Model on Speech Recognition

Todo List

1. Chiu, Chung-Cheng, et al. "State-of-the-art speech recognition with sequence-to-sequence models." *2018 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. IEEE, 2018.
2. Zou, Wei, et al. "Comparable study of modeling units for end-to-end mandarin speech recognition." *2018 11th International Symposium on Chinese Spoken Language Processing (ISCSLP)*. IEEE, 2018.
3. Park, Daniel S., et al. "Specaugment: A simple data augmentation method for automatic speech recognition." *arXiv preprint arXiv:1904.08779* (2019).
4. Hannun, Awni, et al. "Deep speech: Scaling up end-to-end speech recognition." *arXiv preprint arXiv:1412.5567* (2014).
5. Amodei, Dario, et al. "Deep speech 2: End-to-end speech recognition in english and mandarin." *International conference on machine learning*. 2016.
6. Battenberg, Eric, et al. "Exploring neural transducers for end-to-end speech recognition." *2017 IEEE Automatic Speech Recognition and Understanding Workshop (ASRU)*. IEEE, 2017.

7. T. N. Sainath and C. Parada. Convolutional neural networks for small-footprint keyword spotting. In Sixteenth Annual Conference of the International Speech Communication Association, 2015
8. tensorflow优化器源码

Optimizer in Deep Learning

梯度下降法（Gradient Descent）

梯度下降法的计算过程就是沿梯度下降的方向求解极小值，也可以沿梯度上升方向求解最大值。使用梯度下降法更新参数：

$$\theta_{t+1} = \theta_t - \alpha \cdot \nabla_\theta J(\theta)$$

批量梯度下降法（BGD）

在整个训练集上计算梯度，对参数进行更新：

$$\theta_{t+1} = \theta_t - \alpha \cdot \frac{1}{n} \cdot \sum_{i=1}^{n} \nabla_\theta J_i(\theta, x^i, y^i)$$

因为要计算整个数据集，收敛速度慢，但其优点在于更趋近于全局最优解；

随机梯度下降法（SGD）

每次只随机选择一个样本计算梯度，对参数进行更新：

$$\theta_{t+1} = \theta_t - \alpha \cdot \nabla_\theta J_i(\theta, x^i, y^i)$$

训练速度快，但是容易陷入局部最优点，导致梯度下降的波动非常大；

小批量梯度下降法（Mini-batch Gradient Descent）

每次随机选择 n 个样本计算梯度，对参数进行更新：

$$\theta_{t+1} = \theta_t - \alpha \cdot \frac{1}{m} \cdot \sum_{i=x}^{i=x+m-1} \cdot \nabla_\theta J_i(\theta, x^i, y^i)$$

这种方法是 BGD 和 SGD 的折衷；

动量优化法

Momentum

参数更新时在一定程度上保留之前更新的方向，同时又利用当前 batch 的梯度微调最终的更新方向。在 SGD 的基础上增加动量，则参数更新公式如下：

$$m_{t+1} = \mu \cdot m_t + \alpha \cdot \nabla_\theta J(\theta)$$
$$\theta_{t+1} = \theta_t - m_{t+1}$$

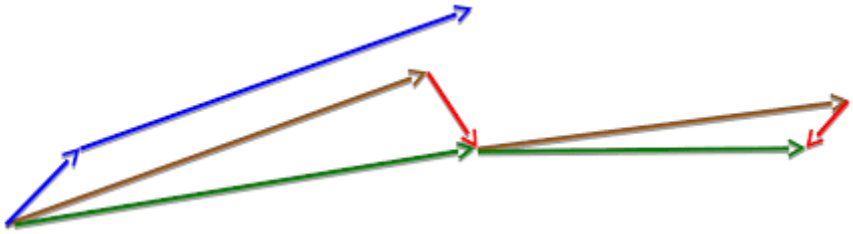
在梯度方向发生改变时，Momentum 能够降低参数更新速度，从而减少震荡；在梯度方向相同时，Momentum 可以加速参数更新，从而加速收敛。

NAG（Nesterov Accelerated Gradient）

与 Momentum 不同的是，NAG 是在更新梯度是做一个矫正，即提前往前探一步，并对梯度作修改。参数更新公式如下：

$$
\begin{aligned}
m_{t+1} &= \mu \cdot m_t + \alpha \cdot \nabla_\theta J(\theta - \mu \cdot m_t) \\
\theta_{t+1} &= \theta_t - m_{t+1}
\end{aligned}
$$

两者的对比：蓝色为 Momentum， 剩下的是 NAG；



自适应学习率

AdaGrad

AdaGrad 算法期望在模型训练时有一个较大的学习率，而随着训练的不断增多，学习率也跟着下降。参数更新公式如下：

$$
\begin{aligned}
g &\leftarrow \nabla_\theta J(\theta) \\
r &\leftarrow r + g^2 \\
\Delta\theta &\leftarrow \frac{\delta}{\sqrt{r+\epsilon}} \cdot g \\
\theta &\leftarrow \theta - \Delta\theta
\end{aligned}
$$

学习率随着 **梯度的平方和**（r）的增加而减少. 缺点：

- 需要手动设置学习率 δ ，如果 δ 过大，会使得正则化项 $\frac{\delta}{\sqrt{r+\epsilon}}$ 对梯度的调节过大；
- 中后期，参数的更新量会趋近于 0，使得模型无法学习；

Adadelta

Adadelta 算法将 梯度的平方和 改为 **梯度平方的加权平均值。**参数更新公式如下：

$$
\begin{aligned}
g_t &\leftarrow \nabla_\theta J(\theta) \\
E[g^2]_t &\leftarrow \gamma \cdot E[g^2]_{t-1} + (1-\gamma) \cdot g_t^2 \\
\Delta\theta_t &\leftarrow -\frac{\eta}{\sqrt{E[g^2]_t + \epsilon}} \cdot g_t \\
\theta_{t+1} &\leftarrow \theta_t + \Delta\theta_t
\end{aligned}
$$

上式中仍存在一个需要自己设置的全局学习率 η，可以通过下式**消除全局学习率的影响**：

$$
\begin{aligned}
E[\Delta\theta^2]_t &\leftarrow \gamma \cdot E[\Delta\theta^2]_{t-1} + (1-\gamma) \cdot \Delta\theta_t^2 \\
\Delta\theta_t &\leftarrow -\frac{\sqrt{E[\Delta\theta^2]_{t-1} + \epsilon}}{\sqrt{E[g^2]_t + \epsilon}} \cdot g_t \\
\theta_{t+1} &\leftarrow \theta_t + \Delta\theta_t
\end{aligned}
$$

RMSProp

RMSProp 算法是 AdaGrad 算法的改进，修改 梯度平方和 为 **梯度平方的指数加权移动平均**，解决了学习率急剧下降的问题。参数更新公式如下：

$$
\begin{aligned}
g_t &\leftarrow \nabla_\theta J(\theta) \\
E[g^2]_t &\leftarrow \rho \cdot E[g^2]_{t-1} + (1-\rho) \cdot g_t^2 \\
\Delta\theta &\leftarrow \frac{\delta}{\sqrt{E[g^2]_t + \epsilon}} \cdot g \\
\theta &\leftarrow \theta - \Delta\theta
\end{aligned}
$$

Adam（Adaptive Moment Estimation）

Adam 算法在动量的基础上，结合了偏置修正。参数更新公式如下：

$$
\begin{aligned}
g_t &\leftarrow \Delta_\theta J(\theta) \\
m_t &\leftarrow \beta_1 \cdot m_{t-1} + (1-\beta_1) \cdot g_t \\
v_t &\leftarrow \beta_2 \cdot v_{t-1} + (1-\beta_2) \cdot g_t^2 \\
\hat{m}_t &\leftarrow \frac{m_t}{1-(\beta_1)^t} \\
\hat{v}_t &\leftarrow \frac{v_t}{1-(\beta_2)^t} \\
\theta_{t+1} &= \theta_t - \frac{\delta}{\sqrt{\hat{v}_t} + \epsilon} \cdot \hat{m}_t
\end{aligned}
$$

论文伪代码：

Algorithm 1: *Adam*, our proposed algorithm for stochastic optimization. See section 2 for details, and for a slightly more efficient (but less clear) order of computation. g_t^2 indicates the elementwise square $g_t \odot g_t$. Good default settings for the tested machine learning problems are $\alpha = 0.001$, $\beta_1 = 0.9$, $\beta_2 = 0.999$ and $\epsilon = 10^{-8}$. All operations on vectors are element-wise. With β_1^t and β_2^t we denote β_1 and β_2 to the power t.

Require: α: Stepsize
Require: $\beta_1, \beta_2 \in [0, 1)$: Exponential decay rates for the moment estimates
Require: $f(\theta)$: Stochastic objective function with parameters θ
Require: θ_0: Initial parameter vector
  $m_0 \leftarrow 0$ (Initialize 1st moment vector)
  $v_0 \leftarrow 0$ (Initialize 2nd moment vector)
  $t \leftarrow 0$ (Initialize timestep)
  while θ_t not converged **do**
    $t \leftarrow t + 1$
    $g_t \leftarrow \nabla_\theta f_t(\theta_{t-1})$ (Get gradients w.r.t. stochastic objective at timestep t)
    $m_t \leftarrow \beta_1 \cdot m_{t-1} + (1-\beta_1) \cdot g_t$ (Update biased first moment estimate)
    $v_t \leftarrow \beta_2 \cdot v_{t-1} + (1-\beta_2) \cdot g_t^2$ (Update biased second raw moment estimate)
    $\widehat{m}_t \leftarrow m_t / (1-\beta_1^t)$ (Compute bias-corrected first moment estimate)
    $\widehat{v}_t \leftarrow v_t / (1-\beta_2^t)$ (Compute bias-corrected second raw moment estimate)
    $\theta_t \leftarrow \theta_{t-1} - \alpha \cdot \widehat{m}_t / (\sqrt{\widehat{v}_t} + \epsilon)$ (Update parameters)
  end while
  return θ_t (Resulting parameters)

Links

- 参考链接：优化算法Optimizer比较和总结
- Adam 论文链接：Kingma D P, Ba J. Adam: A method for stochastic optimization[J]. arXiv preprint arXiv:1412.6980, 2014.

Deep speech: Scaling up end-to-end speech recognition

Notes

代码阅读

Links

- 论文链接：Hannun A, Case C, Casper J, et al. Deep speech: Scaling up end-to-end speech recognition[J]. arXiv preprint arXiv:1412.5567, 2014.
-

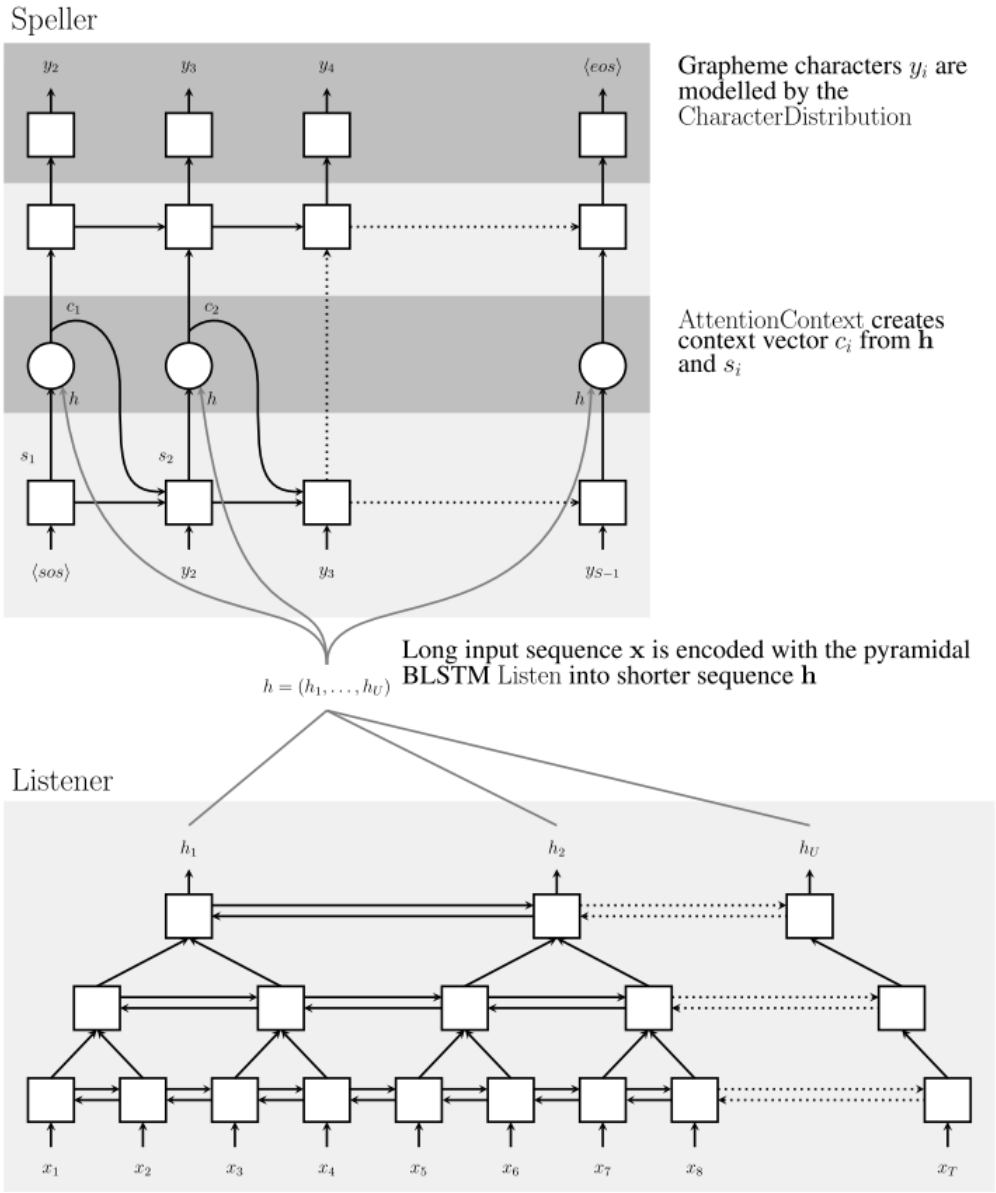
Listen, Attend and Spell

Contribution

1. 提出了一种**新的端到端**的语音识别模型；

Notes

1. **模型架构**：分成两个模块，一个是 Listen Encoder 模块，从语音时序序列中提取出高维特征，采用 pBLSTM (pyramid BLSTM) 的架构；另一个是 Attend and Spell 模块，从语音高维特征中输出单词，采用 Attention + LSTM 架构。架构图如下：（直接从这个图看的话，感觉模型是比较简洁的，但是真的从代码层面再来画一个细粒度的架构图，其实复杂得多。）



2. **Listen Encoder 模块**，使用 pBLSTM 的架构，每层在时间维度上减少一倍，带来的优点有两个：

   (1) 减少模型的复杂性（一定程度上是比较合理的，因为语音的前后帧之间有非常多的冗余信息）；

   (2) 加快模型的拟合速度（作者发现直接用 BLSTM 的话，用一个月的时间训练都没有办法得到好的结果）；

   形式化的公式为：（和代码结合来看：关注公式 $[h_{2i}^{j-1}, h_{2i+1}^{j-1}]$ 部分，在程序的实现中，首先通过 设置 LSTM的特征维度 将**特征逐层降维**，然后通过合并前后帧的特征 对**时间降维**（而特征维度则升高））

$$h_i^j = \text{pBLSTM}(h_{i-1}^j, \left[h_{2i}^{j-1}, h_{2i+1}^{j-1}\right])$$

3. **Attend and Spell 模块**，该模块采用 2 层 LSTM 单元来记忆、更新模型的状态 s (**模型的状态包括 LSTM 的状态和 Attention 上下文状态**)：

$$\begin{aligned} c_i &= \text{AttentionContext}(s_i, \mathbf{h}) \\ s_i &= \text{RNN}(s_{i-1}, y_{i-1}, c_{i-1}) \\ P(y_i|\mathbf{x}, y_{<i}) &= \text{CharacterDistribution}(s_i, c_i) \end{aligned}$$

   (1) Attention 单元：根据当前的状态 s_i （**在代码中，s_i 指的是第二层 LSTM 单元的输出**）从语音特征 h 中分离出“当前模型关心的”上下文信息 c_i；

(2) LSTM 单元：根据前一时刻的状态 s_{i-1} （**在代码中，这个 s_{i-1} 指的是第二层 LSTM 单元的状态**）、前一时刻输出的字符 y_{i-1} 和前一时刻的上下文信息 c_{i-1} 来更新产生当前时刻的状态 s_i；

(3) MLP 单元：根据当前状态 s_i 和上下文信息 c_i 计算得到最可能的字符 y_i；

另外， **Attention 单元在模型中的具体实现**：将模型状态 s_i 和语音特征 h 分别经过两个不同的 MLP 模型，计算出一个标量能量 (点积) e ，经过 softmax 层归一化后作为权重向量，和原来的特征 h 加权生成上下文信息 c_i。形式化的公式如下：

$$
\begin{aligned}
e_{i,u} &= \langle \phi(s_i), \psi(h_u) \rangle \\
\alpha_{i,u} &= \frac{\exp(e_{i,u})}{\sum_u \exp(e_{i,u})} \\
c_i &= \sum_u \alpha_{i,u} h_u
\end{aligned}
$$

4. **Learning**. 模型的目标是，在给定 **全部** 语音信号和 **上文** 解码结果的情况下，模型输出正确字符的概率最大。形式化的公式如下：

$$
\max_\theta \sum_i \log P(y_i | \mathbf{x}, y^*_{<i}; \theta)
$$

在训练的时候，我们给的 y 都是 ground truth，但是解码的时候，模型不一定每个时间片都会产生正确的标签。虽然模型对于这种错误是具有宽容度，单训练的时候可以增加 **trick**：以 **10%** 的概率从前一个解码结果中挑选 (根据前一次的概率分布) 一个标签作为 ground truth 进行训练。形式化公式如下：

$$
\begin{aligned}
\tilde{y}_i &\sim \mathrm{CharacterDistribution}(s_i, c_i) \\
\max_\theta \sum_i \log P(y_i | \mathbf{x}, \tilde{y}_{<i}; \theta)
\end{aligned}
$$

另外，作者发现预训练 (主要是预训练 Listen Encoder 部分) 对 LAS 模型没有作用。

5. **Decoding & Rescoring**. 解码的时候使用 Beam-Search 算法，目标是希望得到概率最大的字符串。形式化公式如下：

$$
\hat{\mathbf{y}} = \arg\max_{\mathbf{y}} \log P(\mathbf{y} | \mathbf{x})
$$

可以用语言模型对最后一轮 Beam-Search 的结果进行重打分，形式化公式如下：

$$
s(\mathbf{y}|\mathbf{x}) = \frac{\log P(\mathbf{y}|\mathbf{x})}{|\mathbf{y}|_c} + \lambda \log P_{\mathrm{LM}}(\mathbf{y})
$$

增加解码结果的长度项 |y| 来**平衡产生长句、短句的权重**，另外语言模型的权重 lambda 可以通过验证集数据来确定。

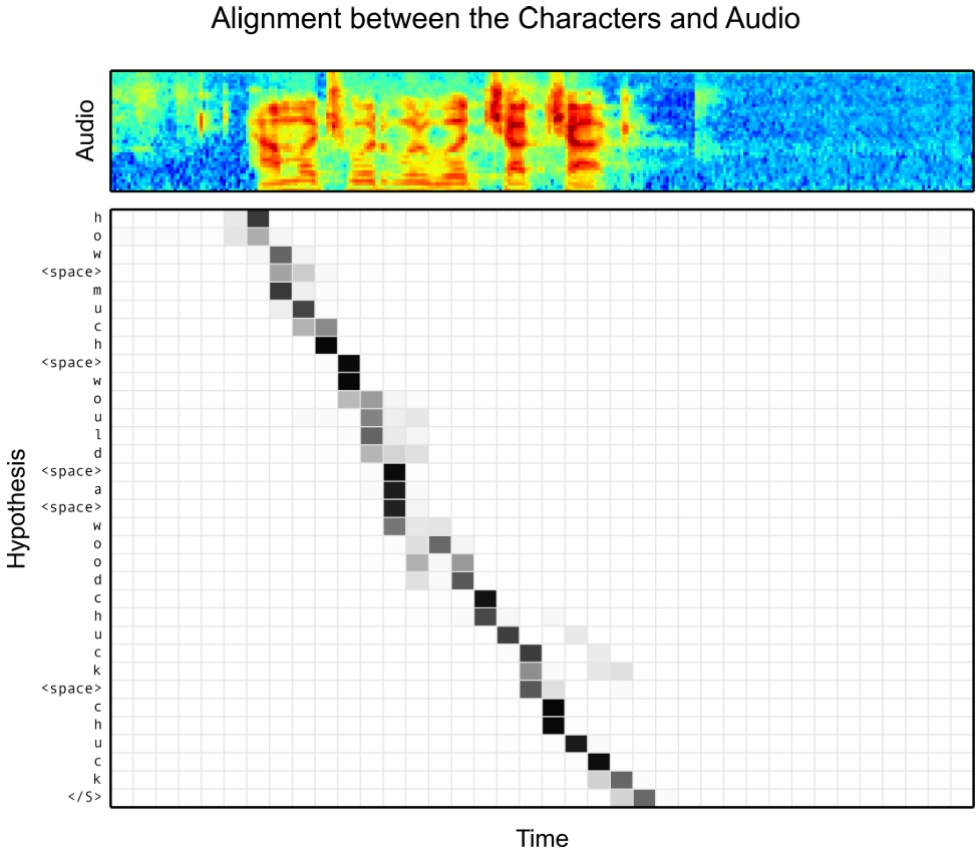
6. 实验结果：

(1) 使用 log-mel filter bank 特征

(2) 整体对比，LAS 刚出来的时候并打不过传统的 DNN-HMM 模型；

Model	Clean WER	Noisy WER
CLDNN-HMM [20]	8.0	8.9
LAS	16.2	19.0
LAS + LM Rescoring	12.6	14.7
LAS + Sampling	14.1	16.5
LAS + Sampling + LM Rescoring	10.3	12.0

(3) Attention 模块确实更加关注对应时间片段的特征；



(4) 模型对于较短的语句或者较长的语句效果都不是很好；

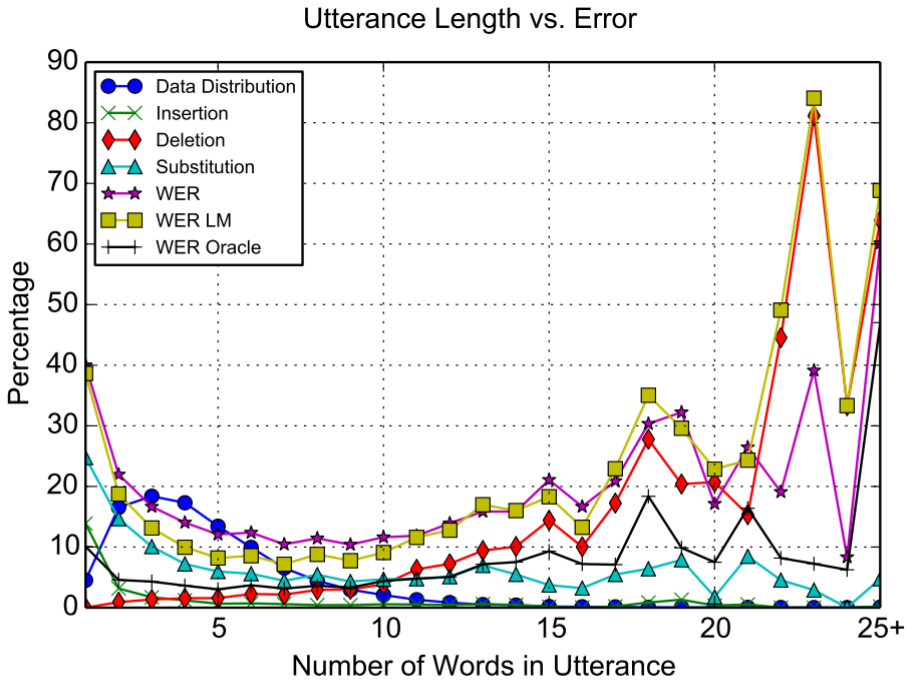


Figure 4: The correlation between error rates (insertion, deletion, substitution and WER) and the number of words in an utterance. The WER is reported without a dictionary or language model, with language model rescoring and the oracle WER for the clean Google voice search task. The data distribution with respect to the number of words in an utterance is overlaid in the figure. LAS performs poorly with short utterances despite an abundance of data. LAS also fails to generalize well on longer utterances when trained on a distribution of shorter utterances. Insertions and substitutions are the main sources of errors for short utterances, while deletions dominate the error for long utterances.

Shortcoming

1. 必须要得到整个语音后才能解码，限制了模型的流式处理能力；
2. Attention 机制需要消耗大量的计算量；
3. 输入长度对于模型的影响较大；

代码理解

Tensorflow 2 （Keras）实现

> 这个库只实现了 LAS 模型部分，没有完整的预处理等过程，故先通过这个库来简单学习下 LAS 模型原理以及 Tensorflow 的使用，期待一下库作者的更新；

(1) 整体框架：

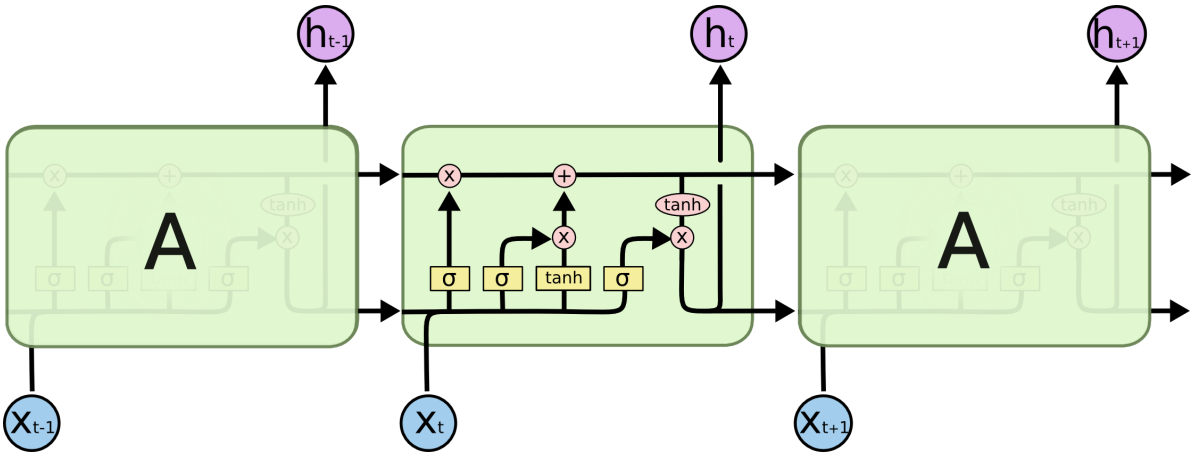
```
def LAS(dim, f_1, no_tokens): # dim-神经网络内特征维度，f_1-输入特征维度，no_tokens-分类维度
  input_1 = tf.keras.Input(shape=(None, f_1))  # shape: (..., None, f_1)
  input_2 = tf.keras.Input(shape=(None, no_tokens))  # shape: (..., None, no_tokens)

  #Listen; Lower resoultion by 8x
  x = pBLSTM( dim//2 )(input_1) # (..., audio_len//2, dim*2)
  x = pBLSTM( dim//2 )(x) # (..., audio_len//4, dim*2)
  x = pBLSTM( dim//4 )(x) # (..., audio_len//8, dim)

  #Attend
  x = tf.keras.layers.RNN(att_rnn(dim), return_sequences=True)(input_2, constants=x) # (..., seq_len, dim*2)

  #Spell
  x = tf.keras.layers.Dense(dim, activation="relu")(x) # (..., seq_len, dim)
  x = tf.keras.layers.Dense(no_tokens, activation="softmax")(x) # (..., seq_len, no_tokens)

  model = tf.keras.Model(inputs=[input_1, input_2], outputs=x)
  return model
```

(2) Listen 模块：使用 3 层 `pBLSTM` 实现，其中需要注意的是 `tf.keras.layers.Bidirectional` 的使用（我一开始判断错了输出的维度）

```
class pBLSTM(tf.keras.layers.Layer):
  def __init__(self, dim):
    super(pBLSTM, self).__init__()

    self.dim        = dim
    self.LSTM       = tf.keras.layers.LSTM(self.dim, return_sequences=True)
    self.bidi_LSTM  = tf.keras.layers.Bidirectional(self.LSTM)

  @tf.function
  def call(self, inputs):
    y = self.bidi_LSTM(inputs) # (..., seq_len, dim*2)

    if tf.shape(inputs)[1] % 2 == 1:
      y = tf.keras.layers.ZeroPadding1D(padding=(0, 1))(y)

    y = tf.keras.layers.Reshape(target_shape=(-1, int(self.dim*4)))(y) # (..., seq_len//2, dim*4)
    return y
```

(3) Attend 模块：

- 如果对 LSTM 不太熟悉的话，结合 LSTM 的结构图一起来看代码会轻松一点：



- 双层 LSTM 代码如下：使用 2 层 LSTM 模型来存储模型的状态；

```
class att_rnn( tf.keras.layers.Layer):
  def __init__(self, units,):
    super(att_rnn, self).__init__()
    self.units      = units
    self.state_size = [self.units, self.units]

    self.attention_context  = attention(self.units)
    self.rnn                = tf.keras.layers.LSTMCell(self.units)  # LSTM 1，用来记忆模型的状态
    self.rnn2               = tf.keras.layers.LSTMCell(self.units)  # LSTM 2，用来记忆模型的状态

  def call(self, inputs, states, constants):
    h       = tf.squeeze(constants, axis=0)  # 删除为1的维度，shape: (..., seq_len, F)

    s       = self.rnn(inputs=inputs, states=states) # [(..., F), [(..., F), (..., F)]]
    s       = self.rnn2(inputs=s[0], states=s[1])[1] # [(..., F), (..., F)]

    c       = self.attention_context([s[0], h]) # (..., F)
    out     = tf.keras.layers.concatenate([s[0], c], axis=-1) # (..., F*2)

    return out, [c, s[1]]
```

- Attention 代码如下：全连接层（变换维度） -> 向量点积（计算权重） -> softmax（权重归一化） -> 得到重要的上下文信息；

```
class attention(tf.keras.layers.Layer):  # Attention 类，用来计算上下文的权重
  def __init__(self, dim):
    super(attention, self).__init__()

    self.dim      = dim
    self.dense_s  = tf.keras.layers.Dense(self.dim)
    self.dense_h  = tf.keras.layers.Dense(self.dim)

  def call(self, inputs):
    # Split inputs into attentions vectors and inputs from the LSTM output
    s     = inputs[0] # (..., depth_s)
    h     = inputs[1] # (..., seq_len, depth_h)
```

```
    # Linear FC
    s_fi   = self.dense_s(s) # (..., F)
    h_psi  = self.dense_h(h) # (..., seq_len, F)

    # Linear blendning < ϕ(s_i), ψ(h_u) >
    # Forced seq_len of 1 since s should always be a single vector per batch
    e = tf.matmul(s_fi, h_psi, transpose_b=True) # (..., 1, seq_len)

    # Softmax vector
    alpha = tf.nn.softmax(e) # (..., 1, seq_len)

    # Context vector
    c = tf.matmul(alpha, h) # (..., 1, depth_h)
    c = tf.squeeze(c, 1) # (..., depth_h)

    return c
```

(4) Spell 模块：两个全连接层，输出最后的概率；

```
x = tf.keras.layers.Dense(dim, activation="relu")(x) # (..., seq_len, dim)
x = tf.keras.layers.Dense(no_tokens, activation="softmax")(x) # (...,
seq_len, no_tokens)
```

Links

- 论文链接：Listen, Attend and Spell
- LAS 模型缺点参考链接：LAS 语音识别框架发展简述
- Tensorflow 2 （Keras）实现：Listen, attend and spell
- Pytorch 实现：End-to-end-ASR-Pytorch (**暂未阅读代码**)
- LSTM 详解：Understanding LSTM Networks

Lingvo: a modular and scalable framework for sequence-to-sequence modeling

谷歌开源的基于tensorflow的序列模型框架。

Notes

Links

- 论文链接：Lingvo: a modular and scalable framework for sequence-to-sequence modeling
- Github：Lingvo