个性化语音合成项目
技术概览

2020.12.06

本周主要内容：

- 个性化语音合成技术整体架构。
- GE2E 的设计与实现。
- LSTM 结构设计及实现。
- BiLSTM的实现逻辑。

1 可控 ⇒ 个性化一个方面

duration

emotion Embedding

GST global style

2 多说话模型

文本 → 语音

dilated CNN

wavenet

waveRNN

waveglow

Tacotron2 fastspeech Deepvoice

attention

80 → 24000 上采样 conv

3 声码器

基本概念-声音的表示

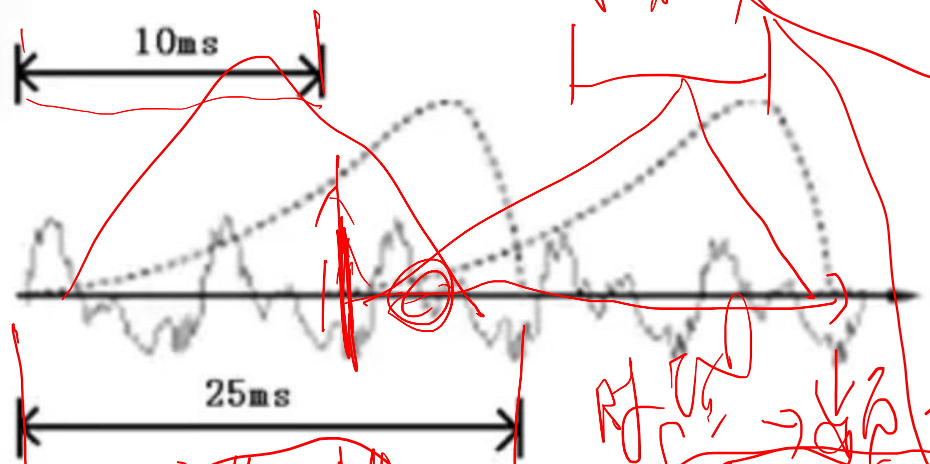
声音是一种波。常见的mp3、wmv等格式都是压缩格式，必须转成非压缩的纯波形文件来处理，比如Windows PCM文件，也就是俗称的wav文件。wav文件由存储的文件头和声音波形的采样点构成。

采样率 16k 24k scipy wavfile soundfile torchaudio

22.04

基本概念-帧长和帧移

在语音识别时需要把首尾端的静音切除以降低对后续步骤造成的干扰，这个静音切除的操作一般称为VAD。要对声音进行分析，需要对声音分帧，也就是把声音切开成小段，每小段称为一帧。分帧操作一般不是简单的切开，而是使用移动窗函数来实现。帧与帧之间一般是有交叠的。一般情形每帧的长度为25毫秒，每两帧之间有25-10=15毫秒的交叠。我们称为以帧长25ms、帧移10ms分帧。



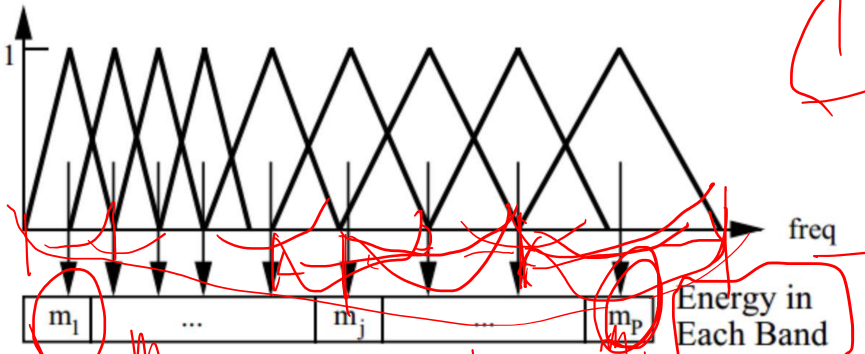
sel

秒 1ms

线程中 周期性

基本概念

- 人耳对声音频谱的响应是非线性的。前端处理算法，以类似于人耳的方式对音频进行处理，可以提高语音识别的性能。FilterBank分析就是这样的一种算法。FBank特征提取要在预处理之后进行，这时语音已经分帧，我们需要逐帧提取FBank特征。



基本概念-声码器

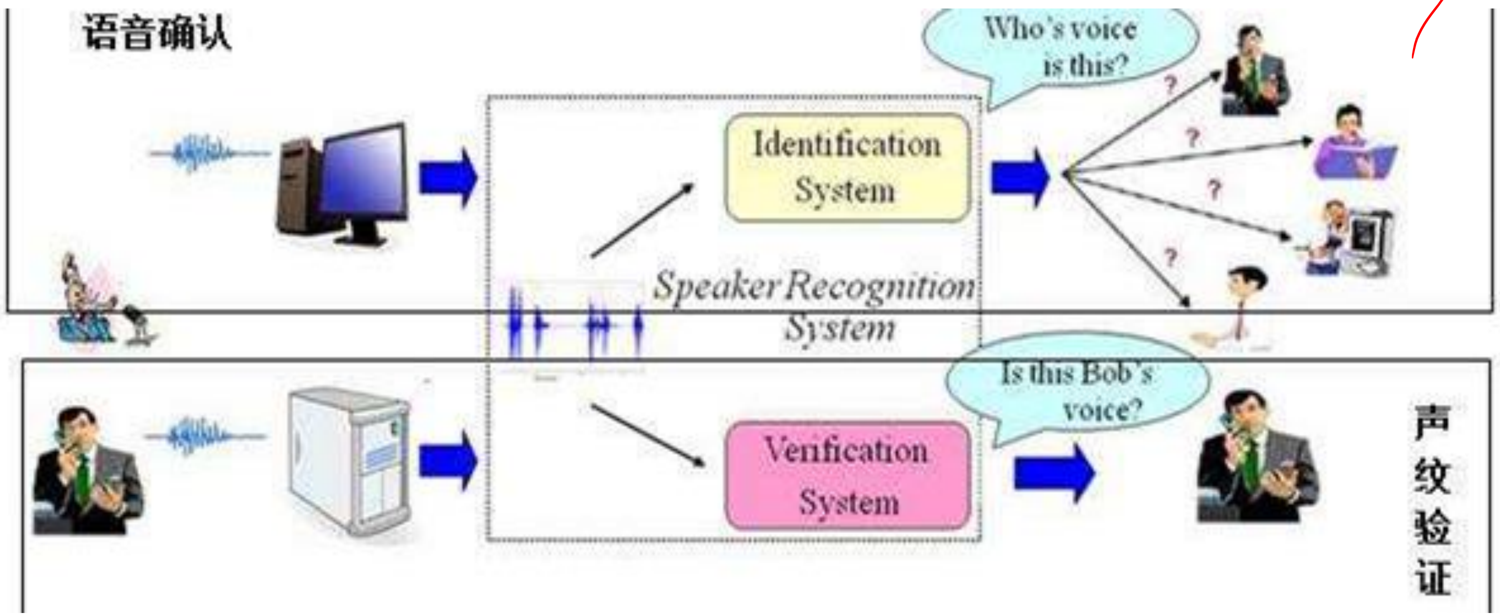
- 声码器在发送端对语音信号进行分析，提取出语音信号的特征参量加以编码和加密，以取得和信道的匹配，经信息通道传递到接受端，再根据收到的特征参量恢复原始语音波形。分析可在频域中进行，对语音信号作频谱分析，鉴别清浊音，测定浊音基频，进而选取清-浊判断、浊音基频和频谱包络作为特征参量加以传送。分析也可在时域中进行，利用其周期性提取一些参数进行线性预测，或对语音信号作相关分析。
- 声码器可以分成：通道式声码器、共振峰声码器、图案声码器、线性预测声码器、相关声码器、正交函数声码器。

频域特征

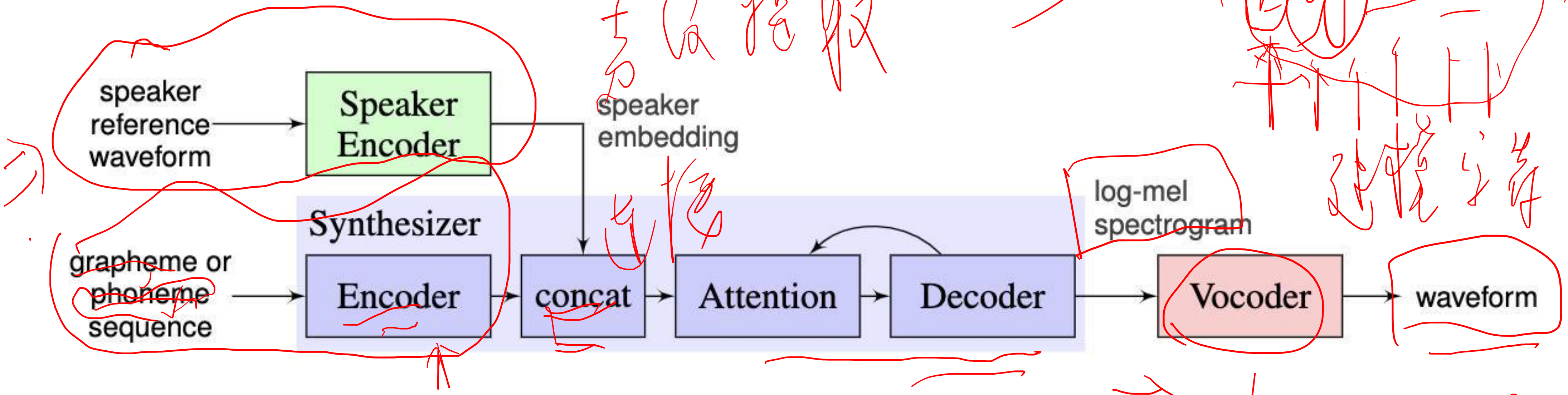
wave

基本概念-声纹 (VoicePrint)

所谓声纹(Voiceprint)，是用电声学仪器显示的携带言语信息的声波频谱。人类语言的产生是人体语言中枢与发音器官之间一个复杂的生理物理过程，人在讲话时使用的发声器官--舌、牙齿、喉头、肺、鼻腔在尺寸和形态方面每个人的差异很大，所以任何两个人的声纹图谱都有差异。每个人的语音声学特征既有相对稳定性，又有变异性，不是绝对的、一成不变的。这种变异可来自生理、病理、心理、模拟、伪装，也与环境干扰有关。



个性化语音合成整体架构



该系统由3个可独立训练模块构成：

说话人特征编码器-speaker encoder ： 实现对目标说话人发音特征提取。

合成器-synthesizer：实现文本到梅尔谱(mel spectrogram) Seq2Seq变换。

声码器-vcoder: 实现梅尔谱（mel spectrogram）到波形(Waveform)变化。

Speaker Encoder

- 说话人特征编码器具有鉴别特定说话人的功能。输入为短时自适应语音信号，输出为说话人在发音特征向量空间中的特征向量。该特征编码与语音对应的文本及背景噪音独立，是对说话人发音特质的描述。说话人特征编码器为语音合成模块提供个性化定制参考。

Speaker Encoder

256维

网络由3个LSTM层构成。每层LSTM包含768个cell，后接仿射层(Projection，通常目的是减少计算量和特征降维)变换到256维。最终的Embeding由该说话人最后一帧在神经网络运算结束后通过对最后一层在每个时间步的输出进行平均和L2 Norm生成。

```python
def __init__(self, device, loss_device):
    super().__init__()
    self.loss_device = loss_device

    # Network defition
    self.lstm = nn.LSTM(input_size=mel_n_channels,
                        hidden_size=model_hidden_size,
                        num_layers=model_num_layers,
                        batch_first=True).to(device)
    self.linear = nn.Linear(in_features=model_hidden_size,
                            out_features=model_embedding_size).to(device)
    self.relu = torch.nn.ReLU().to(device)

    # Cosine similarity scaling (with fixed initial parameter values)
    self.similarity_weight = nn.Parameter(torch.tensor([10.])).to(loss_device)
    self.similarity_bias = nn.Parameter(torch.tensor([-5.])).to(loss_device)

    # Loss
    self.loss_fn = nn.CrossEntropyLoss().to(loss_device)
```

TE2E

k個人, 1…M 不同的語句

- Tuple-based end-to-end model

$$\{\mathbf{x}_{j\sim}, (\mathbf{x}_{k1}, \ldots, \mathbf{x}_{kM})\} \longrightarrow \{\mathbf{e}_{j\sim}, (\mathbf{e}_{k1}, \ldots, \mathbf{e}_{kM})\}.$$

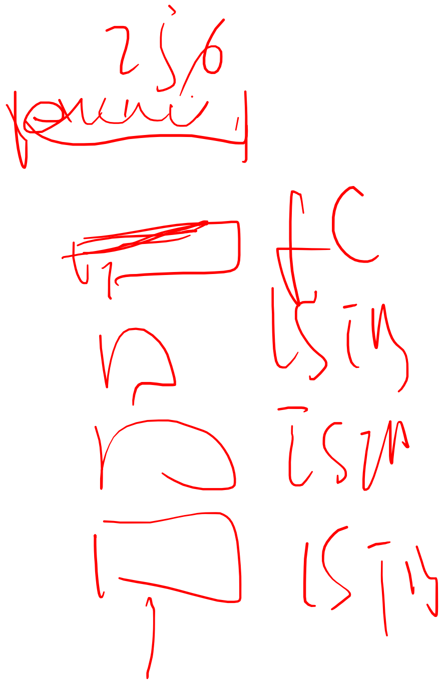
中心

$$\mathbf{c}_k = \mathbb{E}_m[\mathbf{e}_{km}] = \frac{1}{M}\sum_{m=1}^{M} \mathbf{e}_{km}.$$

$$s = w \cdot \cos(\mathbf{e}_{j\sim}, \mathbf{c}_k) + b,$$

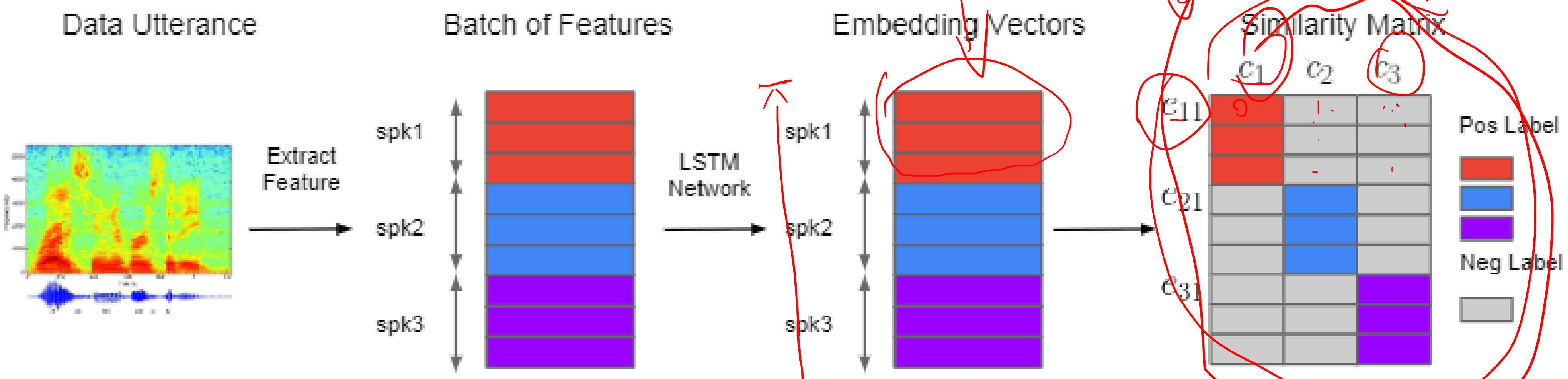
$$L_{\mathrm{T}}(\mathbf{e}_{j\sim}, \mathbf{c}_k) = \delta(j,k)\Big(1 - \sigma(s)\Big) + \Big(1 - \delta(j,k)\Big)\sigma(s).$$

$$\sigma(x) = 1/(1 + e^{-x})$$



GE2E

GE2E系统结构：



utterance

similarity

GE2E

embedding

$$\mathbf{e}_{ji} = \frac{f(\mathbf{x}_{ji}; \mathbf{w})}{\|f(\mathbf{x}_{ji}; \mathbf{w})\|_2}.$$

L2-norm

$$\mathbf{S}_{ji,k} = w \cdot \cos(\mathbf{e}_{ji}, \mathbf{c}_k) + b,$$

$$\mathbf{c}_j^{(-i)} = \frac{1}{M-1} \sum_{\substack{m=1 \\ m \neq i}}^{M} \mathbf{e}_{jm},$$

$$\mathbf{S}_{ji,k} = \begin{cases} w \cdot \cos(\mathbf{e}_{ji}, \mathbf{c}_j^{(-i)}) + b & \text{if} \quad k = j; \\ w \cdot \cos(\mathbf{e}_{ji}, \mathbf{c}_k) + b & \text{otherwise.} \end{cases}$$

矩阵

1

$$L(\mathbf{e}_{ji}) = 1 - \sigma(\mathbf{S}_{ji,j}) + \max_{\substack{1 \leq k \leq N \\ k \neq j}} \sigma(\mathbf{S}_{ji,k}),$$

log max

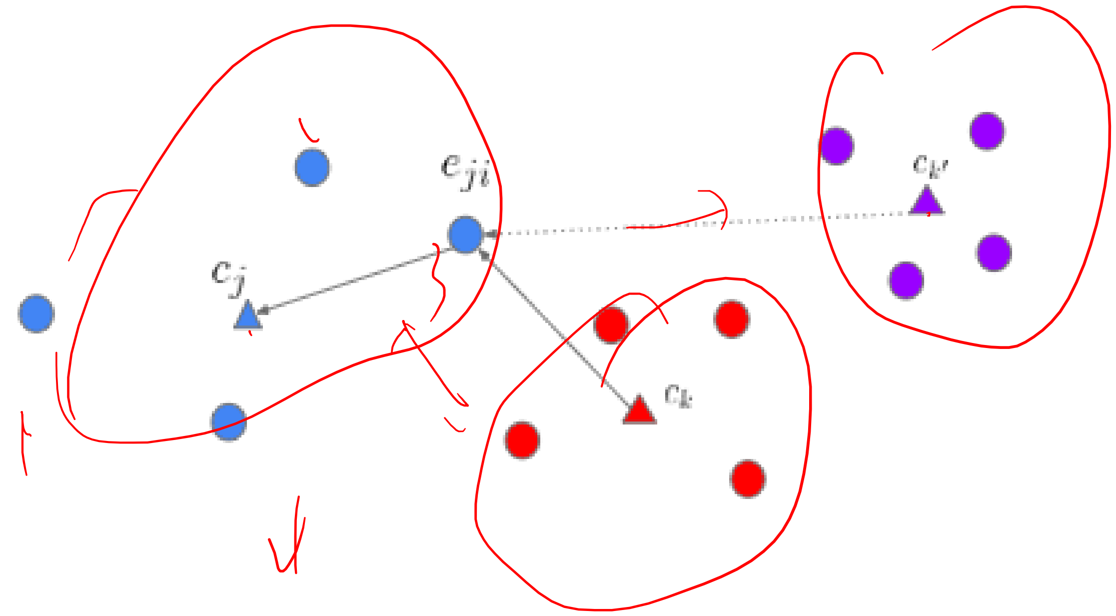
2

$$L(\mathbf{e}_{ji}) = -\mathbf{S}_{ji,j} + \log \sum_{k=1}^{N} \exp(\mathbf{S}_{ji,k}).$$

$$L_G(\mathbf{x}; \mathbf{w}) = L_G(\mathbf{S}) = \sum_{j,i} L(\mathbf{e}_{ji}).$$

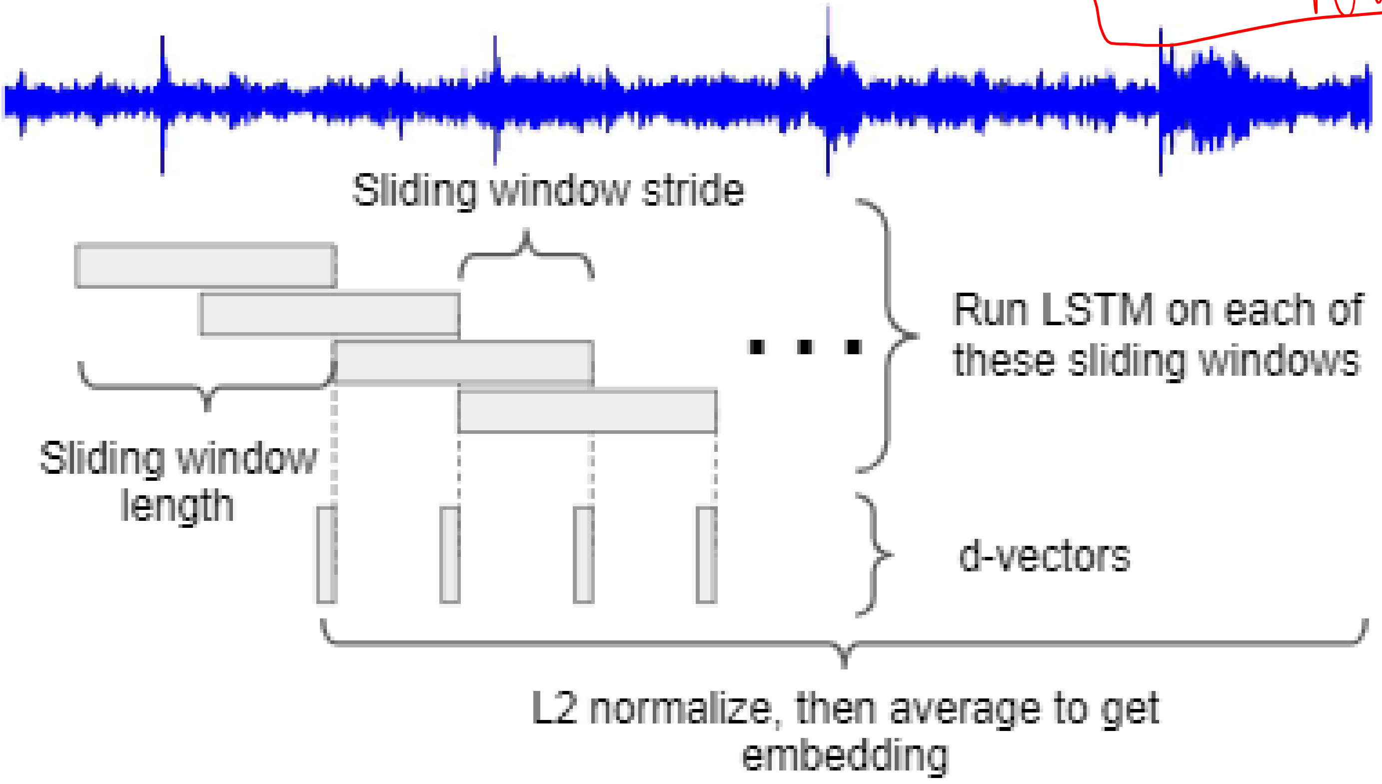
GE2E

GE2E优化过程：



GE2E

GE2E 滑动窗口机制



训练的 10个

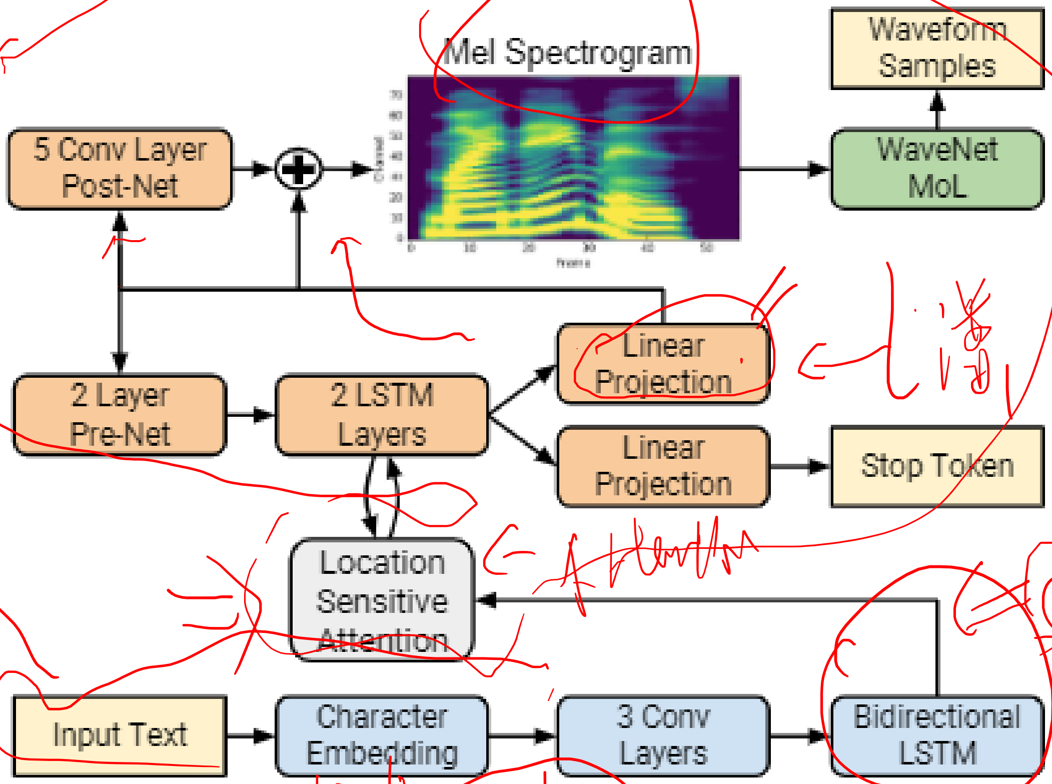
random

数据扩充

增强

合成器(Synthesizer)

Nvidia / tacotron2



decoder

MFCC / Fbank

attention

concat

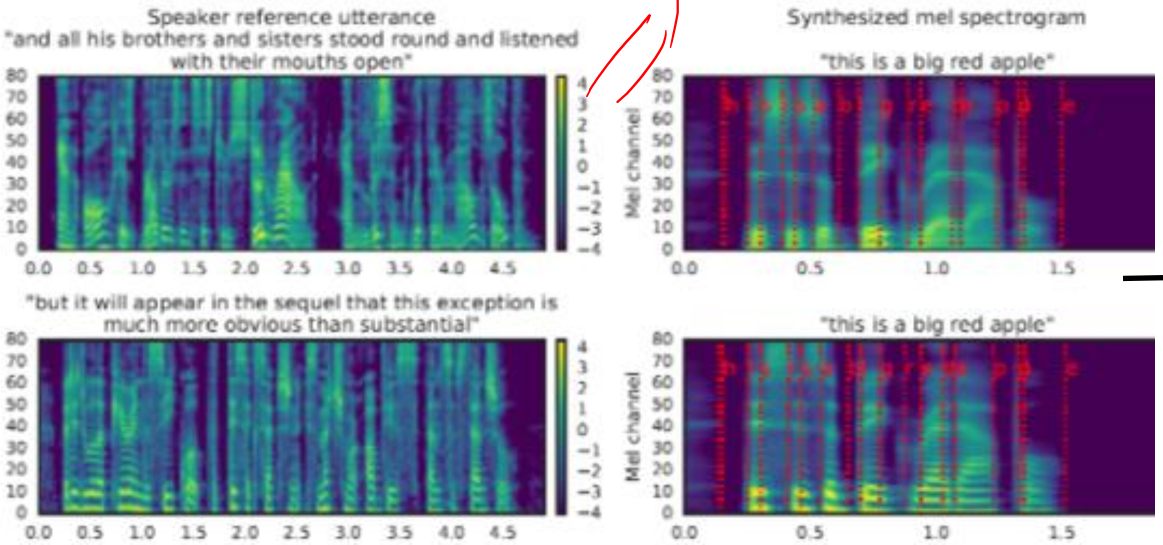
encoder

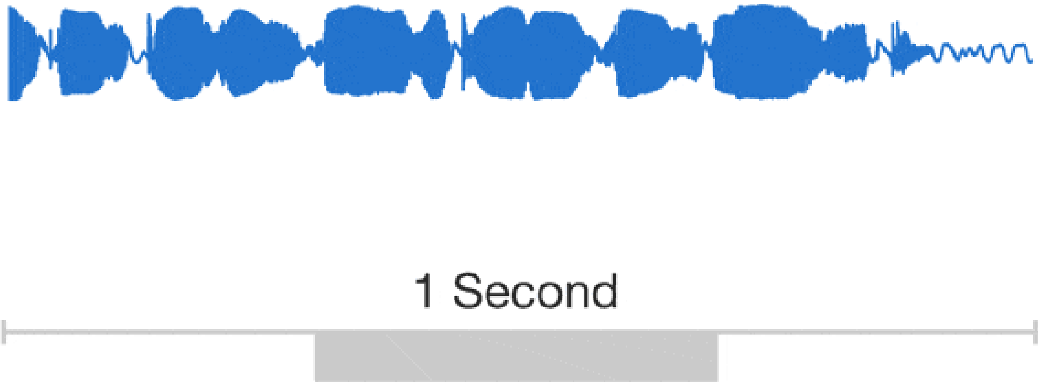
en → cn

声码器

频谱到波形的变换过程。

wave form





声码器-传统算法

- Griffin-lim算法

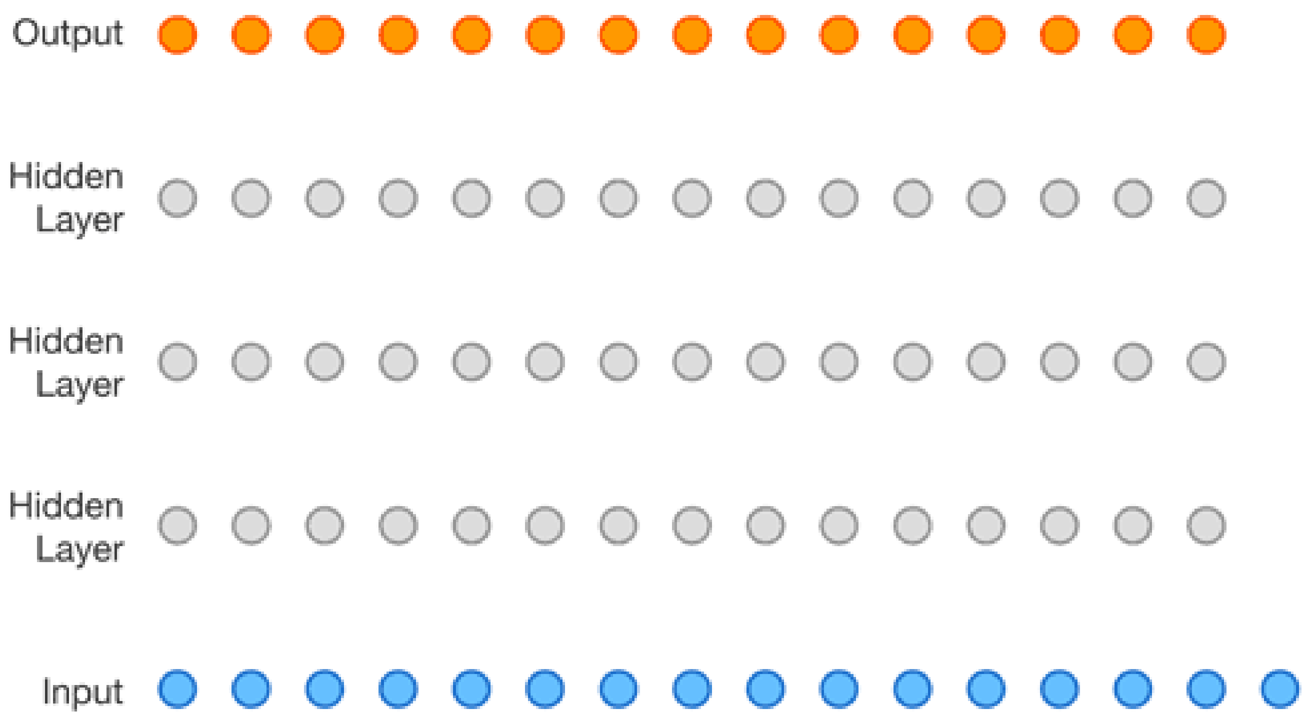
griffin-lim重建语音信号需要使用到幅度谱和相位谱。而MEL谱当中是不含相位信息的，因此griffin-lim在重建语音博形的 时候只有MEL谱可以利用，利用帧与帧之间的关系估计出相位信息，重建语音波形。

这里的MEL谱可以看做是实部，而相位信息可以看做是虚部，通过对实部和虚部的运算，得到最终的结果。

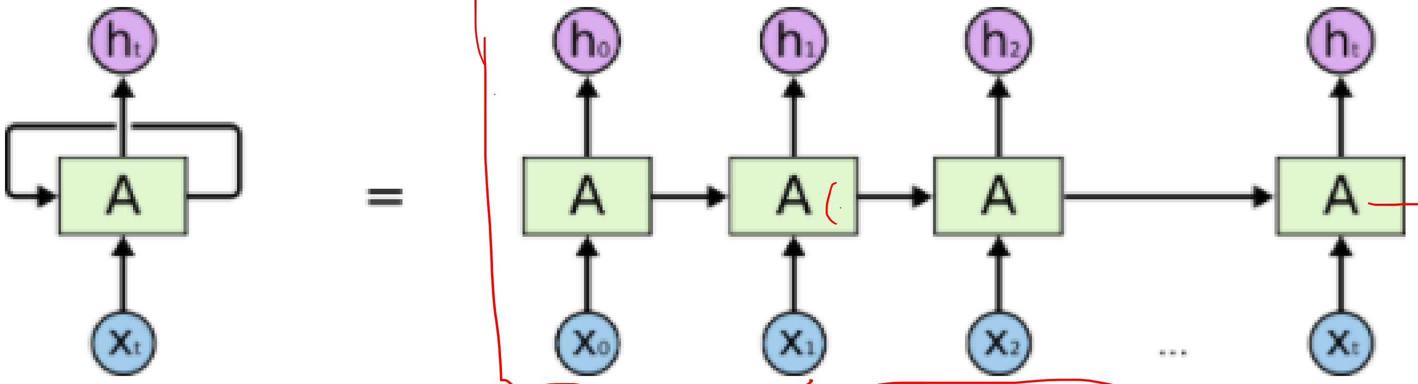
声码器-griffin-lim算法流程

- 随机初始化一个相位谱
- 用这个相位谱与已知的幅度谱（来自MEL谱）经过ISTFT（逆傅里叶变换）合成新的语音波形
- 用合成语音做STFT， 得到新的幅度谱和新的相位谱
- 丢弃新的幅度谱，用已知幅度谱与新的相位谱合成新的语音
- 重复2,3,4多次，直至合成的语音达到满意的效果或者迭代次数达到设定的上限

声码器-wavenet



Recurrent Neural Networks



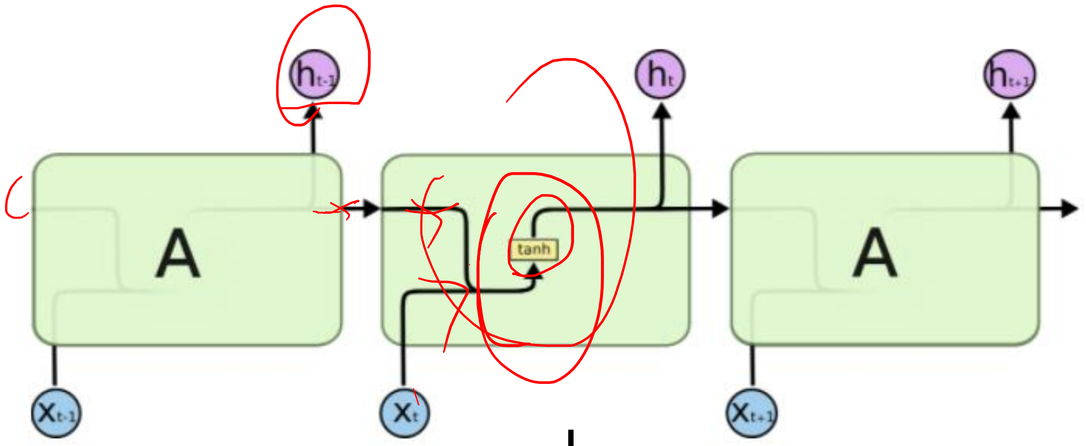
An unrolled recurrent neural network.

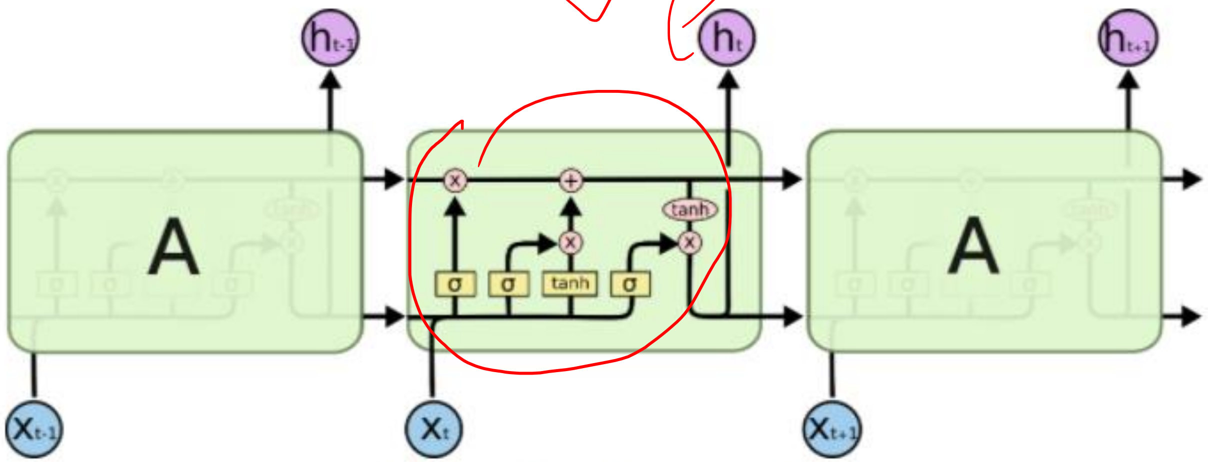
cell

状态变量

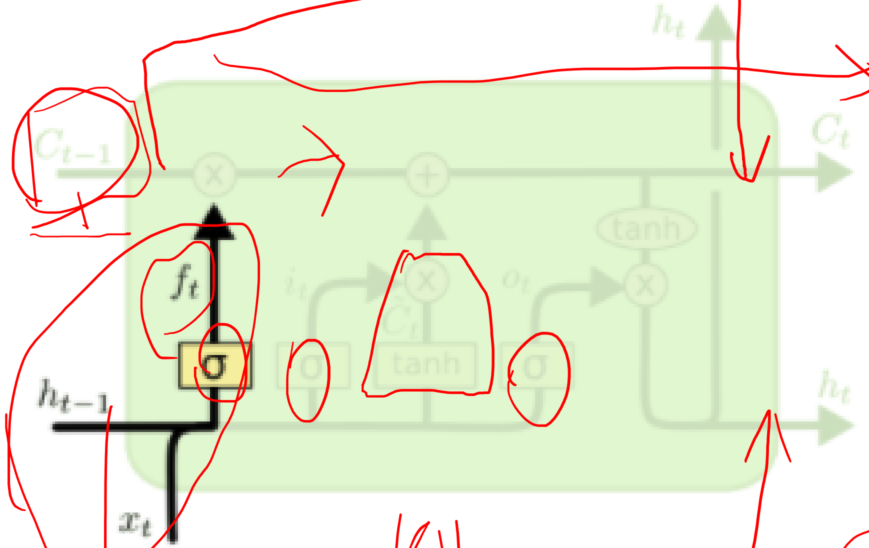
update

RNN->LSTM





LSTM遗忘门



$$f_t = \sigma(W_f \cdot [h_{t-1}, x_t] + b_f)$$

LSTM GRU

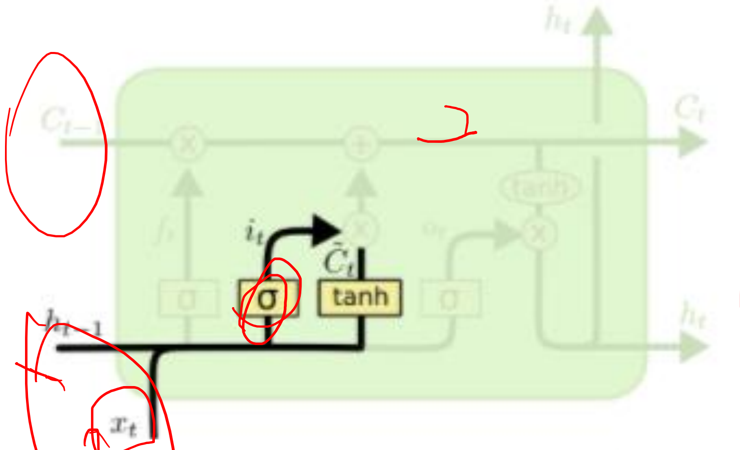
ResNet

H(x)=F(x)+x

projection layer

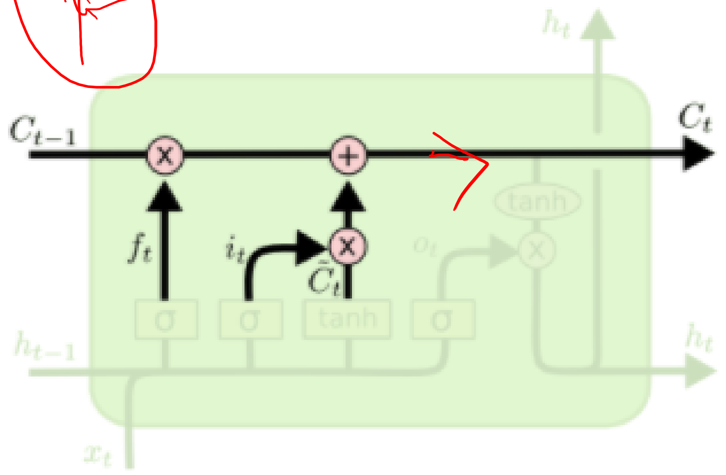
bottle neck

LSTM记忆门



$$i_t = \sigma\left(W_i \cdot [h_{t-1}, x_t] \ + \ b_i\right)$$
$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] \ + \ b_C)$$

记忆 知识

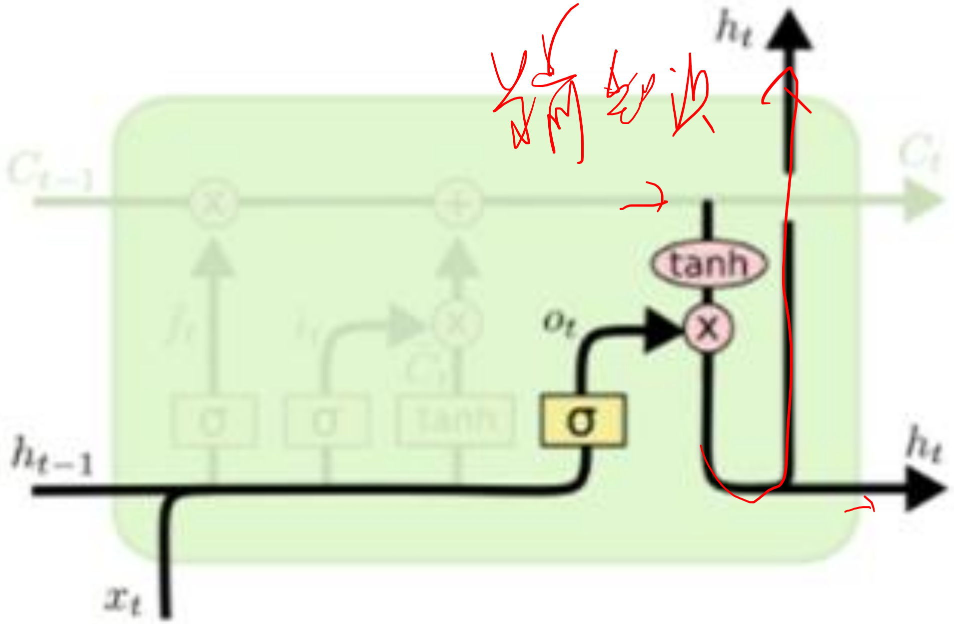


$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$$

剩余知识

当前 新增知识量

LSTM输出门



$$o_t = \sigma\left(W_o\left[h_{t-1}, x_t\right] + b_o\right)$$

$$h_t = o_t * \tanh\left(C_t\right)$$

LSTM实现

```
class Gate {
public:
    int     _n_sub_seq_size, _n_batch_size;
    int     _n_tbptt;
    size_t _n_feat_dim, _n_cell_dim, _n_rec_dim;

    FMatrix *_bias, *_wc;
    Weight *_wx, *_wr;

    StateMatrix _stat_oe;

    Gate(int tbptt, size_t feat_dim, size_t cell_dim, size_t outDim);
    ~Gate();
    void set_batch_size(int sub_seq_size, int batch_size);
    void input_forward(InOutput &in_out, IN_OUT_TYPE_T i_type, IN_OUT_TYPE_T o_type);
    void time_forward(int t, FMatrix &oc, FMatrix &or_);
};
```

```
class OutGate : public Gate {
public:
    OutGate(int tbptt, size_t feat_dim, size_t cell_dim, size_t rec_dim)
        : Gate(tbptt, feat_dim, cell_dim, rec_dim) {}
    void time_forward(int t, FMatrix &Oc, FMatrix &Or);
};
```

Gate的定义。不同的Gate实现具体的time forward算法。

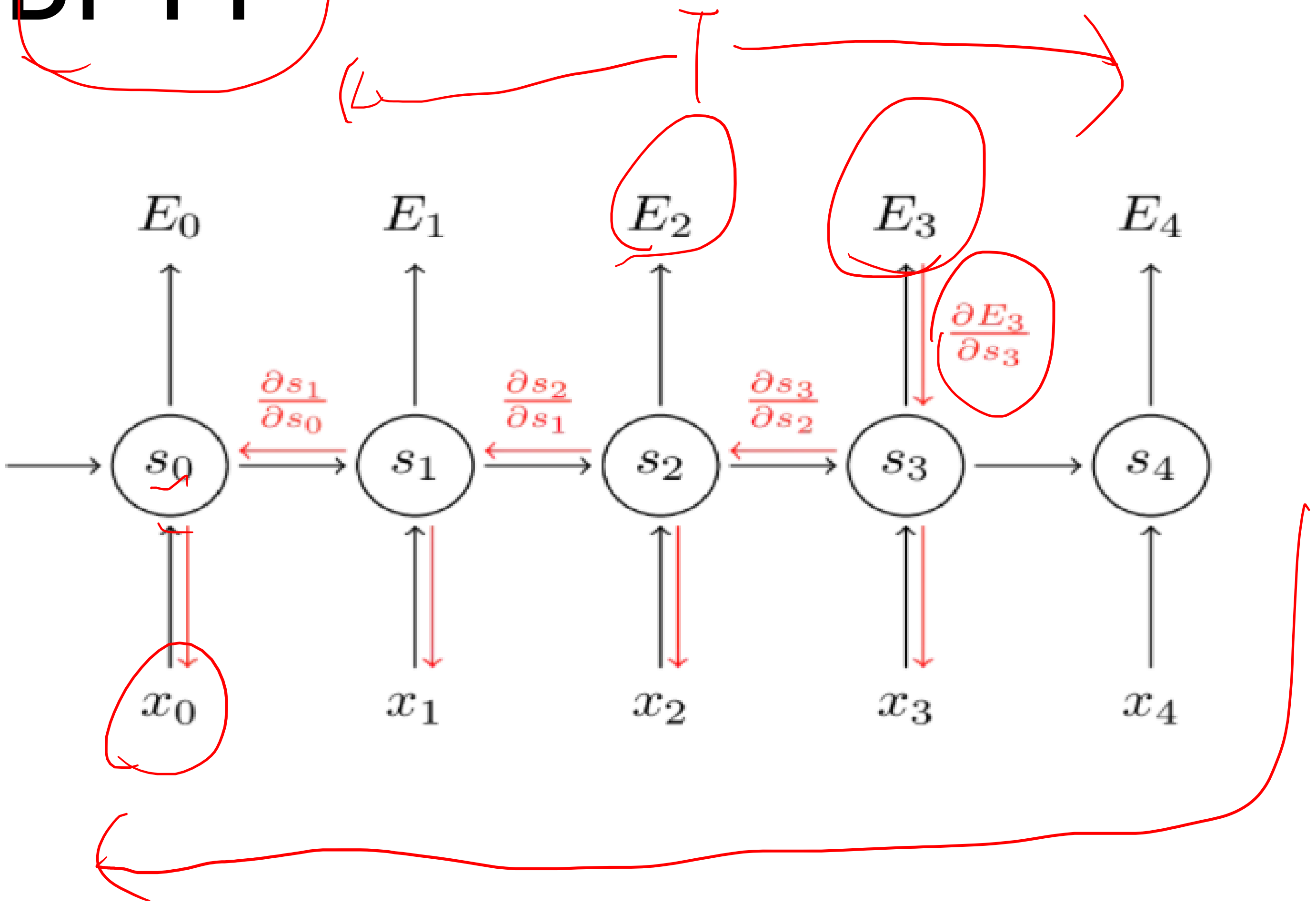
LSTM实现

```cpp
LstmLayer::LstmLayer(LstmConfig &conf) : Layer(conf) {
  init();
  _weights = static_cast<LstmWeights *>(conf._weights);
  size_t feat_dim = conf.in_dim();
  size_t cell_dim = conf.cell_dim();
  size_t out_dim = conf.out_dim();
  size_t prj_dim = conf.prj_dim();
  size_t rec_dim = conf.rec_dim();

  _n_tbptt = 1;  // Tbptt;
  _n_feat_dim = feat_dim;
  _n_cell_dim = cell_dim;
  _n_rec_dim = rec_dim;
  _n_prj_dim = prj_dim;
  _n_out_dim = out_dim;

  // gate
  int rec = (int)_n_rec_dim;
  if (rec == 0) {
    rec = (int)_n_out_dim;
  }
  _output_g = new OutGate(1, feat_dim, cell_dim, rec);
  _input_g = new Gate(1, feat_dim, cell_dim, rec);
  _forget_g = new Gate(1, feat_dim, cell_dim, rec);
  _cells = new Cells(1, feat_dim, cell_dim, rec);

  set_weights(static_cast<LstmWeights *>(conf._weights));
  _cells->_min = _weights->_cec_out_limit_lo;
  _cells->_max = _weights->_cec_out_limit_hi;

  _rec_act = Activation::create(conf.rec_act());

  set_batch_size(conf.sub_seq_size(), conf.batch_size());
}
```

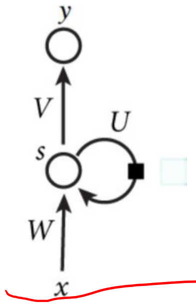
project

```cpp
for (int t = 0; t < (int)sub_seq_size; t++) {
  size_t start_row = _n_tbptt + t - 1;
  size_t end_row = _n_tbptt + t;

  FMatrix &r_pre_o =
      _statoe_r._o->range_row(start_row, end_row, _n_batch_size);
  _in_out.set_input(&r_pre_o);
  _in_out.set_output(&_wx_out.range_row(t, t + 1, _n_batch_size));
  _weights->_wr_iofc.mul(_in_out, INOUT_TYPE, INOUT_TYPE, NULL, 1.0f, 1.0f);
  _in_out.clear_input(INOUT_TYPE);
  _in_out.clear_output(INOUT_TYPE);

  // input gage Wic * Ct-1
  FMatrix &wxwr_out = _wx_out.range_row(t, t + 1, _n_batch_size);
  _i_out.copy_from(wxwr_out.range_col(0, 1 * _n_cell_dim, 1));
  _i_out.mul_diag_mat_sigmoid(
      _statoe_c._o->range_row(start_row, end_row, _n_batch_size),
      *_weights->_i_wc._f_w, _i_out, 1.0f, 1.0f);

  // forget gate Wfc * Ct-1
  _f_out.copy_from(wxwr_out.range_col(1 * _n_cell_dim, 2 * _n_cell_dim, 1));
  _f_out.mul_diag_mat_sigmoid(
      _statoe_c._o->range_row(start_row, end_row, _n_batch_size),
      *_weights->_f_wc._f_w, _f_out, 1.0f, 1.0f);
  // cell
  FMatrix &c_cur_out = _statoe_c._cur_o->range_row(t, t + 1, _n_batch_size);
  FMatrix &c_pre_out = _statoe_c._o->range_row(t, t + 1, _n_batch_size);
  _g_out.copy_from(wxwr_out.range_col(3 * _n_cell_dim, 4 * _n_cell_dim, 1));
  // g()
  _g_out.tanh();
  // Ft . Ct-1 + It . g()
  c_cur_out.elem_mul_add(_g_out, _i_out, _f_out, c_pre_out);
```

BPTT


E_0
E_1
E_2
E_3
E_4
$\frac{\partial E_3}{\partial s_3}$
$\frac{\partial s_1}{\partial s_0}$
$\frac{\partial s_2}{\partial s_1}$
$\frac{\partial s_3}{\partial s_2}$
s_0
s_1
s_2
s_3
s_4
x_0
x_1
x_2
x_3
x_4

BPTT



RNN

$$
\begin{cases}
s_t = Uh_{t-1} + Wx_t \\
h_t = tanh(s_t) \\
z_t = Vh_t \\
\hat{y}_t = \text{softmax}(z_t) \\
E_t = -y_t^T log(\hat{y}_t) \\
E = \sum_t^T E_t
\end{cases}
$$

$$
\begin{aligned}
\frac{\partial E_t}{\partial V_{ij}} &= tr\left(\left(\frac{\partial E_t}{\partial z_t}\right)^T \cdot \frac{\partial z_t}{\partial V_{ij}} \right) \\
&= tr\left((\hat{y}_t - y_t)^T \cdot \begin{bmatrix} 0 \\ \vdots \\ \frac{\partial z_t^{(i)}}{\partial V_{ij}} \\ \vdots \\ 0 \end{bmatrix} \right) \\
&= r_t^{(i)} h_t^{(j)}
\end{aligned}
$$

$$\frac{\partial E_t}{\partial U} = \sum_{k=0}^{t} \delta_k \otimes h_{k-1}$$

$$\frac{\partial E}{\partial U} = \sum_{t=0}^{T} \sum_{k=0}^{t} \delta_k \otimes h_{k-1}$$

$$\frac{\partial E_t}{\partial W} = \sum_{k=0}^{t} \delta_k \otimes x_k$$

$$V := V - \lambda \sum_{t=0}^{T} (\hat{y}_t - y_t) \otimes h_t$$

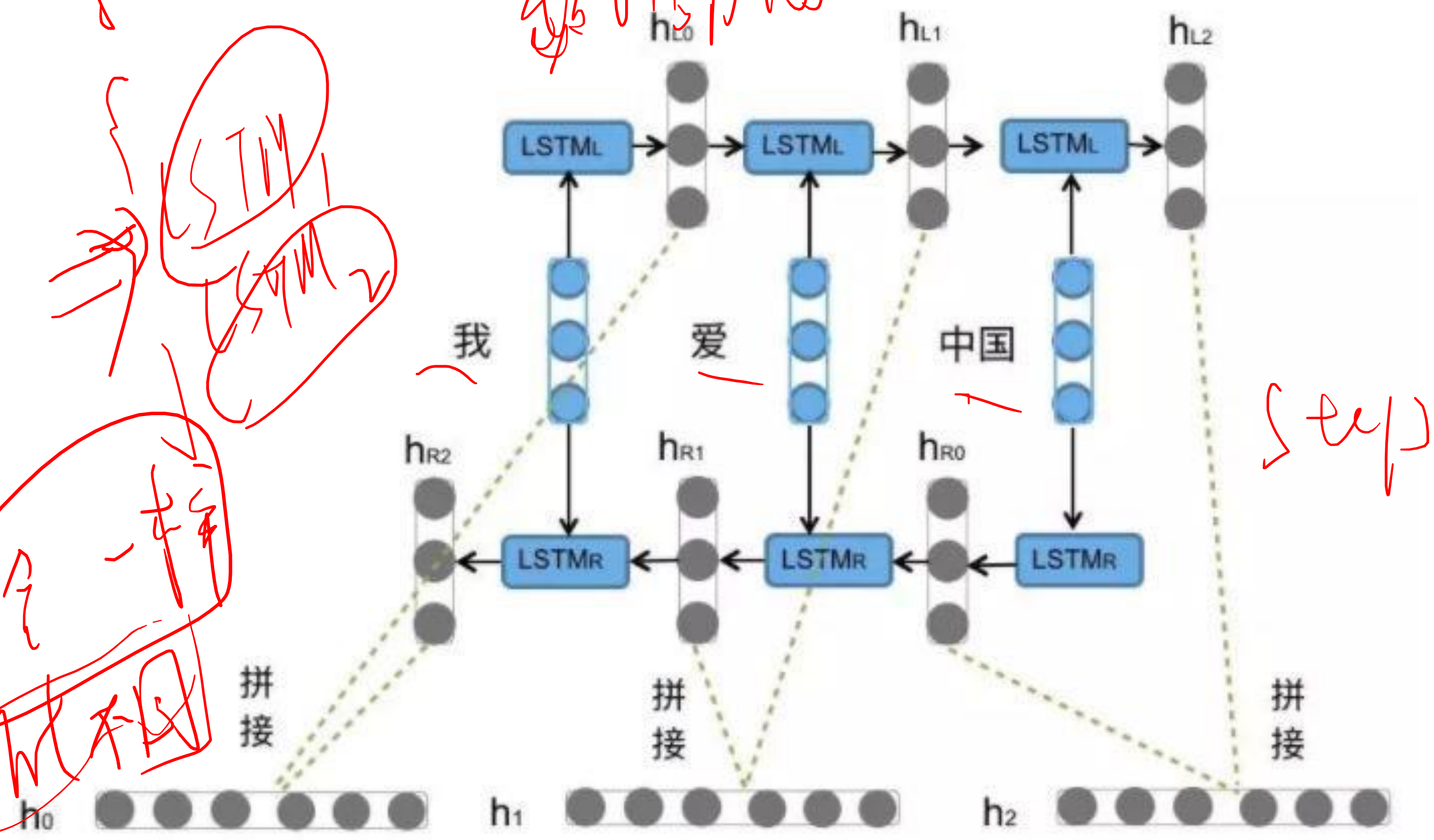
$$U := U - \lambda \sum_{t=0}^{T} \sum_{k=0}^{t} \delta_k \otimes h_{k-1}$$

$$W := W - \lambda \sum_{t=0}^{T} \sum_{k=0}^{t} \delta_k \otimes x_k$$

[BPTT算法推导 - Hiroki - 博客园 (cnblogs.com)]

BiLSTM

- 前向的依次输入“我”，“爱”，“中国”得到三个向量。
- 后向的依次输入“中国”，“爱”，“我”得到三个向量。
- 最后将前向和后向的隐向量进行拼接得到。



两个LSTM → forward 操作

Concat

set weight

数据反向

LSTM

LSTM2

完全一样

weight 不同

step

Bilstm的实现

```cpp
class FastBiLstmLayer : public Layer {
public:
    FastBiLstmLayer(FastBiLstmConfig& config);
    ~FastBiLstmLayer();
    void   forward(InOutput &in_out, IN_OUT_TYPE_T i_type);
    void   set_batch_size(int frame_size);
    void   reset(int sent_idx);
    void   store_current_out();
    inline int is_append() {
        return _is_append;
    }

protected:
    InOutput    _invert_in_out;
    InOutput    _in_out;//正向输入
    FMatrix     _invert_in_mat, _inver_out_mat;
    //记录forward和backward LSTM的输入数据的映射顺序
    CpuIVector _map_id_vec1, _map_id_vec2; //used for encode and decode mapId
    int         _map_length;
    int         _is_append;

    ///////////////////////////
    FastLstmLayer    *_lstm_layer;
    FastLstmWeights *_fwd_weight;
    Container<FMatrix*> _fwd_his_o;

    FastLstmWeights *_bwd_weight;
    Container<FMatrix*> _bwd_his_o;
};
```

```cpp
BiLstmLayer::BiLstmLayer(BiLstmConfig &conf) : Layer(conf) {
    //结构一致
    _lstm_layer = new LstmLayer(*conf.normal_config());
    //前向weight和反向weight
    _fwd_weight = static_cast<LstmWeights *>(conf.normal_config()->_weights);
    _bwd_weight = static_cast<LstmWeights *>(conf.inver_config()->_weights);
    //是否采用append模式
    _is_append = conf.is_append();
    _n_batch_size = 0;
    _fwd_his_o.clear();
    _bwd_his_o.clear();
}
```

Bilstm的forward

```
/*
 *双向lstm
 * */
void BiLstmLayer::forward(InOutput &in_out, IN_OUT_TYPE_T i_type) {
  int height = (int)in_out.get_height();
  int width = (int)in_out.get_in_width();
  in_out.trans_in(i_type, INOUT_TYPE);
  FMatrix *in = in_out.f_in();

  // normal-forward
  _in_out.set_input(in);
  _lstm_layer->set_weights(_fwd_weight);
  _lstm_layer->set_history(_fwd_his_o);  //++++++++++++++
  _lstm_layer->forward(_in_out, i_type);
  _lstm_layer->store_current_out();
  _lstm_layer->get_history(_fwd_his_o);  //++++++++++++++

  _in_out.clear_input(INOUT_TYPE);
```

正向执行

Bilstm的forward

```
// inverse-forward
_invert_in_out.resize_in(height, width, INOUT_TYPE);
FMatrix *inverIn = _invert_in_out.f_in();

//把输入顺序进行反转
for (int ii = 0; ii < height; ii++) {
  int pos = _map_id_vec1.get_value(ii);
  inverIn->range_row(pos, pos + 1).copy_from(in->range_row(ii, ii + 1, 1));
}
_lstm_layer->set_weights(_bwd_weight);
_lstm_layer->set_history(_bwd_his_o);
_lstm_layer->forward(_invert_in_out, INOUT_TYPE);
_lstm_layer->store_current_out();
_lstm_layer->get_history(_bwd_his_o);  //++++++++++++++

// translate the tow outputs
_in_out.trans_out(_lstm_layer->o_type(), INOUT_TYPE);

_invert_in_out.trans_out(_lstm_layer->o_type(), INOUT_TYPE);
FMatrix *inter_out = _in_out.f_out();
FMatrix *inver_out = _invert_in_out.f_out();
in_out.trans_out(_lstm_layer->o_type(), INOUT_TYPE);
FMatrix *out = in_out.f_out();
```

输入反转，反向lstm执行

Bilstm执行

```
    // according to the requre to combine the output
    if (!_is_append) {
      out->resize(inter_out->get_height(), inter_out->get_width());
      out->copy_from(*inter_out);
      for (int ii = 0; ii < height; ii++) {
        int pos = _map_id_vec2.get_value(ii);
        out->range_row(pos, pos + 1).add(inver_out->range_row(ii, ii + 1));
      }
    } else {
      CHECK(inter_out->get_width() == inver_out->get_width(), "Not Matched");
      width = (int)inter_out->get_width();

      out->resize(height, 2 * width);
      for (int ii = 0; ii < height; ii++) {
        int pos = _map_id_vec2.get_value(ii);
        out->range_row(ii, ii + 1)
            .range_col(0, width)
            .copy_from(inter_out->range_row(ii, ii + 1));
        out->range_row(pos, pos + 1)
            .range_col(width, 2 * width)
            .copy_from(inver_out->range_row(ii, ii + 1));
      }
    }
    // layer activation
    _act->forward(*in_out.f_out(), *in_out.f_out());
    in_out.trans_out(INOUT_TYPE, _o_type);
}
```

结果拼合

激活

本周作业

- 个性化合成声纹模型训练和ONNX导出。

参考:

项目地址：https://github.com/CorentinJ/Real-Time-Voice-Cloning.git

encoder_preprocess.py： 执行wav文件预处理成系统指定格式。

encoder_train.py: 执行声纹提取的训练

模型导出：https://pytorch.org/docs/stable/onnx.html

签到 & 反馈