这是在本项目进行过程中的学习记录和总结。项目过程中需要用到的相关知识，处理细节等记录如下：

RNN

- 最初用来训练**语言模型**
- 循环神经网络的每一时间步隐藏层不仅与输入有关， 而且与前一时间步的隐藏层有关。
- 前向RNN只能记前面的东西，而双向RNN来捕捉前后的信息。
- RNN只有一套参数。
- **RNN的训练和BPTT：**假设参数U是输入和隐藏层的权值矩阵，W是上一个时间步和本个时间步的权值矩阵。那么，前向传播时，每个时间步计算误差，$L1$，$L2$，$L3$，$\ldots$，Ln，n为句子长度。注意有结束符$< EOS >$。计算总误差：$L = L1 + L2 + L3 + L4 + L5 + L6$。显然，$L6$的计算不仅仅依靠于输入$x6$，还依赖于$L5$，进而依赖于$x5$，$L4$，...，直到$x1$，$L0$。（$L0$是第一步的填充向量）。那么也就是说，对于$L6$的误差计算，也用到了之前的时间步。因此在反向传播时，优化U和W，L对U和W分别求偏导数，计算的项其实很多，越往后计算的越多。这就是**BPTT**，随时间步的反向传播。每个句子优化一次U和W，而不是每一时间步都优化一次。
- **双向RNN的训练：**假如有个句子：我迟到了，老师惩罚了（）。很显然，前向就可以得知有用信息。而对于句子，（）迟到了，老师惩罚了我。RNN却不能得知信息。而且，实践表明，句子一开始的单词对于整个句子的信息提取至关重要，随着RNN时间步的推进，遗忘会加重，从而遗失了最开始的重要信息。Bi-RNN可以解决这个问题，两个方向的RNN**不共享参数**，而是使用两套参数，最后计算Loss时，将两个方向的Loss加和。
- 传统RNN具有梯度消失和梯度爆炸的问题。

Seq2Seq模型

Sequence to Sequence Learning在机器翻译等领域有成功的应用，它的本质是一种学习序列到序列的映射的模型框架。模型采用Encoder-Decoder结构，其中：

- Encoder：负责处理输入序列，提取信息，相当于“**编码**”
- Decoder：负责处理Encoder提取的信息，将其转换为目标序列，相当于“**解码**”

Encoder-Decoder网络的每个单元结构可以采用LSTM/GRU等。以翻译为例，可以这么理解，人在处理输入句子的时候，并不是每次对一个单词进行翻译，而是读取整个句子，理解其意义，在脑海中形成一种“中间语言”，再将其翻译为目标句子。Encoder-Decoder可以看作模拟了这个过程。

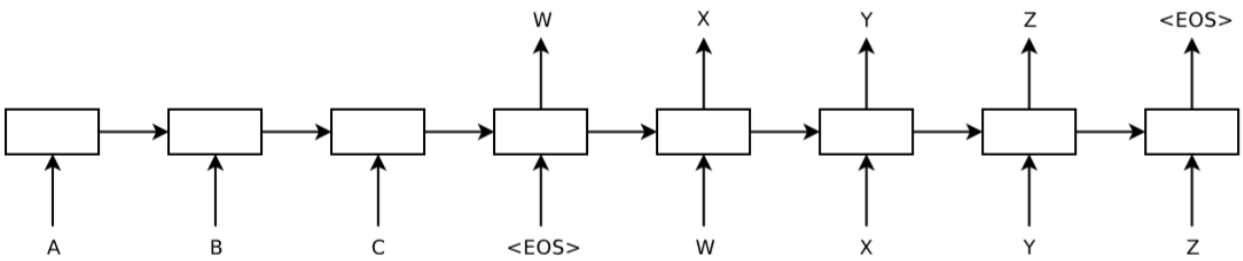
一般来说，Encoder-Decoder也可能遗忘。序列数据处理初期的信息往往对整个处理过程具有重要作用，而随着时间步的推进，初期的信息遗忘会加重，因此，一种可能的解决方案是：Encoder采用双向结构，一边从前向后处理输入，另一边反向处理。

其他一些细节：

- 论文提出，建议使用4层LSTM单元，深层的LSTM表现的更好；
- 初始化参数使用服从均匀分布$U(-0.8, 0.8)$ 的随机初始化；
- Decoder阶段输出层概率采用的是一个很大的softmax，这个占用了绝大多数的计算资源
- 原论文学习过程中，学习率初始0.7，迭代7.5次，前5次固定学习率是0.7，之后每半次迭代学习率减半一次
- 使用mini-batch，且`batch_size=128`
- 为了避免梯度消失和梯度爆炸，梯度裁剪。如果梯度g的二范数$||g|| > 5$，就进行$g = 5 * \frac{g}{||g||}$的转换。
- 论文实验的LSTM时间步为1000步，这是浪费的，但可以尽可能让同一batch里的句子长度几乎相同。这样是2倍加速效果的。
- 论文的实验采用8个GPU，其中4个用来处理LSTM的每一层，其余的处理softmax层。

[论文链接]

模型结构示意图如下：



GRU

- Gated Recurrent Unit.
- GRU是LSTM的一个简化版本，但也可以解决传统RNN的
- 设计GRU的思想是，如何让网络记住更多有效信息，而忽略不重要的信息，如a, ha这样的词。
- 使用“门”来让网络学习什么该记住，什么该忘记。
- **GRU单元的设计：**设时间步t的隐藏层为c_t，输入是x_t，输出是o_t。
  - 那么，传统RNN的计算是：$c_t = W_c\{c_{t-1}, x_t\}$。（忽略激活函数）

而GRU首先增加一个门G_u，来学习该记住多少，忘记多少:$G_u = sigmoid(W_u\{c_{t-1}, x_t\}$）

这样，原先计算c_t的方式不再直接作为c_t，而作为候选的c_t，记做c'_t，即：

：$c'_t = W_c\{c_{t-1}, x_t\}$。他表征了该从综合信息中得到多少。

为了让GRU自己学习该忘什么，该记什么，c_t的计算方式为:

$c_t = G_u * (c'_t) + (1 - G_u) * c_{t-1}$

显然，当$G_u = 0$，则当前输入就有很大作用。当$G_u = 1$，则完全忽视了本次输出x_t而完全记住了上一时间步的信息。

最后，上面式子还需再改进一下，再加一个门G_r，用来学习对于候选的c_t，上一步的c_{t-1}有多大计算上的贡献。即学习：$G_r = sigmoid(W_r\{c_{t-1}, x_t\})$

对候选的c'_t做更新：

$c'_t = W_c\{G_r * c_{t-1}, x_t\}$。

- 综上，标准的GRU单元计算为：

  $c_t = tanh(G_u * (c'_t) + (1 - G_u) * c_{t-1})$

  $c'_t = tanh(W_c\{G_r * c_{t-1}, x_t\})$

  $G_u = sigmoid(W_u\{c_{t-1}, x_t\}$）

  $G_r = sigmoid(W_r\{c_{t-1}, x_t\})$

LSTM

- 实际上LSTM是首先提出来的，GRU是LSTM的简化版本。
- **LSTM的论文很难懂，他深入探讨了梯度消失的问题。**
- **LSTM单元结构：**

  不再只有两个门，而是**三个门。**

  沿用GRU的符号。

  GRU的隐藏层c_t就是实际的一个时间步输出，而LSTM不是这样，LSTM一个时间步的输出记做a_t，并且：

  $a_t = G_o * (c_t)$。G_o称作输出门(output)，通过学习这个输出门，让网络自行决定对于这个时间步的隐藏层值的多少作为输出。输出门计算：

  $G_o = sigmoid(W_o[a_{t-1}, x_t])$

  这样，再决定隐藏层c_t的输出时，依然首先决定c_t的候选值，记做c'_t。并且：

  $c'_t = tanh(W_c * [a_{t-1}, x_t])$。

  即，以前学习隐藏层候选用的是上一步的隐藏层，而这里用的是上一步的LSTM输出。

  学习$c_t = G_u * c'_t + G_f * c_{t-1}$。

  可以看到，多了一个门，叫G_f，遗忘门，他学习了对于上一步的隐藏层，该遗忘多少，记住多少。而G_u，更新门，学习了对于本层和上层输出综合作用下，该学习多少信息。因此，LSTM非常善于记住前面的某些重要信息。学习两个门：

  $G_f = sigmoid(W_f[a_{t-1}, x_t])$

  $G_u = sigmoid(W_u[a_{t-1}, x_t])$

- 综上所述：LSTM的式子为：

  $a_t = G_o * c_t$

$$c_t = G_u * c'_t + G_f * c_{t-1}$$

$$c'_t = tanh(W_c[a_{t-1}, x_t])$$

$$G_o = sigmoid(W_o[a_{t-1}, x_t])$$

$$G_u = sigmoid(W_u[a_{t-1}, x_t])$$

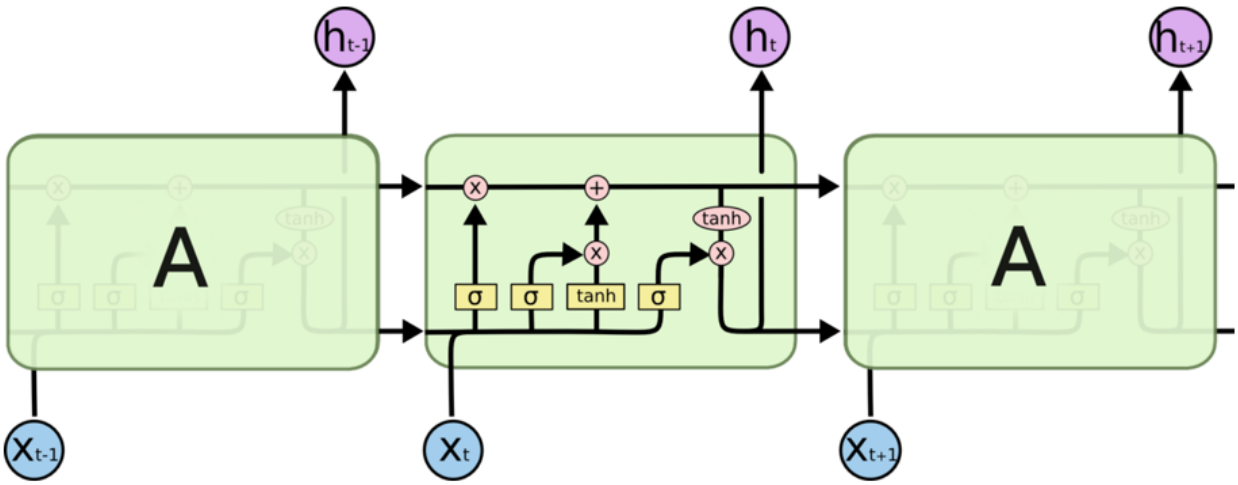
$$G_f = sigmoid(W_f[a_{t-1}, x_t])$$

- 由此可见，对于一个LSTM，实际上可以看作有两个隐藏层c_t和a_t，在时间步向前推进时，这两个都需要。这也是为什么pytorch中LSTM函数需要两个隐藏层输入。
- 三个门的意义：

  输出门：本时间步学到的信息多少用于输出。因为本时间步学到的东西并非全部对本层的输出有用，而是有些信息要保存到下一时间步以供使用。

  更新门：也称作记忆门。通过本时间步的输入，以及上一步的输出，学习得到本层隐藏值的候选值，这部分有多少贡献于本时间步的真正隐藏层值，就用这个门来学。

  遗忘门：上一步的隐藏层学到的值是c_{t-1}，这部分信息有多少要遗忘，有多少依旧记住它。
- 双向LSTM：道理同GRU。
- 经典的LSTM示意图如下：



注意力机制

- 注意力机制解决的问题有两个：**一个是梯度消失问题**，即使是LSTM依然可能忘记；**另一个是将一个长度可变的句子压缩编码进一个长度定长的dense vector**，可能容纳的信息是不全的。
- 注意力机制中，Encoder将句子信息提取，并不放进一个vectov，而是放进一个**列数可变**的**矩阵**。具体来说，Encoder提取信息->Matrix，该Matrix中，行数是特征数，列数是句子长度，也就是Encoder的时间步数。
- 一般来说，Encoder使用Bi-LSTM，设隐藏层维度是`hidden_size`，则矩阵维度是`(hidden_size*2,len(Sentence))`
- Decoder如何使用Matrix呢？ 首先得有一个权重u向量，权重向量a的长度就是Matrix的列数，即u的维度是`(len(Sentence),1)`，这样，$Matrix * u$就得到一个列向量**y**，长度是特征数。Encoder使用Bi-LSTM，则**y**的维数是`(hidden_size*2,1)`。
- 那么如何得到权重**u**：**u**显然必须是归一化了的，将**u**的确定交给Neural Nerwork。有几种方法，典型的有，可以这样学习：

  $t-1$时刻隐藏层输出a_{t-1}（包含了前向和后向的连接，即这是`2*hidden_size`），t时刻输入x_t。显然：

a_{t-1}：$(2 * hidden_size, 1)$

进行一个线性变换：

$r_t = W_r * a_{t-1}$，r_t：$(2 * hidden_size, 1)$，W_r是学习参数。

之后：

$u_t = Matrix^T * r_t$，u_t：$(len(Sentence), 1)$

最后再对u_t做归一化softmax即可。

- 上面的方法是线性的。还有一种非线性方法：

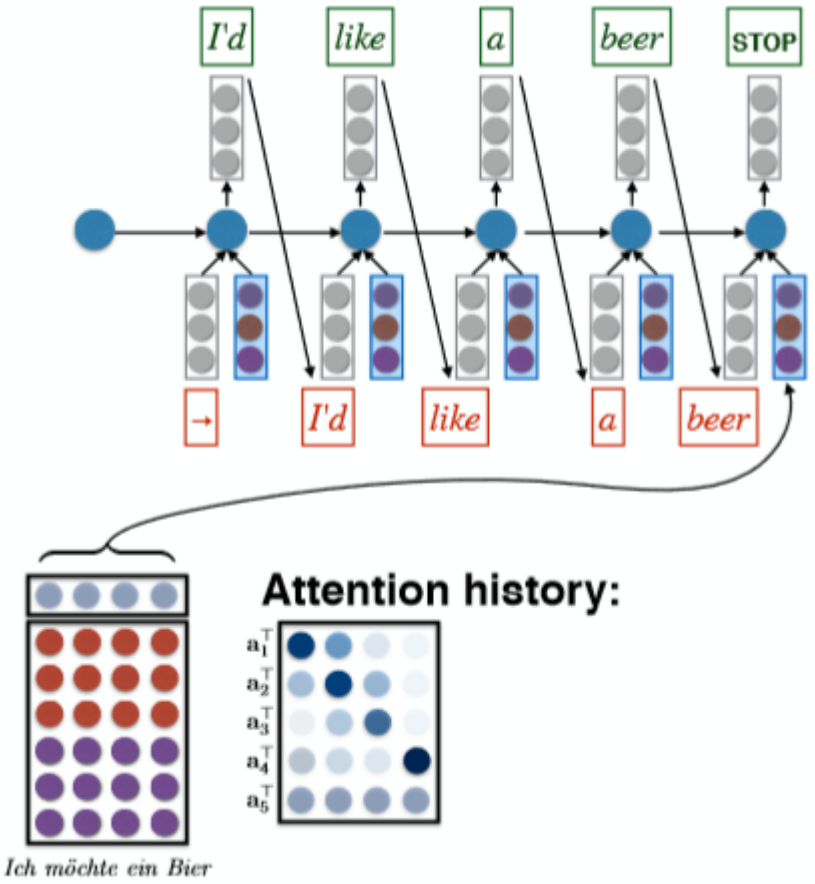
  $r_t = W_r * a_{t-1}$

  $u_t = W_{u2} * tanh(W_{u1} * Matrix + r_t)$

  再做归一化。

- 最后一点是，注意力机制得出的一个上下文向量c，如何用呢？加入Decoder端上一时间步预测出y'_{t-1}，那么预测y'_t时，要么是：将c和y'_{t-1}一起放入RNN网络，即各自乘以参数加权和。或者，RNN只用y'_{t-1}作为输入，得到输出后再同c做加权学习处理。

- 注意力机制示意图如下，图片来自牛津大学自然语言处理课程PPT。



关于pytorch的使用

- nn.Embedding：链接
- pytorch文本对齐填充和逆填充：链接
- 各种优化器

1. 关于pythoch中的Embedding

pytorch中实现了Embedding，下面是关于Embedding的使用。

torch.nn包下的Embedding，作为**训练的一层**，随模型训练得到适合的词向量。

建立词向量层

```
embed = torch.nn.Embedding(n_vocabulary,embedding_size)
```

找到对应的词向量放进网络：词向量的输入应该是什么样子

实际上，上面通过**随机初始化**建立了词向量层后，建立了一个“二维表”，存储了词典中每个词的词向量。每个mini-batch的训练，都要从词向量表找到mini-batch对应的单词的词向量作为RNN的输入放进网络。那么怎么把mini-batch中的每个句子的所有单词的词向量找出来放进网络呢，输入是什么样子，输出是什么样子?

首先我们知道肯定先要建立一个词典，建立词典的时候都会建立一个*dict：word2id*：存储**单词到词典序号的映射**。假设一个mini-batch如下所示：

```
['I am a boy.','How are you?','I am very lucky.']
```

显然，这个mini-batch有3个句子，即`batch_size=3`

第一步首先要做的是：将句子标准化，所谓标准化，指的是：大写转小写，标点分离，这部分很简单就略过。经处理后，mini-batch变为：

```
[['i','am','a','boy','.'],['how','are','you','?'],['i','am','very','lucky','.']]
```

可见，这个list的元素成了一个个list。还要做一步：将上面的三个list按单词数从多到少排列。标点也算单词。至于为什么，后面会说到。

那就变成了：

```
batch = [['i','am','a','boy','.'],['i','am','very','lucky','.'], ['how','are','you','?']]
```

可见，每个句子的长度，即每个内层list的元素数为：5,5,4。这个长度也要记录。

```
lens = [5,5,4]
```

之后，为了能够处理，将batch的单词表示转为在词典中的**index**序号，这就是word2id的作用。转换过程很简单，假设转换之后的结果如下所示，当然这些序号是我编的。

```
batch = [[3,6,5,6,7],[6,4,7,9,5], [4,5,8,7]]
```

同时，每个句子结尾要加**EOS**，假设EOS在词典中的index是1。

```
batch = [[3,6,5,6,7,1],[6,4,7,9,5,1],[4,5,8,7,1]]
```

那么长度要更新：

```
lens = [6,6,5]
```

很显然，这个mini-batch中的句子长度**不一致！**所以为了规整的处理，对长度不足的句子，进行填充。填充PAD假设序号是2，填充之后为：

```
batch = [[3,6,5,6,7,1],[6,4,7,9,5,1],[4,5,8,7,1,2]]
```

这样就可以直接取词向量训练了吗?

不能！上面batch有3个样例，RNN的每一步要输入每个样例的一个单词，一次输入**batch_size**个样例，所以batch要按list外层是时间步数(即序列长度seq_len)，list内层是batch_size排列。**即batch的维度应该是：**

[seq_len,batch_size]

怎么变换呢？**变换方法可以是**：使用**itertools**模块的**zip_longest**函数。而且，使用这个函数，连填充这一步都可以省略，因为这个函数可以实现填充！

```
batch = list(itertools.zip_longest(batch,fillvalue=PAD))
# fillvalue就是要填充的值，强制转成list
```

经变换，结果应该是：

```
batch = [[3,6,4],[6,4,5],[5,7,8],[6,9,7],[7,5,1],[1,1,2]]
```

记得我们还记录了一个lens：

`lens = [6,6,5]`

batch还要转成`LongTensor`：

`batch=torch.LongTensor(batch)`

这里的batch就是词向量层的输入。

词向量层的输出是什么样的?

好了，现在使用建立了的embedding直接通过batch取词向量了，如：

`embed_batch = embed (batch)`

假设词向量维度是6，结果是：

```
tensor([[[-0.2699,  0.7401, -0.8000,  0.0472,  0.9032, -0.0902],
         [-0.2675,  1.8021,  1.4966,  0.6988,  1.4770,  1.1235],
         [ 0.1146, -0.8077, -1.4957, -1.5407,  0.3755, -0.6805]],

        [[-0.2675,  1.8021,  1.4966,  0.6988,  1.4770,  1.1235],
         [ 0.1146, -0.8077, -1.4957, -1.5407,  0.3755, -0.6805],
         [-0.0387,  0.8401,  1.6871,  0.3057, -0.8248, -0.1326]],

        [[-0.0387,  0.8401,  1.6871,  0.3057, -0.8248, -0.1326],
         [-0.3745, -1.9178, -0.2928,  0.6510,  0.9621, -1.3871],
         [-0.6739,  0.3931,  0.1464,  1.4965, -0.9210, -0.0995]],

        [[-0.2675,  1.8021,  1.4966,  0.6988,  1.4770,  1.1235],
         [-0.7411,  0.7948, -1.5864,  0.1176,  0.0789, -0.3376],
         [-0.3745, -1.9178, -0.2928,  0.6510,  0.9621, -1.3871]],

        [[-0.3745, -1.9178, -0.2928,  0.6510,  0.9621, -1.3871],
         [-0.0387,  0.8401,  1.6871,  0.3057, -0.8248, -0.1326],
         [ 0.2837,  0.5629,  1.0398,  2.0679, -1.0122, -0.2714]],
```

```
        [[ 0.2837,  0.5629,  1.0398,  2.0679, -1.0122, -0.2714],
         [ 0.2837,  0.5629,  1.0398,  2.0679, -1.0122, -0.2714],
         [ 0.2242, -1.2474,  0.3882,  0.2814, -0.4796,  0.3732]]],
       grad_fn=<EmbeddingBackward>)
```

维度的前两维和前面讲的是一致的。可见多了一个第三维，这就是词向量维度。所以，Embedding层的输出是：

[seq_len,batch_size,embedding_size]

2. 关于pytorch中的GRU

取词向量，放进GRU。

建立GRU

```
gru = torch.nn.GRU(input_size,hidden_size,n_layers)
# 这里的input_size就是词向量的维度，hidden_size就是RNN隐藏层的维度，这两个一般相同就可以
# n_layers是GRU的层数
```

可见，并不需要指定时间步数，也即seq_len，这是因为，GRU和LSTM都实现了自身的迭代。

GRU的输入应该是什么样子的?

上面的`embed_batch`作为Embedding层的输出，可以直接放进GRU中吗?

理论上可以，但这样不对！**因为GRU并不知道哪些是填充的，并不是每一个句子都满足最大序列长度！**所以我们事先用lens记录了长度。

将输出`embed_batch`转成`pack_padded_sequence`，使用`torch.nn.utils.rnn.`下的`pack_padded_sequence`方法。

```
batch_packed = torch.nn.utils.rnn.pack_padded_sequence(embed_batch, lens)
# 注意这里的输入lens就是前面的长度list
```

这个`batch_packed`就是GRU的输入。

`batch_packed`长啥样?

不妨看一下：

```
PackedSequence(data=tensor([[-0.2699,  0.7401, -0.8000,  0.0472,  0.9032, -0.0902],
        [-0.2675,  1.8021,  1.4966,  0.6988,  1.4770,  1.1235],
        [ 0.1146, -0.8077, -1.4957, -1.5407,  0.3755, -0.6805],
        [-0.2675,  1.8021,  1.4966,  0.6988,  1.4770,  1.1235],
        [ 0.1146, -0.8077, -1.4957, -1.5407,  0.3755, -0.6805],
        [-0.0387,  0.8401,  1.6871,  0.3057, -0.8248, -0.1326],
        [-0.0387,  0.8401,  1.6871,  0.3057, -0.8248, -0.1326],
        [-0.3745, -1.9178, -0.2928,  0.6510,  0.9621, -1.3871],
        [-0.6739,  0.3931,  0.1464,  1.4965, -0.9210, -0.0995],
        [-0.2675,  1.8021,  1.4966,  0.6988,  1.4770,  1.1235],

        [-0.7411,  0.7948, -1.5864,  0.1176,  0.0789, -0.3376],
```

```
        [-0.3745, -1.9178, -0.2928,  0.6510,  0.9621, -1.3871],
        [-0.3745, -1.9178, -0.2928,  0.6510,  0.9621, -1.3871],
        [-0.0387,  0.8401,  1.6871,  0.3057, -0.8248, -0.1326],
        [ 0.2837,  0.5629,  1.0398,  2.0679, -1.0122, -0.2714],
        [ 0.2837,  0.5629,  1.0398,  2.0679, -1.0122, -0.2714],
        [ 0.2837,  0.5629,  1.0398,  2.0679, -1.0122, -0.2714]],
       grad_fn=<PackPaddedBackward>), batch_sizes=tensor([3, 3, 3, 3, 3, 2], grad_fn=
<PackPaddedBackward>))
```

可以看到，属性**batch_sizes**清楚的记录了每个时间步上batch输出是多少，而且去除了PAD。

此外，GRU还需要一个初始隐藏向量（注意层数和方向），嫌麻烦直接传None也无妨。

所以输入应该是(`batch_packed` , `None`)

GRU的输出?

```
output,hidden = gru(batch_packed,None)
```

output：PackedSequence对象

```
PackedSequence(data=tensor([[ 0.0432, -0.0149, -0.0884, -0.0194, -0.0740,  0.1278],
        [-0.0436, -0.0726,  0.0568, -0.0995, -0.1992,  0.1594],
        [ 0.0582,  0.0625, -0.1639,  0.1474,  0.0077,  0.0542],
        [-0.0052, -0.0732,  0.0031, -0.1367, -0.2336,  0.2307],
        [ 0.0131,  0.0234, -0.0681,  0.0535, -0.1651,  0.1864],
        [ 0.0324,  0.1441, -0.1788,  0.1800, -0.0816,  0.1684],
        [-0.0788, -0.0148, -0.0292, -0.1348, -0.3352,  0.3045],
        [ 0.0502,  0.0436, -0.1509,  0.1481, -0.1284,  0.1523],
        [ 0.0627,  0.1626, -0.1888,  0.1341, -0.0984,  0.2627],
        [-0.1391, -0.0149,  0.0473, -0.2069, -0.4410,  0.3690],
        [ 0.1378,  0.0578, -0.2008,  0.1265, -0.0149,  0.2053],
        [ 0.0780,  0.1199, -0.2107,  0.1460, -0.0906,  0.2291],
        [-0.1019,  0.0055, -0.0304, -0.1277, -0.4149,  0.3582],
        [ 0.0906,  0.1025, -0.1646,  0.0933, -0.0953,  0.2905],
        [ 0.1004,  0.1175, -0.1911,  0.0979, -0.0877,  0.2771],
        [-0.0607,  0.0469, -0.0935, -0.1002, -0.3568,  0.3707],
        [ 0.0737,  0.1213, -0.1516,  0.0365, -0.1417,  0.3591]],
       grad_fn=<CatBackward>), batch_sizes=tensor([3, 3, 3, 3, 3, 2], grad_fn=
<PackPaddedBackward>))
```

前三个list对应于第一时间步，mini-batch的三个样例的输出。依次类推。最后只有两个，因为最后是有缺省的。

第二个输出hidden：是个张量。维度**[n_layers,batch_size,hidden_size]**

```
tensor([[[-0.1057,  0.2273,  0.0964,  0.2777,  0.1391, -0.1769],
         [-0.1792,  0.1942,  0.1248,  0.0800, -0.0082,  0.0778],
         [-0.2631,  0.1654,  0.1455, -0.1428,  0.1888, -0.2379]],

        [[-0.0607,  0.0469, -0.0935, -0.1002, -0.3568,  0.3707],
         [ 0.0737,  0.1213, -0.1516,  0.0365, -0.1417,  0.3591],
         [ 0.1004,  0.1175, -0.1911,  0.0979, -0.0877,  0.2771]]],
       grad_fn=<ViewBackward>)
```

所以到这，为什么逆序，为什么记录长度也就清楚了。

3. 关于pytroch中的LSTM

LSTM有两个隐藏层向量，其余基本同GRU。

4. 关于pytorch中的Adam优化器

Adam优化器的参数：

```
params (iterable) – iterable of parameters to optimize or dicts defining parameter groups

lr (float, optional) – learning rate (default: 1e-3)

betas (Tuple[float, float], optional) – coefficients used for computing running averages of
gradient and its square (default: (0.9, 0.999))

eps (float, optional) – term added to the denominator to improve numerical stability (default:
1e-8)

weight_decay (float, optional) – weight decay (L2 penalty) (default: 0)

amsgrad (boolean, optional) – whether to use the AMSGrad variant of this algorithm from the
paper On the Convergence of Adam and Beyond (default: False)
```

注：

- params：要优化的参数，可以使用继承nn.Module的类，如encoder.parameters()获取这些参数对象。参数应该是可迭代的。
- lr：学习率，默认1e-3
- betas，是个元组(b1,b2)，默认(0.9.0.999)，这个就是Adam算法中对应的两个β参数
- eps：无穷小量，看7中有解释
- weight_decay，防止过拟合。正则化项的系数。默认是L2正则化。

5. 关于交叉熵损失函数的使用

- 计算公式<原理>详解：[链接][参考]
- 位于nn.functional下

- cross_entropy本质上是一个继承自nn.Module的类，因此也实现了forward。
- 做cross_entropy不需要先softmax。
- 参数使用：定义cross_entropy后，主要需要3个参数：
    - input：Tensor。维度[batch_size,num_classes]。如做文本序列处理时，num_classes就是词典的单词数

      input就是向量，简而言之每一个元素对应了预测是这个“分类”的概率，但并不用做softmax，因为里面实现了softmax。
    - target:目标Tensor，维度[1,batch_size]。如tensor([1, 0, 1])。即是一行数字，每个数字代表mini-batch的一个样例真正的分类，数字1表示正确结果应该是词典中的第1个单词。
    - ignore_indexes=EOS_token。这表示，当target是EOS_token时，从这里开始忽略计算loss。

6. pytorch防止梯度爆炸的方法

- 防止梯度爆炸的`clip_grad_norm_`
    - 位于torch.nn.utils下
    - 使用clip_grad_norm_(parameters(),max_gradient)

优化算法

1. GD/SGD/mini-batch GD

GD：Gradient Descent，就是传统意义上的梯度下降，也叫batch GD。

SGD：随机梯度下降。一次只随机选择一个样本进行训练和梯度更新。

mini-batch GD：小批量梯度下降。GD训练的每次迭代一定是向着最优方向前进，但SGD和mini-batch GD不一定，可能会“震荡“。把所有样本一次放进网络，占用太多内存，甚至内存容纳不下如此大的数据量，因此可以分批次训练。可见，SGD是mini-batch GD的特例。

2. 动量梯度下降

一个有用的方法：**指数加权平均。**

假如有N天的数据，$a_1, a_2, a_3, \ldots, a_N$

如果想拟合一条比较平滑的曲线，怎么做呢。可以使用指数加权平均。概括的说，就是平均前m天的数据作为一个数据点：

$b_0 = 0$

$b_t = \beta * b_{t-1} + (1-\beta) * a_t$

比如β取0.9，那么就相当于是平均了10天的数据，这会比较平滑。

为什么叫**指数加权平均**？是因为，将b_t计算公式中的b_{t-1}展开，依次展开下去，会发现：

$b_t = \beta * (\beta * b_{t-2} + (1-\beta) * a_{t-1}) + (1-\beta) * a_t$

合并一下，前面的数据乘以β的m次方的平均值。因此叫做指数加权平均。

指数加权平均的偏差修正：

很显然，上面公式汇总，假如$\beta = 0.9$，是要平均前10天的数据，则前9天的数据，做加权平均，并没有足够的数据，这会导致前九天数据偏小。举例来说，第一天b_1仅仅是a_1的十分之一。可以做偏差修正：

$b_0 = 0$

$b_t = \beta * b_{t-1} + (1-\beta) * a_t$

$b_t = \frac{b_t}{1-\beta^t}$

有了指数加权平均的基本知识，就可以讲Moment Gradient Descent。

带动量的梯度下降，他是为了**加快学习速度**的优化算法。假如参数W方向希望学的快一点，b方向学的慢一点。普通的梯度下降的结果可能恰恰相反，使得b方向学的比较快，从而引发**震荡**。所以想要让b方向学的慢一点，这时可以用动量梯度下降。算法如下：

For each epco t:

cal dW, dB for each mini-batch.

$V_{dW} = \beta * V_{d_W} + (1-\beta) * dW$

$V_{db} = \beta * V_{d_b} + (1-\beta) * db$

$W = W - \alpha * V_{dW}$

$b = b - \alpha * V_{db}$

可见，就是将梯度做了指数加权平均。至于为什么叫动量梯度下降，可能是因为V_{dW}的计算式中，把dW看作加速度，把V_{dW}看作速度。

3. RMSprob

Root Mean Square prob。

这是另一种加快学习的方法。其基本思想也是让b学的慢一点，让W学的快一点，从而更快更准的趋向于最优点。

For each epco t:

cal dW, dB for each mini-batch.

$S_{dW} = \beta * S_{dW} + (1-\beta) * (dW)^2$

$S_{db} = \beta * S_{db} + (1-\beta) * (db)^2$

$W = W - \alpha * \frac{dW}{\sqrt{S_{dW}}}$

$b = b - \alpha * \frac{db}{\sqrt{S_{db}}}$

可见，当梯度较大，则会使得S较大，从而使得更新变缓慢。

4. Adam

Adaptive Moment Estimation

这是比较普遍的适用于各种网络的一种方法。称作自适应的动量梯度下降。这是上面动量梯度下降和RMSprob的结合版本，效果比较好。两者做加权指数平均的时候，都做了修正。

For each epco t:

cal dW, dB for each mini-batch.

$V_{dW} = \beta_1 * V_{d_W} + (1 - \beta_1) * dW$

$V_{db} = \beta_1 * V_{d_b} + (1 - \beta_1) * db$

$S_{dW} = \beta_2 * S_{dW} + (1 - \beta_2) * (dW)^2$

$S_{db} = \beta_2 * S_{db} + (1 - \beta_2) * (db)^2$

$V_{dW}^{correction} = \frac{V_{dW}}{1-\beta_1^t}$

$V_{db}^{correction} = \frac{V_{db}}{1-\beta 1^t}$

$S_{dW}^{correction} = \frac{S_{dW}}{1-\beta_2^t}$

$S_{db}^{correction} = \frac{S_{db}}{1-\beta_2^t}$

$W = W - \alpha * \frac{V_{dW}^{correction}}{\sqrt{S_{dW}^{correcti}}+\varepsilon}$

$b = b - \alpha * \frac{V_{db}^{correction}}{\sqrt{S_{db}^{correcti}}+\varepsilon}$

可见，就是在RMSprob中，用动量梯度下降中的梯度指数加权代替梯度，同时所有指数加权项都做了偏差修正。另外，分母加了ε，这是为了防止除以很小的数造成不稳定。公式中共有4个超参数，他们的取值经验是：

α：学习率，需要调试

β_1：取0.9

β_2：Adam的作者建议取0.999

ε：取10^{-8}，并不影响学习效果。

另外，值得注意的是，学习过程中可能有两种窘境，一是困在局部最优，另一个是遇平缓区学习的过慢。采用mini-batch下前者出现的概率很小，即使困在最优也能跳出来，前提是数据量足够。但后者比较棘手，Adam是个比较好的优化算法，一定程度上解决了这个问题。

5. 学习率衰减

训练过程中，随着迭代次数，学习率相应减少。

在FB的一篇论文中，使用了不同时间段（迭代次数区间段）采用不同的学习率的方法，效果比较好。

6. 防止过拟合的正则化方法

0. 范数

链接

1. L2正则化

- 对损失函数加上正则项，可以减少过拟合，减少训练误差。
- cross_entropy中实现了默认L2正则化，即只要指定weight_decay参数即可，这是L2正则化的系数。
- 正则项是加L2范数，但这个范数不是2范数，而是F范数，即矩阵所有元素的平方和。
- 具体来说，设原先loss = loss，则，正则化之后为：

  $loss = loss + \frac{\lambda}{2*m} \sum_{l=1}^{L} ||w^{[l]}||_F^2$

  其中，$w^{[l]}$值第l层的参数。这指的是第l层的权值矩阵的F范数。
- 另外，偏差b的范数通常忽略不写。

2. L1 正则化：

- 加L1范数，即加矩阵所有元素绝对值之和。

3. 为什么正则化可以减少过拟合、减少方差问题?

- 直观的理解是：系数λ设置的比较大，那么权重w就会相对较小，相当于"失活"了部分网络单元，自然就相当于简单化了网络，从而理解为减少过拟合。

7. 另一种防止过拟合方法：关于dropout

- 随机失活的思想。对网络的一层随机失活，以一定概率使得一些单元为0，比如使得单元的输出中，某些维度上的值赋0，从而简单化了网络。
- 网络的最后一层不加dropout，因为我不想让输出是随机的。
- pytorch关于dropout层的添加和使用：

```
dropout
torch.nn.functional.dropout(input, p=0.5, training=True, inplace=False)
[source]

During training, randomly zeroes some of the elements of the input tensor with probability p
using samples from a Bernoulli distribution.
See Dropout for details.


Parameters:
p – probability of an element to be zeroed. Default: 0.5
training – apply dropout if is True. Defualt: True
inplace – If set to True, will do this operation in-place. Default: False
```

8. 关于Batch-norm

之前有提到，特征X输入进网络的时候，可以先做标准化（减去均值，除以标准差），因为这可以让特征的各个维度均匀，从而使得要学习的参数的各个维度也是分布均匀的，这样可以加快速度，减少震荡，具体的原因参考百面机器学习一书。

Batch-norm，含义是：不仅标准化操作应用于输入特征，也应用于深层神经网络的每一层，即，每一层计算出输出之后，先做归一化（零均值，1方差），再做激活函数处理，然后将之输入进下一层网络，叫做Batch-Normlization，它是以每个mini-batch为单位进行处理的。

具体来说，batch-norm+Adam优化算法的处理过程如下：

For each **mini-batch** b:

For each **layer** l:

$z^{[l]} = W^{[l]} * z^{[l-1]} + b^{[l-1]}$

$u = \frac{1}{m} * \sum_{i=1}^{m} z^{[l][i]}$

$\sigma^2 = \frac{1}{m} * \sum_{i=1}^{m} (z^{[l][i]} - u)^2$

$z_{norm}^{[l]} = \frac{z^{[l]} - u}{\sqrt{\sigma^2 + \epsilon}}$

$z_{final}^{[l]} = \gamma * z_{norm}^{[l]} + \beta_z$

$z^{[l]} = z_{final}^{[l]}$

$a^{[l]} = tanh(z^{[l]})$

Use Adam Do Gradient Descent on the mini-batch：

cal $dW, db, d\gamma, d\beta_z$ on the current mini-batch:

$V_{dW} = \beta_1 * V_{dW} + (1 - \beta_1) * dW$

$V_{db} = \beta_1 * V_{db} + (1 - \beta_1) * db$

$V_{d\gamma} = \beta_1 * V_{d\gamma} + (1 - \beta_1) * d\gamma$

$V_{d\beta_z} = \beta_1 * V_{d\beta_z} + (1 - \beta_1) * d\beta_z$

$S_{dW} = \beta_2 * S_{dW} + (1 - \beta_2) * (dW)^2$

$S_{db} = \beta_2 * S_{db} + (1 - \beta_2) * (db)^2$

$S_{d\gamma} = \beta_2 * S_{d\gamma} + (1 - \beta_2) * (d\gamma)^2$

$S_{d\beta_z} = \beta_2 * S_{d\beta_z} + (1 - \beta_2) * (d\beta_z)^2$

$V_{dW}^{correction} = \frac{V_{dW}}{1 - \beta_1^t}$

$V_{db}^{correction} = \frac{V_{db}}{1 - \beta 1^t}$

$V_{d\gamma}^{correction} = \frac{V_{d\gamma}}{1 - \beta_1^t}$

$V_{d\beta_z}^{correction} = \frac{V_{d\beta_z}}{1 - \beta 1^t}$

$S_{dW}^{correction} = \frac{S_{dW}}{1 - \beta_2^t}$

$S_{db}^{correction} = \frac{S_{db}}{1 - \beta_2^t}$

$S_{d\gamma}^{correction} = \frac{S_{d\gamma}}{1 - \beta_2^t}$

$$S_{d\beta_z}^{correction} = \frac{S_{d_beta_z}}{1-\beta_2^t}$$

$$W = W - \alpha * \frac{V_{dW}^{correction}}{\sqrt{S_{dW}^{correcti}} + \varepsilon}$$

$$b = b - \alpha * \frac{V_{db}^{correction}}{\sqrt{S_{db}^{correcti}} + \varepsilon}$$

$$\gamma = \gamma - \alpha * \frac{V_{d\gamma}^{correction}}{\sqrt{S_{d\gamma}^{correcti}} + \varepsilon}$$

$$\beta_z = \beta_z - \alpha * \frac{V_{d\beta_z}^{correction}}{\sqrt{S_{d\beta_z}^{correcti}} + \varepsilon}$$

可以看到，要点有以下几点：

- 式子中，两次出现了无穷小量ϵ，是为了保证数值的稳定性。
- 应用Adam时，梯度的指数加权平均，以及梯度的平方的加权平均都做了“数值修正”。并且，梯度的平方的加权平均做“分母”，能够加快梯度下降过程，具体原因见7.3.
- 对每一个Batch做Norm，对网络的每一层，先做Norm，再进行激活。Norm的过程为，计算均值方差，减去均值除以标准差，达到“0均值1方差”的归一化效果，使得分布更稳定，基础更牢固，可以看到，又学习了γ和β_z两个参数，使得网络更稳定；另外，当γ等于标准差，β_z等于均值，那就是做了标准化的复原。究其原因，还有一个原因是，这样的方差和均值是由噪音的，只是用一个batch的均值取近似全局的均值。

Batch-norm为什么奏效?

- 直观的原因是，我们前面看到，特征X输入网络，先做归一化，使得分布均匀，学习速率快，学习效果好。
- 现在考虑第N层网络，它的输入是第N-1层网络的输出结果，训练过程中，前一层的输出是变化的，那么就相当于从第N层到之后所有层的网络，要动态的适应分布变化了的输入（即第N-1层输出），那么网络就经过不稳定的变化去适应新的分布，倘若做了norm，就可以更多的利用前面训练的信息，前面学习到的映射不需要进行很大调整，从而加快训练速度，提高稳定性。更进一步说明见下。
- 训练的实质是将一个分布映射到另一个分布。假如现在将X分布映射到了Y分布，当X分布发生了变化，这个映射显然就不成立了。但是，如果将X标准化，学习映射X->Y，即使X分布发生变化变成X'，X'的标准化结果依然是可以通过已经学过的映射变换到Y的，这就是norm的好处：稳定，速率快。对应上面第二点。
- Norm有利于减少梯度消失的情况，因为他将变量大小变换到小的范围。
- Norm有点类似于正则化的效果：即减少过拟合。这是因为，一个batch计算中的均值，是有噪音的，那么z的计算也是有噪音的，减去均值的操作也会使得部分点为0，相当于失活了部分网络unit。增大batch_size使得均值更准确，相当于减少了正则化效果。
- 总之，Norm是有益的。

测试时的Batch-Norm：

- 测试只输入一条数据，无法计算均值方差，怎么办?
- 方法是，训练过程中计算出每层，所有mini-batch的均值，保存，对这些batch的均值、方差，做指数加权平均，作为预测时一条数据的均值方差使用。
- 本质是：估计均值和方差。

Teach-forcing OR Curriculum learining?

这是一个学习过程中的细节问题，值得重视。

[有用的学习链接]

[有用的详解]

问题：seq2seq的Decoder中，上一时刻的输出作为下一时刻的输入。但上一时刻的输出错了怎么办？是把模型预测的结果放进下一时刻的输入，还是把真实的结果放进去？

把预测出的错误的结果放进去，显然是不对的，这会让之后的预测越来越差，模型稳定性不好。

把真实的结果放进去，这样相当于纠正了上一时刻的错误，这叫'**Teach Forcing**'。

还有一种 **curriculum learining**，译为课程学习，其原理是，下一时刻的输入究竟取哪一个，随机选择，要么选真是结果，要么干脆放进去预测结果。

评估指标

1. 交叉熵损失

关于交叉熵损失的详解，以及为什么用average cross-entropy loss:链接

下面简单总结一下：

能够想到的最直接的损失是：均方误差MSE，但它是非凸的！有许多局部最优，但是，回归问题还是经常使用这个的，即使他有缺点，但回归问题只预测一个值，没大问题。

为什么不用绝对误差？绝对误差并不能真正衡量两个模型的实际区别，因为可能A模型优于B，但某些数据上预测结果却一样，绝对误差也就一样，这时候不能说A=B。

更好的选择是：**交叉熵损失。**

计算公式：

$$Entropy_{loss} = -\frac{1}{m} * \sum_{i}^{m} y * log(y^{hat})$$

可以看到，就是对真实值乘以预测值的log，加和，再平均。

这个的好处是：

- 准确的衡量模型内在的差异，给出真实的评价
- 这是个凸优化，不存在局部解

2. 困惑度Perplexity

有用的链接

其他

- 一般用困惑度大致衡量模型的收敛情况。越低越好。
- 困惑度并不能真正衡量模型的好坏，有用的经验是：标点，的、了这类副词，对翻译的结果没有很大影响，但对perplexity的大小很有影响。
- perplexity衡量的是模型在预测样本上的优劣程度。计算公式有很多。