基于Kaldi+GStreamer搭建线上的实时语音识别器

本文主要介绍如何基于Kaldi语音识别工具箱和两个GStreamer插件(gst-kaldi-nnet2-online、kaldi-gstreamer-server)来搭建线上的实时语音识别系统。Kaldi是目前非常流行的语音识别开源工具箱，可用来训练声学模型，包括传统的GMM、SGMM和流行的DNN、TDNN等；也可以用来搭建基于WFST的解码器。GStreamer是开源的多媒体数据处理工具，基于GStreamer框架可以编写处理音频或视频的应用程序。Tanel Alumäe基于GStreamer编写了两个插件，其中一个是Kaldi实时解码器插件gst-kaldi-nnet2-online，gst-kaldi-nnet2-online整合了Kaldi实时解码器，使用GStreamer的音频处理功能，可实现对音频的实时解码；另外一个插件是kaldi-gstreamer-server，kaldi-gstreamer-server使用Tornado(python的第三方库)在服务器端和客户端之间建立基于Websocket协议的全双工通信，服务器端是Kaldi实时解码器gst-kaldi-nnet2-online，负责接收客户端传输进来的音频数据流、实时解码并返回识别结果；客户端使用GStreamer录音并将音频数据传送到服务器端，客户端既可以是Android应用，也可以是Javascript实现的Web应用。

Kaldi

实时识别系统的好坏取决于语音识别的性能，语音识别包含特征提取、声学模型、语言模型、解码器等部分。Kaldi工具箱集成了几乎所有搭建语音识别器需要用到的工具。本文首先介绍Kaldi工具箱的安装，由于声学模型的训练需要用到GPU，所以在编译Kaldi工具箱之前，首先需要安装配置CUDA。(注：如果本机没有GPU显卡，可以跳过安装CUDA这一步骤，后续使用现成的训练好的声学模型来测试。)

下载安装CUDA工具包

CUDA工具包为C和C++开发者提供了一整套开发基于GPU加速的应用程序的开发环境，包括编译器、数学运算库等。目前英伟达公司提供的CUDA工具包的下载和安装已变得十分简单，具体操作参见英伟达官网教程。此处推荐使用Ubuntu14.04的Linux系统，直接下载deb安装包，使用apt-get安装。如果之前安装过CUDA工具包，推荐先升级到最新版本，再安装Kaldi，以免编译Kaldi时出现错误。

下载安装Kaldi

首先使用git工具下载最新的kaldi版本。

```
git clone https://github.com/kaldi-asr/kaldi.git kaldi --origin upstream
```

接下来安装Kaldi，首先进入tools目录下，检查所有需要提前满足的依赖关系，接着下载并安装所有kaldi用到的外部库(OpenFst、ATLAS、CLAPACK等)，详细过程参见tools目录下的INSTALL文件。

```
cd kaldi/tools/

./extras/check_dependencies.sh

make -j 4
```

接下来是编译kaldi源文件，详细过程参见src目录下的INSTALL文件。

```
cd ../src

./configure --shared

make depend -j 4

make -j 4
```

以上过程中如果某个步骤出错，需要停止继续操作，检查出错原因。此处提供了有关Kaldi编译过程的详细信息。

声学模型训练

在语音识别领域，深度神经网络模型已经取代高斯混合模型，成为主流的声学模型。目前Kaldi支持feedforward neural network、TDNN、LSTM等多种类型的神经网络模型，在训练策略上，也支持单GPU训练、多CPU或多GPU并行训练。发展至今，在神经网络声学模型的训练上，Kaldi存在三个不同分支，分别是nnet、nnet2和nnet3。nnet由*Karel Vesely*维护，nnet2和nnet3主要由*Daniel Povey*维护。nnet和nnet2都是用来训练feedforward neural network的，虽然nnet2一开始是基于nnet改写的，但目前两者已有很大差别，nnet使用单GPU训练，实现和修改相对容易；nnet2既支持多GPU并行训练，也支持多CPU多线程并行训练。nnet3是最新的DNN声学模型训练工具，与nnet和nnet2相比，nnet3支持RNNs和LSTMs等神经网络模型。有关这三种不同的神经网络模型训练机制，详细介绍可参考Deep Neural Networks in Kaldi。

目前，nnet2和nnet3在训练速度和模型性能上都要优于nnet，所以推荐使用nnet2或nnet3。考虑到目前RNNs和LSTMs已经开始被广泛使用，所以最好尝试使用nnet3训练声学模型。由于gst-kaldi-nnet2-online一开始只支持nnet2，直到2016-10-14才开始支持nnet3模型，gst-kaldi-nnet2-online插件对于nnet3的支持还不是很好，本文后面也会涉及到在gst-kaldi-nnet2-online上做的一些小的修改以便更好的支持nnet3模型。

nnet3 chain model

此处简单介绍如何使用nnet3工具训练chain model，作为gst-kaldi-nnet2-online调用的模型。chain model属于DNN-HMM模型的一种，其模型的准确度略优于传统的DNN-HMM，解码的速度是传统的DNN-HMM三倍左右，因此对于实时解码任务，最好选用chain model。chain model的详细介绍可参考'Chain' models。

Kaldi提供了训练chain model的脚本，egs/swbd/s5c/local/chain/tuning目录下有不同版本的chain model训练脚本，新的版本都是在旧的版本上加入一些新特性，目前已更新到run_tdnn_7f.sh这个脚本。本文使用run_tdnn_7d.sh训练chain model。在介绍chain model的训练之前，需要首先介绍ivector特征提取。由于实时的语音识别无法使用CMVN对输入特征做speaker normalization，Kaldi的实时语音识别系统采用ivector特征来承载说话人信息。因此，在训练nnet3 online chain model之前，首先要训练一个ivector extractor，解码阶段，会使用提前训练好的ivector extractor计算音频的ivector，再将ivector提供给神经网络，来区分不同说话人。ivector广泛应用在说话人识别、语种识别等领域，详细的介绍可以参考MIT的这个tutorial。下面是对Kaldi训练ivector extractor的训练过程的简单介绍，参考的是egs/hkust/s5/local/nnet3/run_ivector_common.sh这个脚本。

egs/hkust/s5/local/nnet3/run_ivector_common.sh这个脚本总共包含7个步骤：

- 第一步是随机地将训练集中原始音频(original data)的音量调高或调低，得到train_hires训练集，之后在train_hires上提取更高分辨率的MFCC特征(mel-bin个数为40，保留前40维的倒谱系数，即保留所有的倒谱信息)；
- 第二步是基于已有的切分信息，使用LDA+MLLT变换的特征训练triphone模型；
- 第三步是使用steps/online/nnet2/train_diag_ubm.sh脚本，训练UBM(UBM，通用背景模型，用来训练ivector extractor)，训练数据为train_hires；
- 第四步是使用第三步得到的UBM模型，训练online ivector extractor，训练脚本为steps/online/nnet2/train_ivector_extractor.sh；
- 第五步首先是修改原始音频(original data)的音频播放速度，分别是0.9倍速和1.1倍速，然后将调整速率后得到的两套音频和原始音频(original data)放到一起，命名为train_sp训练集(此处的sp代表speed perturbed)；接着复制一份train_sp，更名为train_sp_hires。对train_sp提取普通的前13维倒谱的MFCC参数，对train_sp_hires提取高分辨率的MFCC参数；
- 第六步首先通过steps/online/nnet2/copy_data_dir.sh将训练数据中每条音频都复制一份，相当于把训练数据扩充一倍。接着使用steps/online/nnet2/extract_ivectors_online.sh脚本提取扩充之后的训练数据的ivectors；
- 第七步是使用steps/online/nnet2/extract_ivectors_online.sh提取测试集的ivectors。

提取完训练集和测试集的ivectors之后，接下来是训练chain model；以下是对egs/swbd/s5c/local/chain/tuning/run_tdnn_7d.sh这个脚本的简单介绍。脚本里在训练chain model之前，使用local/nnet3/run_ivector_common.sh提取ivectors，上文已经介绍，此处跳过。整个chain model的训练和测试总共包含七个步骤：

- 第一步是使用steps/align_fmllr_lats.sh对训练集train_sp做切分，输出存成lattice格式。这一步的输出在后面训练神经网络时会用到；
- 第二步是复制data/lang文件夹,命名为data/lang_chain_2y，将HMM对应的topo文件修改成每个音素只有一个状态；
- 第三步使用上一步得到的新的data/lang_chain_2y，由steps/nnet3/chain/build_tree.sh脚本创建决策树；
- 第四步是生成神经网络初始化的配置文件，生成的文件存放在exp/chain/tdnn_7d/configs
- 第五步是训练神经网络，脚本是steps/nnet3/chain/train.py，此处要注意的是，在执行脚本之前，要先设置GPU计算模式 sudo nvidia-smi -c 1。可以使用 nvidia-smi --help命令查看各种参数的功能，以下是-c参数的功能：

  -c, --compute-mode= Set MODE for compute applications:

  ```
  0/DEFAULT, 1/EXCLUSIVE_PROCESS,
  2/PROHIBITED
  ```

  另外，由于此处的egs操作会占用大量空间(一两千小时的数据大概需要1T左右的空间)，所以必须保证--egs.dir对应的文件夹空间充足(可以使用ln -s 命令将此文件夹链接到挂载的磁盘上)。
- 第六步是构建decode需要的解码图；
- 第七步是识别测试集数据，得到测试结果。

以上是nnet3 chain model的整个训练和测试过程。在生成的所有文件中，gst-kaldi-nnet2-online插件需要用到文件包括：final.mdl、HCLG.fst、words.txt、conf文件夹；conf文件夹里又包括：final.dubm、final.ie、final.mat、global_cmvn.stats、ivector_extractor.conf、mfcc.conf、online_cmvn.conf、splice.conf。有了以上的这些文件，就可以使用gst-kaldi-nnet2-online实现实时的语音识别了，再配合kaldi-gstreamer-server，即可实现线上的实时语音识别系统。以下是有关gst-kaldi-nnet2-online、kaldi-gstreamer-server这两个插件的详细介绍。

GStreamer插件

安装gst-kaldi-nnet2-online

在安装gst-kaldi-nnet2-online之前，首先要保证已正确安装Kaldi工具箱。确保Kaldi安装正确之后，还要安装gstreamer-1.0。

> sudo apt-get install gstreamer1.0-plugins-bad gstreamer1.0-plugins-base gstreamer1.0-plugins-good gstreamer1.0-pulseaudio gstreamer1.0-plugins-ugly gstreamer1.0-tools libgstreamer1.0-dev

如果是Ubuntu14.04以上，可以直接执行以上命令。如果是Ubuntu12.04，需要首先执行以下两条命令来添加1.0版的gstreamer backport ppa。

> sudo add-apt-repository ppa:gstreamer-developers/ppa
>
> sudo apt-get update

接着是安装Jansson-dev，安装这个软件包的目的是解析JSON格式的文件。

```
sudo apt-get install libjansson-dev
```

接下来就是下载编译安装gst-kaldi-nnet2-online。

```
git clone https://github.com/alumae/gst-kaldi-nnet2-online.git

cd src

make depend

KALDI_ROOT=path of your kaldi installation directory

make
```

整个编译过程如果没有出现错误，那么应该会在src目录下生成libgstkaldionline2.so这个文件。

接下来设置GST_PLUGIN_PATH变量。为了避免每次打开新的terminal都要重新设置，可用将该变量添加到~/.bashrc中。

```
sudo apt-get install vim

vim ~/.bashrc

在文件最下方添加一行 export GST_PLUGIN_PATH= your gst-kaldi-nnet2-online installation directory/src
```

接下来查看gst-kaldi-nnet2-online这个插件的详细信息。

```
gst-inspect-1.0 kaldinnet2onlinedecoder | less
```

如果安装正确，会显示下列输出：

```
Factory Details:
  Rank                  none (0)
  Long-name               KaldiNNet2OnlineDecoder
  Klass                 Speech/Audio
  Description            Convert speech to text
  Author                Tanel Alumae tanel.alumae@phon.ioc.ee

Plugin Details:
  Name                   kaldinnet2onlinedecoder
  Description            kaldinnet2onlinedecoder
  Filename               /home/lijian/kaldionline/gstkaldionline/src/libgstkaldionline2.so
  Version               1.0
  License               unknown
  Source module           Kaldi
  Binary package          GStreamer
  Origin URL             http://gstreamer.net/

...
```

本地的语音识别系统

安装完gst-kaldi-nnet2-online插件之后，配合kaldi工具箱，即可实现实时的语音识别。gst-kaldi-nnet2-online提供了两个使用样例，位于gst-kaldi-nnet2-online/demo目录下。第一个是比较简单的在命令行界面下识别一整段音频文件；第二个样例展示了使用GUI界面控制录音和实时显示识别结果。接下来是对两个样例的详细介绍。

在运行两个样例之前，要先从Kaldi官网下载训练好的英语的nnet2模型，直接运行gst-kaldi-nnet2-online/demo路径下的prepare-models.sh脚本。

```
cd demo

chmod +x prepare-models.sh

./prepare-models.sh
```

下载的文件包括:

final.mdl；

ivector_extractor文件夹，包含 final.ie、final.dubm、final.mat、global_cmvn.stats四个文件；

conf文件夹，包含ivector_extractor.conf、online_nnet2_decoding.conf、mfcc.conf、online_cmvn.conf、splice.conf五个文件；

HCLG.fst；

words.txt；

下载完成后，直接运行transcribe-audio.sh这个脚本。

```
chmod +x transcribe-audio.sh

./transcribe-audio.sh dr_strangelove.mp3
```

在一大堆警告之后，会得到以下输出信息。

```
LOG (ComputeDerivedVars():ivector-extractor.cc:183) Computing derived variables for iVector extractor
LOG (ComputeDerivedVars():ivector-extractor.cc:204) Done.
```

接下来是输出的识别结果，可以看到识别结果是一句一句输出的，句子和句子之间有较长的停顿。以上运行的是第一个样例，从transcribe-audio.sh这个脚本中可以看出如何使用gst-kaldi-nnet2-online。以下是脚本的核心部分。

```
GST_PLUGIN_PATH=../src gst-launch-1.0 --gst-debug="" -q filesrc location=$audio ! decodebin !
audioconvert ! audioresample ! \
kaldinnet2onlinedecoder \
  use-threaded-decoder=true \
  model=final.mdl \
  fst=HCLG.fst \
  word-syms=words.txt \
  feature-type=mfcc \
  mfcc-config=conf/mfcc.conf \
  ivector-extraction-config=conf/ivector_extractor.fixed.conf \
  max-active=7000 \
  beam=11.0 \
  lattice-beam=5.0 \
  do-endpointing=true \
  endpoint-silence-phones="1:2:3:4:5:6:7:8:9:10" \
  chunk-length-in-secs=0.2 \
! filesink location=/dev/stdout buffer-mode=2
```

GST_PLUGIN_PATH=../src是设置gst插件的路径，上文中已经在~/.bashrc中设置过，此处可以忽略。gst-launch-1.0是gstreamer的工具，可以使用以下命令查看该工具的详细信息。

```
gst-launch-1.0 --help-gst
```

kaldinnet2onlinedecoder是gstreamer插件，use-threaded-decoder、model等都是插件对应的参数。可以使用gst-inspect-1.0 kaldinnet2onlinedecoder查看这些参数的详细信息(Element Properties部分)。

```
Element Properties:
  name  : The name of the object

  parent  : The parent of the object

  nnet-mode  : 2 for nnet2, 3 for nnet3

  silent  : Silence the decoder

  model  : Filename of the acoustic model

  fst  : Filename of the HCLG FST

  word-syms  : Name of word symbols file (typically words.txt)

  phone-syms  : Name of phoneme symbols file (typically phones.txt)

  do-phone-alignment  : If true, output phoneme-level alignment

  do-endpointing  : If true, apply endpoint detection, and split the audio at endpoints

  adaptation-state  : Current adaptation state, in stringified form, set to empty string to reset

  inverse-scale  : If true, inverse the acoustic scaling of the output lattice

  lmwt-scale  : LM scaling for the output lattice, usually in conjunction with inverse-scaling=true

  chunk-length-in-secs: Smaller values decrease latency, bigger values (e.g. 0.2) improve speed if
multithreaded BLAS/MKL is used

  traceback-period-in-secs: Time period after which new interim recognition result is sent

  lm-fst  : Old LM as FST (G.fst)
```

接下来运行第二个样例：

```
python gui-demo.py
```

在一堆警告之后，弹出录音的GUI，点击Speak按钮后开始实时识别语音。以下是参考Gtk、GStreamer、和Gdk的用户手册对gui-demo.py程序做的详细注解。

```
import sys
import os
import gi  # import PyGObject
gi.require_version('Gst','1.0')   # 确保Gst的版本是1.0
from gi.repository import GObject, Gst, Gtk, Gdk
GObject.threads_init() # Before using Python threads, or libraries using threads
                       # (GStreamer for example), you have to call
                       # GObject.threads_init()

                       # Since PyGObject 3.10.2, calling GObject.threads_init() this is
                       # no longer needed

                       # 可以用print gi.__version__ 命令查看PyGObject版本

Gdk.threads_init() # The gtk.gdk.threads_init() function initializes PyGTK to use the
                   # Python macros that allow multiple threads to serialize access to
                   # the Python interpreter (using the Python Global Interpreter Lock
                   # (GIL)).

Gst.init(None) # 初始化Gstreamer库

class DemoApp(object)
    """GStreamer/Kaldi Demo Application"""
    def __init__(self): # 创建DemoApp时首先运行__init__()函数
        """Initialize a DemoApp object"""
        self.init_gui() # 初始化GUI界面
        self.init_gst() # 初始化语音识别器

    def init_gui(self):
        """Initialize the GUI components"""
        self.window = Gtk.Window(title="") # 创建GUI主窗口，title为标题栏内容

        self.window.connect("destroy",self.quit) # 将'destroy'信号与quit()这一操作连接起来
                                                   # Gtk.main()一旦检测到'estroy'，就执行
                                                 # quit()函数

        self.window.set_default_size(400,200)    # 设置默认的窗口大小

        self.window.set_border_width(10)         # 设置窗口外部边框的宽度(外边框距离文本框)

        vbox = Gtk.VBox()                        # VBox容器可以将放在其中的子控件排成一列

        self.text = Gtk.TextView()               # 创建可编辑添加文字的多行文本显示窗口

        self.textbuf = self.text.get_buffer()    # 获取文本显示窗口对应的缓冲区

        self.text.set_wrap_mode(Gtk.WrapMode.WORD) # 设置文本显示窗口换行的模式，共有以下几种
                                                   # gtk.WRAP_NONE, gtk.WRAP_CHAR,
                                                   # gtk.WRAP_WORD or gtk.WRAP_WORD_CHAR

        vbox.pack_start(self.text,True,True,1)     # 将text控件放到vbox容器中，可查看
                                                   # gst class reference中

                                                   # gtk.Box.pack_start()的参数介绍
```

```
        self.button = Gtk.Button("Speak")          # 创建标签为'Speak'的Button控件

        self.button.connect('clicked',self.button_clicked) # 将'clicked'信号
                                                           # 与'button_clicked'连在一起

        vbox.pack_start(self.button, False, False, 5) # 将button控件放到vbox容器中

        self.window.add(vbox)                          # 将vbox控件添加到window控件中

        self.window.show_all()                         # The show_all() method
                                                       # recursively shows the widget,
                                                       # and any child widgets (if the
                                                       # widget is a container).
        def quit(self, window):
            Gtk.main_quit()                            # 定义quit()函数，执行后Gtk结束当前的
                                                       # main()循环
        def init_gst(self):
            """Initialize the speech components"""
            self.pulsesrc = Gst.ElementFactory.make("pulsesrc", "pulsesrc")
                                                       # Create a new element of the type
                                                       # defined by the given element
                                                       # factory

                                                       # pulsesrc是gstreamer的一个插件，用来
                                                  # 获取声音 (Captures audio from a
                                                       # PulseAudio server)
            if self.pulsesrc == None:
                print >> sys.stderr, "Error loading pulsesrc GST plugin. You probably need the
gstreamer1.0-pulseaudio package"
                sys.exit()                              # ElementFactory.make失败时，返
                                                       # 回None，而不是抛出异常，所以此处检测
                                                       # 是否成功创建pulsesrc element

            self.audioconvert = Gst.ElementFactory.make("audioconvert", "audioconvert")
                                                       # 导入audioconvert插件
                                                       # Audioconvert converts raw audio
                                                       # buffers between various possible
                                                       # formats. It supports integer to
                                                       # float conversion, width/depth
                                                       # conversion, signedness and
                                                       # endianness conversion and
                                                       # channel transformations (ie.
                                                       # upmixing and downmixing), as
                                                       # well as dithering and noise-
                                                       # shaping.

            self.audioresample = Gst.ElementFactory.make("audioresample", "audioresample")
                                 # 导入audioresample插件
                                                       # audioresample resamples raw
                                                       # audio buffers to different
                                                       # sample rates

            self.asr = Gst.ElementFactory.make("kaldinnet2onlinedecoder", "asr")
```

```
                                                # 导入kaldinnet2onlinedecoder插件
            self.fakesink = Gst.ElementFactory.make("fakesink", "fakesink")
                                                # Dummy sink that swallows
                                                # everything.
            if self.asr:
                model_file="final.mdl"
                if not os.path.isfile(model_file):
                    print >> sys.stderr, "Models not downloaded? Run prepare-models.sh first!"

                    sys.exit(1)                 # 判断当前路径是否存在final.mdl文件，
                                                # 如果不存在，需要运行脚本从kaldi官网下
                                                # 载
                self.asr.set_property("fst", "HCLG.fst")
                self.asr.set_property("model", "final.mdl")
                self.asr.set_property("word-syms", "words.txt")
                self.asr.set_property("feature-type", "mfcc")
                self.asr.set_property("mfcc-config", "conf/mfcc.conf")
                self.asr.set_property("ivector-extraction-config",
"conf/ivector_extractor.fixed.conf")
                self.asr.set_property("max-active", 7000)
                self.asr.set_property("beam", 10.0)
                self.asr.set_property("lattice-beam", 6.0)
                self.asr.set_property("do-endpointing", True)
                self.asr.set_property("endpoint-silence-phones", "1:2:3:4:5:6:7:8:9:10")
                self.asr.set_property("use-threaded-decoder", False)
                self.asr.set_property("chunk-length-in-secs", 0.2) # 以上为设置nnet2模型参
                                                                   # 数
            else:
                print >> sys.stderr, "Couldn't create the kaldinnet2onlinedecoder element."
                if os.environ.has_key("GST_PLUGIN_PATH"):
                    print >> sys.stderr, "Have you compiled the Kaldi GStreamer plugin?"
                else:
                    print >> sys.stderr, "You probably need to set the GST_PLUGIN_PATH
envoronment variable"
                    print >> sys.stderr, "Try running: GST_PLUGIN_PATH=../src %s" % sys.argv[0]
                sys.exit();

            # initially silence the decoder
            self.asr.set_property("silent", True)

            self.pipeline = Gst.Pipeline() # 创建pipeline
            for element in [self.pulsesrc, self.audioconvert, self.audioresample, self.asr,
self.fakesink]:
                self.pipeline.add(element) # 将以上五个插件添加到pipeline里
            self.pulsesrc.link(self.audioconvert)
            self.audioconvert.link(self.audioresample)
            self.audioresample.link(self.asr)
            self.asr.link(self.fakesink)

            self.asr.connect('partial-result', self._on_partial_result)
                                        # 将'partial-result'信号与_on_partial_result连
                                        # 在一起?
```

```
            self.asr.connect('final-result', self._on_final_result)
                                        # 将'final-result'信号与_on_final_result连
                                        # 在一起?
            self.pipeline.set_state(Gst.State.PLAYING)

    def _on_final_result(self, asr, hyp):
        Gdk.threads_enter()
        """Insert the final result."""
        # All this stuff appears as one single action
        self.textbuf.begin_user_action()
        self.textbuf.delete_selection(True, self.text.get_editable())
        self.textbuf.insert_at_cursor(hyp)
        if (len(hyp) > 0):
            self.textbuf.insert_at_cursor(" ")
        self.textbuf.end_user_action()
        Gdk.threads_leave()

    def _on_final_result(self, asr, hyp):
        Gdk.threads_enter()
        """Insert the final result."""
        # All this stuff appears as one single action
        self.textbuf.begin_user_action()
        self.textbuf.delete_selection(True, self.text.get_editable())
        self.textbuf.insert_at_cursor(hyp)
        if (len(hyp) > 0):
            self.textbuf.insert_at_cursor(" ")
        self.textbuf.end_user_action()
        Gdk.threads_leave()

    def button_clicked(self, button):
        """Handle button presses."""
        if button.get_label() == "Speak":
            button.set_label("Stop")
            self.asr.set_property("silent", False)
        else:
            button.set_label("Speak")
            self.asr.set_property("silent", True)

if __name__ == '__main__':
    app = DemoApp()
    Gdk.threads_enter() # The gtk.gdk.threads_enter() function marks the beginning of a
                        # critical section that only one thread can operate within at a
                        # time.

    Gtk.main() # 开启Gtk主循环，持续检测是否有新产生的事件发生

    Gdk.threads_leave() # The gtk.gdk.threads_leave() function marks the end of a
                        # critical section started by the gtk.gdk.threads_enter()
                        # function.
```

以下是Gtk、GStreamer、和Gdk的用户手册，很多类和函数都能在里面找到。

Gtk Class Reference

GStreamer Good Plugins 1.0 Plugins Reference Manual

GStreamer Base Plugins 1.0 Plugins Reference Manual

GStreamer Core Plugins 1.0 Plugins Reference Manual

gtk.gdk Functions

以下是有关Gtk多线程的资料。

Using threads in PyGTK

gst-kaldi-nnet2-online使用nnet3模型

以上两个例子都是使用kaldi的nnet2模型，在前文中提到过，gst-kaldi-nnet2-online直到2016-10-14才开始支持nnet3模型，gst-kaldi-nnet2-online插件对于nnet3的支持还不是很好，以下是对gst-kaldi-nnet2-online做的一些小的修改以便更好的支持nnet3模型。

首先是指定nnet模型的类型。在调用kaldinnet2onlinedecoder插件时，在传入的参数里加入 nnet-mode=3。这样kaldinnet2onlinedecoder就支持nnet3模型了。对应GUI的例子，修改gst-kaldi-nnet2-online/demo/gui-demo.py里面的DemoApp这个类，在init_gst(self)这个函数里，找到asr.set_property这个关键字，在所有asr.set_property之前，加入self.asr.set_property("nnet-mode", 3)。

但是，这样修改后的asr识别的结果不正确，出来的都是

```
filter->nnet3_decoding_config->Register(filter->simple_options)
```

后面加入一行

```
filter->nnet3_decoding_config->decodable_opts.frame_subsampling_factor=3
```

然后在gst-kaldi-nnet2-online文件夹下make clean，再make depend，然后make。最后得到的asr模型识别出来的结果就好多了。

以上是使用Gstreamer的第一个插件gst-kaldi-nnet2-online搭建本地的实时语音识别服务。要想把语音识别作为线上服务，允许客户端程序远程访问，还需要使用Gstreamer的第二个插件kaldi-gstreamer-server。下面是该插件的安装和使用。

安装Kaldi-gstreamer-server

kaldi-gstreamer-server的使用需要依赖Tornado、ws4py这两个websocket库，以及yaml格式解析库YAML、json格式解析库JSON，如果之前没有安装，可以使用以下命令安装相关的python库。

> sudo pip install tornado
>
> sudo pip install ws4py==0.3.2
>
> sudo pip install pyyaml

测试是否已满足所有依赖关系

> python
>
> >>> import tornado
>
> >>> import ws4py
>
> >>> import yaml

```
>>> import json
```

在python解释器下导入以上几个module没有出现错误，则表示软件环境安装正确。接下来下载安装kaldi-gstreamer-server插件，在合适的路径下运行git clone命令

```
git clone https://github.com/alumae/kaldi-gstreamer-server.git
```

下载完成后，不需要编译。kaldi-gstreamer-server/kaldigstserver下存放的是核心程序。kaldi-gstreamer-server的目的是帮助用户实现线上的语音识别服务器，整个server包含两个部分，其中一个是kaldi-gstreamer-server/kaldigstserver/master_server.py，另外一个是kaldi-gstreamer-server/kaldigstserver/worker.py。master_server负责和client端通信，接收client传过来的数据，并且将识别结果传回client端，但master_server本身不负责识别，具体的识别任务由worker负责，当有多个client同时请求时，master_server将每个任务分配给不同的worker。master_server与client之间的通信是全双工通信，即client向master_server发送音频数据的同时，master_server也可向client发送部分识别结果；master_server与worker之间也是全双工通信。全双工通信通过web socket连接实现。

线上的语音识别服务器

kaldi-gstreamer-server提供了client的python实现，即kaldi-gstreamer-server/kaldigstserver/client.py。下面展示如何开启server和client。

首先启动master server，在kaldi-gstreamer-server目录下运行

```
python kaldigstserver/master_server.py --port=8888
```

接下来要开启worker，worker负责语音识别部分，在运行worker之前，要先下载已经训练好的nnet2模型，用作测试。

```
cd test/models

./download-tedlium-nnet2.sh

cd ../../
```

下载的模型为使用TED演讲的音频数据训练的英语识别系统。

```
python kaldigstserver/worker.py -u ws://localhost:8888/worker/ws/speech -c
sample_english_nnet2.yaml
```

此处localhost需要设置成master server所在的ip地址，如果是在本机测试，可以直接使用localhost，端口号要与开启master server时设置的端口号一致。

master server和worker全部正常开启之后，server端已经可以响应client端发送的请求了。下面是使用kaldi-gstreamer-server自带的client程序和英语测试音频来进行测试。

```
python kaldigstserver/client.py -r 8192 test/data/bill_gates-TED.mp3
```

如果一切正常，等待几秒钟之后，屏幕已经开始实时显示部分识别结果了。此处的client端是通过建立websocket连接与master_server通信的，也可以换成使用HTTP的方式：

```
curl  -T *.mp3  "http://localhost:8888/client/dynamic/recognize"
```

kaldi-gstreamer-server较详细的使用方法参见kaldi-gstreamer-server。

几点说明

- master_server.py基于python的tornado实现web socket连接，有关Tornado库，详细资料参见Tornado Web Server Docs。
- worker,py除了使用tornado库，还是用了python的ws4py库来实现web socket连接
- 开启worker时用到的*.yaml文件为nnet2模型的配置文件，worker.py中使用yaml.safe_load()将yaml格式的配置文件解析并存成字典格式，具体语法可查看PYyaml Documentation，所有nnet2模型的相关配置都要在.yaml文件中完成
- .yaml配置文件中包含两个参数，分别是post-processor和full-post-processor，可以实现自定义的程序来对识别结果做后期处理，比如加入标点符号、计算confidence score等
- 返回的识别结果是以json格式存储的，具体的json解析可以参考python 2.7 Documentation

语音识别客户端

语音识别的客户端，除了kaldi-gstreamer-server自带的client.py，完全可以自己实现，比如Android手机客户端、Javascript版Web应用等，以下是对客户端的详细介绍。

Android客户端

前期准备工作

在一个Android程序的众多文件中，有一个重要的清单文件AndroidManifest.xml，该文件包含了关于应用程序的详细信息，它随着程序的新建由IDE自动生成。文件中的package定义了程序应用的包名。由于在线语音识别的Android客户端需要实现录音和联网功能，所以需要在定义package变量那一行的下方加入下列代码，以获取录音、额外存储空间和联网权限。

```
<uses-permission android:name="android.permission.RECORD_AUDIO" />
<uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE" />
<uses-permission android:name="android.permission.INTERNET" />
```

在获取权限之后，需要在程序中写两个相关类 wsConnect.java和ExtAudioRecorder.java，分别实现web socket连接和手机端录音的功能。现有的java_websocket.jar已经集成了相关功能，使我们能更简便的实现Android端即时通讯的功能，下载并将其导入libs，我们仅需要定义消息处理的回调函数即可。下面就实现这两个功能的注意事项进行简要叙述。

Web Socket

Web Socket 是用于在Web Socket应用和服务器之间进行任意的双向数据传输的一种技术。Web Socket协议基于TCP协议实现，包含初始的握手过程，以及后续的多次数据帧双向传输过程。其目的是在Web Socket应用和WebSocket服务器进行频繁双向通信时，可以使服务器避免打开多个HTTP连接进行工作来节约资源，提高了工作效率和资源利用率。有关web socket的介绍可以看这里。

实现音频在线上传的基本思路是，通过录音按钮的监听函数依次实现：建立web socket连接、开始录音 的工作，均需另写方法实现。录音工作在下节会有介绍，我们来说一下web socket连接。在java_websocket.jar包中已经集成了相关函数，举例来说我们要做的就是新建wsConnect.java这个类并继承jar包中的WebSocketClient。定义回调函数onMessage()以处理server返回的识别结果。识别结果为JSON格式，相关样例见Structured Results 。在JSON样例中，首先使用StringBuilder记录识别结果，然后分别用JSONObject和JSONArray获取result和hypotheses，进一步得到transcript和final，这两个分别是单纯的识别结果和识别状态。这样在建立web socket连接的方法中向新建的web socket 实例提供server的URI即可，关于URI的语法请看下节*web socket C/S*协议 。需要注意的是，可以使用Bundle传递识别结果，用Handler更新UI的识别结果。

ExtAudioRecorder

参考开源代码 How to record voice in “wav” format in Android 实现Android端的清晰录音。音频的上传原理是，在建立websocket连接后，服务器端通过prepare()和start()识别并获取Android端的录音设备，实现音频的实时录音。同时ExtAudioRecorder类还支持音频的本地存储和上传，存储格式为.wav文件。通过输入正确的文本信息，可对识别结果进行打分，便于以后的模型调试。需要注意的是，在上述代码的基础上，使用Android的Looper、Handler、Thread三个类对录制音频的系统消息进行了处理。下面对这几个类的使用做简单介绍：

- Message：消息，其中包含了消息ID，消息处理对象以及处理的数据等，由MessageQueue统一列队，终由Handler处理。
- Handler：处理者，负责Message的发送及处理。使用Handler时，需要实现handleMessage(Message msg)方法来对特定的Message进行处理，例如更新UI等。
- MessageQueue：消息队列，用来存放Handler发送过来的消息，并按照FIFO规则执行。当然，存放Message并非实际意义的保存，而是将Message以链表的方式串联起来的，等待Looper的抽取。
- Looper：消息泵，不断地从MessageQueue中抽取Message执行。因此，一个MessageQueue需要一个Looper。
- Thread：线程，负责调度整个消息循环，即消息循环的执行场所。

Android 系统的消息队列和消息循环都是针对具体线程的，一个线程可以存在（当然也可以不存在）一个消息队列和一个消息循环（Looper），特定线程的消息只能分发给本线程，不能进行跨线程，跨进程通讯。但是创建的工作线程默认是没有消息循环和消息队列的，如果想让该线程具有消息队列和消息循环，需要在线程中首先调用Looper.prepare()来创建消息队列，然后调用Looper.loop()进入消息循环。

```
class LooperThread extends Thread {
        public Handler mHandler;

        public void run() {
         Looper.prepare();//给线程创建一个消息循环
         mHandler = new Handler() {
        public void handleMessage(Message msg) {
         // process incoming messages here
         }
        };
         Looper.loop();//使消息循环起作用，从消息队列里取消息，处理消息
  }
}
```

写在Looper.loop()之后的代码不会被立即执行，当调用后 mHandler.getLooper().quit()后，loop才会中止，其后的代码才能得以运行。Looper对象通过MessageQueue 来存放消息和事件。一个线程只能有一个Looper，对应一个MessageQueue。这样你的线程就具有了消息处理机制了，在Handler中进行消息处理。

Web客户端

基于Websocket的Client/Server协议

dictate.js 是一个基于浏览器的实时语音识别JS包，该JS包使用Recorderjs捕获音频，并用WebSocket连接Kaldi GStreamer 服务器，实现online语音识别。web-demo 是一个基于该JS包的非常基础的识别应用。Web端的实现过程可概括为：建立WebSocket连接并打开会话，发送音频，读取结果。可参考以下过程实现：

打开一个会话

在server端开启并配置好worker后，连接指定服务器地址(例如ws://localhost:8888/client/ws/speech)以打开一个会话。在服务器默认情况下，输入的音频格式为：16 kHz，单声道，16-bit little-endian 格式。可以使用'content-type'这个参数修改服务器接收音频的格式，但类型必须符合GStreamer 1.0 规定的格式，例如发送44100 Hz，单声道，16-bit的音频，写法为："audio/x-raw, layout=(string)interleaved, rate=(int)44100, format=(string)S16LE, channels=(int)1"。由于使用url编码，所以实际的请求参考如下格式：

```
ws://localhost:8888/client/ws/speech?content-type=audio/x-raw,+layout=(string)interleaved,+rate=(int)44100,+format=(string)S16LE,+channels=(int)1
```

音频也可使用经GStreamer验证过的任何codec编码（假设服务器已安装了所需的软件包）。GStreamer可以通过container和codec自动检测音频流的格式，所以不必指定content type。例如，在Ogg container中用Speex对音频编码并发送给服务器，可以使用下面的URL打开会话（服务器应能自动识别出codec类型）：

```
ws://localhost:8888/client/ws/speech
```

发送音频

在会话打开时，使用指定的编码将语音按原始数据块(in raw blocks of data)发送到服务器上。建议发送的频率为每秒4次(数据块发送不够频繁会增加识别时间间隔)。数据块不必大小相等。在发送出语音数据的最后一块后，需要向服务器发送一个特殊的3字节的ANSI编码的字符串"EOS"("end-of-stream")。这表示告诉服务器没有更多的语音要被发送过来了，识别可以结束了。发送"EOS"后，客户端需要保持web socket的连接来接收从服务器端传来的识别结果。当所有的识别结果都发送到客户端后，服务器端断开连接。当"EOS"发出后，不再通过同一个web socket发送音频。为了处理一个新的数据流，客户端需要创建一个新的web socket连接。

读取结果

server 返回的response有以下内容：

- status -- 返回状态（类型为integer）
- message --（可选）状态信息
    - 0 -- 成功. 代表识别结果已发送
    - 2 -- 中断. 由于某种原因识别被中断
    - 1 -- 无语音. 代表传来的语音包含一大段静音或无语音
    - 9 -- 不接收. 代表识别器线程正在使用，暂时不能识别
- result -- （可选）识别结果，包含以下字段：
    - hypotheses - 识别出的单词，是一个包含以下信息的列表：
        - transcript -- 识别出的单词
        - confidence -- (可选) hypothesis 的可信度(float, 0..1)
    - final --hypothesis 结束时返回true

当status非零时，断开web socket连接。response 样例参考： Reading result。

在整个识别过程中，服务器会将传入的音频进行分段。对于每个分段，识别结果由许多non-final假设和一个final假设组成。Non-final假设用来向客户端展示部分识别结果，并最终由final假设结尾。在发送出一个分段的final假设后，服务器开始对下一个分段进行解码。或者如果处理完客户端送来的所有的音频后，关闭连接。而客户端只要负责将结果以合适的方式展示用户即可。

总结

整个文挡旨在介绍使用Kaldi+Gstreamer搭建线上的实时语音识别系统的全过程。由于水平有限，免不了会许多错误，希望各位老师同学在查阅的过程中发现错误后及时指正，大家一起努力让这个教程越来越完善。