## 离线端到端语音识别数据结构说明

## 1神经网络计算功能开发

语音信号的字符概率计算是将语音特征数据进行端到端语音识别模型的前向深度神经网络计算，得到语音特征对应的字符后验概率。

神经网络计算模块是计算语音声学得分的重要模块，在提取语音特征后，将特征数据送入已训练好的端到端语音识别模型中，计算出字符的后验概率，作为语音的声学得分。

### 1.1神经网络组件开发

（1）**BaseComponent**类

BaseComponent是神经网络基类，包含了各种神经网络中通用的成员变量和成员函数。关键的成员变量中包括：

* m\_id：神经网络名称；
* m\_pplane:：指定该层神经网络的输入来自哪些层的神经网络；
* m\_pfmap：该层神经网络的输入矩阵指针，可以存在多个输入矩阵；
* m\_fmap：该层神经网络的输出矩阵。

关键的成员函数包括：

* FLOAT \* setparam(FLOAT \* param):

功能：读取并设置该层神经网络的模型参数。

形参：

param：读取二进制端到端语音识别模型的指针。

* Matrix<FLOAT> \* fprop():

功能：定义该层神经网络的计算方式，由于不同神经网络的输入类型和个数不同，需要重载该函数。

* Matrix<FLOAT> \* getfmap ():

功能：获取该层神经网络的输出矩阵指针。

* int setfmap (FLOAT \* source, int nrow, int ncol):

功能：读取并设置该层神经网络的输出矩阵。

形参：

source：数组指针，一维数组表示二维矩阵，行优先存储；

nrow：二维矩阵行数；

ncol：二维矩阵列数。

（2）**InputComponent**类

InputComponent类是BaseComponent类的子类。InputComponent是输入层，作为整个模型的第一层。该层无模型参数，无前向计算，只需要调用函数setfmap，将输入特征直接写入该层的输出矩阵。

（3）**ReluComponent**类

ReluComponent类是BaseComponent类的子类。ReluComponent是ReLU激活函数网络层，该层无模型参数。输入为x，输出为

（4）**TanhComponent**类

TanhComponent类是BaseComponent类的子类。TanhComponent是Tanh激活函数网络层，该层无模型参数。输入为x，输出为

（5）**MultiHeadAttComponent**类

MultiHeadAttComponent类是BaseComponent类的子类。MultiHeadAttComponent是标准Transformer中的多头注意力层，关键的成员变量包括：

* m\_weight\_q: query的变换矩阵W\_q；
* m\_weight\_k: key的变换矩阵W\_k；
* m\_weight\_v: value的变换矩阵W\_v；
* m\_weight\_o: output的变换矩阵W\_o；
* m\_bias\_q: query的偏置向量b\_q；
* m\_bias\_k: key的偏置向量b\_k；
* m\_bias\_v: value的偏置向量b\_v；
* m\_bias\_o: output的偏置向量b\_o。

计算过程参考文章《Attention Is All You Need》(https://arxiv.org/abs/1706.03762)。其构造函数的若干重要参数包括：

* qdim: query输入维度；
* kdim: key/value输入维度；
* nhead: 多头个数；
* isbias: 是否存在偏置向量。

MultiHeadAttComponent重载了前向计算函数fprop的接口，除了从上一层获得query之外，还需要另外输入key/value（key和value是相同的）。在线系统中编码器和解码器的自注意力层均使用MultiHeadAttComponent，编码器和解码器中自注意力层的key均是由上一层(LayerNormComponent)输出和历史块中的输出拼接得到，编码器和解码器的成员函数propagate定义如何选取历史块中的输出。

（6）**ConvComponent**类

ConvComponent类是BaseComponent类的子类。ConvComponent是二维卷积神经网络层，关键的成员变量包括：

* m\_weight: 多个卷积核拼接成的矩阵W；
* m\_bias: output的偏置向量b。

其中矩阵W的第一个维度等于输出通道数，第二个维度是输入通道数乘卷积核高度（时域）乘卷积核宽度（频域），向量b的长度等于输出通道数，计算过程参考pytorch教程（https://pytorch.org/docs/stable/nn.html?highlight=conv2d#torch.nn.Conv2d）。

ConvComponent构造函数的若干重要参数包括：

* in: 输入通道数；
* out: 输出通道数；
* kernel: 卷积核高度和宽度；
* stride: 卷积核高度和宽度方向移动的步长；
* pad: 对输入的二维矩阵首尾两端填充零的行数和列数；
* dilation: 空洞卷积核高度和宽度方向的间隙；
* isbias: 是否存在偏置向量。

ConvComponent前向计算采用im2col优化卷积运算，对于多通道的特征图，系统一律采用单个二维矩阵存储，矩阵的行代表特征图的时域（语音特征序列长度），矩阵的列代表特征图的频域（语音特征维度）乘通道数。

（7）**LinearComponent**类

LinearComponent类是BaseComponent类的子类。LinearComponent是全连接网络层，关键的成员变量包括：

* m\_weight: 参数矩阵W；
* m\_bias: output的偏置向量b。

输入为x，输出为

LinearComponent构造函数的若干重要参数包括：

* in: 输入通道数；
* out: 输出通道数；
* isbias: 是否存在偏置向量。

（8）**LogSoftmaxLinearComponent**类

LogSoftmaxLinearComponent类是BaseComponent类的子类。LogSoftmaxLinearComponent是全连接网络层和LogSoftmax网络层的组合，关键的成员变量包括：

* m\_weight: 参数矩阵W；
* m\_bias: output的偏置向量b。

输入为x，输出为

LogSoftmaxLinearComponent构造函数的若干重要参数包括：

* in: 输入通道数；
* out: 输出通道数；
* isbias: 是否存在偏置向量。

（9）**SoftmaxLinearComponent**类

SoftmaxLinearComponent类是BaseComponent类的子类。LogSoftmaxLinearComponent是全连接网络层和Softmax网络层的组合，关键的成员变量包括：

* m\_weight: 参数矩阵W；
* m\_bias: output的偏置向量b。

输入为x，输出为

SoftmaxLinearComponent构造函数的若干重要参数包括：

* in: 输入通道数；
* out: 输出通道数；
* isbias: 是否存在偏置向量。

（10）**PosFFComponent**类

PosFFComponent类是BaseComponent类的子类。PosFFComponent是标准Transformer中的前馈网络层，关键的成员变量包括：

* m\_weight\_1: 第一个参数矩阵W\_1；
* m\_weight\_2: 第二个参数矩阵W\_2；
* m\_bias\_1: 第一个偏置向量b\_1；
* m\_bias\_2: 第二个偏置向量b\_2。

输入为x，输出为

PosFFComponent构造函数的若干重要参数包括：

* in: W\_1的输入维度，W\_2的输出维度；
* inner: W\_2的输入维度，W\_1的输出维度。

（11）**EmbedComponent**类

EmbedComponent类是BaseComponent类的子类。EmbedComponent是词嵌入网络层，作为标准Transformer中解码器的第一层，负责将字符标签通过查表的方式转换为特征向量，关键的成员变量包括：

* m\_weight: 词嵌矩阵W。

EmbedComponent构造函数的若干重要参数包括：

* ndim: 特征向量维度；
* vnum: 词汇表大小。

（12）**KQVComponent**类

KQVComponent类是BaseComponent类的子类。KQVComponent是标准Transformer中自注意力网络层的部分计算，关键的成员变量包括：

* m\_weight\_q: query的变换矩阵W\_q；
* m\_weight\_k: key的变换矩阵W\_k；
* m\_weight\_v: value的变换矩阵W\_v；
* m\_bias\_q: query的偏置向量b\_q；
* m\_bias\_k: key的偏置向量b\_k；
* m\_bias\_v: value的偏置向量b\_v。

输入为x，输出为

KQVComponent构造函数的若干重要参数包括：

* qdim: query输入维度；
* kdim: key/value输入维度；
* isbias: 是否存在偏置向量。

（13）**LayerNormComponent**类

LayerNormComponent类是BaseComponent类的子类。LayerNormComponent是层归一化网络层，关键的成员变量包括：

* m\_weight: 权重向量；
* m\_bias: output的偏置向量；
* m\_eps: 精度值。

输入为，输出为

LayerNormComponent构造函数的若干重要参数包括：

* ndim: 输入/输出维度；
* eps: 输出通道数；
* isbias: 精度值，pytorch默认值1e-12。

（14）**ResidualComponent**类

ResidualComponent类是BaseComponent类的子类。ResidualComponent是残差连接网络层，该层无模型参数，输入为x和y，输出为

（15）**PosEncComponent**类

PosEncComponent类是BaseComponent类的子类。PosEncComponent是标准Transformer中的位置编码层，该层的模型参数是由正弦余弦函数计算得到的位置编码矩阵W={w\_ij}，该层无模型参数，输入为x={x\_ij}，输出为

PosEncComponent构造函数的若干重要参数包括：

* ndim: 输入/输出维度；
* maxlen: 最大长度，训练平台默认值5000，因此编码器降采样后特征序列的最大长度，以及解码器预测序列的最大长度均限制为maxlen。

（16）**LSTMCellComponent**类

LSTMCellComponent类是BaseComponent类的子类。LSTMCellComponent是长短时记忆网络层，关键的成员变量包括：

* m\_weight\_ih: 权重矩阵W\_i=[W\_ii, W\_if, W\_ig, W\_io]；
* m\_weight\_hh: 权重矩阵W\_h=[W\_hi, W\_hf, W\_hg, W\_ho]；
* m\_bias\_ih: 偏置向量b\_i=[b\_ii, b\_if, b\_ig, b\_io]；
* m\_bias\_hh: 偏置向量b\_h=[b\_hi, b\_hf, b\_hg, b\_ho]。

其中下标i, f, g, o代表输入门(input)、遗忘门(forget)、记忆单元(memory cell)和输出门(output)，计算过程参考pytorch教程（https://pytorch.org/docs/stable/nn.html?highlight=lstm#torch.nn.LSTM）。

LSTMCellComponent构造函数的若干重要参数包括：

* idim: 输入维度；
* ndim: 输出维度；
* nbias: 偏置向量个数，pytorch中LSTM的偏置向量个数是2，即b\_i和b\_h。

### 1.2神经网络模型开发

（1）**Encoder**类

Encoder类是组合BaseComponent类的上层类，提供对BaseComponent类的总体管控，控制进行网络参数的载入，神经网络计算的相关配置，以及调用不同的BaseComponent子类进行前向计算。Encoder是不同编码器神经网络的基类，关键的成员变量包括：

* m\_fmap: 存储Encoder输出的特征矩阵；
* m\_plane: 管理Encoder中所有BaseComponent类的指针。

关键的成员函数包括：

* Matrix<FLOAT> \* getfmap (): 获取存储Encoder输出的特征矩阵的指针；
* FLOAT \* setparam(FLOAT\* param): 从首地址param开始读取Encoder模型参数，继承的子类必须实现自己的读取模型参数顺序；
* void propagate(Encoder \* pGlobalEncoder, FLOAT \* feats, int len): 定义Encoder内部的神经网络前向计算，继承的子类必须实现自己的神经网络前向计算。

（2）**SelfAttEncoder**类

SelfAttEncoder类继承Encoder类，实现Speech Transformer中Encoder的前向计算。具体包括2层ConvComponent、1层PosEncComponent和N层Transformer block。计算过程参考文章《Improving Transformer-Based End-to-End Speech Recognition with Connectionist Temporal Classification and Language Model Integration》（https://www.isca-speech.org/archive\_v0/Interspeech\_2019/pdfs/1938.pdf）。关键的成员变量包括：

* m\_layer: Transformer block层数；
* m\_idim: 输入特征维度;
* m\_odim: 输出特征维度;
* m\_channel: 输出特征通道数，默认为1;
* m\_maxlen: 输出特征序列长度的最大值;
* m\_nhead: self-attention中的头个数;
* m\_inner: feed-forward中隐藏层维度;
* m\_filter1: 一次层CNN中卷积个数，默认为0时，卷积个数等于m\_odim;
* m\_filter2:第二层CNN中卷积个数，默认为0时，卷积个数等于m\_odim。

（3）**Decoder**类

Decoder调用BaseComponent类的上层类，提供对BaseComponent类的总体管控，控制进行网络参数的载入，神经网络计算的相关配置，以及调用不同的BaseComponent子类进行前向计算。Decoder是不同解码器神经网络的基类，关键的成员变量包括：

* m\_fmap: 指向Decoder输出的特征矩阵的指针；
* m\_fmap\_enc: 指向Encoder输出的特征矩阵的指针；
* m\_plane: 管理Decoder中所有BaseComponent类的指针；
* m\_odim: 解码器输出维度，即端到端模型建模单元个数；
* m\_endLabel: <eos>的id;
* m\_useBuff: 是否使用Buffer中存储的计算结果，减少冗余计算。

关键的成员函数包括：

* Matrix<FLOAT> \* getfmap (): 获取存储Decoder输出的特征矩阵的指针；
* FLOAT \* setparam(FLOAT\* param): 从首地址param开始读取Decoder模型参数，继承的子类必须实现自己的读取模型参数顺序；
* void propagate(Decoder \* pGlobalDecoder, Matrix<FLOAT> \* states, FLOAT \* newstate, int label, int step): 定义Decoder内部的神经网络前向计算，继承的子类必须实现自己的神经网络前向计算。
* FLOAT getprob(int label): 在调用一次前向计算后返回label对应的概率；
* int getStateSize ():返回解码器单个字符的前向计算时需要的生成的新状态大小；
* int getVocabularySize ():返回解码器建模单元的个数。

（4）**SelfAttDecoder**类

SelfAttEncoder类继承Decoder类，实现Speech Transformer中Decoder单个字符的前向计算。具体包括一层EmbedComponent、一层PosEncComponent、N层Transformer block和一层LogSoftmaxLinearComponent。计算过程参考文章《Improving Transformer-Based End-to-End Speech Recognition with Connectionist Temporal Classification and Language Model Integration》（https://www.isca-speech.org/archive\_v0/Interspeech\_2019/pdfs/1938.pdf）。关键的成员变量包括：

* m\_layer: Transformer block层数；
* m\_idim: 输入词向量维度;
* m\_kdim: Encoder的输出特征维度;
* m\_maxlen: 输出特征序列长度的最大值;
* m\_nhead: self-attention中的头个数;
* m\_inner: feed-forward中隐藏层维度。

（5）**CTC**类

CTC类负责解码过程中所有与CTC相关的功能，其中前缀分数的计算过程参考文章《Hybrid CTC/attention architecturefor end-to-end speech recognition》（https://ieeexplore.ieee.org/abstract/document/8068205）。成员变量包括：

* m\_idim: 输入特征的维度，一般等于Encoder输出特征的维度；
* m\_odim: 输出特征的维度，即CTC建模单元的个数；
* m\_blank: <blank>符号的id;
* m\_eos: <eos>符号的id;
* m\_length: 输入特征序列的长度；
* m\_isbias: 是否有偏置向量；
* m\_weight: 权重矩阵；
* m\_bias: 偏置向量；
* m\_posteriors：输入语音信号特征对应的字符后验概率分布。

重要的成员函数包括：

* FLOAT \* setparam(FLOAT\* param): 从首地址param开始读取CTC模型参数；
* void init (CTC \* pGlobalCTC, Matrix<FLOAT> \* input, FLOAT \* state): 由输入的特征矩阵input计算m\_posterior，并且初始化CTC前缀分数使用的内部状态state;
* void computeCTCPrefixScore(FLOAT \* state, Node \* node): 根据上一步CTC前缀分数的内部状态state和当前的节点node，计算在旧前缀后面扩展node.label时对应的新前缀分数;
* bool forceAlign(Node \* node, int \* align, FLOAT \* probs): 从节点node回溯识别的字符串，使用m\_posterior和维特比算法获得概率最大的对齐方式并存储在align中，每一帧对齐结果对应的概率存储在probs中。

（6）**E2EModel**类

E2Emodel类实现端到端模型的基类，其成员变量包括：

* m\_encoder: 管理Encoder类的指针；
* m\_decoder: 管理Decoder类的指针；
* m\_ctc: 管理CTC类的指针;
* m\_maxlen: Encoder和Decoder输出序列的最大长度。

重要的成员函数包括：

* FLOAT \* setparam(FLOAT \* param): 从首地址param开始读取端到端模型参数；
* void doEncoder(Encoder \* pGlobalEncoder, FLOAT \* feats, int len): 调用Encoder的propagate函数；
* void doDecoder(Decoder \* pGlobalDecoder, Matrix<FLOAT> \* states, FLOAT \* newstate, int label, int step): 调用Decoder的propagate函数;
* void resetDecoderBuffer (): 清空Decoder的Buffer；
* void initCTC(CTC \* pGlobalCTC, FLOAT \* state): 调用CTC的initCTC函数；
* void doCTCPrefixScore(FLOAT \* state, Node \* node): 调用CTC的computeCTCPrefixScore函数
* bool CTCForceAlign (Node \* node, int \* align, FLOAT \* probs): 调用CTC的forceAlign函数
* FLOAT getDecoderProb(int label): 返回当前解码器中label对应的概率。

其他的成员函数可以从字面直接理解其功能，这里不再赘述。

（7）**Transformer**类

Transformer类继承E2EModel类，它的Encoder和Decoder分别使用SelfAttEncoder和SelfAttDecoder。其构造函数接口中涉及的参数均与SelfAttEncoder和SelfAttDecoder有关。

（8）**LM**类

LM类是调用BaseComponent类的上层类，提供对BaseComponent类的总体管控，控制进行网络参数的载入，神经网络计算的相关配置，以及调用不同的BaseComponent子类进行前向计算。LM是不同语言模型的基类，关键的成员变量包括：

* m\_fmap: 存储LM输出的特征矩阵；
* m\_plane: 管理LM中所有BaseComponent类的指针；

关键的成员函数包括：

* Matrix<FLOAT> \* getfmap (): 获取存储LM输出的特征矩阵的指针；
* FLOAT \* setparam(FLOAT\* param): 从首地址param开始读取LM模型参数，继承的子类必须实现自己的读取模型参数顺序；
* void propagate(LM \* pGlobalLM, FLOAT \* prevstate, FLOAT \* newstate, int label): 定义LM内部的神经网络前向计算，继承的子类必须实现自己的神经网络前向计算。
* FLOAT getprob(int label): 在调用一次前向计算后返回label对应的概率；
* int getStateSize ():返回语言模型单个字符的前向计算时需要的生成的新状态大小；
* int getVocabularySize ():返回语言模型建模单元的个数。

（8）**LSTMLM**类

LSTMLM类继承LM类，实现基于LSTM的神经网络语言模型单个字符的前向计算。具体包括1层EmbedComponent、N层LSTMCellComponent和一层LogSoftmaxLinearComponent。关键的成员变量包括：

* m\_layer: LSTMCellComponent层数；
* m\_idim: 输入词向量维度;
* m\_odim: LM的输出特征维度，即建模单元个数;
* m\_ndim: LSTM内层状态的维度;
* m\_nbias: LSTM中使用的偏置向量个数，pytorch默认为2。

## 2 解码搜索功能开发

语音识别解码的目标是基于输入的语音信号，输出概率最大的文字串。在端到端语音识别引擎中，通过用户提供的端到端语音识别模型和语言模型，引擎内部将构建搜索空间并快速搜索出概率最大的文字串。

### 2.1 WFST构建功能开发

在本套端到端语音识别引擎中使用WFST时，默认使用的是基于CTC的端到端语音识别模型。在构建WFST时，需要用户提供端到端语音识别模型的建模单元列表，Arpa格式的语言模型。引擎提供脚本生成WFST，其中会依赖openFST。下面详细解释生成WFST的步骤。

（1）获取WFST的输入/输出列表

* get\_words\_from\_arpa.py <lm.arpa> <words.txt> <lexicon.txt> [<non-language-symbols.txt>]

该脚本从<lm.arpa>中读取1-gram，生成输出列表<words.txt>。<words.txt>形式如下：

<eps> 0

<unk> 1

…

#0 N

其中编号为0的<eps>和编号N的#0是openFST中默认的。编号1～N-1是用户自定义使用的词汇表。因为输出列表中删掉了<s>和</s>，增加了<eps>和#0，并且从0开始编号，所以N正好比1-gram的个数少一个。<eps>用于表示空字符，#0用于语言模型回退计算时的消歧。

同时生成拼写词典<lexicon.txt>。在中文端到端语音识别引擎中，拼写词典中的中文词语被拆成汉字串，英文单词被拆成字母串，中英混合的词语被拆成汉字和字母组合的字符串。在使用子词（e.g. BPE）的英文系统中，需要用户额外提供拼写词典，根据既定规则以唯一的方式将词语拆分成子词串。这里仅以中文端到端语音识别引擎为例，使用的<lexicon.txt>形式如下：

<unk> <unk>

wifi w i f i

无线 无 线

无线wifi 无 线 w i f i

…

其中<lexicon.txt>中第一列包含了<words.txt>中编号1～N-1词语，从第二列开始是按照拼写规则拆分词语的到的字符串（用空格间隔开）。表示非语言的词语，例如样例中的<unk>，不应该被拆分成字符串（< u n k >），需要在<non-language-symbols.txt>中申明。<non-language-symbols.txt>的格式如下：

<unk>

<noise>

…

如果不提供<non-language-symbols.txt>，则编号1～N-1词语全部被拆成字符串。

脚本还会生成列表units.txt，它是<lexicon.txt>从第二列开始出现的字符集合。将units.txt的编号删掉生成units.list。

* fix\_units.py <vocab.txt> <units.list> <units.fix.list>

由于units.list中的字符列表和提供的端到端语音识别模型的建模单元列表<vocab.txt>不同，脚本将取两个列表的交集得到<units.fix.list>，其中<vocab.txt>的字符在<units.fix.list>中排在前，且原始顺序不变。在<units.fix.list>的基础上在首位增加<eps>，在末尾增加消歧符号#0、#1 …，生成输入序列tokens.txt。tokens.txt形式如下：

<eps> 0

<blank> 1

<unk> 2

…

<eos> V

…

#0 N+1

#1 N+2

#2 N+3

…

其中编号为0的<eps>和从编号N+1开始的#0、#1、#2是openFST中默认的，编号为1的<blank>和编号为V的<eos>是基于CTC的端到端语音识别模型中默认的，编号1～V是端到端语音识别模型的建模单元。端到端模型中的建模单元实际应该从0开始编号，但需要把首位留给<eps>，所以WFST中输入字符的编号比端到端模型中对应字符的编号相差1。编号V+1～N的字符未被端到端模型采用，但被语言模型采用，依然参与构建WFST。但是在解码搜索过程中，端到端模型无法给出其分数，搜索路径不经过这些字符。

（2）构建CTC塌缩规则的WFST

* ctc\_token\_fst.py <tokens.txt> | fstcompile --isymbols=<tokens.txt> --osymbols=<tokens.txt> --keep\_isymbols=false --keep\_osymbols=false | fstarcsort --sort\_type=olabel > <T.fst>

首先通过ctc\_token\_fst.py生成FST的txt格式，再通过fstcompile编译为二进制格式，其中<T.fst>将基于CTC的端到端模型输出的序列塌缩为字符串，塌缩规则是先将连续相同的字符串塌缩为单个字符，再删除所有的<blank>字符，得到目标字符串。

（3）构建拼写规则的WFST

* perl -ape 's/(\S+\s+)(.+)/${1}1.0\t$2/;' < ${lexicon.txt} > ${lexiconp.txt}

首先在${lexicon.txt}的每个词条中加入拼写概率，默认为1.0。

* add\_lex\_disambig.pl ${lexiconp.txt} ${lexiconp\_disambig.txt}

然后在${lexiconp.txt}的某些词条中加入消歧符号。消歧符号从#1开始编号，#0留给语言模型使用。

* make\_lexicon\_fst.pl --pron-probs <lexiconp\_disambig.txt> | fstcompile --isymbols=<tokens.txt> --osymbols=<words.txt> --keep\_isymbols=false --keep\_osymbols=false | fstaddselfloops <in-disambig-list> <out-disambig-list> | fstarcsort --sort\_type=olabel > <L.fst>

最后由make\_lexicon\_fst.pl生成FST的txt格式，通过fstcompile编译为二进制格式，其中<L.fst>将塌缩后的字符串转换成可能的词语串。fstaddselfloops在<L.fst>中加入#0的自环，以供后续复合FST。

（4）构建语言模型的WFST

* arpa2fst <lm.arpa> | fstprint | eps2disambig.pl | s2eps.pl | fstcompile --isymbols=<words.txt> --osymbols=<words.txt> --keep\_isymbols=false --keep\_osymbols=false | fstrmepsilon | fstarcsort --sort\_type=ilabel > <G.fst>

首先通过arpa2fst生成FST的txt格式，其中eps边被改成#0，<s>和</s>被改成<eps>。然后通过fstcompile编译为二进制格式，其中<G.fst>给出输入词语串的语言模型概率。

（5）生成最终的WFST

* fsttablecompose <L.fst> <G.fst> | fstdeterminizestar --use-log=true | fstminimizeencoded | fstpushspecial | fstarcsort --sort\_type=ilabel > <LG.fst>
* fsttablecompose <T.fst> <LG.fst> | fstminimizeencoded | fstarcsort --sort\_type=ilabel > <TLG.fst>

首先使用fsttablecompose、fstdeterminizestar和fstminimizeencoded实现

生成的<TLG.fst>将CTC输出的序列转换为词语串，并赋予语言学分数。

* fstprint <TLG.fst> <TLG.int>
* WFST2Bin -txt <TLG.int> -in <tokens.txt> -out <words.txt> -bin <TLG.bin>

然后使用WFST2Bin将openFST格式的<TLG.fst>转为端到端语音识别引擎支持的二进制文件< TLG.bin>。

### 2.2 非流式语音帧同步解码功能开发

在本套端到端语音识别引擎中使用语音帧同步解码时，默认使用的是基于CTC的端到端语音识别模型（SelfAttEncoder）和基于N-gram的统计语言模型（WFSTNetwork）。语音帧同步解码的搜索空间由2.1中生成的TLG.bin决定，其中语言学分数来自TLG.bin中的转移边上的分数，声学分数来自端到端模型计算出的每一帧语音特征对应的字符后验概率。

（1） 搜索空间WFST相关内容在wfst/WFSTNetwork.h中，关键的结构体和类包括：

* **WFSTTransition**: 储存TLG.bin中的转移边，成员变量包括：
* id: 按照TLG.bin中转移边排序的id；
* toState: 转移边转向的状态id；
* weight: 转移边权重；
* inLabel: 转移边输入字符id，0表示<eps>；
* outLabel: 转移边输出字符id，0表示<eps>。
* **WFSTState**: 存储TLG.bin中的状态，成员变量包括：
  + start\_trans: 状态首个出边id（同一个状态的出边id连续）；
  + nTrans: 状态出边个数。
* **WFSTFinalState**: 存储TLG.bin中的终止状态，成员变量包括：
  + id: 状态id；
  + weight: 终止状态的分数。
* **WFSTNetwork**: 读取TLG.bin，管理输入字符、输出字符、状态和转移边。重要的成员变量包括：
  + inputAlphabet: 管理输入字符集合；
  + outputAlphabet: 管理输出字符集合；
  + initState: 起始状态id，默认为0。
  + nStates: 状态个数；
  + states: 管理状态集合；
  + nFinalStates: 终止状态个数；
  + finalStates: 管理终止状态集合；
  + finalStatesMap: 终止状态id对其结构体的映射，用于快速判断是否为终止状态；
  + nTransitions: 转移边个数；
  + transitions: 管理转移边集合。

重要的成员函数包括：

* + int getTransitions(WFSTTransition \*prev, WFSTTransition \*\*next): 从转移边prev得到指向的状态，再由状态得到所有出边的首地址next，返回出边个数；
  + real getFinalStateWeight( int stateIndex ): 返回终止状态的分数，若非终止状态返回最小值。
  + const char \* GetInLabel(int pos): 返回输入字符id对应的字符串；
  + const char \* GetOutLabel(int pos): 返回输出字符id对应的字符串。

（2） 搜索算法相关内容在wfst/WFSTDeocderCTC.h中，关键的结构体和类包括：

* **Token**: 记录在TLG.bin上传递的活跃令牌，成员变量包括：
  + trans: 对应转移边的指针；
  + prev: 同一帧下在网络中活跃令牌的链表指针；
  + acScore: 累计声学分数；
  + lmScore: 累计语言学分数；
  + score: 累计总分数；
  + istate: 记录从哪个状态过来的；
  + acPath: 记录输入序列（声学序列）路径的指针；
  + lmPath:记录输出序列（语言学序列）路径的指针。
* **Path**: 记录输入或输出序列（声学序列或语言学序列）路径，成员变量包括：
  + prev: 路径链表指针；
  + label: 输入字符id；
  + timestep: 出现的时间戳（帧数）。
* **WFSTDecoderCTC**: 实现帧同步解码算法的类，重要的成员变量包括：
  + m\_network: 加载了TLG.bin的WFSTNetwork的指针；
  + m\_model: 加载了端到端模型的E2EModel的指针;
  + m\_tokenList: 当前活跃令牌的链表头;
  + m\_bestToken: 当前分数最高的令牌;
  + m\_transTable: 查询转移边的哈希表;
  + m\_bestScore: 当前最高分数;
  + m\_normScore: 当前归一化分数;
  + m\_factor: 分数缩放因子，更大的值可以增加直方图剪枝精度;
  + m\_acWeight: 声学分数权重;
  + m\_lmWeight: 语言学分数权重;
  + m\_mainBeam: 基于当前最高分数，对活跃令牌剪枝的参数;
  + m\_odim: 端到端模型建模单元个数;
  + m\_hlength: 端到端模型输出语音特征序列的长度;
  + m\_blank\_penalty: 对CTC中<blank>概率的惩罚系数，e.g. 0.8指<blank>实数域概率乘以0.8;
  + m\_blank\_thresh: CTC中<blank>概率的阈值，e.g. 0.95指<blank>实数域概率超过0.95时，搜索路径只扩展<blank>符号；
  + m\_maxHyps: 当前允许的活跃令牌个数;
  + m\_hist: 直方图剪枝类，用于将活跃令牌裁剪到指定个数附近;
  + m\_result: 记录识别结果。

重要的成员函数包括：

* + void propagate(Token \* tok, int timestep, bool onlyBlk): 该函数基于timestep时刻的声学特征，将活跃令牌tok传递到其他转移边，过程中将生成一组新的活跃令牌。该函数以递归的形式进行，当传递到输入符号非<eps>时终止递归调用。若当前声学特征预测<blank>的概率超过m\_blank\_thresh，即onlyBlk为真，活跃令牌tok传递只能传递到输入符号为<blank>的转移边。
  + void propagate2final(Token \* tok): 该函数在所有声学特征上令牌传递结束后调用，将活跃令牌继续传递到终止状态。该函数以递归的形式进行，传递过程中转移边的输入符号只能是<eps>，即不再接受声学特征；
  + void start(E2EModel \* pGlobalModel, FLOAT \* feats, int len): 该函数读取语音信号特征的首地址以及长度，开始进行帧同步解码。若一次性输入所有语音信号特征，即认为是非流式解码；若后续还有语音信号特征要处理，则认为是流式解码。多线程之间使用同一个端到端模型，该模型参数通过pGlobalModel访问获取。首先，基于端到端模型计算输入语音信号特征对应的字符后验概率。然后，循环每一帧语音信号，调用propagate函数，传递活跃令牌，传递过程中会依据当前最高分数删除分数过低的令牌。处理完每一帧后会使用直方图剪枝限制当前活跃令牌的个数。最后，循环结束且后续无语音特征，调用propagate2final将所有活跃令牌传递到终止状态。
  + void finish(std::string featureName): 该函数在帧同步解码结束后对结果进行后处理。首先，从活跃令牌中选出分数最高的令牌，可以通过令牌上指向acPath的指针，或者指向lmPath的指针，以回溯的方式得到路径上的输入字符序列或输出字符序列。对于输入字符序列，需要通过塌缩规则的到最终的识别结果，而输出字符序列是经过分词后的识别结果。若需要获取每个输入字符对应的时间点，将会调用端到端模型中的CTC模块，使用维特比算法，实现识别结果和语音信号特征之间的强制对齐。

### 2.3 非流式字符同步解码功能开发

在本套端到端语音识别引擎中使用非流式字符同步解码时，默认使用的是基于编-解码器的端到端语音识别模型（Transformer）和基于RNN的语言模型（LSTMLM），其中端到端模型中仍保留CTC分支。字符同步解码的算法采用标准的束搜素算法（Beam Search），是一种基于前缀树的宽度优先搜索算法。在搜索空间动态扩展的过程中，通过限制每次拓展节点的个数，限制搜索空间的大小；解码器、CTC和语言模型对扩展的节点打分，根据分数排序的结果保留分数靠前的节点。

该解码算法实现过程中涉及关键的结构体或类包括：

* **Node**: 实现解码算法中的令牌类定义在search/node.h中，成员变量包括：
  + parent: 令牌父节点；
  + prev: 活跃令牌的链表指针;
  + label: 输出字符id;
  + score: 累计得分;
  + decScore: 当前节点端到端模型解码器得分;
  + ctcScore: 累计端到端模型CTC得分;
  + lmScore: 当前节点语言模型得分;
  + decMem: 当前节点端到端模型解码器内部状态;
  + ctcMem: 当前节点端到端模型CTC内部状态;
  + lmMem: 当前节点语言模型内部状态。
* **BeamSearch**: 实现字符同步解码的类定义在search/search.h中，重要的成员变量包括：
  + m\_root: 动态解码中前缀树的根节点，label对应<sos>，内部状态均为初始值；
  + m\_beamsz: 扩展前缀树时的最大节点个数;
  + m\_ctcBeamsz: 计算CTC分数时的最大节点个数;
  + m\_depth: 当前前缀树深度;
  + m\_maxDepth: 扩展前缀树的最大深度，限制识别结果字符个数;
  + m\_odim: 端到端模型建模单元个数;
  + m\_lmVocabsz: 语言模型建模单元个数，必须和m\_odim相等;
  + m\_hlength: 编码器输出声学特征序列的长度;
  + m\_dend: 判断是否满足继续扩展前缀树的条件，当前以<eos>结尾的最佳节点的分数减去所有以<eos>结尾的最佳节点的分数小于m\_dend时，不满足继续扩展前缀树的条件;
  + m\_toleration: 决定是否继续扩展前缀树的参数，当连续m\_toleration次不满足继续拓展前缀树的条件时，停止动态解码的过程;
  + m\_ctcWeight: CTC分数的权重，取值在[0, 1)，解码器分数的权重为1-m\_ctcWeight;
  + m\_lmWeight: 语言模型分数的权重;
  + m\_activeHypList: 当前活跃节点的链表头;
  + m\_endHypList: <eos>节点的链表头;
  + m\_bestEndScore: <eos>节点中的最大值;
  + m\_bestEndScores: 等深的<eos>节点中的最大值;
  + m\_forceEnd: 是否每轮强制拓展<eos>节点;
  + m\_bestPreEndScore: 在m\_forceEnd为真时，记录强制拓展<eos>节点中的最大值;
  + m\_bestPreEndHyp: 在m\_forceEnd为真时，记录强制拓展<eos>节点中分数最高的节点;
  + m\_forceAlign: 是否对识别结果强制对齐获取每个字符的时间点;
  + m\_silTreshold: 平滑字符时间点参数，取值（-∞, 0），阈值越高平滑效果越明显;
  + m\_first\_candidates: 对某个节点的所有子节点排序，筛选前m\_ctcBeamsz个节点;
  + m\_final\_candidates: 对当前拓展的节点排序，筛选前m\_beamsz个节点;
  + m\_getnbest: 输出多个候选结果个数;
  + m\_model: 加载了端到端模型的E2EModel的指针;
  + n\_lm: 加载了语言模型的LM的指针;
  + m\_vocab: 端到端模型使用的建模单元id到字符串的映射;
  + m\_result: 保存识别结果;
  + m\_nbest: ;保存多候选结果
  + m\_mlf: 保存识别结果中各个字符的时间点;
  + m\_timeReduction: 编码器对原始语音特征的采样倍数，e.g.，原始语音特征帧移10 ms，编码器输出特征帧移40 ms，采样倍数为4;
  + m\_speechSpeed: 每帧输入语音（10 ms），编码器最多输出的字符数目，e.g.，采样倍数为4时，每4帧（40 ms，对应编码器输出一帧），默认解码器最多输出一个字符，上限值m\_speechSpeed=0.25。中文环境下若每秒不超过10个字，即m\_speechSpeed=0.1。

重要的成员函数包括：

* + void start(E2EModel \* pGlobalModel, LM \* pGlobalLM, FLOAT \* feats, int len): 该函数读取语音信号特征的首地址以及长度，开始进行非流式字符同步解码。多线程之间使用同一个端到端模型和语言模型，模型参数分别通过pGlobalModel和pGlobalLM访问获取。首先，端到端模型的编码器进一步对输入语音信号特征编码，得到新的特征序列。如果m\_ctcWeight > 0，CTC模块计算输入语音信号特征对应的字符后验概率，为后面计算CTC分数做准备。然后，循环扩展前缀树的每一层。在循环内部，算法遍历当前层的活跃节点，每个活跃节点扩展出新的节点，并且调用解码器、CTC和语言模型计算新节点的分数。剪枝时，要求每个活跃节点扩展的节点数不超过m\_ctcBeamsz，整层拓展的活跃节点数不超过m\_beamsz。该层节点拓展结束后，若不满足继续拓展前缀树的要求，则停止循环。
  + void finish(std::string featureName): 该函数在字符同步解码结束后对结果进行后处理。首先，从<eos>节点中选出分数最高的节点，可以通过节点上指向父节点的指针，以回溯的方式得到识别结果。若需要输出多候选，对分数排序前N的<eos>节点进行回溯，的到多候选结果。若需要获取每个输入字符对应的时间点，将会调用端到端模型中的CTC模块，使用维特比算法，实现识别结果和语音信号特征之间的强制对齐。
  + void retrieveStates(Node \* node, Matrix<FLOAT> & states): 端到端模型的解码器在计算新扩展的子节点分数时，需要回溯当前节点node到根节点m\_root的所有历史状态，并存储到states中。设计该函数可以避免端到端模型解码器中的冗余计算，最小化存储历史状态所需的空间。

### 2.4 热词功能开发

本识别引擎提供的热词功能基于WFST，是基于3.3.2所述解码功能的扩展。热词功能是指提高用户定义的若干词条识别准确率的一种算法。实现方法分为以下两步：首先，将用户提交的若干词条编译为WFST；然后，识别引擎加载热词WFST干预识别。

（1）编译热词WFST

* buildKWFST -inbfsm <base.bin> -key <keywords.txt> -outbfsm <kw.bin>

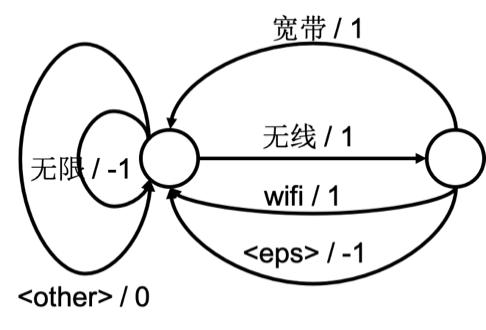
用户提供的<keywords.txt>包括若干词条，其形式可以是词语、短语、中英混合词语、集外词，无数量限制，不区分英文大小写（统一转为小写）。用户可以给每个词条赋予非零的整数作为权值（空格间隔）。正整数表示用户希望识别结果中出现该词条，负整数表示不希望出现该词条。不指定分数时默认该词条的权值为1。样例如下：

无线宽带

无线wifi 1

无限 -1

它表示用户希望识别出“无限宽带”和“无线wifi”，权值为1；希望不识别出“无限”，权值为-1。编译<kw.bin>时还需要提供解码使用的<base.bin>，即3.3.2中提到的TLG.bin。通过<base.bin>输出字符集合对词条分词。<kw.bin>的功能是接收任意的词语串，当词语串中包含<keywords.txt>中的词条，返回非零的分数。热词WFST的示意图如下：



（2） 热词WFST干预识别

在3.3.2基础上增加热词功能，涉及到的关键结构体和类包括：

* **RescoreToken**: 在wfst/WFSTDecoderCTC.h中定义了结构体，成员变量包括：
* next: RescoreToken的链表指针;
* id: 输出字符id;
* score: 累计分数；
* omitedlmScore: score中从<base.bin>上累计获取的分数。
* **TokenHotWord**: 在wfst/WFSTDecoderCTC.h中重新定义了结构体，与**Token**相比新增了以下成员变量：
* rescoreTokenList: RescoreToken的链表头;
* nActiveRescoreHyps: RescoreToken链表中的活跃令牌数;
* bestRescore: RescoreToken链表中活跃令牌分数的最大值。
* **WFSTDecoderCTCHotWord**: 在wfst/WFSTDecoderCTC.h中继承父类**WFSTDecoderCTC**，其中成员变量增加了：
* m\_rescoreNetwork: 加载了<kw.bin>的WFSTNetwork;
* m\_keyWeight: RescoreToken.score计入TokenHotWord.score的权值;
* m\_lmScoreOmit: TokenHotWord.lmScore计入RescoreToken.score的权值。

新增重要的成员函数包括：

* void propagateRescoreToken: 当propagate函数中令牌传递到输出字符不为<eps>的转移边时，调用该函数，将令牌上绑定的子令牌在热词WFST传递。该函数以递归的形式进行，当传递到输入符号非<eps>时终止递归调用。

在热词干预识别过程中，每条搜索路径满足如下性质：命中热词部分的分数为 m\_acWeight \* acScore + m\_lmWeight \* (1-m\_lmScoreOmit) \* lmScore + m\_keyWeight \* bestRescore；未命中热词部分的分数为m\_acWeight \* acScore + m\_lmWeight \* lmScore。通过这种设计方式，提升热词识别的准确率，并且不影响非热词识别的准确率。

### 2.5多候选和置信度功能开发

本引擎提供了多候选和置信度功能开发。分别支持在解码过程中提供多个候选识别结果，每个候选结果进行置信度打分，并最终根据置信度得分对多个候选结果进行排序。

主要实现的思路则是在识别进行到最终阶段时，根据还处于活跃状态的hyps分别进行打分，并且根据所需要的多候选结果数目，保留得分最高的N个hyps作为多候选结果。此外，考虑到本引擎使用了WFSTCTC方式进行解码，可能会存在多个hyps对应同一个文本识别结果的情况，根据这种情况对hyps进行合并，并最终返回识别结果。

本功能所使用的到类和变量有：

在WFSTDecoderCTC.h

m\_Nbest：多候选结果的数目

m\_mlf：储存多个候选文本结果的指针

m\_mlf\_start：储存多个候选结果起始时间点的指针

m\_mlf\_end：储存多个候选结果终止时间点的指针

m\_mlf\_prob：储存多个候选结果似然度得分的指针

m\_confidence：储存多个候选结果置信度得分的指针

m\_result\_hist：对多候选识别结果进行直方图剪枝的对象

get1best：判定是否多候选为1，也就是不进行多候选操作

本功能所额外需要或改动的函数有：

WFSTDecoderCTC.cpp:

void propagate2final(Token \* tok, bool getmid)：令牌传递的最终状态，在本状态中将活跃的所有hyps存入直方图，并进行直方图剪枝获取Nbest结果

void finish()：最终处理多候选结果，将其整理成外部接口所需要的形式，包括文本结果，时间点，置信度等多个信息。

void updataCands( Token\* in)：多候选结果处理，用于合并对应相同文本结果的不同hyps。并对合并之后的结果进行排序，确保排名第一的候选结果和不做多候选的识别结果相吻合。

void WFSTCTCForceAlign(list<int> hyp, list<int> timestep\_list, int candsNum)：

对多候选结果进行逐条处理，分别得到每个候选结果的时间点，文本结果和置信度。

WFSTDecoderCTC.h

std::string getresults(int i)：返回第i个多候选结果

std::vector<int>\* getmlfStart(int i)：返回第i个多候选结果的起始时间的点

std::vector<int>\* getmlfEnd(int i) ：返回第i个多候选结果的结束时间点

std::vector<std::string>\* getmlfWord(int i) ：返回第i个多候选结果的文本信息

std::vector<FLOAT>\* getmlfProb(int i) ：返回第i个多候选结果的似然度

FLOAT getconfidence(int i) ：返回第i个多候选结果的置信度打分

int getCandNum()：返回多候选结果的总数

E2EEngine.cpp:

WordResult\* GetE2EResult()：进一步将多候选结果整理成WordResult结构体指针的形式。