[论文阅读] RNN 在阿里DIEN中的应用

原创 罗西的思考 罗西的思考 11月7日

[论文阅读] RNN 在阿里DIEN中的应用

0x00 摘要

本文基于阿里推荐DIEN代码，梳理了下RNN一些概念，以及TensorFlow中的部分源码。本博客旨在帮助小伙伴们详细了解每一步骤以及为什么要这样做。

0x01 背景知识

1.1 RNN

RNN，循环神经网络，Recurrent Neural Networks。

人们思考问题往往不是从零开始的，比如阅读时我们对每个词的理解都会依赖于前面看到的一些信息，而不是把前面看的内容全部抛弃再去理解某处的信息。应用到深度学习上面，如果我们想要学习去理解一些依赖上文的信息，RNN 便可以做到，它有一个循环的操作，可以使其可以保留之前学习到的内容。

最普通的RNN定义方式是：

```
output = new_state = f(W * input + U * state + B) = act(W * input + U * state + B)
```

- U，W 是网络参数（权重矩阵），b 是偏置参数，这些参数通过后向传播训练网络学习得到。
- act 是激活函数，通常选择 sigmoid 或 tanh 函数。

1.2 DIEN项目代码

在DIEN项目中，把TensorFlow的rnn代码拿到自己项目中，做了一些修改，具体是：

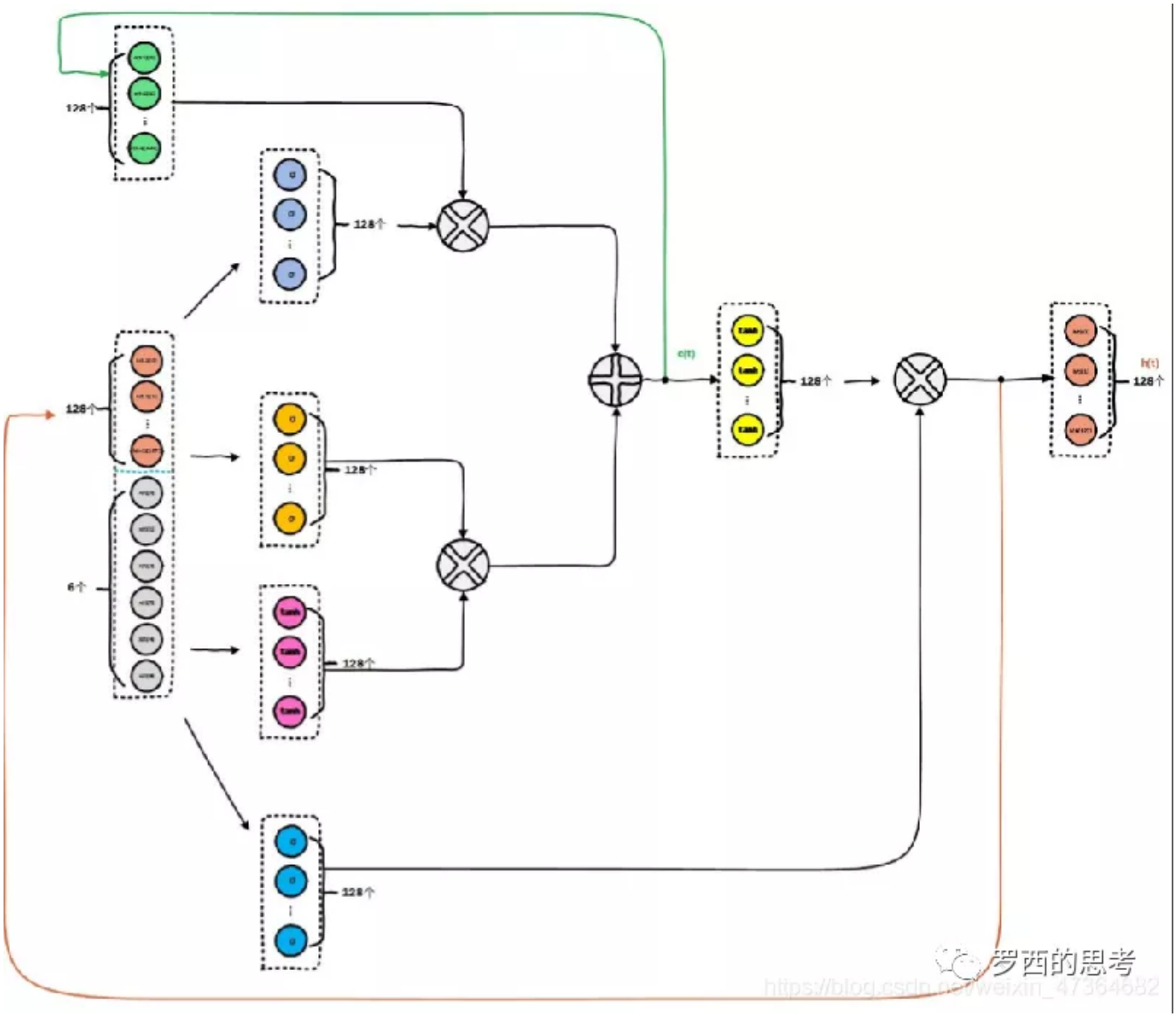
- 使用了 GRUCell；
- 自定义了 VecAttGRUCell；

- 因为修改了VecAttGRUCell接口，所以修改了rnn.py；

0x02 Cell

RNN的基本单元被称为Cell，别小看那一个小小的cell，它并不是只有1个neuron unit，而是n个hidden units。

因此，我们注意到tensorflow中定义一个cell（BasicRNNCell/BasicLSTMCell/GRUCell/RNNCell/LSTMCell）结构的时候需要提供的一个参数就是hidden_units_size。



在这里插入图片描述

在实际的神经网络中，各个门处理函数 其实是由一定数量的隐含层神经元来处理。

在RNN中，M个神经元组成的隐含层，实际的功能应该是 f(wx + b), 这里实现了两步：

- 首先M个隐含层神经元与输入向量X之间全连接，通过w参数矩阵对x向量进行加权求和；

- 其次就是对x向量各个维度上进行筛选，加上bias偏置矩阵后，通过f激励函数， 得到隐含层的输出；

在LSTM Cell中，一个cell 包含了若干个门处理函数，假如每个门的物理实现，我们都可以看做是由num_hidden个神经元来实现该门函数功能， 那么每个门各自都包含了相应的w参数矩阵以及bias偏置矩阵参数，这就是在上图中的实现。

从图中可以看出，cell单元里有四个门，每个门都对应128个隐含层神经元，相当于四个隐含层，每个隐含层各自与输入x 全连接，而输入x向量是由两部分组成，一部分是上一时刻cell 输出，大小为128， 还有部分就是当前样本向量的输入，大小为6，因此通过该cell内部计算后，最终得到当前时刻的输出，大小为128，即num_hidden，作为下一时刻cell的一部分输入。

下面我们结合TensorFlow来具体剖析下Cell的实现机制和原理。

2.1 RNNCell（抽象父类）

2.1.1 基础

“RNNCell”，它是TensorFlow中实现RNN的基本单元，每个RNNCell都有一个call方法，使用方式是：(output, next_state) = call(input, state)。

RNNCell是一个抽象的父类，其他的RNNcell都会继承该方法，然后具体实现其中的call()函数。

RNNCell是包含一个State（状态）并且能够执行一些处理输入矩阵的对象。RNNCell将输入的矩阵（Input Matrix）运算输出一个包含”self.output”列的输出矩阵（Ouput Matrix）。

state: state就是rnn网络中rnn cell的状态，比如说如果你的rnn定义包含了N个单元（也就是你的self.state_size是个整数N），那么在你每次执行RNN网络时就应该给一个[batch_size,self.state_size] 形状的2D Tensor来表示当前RNN网络的状态，而如果你的self.state_size 是一个元祖，那么给定的状态也应该是一个Tuple，每个Tuple里的状态表示和之前的方式一样。

- 如果定义了 “self.state_size”这个属性，并且取值为一个整数，那么RNNCell则会同时输出一个状态矩阵（State Matrix），包含 “self.state_size” 列。
- 如果 “self.state_size” 定义为一个整数的Tuple，那么则是输出对应长度的状态矩阵的Tuple，Tuple中的每一个状态矩阵长度还是和之前的一样，包含 “self.state_size” 列。

RNNCell其主要是zero_state()和call()两个函数。

- zero_state 用于初始化初始状态 h0 为全零向量。
- call 定义实际的RNNCell的操作（比如RNN就是一个激活，GRU的两个门，LSTM的三个门控等，不同的RNN的区别主要体现在这个函数）。

除了call方法外，对于RNNCell，还有两个类属性比较重要，其中 state_size() 和 output_size() 方法设置为类属性，可以当做属性来调用（这里用到的是Python内置的@property装饰器，就是负责把一个方法变成属性调用的，很像C#中的属性、字段的那种概念）：

- state_size，是隐层的大小（代表 Cell 的状态 state 大小）
- output_size，是输出的大小（输出维度）

比如我们通常是将一个batch送入模型计算，设输入数据的形状为(batch_size, input_size)，那么计算时得到的隐层状态就是(batch_size, state_size)，输出就是(batch_size, output_size)。

但这里两个方法都没有实现，意思是说我们必须要实现一个子类继承 RNNCell 类并实现这两个方法。

```
class RNNCell(base_layer.Layer):

  def __call__(self, inputs, state, scope=None):
    if scope is not None:
      with vs.variable_scope(scope,
                             custom_getter=self._rnn_get_variable) as scope:
        return super(RNNCell, self).__call__(inputs, state, scope=scope)
    else:
      with vs.variable_scope(vs.get_variable_scope(),
                             custom_getter=self._rnn_get_variable):
        return super(RNNCell, self).__call__(inputs, state)

  def _rnn_get_variable(self, getter, *args, **kwargs):
    variable = getter(*args, **kwargs)
    if context.in_graph_mode():
      trainable = (variable in tf_variables.trainable_variables() or
                   (isinstance(variable, tf_variables.PartitionedVariable) and
                    list(variable)[0] in tf_variables.trainable_variables()))
    else:
      trainable = variable._trainable  # pylint: disable=protected-access
    if trainable and variable not in self._trainable_weights:
      self._trainable_weights.append(variable)
    elif not trainable and variable not in self._non_trainable_weights:
```

```
      self._non_trainable_weights.append(variable)
    return variable

  @property
  def state_size(self):
    raise NotImplementedError("Abstract method")

  @property
  def output_size(self):
    raise NotImplementedError("Abstract method")

  def build(self, _):
    pass

  def zero_state(self, batch_size, dtype):
    with ops.name_scope(type(self).__name__ + "ZeroState", values=[batch_size]):
      state_size = self.state_size
      return _zero_state_tensors(state_size, batch_size, dtype)
```

2.1.2 call

每个派生的RNNCell必须有以下的属性并实现具有如下函数签名的函数(output, next_state) = call(input, state)。 可选的第三个输入参数‘scope’，用于向下兼容，给子类定制化使用。scope传入的值是tf.Variable类型，用于更方便的管理变量。

> 从给定的state开始运行，根据rnn cell的输入
>
> args:
>
> > inputs：是一个具有二维的张量shape为[batch_size, input_size] states：如果 `self.state_size` 是一个整数，state就应该是一个二维张量 shape是 `[batch_size, self.state_size]` ,否则，如果 `self.state_size` 是一个整数的tuple（例如LSTM需要计算cell state和 hidden unit state ， 就是一个tuple），那么state就应该是 `[batch_size, s] for s in self.state_size` 形状的tuple。Scope：由其他子类创建的变量。
>
> Return：

> 是一对，包括：输出： `[batch_size, self.output_size]` State: 和state相匹配的shape

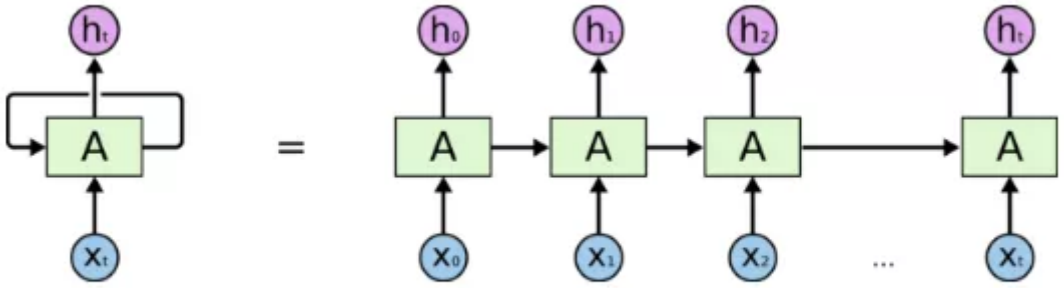
每调用一次RNNCell的call方法，就相当于在时间上“推进了一步”，这就是RNNCell的基本功能。

2.2 BasicRNNCell（基础类）

2.2.1 基础

RNNCell只是一个抽象类，我们用的时候都是用的它的两个子类 BasicRNNCell 和 BasicLSTMCell。顾名思义，前者是RNN的基础类，后者是LSTM的基础类。

BasicRNNCell 就是我们常说的 RNN。



An unrolled recurrent neural network. 罗西的思考

最简单的RNN结构如上图所示。其代码如下：

```
class BasicRNNCell(RNNCell):
  def __init__(self, num_units, activation=None, reuse=None):
    super(BasicRNNCell, self).__init__(_reuse=reuse)
    self._num_units = num_units
    self._activation = activation or math_ops.tanh
    self._linear = None

  @property
  def state_size(self):
    return self._num_units

  @property
  def output_size(self):
```

```
    return self._num_units

  def call(self, inputs, state):
    """Most basic RNN: output = new_state = act(W * input + U * state + B)."""
    if self._linear is None:
      self._linear = _Linear([inputs, state], self._num_units, True)

    output = self._activation(self._linear([inputs, state]))
    # output = Ht = tanh([x,Ht-1]*W + B)
    # 一个output作为下一时刻的输入Ht，另一个作为下一层的输入 Ht
    return output, output
```

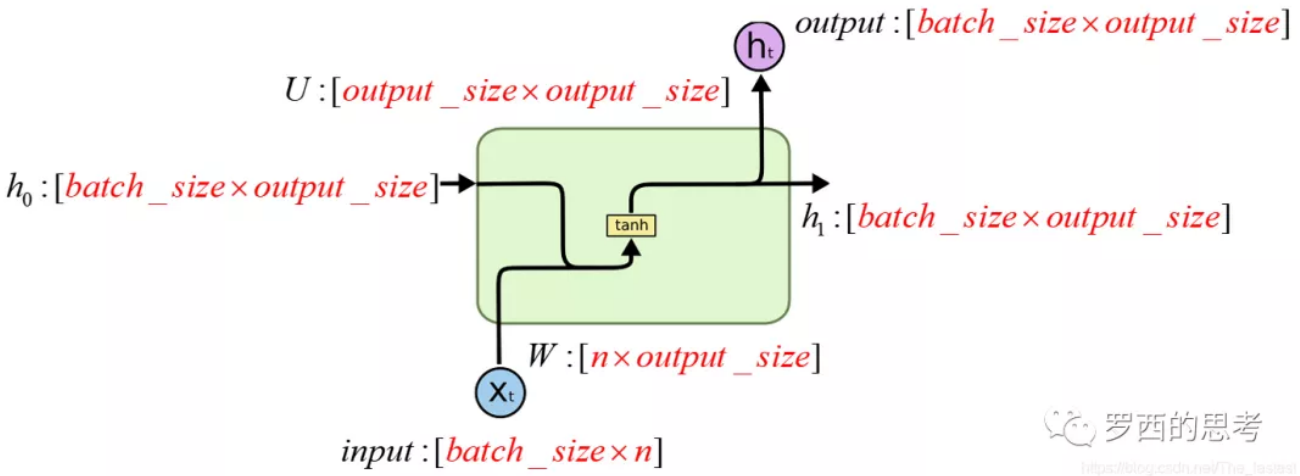
2.3.2 参数意义

可以看到在初始化 `__init__` 中有若干参数。

```
def __init__(self, num_units, activation=None, reuse=None):
```

`__init__` 最重要的一个参数是 num_units，意思就是这个 Cell 中神经元的个数，另外还有一个参数 activation 即默认使用的激活函数，默认使用的 tanh，reuse 代表该 Cell 是否可以被重新使用。

我们知道一个最基本的RNN单元中有三个可训练的参数W, U, B，以及两个输入变量。所以我们在构造RNN的时候就需要指定各个参数的维度了。



注，上图中的 `n` 表示的是输入的维度dim

从源码中可以看出BasicRNNCell中：

- state_size 就是num_units : `def state_size(self): return self._num_units`

- output_size 就是num_units： `def output_size(self): return self._num_units`
- 即 把state_size和output_size定义成相同，
- ht和output也是相同的（call函数输出是两个output： `return output, output` ，即其并未定义输出部分）。
- 从 _linear 可以看出， `output_size` 指的是偏置B的维度（下文会讲解 _Linear）。

2.2.3 功能

其主要功能实现就是call函数的第一行注释，就是input和前一时刻状态state经过一个线性函数在经过一个激活函数即可，也就是最普通的RNN定义方式。也就是说

```
output = new_state = f(W * input + U * state + B) = act(W * input + U * state + B)
```

在 state_size()、output_size() 方法里，其返回的内容都是 num_units，即神经元的个数。

接下来 call() 方法中：

- 传入的参数为 inputs 和 state，即输入的 x 和 上一次的隐含状态
- 首先实例化了一个 _Linear 类，这个类实际上就是做线性变换的类，将二者传递过来，然后直接调用，就实现了 w * [inputs, state] + b 的线性变换： `output = new_state = tanh(W * input + U * state + B).`
- 其次回到 BasicRNNCell 的 call() 方法中，在 _linear() 方法外面又包括了一层 _activation() 方法，即对线性变换应用一次 tanh 激活函数处理，作为输出结果。
- 最后返回的结果是 output 和 output，第一个代表 output，第二个代表隐状态，其值也等于 output。

2.2.4 Linear

上面写到使用了 _linear 类，现在我们就介绍下。

这个类传递了 [inputs, state] 作为 **call**() 方法的 args，会执行 concat() 和 matmul() 方法，然后接着再执行 bias_add() 方法，这样就实现了线性变换。

而output_size是输出层的大小，我们可以看到

- BasicRNNCell中，output_size就是_num_units；
- GRUCell中是2 * _num_units；

- BasicLSTMCell中是4 * _num_units；

这是因为_linear中执行的是RNN中的几个等式的 `Wx + Uh + B` 的功能，但是不同的RNN中数量不同，比如LSTM中需要计算四次，然后直接把output_size定义为4_num_units，再把输出进行拆分成四个变量即可。

下面是源码缩减版

```
class _Linear(object):

  def __init__(self, args, output_size, build_bias, bias_initializer=None,
               kernel_initializer=None):

    self._build_bias = build_bias

    if not nest.is_sequence(args):
      args = [args]
      self._is_sequence = False
    else:
      self._is_sequence = True

    # Calculate the total size of arguments on dimension 1.
    total_arg_size = 0
    shapes = [a.get_shape() for a in args]
    for shape in shapes:
     total_arg_size += shape[1].value

    dtype = [a.dtype for a in args][0]

  # 循环该函数 num_step(句子长度) 次，则该层计算完；
  def __call__(self, args):

    # 如果是第 0 时刻，那么当前的 state(即上一时刻的输出H0)的值全部为0；
    # input 的 shape为: [batch_size, emb_size]
    # state 的 shape为：[batch_zize, Hidden_size]
    # matmul : 矩阵相乘
    # array_ops.concat: 两个矩阵连接,连接后的 shape 为 [batch_size,input_size + Hidden_size],实际就

    if not self._is_sequence:
      args = [args]

    if len(args) == 1:
      res = math_ops.matmul(args[0], self._weights)
```

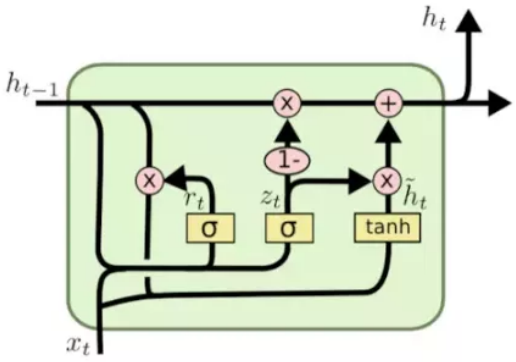
```
    else:
      # 此时计算: [input,state] * [W,U] == [Xt,Ht-1] * W，得到的shape为:[batch_size,Hidden_size]
      res = math_ops.matmul(array_ops.concat(args, 1), self._weights)

    # B 的shape 为:【Hidden_size】
    # [Xt,Ht-1] * W 计算后的shape为：[batch_size,Hidden_size]
    # nn_ops.bias_add,这个函数的计算方法是，让每个 batch 得到的值，都加上这个 B
  # 加上B后：Ht = tanh([Xt, Ht-1] * W + B),得到的 shape 还是: [batch_size,Hidden_size]
  # 那么这个 Ht 将作为下一时刻的输入和下一层的输入；
    if self._build_bias:
      res = nn_ops.bias_add(res, self._biases)
    return res
```

2.3 GRUCell

GRU，Gated Recurrent Unit。在 GRU 中，只有两个门：重置门（Reset Gate）和更新门（Update Gate）。同时在这个结构中，把 Ct 和隐藏状态进行了合并，整体结构比标准的 LSTM 结构要简单，而且这个结构后来也非常流行。

接下来我们看一下GRU的定义，相比BasicRNNCell只改变了call函数部分，增加了重置门和更新门两部分，分别由r和u表示。然后c表示要更新的状态值。其对应的图及公式如下所示：



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


```
r = f(W1 * input + U1 * state + B1)
u = f(W2 * input + U2 * state + B2)
c = f(W3 * input + U3 * r * state + B3)
h_new = u * h + (1 - u) * c
```

GRUCell的实现代码缩减版如下：

```
class GRUCell(RNNCell):
```

```
def __init__(self,
             num_units,
             activation=None,
             reuse=None,
             kernel_initializer=None,
             bias_initializer=None):
  super(GRUCell, self).__init__(_reuse=reuse)
  self._num_units = num_units
  self._activation = activation or math_ops.tanh
  self._kernel_initializer = kernel_initializer
  self._bias_initializer = bias_initializer
  self._gate_linear = None
  self._candidate_linear = None

@property
def state_size(self):
  return self._num_units

@property
def output_size(self):
  return self._num_units

def call(self, inputs, state):

  value = math_ops.sigmoid(self._gate_linear([inputs, state]))
  r, u = array_ops.split(value=value, num_or_size_splits=2, axis=1)

  r_state = r * state
  if self._candidate_linear is None:
    with vs.variable_scope("candidate"):
      self._candidate_linear = _Linear(
          [inputs, r_state],
          self._num_units,
          True,
          bias_initializer=self._bias_initializer,
          kernel_initializer=self._kernel_initializer)
  c = self._activation(self._candidate_linear([inputs, r_state]))
  new_h = u * state + (1 - u) * c
  return new_h, new_h
```

具体函数功能解析如下：

在 state_size()、output_size() 方法里，其返回的内容都是 num_units，即神经元的个数。

call() 方法中，因为 Reset Gate rt 和 Update Gate zt 分别用变量 r、u 表示，它们需要先对 ht-1 即 state 和 xt 做合并，然后再实现线性变换，再调用 sigmod 函数得到：

```
value = math_ops.sigmoid(self._gate_linear([inputs, state]))
r, u = array_ops.split(value=value, num_or_size_splits=2, axis=1)
```

然后需要求解 ht~，首先用 rt 和 ht-1 即 state 相乘：

```
r_state = r * state
```

然后将其放到线性函数里面 `_Linear` ，再调用 tanh 激活函数即可：

```
c = self._activation(self._candidate_linear([inputs, r_state]))
```

最后计算隐含状态和输出结果，二者一致：

```
new_h = u * state + (1 - u) * c
```

这样即可返回得到输出结果和隐藏状态。

```
return new_h, new_h
```

2.4 自定义RNNCell

自定义RNNCell的方法比较简单，那就是继承_LayerRNNCell这个抽象类，然后一定要实现__init__、build、__call__这三个函数就行了，其中在call函数中实现自己需要的功能即可。（注意：build只调用一次，在build中进行变量实例化，在call中实现具体的rnncell操作）。

2.5 DIEN之VecAttGRUCell

调用VecAttGRUCell的代码如下：

```
rnn_outputs2, final_state2 = dynamic_rnn(VecAttGRUCell(HIDDEN_SIZE), inputs=rnn_outputs,
```

首先我们要注意到 tf.expand_dims的使用，这个函数是用来把 alphas 增加一维。

```
alphas = Tensor("Attention_layer_1/Reshape_4:0", shape=(?, ?), dtype=float32)
```

-1 表示在最后增加一维。

```
att_scores = tf.expand_dims(alphas, -1)
```

阿里在这里做的修改主要是call函数，是关于att_score的修改：

```
u = (1.0 - att_score) * u
new_h = u * state + (1 - u) * c
return new_h, new_h
```

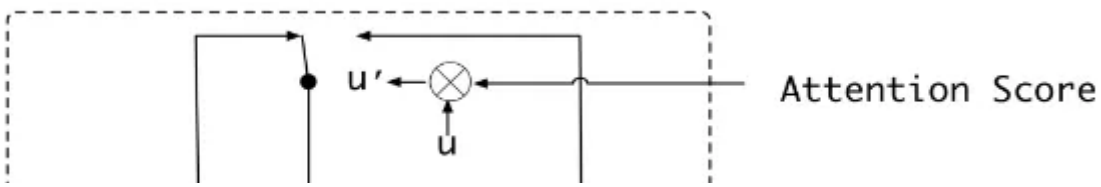
具体代码是：

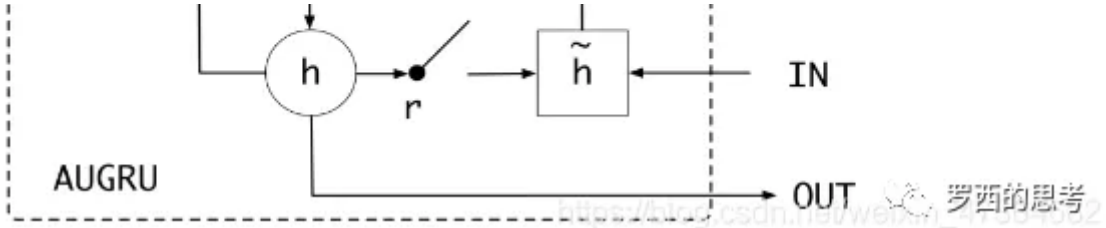
```
def call(self, inputs, state, att_score=None):
    ......
    c = self._activation(self._candidate_linear([inputs, r_state]))
    u = (1.0 - att_score) * u  # 这里是新增加的
    new_h = u * state + (1 - u) * c # 这里是新增加的
    return new_h, new_h
```

其中运行时变量如下：

```
inputs = {Tensor} Tensor("rnn_2/gru2/while/TensorArrayReadV3:0", shape=(?, 36), dtype=float32)
state = {Tensor} Tensor("rnn_2/gru2/while/Identity_2:0", shape=(?, 36), dtype=float32)
att_score = {Tensor} Tensor("rnn_2/gru2/while/strided_slice:0", shape=(?, 1), dtype=float32)
new_h = {Tensor} Tensor("rnn_2/gru2/while/add_1:0", shape=(?, 36), dtype=float32)
u = {Tensor} Tensor("rnn_2/gru2/while/mul_1:0", shape=(?, 36), dtype=float32)
c = {Tensor} Tensor("rnn_2/gru2/while/Tanh:0", shape=(?, 36), dtype=float32)
```

具体对应论文中就是：





在这里插入图片描述

0x03 RNN

3.1 一次执行多步

3.1.1 基础

基础的RNNCell有一个很明显的问题：对于单个的RNNCell，我们使用它的call函数进行运算时，只是在序列时间上前进了一步。比如使用x1、h0得到h1，通过x2、h1得到h2等**。这样的话，如果我们的序列长度为10，就要调用10次call函数，比较麻烦。对此，TensorFlow提供了一个tf.nn.dynamic_rnn函数，使用该函数就相当于调用了n次call函数。**即通过{h0,x1, x2, …., xn}直接得{h1,h2…,hn}。

```
def dynamic_rnn(cell, inputs, att_scores=None, sequence_length=None, initial_state=None,
                dtype=None, parallel_iterations=None, swap_memory=False,
                time_major=False, scope=None):
```

重要参数介绍：

- **cell**：LSTM、GRU等的记忆单元。cell参数代表一个LSTM或GRU的记忆单元，也就是一个cell。例如，cell = tf.nn.rnn_cell.LSTMCell((num_units)，其中，num_units表示rnn cell中神经元个数，也就是下文的cell.output_size。返回一个LSTM或GRU cell，作为参数传入。
- **inputs**：输入的训练或测试数据，一般格式为[batch_size, max_time, embed_size]，其中batch_size是输入的这批数据的数量，max_time就是这批数据中序列的最长长度，embed_size表示嵌入的词向量的维度。
- **sequence_length**：是一个list，假设你输入了三句话，且三句话的长度分别是5,10,25，那么sequence_length=[5,10,25]。
- **time_major**：决定了输出tensor的格式，如果为True, 张量的形状必须为 [max_time, batch_size,cell.output_size] 。如果为False, tensor的形状必须为[batch_size, max_time, cell.output_size]，cell.output_size表示rnn cell中神经元个数。

返回值如下：

outputs 就是 time_steps 步里所有的输出。它的形状为 (batch_size, time_steps, cell.output_size)。

state是最后一步的隐状态，它的形状为(batch_size, cell.state_size)。

详细如下：

- **outputs**. outputs是一个tensor，是每个step的输出值。
  - 如果time_major==True，outputs形状为 [max_time, batch_size, cell.output_size]（要求rnn输入与rnn输出形状保持一致）
  - 如果time_major==False（默认），outputs形状为 [batch_size, max_time, cell.output_size]
- **state.** state是一个tensor。state是最终的状态，也就是序列中最后一个cell输出的状态。一般情况下state的形状为 [batch_size, cell.output_size]，但当输入的cell为BasicLSTMCell时，state的形状为[2，batch_size, cell.output_size]，其中2也对应着LSTM中的cell state和hidden state

max_time就是这批数据中序列的最长长度，如果输入的三个句子，那max_time对应的就是最长句子的单词数量，cell.output_size其实就是rnn cell中神经元的个数。

3.1.2 使用

假设们输入数据的格式为(batch_size, time_steps, input_size)，其中：

- batch_size是输入的这批数据的数量；
- time_steps表示序列本身的长度，如在Char RNN中，长度为10的句子对应的time_steps就等于10；
- input_size就表示输入数据单个序列单个时间维度上固有的长度；

如下我们已经定义好了一个RNNCell，调用该RNNCell的call函数time_steps次

```
# inputs: shape = (batch_size, time_steps, input_size)
# cell: RNNCell
# initial_state: shape = (batch_size, cell.state_size)。初始状态。一般可以取零矩阵
outputs, state = tf.nn.dynamic_rnn(cell, inputs, initial_state=initial_state)
```

对于参数举例如下：

> 样本数据：
>
> > 小明爱学习
> >
> > 小王爱学习
> >
> > 小李爱学习
> >
> > 小花爱学习
>
> 通常样本数据会以 `(batch_size, time_step, embedding_size)` 送入模型，对应的可以是（4，5，100）。
>
> 4表示批量送入也就是（小，小，小，小）第二批是（明，王，李，花）...
>
> 5表示时间步长，一句话共5个字。

又比如如下代码：

```
import tensorflow as tf
import numpy as np
from tensorflow.python.ops import variable_scope as vs

output_size = 5
batch_size = 4
time_step = 3
dim = 3
cell = tf.nn.rnn_cell.BasicRNNCell(num_units=output_size)
inputs = tf.placeholder(dtype=tf.float32, shape=[time_step, batch_size, dim])
h0 = cell.zero_state(batch_size=batch_size, dtype=tf.float32)
X = np.array([[[1, 2, 1], [2, 0, 0], [2, 1, 0], [1, 1, 0]],  # x1
              [[1, 2, 1], [2, 0, 0], [2, 1, 0], [1, 1, 0]],  # x2
              [[1, 2, 1], [2, 0, 0], [2, 1, 0], [1, 1, 0]]])  # x3
outputs, final_state = tf.nn.dynamic_rnn(cell, inputs, initial_state=h0, time_major=True)

sess = tf.Session()
sess.run(tf.global_variables_initializer())
a, b = sess.run([outputs, final_state], feed_dict={inputs:X})
print(a)
print(b)
```

3.1.3 time_step

具体解释如下：

> 文字数据
>
> 如果数据有1000段时序的句子，每句话有25个字，对每个字进行向量化，每个字的向量维度为300，那么batch_size=1000，time_steps=25，input_size=300。
>
> 解析：time_steps一般情况下就是等于句子的长度，input_size等于字量化后向量的长度。
>
> 图片数据
>
> 拿MNIST手写数字集来说，训练数据有6000个手写数字图像，每个数字图像大小为28*28，batch_size=6000没的说，time_steps=28，input_size=28，我们可以理解为把图片图片分成28份，每份shape=(1, 28)。
>
> 音频数据
>
> 如果是单通道音频数据，那么音频数据是一维的，假如shape=(8910,)。使用RNN的数据必须是二维的，这样加上batch_size，数据就是三维的，第一维是batch_size，第二维是time_steps，第三位是数据input_size。我们可以把数据reshape成三维数据。这样就能使用RNN了。

3.2 如何循环调用

dnn有static和dynamic的分别。

- static_rnn会把RNN展平，用空间换时间。
- dynamic_rnn则是使用for或者while循环。

调用static_rnn实际上是生成了rnn按时间序列展开之后的图。打开tensorboard你会看到sequence_length个rnn_cell stack在一起，只不过这些cell是share weight的。因此，sequence_length就和图的拓扑结构绑定在了一起，因此也就限制了每个batch的sequence_length必须是一致。

调用dynamic_rnn不会将rnn展开，而是利用tf.while_loop这个api，通过Enter, Switch, Merge, LoopCondition, NextIteration等这些control flow的节点，生成一个可以执行循环的图（这个图应该还是静态图，因为图的拓扑结构在执行时是不会变化的）。在tensorboard上，你只会看到一个rnn_cell, 外面被一群control flow节点包围着。对于

dynamic_rnn来说，sequence_length仅仅代表着循环的次数，而和图本身的拓扑没有关系，所以每个batch可以有不同sequence_length。

对于DIEN，程序运行时候，堆栈如下：

```
call, utils.py:144
__call__, utils.py:114
<lambda>, rnn.py:752
_rnn_step, rnn.py:236
_time_step, rnn.py:766
_BuildLoop, control_flow_ops.py:2590
BuildLoop, control_flow_ops.py:2640
while_loop, control_flow_ops.py:2816
_dynamic_rnn_loop, rnn.py:786
dynamic_rnn, rnn.py:615
__init__, model.py:364
train, train.py:142
<module>, train.py:231
```

循环的实现主要是在 control_flow_ops.py 之中。

while_loop 会 在 cond 参数为true时候，一直循环 body 参数对应的代码。

```
def while_loop(cond, body, loop_vars, shape_invariants=None,
               parallel_iterations=10, back_prop=True, swap_memory=False,
               name=None):
  """Repeat `body` while the condition `cond` is true.

  `cond` is a callable returning a boolean scalar tensor. `body` is a callable
  returning a (possibly nested) tuple, namedtuple or list of tensors of the same
  arity (length and structure) and types as `loop_vars`. `loop_vars` is a
  (possibly nested) tuple, namedtuple or list of tensors that is passed to both
  `cond` and `body`. `cond` and `body` both take as many arguments as there are
  `loop_vars`.

  Args:
    cond: A callable that represents the termination condition of the loop.
    body: A callable that represents the loop body.
    loop_vars: A (possibly nested) tuple, namedtuple or list of numpy array,
      `Tensor`, and `TensorArray` objects.
```

```
  """
    if context.in_eager_mode():
      while cond(*loop_vars):
        loop_vars = body(*loop_vars)
      return loop_vars

    if shape_invariants is not None:
      nest.assert_same_structure(loop_vars, shape_invariants)

    loop_context = WhileContext(parallel_iterations, back_prop, swap_memory)  # pylint: disable=r
    ops.add_to_collection(ops.GraphKeys.WHILE_CONTEXT, loop_context)
    result = loop_context.BuildLoop(cond, body, loop_vars, shape_invariants)
    return result
```

比如如下例子：

```
i = tf.constant(0)
c = lambda i: tf.less(i, 10)
b = lambda i: tf.add(i, 1)
r = tf.while_loop(c, b, [i])
print(sess.run(r) ) # 10
```

在rnn中，_time_step 就对 while_loop 进行了调用，这样就完成了迭代。

```
    _, output_final_ta, final_state = control_flow_ops.while_loop(
        cond=lambda time, *_: time < time_steps,
        body=_time_step,
        loop_vars=(time, output_ta, state),
        parallel_iterations=parallel_iterations,
        swap_memory=swap_memory)
```

3.3. DIEN之rnn

DIEN项目中，修改的部分主要是_time_step函数，因为需要加入att_scores参数。

其主要是：

- 通过 `lambda: cell(input_t, state, att_score)` 调用 cell # call 函数，即我们事先写的业务逻辑；
- 通过调用 `control_flow_ops.while_loop(cond=lambda time, *_: time < time_steps, body=_time_step...)` 来进行循环迭代；

缩减版代码如下：

```
def _time_step(time, output_ta_t, state, att_scores=None):
  """Take a time step of the dynamic RNN.
  Args:
    time: int32 scalar Tensor.
    output_ta_t: List of `TensorArray`s that represent the output.
    state: nested tuple of vector tensors that represent the state.

  Returns:
    The tuple (time + 1, output_ta_t with updated flow, new_state).
  """

  ......

  if att_scores is not None:
      att_score = att_scores[:, time, :]
      call_cell = lambda: cell(input_t, state, att_score)
  else:
      call_cell = lambda: cell(input_t, state)

  ......

  output_ta_t = tuple(
      ta.write(time, out) for ta, out in zip(output_ta_t, output))

  if att_scores is not None:
      return (time + 1, output_ta_t, new_state, att_scores)
  else:
      return (time + 1, output_ta_t, new_state)

if att_scores is not None:
    _, output_final_ta, final_state, _ = control_flow_ops.while_loop(
        cond=lambda time, *_: time < time_steps,
        body=_time_step,
        loop_vars=(time, output_ta, state, att_scores),
        parallel_iterations=parallel_iterations,
        swap_memory=swap_memory)
else:
    _, output_final_ta, final_state = control_flow_ops.while_loop(
        cond=lambda time, *_: time < time_steps,
        body=_time_step,
        loop_vars=(time, output_ta, state),
        parallel_iterations=parallel_iterations,
        swap_memory=swap_memory)
```

.....

0xFF 参考

通过代码学习RNN，彻底弄懂time_step

LSTM 实际神经元隐含层物理架构原理解析

机器学习之LSTM

知乎-何之源：TensorFlow中RNN实现的正确打开方式

解读tensorflow之rnn

char-rnn-tensorflow源码解析及结构流程分析

循环神经网络（LSTM和GRU）(2)

TensorFlow中RNN实现的正确打开方式

完全图解RNN、RNN变体、Seq2Seq、Attention机制

Tensorflow中RNNCell源码解析

tensorflow中RNNcell源码分析以及自定义RNNCell的方法

Tensorflow rnn_cell api 阅读笔记

循环神经网络系列（一）Tensorflow中BasicRNNCell

循环神经网络系列（二）Tensorflow中dynamic_rnn

LSTM中tf.nn.dynamic_rnn处理过程详解

小白循环神经网络RNN LSTM 参数数量 门单元 cell units timestep batch_size

tensorflow笔记：多层LSTM代码分析

Tensorflow dynamic rnn，源代码的逐行解读

Tensorflow RNN源码理解

Tensorflow RNN源代码解析笔记1：RNNCell的基本实现

Tensorflow RNN源代码解析笔记2：RNN的基本实现