有意义的)。我们不会用到2048或者更大的。对于一个很大的网络，我们一般使用1024个隐含层节点。

对于p-norm网络来说，它是没有参数–hidden-layer-dim；取代它的是其他两个参数，–pnorm-output-dim和–pnorm-input-dim。它们各自的默认值为(3000, 300)。 output-dim必须是input-dim的一个整数倍；我们一般使用时5倍或者10倍。这个会影响参数的数目；如果是大数据集，这个值将变大，但是跟tanh网络的隐含层大小一样，它的增大跟数据量的多少仅仅是缓慢变化的。

与参数的数目相关的其他选项是–mix-up。 它负责为每个叶子节点提供多重虚拟的目标，最后的softmax层的大小是随着决策树里的叶子节点的数目而变化(叶子节点的数目可以通过am-info函数，对神经网络训练里的输入目录里的final.mdl来得到)；通常它的值为几千。参数 –mix-up一般是叶子节点的2倍左右，但是一般错误率对其不是很敏感。

Learning rates

另一个重要的参数就是学习率。这里有两个主要的参数：–initial-learning-rate和–final-learning-rate。 它们默认的值分别为0.04和0.004。我们一般设定最终的学习率是最初学习率的五分之一或者十分之一。默认值0.04和0.004仅仅对小数据集是合适的，比如rm数据集，大约3个小时。如果数据量越大，你将训练更长时间，所以不需要这么大的学习率。 对于几百小时的数据集来说，这个学习率的十分之一也许是合适的。下面将介绍学习率和任务数之间的关系。

如果不画出目标函数在时间轴上的曲线图，我们很难去判断学习率是太高还是太低。如果学习率很大，目标函数很快就收敛，但是得不到一个很好的目标函数值(就像被噪声梯度隐藏起来了)。但你可能使参数波动，将会得到一个非常差的目标函数值(如果minbatch的大小太大或者你使用多线程，这种情况将最可能发生)。如果学习率太低，目标函数将更新缓慢和将花费很长的时间到达最值。

你可能不需要调整的一个学习率参数是在脚本train_tanh.sh里的一个配置值–final-learning-rate-factor，默认设置为0.5。在最后2层中，将使用给定学习率的一半(比如：softmax层和最后的隐含层的参数)。脚本train_pnorm.sh script支持一个相同的配置值 –soft-max-learning-rate-factor，它将影响最后的softmax层之前的参数，但是它的默认值为1.0。

Minibatch size

另一个可调整的参数就是minibatch的大小。我们一般使用2的次方，典型的有128，256或者512。一般一个大点的minbatch会更有效，因为它可以更好的与矩阵相乘代码里使用到的优化进行交互，尤其在使用GPU时，但是它如果太大(和如果学习率太大)，在更新时会导致不稳

定。对于基于CPU训练时，使用多线程的Hogwild!形式更新，如果minbatch的大小太大，更新将变不稳定。对于多线程的CPU训练时，minbatch的大小一般为128；对于基于GPU的训练，minbatch的大小为512。这就不需要再去调整了。我们需要注意的是，minibatch的大小跟接下来讨论的–max-change选项有关，如果minbatch越大，也就意味着–max-change越大。

Max-change

在脚本train_tanh.sh和train_pnorm.sh里有个选项–max-change，这个值将传递给包含权重矩阵的那些成分的初始化(它们是类型AffineComponent或者AffineComponentPreconditioned)。–max-change 选项限制了每个minibatch允许多少参数来修改，以l2范数来衡量，比如这个矩阵表示在任何给定的minibatch，任何给定层的参数的改变都不能超过这个值。为了做到这一点，我们使用一个临时矩阵来存储这些参数的变化。这是浪费的，因为事实上这个界是这个minbatch所有成员的l2范数贡献之和。如果这个超过了"max-change"，我们就为这个minbatch的学习率乘以一个常数，以使得它不超过这个限制。如果选项max-change约束被激活，你将在train.*.log里看到如下y的一些信息：

```
LOG <snip> Limiting step size to 40 using scaling factor 0.189877, for component index 8
LOG <snip> Limiting step size to 40 using scaling factor 0.188353, for component index 8
```

(事实上，这个因子比正常的要小——打印出来的这些因子通常是接近这个。 也许对于这个特定的迭代，学习率太高了。–max-change是一个故障安全机制，如果学习率太高(或者minibatch大小太大)，它将不会导致不稳定。–max-change可以减慢训练中学习过快的问题，特别是对最后的一层或者2层；稍后再训练过程中，这个约束将不再起作用，和在训练的结束中你不需要去看logs文件里的这些信息。这个参数将不再是重要的。如果minbatch的大小是512时，我们通常设置它为40(比如：当使用GPU时)，和如果minbatch的大小为128时，它为10(比如：当使用CPU时)。 这个是有意义的，因为限制这个数量跟minbatch样本的数量是成正比的。

Number of epochs, etc.

训练的迭代次数这里有两个配置变量：–num-epochs (默认为15)，和 –num-epochs-extra (默认为5)。 训练的迭代次数–num-epochs的准则就是从–initial-learning-rate从几何学来减少学习率到–final-learning-rate，和对于迭代次数为 –num-epochs-extra，我们最后保持–final-learning不变。一般情况下，改变迭代次数是没必要的，除了有时是小数据集时，我们训练的迭代次数为20+5，而不是15+5。同时，如果数据量很大，和我们的计算环境不是很强大， 你也许通过减少训练迭代次数来节省时间。这样也许会轻微地降低最终性能。

有时候，这个也与参数–num-iters-final有关。这个决定了我们最终的模型组合的迭代次数，在训练的最后时(看Final model combination)。我们坚信这不是一个非常重要的参数。

Feature splicing width.

这里有个选项–splice-width，默认的值为4，它将影响我们分配多少帧的特征给输入。这个会影响神经网络的初始化，和样例的生成。这个值4意思是在这个中间帧的左边和右边各4帧，总共9帧数据。 参数–splice-width事实上是一个相当重要的参数，但是对于正常的全处理特征来说(比如：从MFCC+splice+LDA+MLLT+fMLLR里得到的40维特征)，4就是一个最佳值。注意LDA+MLLT特征就是根据中间帧的每一边都是3或者4帧，也就是表示整个有效的声学上下文的神经网络可以看到每一边都是7或者8帧。如果你使用"raw" MFCC或者log-filterbank-energy特征(看脚本get_egs.sh和get_lda.sh里的选项"--feat-type raw")， 你也许可以把–splice-width设置大一点，比如为5或者6。

许多人问我们，为什么使用超过4帧的上下文会不好？如果我们的目标是获得最好目标函数和去分每一个隔离的帧，或者你在对像没有使用语言模型里的TIMIT数据库的解码，整个答案是肯定的。问题就是如果使用更多的上下文，会降低我们整个系统的性能。我们认为HMMs基于state-conditional帧独立的假设，系统很难与其交互好。总之，无论什么原因，它看起来都没有起作用。

Configuration values relating to the LDA transform.

在训练神经网络前，我们对特征使用了一个去相关变换。这个变换事实上称为神经网络的一部分—— 我们在之前固定的FixedAffineComponent类型，它是不需要训练的。我们称它为LDA变换，但是它跟传统的LDA是不一样的，因为这里我们对变换的行进行尺度拉伸。 这个部分将处理影响这个变换的配置值。这些通过选项–lda-opts ""传递到脚本get_lda.sh。

注意这里除了对数据进行去相关，我们还得使数据是零均值；这也许是因为输出是一个仿射变换(linear term plus bias)，表示一个d*(d+1)的矩阵，而不是d*d(这里的d表示特征的维度，一般为40*9)。 默认地，这个变换是一个非降维形式的LDA，比如：我们保持整个维度。这也许听起来有点奇怪，因为LDA的作用就是降维。但是我们这里将它用来对数据去相关。

在传统的LDA，大多数会这么去做，对数据进行归一化，以使得类内方差为单位矩阵，类间方差是对类与类之间的方差按从最大到最小排列的对角矩阵。所以在这个变换之后，整个方差(类内和类间)在第i个对角上的值为1.0 + b(i)，这里的b(i)是与数据无关的，和随i降低的。我们修改的LDA，不是真正的LDA，采用这个变换和对每一行乘以 $\sqrt{\frac{\text{within-class-factor}+b(i)}{1+b(i)}}$，这里默认的 within-class-factor为0.0001。这个方差的影响是这个因子的平方根，所以方差的第i个元素的影响因子默认地不是1.0 + b(i)，而是 0.0001 + b(i)。基本上，我们缩减那些无信息的维度，因为我们的经验是添加无信息的数据到神经网络的输入会使性能变差，和简单地通过缩小它，可以使SGD训练在大多数情况下忽略它，这个是有用的。 我们怀疑如果对神经网络做一个简单的假设，比如它仅仅是一个逻辑斯特回归或者更简单的一个，这样的方案有时候是最佳的(也是使用0而不是0.0001)。不管怎么说，这个仅仅是一个技巧。

这里的配置参数–lda-dim有时候是用来强制这个变换来降维，而不是使所有的维度都通过。当处理一个输入维度特别大，我们会常常使用它，但是也不一定是有用的。

Other miscellaneous configuration values

对于脚本train_tanh.sh，有个选项–shrink-interval(默认为5)，它决定我们多久对模型做一个收缩(看Model "shrinking" and "fixing")，我们使用训练数据集的一小部分来优化不同层参数的拉伸。这个不是很重要的。

选项–add-layers-period(默认为2)控制了在添加层之间我们需要等待的迭代次数，训练一开始我们就添加新的层。 这个可能是有差别的，但是我们一般不去调整它。

Preconditioned Stochastic Gradient Descent

这里我们不使用传统的Stochastic Gradient Descent (SGD)，而使用一个特殊的预处理形式的SGD。这就意味着，不是使用一个通常的学习率，而是使用一个矩阵值的学习率。这个矩阵有一个特殊的结构，所以实际上是每个仿射成分的输入维度的一个矩阵和输出维度上的一个矩阵。如果你想把它看成一个单个大矩阵，它将是一个对角块结构的矩阵，其中每个块是两个矩阵的Kronecker积 (一个是输入维度和一个对应仿射成分的输出维度)。除此之外，每个minibatch中矩阵被估计。最基本的思想就是当衍生物有一个高的方差就降低学习率；这将趋去控制不稳定性和在任何一个方向上因为太快而停止参数更新。

这里我们还没有足够的时间来做一个详细的总结，但是可以看源文件nnet2/nnet-precondition.h和 nnet2/nnet-precondition-online.h的说明，这里将描述的更详细。我们需要注意的是nnet2/nnet-precondition.h包含这个方法最原始的版本，对每一个minibatch来估计预处理矩阵。而 nnet2/nnet-precondition-online.h包含这个方法的最新版本，这里的预处理矩阵是一个特殊的低秩加单元矩阵的结构和通过在线估计；如果使用GPU来实现会更加有效，因为旧的方法依赖对称矩阵求逆，而且是在一个相当高的维度上做的(比如：512)，在一个GPU上很难做到是有效的，然后就变成了一个瓶颈。

kaldi中的在线解码器

翻译：wbgxx333@163.com

时间：2015年5月

本文档将介绍kaldi里在线解码。

在线解码，我们的意思是实时地处理特征来解码，和我们不需要等待整个音频都获取后再解码。(我们不使用实时解码这个词，主要原因是实时解码经常用在那些跟实时不相上下的解码中，即使它在批处理模式中经常使用)。

在kaldi最初的几年里，我们仅仅考虑离线识别的方法，为了尽可能快地达到先进水平，我们现在尽最大的努力来支持在线解码。

这里有两种在线解码的例子：一个是旧的在线解码，在目录online/和onlinebin/，和新的解码例子，在目录online2/和online2bin/。这个旧的在线解码已经被弃用，它们最终将在trunk中移除(但会在.../branches/complete里保留)。

对于旧的解码例子，这里有一些文档，但是我们建议先读这页。

Scope of online decoding in Kaldi

在kaldi中，我们的目标是为在线解码提供类似于库函数的接口。也就是说，我们为其提供函数特性的接口而不是命令行的形式。原因是，根据数据的获取和传输，不同的人的需求是不同的。在旧的在线解码中，我们提供在UDP上传输数据的接口和类似的解码接口，但是在新的在线解码里，我们的目标仅仅是去演示内部的代码，和从现在开始我们不提供任何样例程序，你可以根据情况挂接到你真实的实时音频采集中，你可以用你自己的方式来做。

我们这里为基于GMM模型提供了解码程序(看下一部分)和对于神经网络模型也提供了解码程序(你可以看 Neural net based online decoding with iVectors).

online decoding

程序online2-wav-gmm-latgen-faster.cc是目前基于GMM模型的在线解码的最原始样例程序。它读取整个wav文件，但是在内部是逐块来处理的。在样例脚本egs/rm/s5/local/online/run_gmm.sh中，你可以看到一个样例脚本是如何利用这个程序来建立模型和评估的。这个程序的主要目的就是在一个典型的批处理框架下使用基于GMM的在线解码，这样你可以很简单的评估字错误率。我们计划对SGMMs和DNNs添加相同的程序。事实上为了做在线解码，你可以修改这个程序。你应该注意到(和这个对于语音识别的人是明显的，但对于其他的人就不是)音频的采样率应该跟你训练使用的音频的采样率匹配(和过采样将不行，但是下采样可以)。

Decoders versus decoding programs

在Kaldi中，当我们使用解码这个词，我们一般不是指整个解码程序。我们指的是内部的解码对象，是类型LatticeFasterDecoder的一种。整个对象使用解码图(一个FST)，和解码对象(看The Decodable interface)。所有的解码器一般都支持在线解码；只不过解码程序里的代码需要去修改。我们需要注意的是，在线解码器调用decoder时的一些不同就行。

- 在旧的在线解码中(在online/)，如果"decoder"是一些decoder (比如：类型LatticeFasterDecoder)和"decodable"是一个decodable对象的一个合适类型，你需要调用decoder.Decode(&decodable)，和当输入完成了，这个调用将阻塞(因为decoder调用了decodable.IsLastFrame()，而这个会阻塞)。
- 在新的在线解码中(在online2/)，你将调用decoder.InitDecoding()，和然后每次你得到更多的特征数据时，你将调用decoder.AdvanceDecoding()。对于离线使用，你将调用Decode()。

我们需要注意的是，在旧的在线解码中，有一个decoder调用OnlineFasterDecoder。从这个名字可以看出，这个唯一的一个支持在线解码的decoder。这个OnlineFasterDecoder最特殊的是它有能力去推断哪些字是接下来需要解码的，但是不需要知道未来的音频数据是什么。所以你可以输出这些单词。这个对于在线转写来说是非常重要的，和如果这是个需求的话，我们也许将目录online/的decoder移到目录decoder/和使得对新的在线解码是兼容的。

Feature extraction in online decoding

在线解码中的绝大部分复杂度是与特征提取和自适应相关的。

在online-feature.h里，我们提供了特征提取的各种成分的类，所有的都是从类OnlineFeatureInterface衍生出来的。OnlineFeatureInterface是在线特征提取的一个基类。这个接口指定这个对象如何把特征给调用者(OnlineFeatureInterface::GetFrame())和或者已经准备了多少帧的特征(OnlineFeatureInterface::NumFramesReady())，但是不能知道是怎么获得这些特征的。这个取决于子类。

在online-feature.h中，我们定义了类OnlineMfcc和OnlinePlp，它们都是底层的特征。它们都有一个成员函数OnlineMfccOrPlp::AcceptWaveform()，当获取到数据时，使用者可以调用。在online-feature.h 里的所有其他在线特征类型是"derived"特征，所以在它们的构造函数中采用了一个对象OnlineFeatureInterface和通过一个存储指针指向这个对象来获取它们的输入特征。

在online-feature.h中的在线特征提取代码，唯一不繁琐的部分就是the cepstral mean and variance normalization (CMVN) (和注意fMLLR，或者线性变换，和估计都不是繁琐的，复杂度在其他地方)。接下来我们将介绍CMVN。

Cepstral mean and variance normalization in online decoding

倒谱均值归一化是使用原始数据减去数据平均值（一般是MFCC特征数据）的归一化方法。"Cepstral"仅仅指的是正常的特征类型；在MFCC中的第一个C的意思就是"Cepstral"，倒谱是对数频谱的逆傅里叶变换，尽管这里我们一般使用余弦变换。无论任何方式，在倒谱方差归一化中，每个特征维度被尺度化，以使得它的方差为1。在所有的当前脚本里，我们关掉倒谱方差归一化和仅仅使用倒谱均值归一化，但是相同的代码都可以处理它们。在接下来的讨论中，为了简化，我们仅仅讨论倒谱均值归一化。

在Kaldi的脚本中，倒谱均值和方差归一化(CMVN)一般是对每一个说话人来做的。明显地，在一个在线解码的例子中，这个是不可能做的，因为它是非因果的(当前的特征依赖于未来的特征)。

我们的基本解决方案就是使用滑动窗来做倒谱均值归一化。我们在一个滑动窗上，默认的是6秒，来计算均值(看目录online2bin/里程序的选项"--cmn-window"，默认设置为600)。为这个计算的选项类OnlineCmvnOptions有一些额外的配置变量，speaker-frames (default: 600),和global-frames (default: 200)。这些指定了我们将从相同的说话人中怎么利用这些先验的信息，或者倒谱的一个广义平均，来改善每句话的一开始几秒的估计。程序apply-cmvn-online可以把这个归一化作为训练的一部分，以至于我们可以在匹配的特征上进行训练。

Freezing the state of CMN

类OnlineCmvn有函数GetState和SetState，这就使得可以跟踪说话人间的CMVN的的状态。它也有一个函数Freeze()。这个函数可以把倒谱均值归一化的状态设置为一个特定的值，以至于在调用这个函数Freeze()之后，，甚至更早些时候，当用户调用Freeze()，调用GetFrame()将应用我们使用的均值offset。通过调用函数GetState和SetState，这个冻结状态将传播到这个说话人的未来的语句中。我们这么做的原因是我们不认为在一个经常变化的CMN offset上用 fMLLR做说话人自适应是有意义的。所以当我们开始估计fMLLR(看接下来的),我们将冻结CMN状态和在未来让他固定。我们冻结的时刻，CMN的值不是特别重要的，因为fMLLR包含CMN。我们冻结CMN状态到一个特定值的原因，不是当我们开始估计fMLLR仅仅跳过CMN，而是当我们可以增量的估计这个参数时使用了一种叫basis-fMLLR的方法，，和它对offsets来说不是完全不变的。

Adaptation in online decoding

在语音识别中使用的最标准的自适应方法是feature-space Maximum Likelihood Linear Regression (fMLLR)，在我们的书籍上称为受限制的MLLR (CMLLR)，但是我们在kaldi的代码和文档里使用词语fMLLR。fMLLR是由特征的一个仿射(linear + offset)变换构成的；参数的数目是d * (d+1)，这个d是最终的特征维度(一般为40)。当解码更多数据时，在线解码需要使用一个基本的方法来增量估计不断增加的变换参数。在解码层的最高层逻辑大多通过类SingleUtteranceGmmDecoder来实现。

fMLLR估计不是连续去做的，而是分阶段去做的，因为它涉及到计算词图后验概率和这个使用连续的方式是不容易去做的。在类 OnlineGmmDecodingAdaptationPolicyConfig里的配置变量决定了我们数目时候去重新估计fMLLR。默认的情况是，在第一个语句中，在2秒后开始估计，和此后以1.5倍的速度几何级数的增长(所以在2秒，3秒，4.5秒...)。对于后面的语句，我们在5秒，10秒，20秒等等的时候估计它。对于所有的语句，我们在语句的末尾估计它。

注意CMN适应的状态是冻结的，就像我们上面提到的那样，我们为一个说话人第一次估计fMLLR时，默认的是在第一个语句的2秒。

Use of multiple models in GMM-based online decoding

在online-gmm-decoding.h中为GMM的在线解码中，有三个模型可以被使用。它们都是类OnlineGmmDecodingModels，如果很少的模型被提供时，你需要根据一定的逻辑来使用不同目的的模型。这三个模型分别为：

- 说话人独立模型，使用apply-cmvn-onlin.cc apply-cmvn-online来训练得到一个在线模式的CMVN
- 说话人自适应模型，使用fMLLR训练得到
- 说话人自适应模型的区分性训练版本。我们这里使用最大似然估计模型来估计自适应的参数，这样比使用一个区分性训练模型更能与最大似然框架保持一致，尽管这个区别很小和你使用区分性训练模型可能损失的信息很少(和节约一些内存)。

Neural net based online decoding with iVectors

我们最好的在线解码样例，我们推荐使用的就是基于神经网络的解码器。适应的哲学就是给你未适应的神经网络和没有均值归一化的特征(在我们的样例中是MFCCs)，和给他一个iVector。一个iVector 是一个具有几百维度的向量(在这个特殊的例子里，是一二百)，它表示说话人的特性。为了了解更多的信息，你可以去看说话人识别文献。我们的意思就是iVector可以给我们需要知道的说话人的信息。这个被证明是有用的。iVector的估计是以从左向右的方式，也就是说在某个时间t上，你可以看到从时间0到t上的输入。它可以看到这个说话人的之前的语句信息。iVector的估计是通过最大似然来计算的，涉及到混合高斯模型。

如果pitch被使用(比如：对于有声调的语言)，为了简化这个事情，在iVector估计中我们没有把它包含在特征里；我们仅仅把它包含在神经网络的特征中。对于有声调的语言，我们目前还有没有在线神经网络解码的样例脚本；它还在调试中。

在我们的在线解码的样例例子中的神经网络是p-norm神经网络，一般在多GPUs并行训练中使用。对于不同的样例，我们有一些样例脚本，比如：egs/rm/s5，egs/wsj/s5，egs/swbd/s5b，和egs/fisher_english/s5。最顶层的样例脚本一般是local/online/run_nnet2.sh。在RM数据集上，也使用脚本local/online/run_nnet2_wsj.sh。 这就演示了如何使用一个大的神经网络来训练相同采样率的其他数据库的语音（在我们的例子中是WSJ），和再原来的数据上重新训练。用这种方式，我们在RM上获得了最好的结果。

现在我们已为这个例子添加区分性训练的样例脚本。

Example for using already-built online-nnet2 models

这部分我们将解释如何从www.kaldi-asr.org下载已经训练好的 online-nnet2模型和使用自己的数据来评估。

你可以从 `http://kaldi-asr.org/downloads/build/2/sandbox/online/egs/fisher_english/s5` 里下载模型和相关的文件，它是由fisher_english样例来建立的。为了使用online-nnet2模型，你仅仅需要下载2个目录：exp/tri5a/graph和exp/nnet2_online/nnet_a_gpu_online。使用下面的命令来下载这些文件：

```
wget http://kaldi-asr.org/downloads/build/5/trunk/egs/fisher_english/s5/exp/nnet2_online/
wget http://kaldi-asr.org/downloads/build/2/sandbox/online/egs/fisher_english/s5/exp/tri5
mkdir -p nnet_a_gpu_online graph
tar zxvf nnet_a_gpu_online.tar.gz -C nnet_a_gpu_online
tar zxvf graph.tar.gz -C graph
```

这些文件将被下载到本地目录。我们需要去修改config文件里的路径名，像下面这样：

```
for x in nnet_a_gpu_online/conf/*conf; do
  cp $x $x.orig
  sed s:/export/a09/dpovey/kaldi-clean/egs/fisher_english/s5/exp/nnet2_online/:$(pwd)/: <
done
```

接下来，选择一个单wav文件来解码。你可以从这里下载一个文件：

```
 wget http://www.signalogic.com/melp/EngSamples/Orig/ENG_M.wav
```

这是一个采样率为8kHz的音频文件(不幸的是它是英国英语，所以正确率不是那么好)。可以使用接下来的命令来解码：

```
~/kaldi-online/src/online2bin/online2-wav-nnet2-latgen-faster --do-endpointing=false \
    --online=false \
    --config=nnet_a_gpu_online/conf/online_nnet2_decoding.conf \
    --max-active=7000 --beam=15.0 --lattice-beam=6.0 \
    --acoustic-scale=0.1 --word-symbol-table=graph/words.txt \
   nnet_a_gpu_online/smbr_epoch2.mdl graph/HCLG.fst "ark:echo utterance-id1 utterance-id1
   ark:/dev/null
```

我们添加了选项–online=false，因为它可以稍微改善了结果。你可以在log文件里看到结果(尽管有其他的方式来检索这个)。对于我们来说，这个log文件的输出如下：

```
/home/dpovey/kaldi-online/src/online2bin/online2-wav-nnet2-latgen-faster --do-endpointing
LOG (online2-wav-nnet2-latgen-faster:ComputeDerivedVars():ivector-extractor.cc:180) Compu
LOG (online2-wav-nnet2-latgen-faster:ComputeDerivedVars():ivector-extractor.cc:201) Done.
utterance-id1 tons of who was on the way for races two miles and then in nineteen ninety
LOG (online2-wav-nnet2-latgen-faster:main():online2-wav-nnet2-latgen-faster.cc:253) Decod
LOG (online2-wav-nnet2-latgen-faster:Print():online-timing.cc:51) Timing stats: real-time
LOG (online2-wav-nnet2-latgen-faster:main():online2-wav-nnet2-latgen-faster.cc:259) Decod
LOG (online2-wav-nnet2-latgen-faster:main():online2-wav-nnet2-latgen-faster.cc:261) Overa
```

关键词搜索模块

本文由@冒顿翻译，后由@wbgxx333搬移到这里。

联系方式：wbgxx333@163.com

介绍

本文描述了在kaldi中的关键词搜索模块。我们实现了以下功能：

- 对网络进行索引以便快速检索关键词
- 使用代理关键词以处理表外词（OOV）问题

在以下的文档中，我们将重点讨论为特殊目的而进行的基于词的关键词检索，但我们同样支持子词级别的关键词检索。我们的语音识别模块和关键词检索模块都使用加权有限状态转换器wfst，这个算法也支持使用符号表将词/子词映射到整数。

本文剩下的部分结构如下：在Typical Kaldi KWS system部分, 我们描述了kaldi kws 系统的基本模块；在Proxy keywords部分, 我们解释了我们如何使用代理关键词处理不在词汇表中的关键词；最后在Babel scripts部分, 我们介绍了为iarpa bable项目所做的kws相关的脚本。

Typical Kaldi KWS system

我们在文章"Quantifying the Value of Pronunciation Lexicons for Keyword Search in Low Resource Languages", G. Chen, S. Khudanpur, D. Povey, J. Trmal, D. Yarowsky and O. Yilmaz中可以看到kaldi kws 系统的例子。一般来说，一个kws系统包括两个部分：一个lvcsr模块解码检索集合并且产生相应的网格，一个kws 模块生成网格索引并从索引中查找关键词。

我们的基础lvcsr 系统是一个sgmm+mmi 系统，我们使用标准的PLP分析器抽取13维的语音特征，然后用一个典型的最大似然估计进行语音训练，以一个平滑的上下文无关的音素HMM做初始值开始，以说话人自适应（SAT）的状态集群三音素hmm-gmm做为输出结束。在说话人变换训练集的统一背景模型（UBM)进行训练，然后得到用于训练HMM发射概率的子空间高斯混合模型（SGMM），最后，所有的训练语音使用SGMM系统进行解码，然后对sgmm的参数进行bmmi训练。更详细的细节参见egs/babel/s5b/run-1-main.sh。

我们在sgmm_mmi系统之外还创建了一些附加的系统，如：一个混合深度神经网络(DNN).详情见egs/babel/s5b/run-2a-nnet-gpu.sh,一个瓶颈特殊(BNF)系统，详情见egs/babel/s5b/run-8a-kaldi-bnf.sh 等等。所有这些系统都是对相同的检索集合进行解码并且生成网格，随后送到kws 模块进行索引和检索。我们在检索结果上而不是在网格上将这些系统组织起来。由上述lvscr 系统产生的网格，使用在文章"Lattice indexing for spoken term detection", D. Can, M.

Saraclar, Audio, Speech, and Language Processing中描述的网格索引技术进行处理。所有待检索集语句中的网格都被从单一加权有限状态转换成一个单广义因数变送器结构，将每个词的开始时间，结束时间和网格后验概率这三维数据存储起来。给一个词或短语，我们创建他的简单有限状态机，可以得到这个关键词/短语并且将他与因数变送器得到关键词/短语在检索集合中所有出现过的地方，和一个语句的ID号，开始时间，结束时间，以及每个地方网格的后验概率。所有检索出来的结果以他们的后验概率进行排序，并且使用论文"Rapid and Accurate Spoken Term Detection"中的方法来对每个实例判断是或否。

Proxy keywords代理关键词

我们的代理关键词产生过程在论文 "Using Proxies for OOV Keywords in the Keyword Search Task", G. Chen, O. Yilmaz, J. Trmal, D. Povey, S. Khudanpur做了描述。我们最早使用这个方法解决词网格中的oov问题，如果关键词不在lvcsr系统的词汇表中，尽管实际上这个词在谈话中已经被提到了，但他也不会出现在检索集的网格中。这是lvcsr 为基础的关键词检索系统的一个老问题，并且已经有办法解决它，比如，创建一个子词系统。我们的办法是寻找语音中与词汇表中相似的词，并且使用这些词做为代理关键词替换那些表外词汇。这样做的好处是我们不用创建额外的子词系统。在即将到来的interspeech 的论文"Low-Resource Open Vocabulary Keyword Search Using Point Process Models", C. Liu, A. Jansen, G. Chen, K. Kintzley, J. Trmal, S. Khudanpur中我们证明这技术可以同一个建立在点处理模型上的音素检索模型相媲美。代理关键词是一种模糊检索方法，他在处理表外词的同时也可以加强IV关键词性能。 通常的代理关键词处理过程可以用以下公式产生：

$$K' = \mathrm{Project}\left(\mathrm{ShortestPath}\left(\mathrm{Prune}\left(\mathrm{Prune}\left(K \circ L_2 \circ E'\right) \circ L_1^{-1}\right)\right)\right)$$

这里 $\beta = \sum_{j,m} \gamma_{jm}$ 为原始的关键词， $\mu = \frac{1}{\beta} \sum_{j,m} \mu_{j,m}$ 是包含发音 $\beta = \sum_{j,m} \gamma_{jm}$ 的 词汇。如果 $\beta = \sum_{j,m} \gamma_{jm}$ 是表外词，这个词汇可以使用G2P工具得到。 $O(K^3)$ 是编辑距离转换，表示音素与训练集的差异程度。 $O(K^2)$ 是原始词汇。 K' 是包含多个IV词的WFST，是与原始发音 相似的发音词。我们把他当成原始词 $\beta = \sum_{j,m} \gamma_{jm}$ 进行检索。 注意两个修正阶段是必不可少的，尤其当你的词汇表非常大的时候。我们同时实现了一个简单的组合算法，只产生必须的组合状态（比如，不产生以后会被修正的状态）。这避免了在计算 $K \circ L_2 \circ E'$ 和 L_1^{-1} 组合的时候使用内存太多。

Babel scripts

A highlevel look 概述

我们为IARPA Babel 项目建立了一个“一键”脚本。如果你在Babel 上进行实验时想用我们的脚本，你可以用以下几步建立一个SGMM+MMI的关键词检索系统（假设你的工作目录是egs/babel/s5b/)

- 安装F4DE 并且将他设置在你的path.sh中。
- 修改cmd.sh以保证可以运行在你的集群 。

- 将conf/languages中的一个文件连接到./lang.conf, 比如 "ln -s conf/languages/105-turkish-limitedLP.official.conf lang.conf"
- 修改lang.conf中的JHU集群中的数据使它指向你的数据文件。
- 运行run-1-main.sh ,建议lvcsr系统。
- 运行run-2-segmentation.sh, 产生eval 数据的分割
- 运行run-4-anydecode.sh,解码eval 数据，生成索引检索关键词。

同时，你可以建立DNN系统，BNF系统，Semi-supervised 系统等等语音识别系统。关键词检索任务在run-4-anydecode.sh中执行。我们将在下面介绍如何进行关键词检索的细节，你可以把这些方法用于其它的数据上。我们假设你已经解码了检索集合并且产生的相应的网格。

关键词检索数据准备

一般来说，我们在检索数据的目录中建立kws的数据目录。例如，如果你有一个叫做dev10h.uem的检索集。数据所在的目录为data/dev10.uem/. 我们在在这个目录下面创建kws数据目录，如data/dev10h.uem/kws. 在创建kws目录之前，你必须有三个文件，一个包含检索信息的ecf文件，一个关键词列表kwlist文件，一个打分文件rttm. 有时你必须自己创建这些文件，例如，你可以创建一个rttm文件指定模型中的检索的参数。下面我们列出这些文件的格式：ECF文件的例子：

```
<ecf source_signal_duration="483.825" language="" version="Excluded noscore regions">   <
```

KWLIST文件的例子：

```
<kwlist ecf_filename="ecf.xml" language="tamil" encoding="UTF-8" compareNormalize="" vers
```

RTTM 文件的例子：

```
SPEAKER YOUR_AUDIO_FILENAME 1 5.87 0.370 <NA> <NA> spkr1 <NA> LEXEME YOUR_AUDIO_FILENAME
```

准备好这些文件后，你可以开始准备kws 数据目录。如果你只是想进行基本关键词检索，运行以下命令：

```
local/kws_setup.sh \   --case_insensitive $case_insensitive \   --rttm-file $my_rttm_file
```

如果你要对表外词汇进行模糊检索，你可以运行以下几个命令，这些命令首先收集相似音素，然后训练G2P模型，创建KWS数据目录：

```
#Generate the confusion matrix #NB, this has to be done only once, as it is training corp
```

Indexing and searching索引和检索

在这一阶段，我们假设你已经解码了检索集合并且生成了相应的网格。运行以下脚本将进行索引和检索：

```
local/kws_search.sh --cmd "$cmd" \   --max-states ${max_states} --min-lmwt ${min_lmwt} \
```

如果你的KWS数据目录有一个额外的ID号，比如 oov(当你进行不同的kws 时，这很有用，这种情况下，你的目录可以是这样的data/dev10h.uem/kws_oov),你必须使用extraid 选项：

```
local/kws_search.sh --cmd "$cmd" --extraid $extraid  \   --max-states ${max_states} --min
```

Kaldi中的并行化

翻译：@老那(asr.naxingyu@gmail.com)

时间：2015年4月

简介

使用Kaldi最理想的环境是配备集群任务分发工具，如Sun GridEngine。如果同时使用多个服务器组成的集群，还需要能同时访问的共享文件系统，如NFS。即便没有这些，你也可以在单个一台服务器上方便地安装Kaldi。

在主示例脚本中，如 `egs/wsj/s5/run.sh` ，可以看到如下的命令

```
steps/train_sat.sh --cmd "$train_cmd" \
  4200 40000 data/train_si284 data/lang exp/tri3b_ali_ai284 exp/tri4a
```

在 `run.sh` 的最上面，可以看到脚本引用了一个叫做 `cmd.sh` 的文件

```
. ./cmd.sh
```

在 `cmd.sh` 里面，可以看到如下的变量赋值语句

```
export train_cmd="queue/pl -l arch=*64"
```

如果你没有配置GridEngine，或者你的任务队列设置与约翰霍普金斯大学的CLSP实验室（译者注：Daniel Povey所在的实验室，下文中简称CSLP@JHU）不同，那么你需要更改这个变量的值。如果在一台本地服务器上运行，则要设置 `export train_cmd=run.pl`

在 `steps/train_sat.sh` 中，变量 `cmd` 被传给了 `--cmd` 选项，也就是说在这个示例中， `--cmd` 选项的值被设为 `queue/pl -l arch=*64` 。在任务脚本中，可以看到如下的命令

```
$cmd JOB=1:$nj $dir/log/fmllr.$x.JOB.log \
  ali-to-post "ark:gunzip -c $dir/ali.JOB.gz|" ark:-  \| \
  weight-silence-post $silence_weight $silphonelist $dir/$x.mdl ark:- ark:- \| \
  gmm-est-fmllr --fmllr-update-type=$fmllr_update_type \
  --spk2utt=ark:$sdata/JOB/spk2utt $dir/$x.mdl \
  "$feats" ark:- ark:$dir/tmp_trans.JOB || exit 1;
```

这条命令的意思是执行 `$cmd` （如 `queue.pl` 或者 `run.pl` ）指令。它负责产生任务，然后等待它们结束，最后如果中间出现什么差错的话返回一个非零的状态标记。这些指令（除此之外还有其他的如 `slurm.sh` 和 `ssh.sh` 等）的基本使用方法是

```
queue.pl <options> <log-file> <command>
```

最简单的一个例子是

```
run.pl foo.log echo hellow world
```

（举这个 `run.pl` 的例子是因为它可以在任何系统上运行，不依赖于GridEngine）。也可以尝试创建一个任务队列，如

```
run.pl JOB=1:10 foo.JOB.log echo hello world number JOB
```

这样，被执行的指令就会将所有 `JOB` 字符替代为给定范围的数字。所以，需要确认你的工作目录中不包含 `JOB` ，否则无法正常工作。借助引用符和输出符，你也可以用 `pipe` 或者 `redirection` 提交任务

```
run.pl JOB=1:10 foo.JOB.log echo "hello world number JOB" \| head -n 1 \> output.JOB
```

这种情况下，实际被执行的指令是

```
echo "hello world number JOB" | head -n 1 > output.JOB
```

如果想查看实际执行的指令，可以打开 `foo.1.log` 文件，其内容如下

```
# echo "hello world number 1" | head -n 1 output.1
# Started at Sat Jan 3 17:44:20 PST 2015
#
# Accounting: time=0 threads=1
# Ended (code 0) at Sat Jan 3 17:44:20 PST 2015, elapsed time 0 seconds
```

并行化工具的通用接口

这部分介绍并行化工具的使用。这些工具在使用时是可替换的，也就是说，使用一种并行化工具测试成功的脚本，也可以使用其他并行化工具。只需改变 `$cmd` 的值，就可以在不同的并行化工具之间切换。

这些工具的基本用法如下

```
queue.pl <options> <log-file> <command>
```

下面介绍的内容同样适用于 `run.pl` ， `ssh.pl` 和 `slurm.pl` 。

`< options>` 可以包含以下内容：

- 任务序号标记（例如 `JOB=1:10` ）。名称使用大写仅为方便使用，名称中也可以包含下划线。起始标记必须大于等于1，这是GridEngine的要求。
- 符合GridEngine要求的用于传递给 `qsub` 的选项。例如 `-l arch=*64*, or -l mem_free=6G,ram_free=6G, or -pe smp 6` 。除 `queue.pl` 以外的其他脚本会忽略这个选项。
- 新定义的选项，例如 `-mem 10G` （见下文）。

`< log-file>` 是一个文件名，对于任务队列，文件名可以包含队列的标记（如 `exp/foo/log/process_data.JOB.log` ）。

`< command>` 可以是任何字串，可以包含shell可解释的符号。但是，如果一个字串先被 `bash` 解释了，那就无法再被 `queue.pl` 处理。比如，如下的写法是错误的

```
queue.pl test.log  echo foo | awk 's/f/F/';
```

因为 `queue.pl` 无法看到 `|` 符号后面的内容，而是会将它的输出交给awk指令。正确的写法是

```
queue.pl test.log  echo foo \| awk 's/f/F/';
```

对于 `|` 、 `;` 和 `>` 的使用都应当注意。另外，如果 `< command>` 的选项中包含空格， `queue.pl` 会假设它应当加 `\` 符，并且在传给bash之前替你加上 `\` 符号。默认情况下，使用单引用符，如果指令本身包含单引用符，就使用双引用符 `\\` 。这种方法通常可以行得通。运行 `queue.pl` 的shell环境变量 `PATH` 会被传入被执行的脚本中。为保证所需的变量都已定义， `./path.sh` 会再次被引用。然后用bash执行指令。

新定义的选项（统一接口）

当我们编写Kaldi示例脚本时，如果需要指定内存需求，就把类似于 `-l ram_free=6G,mem_free=6G` 的选项传递给 `queue.pl` 。在使用 `steps/train_sat.sh` 等脚本时，由于无法获知GridEngine配置的方式（甚至到底有没有GridEngine），这类选项需要从最外层的脚本传进去，这样很奇怪。最近，我们为并行化工具定义了新的接口，使得它们可以理解如下的选项

```
--config conf/queue_mod.conf
--mem 10G
--num-threads 6
--max-jobs-run 10
--gpu 1
```

其中，配置文件定义了如何将新接口转换为GridEngine（或者你使用的其他任务分发工具）支持的格式。目前，只有 `queue.pl` 支持这些选项，其他的脚本会忽略这些选项。我们计划逐渐修改 `steps/` 中的脚本，使得它们都能支持这种新的接口，并且使 `queue.pl` 能够支持任务分发工具的其他选项（如有必要）。

如果有 `conf/queue.conf` 配置文件， `queue.pl` 会读取它。否则，会使用代码中定义的默认值。配置文件定义了如何将新的接口转换为GridEngine或者其他任务分发工具支持的选项。下面的例子是默认的配置转换方法

新接口	转换后 **(GridEngine**格式**)**	注释
–mem 10G	-l mem_free=10G,ram_free=10G	
–max-jobs-run 10	-tc 10	(We use this for jobs that cause too much I/O).
–num-threads 6	-pe smp 6	(general case)
–num-threads 1	(no extra options)	(special case)
–gpu 1	-q g.q -l gpu=1	(general case)
–gpu 0	(no extra options)	(special case for gpu=0)

可以使用这种格式定义其他选项，也就是形如 `–foo-bar` 后面跟着一个参数。默认的配置是与CLSP的集群配合的，在其他地方可能无效，因为GridEngine的配置方式各异。因此，可以创建一个配置文件 `conf/queue.conf` ，以配合你的集群。下面的配置文件就是默认配置，可以以此为例创建你自己的配置文件

```
# Default configuration
command qsub -v PATH -cwd -S /bin/bash -j y -l arch=*64*
option mem=* -l mem_free=$0,ram_free=$0
option mem=0          # Do not add anything to qsub_opts
option num_threads=* -pe smp $0
option num_threads=1  # Do not add anything to qsub_opts
option max_jobs_run=* -tc $0
default gpu=0
option gpu=0
option gpu=* -l gpu=$0 -q g.q
```

`command` 开头的行定义了指令中不可更改的部分，你可以修改这行使它能配合其他任务分发工具使用。 `option` 开头的行定义了如何转换输入选项，例如 `-mem` 。以" `option mem=*` "开头的行处理一些通用的选项（ `$0` 被选项的实际输入参数代替）。形如 `option gpu=0` 的行允许你指定一些参数的特殊情况，比如在这个例子中的 `gpu=0` 。 `default gpu=0` 定义了这个选项的默认值，如果你没有给出 `-gpu` 的值， `queue.pl` 就会默认你指定了 `-gpu 0` 。在这个示例中，我们可以忽略 `default gpu=0` 这个配置，因为它没有给出额外的参数信息。在之前的版本中，曾使用 `option gpu=0 -q all.q` ，在那种情况下， `default gpu=0` 是有作用的。

由配置文件定义参数到qsub选项参数的转换规则有时需要在perl代码中调整。比如，在现在的脚本中，非分布式任务会忽略 `-max-jobs-run` 这个选项。

计算网格相关配置示例

我们举一个在真实环境中如何使用配置文件的例子。有一段时间，在使用K20训练神经网络时，程序会崩溃，但是用K10就没事。缘由是我们当时在网格中安装的一个过时的CUDA开发库中存在一个bug。因此，我们创建了一个名为 `conf/no_k20.conf` 的配置文件，其内容与上面的示例类似，但加入了如下几行

```
default allow_k20=true
option allow_k20=true
option allow_k20=false -l 'hostname=!g01*&!g02*&!b06*'
```

然后将脚本中的 `$cmd` 设置为 `queue.pl -config conf/no_k20.conf -allow-k20 false` 。另外一个等效的方法是直接编辑配置文件中的command行

```
command qsub -v PATH -cwd -S /bin/bash -j y -l arch=*64* -l 'hostname=!g01*&!g02*&!b06*'
```

这样就不需要添加 `-allow-k20 false` 了。

如何使用不同的脚本进行并行训练

在这一部分中，我们解释不同的并行化工具是如何使用的。

使用**queue.pl**并行化

`queue.pl` 是Kaldi首选的并行训练工具。设计的初衷是配合GridEngine使用。不过由于我们定义了新的接口，现在它应当可以与其他并行任务分发软件配合了，如Tork和slurm。如果你开发了与这些软件配合的配置文件，请与Kaldi维护团队联系。为了更好的支持其他软件， `queue.pl` 也有必要修改。因为有些命令行是无法通过调用参数配置的，例如，加入 `-o foo/bar/q/jobname.log` 以使输出由qsub本身的控制转到另外一个独立的日志文件。另外，对

于队列任务，可以向命令行中添加如 `-t 1:40` 的选项。在调用qsub运行某个脚本时，它利用了一个环境变量，叫做 `$SGE_TASK_ID`，这是一个SGE用来标识不同任务的变量。我们的计划是扩展现有的配置文件机制，使它能够兼容为配合不同任务分发软件所做的任何合理修改。

因为在上文中我们已经用了很多篇幅解释 `queue.pl` 的工作原理，这里就不提供更多细节了。但是，请看下面的如何为*Kaldi*安装*GridEngine*。

使用**run.pl**并行化

如果没有GridEngine， `run.pl` 是一个不错的选择。这个脚本非常简单，它会在本机上运行你所提交的任务，如果你指定了任务队列，如 `JOB=1:10`，它将在本机上提交一系列并行的任务。它不会监测可用的CPU数目和以及内存的容量。所以，如果一个脚本本来应当用 `queue.pl` 在一个大的网格中提交，而你用 `run.pl` 在本地提交，最后可能耗尽内存或负载资源导致任务失败。如果要使用 `run.pl`，建议你仔细研究将要提交的任务脚本，对于一些解码任务要尤其小心，因为它们通常被提交到后台运行（用&符）。此外，对于一些并行任务数较多的也要格外注意，如 `-nj 50`。通常，你可以减小 `-nj` 的值，这样做一般不会影响结果。但在有些脚本中，同样的 `-nj` 被多个脚本使用，它们必须保持一致，否则后续的步骤就会崩溃。

除了任务队列标记外， `run.pl` 会忽略其他所有的参数。

使用**ssh.pl**并行化

`ssh.pl` 是穷人版的 `queue.pl`，如果你有一个小的服务器集群，又不想费力安装和配置GridEngine，就可以使用它。与 `run.pl` 类似，它并不监测CPU或者内存的使用情况。它的工作机制与 `run.pl` 几乎相同，唯一的区别是，它可以把任务分发到不同的机器上去。为了使用它，你需要创建一个叫做 `.queue/machines` 的文件（ `.queue` 是训练环境的一个子目录）。这个文件的每一行包含一个机器名。使用这个脚本的前提是，这些机器必须支持免密码的ssh登录，也就是说你需要建立一个ssh公钥。

使用**slurm.pl**并行化

`slurm.pl` 是用来兼容slurm任务分发软件的。这个软件的功能和GridEngine差不多。不过这个脚本最近没有测试。现在更好的选择是用 `queue.pl` 搭配一个兼容slurm的配置文件，这样就不需要 `slurm.pl` 了。

如何为**Kaldi**安装**GridEngine**

Sun GridEngine（SGE）是一个开源的计算集群\网格管理工具。是Kaldi的维护团队使用最多的版本。目前由甲骨文负责维护，所以他们现在管它叫Oracle GridEngine（译者注：从2013年10月22日起，GridEngine的维护和客户服务已经由Univa软件公司接管）。在CSLP@JHU

使用的版本是6.2u5。SGE是一个稳定的工具，所以使用哪个版本其实没那么重要。除SGE外，还有很多其他类似的软件，有一些是后期创建的开源分支。在这个文档中，我们仅指由Oracle维护的原版SGE。

在这一部分，我们将说明如何在一个集群中安装并配置GridEngine。如果你的集群使用亚马逊云EC2，并且要一个能够方便扩展新节点的工具，我建议你看看MIT的StarCluster项目。我们在Sourceforge上也创建过一个类似的项目，叫做kluster，因为当时StarCluster不是很稳定。但相信它现在已经有很大进步了。

安装GridEngine

首先，你需要安装GridEngine的基本工具。在GridEngine中，队列管理软件运行在主控节点（master）上，另外一些工具运行在所有队列节点上。master本身也可以加入队列。这里有个概念叫做影子主控（shadow master），相当于master的备份，以防master节点掉线。但在这里我们不做探讨（作者注：我们感觉就是在另外一个节点上安装gridengine-master，然后把master设置为原来的master。但不确定这么做是否可行）。

根据我们的经验，从源代码编译安装GridEngine是非常痛苦的。如果你的Linux发行版的源中包含GridEngine包，那就简单多了。你得谨慎一点，因为并不是所有的发行版中都包含GridEngine包。我们是在Debian上安装的，所以下面介绍如何在Debian上安装和配置。

在主控节点上安装GridEngine的指令是

```
sudo apt-get install gridengine-master gridengine-client
```

在选择界面中选择yes进行自动配置。安装程序将会让你输入cell name，可以用默认的default。然后它会问你master的名字，需要填上你所选择作为主控节点的计算机的主机名。通常，需要填写主控计算机的域名全称，但我觉得任何一个能通过主机名查找的方式找到这台计算机名字都可以。需要注意的是，有时候GridEngine会比较挑剔，要求主机名查找和域名（DNS）查找的结果完全匹配。很多GridEngine的运行问题都是由这个问题导致的。另外需要注意，使用 `apt-get remove` 删除这些工具并重新安装并不会要求你重新配置，因为Debian会记住你上次安装时的选择（译者注：管理员可以使用 `aptitude purge` 完全删除软件绑定的配置文件）。

然后把你自己加入SGE的管理员组

```
sudo qconf -am <your-user-id>
```

这里 `-am` 表示添加（add）管理员（manager），以此类推， `-dm` 用来删除（delete）管理员， `-sm` 用来显示（show）管理员列表。使用 `qconf -help` 查看所有可用的选项。

在普通节点上安装GridEngine的指令是

```
sudo apt-get install gridengine-client gridengine-exec
```

同样，cell name不变，master就是你之前安装了主控节点的主机名。你可以在主控节点上安装，这样主控节点也可以运行队列中的任务。

安装完成后，运行qstat和qhost -q，你会看到SGE是否已经开始工作了。下面就是一切正常的情况下你会看到的内容

```
dpovey_gmail_com@instance-1:~$ qstat
dpovey_gmail_com@instance-1:~$ qhost -q
HOSTNAME                ARCH         NCPU  LOAD  MEMTOT  MEMUSE  SWAPTO  SWAPUS
----------------------------------------------------------------------------------
global                  -               -     -       -       -       -       -
instance-1.c.analytical-rig-638.internal lx26-amd64      1  0.07    3.6G  133.9M     0.0
```

到这一步时，还没有完全配置好，这一步只是用来检查工作节点和主控节点是否已经连通。如果在这一步出现错误，那很可能跟DNS查找、反DNS查找、或者是你的 `/etc/hostname` 和 `/etc/hosts` 文件的内容有关。当这些信息不一致的时候，GridEngine会罢工。如果你想改变主控节点的名字，需要编辑文件

```
/var/lib/gridengine/default/common/act_qmaster
```

（作者注：这是在Debian Wheezy发行版中的位置）（译者注：修改之后最好运行 `/etc/init.d/gridengine-exec restart` 或者重新启动，以确保配置成功被加载）

配置GridEngine

首先需要定义一个队列（queue）。因为GridEngine默认不会定义任何队列。我们先来创建一个叫做 `all.q` 的队列。请确保shell环境变量 `EDITOR` 的值是你常用的编辑器，如 `vim` 或 `emacs` 。以下的配置方法适用于主控节点和工作节点。

```
qconf -aq
```

这条指令会打开一个编辑器。在编辑器中编辑以下这行

```
qname template
```

将它改为

```
qname all.q
```

然后，将 `shell` 的值改为 `/bin/bash` 。这个设置比默认值更好，不过对于Kaldi没有影响。退出编辑器后，这些设置会自动保存，但是如果存在语法错误， `qconf` 会拒绝你的修改。后面我们会用 `qconf -md all.q` 进一步修改这个队列的属性。

在GridEngine中保存着一些全局配置，这些配置不专属于某个队列。使用 `qconf -sconf` 查看这些配置项，或者用 `qconf -mconf` 进行编辑修改。比如下面这行

```
administrator_mail           root
```

如果你能在服务器上发送邮件（作者注：测试方法是，在服务器上执行 `mail foobar@gmail.com` 指令。），那么你就可以把 `root` 改成一个有效的邮件地址，这样如果GridEngine出现问题你就会通过邮件收到通知。需要注意的是，由于反垃圾邮件策略，在EC2上从云端发送邮件是非常麻烦的，要想用Google云服务发送邮件更是几乎不可能。所以最好不要修改这个配置项。另外一个可以修改的配置项是

```
flush_time=00:00:10
```

其默认值是 `00:00:15` 。这样修改可以使任务提交的速度加快。

在GridEngine中，存在一个叫做“资源”的概念，你可以在任务中请求或指定资源。可以通过 `qconf -sc` 查看资源或用 `qconf -mc` 修改。修改默认的内存需求的方法是，编辑 `mem_free` 行的值，由0改为1G。

```
#name               shortcut    type        relop requestable consumable default  urgency
#-------------------------------------------------------------------------------------------
<snip>
mem_free            mf          MEMORY      <=    YES         NO         1G        0
```

建议加上下面这两行，在任何地方插入都可以

```
#name               shortcut    type        relop requestable consumable default  urgency
#-------------------------------------------------------------------------------------------
<snip>
gpu                 g           INT         <=    YES         YES        0        10000
ram_free            ram_free    MEMORY      <=    YES         JOB        1G       0
```

只有当你想在队列中添加GPU服务时才需要"gpu"项。我们发现ram_free项在管理节点的内存方面用处很大，而系统默认的mem_free似乎并不能起到作用。后面往集群中添加节点时，将使用 `qconf -me <some-hostname>` 然后编辑 `complex_values` 如下

```
complex_values        ram_free=112G,gpu=2
```

（作者注：此例是对于一个有112G物理内存和2个GPU卡的机器）。如果提交一个内存需求10G的任务，可以在qsub的选项中指定 `-l mem_free=10G,ram_free=10G` ， `mem_free` 会确保在任务开始时工作节点有足够的内存。 `ram_free` 确保我们不会往同一个节点上同时提交多个需要大量内存的任务。除了添加 `ram_free` 资源需求外，我们还尝试了用 `qconf -mc` 编辑 `mem_free` 项的 `consumable` 值为 `YES` ，以确保GridEngine会持续监测内存需求。但是这种方法好像不太管用。注意， `ram_free` 和 `gpu` 都是我们选择的名字，对于GridEngine而言它们没有特别的意义，而其他默认的项目，如 `mem_free` 是有确切含义的。 `ram_free` 项的 `consumable` 值 `JOB` 的意义是 `ram_free` 资源是对每个任务指定的，而不是针对每个线程。这样对Kaldi而言更有用处。

接下来，你需要往GridEngine中添加并行环境 `smp` 。大部分GridEngine的管理员都会这样做，但是并没有被默认支持。这是一个非常简单的并行环境，GridEngine并不会做任何特别的事，它只是为你保留一定数目的CPU槽，这样当你执行 `qsub -pe smp 10 <your-script>` 时，就会有10个CPU槽可用，这对于多线程或多进程的任务是有帮助的。执行 `qconf -ap smp` ，并编辑 `slots` 项为999

```
pe_name            smp
slots              9999
...
```

然后使用 `qconf -mq all.q` ，编辑 `pe_list` 项以添加 `smp` 。

```
pe_list               make smp
```

这样就在all.q中添加了smp并行环境。

配置GridEngine（高级）

在这一部份，我们会介绍一些可能有帮助、但对于基本使用并不必需的小技巧。在CLSP集群中，我们编辑了 `qconf -mq all.q` 中的 `prolog` 域

```
prolog                /var/lib/gridengine/default/common/prolog.sh
```

这一项的默认值是NONE。我们将这个脚本拷贝到每个节点上相同的目录中。这一步的作用是，如果任务脚本当前无法获取，管理节点会等待一段时间，因为如果脚本刚刚被编辑，需要给NFS一些时间完成同步。这个脚本的内容如下：

```
#!/bin/bash

function test_ok {
  if [ ! -z "$JOB_SCRIPT" ] && [ "$JOB_SCRIPT" != QLOGIN ] && [ "$JOB_SCRIPT" != QRLOGIN
    if [ ! -f "$JOB_SCRIPT" ]; then
       echo "$0: warning: no such file $JOB_SCRIPT, will wait" 1>&2
       return 1;
    fi
  fi
  if [ ! -z "$SGE_STDERR_PATH" ]; then
    if [ ! -d "`dirname $SGE_STDERR_PATH`" ]; then
      echo "$0: warning: no such directory $JOB_SCRIPT, will wait." 1>&2
      return 1;
    fi
  fi
  return 0;
}

if ! test_ok; then
  sleep 2;
  if ! test_ok; then
     sleep 4;
     if ! test_ok; then
        sleep 8;
     fi
  fi
fi

exit 0;
```

等待至多14秒，这对我们的NFS来说足够了，因为我们在NFS中设置了刷新缓冲目录的最长等待时间为 `acdirmax=8` 。

同时在队列配置中，我们设置rerun为TRUE

```
rerun                TRUE
```

如果任务失败，它们会在qstat的输出中显示为Eqw状态，E表示出错了，通过运行 `qmod -cj <numeric-job-id>` 清除出错标记，你可以要求队列重新分配这些任务。或者如果你不想重新运行这些队列，可以用 `qmod -dj <numeric-job-id>0` 删除它们。将队列设置为允许重运行，可以避免从头开始。

除此以外，我们还对CLSP队列做了如下修改

```
rlogin_daemon                /usr/sbin/sshd -i
rlogin_command               /usr/bin/ssh
qlogin_daemon                /usr/sbin/sshd -i
qlogin_command               /usr/share/gridengine/qlogin-wrapper
rsh_daemon                   /usr/sbin/sshd -i
rsh_command                  /usr/bin/ssh
```

这些项目原来的默认值是

```
qlogin_command               builtin
qlogin_daemon                builtin
rlogin_command               builtin
rlogin_daemon                builtin
rsh_command                  builtin
rsh_daemon                   builtin
```

这么做的原因已经无从考证了，但是如果你的qlogin和qrsh出现故障时，可以试试这些配置。

配置GridEngine（添加节点）

在这一部分中，我们介绍如何在你的队列中添加节点。如上所述，你可以在新的节点上安装GridEngine

```
sudo apt-get install gridengine-client gridengine-exec
```

集群名默认，主控节点设为你配置为master的那个节点的主机名。

安装好并不意味着你的新节点已经添加到队列资源中。GridEngine将主机分为管理主机（administrative hosts）、执行主机（execution hosts）和提交主机（submit hosts）。你的新机器需要同时具有这三个角色。你可以通过 `qconf -sh` ， `qconf -sel` 和 `qconf -ss` 查看这三类主机的列表。通过如下命令将新节点设置为管理主机和提交主机

```
qconf -ah <新节点的主机名>
qconf -as <新节点的主机名>
```

用如下命令将新节点配置为执行主机

```
qconf -ae <新节点的主机名>
```

这一操作会打开一个编辑器，你可以将前面定义的 `ram_free` 和GPU项添加进去

```
complex_values    ram_free=112G,gpu=1
```

到这里，你可能注意到了在命令行中存在不对称现象，一边是 `qconf -sh` 和 `qconf -ss` ，而另一边是 `qconf -sel` 。后者的"l"表示以列表为命令执行对象。这里的区别在于，管理节点和提交节点的列表中只包含主机，而执行节点的列表包含一系列相关信息，因此，对于执行节点用不同的数据结构加以区别。如果你想查看某个执行节点的详细信息，可以使用 `qconf -se <节点的主机名>` 。如果添加或者修改执行节点信息，则分别使用 `qconf -ae <节点的主机名>` 和 `qconf -me <节点的主机名>` 。这是GridEngine的通用格式。对于包含详细信息的资源，如队列，可以使用"l"结尾的指令查看资源的列表，如 `qconf -sql` ，或者用"a"和"m"指令进行资源的操作。

仅仅告知GridEngine执行节点的存在还不够，你得把它加到队列里，然后告诉队列这个节点可以分配多少个任务槽。首先要明确新节点的cpu核心数目：

```
grep proc /proc/cpuinfo | wc -l
```

假设是48，那么你可以用一个略小于这个数字的值，如40，作为任务槽的数目。使用 `qconf -mq all.q` 编辑队列资源，将新的节点添加到主机列表中，设置任务槽数目，举例如下

```
qname                 all.q
hostlist              gridnode1.research.acme.com,gridnode2.research.acme.com
<省略若干行>
slots                 30,[gridnode1.research.acme.com=48],[gridnode1.research.acme.com=48
<省略若干行>
```

在slots项中，开头的30是一个默认值，如果不特别指定，你可以不必把节点添加到这一项。除了这种方法以外，你还可以用主机群的方式简化 `hostlist` 项的配置。方法就是，使用 `qconf -ahgrp @allhosts` 创建一个主机群，你也可以用过 `qconf -mhgrp @allhosts` 修改现有的主机群来添加你的新节点。如果使用这种方法，那么上面配置文件的第二行就应当写为

```
hostlist            @allhosts
```

用哪种方法配置取决于你自己的选择，对于小型队列，直接写主机列表可能就够了。

用于显示GridEngine所有主机列表，以及每个主机属于哪个队列的指令是 `qhost -q` ，结果示例如下

```
# qhost -q
HOSTNAME                ARCH         NCPU  LOAD  MEMTOT  MEMUSE  SWAPTO  SWAPUS
----------------------------------------------------------------------------------
global                  -               -     -       -       -       -       -
a01.clsp.jhu.edu        lx26-amd64     24 12.46  126.2G   11.3G   86.6G  213.7M
   all.q                BIP   0/6/20
a02.clsp.jhu.edu        lx26-amd64     24 16.84  126.2G   12.4G   51.3G  164.5M
   all.q                BIP   0/18/20
<省略若干行>
```

如果你在"BIP"的位置看见的是"E"，那说明这个节点出现问题了。其他你不会想看到的标志是，"a"表示警告（通常表示节点的情况不太好），"u"标志状态无法获取，"d"表示这个节点被GridEngine管理员禁用了。有时候运行任务出现问题时，节点会出现"E"标志，这通常是由NFS或其他挂载的问题导致的。你可以清除错误标志

```
qmod -c all.q@a01
```

但是，如果节点真的存在严重问题，最好先修复它。这时候你可以禁用它

```
qmod -d all.q@a01
```

然后用 `qmod -e all.q@a01` 来恢复。

GridEngine出现问题的一种典型症状是当你觉得节点可用时，任务一直在等待。最简单的调试方法是用qstat查看任务标志，然后用 `qstat -j <任务标志>` 查看任务没有被执行的具体原因。

你可以用下面的指令查看所有用户提交的任务

```
qstat -u '*'
```

让你的集群保持稳定

在这一部份，我们给出一些小提示，以便让你的集群保持稳定并更好的配合Kaldi的使用。

内存耗尽（**OOM**）

计算集群崩溃的一个重要原因就是内存耗尽，Linux自带的OOM杀手并不太好，当你内存耗尽时，可能会杀掉某个重要的系统进程，导致很难诊断的后果。即便没有杀掉任何进程，malloc()的调用也会失败，而很少有程序能友善的处理这个问题。在CLSP集群中，我们自己编写了一个OOM杀手，使用root执行。并编写了相应的初始化脚本。当我们的OOM杀手检测到内存过载时，它会杀掉任意非系统用户进程中占用内存最大的那个。通常这样做是很安全的。这些脚本在我们的kluster项目中开源了，你可以用下面的方式获取并添加到你的系统中。要想让下面的指令正常工作，你需要确认你的系统初始化脚本是LSB风格的，例如Debian Wheezy。在Debian的下一个发布版，jesse中，根本不存在初始化脚本，而是使用systemd。如果你搞清楚了如何在systemd中使用下面的工具，请告诉我们。首先执行

```
sudo bash
```

然后作为root执行

```
apt-get install -y subversion
svn cat https://svn.code.sf.net/p/kluster/code/trunk/scripts/sbin/mem-killer.pl > /sbin/m
chmod +x /sbin/mem-killer.pl
cd /etc/init.d
svn cat https://svn.code.sf.net/p/kluster/code/trunk/scripts/etc/init.d/mem-killer >mem-k
chmod +x mem-killer
update-rc.d mem-killer defaults
service mem-killer start
```

如果某个进程被杀， `mem-killer.pl` 会告知管理员和那个用户。但前提是你的系统有邮件发送功能。

共享文件系统（NFS）

我们不会在这里给出NFS安装教程，但是我们希望说明一些潜在的问题，并给出一些可行的解决方法。首先，NFS不是唯一的共享文件系统，还有其他更新的分布式文件系统，但是我们并没有相关经验。

如果配置不当，NFS的性能可能会非常糟糕。下面是我们使用的配置，是用/etc/fstab获取的。其实这不是我们真正使用的方法（我们使用NIS和automount），但是下面的方法更简单。

```
# grep a05 /etc/fstab
a05:/mnt/data /export/a05 nfs rw,vers=3,rsize=8192,wsize=8192,acdirmin=5,acdirmax=8,hard,
```

"vers=3"表示我们使用NFS版本3，我们尝试过版本4，但是经常崩溃。

acdirmin=5和acdirmin=8选项指定了NFS重读缓存目录信息的最短和最长等待时间。默认值是30和60秒。这一点对于Kaldi脚本很重要，因为我们用GridEngine提交的脚本在被提交之前才刚刚生成，所以默认的值会导致脚本的执行节点上无法获取。在前面我们提到了 `/var/lib/gridengine/default/common/prolog.sh` 这个脚本指定等待14秒。很明显，14秒大于8秒，也就是说prolog脚本等待的时间比NFS最长的目录刷新周期还长。

hard选项也很重要，它表示如果服务器忙，客户端会等待而不是返回一个错误（例如fopen返回错误）。如果指定soft，Kaldi会崩溃。hard是默认选择，所以可以不用管它。

proto=tcp也是目前Debian的默认值，另外一种选择是proto=udp。如果网络比较拥挤，TCP协议更稳定，这是我们的经验。

rsize=8192,wsize=8192是数据包的大小，它们对于NFS的性能来说是关键因素。Kaldi的读写操作通常连在一起，并且通常不在一个文件内索引，因此大的数据包更好。

另外一个你可以调节的是NFS服务器的线程数。用如下方法查看

```
/$ head /etc/default/nfs-kernel-server
# Number of servers to start up
RPCNFSDCOUNT=64
```

更改这个文件，然后重启服务（Debian中：service nfs-kernel-server restart）。显然，最好不要让这个数字小于可能同时访问NFS服务的客户端的数目，你可以用下面的指令，通过retrans观察你的线程数是不是过小了

```
nfsstat -rc
Client rpc stats:
calls      retrans    authrefrsh
434831612   6807461    434979729
```

这个例子中的retrans非常大，理想状态下应该是0。当我们大部分节点在工作时，我们设置的NFS线程数是24，这个值小于可能访问的客户端的数量，所以retrans值比较高。如果有很多客户端同时访问，并且都在写大量数据，它们会占满全部线程，当另一个客户端企图连接时，你会在客户端的日志里发现如下错误 `nfs: server a02 not responding, still trying.` 。有些时候这会导致任务失败。如果你使用automount，甚至可能导致一些并没有连接NFS服务的任务失败或被延迟（automount有个脑残的，单线程的设计，所以一个automount请求的失败会导致其他所有automount请求被挂起）。

其他常见问题

这一部份我们介绍在计算集群中观察到的一些现象，以及如何配置和管理计算集群。

在CLSP，我们使用多个NFS主机，不仅仅用一两个。实际上，我们大部分的节点还用NFS输出数据。如果你也这样，应该考虑使用我们的mem-killer.pl，否则你会遇到由于用户误操作导致的大量内存耗尽问题。对于很多人共享的任务队列，使用多个文件服务器是个更好的选择，因为用户会难以避免的过载文件服务器。如果只有一两个文件服务器，那么你的队列的性能会收到很大影响。CLSP的单个文件服务器的网络带宽是相当低的，因为成本问题，我们使用1G以太网。但是互相之间用一个强大（昂贵）的Cisco路由连接，所以总的来说网络不会成为瓶颈。这意味着单个文件服务器可能很容易崩溃，但是由于存在大量独立文件服务器，通常只会影响使用那个崩溃的服务器的用户。

当我们检测到某个文件服务器负载较大（比如响应缓慢，或者用dmesg输出nfs: server a15 not responding之类的错误），我们尝试追踪问题的缘由。通常，这都是由用户的不良习惯导致的，例如，用户提交了太多大量占用IO的任务。通常，通过对比iftop和qstat的输出，我们可以知道是哪个用户导致了这个问题。为了保持队列问题，有必要纠正用户的使用习惯，限制类似任务的提交数量。通过给用户发送邮件，提醒他们修改自己的配置。如果不这样做的话，队列资源是根本无法使用的，因为用户会坚持他们的不良习惯。

与我们类似，很多其他的机构也有多个文件服务器，但仍然将所有的流量指向某一个服务器，因为服务器是逐步购买扩充的，而他们总是在最新买的那个服务器上分配新的空间。这样做很蠢。在CLSP，我们将所有的NFS服务器设为全局可写，告诉用户如何用自己的帐户在自己选择的服务器上创建文件夹，避免每个人都去找管理员开辟空间的麻烦。我们编写了脚本，当任何一个文件服务器的用量超过95%时管理员会收到邮件通知，内容包括哪个文件夹占用了大量空间。然后可以告诉相关的用户，让他门删除一些自己的数据，或者根据需要由管理员删除，比如用户已经离开。我们还编写了一个脚本，用来检查在队列中哪个用户占用了大部分资源，并用邮件通知他们在哪里占用了多少资源。另外，当出现一些特殊情况时，管理员也会要求用户清理空间，比如某个高年级学生无缘由地使用了大量的存储空间。存储空间任何时候都不应当成为瓶颈，因为在大部分情况下都有95%是垃圾，没人关心，甚至没有人记得。这个机制只是为了找到有责任清理的那个人，告诉他们自己都保存了哪些没用的东西，让他门的工作效率更高。

另外一个有用的技巧是，在一个不用于实验的文件服务器上分配home目录，这可以保证用户总能得到快速响应，即便其他的用户在做一些愚蠢的事。这也使我们的备份工作更容易进行。我们只备份home目录，而不是那些用于搭建实验环境的NFS卷，并且我们明确告诉用户，这些内容是不进行备份的（当然，有时候学生们会忘记备份他们的重要数据，导致数据丢失）。禁止在home目录下进行实验是一个必要的、需要强调的措施。我们不得不频繁的告诉用户停止那些使用home目录下数据的并行任务。这是导致计算集群出错的最常见的原因。

Kaldi工具

在线解码器

警告：这是一个旧版本的在线解码器的文档。最新的在线解码文章在这里。

在kaldi 的工具集里有好几个程序可以用于在线识别。这些程序都位在src/onlinebin文件夹里，他们是由src/online文件夹里的文件编译而成(你现在可以用make ext 命令进行编译).这些程序大多还需要tools文件夹中的portaudio 库文件支持， portaudio 库文件可以使用tools文件夹中的相应脚本文件下载安装。这些程序罗列如下：

- online-gmm-decode-faster: 从麦克风中读取语音，并将识别结果输出到控制台
- online-wav-gmm-decode-faster:读取wav文件列表中的语音，并将识别结果以指定格式输出。
- online-server-gmm-decode-faster:从UDP连接数据中获取语音MFCC向量，并将识别结果打印到控制台。
- online-net-client :从麦克风录音，并将它转换成特征向量，并通过UDP连接发送给online-server-gmm-decode-faster
- online-audio-server-decode-faster :从tcp连接中读取原始语音数据，并且将识别结果返回给客户端
- online-audio-client :读出wav文件列表，并将他们通过tcp 发送到online-audio-server-decode-fater,得到返回的识别结果后，将他存成指定的文件格式

代码中还有一个java版的online-audio-client ,他包含更多的功能，并且有一个界面。另外，还有一个与GStreamer 1.0兼容的插件，他可以对输入的语音进行识别，并输出的文字结果。这个插件基于onlinefasterDecoder,也可以做为在线识别程序

在线语音服务器

online-server-gmm-decode-faster 和online-audio-server-decode-faster的主要区别是各自的输入不同，前者接受特征向量做为输入，而后者则接收原始音频。后者的好处是他可以处理任意客户端直接传过来的数据：不管他是互联网上的计算机还是一部移动设备。最主要的是客户端不需要知道训练模型使用了何种特征集，只要设定好预先定义的采样率和位深度，他就可以跟服务器进行通信。使用特征向量做为输入的服务器的一个好处是，在客户端和服务器之间传输的数据比音频服务器小得多，而这只需使用简单的音频代码就可以实现。

在online-audio-client和online-audio-server-decode-faster之间的通信包括两个步骤：第一步客户端将原始音频数据包发送给服务器，第二步服务器将识别结果发回给客户端。这两步可以同步进行，这意味着解码器不用等音频数据发送完毕，就可以在线输出结果，这给未来新的应用提供了很多可以扩展的功能。

音频数据

音频数据格式必须是采样率16KHz，16-bit 位深，单声道，线性PCM编码的硬编码数据。 通信协议数据分为块和前缀，每一个块包含4字节指示此块的长度。这个长度值也是一个long型little-endian值,可以是正也可以是负（因为是16-bit采样)。最后一个长度为0的包被当作音频流的结束，这将迫使解码器导出所有剩余的结果，并且结束识别过程

识别结果

服务器返回两种类型的结果。时间对齐结果和部分结果。

解码器识别出每一个词都会立即发送一个部分结果，而当解码器识别到一次发音结束时则会发送一个时间对齐结果（这可能是也可能不是一次停顿或者句子结束）。

每一个部分结果包会有以字符串：“PARTIAL:"开头，后面跟一个词，每一个词发送一次不同的部分结果包。时间对齐结果包会以字符串："RESULT:"开头。后面是以逗号为分格的key=value参数列表，这些参数列表包含一些有用信息。

- NUM:后面词的个数
- FORMAT :返回结果的格式，目前只支持WSEC格式（词-开始-结束-置信度),未来会允许修改
- RECO-DUR: 识别语音所使用的时间。（以秒计的浮点型值）
- INPUT-DUR:输入语音的时间长度（以秒计的浮点型值) 你可以用识别时间除以输入时间得到解码器的实时识别速度。当服务器将识别结果全部返回后会发送一个"RESULT:DONE", 这种情况下服务器会等待输入或是断开。

在数据头部后面包含有NUM行以FORMAT格式组成的词行，现在的WSEC格式，只是简单以逗号分隔的四个部分：词，起始时间，终止时间（以秒计的浮点数），置信度（0-1之间浮点数）要记住这些词只是在词典中的编码，因此客户端必须执行转换操作。online-audio-client在生成webvtt文件时不进行任何字符转换，因此你需要用iconv将结果文件转换成UTF8

服务器返回的结果的例子如下：

```
RESULT:NUM=3,FORMAT=WSEC,RECO_DUR=1.7,INPUT_DUR=3.22
 one,0.4,1.2,1
 two,1.4,1.9,1
 three,2.2,3.4,0.4
 RESULT:DONE
```

使用举例

命令行启动服务器：

```
online-audio-server-decode-faster --verbose=1 --rt-min=0.5 --rt-max=3.0 --max-active=6000
```

各参数含义如下：

- final.mdl :声音模型文件
- HCLG.fst :完全的fst
- words.txt:发音词典（将词的id号映射到对应的字符上)
- '1:2:3:4:5' ：sil 的id号
- 5010 ：服务器的端口号
- word_boundary_phones.int ：词对齐时使用的音素边界信息
- final.mat ：特征的LDA矩阵

命令行启动客户端

```
online-audio-client --htk --vtt localhost 5010 scp:test.scp
```

各参数含义如下

- –htk 结果存成htk 标注文件
- –vtt 结果存成webvtt文件
- localhost 服务器地址
- 5010 服务器端口号
- scp:test.scp Wav文件的列表文件

命令行方式启动java客户端

```
java -jar online-audio-client.jar
```

或者直接在图形界面上双点jar文件

GStreamer 插件

Kaldi 工具集为GStream 多媒体流框架提供了一个插件（1.0或兼容版)。此插件可以做为过滤器，接收原始音频做为输入并将识别结果做为输出。这个插件的主要好处是他使得kaldi 在线语音识别功能对所有支持GStreamer1.0的编程语言都可用(包括python,ruby,java,vala等等)。这也使得可以把kaldi 在线解码器集成到支持GStreamer 通信标准的应用程序中

安装

GStreamer 插件的源码位于src/gst-plugin 目录。要完成编译，kaldi工具集的其它部分必须使用共享库进行编译。因此在配置的时候必须使用--shared 参数，同进需要编译在线扩展（make ext).确保支撑GStreamer 1.0开发头文件的包已经安装在你的系统中。在Debian

Jessie系统版本中，需要包libgstreamer1.0-dev。在Debian Wheezy版本中，GStreamer 1.0可以从backports源中下载到。需要安装gstreamer1.0-plugins-good和gstreamer1.0-tools这两个包。演示程序还需要PulseAudio Gstreamer插件，包名：gstreamer1.0-pulseaudio

最后，在src/gst-plugin目录中运行make depend和make。这样就会生成一个文件src/gst-plugin/libgstkaldi.so，他包含了GStreamer 插件。为确保GStreamer 可以找到kaldi 插件，必须把src/gst-plugin目录加到他的插件搜索目录。因此需要把这个目录加入环境变量中 export GST_PLUGIN_PATH=$KALDI_ROOT/src/gst-plugin 当然，你需要把$KALDI_ROOT修改成你自己文件系统中kaldi所在 的根目录。

未完待续。