# **viviEngine Functional Specification**

**VERSION HISTORY**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Version#** | **Implemented****By** | **Revision****Date** | **Approved****By** | **Approval****Date** | **Reason** |
| 1.0 | Shiyue Zhang | 2016.09.10 |  |  |  |
| 1.1 | Dong Wang | 2016.09.10 |  |  |  |

目录

[**viviEngine Functional Specification** 1](#_Toc461272242)

[**1.** **Introduction** 3](#_Toc461272243)

[**2. Overview** 4](#_Toc461272244)

[**2.1 Use Case** 4](#_Toc461272245)

[**2.2 Requirement** 4](#_Toc461272246)

[**2.2.1 Functional Requirement** 4](#_Toc461272247)

[**2.2.2 Efficiency Requirement** 4](#_Toc461272248)

[**2.2.3 Dependence Requirement** 4](#_Toc461272249)

[**3.1 Architecture** 6](#_Toc461272250)

[**3.2 Components** 6](#_Toc461272251)

[**3.2.1 Feature Extraction** 6](#_Toc461272252)

[**3.2.2 CMVN** 7](#_Toc461272253)

[**3.2.3 Decodable** 7](#_Toc461272254)

[**3.2.4 Decoder** 9](#_Toc461272255)

[**3.3** **Encapsulation** 11](#_Toc461272256)

[**3.3.1 Enginelib** 11](#_Toc461272257)

[**3.4** **Usage** 12](#_Toc461272258)

[**3.4.1 本地测试** 12](#_Toc461272259)

[**3.4.2 基于PyServer测试** 13](#_Toc461272260)

[**3.5** **Resource** 16](#_Toc461272261)

[**4. Appendix** 16](#_Toc461272262)

[**5. Reference** 16](#_Toc461272263)

1. **Introduction**

语音识别引擎（ASR engine）通俗地讲就是一个能“听得懂人话”的机器。传统的语音识别系统一般基于HMM算法，而随着深度神经网络的发展，更多的语音识别系统开始选择采用DNN作为解码器的核心。本文档描述FreeNeb viviEngine语音识别引擎的设计原则和基本功能。ViviEngine的框架结构如图1所示，其中包含一个特征提取模块，特征标准化模块，含有两个深度神经网络的decodable模块，以及基于解码图HCLG的Decoder模块。具体的实现利用了开源的语音识别工具箱Kaldi。



**图1. ASR Engine总体框架**

vivEngine实现端到端的语音识别：从原始音频文件输入到文字输出。基本执行流程为：原始wav格式的音频文件输入到ASR引擎中后，先经过一个特征提取模块，该模块可提取音频fBank特征；再经过一个CMVN模块，将特征标准化；而后输入decodable模块，其中包含两个已训练好的深度神经网络DNN-nnet1和TDNN-nnet3，可以选择一个使用，输出溃音素的后验概率，并减去先验概率得到每个音素上的likelihood；最后将likelihood叠加到一个预先编译的HCLG译码网络，搜索其中权重最大的路径，得到最终的解码文本。

本文档目的是对该语音识别引擎进行详细介绍，明确系统结构，以及系统各个模块的功能和相互间的关系。最后给出本地测试方法，以及结合PyServer的测试方法。

**2. Overview**

**2.1 Use Case**

vivEngine 的基本功能是将语音识别成文本，其实现方式是一个标准C/C++动态库。该动态库提供对模型进行初始化，接收数据，输出识别结果三个简单接口。

基于该动态库，语音识别功能可有两种方式实现：命令行模式和viviServer模式。在命令行模式时，用户在命令行中输入如下命令：

**<vivEngine command> <cfg 文件> <wav 文件列表>**

识别结果即可印到屏幕上。这一方式适合开发者进行系统调试和大规模离线测试。在vivServer模式，viviEngine被包装成一个本地服务，从STDIN读入语音数据流，从STDOUT输出识别结果。vivServer需要配合一个wrapper程序，以获取数据和传送输出结果，例如PyServer。PyServer基于Python的命令调用功能如下命令:

**<viviServer command> <cfg 文件>**

即可启动viviServer，准备接收STDIN数据和向STDOUT输出识别结果。PyServer同时利用Python 的多线程能力，接受多用户请求，用加封机制实现请求排队。详见附录中PyServer说明文档。

**2.2 Requirement**

**2.2.1 Functional Requirement**

* 基于Kaldi，实现语音到文本的识别功能
* 支持NNET1和NNET3两种解码方式
* 实现标准库封装与接口
* 实现基于该封装库的两种执行方式：命令行和vivServer.

**2.2.2 Efficiency Requirement**

* RT < 0.3

**2.2.3 Dependence Requirement**

* OS: Centos7
* Kaldi

**3．Implementation details**

**3.1 Architecture**



**图2. ASR Engine程序执行流程**

**3.2 Components**

本部分介绍该引擎的核心代码，包括其重要的类，函数，以及功能详解。代码的目录为freepulsar/decoder/src/engine。

**3.2.1 Feature Extraction**

* **源文件**

stream-feature-input.h, stream-feature-input.cc

* **功能说明**

提取音频的fbank和mfcc特征。

* **功能模块**
* struct StreamFeatureOptions

参数配置，其中feat\_type可以指定使用fank或者mfcc特征。

* class StreamFeatureInput
	+ **函数说明**
		- * **void Process(const VectorBase<BaseFloat> &in, Matrix<BaseFloat> &out)：**输入原始音频向量，处理后得到fbank或者mfcc特征矩阵
			* **void Reset():** 重置remainder\_向量为0向量
* **功能解说**

该模块利用kaldi中的Mfcc和Fbank两个类实现从原始音频抽取两类特征的功能

**3.2.2 CMVN**

* **源文件**

stream-cmvn-input.h, stream-cmvn-input.cc

* **功能说明**

实现了对音频特征矩阵进行CMVN处理，使得特征具有零均值和单位方差的特点

* **功能模块**
* struct StreamCmvnOptions

参数配置

* class StreamCmvnInput
	+ **函数说明**
		- * **int Process(Matrix<BaseFloat> &feats):** 对输入的特征矩阵进行CMVN处理
			* **int Load(const Matrix<double> &in):** 加载特征矩阵
			* **int Save(Matrix<double> &out):** 保存特征矩阵（暂未实现）
			* **void Reset():** 特征矩阵维度变换
* **功能解说**

该模块利用kaldi中transform组件下的cmvn处理模块，实现了对原始音频特征的标准化处理

**3.2.3 Decodable**

**1. StreamDecodableInterface**

* **源文件**

stream-decoabable.h。

* **功能说明**

定义不同decodable模块共同的虚基类，便于后续实现。

* **功能模块**
	+ **class StreamDecodableInterface: public DecodableInterface**

继承并封装kaldi中的DecodableInterface

**2. DNN-nnet1**

* **源文件**

steam-nnet-decodable.h, steam-nnet-decodable.cc

* **功能说明**

基于一个训练好的kaldi.nnet1模型，估计输入音频数据的likelihood

* **功能模块**
	+ **struct StreamNnetDecodableOptions**

参数配置

* **class StreamNnetDecodable: public StreamDecodableInterface**
	+ **函数说明**
		- **StreamNnetDecodable(StreamNnetDecodableOptions config)：**类构造函数，在构造过程中初始化特征提取对象，cmvn处理对象，并据配置参数导入已训练好的DNN模型和先验假设等。
		- i**nt32 SetData(const char \*data, uint32\_t len, int chunk\_no):** 主函数，完成特征提取模块调用、CMVN处理模块调用、Nnet处理模块调用。
		- **int32 Process(const Matrix<BaseFloat> &in, Matrix<BaseFloat> &out, int chunk\_no)：**Nnet处理函数，将特征数据输入训练好的nnet1深度神经网络模型，前向算法得到后验概率，取log后减去先验概率的log，得到最终的likelihood。
		- **功能解说**

该模块利用一个以训练好的nnet1模型，完成了likelihood的估计，为下一步的解码提供基础

**2. TDNN-nnet3**

* **源文件**

stream-nnet3-decodable.h, stream-nnet3-decodable.cc。

* **功能说明**

基于一个训练好的kaldi.nnet3模型，估计输入音频数据的likelihood。

* **功能模块**
	+ **struct StreamNnet3DecodableOptions**

参数配置

* class StreamNnet3Decodable: public StreamDecodableInterface
	+ **函数说明**
		- **StreamNnet3Decodable(StreamNnet3DecodableOptions config)：**类构造函数，在构造过程中初始化特征提取对象，cmvn处理对象，并据配置参数导入已训练好的TDNN模型等。
		- **int32 SetData(const char \*data, uint32\_t len, int chunk\_no):** 主函数，完成特征提取模块调用、CMVN处理模块调用、Nnet处理模块调用。
		- **int32 Process(const Matrix<BaseFloat> &in, Matrix<BaseFloat> &out, int chunk\_no)：**Nnet处理函数，将特征数据输入训练好的nnet3深度神经网络模型，可直接通过模型输出likelihood。
		- **功能解说**

该模块利用一个以训练好的的nnet3模型，完成了likelihood的估计，为下一步的解码提供基础

**3.2.4 Decoder**

* **源文件**

stream-hclg-decoder.h, stream-hclg-decoder.cc, stream-faster-decoder.h, stream-faster-decoder.cc, stream-biglm-faster-decoder.h, stream-biglm-faster-decoder.cc

* **功能说明**

实现了从HCLG图中译码的功能，输出音频对应的文字

* **功能模块**
* struct StreamHCLGDecoderOptions

参数配置

* class StreamHCLGDecoder
	+ **函数说明**
		- * **StreamHCLGDecoder(const StreamHCLGDecoderOptions &config, StreamDecodableInterface &decodable, fst::Fst<fst::StdArc> \*decode\_fst)：**类构造函数，在构造过程中初始化decadable对象，并根据配置参数导入字形文件，解码的FST网络，选择要使用的decoder：StreamBiglmFasterDecoder或StreamFasterDecode等。
			* **fst::Fst<fst::StdArc> \*ReadNetwork(std::string filename)：**从文件读入解码网络。
			* **int Decode(const char \*data, uint32 len, int chunk\_no):** 调用decodable的主函数，并把输出的likelihood与已知的解码网络相结合。
			* **string GetResult():** 在合并后的解码网络上搜索到权重最大的路径，即为解码结果。
* **功能解说**

该模块实现了完整的译码过程。

* class StreamFasterDecoder: public StreamDecoder , private FasterDecoder
	+ **函数说明**
		- * **void DecodeFrame(DecodableInterface \*decodable, int32 frame)：**将decodable输出的likelihood与已知的解码网络相结合。
			* **bool GetBestPath(fst::MutableFst<LatticeArc> \*fst\_out)：**在FST网络上搜索最佳的路径。
		- **功能解说**

该模块是一个decoder模型。

* class StreamBiglmFasterDecoder: public StreamDecoder, private BiglmFasterDecoderMod
	+ **函数说明**
		- * **void DecodeFrame(DecodableInterface \*decodable, int32 frame)：**将decodable输出的likelihood与已知的解码网络相结合。
			* **bool GetBestPath(fst::MutableFst<LatticeArc> \*fst\_out)：**在FST网络上搜索最佳的路径。
		- **功能解说**

该模块是另一个decoder模型。

* 1. **Encapsulation**

**3.3.1 Enginelib**

为了便于使用，以及保护代码，将核心代码封装为一个整合的decoder，在freepulsar/decoder/src/enginelib目录下。注意区分这个decoder，和我们之前提及的decoder模块，这里的decoder是封装了所有功能模块的集合，而之前的decoder只是其中的一部分。我们所称viviEngine, 从代码角度讲是包括上述所有component的代码集合;但从产品角度讲，viviEngine即指本节所述的Enginelib提供的lib接口方式。

* **源文件**

decoder.cc

* **功能说明**

对核心代码的封装

* **功能模块**
	+ struct decoder\_obj

封装得到的decoder结构体，其中包含一个decodable模块和一个decoder模块。

* + int decoder\_init(struct decoder\_obj \*\*ptr, const char \*filename, int nnet\_version)

根据配置参数初始化decoder结构体，主要是创建decodable和decoder对象。

* + int decoder\_put\_data(struct decoder\_obj \*ptr, const char \*data, int len, int chunk\_no)

输入音频数据，调用了decoder模块中的Decode函数，得到数据的likelihood，并更新解码图。

* + int decoder\_get\_result(struct decoder\_obj \*ptr, char \*result, int len)

输出文本，调用了decoder中的GetResult函数，从解码图中找到最佳路径作为解码输出。

* + int decoder\_release(struct decoder\_obj \*ptr)

结构体的析构函数。

* 1. **Usage**

**3.4.1 本地测试**

本地进行测试的目录在freepulsar/decoder/test/vivi-test下，其中含有nnet1和nnet3两种测试环境，两者相似，唯一不同是使用的深度学习网络不同。因此，这里仅拿nnet1测试环境作为例子，简单介绍测试的过程。测试过程可分为三步：

* **设置参数**

nnet1目录下的decoder.conf文件中配置了引擎需要传入的参数，包括已训练好的神经网络的地址，HCLG解码网络的地址等，如下图所示。在进行测试之前要把这些参数先设置好。



**图3.decoder.conf配置示例**

* **设置音频文件列表**

nnet1目录下的wav.scp文件中配置了要进行识别的音频列表，其中每一行是<id><wavfile>的形式，如下图所示。在进行测试之前，要先设置好需要进行测试的音频。



**图4. wav.scp示例**

* **运行**

nnet1目录下的run.sh是测试代码的接口，其中调用了enginebin下的vivi-decoder-nnet1-example可执行文件，因此在运行代码之前需要编译好enginebin下的代码。在上述都配置好之后，用下述命令即可得到测试结果：

***./run.sh decoder.conf wav.scp***

输出结果如下图所示：



**图5. 本地测试输出结果示例**

这是一个音频文件的识别输出，真实的文字是：津中两国友好关系有着悠久的历史。可以看出基本可以准确识别。

**3.4.2 基于PyServer测试**

PyServer是基于Python构造一个简单的语音识别引擎外围服务程序，同样可以基于PyServer进行引擎的测试。注意PyServer所调用的viviServer在src/engninesrv/bin下。viviServer当前实现中并没有调用viviEngine的library 接口，而是直接调用viviEngine中的Decoder模块。这一方式有可能在未来会发生改变。

基于PyServer测试包括如下两个步骤：

* **启动Server**

Server端启动前，应先配置后server.cfg文件，主要有模型目录、主机ip及端口号等。一个典型的server config如下图。



**图6. server config配置示例**

Server config配置完成后，运行下面命令启动PyServer：

***server.py <server.cfg>***

若 <server.cfg>不提供，则默认config文件为当前目录下的server.cfg。

* **启动Client提交识别请求**

Client端启动前，需要配置文件client.cfg，主要是设定PyServer所在的主机和服务端口。一个典型的client端配置文件如下图所示：



**图7. client cfg配置示例**

配置完成后，用下述命令可执行识别命令：

***client.py wav.lst***

其中wav.lst是待识别的语音文件列表。一个典型的语音文件列表的每一行为如下形式：<id> <wavfile>，如下图所示：



**图8. wav list示例**

Client端输出结果如下图所示：



**图9. client识别结果示例**

* 1. **Resource**

Download the resource here: gitlab:freepulsar

**4. Appendix**

**5. Reference**

1. Kaldi document：http://www.kaldi-asr.org/doc/

2. PyServer Functional Specification: http://cslt.riit.tsinghua.edu.cn/mediawiki/images/f/fb/PyServer.functional.pdf