如很久之前的一篇文章（学会区分 RNN 的 output 和 state）所说，**RNN 的核心就是一个函数** $y_t, s_t = f(x_t, s_{t-1})$，其中 x_t 是当前时间步的输入，s_{t-1} 是前一时间步的 RNN 状态，f 表示 RNN 内部的运算过程，最终返回当前时间步的输出 y_t 和更新后的 RNN 状态 s_t。在 TF 的实现里，所有 RNN 都是 RNNCell 的子类，上面所说的函数 f 就是该类的 call 方法。

做了这层抽象之后，多层 RNN 和单层 RNN 就可以统一起来：多层 RNN 作为一个整体也在沿时间轴循环，其输入是最底层 RNN 的输入，输出是最顶层 RNN 的输出，状态是每一层 RNN 状态组成的元组。

其实 TF 中的 MultiRNNCell 也是这么实现的。在源码（github.com/tensorflow/t）中，MultiRNNCell 也是 RNNCell 的一个子类，它接受一个 RNNCell 实例的列表，然后在 call 函数中自底向上逐层调用每个 RNNCell 实例的 call 方法，最终把最上层 RNNCell 的输出作为整体的输出，把所有层 RNNCell 的新状态都收集起来、类型转换成 tuple，作为整体的新状态。

那注意力机制如何实现呢？

TF 的文档在这里：

https://www.tensorflow.org/api_guides/python/cc
www.tensorflow.org

其中有一段示范代码如下：

```
cell = tf.contrib.rnn.DeviceWrapper(LSTMCell(512), "/device:GPU:0")
attention_mechanism = tf.contrib.seq2seq.LuongAttention(512, encoder_outputs)
attn_cell = tf.contrib.seq2seq.AttentionWrapper(
  cell, attention_mechanism, attention_size=256)
```

简单地说，就是先定义一层普通的 RNNCell（例如 LSTM），然后定义某种 Attention 机制的实例（如 LuongAttention 或者 BahdanauAttention），最后把这俩东西都传给 AttentionWrapper，返回封装后的 RNNCell。

是的！**封装后还是 RNNCell 的实例！**

可这是如何做到的呢？原来，这里有一个偷天换日的技巧，把原先 RNNCell 的状态包装一下就行了。**定义一个新类叫 AttentionWrapperState，原先 RNNCell 的状态只是它的一个域，而它还有别的域，保存和注意力机制相关的东西。这样一来，封装后的单元仍然有类似公式** $y_t, s_t = f(x_t, s_{t-1})$ **的循环**，但是循环中用到的状态变量 s_t 的类型已经变了，从原先 RNNCell 的状态类型变成了 AttentionWrapperState。但是不管怎么说，都维持这个 RNNCell 核心的循环式，因此它仍然是一种 RNNCell，不影响它和别的接口相结合（例如 tf.nn.dynamic_rnn），这样就完成了封装。

具体实现上，AttentionMechanism（包括其子类）和 AttentionWrapper 是这么合作的：

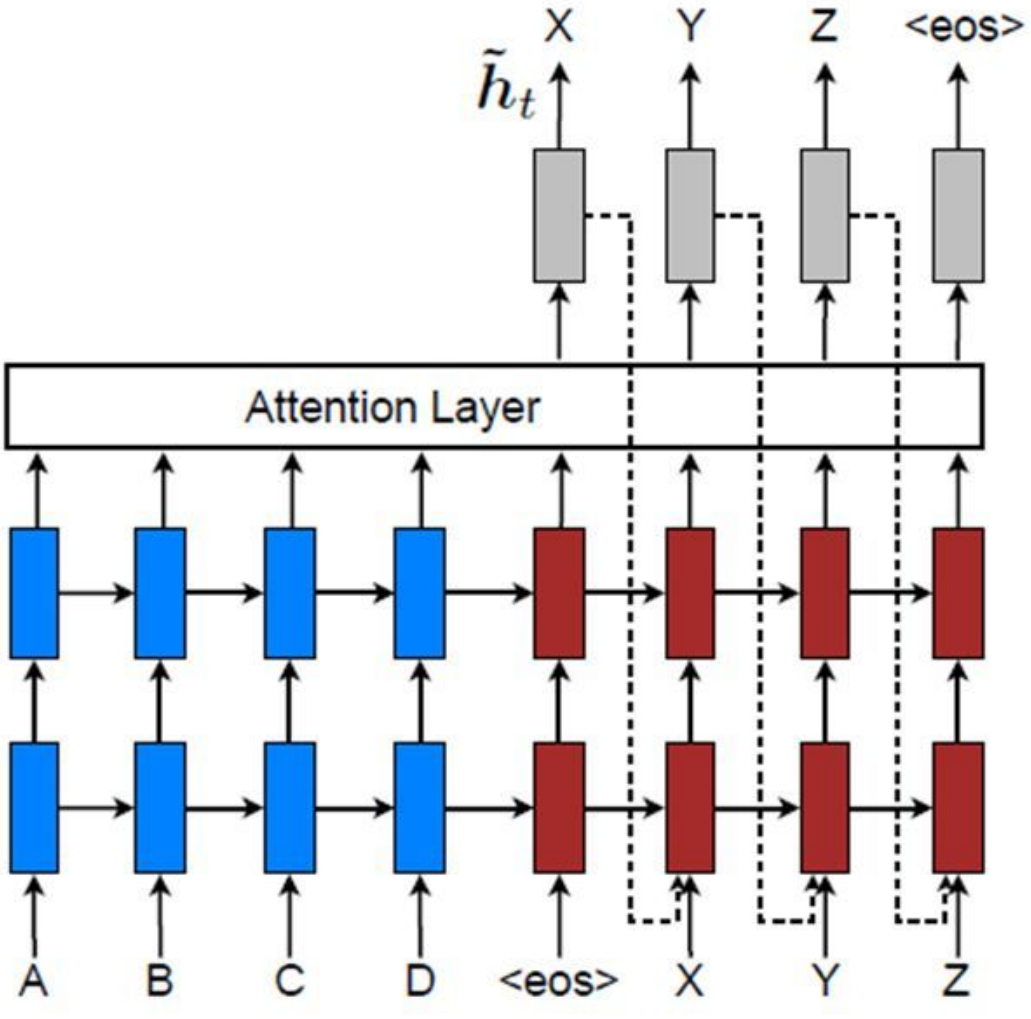
- AttentionMechanism 中保存了 memory bank（在 seq2seq 中就是 encoder 各个时间步的输出组成的张量），同时其中还定义了注意力的计算方法（例如 Luong's style 或 Bahdanau's style）；
- AttentionWrapper 构造时要接受一个 AttentionMechanism 的实例（或者多个 AttentionMechanism 实例组成的列表，应该是为了实现 Multi-head Attention 而设计的），然后它也有一个 call 函数，其中会调用被封装的 cell 的 call 函数，从而更新内部 cell 的状态，然后以更新后的内部 cell 状态为 query，和它吃掉的 AttentionMechanism 实例中保存的 memory 计算注意力权重，得到对齐向量和上下文向量

- 对齐向量：代码中叫 alignments 或者 attention_states，不知道为啥把一个东西叫了俩名字返回两遍，可能是设计遗留问题；文献中常用 $a_{:,j}$ 表示，是一个归一化的概率向量，表示 decoder 在时间步 j 对齐到 encoder 各个时间步的概率
- 上下文向量：代码中叫 attention；文献中多叫做 context vector，常用字母 c 表示。当然这里还稍微有一点小区别，就是代码中的 attention 有可能是 context vector 本身，也有可能是 context vector 再做一次变换得到的结果（用参数 attention_layer_size 来控制是否加这一层）。

文章给了一段示例代码，说明 AttentionWrapper 内部的（简化版的）工作流程：

```
cell_inputs = concat([inputs, prev_state.attention], -1)
cell_output, next_cell_state = cell(cell_inputs, prev_state.cell_state)
score = attention_mechanism(cell_output)
alignments = softmax(score)
context = matmul(alignments, attention_mechanism.values)
attention = tf.layers.Dense(attention_size)(concat([cell_output, context], 1))
next_state = AttentionWrapperState(
  cell_state=next_cell_state,
  attention=attention)
output = attention
return output, next_state
```

事实上，这个流程恰好对应了论文 Thang Luong, Hieu Pham, and Chris Manning. **Effective Approaches to Attention-based Neural Machine Translation**. EMNLP'15。该论文中有一幅示意图如下：



Input feeding

在 decoder 侧取一个时间步（例如取输入 X 所在的时间步），然后把上下两个红框看成一个整体（相当于 TF 代码 cell = MultiRNNCell([GRUCell(layer_size) for _ in range(2)]) 的运行结果），那么：

- 上面参考代码的第一步就是把当前输入 X 和前一步 attention 结果拼起来（如虚线连接所示，论文中称为 input feeding），作为当前时间步的输入。这种做法的直觉是，前一步 attention 的信息可能会对当前预测有帮助，例如让模型避免连续两次注意到同一个地方，跟个结巴似的一直输出一个词。
  - 可以用 cell_input_fn=lambda inputs, attention: inputs 把这种拼接行为关掉
- 第二步是对被封装的 RNNCell 进行正常的更新，取它的输出作为 query 供后续使用
- 第三步到第四步是计算 encoder 每一步和当前 query 的匹配得分 score，然后归一化成概率分布
- 第五步得到 context vector（encoder 每个时间步输出的加权组合），在上图中用灰色框表示
- 第六步是对 context vector 继续做变换的结果（默认不做变换，此时 attention = context vector）
- 最后把新的相关信息封装成新的 AttentionWrapperState，就是下一时间步的状态
- 输出值就是 attention（也可以切换成被封装的 cell 的输出 cell_output，用布尔参数 output_attention 控制）

其他评论：

- 这里的封装方式是先更新内部 RNNCell，再计算 attention。也正因如此，这种封装完美契合了 LuongAttention，但是与注意力机制的开山之作（Bahdanau 那篇）却不太对味儿。
- 其实这种思路是更自然的，Bahdanau 那篇要用前一步的状态做 query 计算 attention，时间上错位一格实现起来很别扭，很多人都看这个不顺眼了
- 如果要严格复现 Bahdanau 那篇开山之作，用现在这个 AttentionWrapper 似乎挺难的。还不如照着这个 AttentionWrapper 写一个 BahdanauAttentionWrapper 类，改改其中计算 attention 的位置即可
- 同理，如果要实现别的或奇葩或复杂的 attention 方式，例如 conditional GRU，最好的办法可能还是重写……这个接口真的不好设计。

提取注意力矩阵：

- AttentionWrapper 有一个参数 alignment_history，默认是关闭的，只要打开就好了
- 当前 batch 的所有对齐向量都被保存在最终的 AttentionWrapperState 的域 alignment_history 中
- 该变量的类型是 TensorArray（比起普通的 Tensor 更接近于传统命令式程序设计中的数组。可以给每一个下标赋值、读取每一个下标处存储的 Tensor，不过只能写一次）
- 只需调用 stack() 方法把它恢复成普通的 Tensor，然后在 Session 里 run 一下即可