变长序列的处理

参考链接1
参考链接2

设批次为B，序列最大长度为L，输入词向量维度为d，隐藏层和输出维度为D
假如B=3，L=4，d=4，D=3
第一条句子4个词，第二条句子3个词，第三条句子2个词
那么如何把这3个句子交给rnn一次性处理呢?
通常有两种做法

In [1]:
```
import torch
```

批处理方式1:

先padding，再通过rnn批计算，再mask

计算步骤

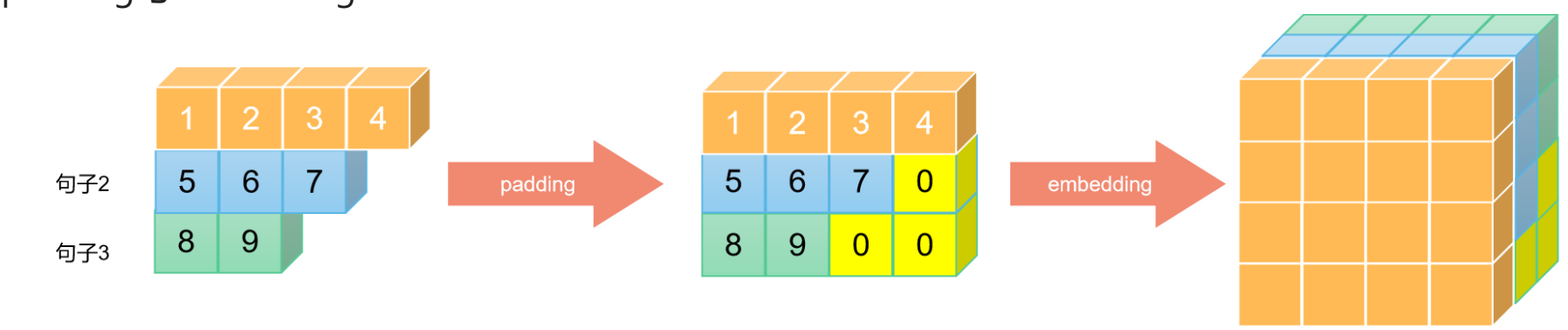
第一步：padding

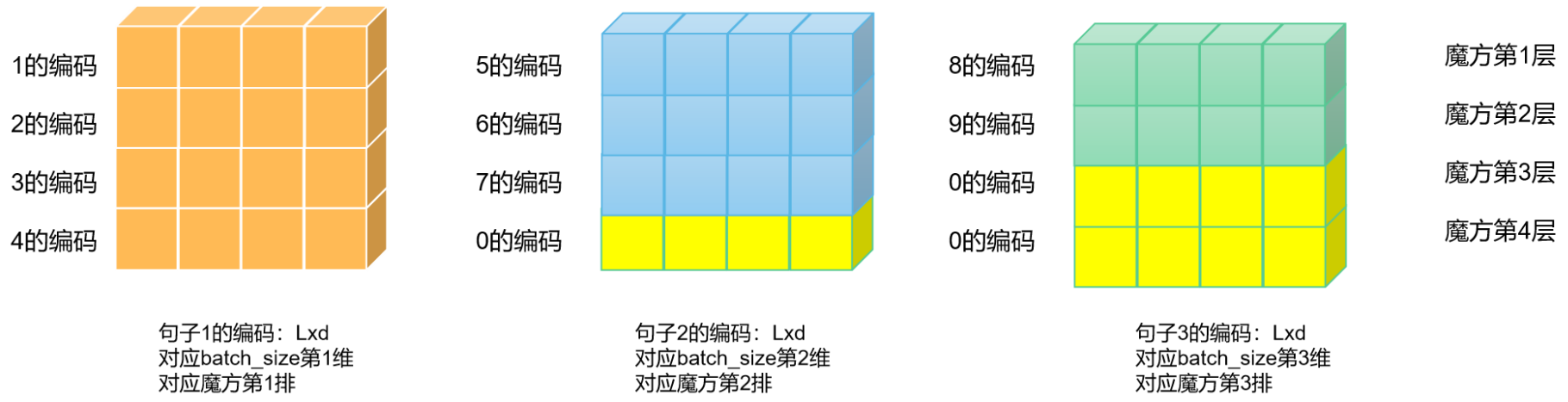
首先3句话通过collate_fn变成等长的，比如:
第一句话=[1,2,3,4],第二句话=[5,6,7]，第三句话=[8,9]
三句话batch_data=[[1,2,3,4], [5,6,7], [8,9]]
pad之后变成batch_data=[[1,2,3,4], [5,6,7,0], [8,9,0,0]]，形状为：BxL=3x4

第二步：embedding

接着这个BxL的张量通过embedding层，根据每个单词id对单词进行编码，
于是这句话中每个单词从数字id变成了4维的向量，整句话有L=4个单词长度，
每个单词是4维向量，于是整句话就是Lxd，也就是4x4.
然后这里有3句话，所以编码后形状就是3xLxd，这里的3就是batch_size
所以经过embedding层后，从BxL变成了BxLxd

padding与embedding





句子1的编码：Lxd
对应batch_size第1维
对应魔方第1排

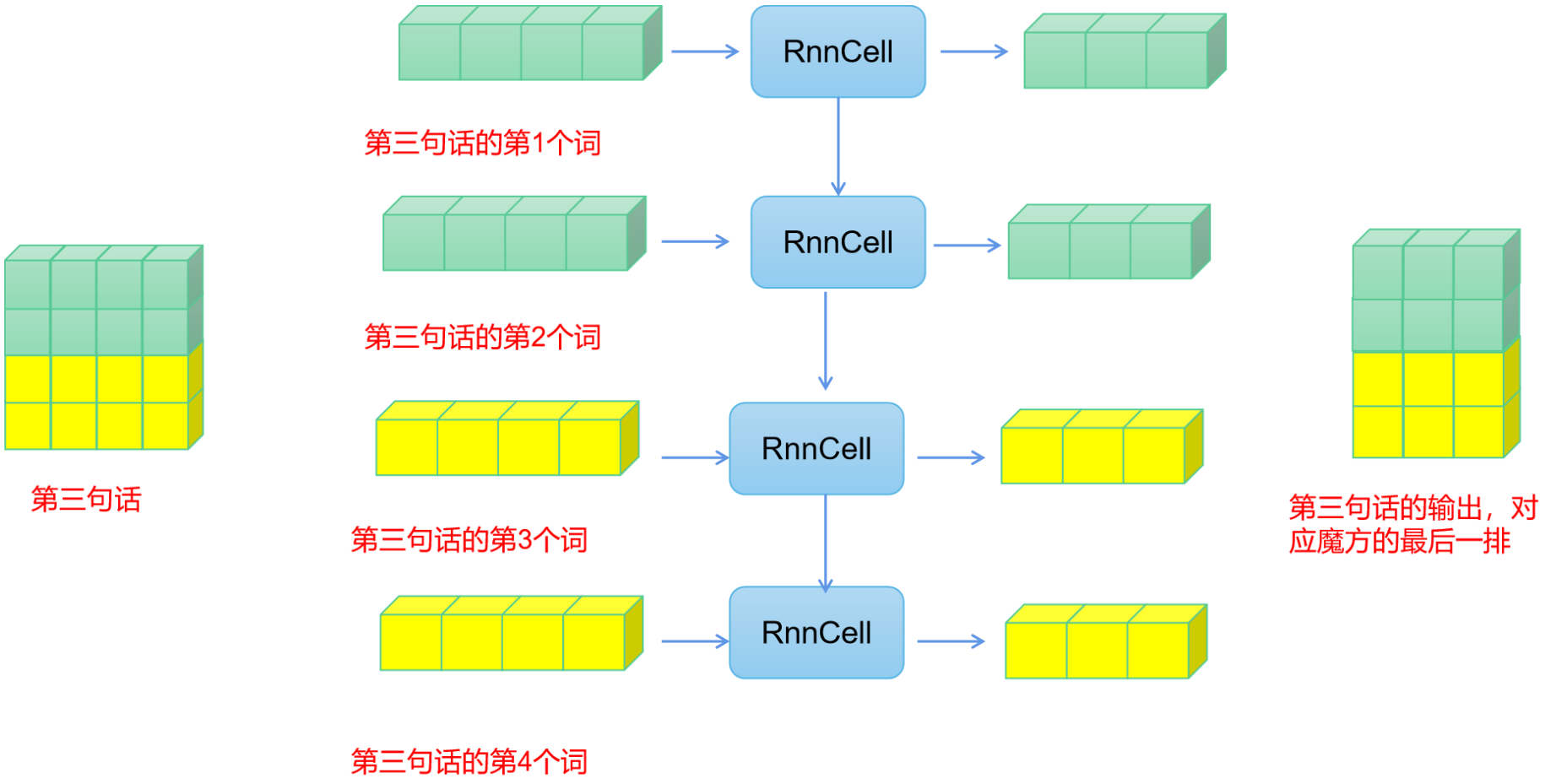
句子2的编码：Lxd
对应batch_size第2维
对应魔方第2排

句子3的编码：Lxd
对应batch_size第3维
对应魔方第3排

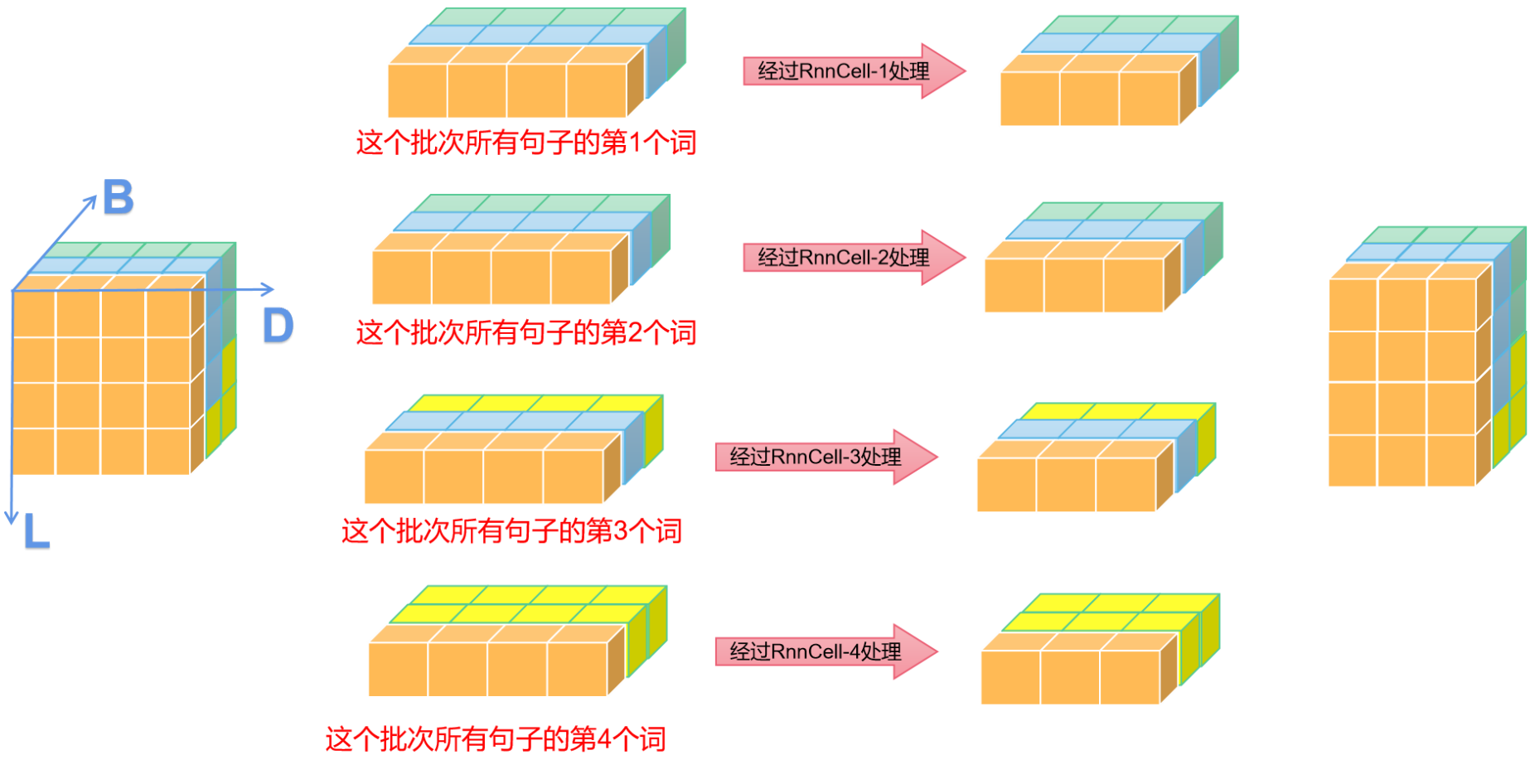
第三步：RNN

接着就是BxLxd这个魔方经过RNN层，具体计算流程如下：
step1：先令L=1，取出3句话的第一个词，形状为3xd，,交给rnncell分别计算第一个词， 输出形状为3xD
step2：L=2,rnncell计算3句话的第二个词，形状为3xd，输出形状为3xD
step3：L=3,rnncell计算3句话的第三个词，形状为3xd，输出形状为3xD
step4：L=4,rnncell计算3句话的第四个词，形状为3xd，输出形状为3xD
正如上图所示，每次仅仅是固定L，这个L表示单词的长度，这里统一成了3，有的地方也把L写成T，表示时间序列的时间步，比如T=t时刻的输入，就是当T=t时，上面魔方的一层Bxd矩阵
然后维度为d的B个词，通过rnncell之后，转换成了维度为D的B个词，这是并行处理的，互不干扰
最后将上面4个时间步输出的BxD方块重新组装成BxLxD的魔方，这就是最终的输出
小结：BxLxd通过RNN，输出BxLxD

简单看看第三句话的计算流程



批处理计算流程



第四步：mask

最后需要做的就是把BxLxD中用pad计算的部分掩码抹为0，从下图可以看出，第三句话总共就2个词，rnn的前两步实际就已经把这句话处理完毕了，后面继续处理pad的部分完全是多余的，因此我们需要将BxLxD中的黄色向量变成0

掩码示例：

矩阵大小：BxLxD=3x4x3
3组句子，最大句子长4，维度3
真实句子长度：[4,3,2]
处理模板：unsqueeze(-1)，比较，unsqueeze(-1)

In [2]:
```
x=torch.randn([3,4,3])
seq_len=torch.tensor([4,3,2])
max_len=x.shape[1]
```

In [3]:
```
seq_len=seq_len.unsqueeze(-1).to(dtype=torch.int64)
seq_len
```

Out[3]:
```
tensor([[4],
        [3],
        [2]])
```

In [4]:
```
mask=torch.arange(max_len)<seq_len
mask
```

Out[4]:
```
tensor([[ True,  True,  True,  True],
        [ True,  True,  True, False],
        [ True,  True, False, False]])
```

In [5]:
```
mask=mask.unsqueeze(-1)
mask
```

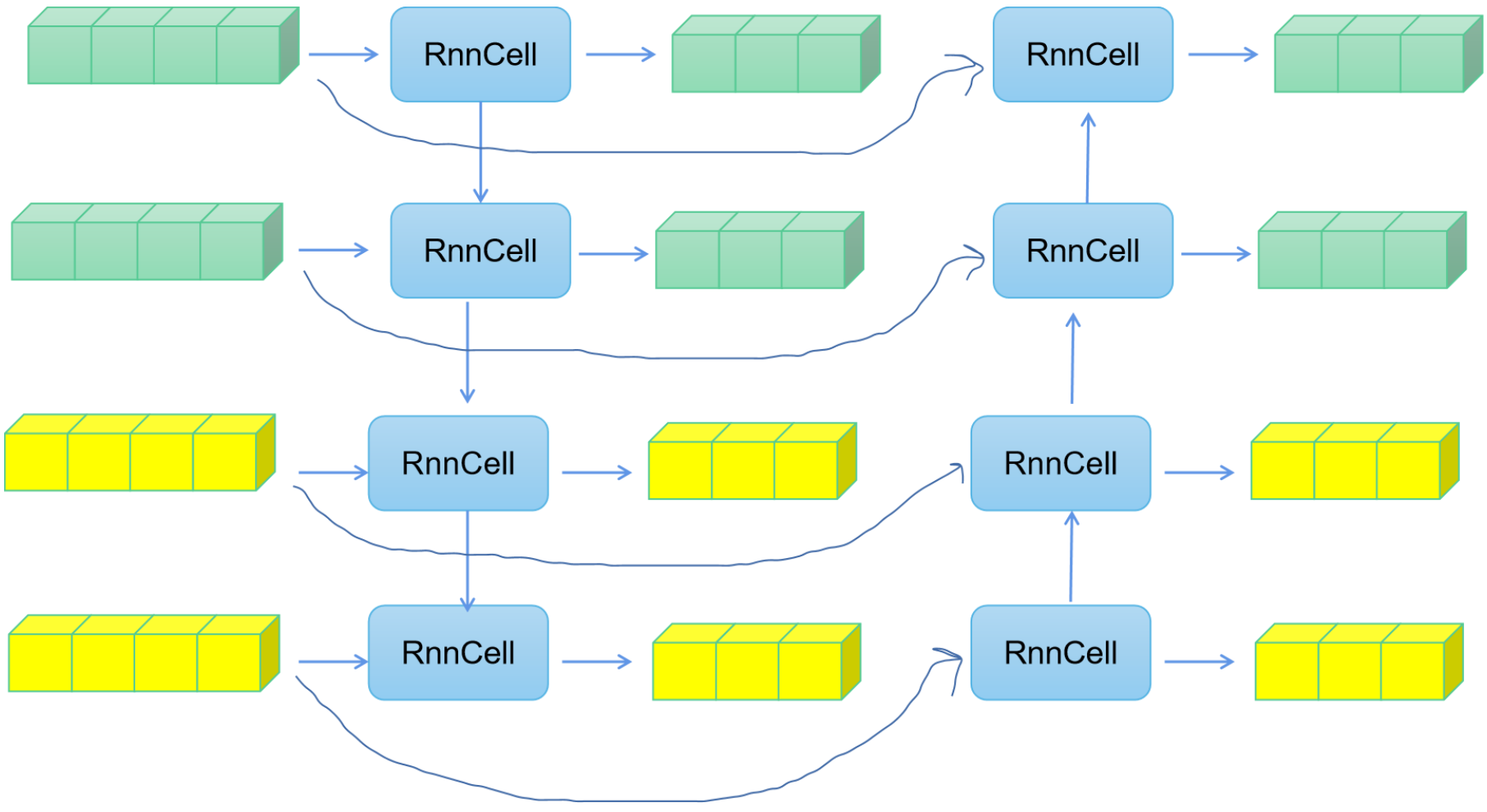
Out[5]:
```
tensor([[[ True],
         [ True],
         [ True],
         [ True]],

        [[ True],
         [ True],
         [ True],
         [False]],

        [[ True],
         [ True],
         [False],
         [False]]])
```

In [6]:
```
xx=torch.multiply(x,mask)
print("BxLxD矩阵： ",x,"\n","掩码后： ",xx)
```

```
BxLxD矩阵：  tensor([[[ 2.4934,  0.3349, -1.2078],
         [-1.2459,  1.0282, -0.6050],
         [ 0.4498,  1.3841,  0.2413],
         [-0.8257,  0.1591, -0.2787]],

        [[ 0.3098,  0.5086, -0.8092],
         [-1.3379, -1.0601,  0.0536],
         [ 0.3070, -0.1175, -1.3581],
         [-0.5746, -0.5089, -0.7666]],

        [[ 0.5134, -1.8761, -0.4032],
         [ 0.1114,  1.6643, -0.2592],
         [ 0.1112,  0.4480, -0.0439],
         [-1.4352,  0.3000, -0.7043]]])
掩码后：  tensor([[[ 2.4934,  0.3349, -1.2078],
         [-1.2459,  1.0282, -0.6050],
         [ 0.4498,  1.3841,  0.2413],
         [-0.8257,  0.1591, -0.2787]],

        [[ 0.3098,  0.5086, -0.8092],
         [-1.3379, -1.0601,  0.0536],
         [ 0.3070, -0.1175, -1.3581],
         [-0.0000, -0.0000, -0.0000]],

        [[ 0.5134, -1.8761, -0.4032],
         [ 0.1114,  1.6643, -0.2592],
         [ 0.0000,  0.0000, -0.0000],
         [-0.0000,  0.0000, -0.0000]]])
```

缺点：

缺点1：计算慢，pad字符的编码每次都会参与计算
缺点2：在包含反向rnn的结构中，计算存在误差，反向计算从pad字符开始，
即使pad编码设置成0，但是初始参数里面有个偏置不为0，通过rnn后计算出来的
隐藏状态也不是0，会参与后面真实单词的计算



批处理方式2：

首先必须填充为BxLxD的格式，然后每步按照实际能取到的词来批量计算
step1：rnncell计算3句话的第一个词，从batch维拿到了3个词，形状为3xd，
step2：rnncell计算3句话的第二个词，从batch维拿到了3个词，形状为3xd，
step3：rnncell计算3句话的第三个词，由于最后一句话没有第三个词，所以形状为2xd，
step4：rnncell计算3句话的第四个词，从batch维拿到了1个词，形状为1xd，
所以每次取出的Bxd大小都不一样，分别为B1xd,B2xd,B3xd,B4xd

每步的输出形状为：B1xD,B2xD,B3xD,B4xD
所以pad后，需要知道每步能实际取到的batch_size=[B1,B2,B3,B4]

二维矩阵的填充，压缩，恢复

In [7]:
```
from torch.nn.utils.rnn import pad_sequence
```

In [8]:
```
x=[torch.tensor([1,2,3,4]),torch.tensor([5,6,7]),torch.tensor([8,9])]
x_pad=pad_sequence(x,batch_first=True)
x_pad
```

Out[8]:
```
tensor([[1, 2, 3, 4],
        [5, 6, 7, 0],
        [8, 9, 0, 0]])
```

In [9]:
```
from torch.nn.utils.rnn import pack_padded_sequence
```

In [10]:
```
x_pack=pack_padded_sequence(x_pad,lengths=[4,3,2],batch_first=True)
x_pack
```

Out[10]:
```
PackedSequence(data=tensor([1, 5, 8, 2, 6, 9, 3, 7, 4]), batch_sizes=tensor([3, 3, 2, 1]), sorted_indices=None, unsorted_indices=None)
```

In [11]:
```
from torch.nn.utils.rnn import pack_sequence
```

In [12]:
```
x_pack2=pack_sequence(x)
x_pack2
```

Out[12]:
```
PackedSequence(data=tensor([1, 5, 8, 2, 6, 9, 3, 7, 4]), batch_sizes=tensor([3, 3, 2, 1]), sorted_indices=None, unsorted_indices=None)
```

In [13]:
```
from torch.nn.utils.rnn import pad_packed_sequence
```

In [14]:
```
x_rebuild=pad_packed_sequence(x_pack,batch_first=True)
x_rebuild
```

Out[14]:
```
(tensor([[1, 2, 3, 4],
         [5, 6, 7, 0],
         [8, 9, 0, 0]]),
 tensor([4, 3, 2]))
```

三维矩阵的填充，压缩，恢复

In [15]:
```
x1=torch.arange(12).reshape(4,-1)
x2=torch.arange(12,21).reshape(3,-1)
x3=torch.arange(21,27).reshape(2,-1)
x=[x1,x2,x3]
x
```

Out[15]:
```
[tensor([[ 0,  1,  2],
         [ 3,  4,  5],
         [ 6,  7,  8],
         [ 9, 10, 11]]),
 tensor([[12, 13, 14],
         [15, 16, 17],
         [18, 19, 20]]),
 tensor([[21, 22, 23],
         [24, 25, 26]])]
```

In [16]:
```
x_pad=pad_sequence(x,batch_first=True)
x_pad
```

Out[16]:
```
tensor([[[ 0,  1,  2],
         [ 3,  4,  5],
         [ 6,  7,  8],
         [ 9, 10, 11]],

        [[12, 13, 14],
         [15, 16, 17],
         [18, 19, 20],
         [ 0,  0,  0]],

        [[21, 22, 23],
         [24, 25, 26],
         [ 0,  0,  0],
         [ 0,  0,  0]]])
```

In [17]:
```
x_pack=pack_padded_sequence(x_pad,lengths=[4,3,2],batch_first=True)
x_pack
```

Out[17]:
```
PackedSequence(data=tensor([[ 0,  1,  2],
        [12, 13, 14],
        [21, 22, 23],
        [ 3,  4,  5],
        [15, 16, 17],
        [24, 25, 26],
        [ 6,  7,  8],
        [18, 19, 20],
        [ 9, 10, 11]]), batch_sizes=tensor([3, 3, 2, 1]), sorted_indices=None, unsorted_indices=None)
```

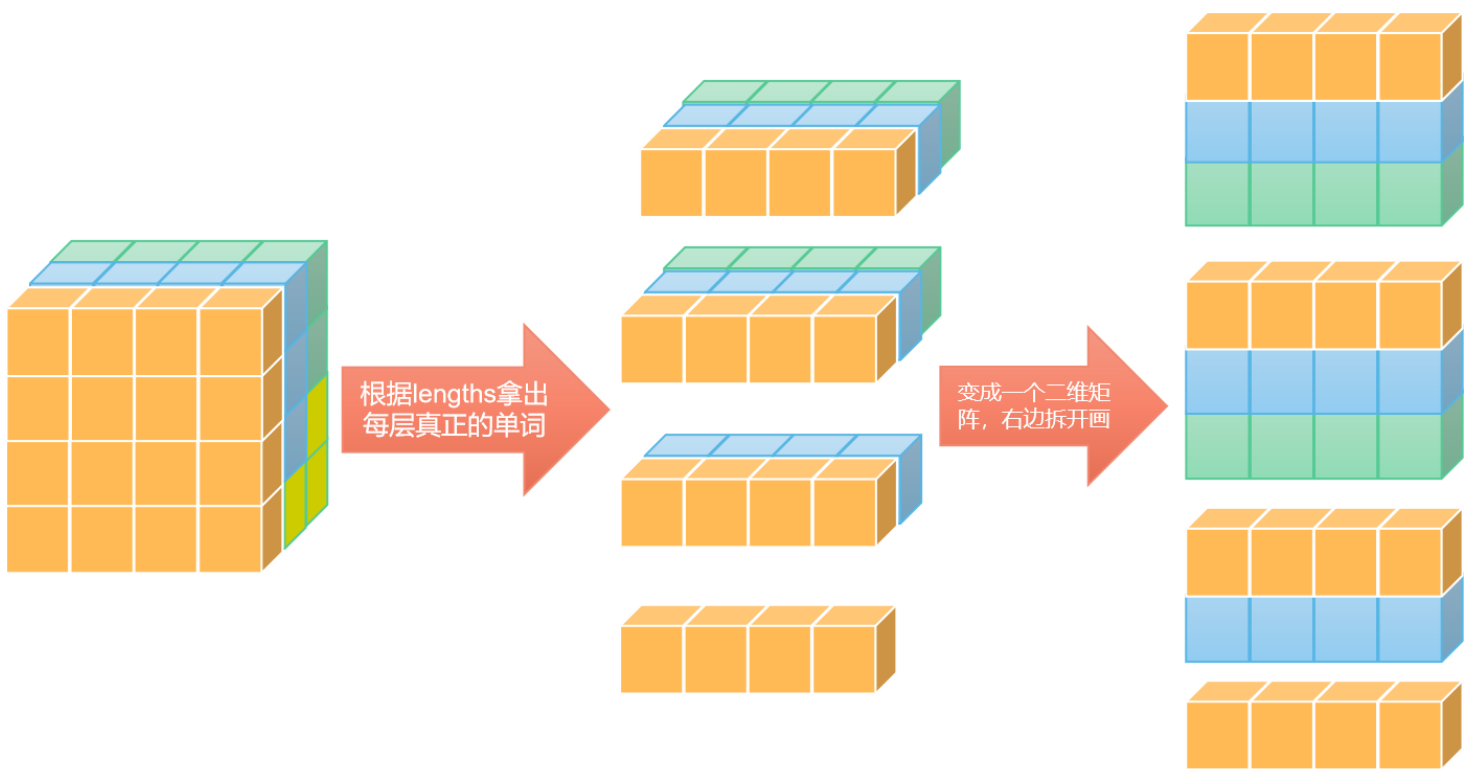
In [18]:
```
x_rebuild=pad_packed_sequence(x_pack,batch_first=True)
x_rebuild
```

Out[18]:
```
(tensor([[[ 0,  1,  2],
          [ 3,  4,  5],
          [ 6,  7,  8],
          [ 9, 10, 11]],

         [[12, 13, 14],
          [15, 16, 17],
          [18, 19, 20],
          [ 0,  0,  0]],

         [[21, 22, 23],
          [24, 25, 26],
          [ 0,  0,  0],
          [ 0,  0,  0]]]),
 tensor([4, 3, 2]))
```

序列处理流程：

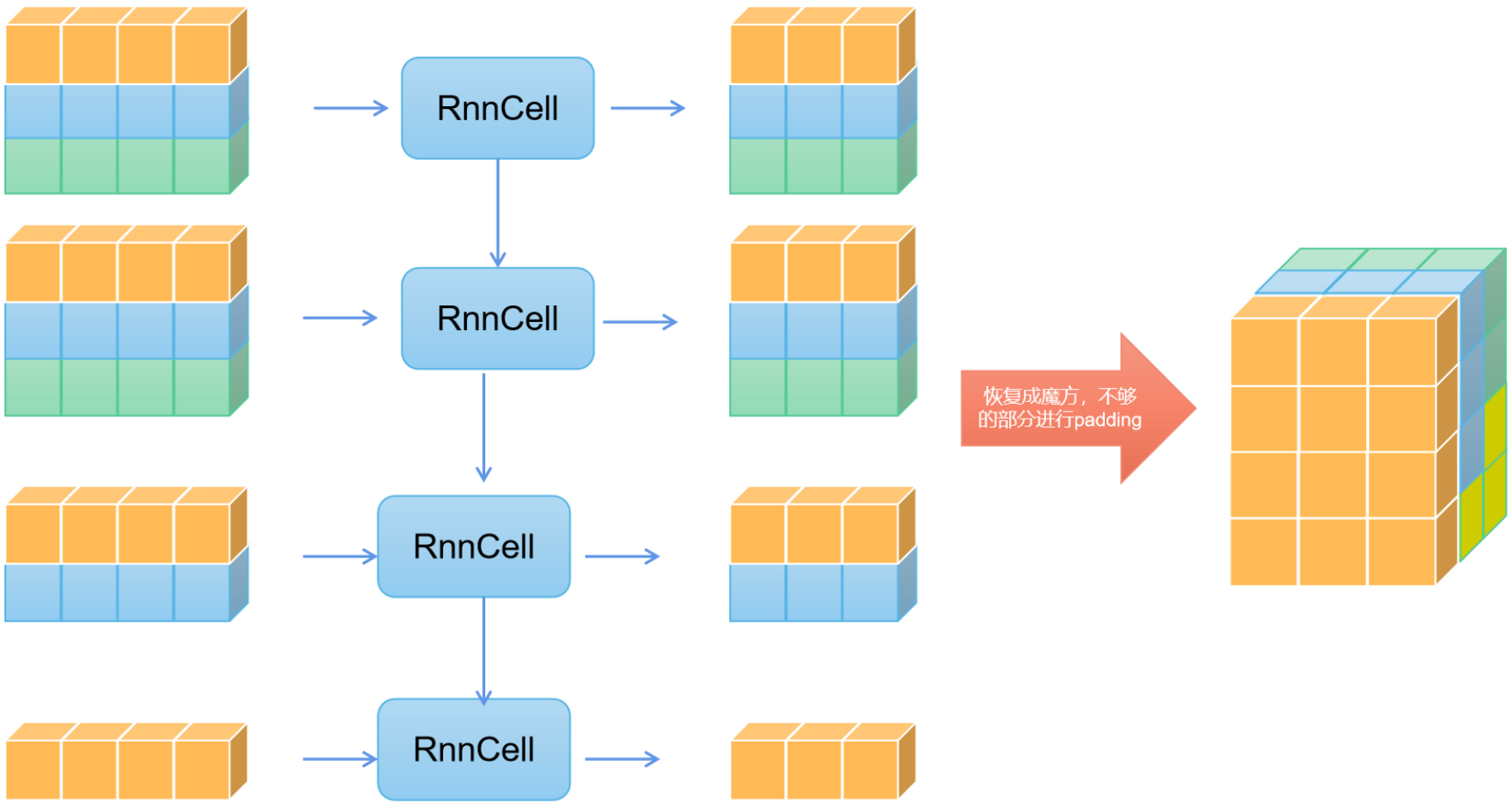
①.在dataloader中对序列id进行padding,然后经过embedding,获得BxLxd的张量,这步与批处理方案一没有什么不同
②.对BxLxd进行pack压缩处理,获得一个二维张量，就是魔方的1，2，3，4层排列下来，让Rnn根据PackedSequence的batch_sizes参数分别取出魔方的1，2，3，4层处理，处理完后还是二维张量，再恢复成三维的即可，每层不够的部分padding补齐

对输出进行pack



batch_sizes=tensor([3, 3, 2, 1])

对输出进行pad



batch_sizes=tensor([3, 3, 2, 1])

注意，batch_sizes[i]决定了第i个时间步的输入的形状， 拿第三个时间步的RnnCell举例，x_3就是从pack后的那个二维矩阵中拿两行， 从batch_sizes[0]+batch_sizes[1]=6行开始拿，h直接从上一个 时间步的输出h拿两行，作为此时间步也就是第三步RnnCell的输入 实际意义解释：还是从第二步到第三步，观察h的第三行，其实就是第三个 句子在这个时间步的处理结果，由于第三个句子只有两个词，所以经过 两个时间步就已经处理完毕了，第三个RnnCell跟第三个句子已经没关系了 上面三种颜色，就是三条句子每个词的计算过程。

使用单向单层的LSTM进行试验

注意事项：batch_first=True

批处理方式1：

丢进去BxLxd，出来BxLxD，再进行掩码
输入输出格式都是3维tensor
还是B=3，L=4，d=4，D=3
第一条句子4个词，第二条句子3个词，第三条句子2个词

构造输入，h，c

In [19]:
```
x=torch.tensor([[[ 0,  1,  2,  11],
         [ 3,  4,  5,  11],
         [ 6,  7,  8,  11],
         [ 9, 10, 11,  11]],

        [[12, 13, 14,  11],
         [15, 16, 17,  11],
         [18, 19, 20,  11],
         [ 0,  0,  0,  0]],

        [[21, 22, 23,  11],
         [24, 25, 26,  11],
         [ 0,  0,  0,  0],
         [ 0,  0,  0,  0]]]).to(torch.float)
x.shape
```

Out[19]:
```
torch.Size([3, 4, 4])
```

In [20]:
```
h=torch.zeros(size=[1,3,3])  # shape=[N,B,D] , BxD输出魔方每层的形状，N就是多少层lstm，如果是双向的就是2N，因为每层都需要一个h作为输入
c=torch.zeros(size=[1,3,3])  # shape=[N,B,D]
```

构造模型

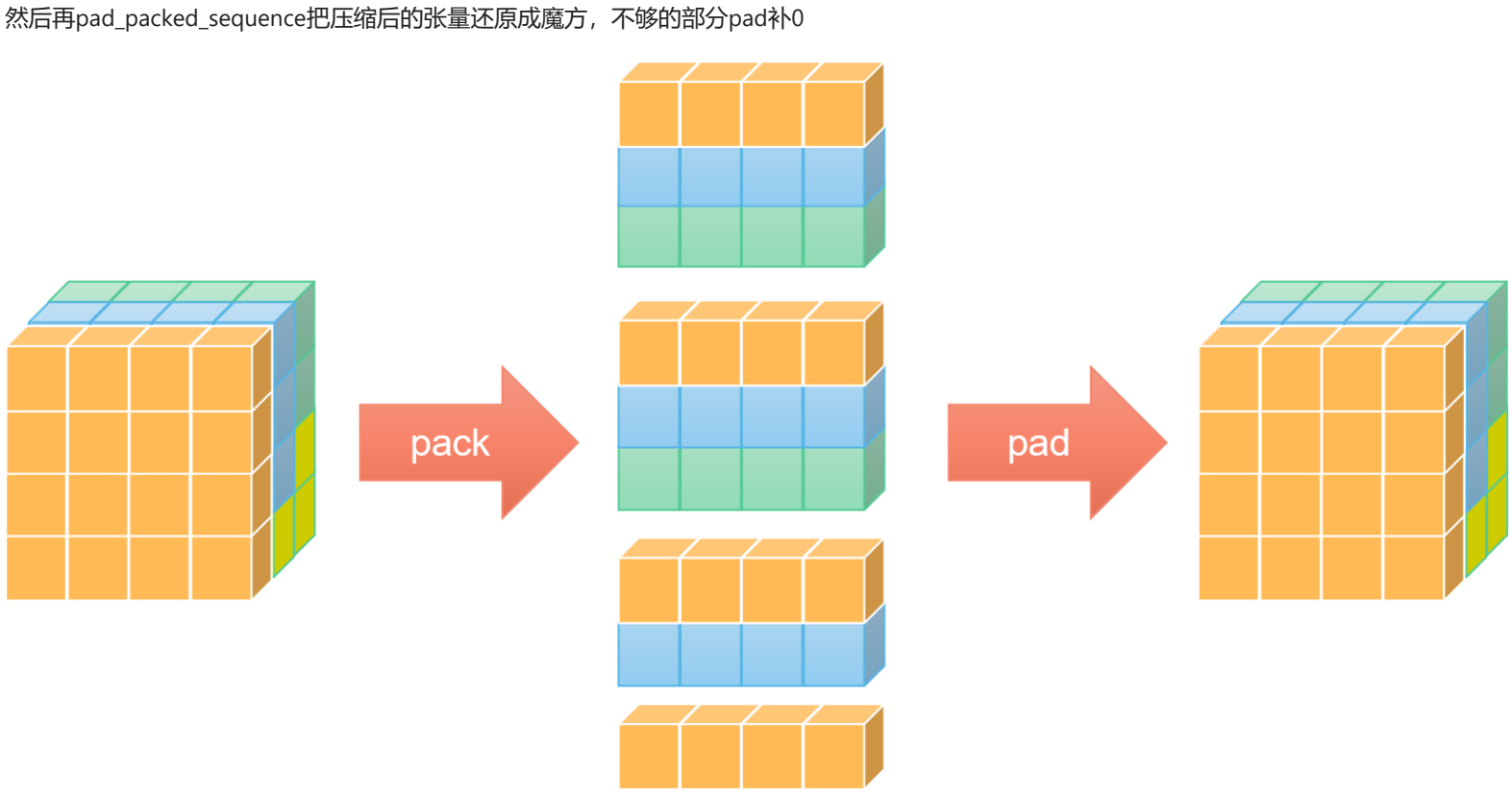
In [21]:
```
lstm=torch.nn.LSTM(input_size=4,hidden_size=3,batch_first=True)
```

计算，输出

In [22]:
```
out,(h_,c_)=lstm(x,(h,c))
out.shape,out
```

Out[22]:
```
(torch.Size([3, 4, 3]),
 tensor([[[-3.3317e-02, -1.4542e-03,  1.4459e-01],
          [-1.1001e-01, -2.4086e-02,  2.0124e-01],
          [-2.7444e-01, -1.6108e-02,  2.6300e-01],
          [-5.5577e-01, -9.6089e-03,  3.2836e-01]],

         [[-5.7880e-01, -6.8107e-03,  3.9326e-01],
          [-8.9280e-01, -3.2349e-03,  4.2472e-01],
          [-9.8095e-01, -1.7532e-03,  4.3782e-01],
          [-3.3405e-01, -2.1133e-01,  1.1036e-01]],

         [[-7.4863e-01, -1.5322e-03,  4.9750e-01],
          [-9.6048e-01, -6.6851e-04,  4.4659e-01],
          [-3.1894e-01, -2.1254e-01,  1.3858e-01],
          [-2.6318e-01, -1.4879e-01,  7.5238e-02]]],
       grad_fn=<TransposeBackward0>))
```

对输出掩码，可以采用之前unsqueeze(-1),比较,unsqueeze(-1)的方法，但是麻烦，这里先压缩再pad回
思路：先用pack_padded_sequence取出每个句子该有的单词，这样就不会取出pad对应的行，
然后再pad_packed_sequence把压缩后的张量还原成魔方，不够的部分pad补0



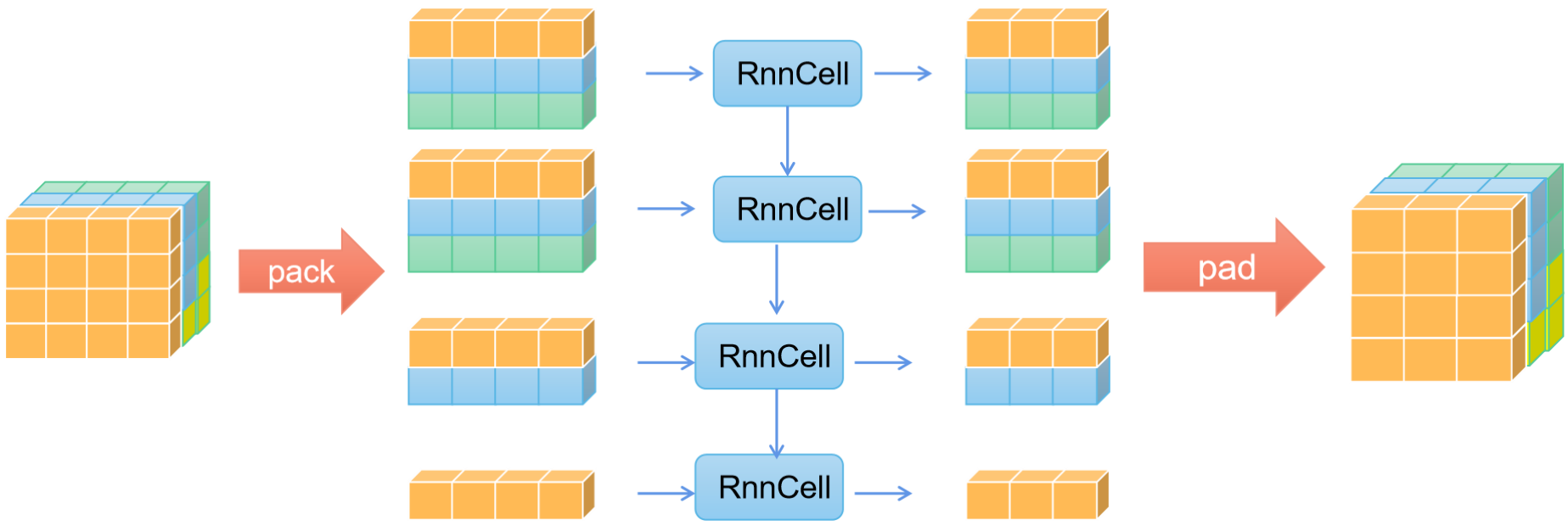
In [23]:
```
padded_out,seq_len=pad_packed_sequence(pack_padded_sequence(out,lengths=[4,3,2],batch_first=True),batch_first=True)
out,padded_out
```

Out[23]:
```
(tensor([[[-3.3317e-02, -1.4542e-03,  1.4459e-01],
          [-1.1001e-01, -2.4086e-02,  2.0124e-01],
          [-2.7444e-01, -1.6108e-02,  2.6300e-01],
          [-5.5577e-01, -9.6089e-03,  3.2836e-01]],

         [[-5.7880e-01, -6.8107e-03,  3.9326e-01],
          [-8.9280e-01, -3.2349e-03,  4.2472e-01],
          [-9.8095e-01, -1.7532e-03,  4.3782e-01],
          [-3.3405e-01, -2.1133e-01,  1.1036e-01]],

         [[-7.4863e-01, -1.5322e-03,  4.9750e-01],
          [-9.6048e-01, -6.6851e-04,  4.4659e-01],
          [-3.1894e-01, -2.1254e-01,  1.3858e-01],
          [-2.6318e-01, -1.4879e-01,  7.5238e-02]]],
        grad_fn=<TransposeBackward0>),
 tensor([[[-3.3317e-02, -1.4542e-03,  1.4459e-01],
          [-1.1001e-01, -2.4086e-02,  2.0124e-01],
          [-2.7444e-01, -1.6108e-02,  2.6300e-01],
          [-5.5577e-01, -9.6089e-03,  3.2836e-01]],

         [[-5.7880e-01, -6.8107e-03,  3.9326e-01],
          [-8.9280e-01, -3.2349e-03,  4.2472e-01],
          [-9.8095e-01, -1.7532e-03,  4.3782e-01],
          [ 0.0000e+00,  0.0000e+00,  0.0000e+00]],

         [[-7.4863e-01, -1.5322e-03,  4.9750e-01],
          [-9.6048e-01, -6.6851e-04,  4.4659e-01],
          [ 0.0000e+00,  0.0000e+00,  0.0000e+00],
          [ 0.0000e+00,  0.0000e+00,  0.0000e+00]]],
        grad_fn=<TransposeBackward0>))
```

批处理方式2：

BxLxd转化为packed_sequence丢进去，出来packed_sequence再转为BxLxD

输入输出格式都是2维PackedSequence



pack

In [24]:
```
packed_input=pack_padded_sequence(x,lengths=[4,3,2],batch_first=True)
```

lstm处理

In [25]:
```
packed_output,(h_out,c_out)=lstm(packed_input,(h,c))
```

pad

In [26]:
```
padded_out,seq_len=pad_packed_sequence(packed_output,batch_first=True)
out,padded_out
```

Out[26]:
```
(tensor([[[-3.3317e-02, -1.4542e-03,  1.4459e-01],
          [-1.1001e-01, -2.4086e-02,  2.0124e-01],
          [-2.7444e-01, -1.6108e-02,  2.6300e-01],
          [-5.5577e-01, -9.6089e-03,  3.2836e-01]],

         [[-5.7880e-01, -6.8107e-03,  3.9326e-01],
          [-8.9280e-01, -3.2349e-03,  4.2472e-01],
          [-9.8095e-01, -1.7532e-03,  4.3782e-01],
          [-3.3405e-01, -2.1133e-01,  1.1036e-01]],

         [[-7.4863e-01, -1.5322e-03,  4.9750e-01],
          [-9.6048e-01, -6.6851e-04,  4.4659e-01],
          [-3.1894e-01, -2.1254e-01,  1.3858e-01],
          [-2.6318e-01, -1.4879e-01,  7.5238e-02]]],
        grad_fn=<TransposeBackward0>),
 tensor([[[-3.3317e-02, -1.4542e-03,  1.4459e-01],
          [-1.1001e-01, -2.4086e-02,  2.0124e-01],
          [-2.7444e-01, -1.6108e-02,  2.6300e-01],
          [-5.5577e-01, -9.6089e-03,  3.2836e-01]],

         [[-5.7880e-01, -6.8107e-03,  3.9326e-01],
          [-8.9280e-01, -3.2349e-03,  4.2472e-01],
          [-9.8095e-01, -1.7532e-03,  4.3782e-01],
          [ 0.0000e+00,  0.0000e+00,  0.0000e+00]],

         [[-7.4863e-01, -1.5322e-03,  4.9750e-01],
          [-9.6048e-01, -6.6851e-04,  4.4659e-01],
          [ 0.0000e+00,  0.0000e+00,  0.0000e+00],
          [ 0.0000e+00,  0.0000e+00,  0.0000e+00]]],
        grad_fn=<TransposeBackward0>))
```