人工神经网络实验报告

作业 4 • RNN

计 54，周正平，2015011314，zhouzp15@mails.tsinghua.edu.cn

2017 年 11 月 18 日

Contents

1 实验内容

本次实验利用 TensorFlow 实现基础版 RNN、GRU、LSTM，用于完成 Stanford Sentiment Treebank（SST）句子情感分类任务。

总体来说，本次作业需要在给出的框架基础上，完成以下实验：

1. **词向量导入**：在 `main.py`中导入预训练的词向量；
2. **模型搭建**：在 `model.py`中实现 `placeholder`等，搭建基于 RNN 的神经网络；
3. **基本单元**：在 `cell.py`中实现 BasicRNNCell, GRUCell, BasicLSTMCell 等基础单元；
4. **模型可视化**：在 `main.py`中加入 TensorBoard 可视化代码。

2 算法实现

2.1 词向量导入

词向量将每个单词映射到一个固定维度（实验中取 300 维），得到每个单词的分布式表示，有助于提取词语的语义信息及共性特征。

OOV 问题 需注意的是，实验中给出的 `vector.txt`只包含 17530 个词，但语料中的词语数目超过了 18000。为此，笔者将语料中其余单词的词向量全 0 初始化：

```
for vocab in vocab_list:
    if vocab in embed_dict:
        embed.append(embed_dict[vocab])
    else:
        embed.append([0.0] * FLAGS.embed_units)
```

2.2 模型搭建

数据读入 在 `model.py`中，需首先实现数据入口处的 `placeholder`。由于 `batch_size, text_length`均为可变参数，故而 `placeholder`的各个维度均为 `None`。

模型建立 以单层的基础版 RNN 为例，需首先新建一个 RNN 单元（Cell），再使用 `tf.nn.dynamic_rnn`得到中间及最终状态向量，并接入全连接层得到最终输出。

与 `tf.nn.static_rnn`相对地，`tf.nn.dynamic_rnn`允许各个 batch 之间的最大句子长度不一样，从而节省了 padding 带来的空间浪费。

```
if num_layers == 1:
    cell = BasicRNNCell(num_units)

outputs, state = dynamic_rnn(cell, self.embed_input, self.texts_length, dtype=tf.float32, scope="rnn")
logits = tf.layers.dense(inputs=state, units=num_labels)
```

2.3 基本单元

基本单元（Cell）为 RNN 模型的核心，以下进行详细说明。

2.3.1 BasicRNNCell

算法原理 BasicRNNCell 为最基础的一类 Cell，从输入到输出，实现的仅为最简单的线性变换，并加以激活，无门机制控制记忆的写入与遗忘：

$$h_t = \tanh([h_{t-1}, x_t] \cdot W + b)$$

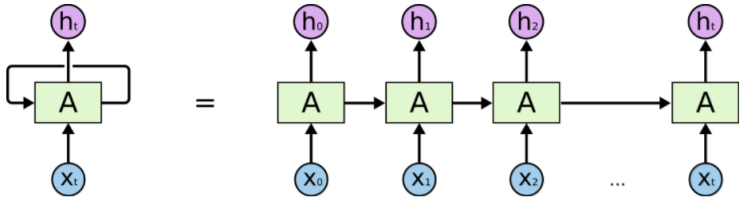


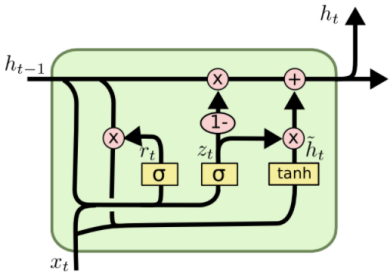
Figure 1: BasicRNNCell

实现方法 实现时，只需模拟上述公式即可：

```
def __call__(self, inputs, state, scope=None):
    with tf.variable_scope(scope or "basic_rnn_cell", reuse=self._reuse):
        W = tf.get_variable('W', [FLAGS.embed_units + self._num_units, self._num_units])
        b = tf.get_variable('b', [self._num_units], initializer=tf.constant_initializer(0.0))
        new_state = self._activation(tf.matmul(tf.concat([inputs, state], axis=1), W) + b)

    return new_state, new_state
```

2.3.2 GRUCell

算法原理 GRUCell 可以看做 LSTM 的简化版，采用了较为简单的门机制，用以控制记忆的遗忘与写入。



$$z_t = \sigma\left(W_z \cdot [h_{t-1}, x_t]\right)$$
$$r_t = \sigma\left(W_r \cdot [h_{t-1}, x_t]\right)$$
$$\tilde{h}_t = \tanh\left(W \cdot [r_t * h_{t-1}, x_t]\right)$$
$$h_t = (1 - z_t) * h_{t-1} + z_t * \tilde{h}_t$$

Figure 2: GRUCell

其中各个变量含义如下：

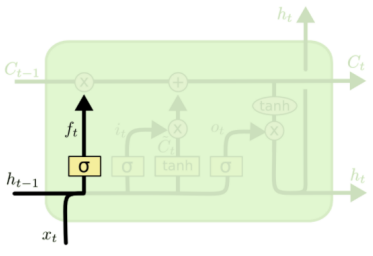
变量	形状	含义
x_t	$[batch\_size \times embed\_units]$	当前时刻的输入
z_t	$[batch\_size \times num\_units]$	update 门，候选状态对新状态的影响
r_t	$[batch\_size \times num\_units]$	reset 门，旧状态对候选状态的影响
W_z, b_z	$[(embed\_units + num\_units) \times num\_units]$, $[num\_units]$	update 门的变换矩阵、偏置
W_r, b_r	$[(embed\_units + num\_units) \times num\_units]$, $[num\_units]$	reset 门的变换矩阵、偏置
W, b	$[(embed\_units + num\_units) \times num\_units]$, $[num\_units]$	从旧状态到候选状态的变换矩阵、偏置
$\tilde{h_t}$	$[batch\_size \times num\_units]$	候选状态
h_t	$[batch\_size \times num\_units]$	产生的新状态

实现方法 实现时，只需模拟上述公式即可。需注意这里将 bias 全 1 初始化，以便在最开始既不 update，也不 reset：

```
def __call__(self, inputs, state, scope=None):
    with tf.variable_scope(scope or "gru_cell", reuse=self._reuse):
        W_gate = tf.get_variable('W_gate', [FLAGS.embed_units + self._num_units, self._num_units * 2])
        b_gate = tf.get_variable('b_gate', [self._num_units * 2], initializer=tf.constant_initializer
            (1.0))

        gates = tf.sigmoid(tf.matmul(tf.concat([inputs, state], axis=1), W_gate) + b_gate)
        z, r = tf.split(gates, num_or_size_splits=2, axis=1)

        W_cand = tf.get_variable('W_cand', [FLAGS.embed_units + self._num_units, self._num_units])
        b_cand = tf.get_variable('b_cand', [self._num_units], initializer=tf.constant_initializer(0.0))
        cand_h = self._activation(tf.matmul(tf.concat([inputs, r * state], axis=1), W_cand) + b_cand)

        new_h = z * state + (1 - z) * cand_h

    return new_h, new_h
```

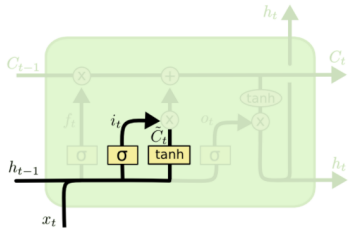
2.3.3 BasicLSTMCell

算法原理 BasicLSTMCell 是 LSTM 的基本组成单元，其中使用精细的门机制控制记忆的写入与遗忘，从而在处理文本逻辑、长文本中发挥着突出的作用。



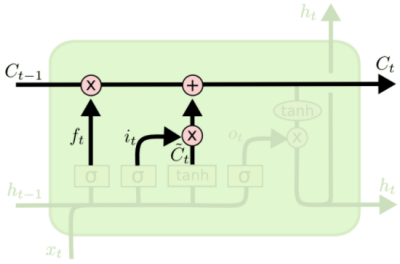
$$f_t = \sigma\left(W_f \cdot [h_{t-1}, x_t] \; + \; b_f\right)$$

Figure 3: LSTM 遗忘门



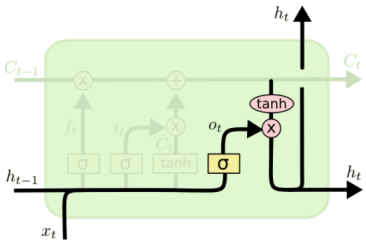
$$i_t = \sigma\left(W_i \cdot [h_{t-1}, x_t] \ + \ b_i\right)$$
$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] \ + \ b_C)$$

Figure 4: LSTM 输入门



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

Figure 5: LSTM Cell 更新



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

Figure 6: LSTM 输出门

其中各个变量含义如下：

变量	形状	含义
x_t	$[batch\_size \times embed\_units]$	当前时刻的输入
f_t	$[batch\_size \times num\_units]$	forget 门，用于控制记忆的遗忘
i_t	$[batch\_size \times num\_units]$	input 门，候选状态对新状态的影响
o_t	$[batch\_size \times num\_units]$	output 门，新状态对隐藏状态的影响
W_f, b_f	$[(embed\_units + num\_units) \times num\_units]$, $[num\_units]$	forget 门的变换矩阵、偏置
W_i, b_i	$[(embed\_units + num\_units) \times num\_units]$, $[num\_units]$	input 门的变换矩阵、偏置
W_o, b_o	$[(embed\_units + num\_units) \times num\_units]$, $[num\_units]$	output 门的变换矩阵、偏置
W_C, b_C	$[(embed\_units + num\_units) \times num\_units]$, $[num\_units]$	从旧状态到候选状态的变换矩阵、偏置
$\tilde{C}_t$	$[batch\_size \times num\_units]$	候选状态
C_t	$[batch\_size \times num\_units]$	产生的新状态
h_t	$[batch\_size \times num\_units]$	产生的新隐藏状态

实现方法 实现时，只需模拟上述公式即可。需注意这里将 forget 门的 bias 全 1 初始化，以便在最开始时减少遗忘：

```
  def __call__(self, inputs, state, scope=None):
     with tf.device('/gpu:0'):
        with tf.variable_scope(scope or "basic_lstm_cell", reuse=self._reuse):
           c, h = state
           W_f = tf.get_variable('W_f', [FLAGS.embed_units + self._num_units, self._num_units])
           b_f = tf.get_variable('b_f', [self._num_units], initializer=tf.constant_initializer(self.
               _forget_bias))

           W_i = tf.get_variable('W_i', [FLAGS.embed_units + self._num_units, self._num_units])
           b_i = tf.get_variable('b_i', [self._num_units], initializer=tf.constant_initializer(0.0))

           W_o = tf.get_variable('W_o', [FLAGS.embed_units + self._num_units, self._num_units])
           b_o = tf.get_variable('b_o', [self._num_units], initializer=tf.constant_initializer(0.0))

           W_c = tf.get_variable('W_c', [FLAGS.embed_units + self._num_units, self._num_units])
           b_c = tf.get_variable('b_c', [self._num_units], initializer=tf.constant_initializer(0.0))

           f = tf.sigmoid(tf.matmul(tf.concat([inputs, h], axis=1), W_f) + b_f)
           i = tf.sigmoid(tf.matmul(tf.concat([inputs, h], axis=1), W_i) + b_i)
           o = tf.sigmoid(tf.matmul(tf.concat([inputs, h], axis=1), W_o) + b_o)

           cand_c = self._activation(tf.matmul(tf.concat([inputs, h], axis=1), W_c) + b_c)
           new_c = f * c + i * cand_c
           new_h = o * self._activation(new_c)

     return new_h, (new_c, new_h)
```

2.4 最终模型

笔者在以上实验的基础上，实现了双向 LSTM：

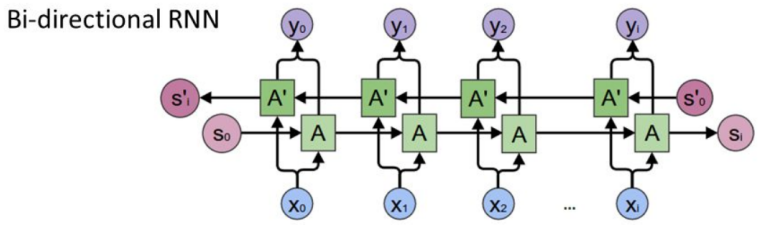


Figure 7: Bi-LSTM

代码实现如下：

```
cell_fw = BasicLSTMCell(num_units)
cell_bw = BasicLSTMCell(num_units)
outputs, states, = tf.nn.bidirectional_dynamic_rnn(cell_fw, cell_bw, self.embed_input, dtype=tf.float32
    , scope='rnn')

(cell_fw, hidden_fw), (cell_bw, hidden_bw) = states
logits = tf.layers.dense(inputs=tf.concat([hidden_fw, hidden_bw], 1), units=num_labels)
```

3 实验结果

3.1 单层 RNN

3.1.1 BasicRNNCell

笔者实现的基础版单层 RNN，在测试集上的精度为 42.35%。
网络超参数配置如下：

超参数	取值	备注
embed_units	300	词向量维数
units	512	RNN 隐藏状态节点数
batch_size	16	

使用 TensorBoard 绘制训练集、开发集、测试集上的 loss-epoch 曲线如下：

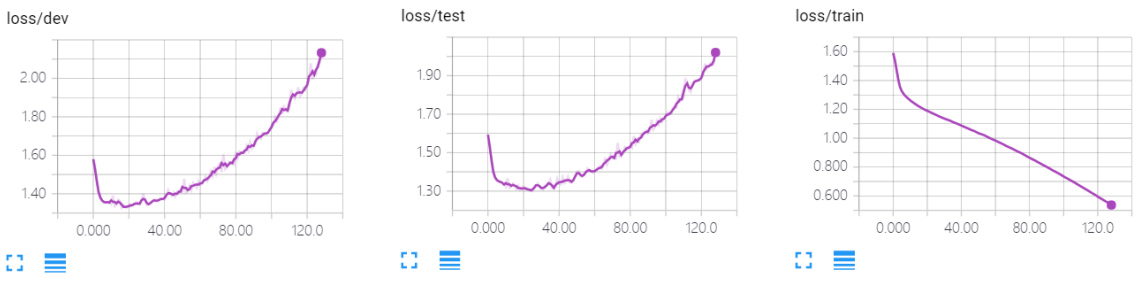


Figure 8: BasicRNNCell loss-epoch 曲线

accuracy-epoch 曲线如下：

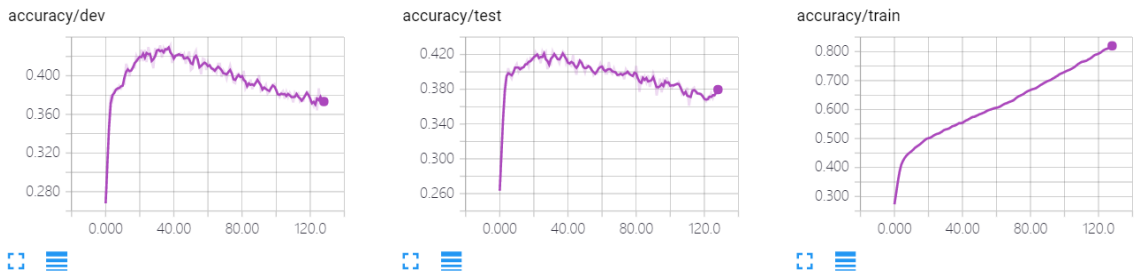


Figure 9: BasicRNNCell accuracy-epoch 曲线

3.1.2 GRUCell

笔者实现的单层 GRU，在测试集上的精度为 44.79%。
网络超参数配置如下：

超参数	取值	备注
`embed_units`	300	词向量维数
`units`	512	RNN 隐藏状态节点数
`batch_size`	16	

使用 TensorBoard 绘制训练集、开发集、测试集上的 loss-epoch 曲线如下：

Figure 10: GRUCell loss-epoch 曲线

accuracy-epoch 曲线如下：

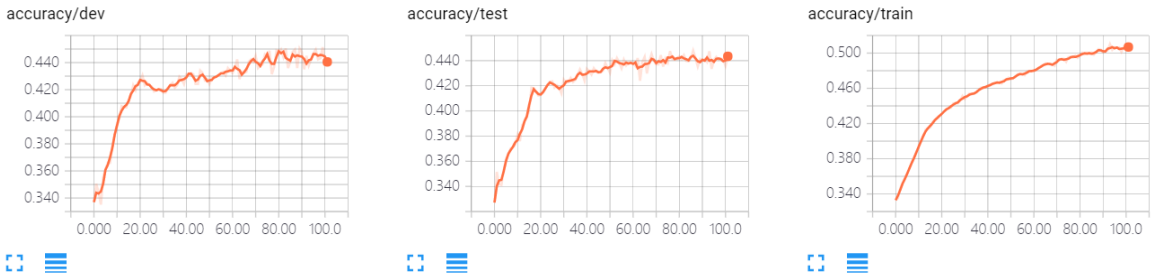


Figure 11: GRUCell accuracy-epoch 曲线

3.1.3 BasicLSTMCell

笔者实现的单层 LSTM，在测试集上的精度为 45.02%。
网络超参数配置如下：

超参数	取值	备注
`embed_units`	300	词向量维数
`units`	512	RNN 隐藏状态节点数
`batch_size`	16	

使用 TensorBoard 绘制训练集、开发集、测试集上的 loss-epoch 曲线如下：

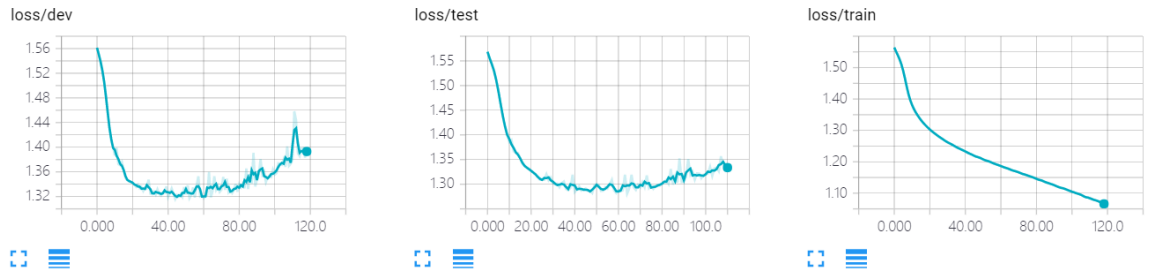


Figure 12: BasicLSTMCell loss-epoch 曲线

accuracy-epoch 曲线如下：

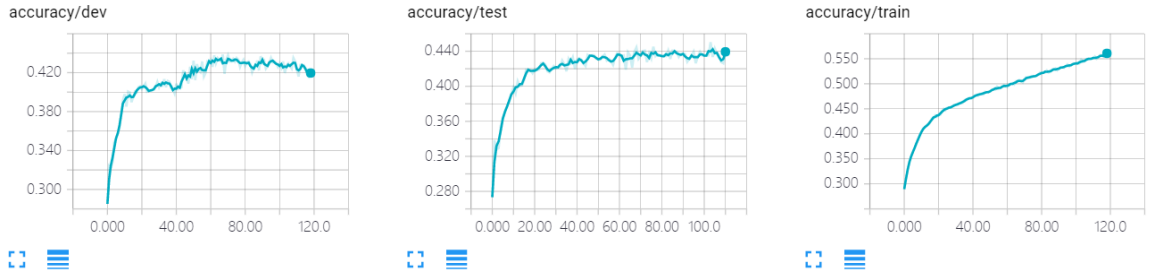


Figure 13: BasicLSTMCell accuracy-epoch 曲线

3.2 最终模型

笔者采用双向 LSTM，在测试集上的精度为 45.34%。
网络超参数配置如下：

超参数	取值	备注
`embed_units`	300	词向量维数
`units`	512	RNN 隐藏状态节点数
`batch_size`	16	

使用 TensorBoard 绘制训练集、开发集、测试集上的 loss-epoch 曲线如下：

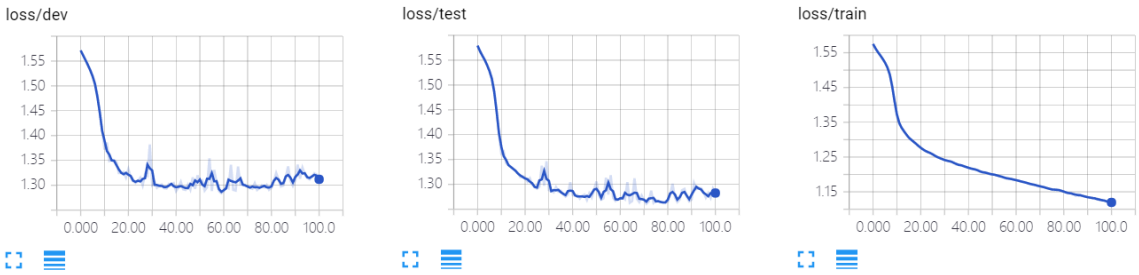


Figure 14: Bi-LSTM loss-epoch 曲线

accuracy-epoch 曲线如下：

Figure 15: Bi-LSTM accuracy-epoch 曲线

4 对比分析

4.1 单层 RNN

以下从训练时间、收敛性、测试精度 3 个方面对 3 种 RNNCell 加以分析：

RNNCell	训练时间	收敛性	测试精度
BasicRNNCell	在 CPU 上约 80s/epoch	收敛后期严重过拟合	42.35%
GRUCell	在 CPU 上约 150s/epoch	无明显过拟合	44.79%
BasicLSTMCell	在 CPU 上约 200s/epoch	略有过拟合	45.02%

使用 TensorBoard 绘制训练集、开发集、验证集上的 loss-epoch 曲线如下：

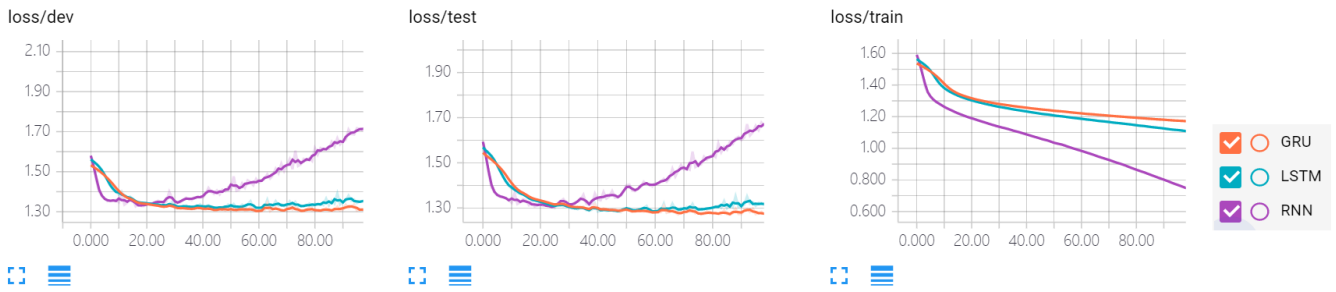


Figure 16: loss-epoch 曲线

accuracy-epoch 曲线如下：

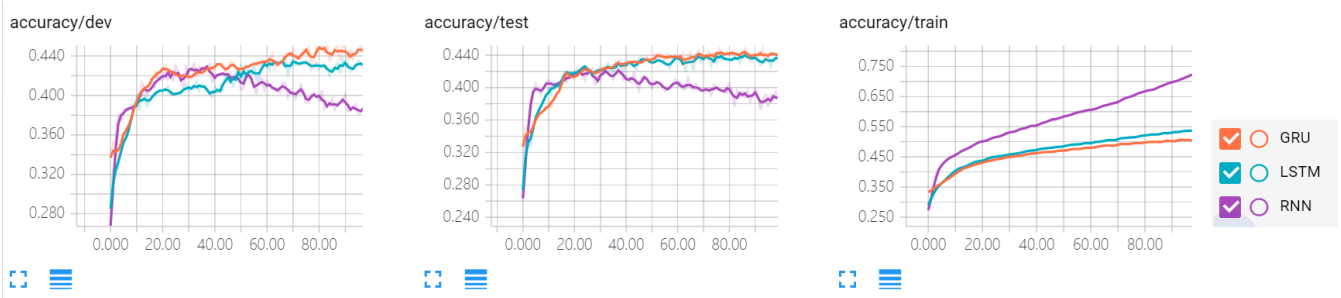


Figure 17: accuracy-epoch 曲线

训练时间 随着计算复杂程度的增大，BasicRNNCell, GRUCell, BasicLSTMCell 的训练时间依次增大。这是由于使用的门机制越精细复杂，需要的计算量也就越大，从而训练时间也就越长。然而结合训练精度来看，GRU 与 LSTM 在门机制上花费的计算量是值得的，有效地管理了记忆的遗忘与更新。

收敛性 从图中可以看出，3 种 RNNCell 均可随着 epoch 较为快速地收敛，其中 Basic RNN 收敛最快，但在收敛后期过拟合严重。GRU 与 LSTM 在这一点上则效果相似，但 LSTM 在训练后期稍有过拟合现象。

测试精度 从结果可以看出，对记忆的门机制管理越精细复杂，最终的测试精度也就越高。这是因为门机制使得 RNN 内部可以更有效地保持长期记忆，并合理更新短期记忆，从而学习到更好的隐藏表示。

综合评价 综合以上几点来看，笔者认为，GRUCell 较好地权衡了训练时长与测试精度的要求，对于本任务来说是较好的选择。

4.2 最终模型

以下从训练时间、收敛性、测试精度 3 个方面对比笔者实现的模型与上面的 GRU：

网络结构	训练时间	收敛性	测试精度
单层 GRU	在 CPU 上约 150s/epoch	无明显过拟合	44.79%
单层 Bi-LSTM	在 CPU 上约 450s/epoch	无明显过拟合	45.34%

使用 TensorBoard 绘制训练集、开发集、验证集上的 loss-epoch 曲线如下：

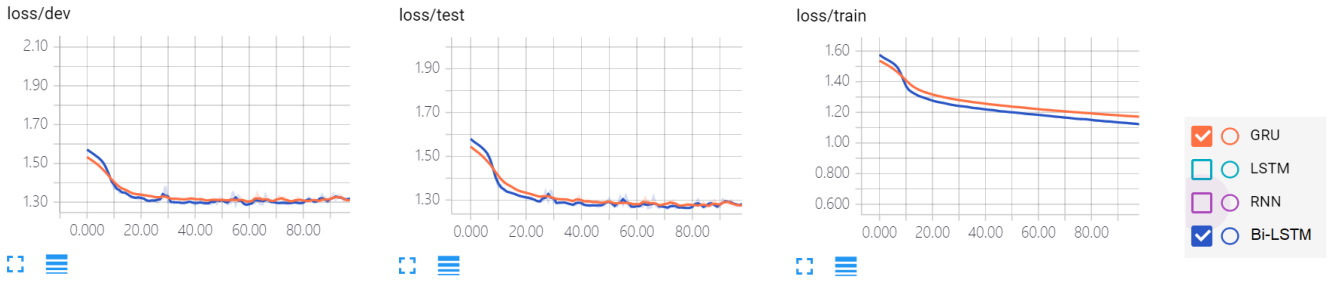


Figure 18: loss-epoch 曲线

accuracy-epoch 曲线如下：

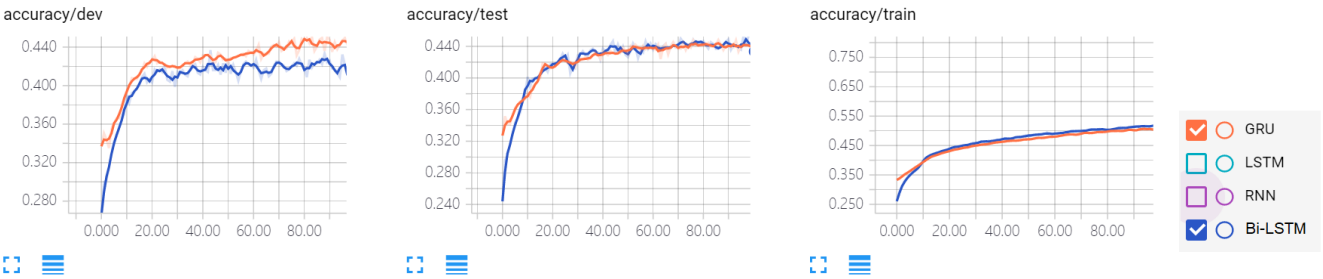


Figure 19: accuracy-epoch 曲线

训练时间 由于 Bi-LSTM 大约需要 LSTM 双倍的计算量，加之 LSTM 本身的门结构就更为复杂，故 Bi-LSTM 的训练时间显著地高于 GRU。

收敛性 从图中可以看出，Bi-LSTM 与 GRU 在收敛性上类似，均可较快收敛，且无显著过拟合现象。

测试精度 从结果可以看出，Bi-LSTM 的测试精度略高于 GRU。这是因为 Bi-LSTM 考虑了更多的上下文信息，而不像 GRU 或 LSTM 将信息的传递限制为单向。

综合评价 综合以上几点来看，Bi-LSTM 在测试精度上略高于 GRU，而 GRU 在训练时长上显著低于 Bi-LSTM。应根据具体任务在测试精度与训练时间之间进行权衡取舍。

5 可视化分析

词向量 PCA 结果 词向量通过 PCA 降维到 3D 空间中，结果如下：

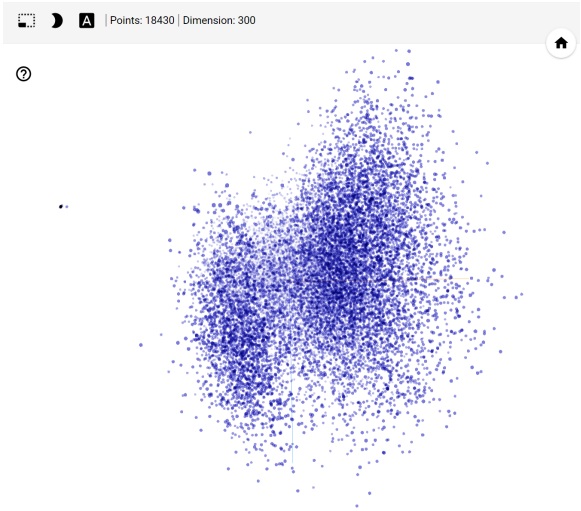


Figure 20: 词向量 PCA 结果

可以发现图中出现了 2 个主要的集群，笔者猜测可能这些代表了情感色彩较为强烈的词，从而在空间上距离较大。

偏置量分布变化 通过直方图观察 BasicLSTMCell 中的偏置量分布变化，结果如下：

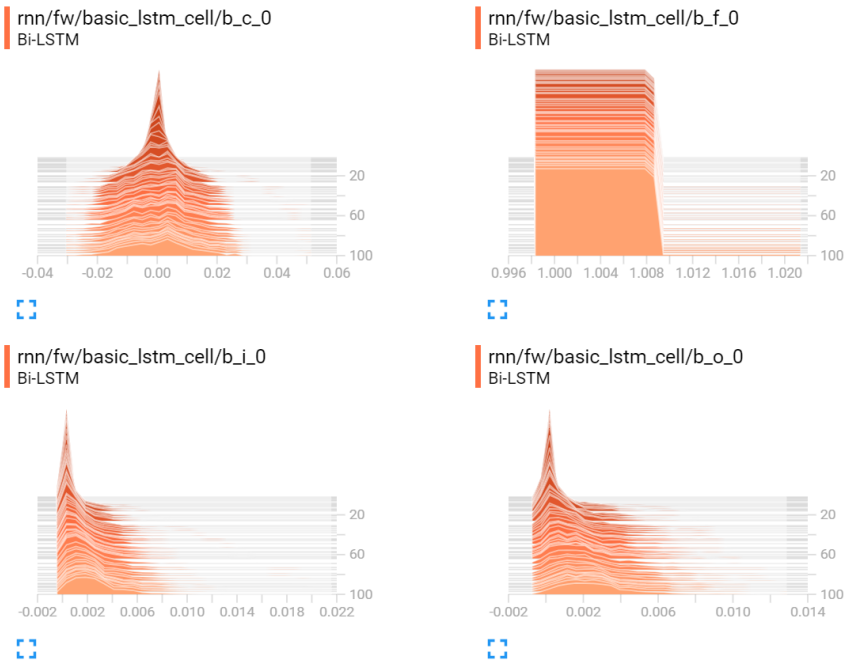


Figure 21: 偏置量分布变化

可以发现偏置量从最初的全 0 或全 1 逐渐扩散为类似正态分布的形态，这也体现了遗忘/记忆在整个训练过程中的分布变化。

References

[1] http://arxiv.org/abs/1406.1078

[2] http://arxiv.org/abs/1409.2329

[3] https://zhuanlan.zhihu.com/p/28196873

[4] http://colah.github.io/posts/2015-08-Understanding-LSTMs/

[5] http://blog.csdn.net/meanme/article/details/48845793

[6] http://blog.csdn.net/wuzqChom/article/details/75453327