# 零

# 一、语言

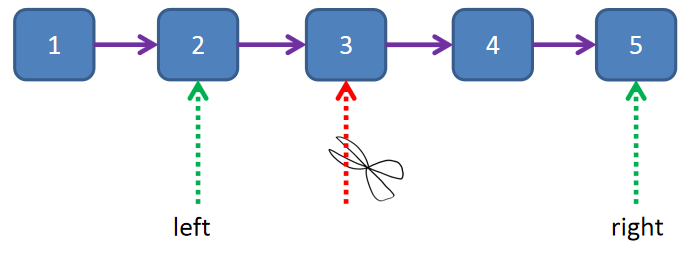
## 1.1 python

## 1.2 C++

## 1.3 常用数据结构

### ### 链表

删除链表的导数第n个节点，如果用快慢指针的思路我们希望构造估下的情形：



假如想删除倒数第三个节点的话1首先把两个指针拉开距离为32然后两指针一起后移直到right指向末节点3此时的情形如图所示：我们直接把left.next设定为left.next.next即可特例：要删除头结点(即删除倒数第5个节点)，判断方法为第二步时right为空

## ## shell

shell就是一个脚本文件，我们平时吧命令输入到命令行中，有了shell脚本，可以把所有的操作写进去，然后直接运行这个sh文件就行了。

## ## pip

拒绝访问，可能是有些报正在使用，关闭pycharm等软件再试试。也有可能权限不够，需要管理员权限。

## ## linux

1.nohup 命令

nohup file >> log.file 2>&1 &

nohup /usr/local/node/bin/node /www/im/chat.js >> /usr/local/node/output.log 2>&1 &

查看运行的后台进程

（1）jobs -l

jobs命令只看当前终端生效的，关闭终端后，在另一个终端jobs已经无法看到后台跑得程序了，此时利用ps（进程查看命令）

（2）ps -ef

ps -aux|grep filename

a:显示所有程序

u:以用户为主的格式来显示

x:显示所有程序，不以终端机来区分

# 二.AI

## 2.1 机器学习

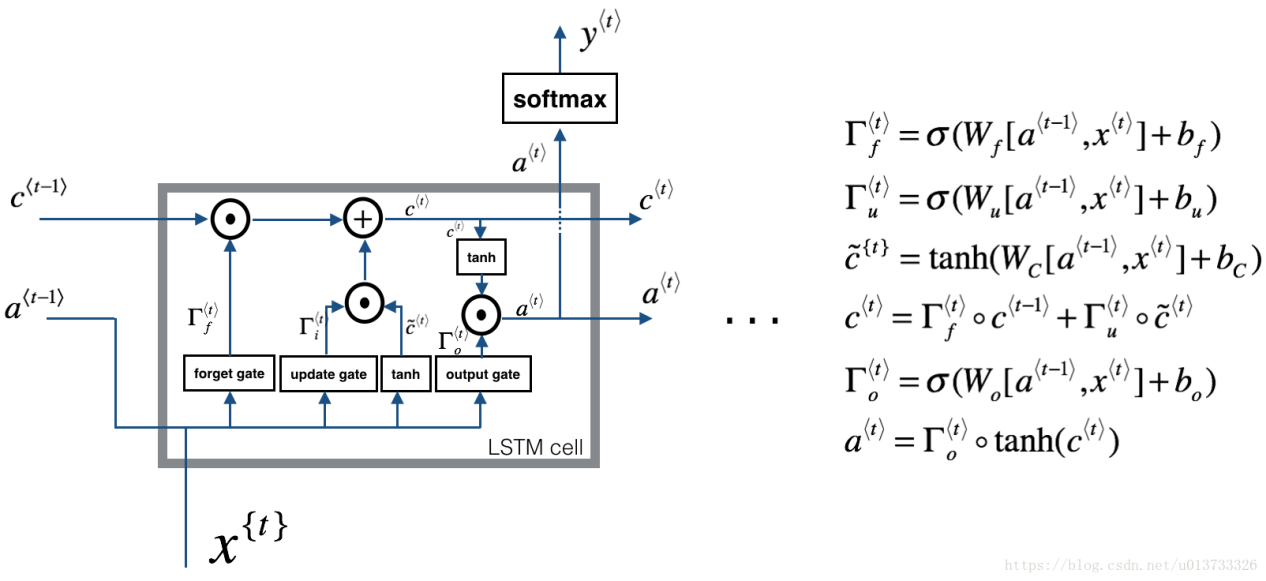
## 2.2、深度学习

### 2.2.1 基础模型

#### 2.2.1.1 循环神经网络

##### 2.2.1.1.1 RNN

##### 2.2.1.1.2 LSTM-三个门控单元



其中，空心点表示Hadamard乘积(同阶矩阵，对应元素相乘)，[]表示矩阵/向量相乘。

其实LSTM的思路很清晰：2条记忆线、一个输入、一个输出。

**历史信号**：进来后，与遗忘门相乘决定多大程度采用历史信息。

**记忆单元**：用双曲激活当前输入后，和更新门相乘再和记忆相加→Ct。

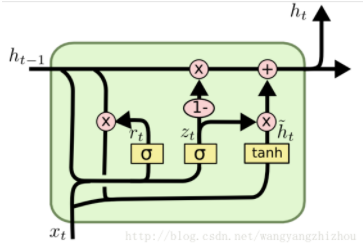
**输出信号**：将选择的历史信息和当前记忆单元求和，然后和输出门相乘决定最后的输出信号。3个门的输入都是a<t-1>和xt。

1、普通的RNN之所以不能捕获长期记忆，是因为W连乘问题导致梯度消失。

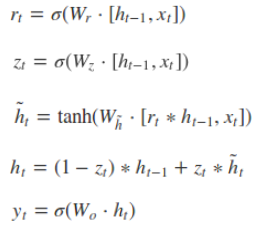
2、LSTM和普通RNN的相比之所以能缓解梯度问题，本质上讲在于信号在不同的时间步之间传播时，其数值不在取决于一个单一的矩阵，而是将数值问题的压力分散到更多的矩阵，抽象一层来讲就是由于门控机制的引入，使得理论上而言当遗忘门数值接近1的时候可以顺利的将历史信息引入并传递下去。

PS：关于激活函数的选取：门控值采用sigmoid函数，候选记忆采用双曲正切。原因解释：为了符合门控的物理定义有必要采取“开关函数”，而选用tanh在于大多场景下特征分布以0为中心，而tanh在0附近有较大的梯度，有利于参数的快速收敛。

##### 2.2.1.1.3 GRU-两个门控单元



没有标明的地方，数据均是往右/下流动，重画 #Todo



其中\*表示Hadamard乘积，.[]表示矩阵/向量乘积，关于GRU的维度：首先隐层会和r、z做Hadamard乘积，所以必定以隐层的维度为目标，故而在GRU内部，Tensor均转化为Hidden Size，理论上有3个矩阵，工程实现上只有两个矩阵，因为显然前两个可以合并。

对于GRU而言，input size namely the size of Xt don't have to equal to hidden size,cause matrix Wwill change the input size to hidden size

out\_put also have the same dimmention with hidden size

in short hidden size is everything

python 文件命名以字母开头，如果以数字开头的话无法import

对于单层GRU而言：

# 输入：seq\_len，batch\_size，input\_size

# 隐层：layers, batch\_size，hidden\_size

# 输出：seq\_len，batch\_size，hidden\_size

# 隐层：layers, batch\_size，hidden\_size

##### 2.2.1.1.4 Esim文本匹配利器

1、编码-通过Bi-Lstm

2、局部推断-attention

3、推断融合-再经过Bi-Lstm

4、池化推断-将各个时间步的向量进行平均池/最大池，然后拼接成一个向量。

5、预测

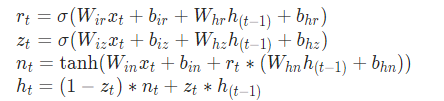
##### 2.2.1.1.5 Dssm

根据余弦相似度来做文本匹配问题。

输入--MLP--特征向量--余弦距离

##### ##### RNN类模型框架使用方法

###### ###### GRU模型



关于GRU的维度：首先隐层会和r、z做Hadamard乘积，所以必定以隐层的维度为目标，故而在GRU内部，Tensor均转化为Hidden Size，理论上有6个矩阵，工程实现上只有两个矩阵，因为有3个矩阵与Xt想乘，3个矩阵与隐向量相乘。

对于GRU而言，X与H的维度不必一致，因为W会把维度转化为H维

out\_put also have the same dimmention with hidden size

python 文件命名以字母开头，如果以数字开头的话无法import

对于单层GRU而言：

# 输入：seq\_len，batch\_size，input\_size

# 隐层：layers, batch\_size，hidden\_size

# 输出：seq\_len，batch\_size，hidden\_size

# 隐层：layers, batch\_size，hidden\_size

无论batch\_first是否为True，隐层的batch\_size都在中间，但是当batch\_size为True的时候，输入的维度应该是batch\_size， seq\_len， input\_size，默认情况下batch\_size为False

变长数据压缩计算问题：黑色为手动处理，蓝色为函数调用

**--- padding等长 （collection函数完成）**

**--- 按照长度倒排，并指示实际长度**

**--- pack\_padded函数生成压缩序列（Packed sequence对象）**

**---rnn前向压缩计算**

**---数据填充pad\_packed**

**理想的方式是可以输入变长的数据即再封装一层**，但是现在还不可以，需要手动处理。为了批量的处理数据，必须让同一个batch的数据等长即：

[batch\_size, seq\_len, hidden\_size] # batch\_first 为true

但是这样做的话会带来错误的数据：由公式可以看出

h\_t = f(W\_{ih} x\_t + b\_{ih} + W\_{hh} h\_{(t-1)} + b\_{hh})

就算x padding的数据为0， 但是由于隐层数据的存在，最终h{t-1}会和权重相乘，在padding为0的位置产生数据，所以解决办法就是：告诉rnn数据在每一个step的实际长度，然后在前向传播的过程中直接跳过padding的位置，即压缩计算，得到压缩的结果后，将数据解压即padding回原来的shape。

PackedSequence里的data和batch\_size其实就是把我们的输入数据重新整理打包成data，同时根据我们传入的 seqlist 计算batch\_size，然后，RNN会根据batch\_size从打包好的data里面取数据，然后一遍遍的执行 forward 函数。

最后要注意一点，因为pack\_padded\_sequence把输入数据按照 seq\_lenseq\_len 从大到小重新排序了，所以后面在计算 loss 的时候，要么把output的顺序重新调整回去，要么把 target 数据的顺序也按照新的 seq\_lenseq\_len 重新排序。代码中已经调整回原来的顺序了。

当 target 是 label 时，调整起来还算方便，但如果 target 也是序列类型的数据，可能会多点体力活。例如：

len=3245

idx\_sort = 3201 -->5432

idx\_unsort = 2310(长度为3的，长度为2的，长度为4的长度为5的)刚好对应原来的顺序

**为什么要倒排：**

这是与seq的计算原理相配合的，首先rnn在内存中的存放形式是同一step的数据放在一块，接着是下一step的数据，这样方便并行数据传输。由于数据是padding的，为了数据采集时候规避填充的数据采用的方法是：根据传入的长度指示计算出每一步实际所需采样的数据，例如序列长度为35,26,17,17,17,13,13,13

在0-12step： batchsize重设为8（没变）

在13-16step：batchsize重设为5

在17-26step：batchsize重设为2

在27-34step：batchsize重设为1

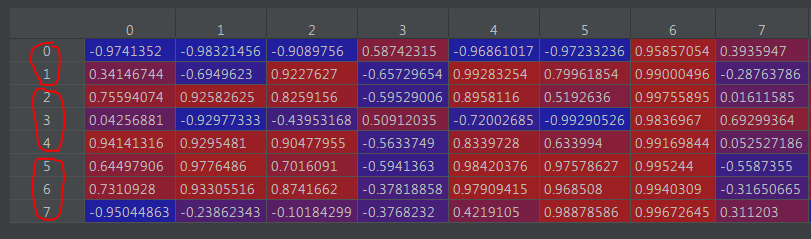
数据还是padding后的张量，只不过采集的时候忽略了某些数据，也就是说batch中的数据是分批结束的。

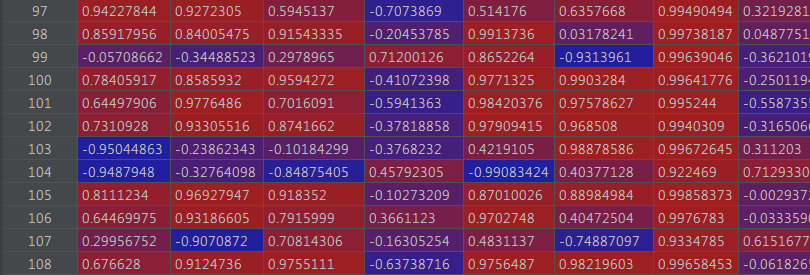
关于隐层与输出：前向传播后，返回y\_packed（packedSequence对象），h\_n

·y\_packed：是前向传播时计算的直接结果，继续上面的例子，那么结果为151\*50的张量，151=35+26+（17+13）\*3，另外带有batch\_size属性，为后面可能需要的y\_padding做好准备。

·h\_n：（num\_layers，batch\_size，embedding\_size）想其他hidden\_state一样，隐层仍然是最后一步所有层的输出，低层在低位，高层在高位。由于有些序列提前结束，所以最后一步是有效数据的最后一步。

数据实例：上图为最后一层的最后一步输出，下图为压缩序列的输出。由于数据经过了倒排，所以567就是长度为13的数据，234为长度 为17的数据，以长度为13为例，由于每个序列的长度都包含前13步，所以567对应的output为18\*8-1=103，然后101,102,103即为这三个序列的隐层最后输出。





**last\_h = torch.index\_select(h\_n[-1], dim=0, index=idx\_unsort) #** 求出最后一层的隐层输出

output的填充：由于y\_packed是压缩后的结果，现在把他重新padding回（batch，seq，hidden\_size）

**output = torch.index\_select(y\_sort, dim=0, index=idx\_unsort)**

至此，完工。

#### 2.2.1.2 卷积神经网络

#### #### 归一化

##### ##### Batch Normnization

##### ##### Layer Normalization

##### ##### Drop Out

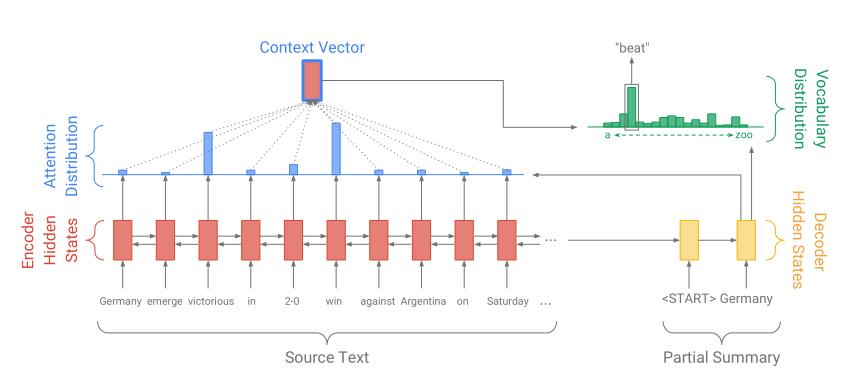
使得部分神经元参与参数的更新，相当于bagging的效果

测试的时候全部激活，但是计算值乘以衰减因子keep\_prob

BN与dropOut都具有正则化的效果，但是一般情况下，两者是不兼容的，一种解释是说网络状态在转换的时候会造成方差的偏移。

### ### NLP

#### #### 摘要模型



Baseline sequence-to-sequence model with attention

基线：encoder侧的隐层h作为k、v，然后decoder侧的隐层s作为q，基于weight sum求出语境向量context vector，将context与st concate之后送入全连接层，求出预测的单词。

##### #### 文献阅读

###### ###### 2019 综述：《基于序列到序列模型的生成式文本摘要研究综述》-安徽财经大学

基线：ABS attention based summarization

改进方案：

**编码侧：--隐向量的处理**

**选择性编码**：对每个step的隐层向量乘以一个衰减系数，系数由语境向量和h向量经过sigmoid生成，其实也是一种attention，只不多不是用解码查询，而是用语境向量查询。

**全局编码**：类似于ESIM的局部推断。

**层级编码**：又可以分为：句子、篇章、段落

句子：用句子的权重分布 更新单词的权重分布

段落：更精细的编码方案

**抽取辅助**：获取关键句的编码信息、基于ROUGE强化训练、关键词抽取[textRank]，然后获取关键词的向量，添加到attention机制中求注意力分布/**直接拼接用于预测**。

**事实抽取**：利用开源信息抽取工具OpenIE和句法句法分析工具[dependency parser]抽取原文的事实，然后分别抽取原文编码和事实编码。

**卷积编码**：ConvS2S 模型多层CNN，+ 残差 + 编解码attention

其他：添加语义特征：NER/POS/TFIDF

**解码侧：**

拷贝机制

重复控制

额外信息：变分模型获取隐层向量(类似于结构/摘要)、抽取实体构成主题向量用于辅助解码。

beam搜索

**训练：**

由最大化词项概率-->最小化摘要距离

基于ROUHE构建强化训练，结果是分数高了，但是可读性降低，所以后来又混合。

focalLoss

**总结：**

一般使用gru/LSTM取代RNN，gru结构更加简单，但是lstm的表达能力更强，所以一般短文本用GRU，长文本用LSTM。

卷积/多层LSTM多层网络可以构建原文本的分布特征、捕捉长距离依赖，但是训练时间更长。所以目前还是单层的比较多。

长文本摘要中，1/3采用了抽取辅助，添加知识驱动、先验知识可以帮助模型更好的理解文本。

未来方向：注意力、先验知识、辅助信息

###### ###### 2017 Get To The Point: Summarization with Pointer-Generator Networks-Stanford University-指针网络

**论文摘要：**

当前sequence2sequence modle 存在的问题与解决办法：

问题1:复制原文不准确的信息

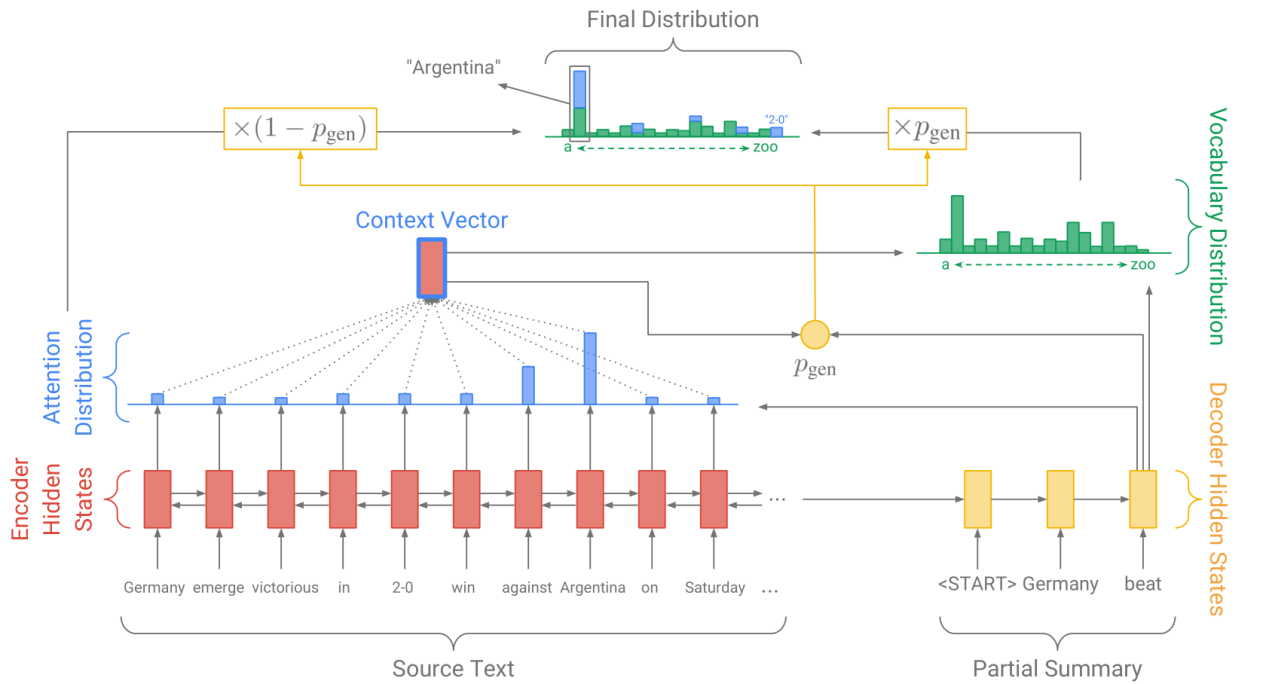
问题2:生成词自己重复

解决方案：pointer-generator network

1、pointer net：从原文复制使得能够准确的复现原文关键信息；另外generate net使得网络具有生成新词的能力；

2、利用coverage追踪已经生成的词语来缓解摘要重复问题。(从机器翻译引入)

**模型：Pointer-Generator-Model**



Pointer-generator model

attention参数和语境向量的计算和baseline一样，开关函数的公式为：



其中ht\*表示语境向量，st表示解码器输出，xt表示解码器输入，其他参数可以学习。每一个词语的生成概率公式为：



Pvocab表示全连接层输出的词表中每一个词的概率，ai为每个单词的attention得分。所以未登录词也会有得分。

覆盖机制：为了缓解解码时候的自我复制问题，借鉴MT(机器翻译)模型中的coverage机制，具体而言，在每一个解码时间步，维护一个覆盖权重：



attention机制也修改为：



其中h仍然是step的隐层向量，st为解码侧隐层向量，ct为覆盖向量。最后的分布权重仍然为at=softmax(et)。

损失函数修改为：



ct与at分别为历史总和与当前值，作为损失函数那么就希望覆盖向量变小，但是在每一个时间步at的和为softmax求和为定值1，所以对已经有很多注意力的位置不利。

**数据集**

CNN/Daily Mail Dataset：新闻内容平均781个词语，摘要平均56个词语，

287,226(28万)training pairs, 13,368(1万) validation pairs and 11,490(1万)test pairs

**实验**

1、并不是一开始就加上***覆盖机制***。一开始就假如覆盖机制的话会降低模型的表现。

2、只把覆盖向量添加到attention机制中，而不添加到损失函数中的话作用不大。

3、使用128维度的词向量，但是不是从头开始训练的，对比模型是从头开始训练的。

数据集：https://github.com/abisee/cnn-dailymail

python2代码：https://github.com/abisee/pointer-generator

python3代码：https://github.com/becxer/pointer-generator/

###### ###### 2019《基于改进Sequence-to-Sequence模型的文本摘要生成方法》-北京林业大学.周健

**背景：**

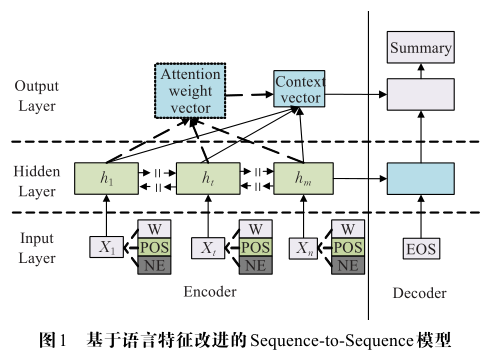
问题1：模型不能充分利用文本的语言特征信息； 对策1：加入语言特征

问题2：生成未登录词，影响阅读；对策2：利用copyNet缓解该问题

**什么是语言特征？**

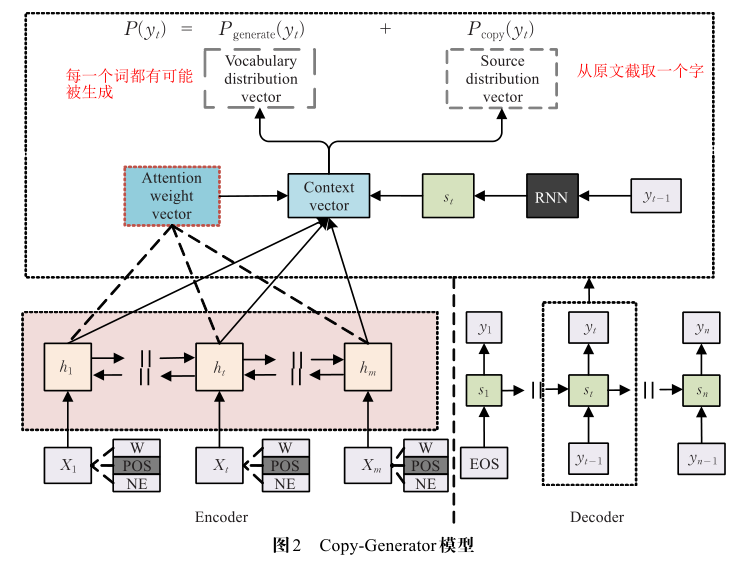
语言特征包括：词性、命名实体等，PS文本还具有统计特征包括tfidf、语义特征等等

**基本模型结构：**



Encoder和Decoder均采用RNN实现。首先在编码阶段，将词向量、词性类别向量、NE类别向量concate、然后就出每一个step的隐层向量。解码的时候，基本的做法是将encoder最后一个step作为隐层输入，然后逐个的生成摘要内容。引入attention的做法是：在每一个解码时间步，将解码隐层作为q，每一个编码隐层作为k、v，然后将两个向量concate过全连接求单词。

**copy-generator模型：**

****

在解码阶段，每个时间步的预测词yt可由两种方式产生：生成模式和拷贝模式。

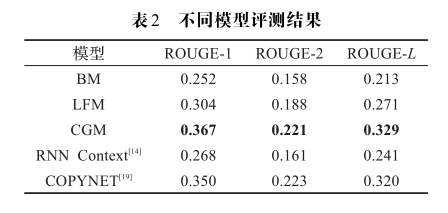
生成模式：基于encoder的隐藏状态和当前decoder步的隐层状态来求出预测词。

拷贝模式：利用两侧的注意力机制求内积，分数高的词被选中。

利用开关机制来决定模式选择：开关量的取值由编码隐向量、解码意向量

实验结果如何？和谁比较有提升？

数据集采用LCSTS，实验结果如下：其中BM表示编码和解码+attention模型，LFM表示BM+语言特征；CGM表示copy-generate model，CopyNet为经典模型。



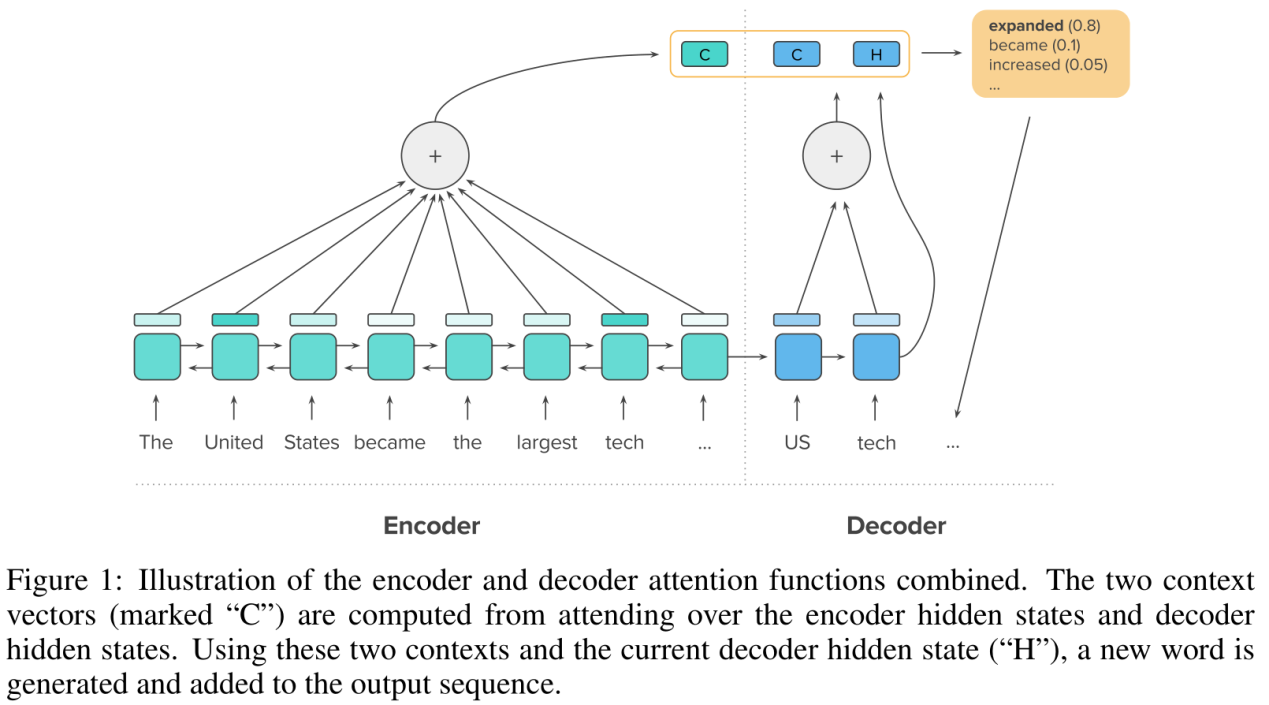
词向量表示方法？

word2vec向量，非关键词用特殊标记代替：（1）NAME\_EN — 英文标记；

（2）NUM — 数字标记；（3）DATE — 日期标志；（4）UNK — 未登录词；

（5）EOS — 文本结束标记

###### ###### 2017 A DEEP REINFORCED MODEL FOR ABSTRACTIVE SUMMARIZATION



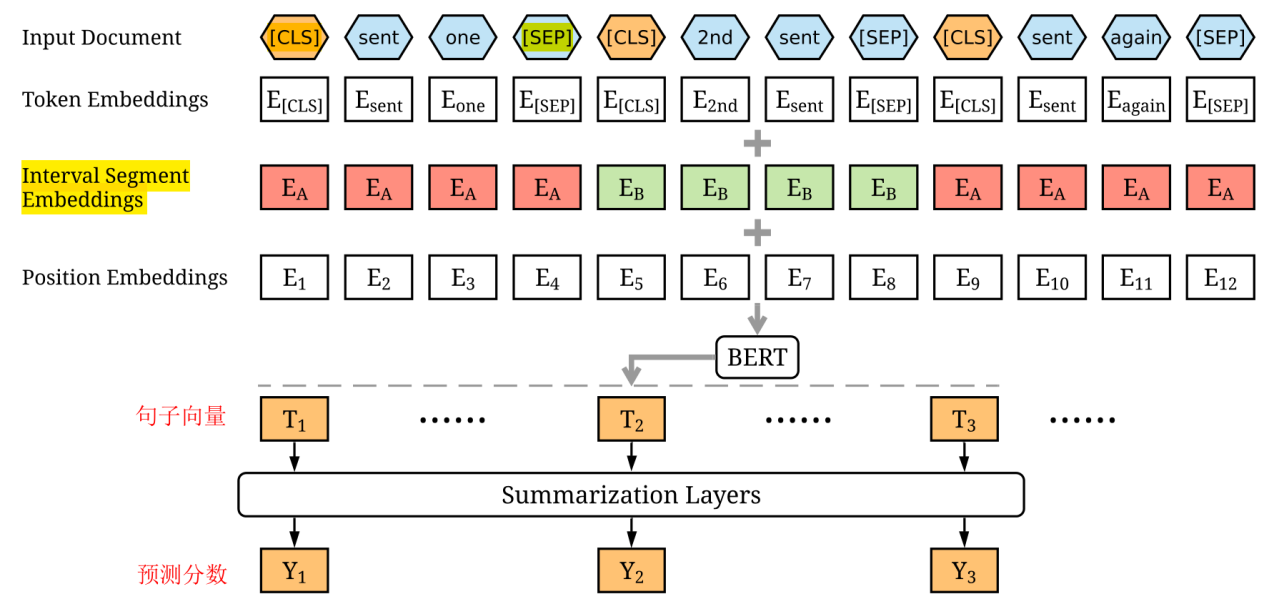
内注意力：除了解码对编码求attention，不同解码时间步之间求attention，然后连接两侧的atention以及当前step的h预测单词。

###### ###### 2019《Fine-tune BERT for Extractive Summarization》

摘要：我们要基于BERT模型做 抽取式摘要 模型，在CNN和NYT(纽约时报)两个数据集上评测结果显示效果最好[比较对象没有基于bert做的，要么是前三句，要么是指针]所以实验结果没有什么说服力。但是可以看看思路模型。

方法：将文章拆分为若干句，然后给每个句子打分为0/1，1表示要抽取这句话。

模型:

 **Encoding Multiple Sentences:**首先把文本分割为若干句话，并且用CLS和SEP框起来；

**Token Embeddings**：获取token 预训练embedding；

**Interval Segment Embeddings**：根据句子的奇偶序，分配为EA或者EB；

**Position Embeddings**：位置编码

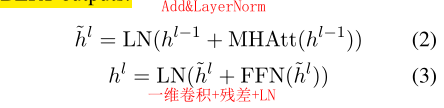
**Ti：**CLS位置对应的语境向量，有几句话就有几个语境向量；

**Yi：**sentence i的分数。

文中采用了3中方法求Y分数：直接对Ti向量打分、先过Transformer的编码结构[所谓的summarization layer]再打分、先过LSTM再打分。

疑问：segment直接标记为0或者1吗，为什么要用A、B标识？？？

**着重看下用transformer编码做摘要层的结构，因为这个效果最好：**

****

h0：就是bert输出，然后加上位置编码；

接着过两层编码(内部就是 多头attention、残差、LN、一维卷积)；

然后出来应该shape：句子个数，d\_model；

接着用sigmoid对每一句话打分。

**数据集：**由于数据集不是为抽取式模型设计的，所以label没办法直接使用，首先首贪心的方法，从原文的句子中选出和标准摘要Gouge分数最高的集合，以此为label。不太明白到底是怎么生成的。

**优点**：为bert式摘要提供baseline

**缺点**:没有控制变量，应该和bert系列的照耀模型做对比，不然没有说服力。

问题：序列化函数是否必要?

###### ##### 2019《Self-Supervised Learning for Contextualized Extractive Summarization》

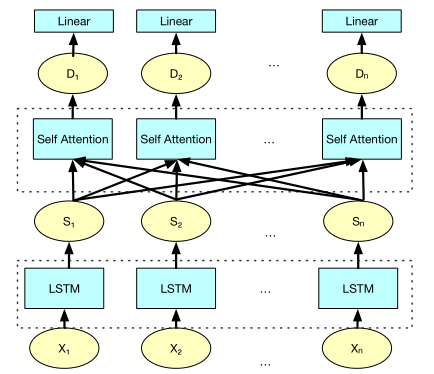
mask：随机的遮住句子，从候选池中预测被mask的句子

replace：将文本中的句子随机被替换为别的文章中的句子，来判断是否有句子被replace

switch：将文本中的句子随机打乱，判断文本中的句子是否被打乱

**贡献**：全文语境训练、3个预训练任务

**Basic Modle**

****

每句话经过BiLstm编码为语境向量[最后一步的输出]Si；

S向量经过Transformer的selfAttention层输出每句话的文档级向量Di；

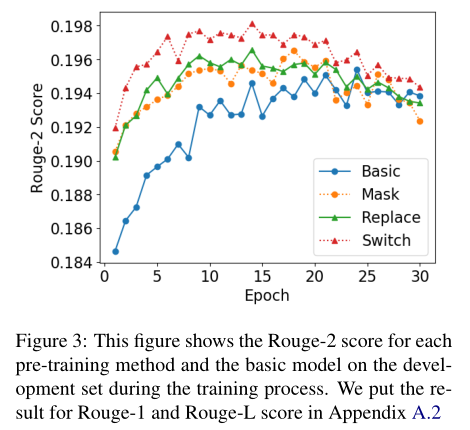
最后经过线性层 判断是否要把这句话作为摘要句。

预训练方法：通过预训练任务，使得模型学会语境级别的向量用于下游任务

MASK：将文本汇中的某些sentence用UNK遮住，然后用模型抽取出对应Di向量，候选池中有若干句话，用LSTM抽取出这些句子的语境向量，然后使用预选相似度、Margin Loss训练模型。

Replace:判断有没有句子被换掉

Switch：判断是否有句子打乱位置



实验：

词向量：glove-100维度

单层LSTM做句子编码、隐层向量200

5层4头的Encoder做编码来抽取句子级别的向量

训练分为两个阶段：1、预训练即用CNN原始无标数据做；2、用句子标签做摘要任务，句子标签怎么打呢？是摘要句打为1否则打为0.可是怎么选择句子呢？

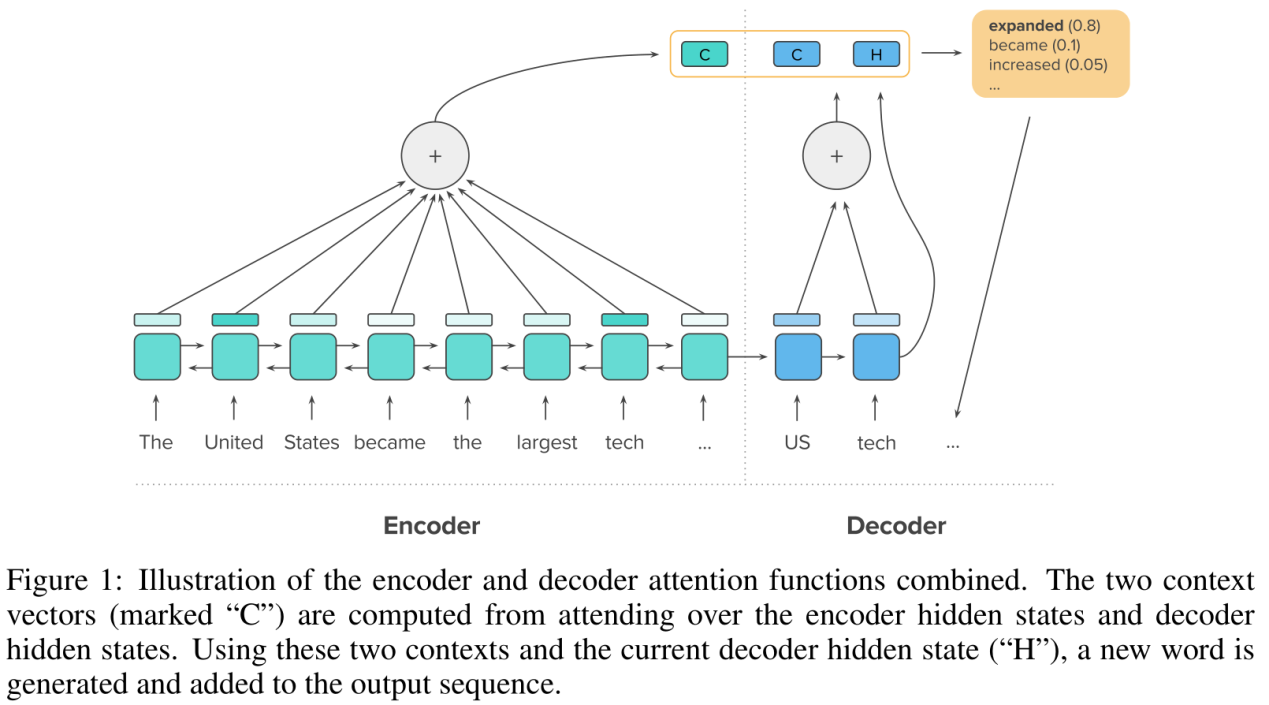
预训练学习率：0.0001、摘要学习率0.00001

预训练停止的条件是：收敛/训练达到30圈、3个预训练mask、replace、switch的概率为25%。

In the terms of **convergence**, the Mask, Replaceand Switch task takes 21,24,17 epochs in thetraining phase respectively, and 18,13,9 epochsto achieve the best performance in the fine-tunephase.

#### #### 摘要创新point

更加精细的attention机制：正交分解/胶囊网络/句子编码用于解码attention。



强化学习

注意力、先验知识、辅助信息

POS分析、NER

熵？

打算用抽取式模型做文本摘要：一是根据勘察用抽取式可以满足大部分微博类文档的摘要抽取工作，而是生成式难度太大。

为什么生成式摘要难度大：NLP目前还不成熟、缺少语法分析、指代消解等、大部分仍然是采用基于Attention和编解码结构进行逐步的输出，但是文本本身干扰性很大、因为模型的attention机制还比较弱。

计划：抽取式模型、数据收集、生成式模型[融合]。

那么抽取式可以做哪些工作呢?

1、预训练 2、编码与attention 3、解码去冗beamSearch等

如果是基于抽取式的话，我们的目的是更好的表达句子向量，并且在此基础上做匹配、打分等工作。可以任务从很多候选人之中抽取出代表-Representation[帮派]。

更好的表达句子向量，或者说找到逻辑相关的句子向量，就要达到以下的效果：

1、显性关联：文本相似度高

2、隐形关联：逻辑关联

所以可以做一些**句子级别的前置任务**来迁移到文本摘要上。

有人做了mask、replace、switch，还可以做：

文本排序：两句话或者更多

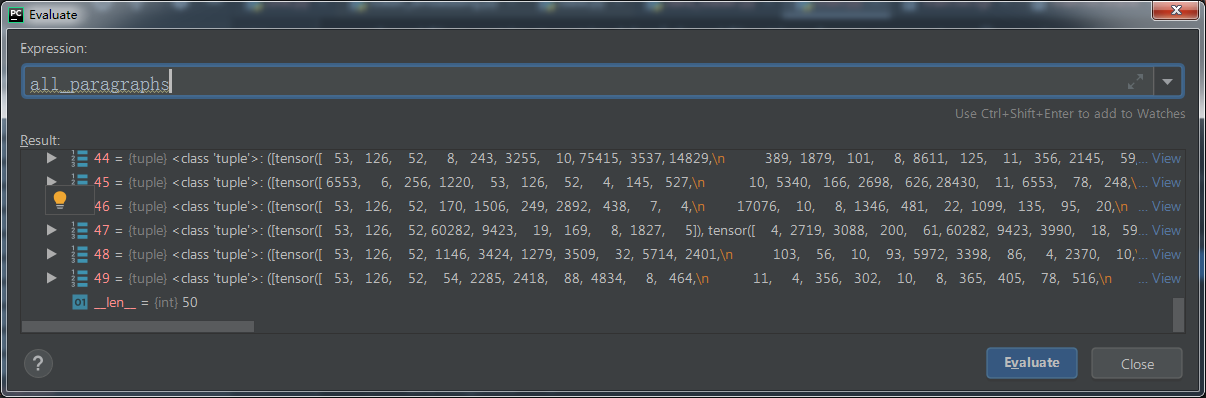
文本填充：用一句话前部分的内容，生成后面的内容

磁化模型：先生成磁铁，然后吸引后面的词语、句子。**难点在于如何确定哪些可以构成磁铁？**attention本质是让被选中的token倾向于调整到平行，因为这样内积才比较大。不用担心一些中性词被带偏，因为他们本身会和很多词共现。所以磁铁还是训练出来。而且可以考虑Esim、正交分解等等。

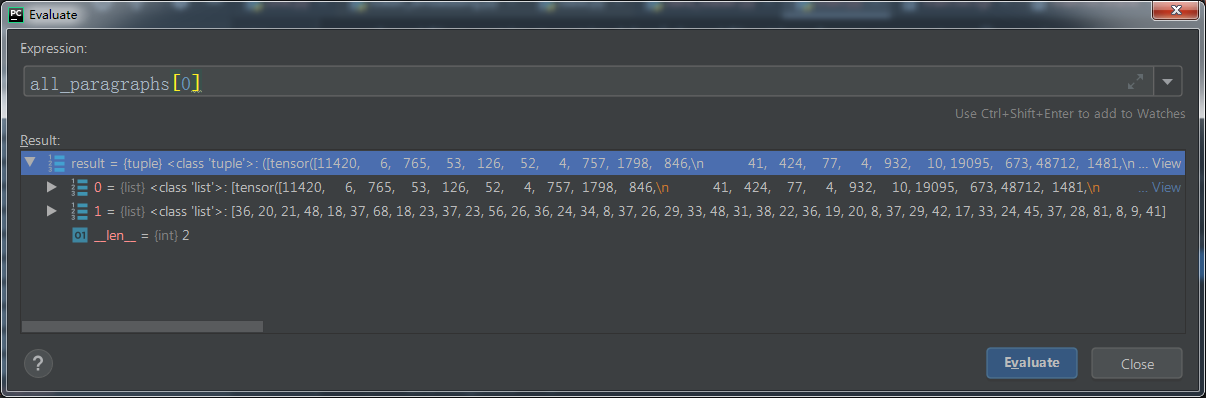
#### #### 论文复现笔记

基于bert的抽取模型-SwitchSum

train函数的all\_paragraphs

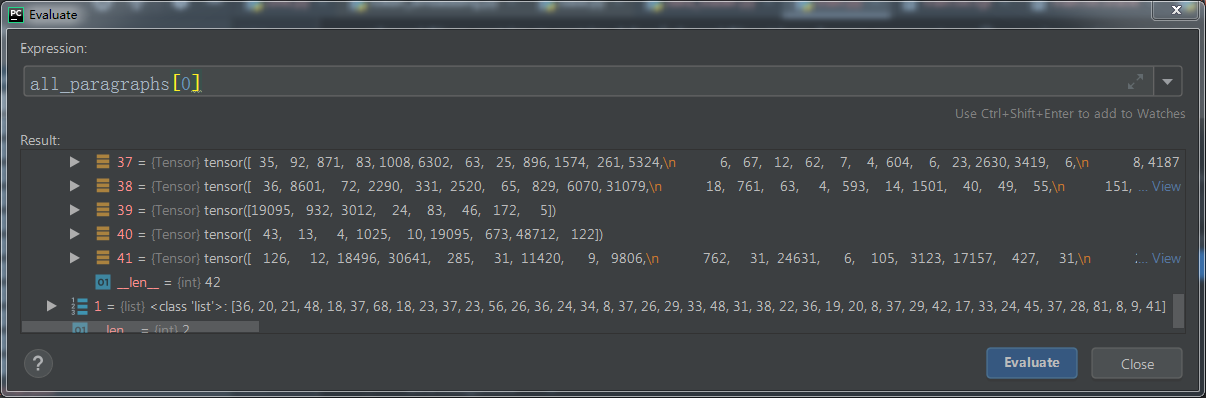


是一个元组列表，每一个元组对应一篇文章。



元组有两个列表构成：

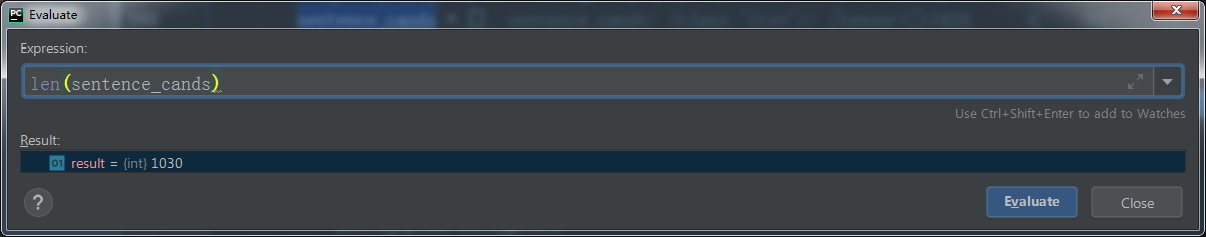
索引列表：每句话的index张量构成列表。



长度列表：表名每句话的长度。

数据集--文章--句子(索引+长度)

sentence\_cans：返回数据集所有的句子，因为all\_paragraphs[i][0]返回句子的index Tensor列表，用 += 操作相当于抽取所有的句子到一个大的列表中。demo的训练集共有50篇文章，句子共有1030个。



switch\_within\_para：

origin\_idx：[2,5,4,6]

cand\_idx： [2,4,6,5]

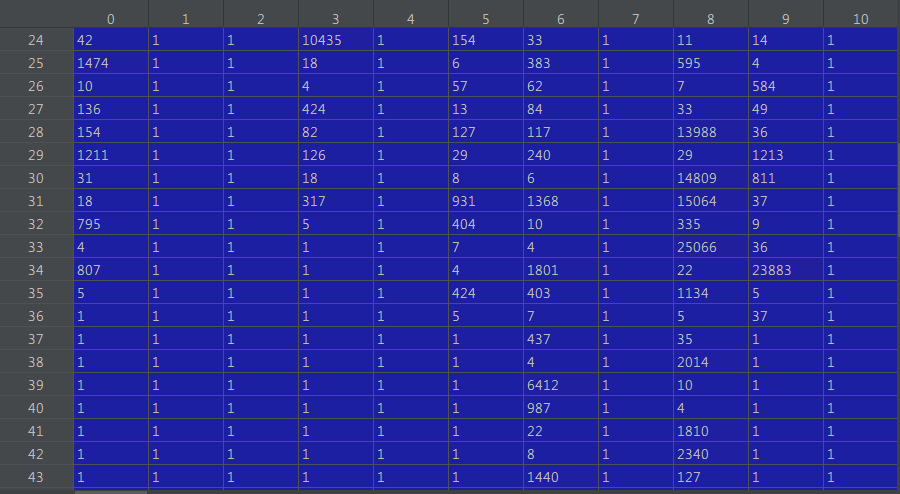
通过这样的办法，造成switch。本质上是利用shuffle前后。

myModle的forward函数：

paragraphs仍然是元组列表：内容为句子的index-tensor、长度列表；

经过pack\_paragraph之后，返回3个容器：批次数据的文章句子数长度、批量句子长度列表、句子会中；

经过rnn的pad\_sequence之后：



每句话对应一列，用设定的1pad。

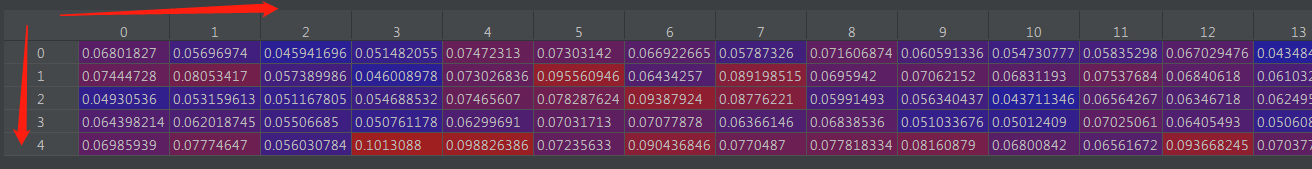
之后跳转到LSTM的forward函数：

lstm的返回是什么呢/

hn:layers\*direcrions, batch, hiddenSize

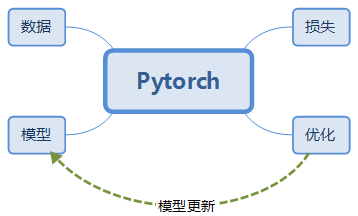
cn:layers\*directions, batch, hiddenSize

句子打分：



2.3框架与常用工具包

### 2.3.1 pytorch



**网络 →state\_dict（） 函数 # 状态字典**

**优化器→param\_groups属性 # 参数组（由param构成）**

**weight参数可以设置不同类别的权重**

By Knight

#### ####、数据

##### #####、dataset

首先在*初始化*的时候要定义一个数据列表，该列表的结构为：

·[(x1，lanel1),(x2，label2),...]，这样的话每次用索引就可以访问一组数据

·x：如果是文本数据，例如词向量，那么**x就是（seq\_len，embedding\_size）的二维张量**。如果是图片数据，就会使（通道，长，宽）三维tensor。

接着重写getiten和len函数：

·getitem接受一个索引号，并返回该索引对应的数据元组

##### #####、dataloder

##### #####、collet\_fn（batch）

针对rnn的padding问题，重写该方法使得批次数据等长。

batch数据 ---> 等长处理 ---> 返回

#### ####、模型

this\_net是一个实例，实例（）就是调用call方法，在父类的call方法中，调用了forward函数，所以等价于直接调用forward，但是父类考虑了更多的情况。

##### #####、CNN

##### #####、RNN

变长数据压缩计算问题：黑色为手动处理，蓝色为函数调用

**--- padding等长 （collection函数完成）**

**--- 按照长度倒排，并指示实际长度**

**--- pack\_padded函数生成压缩序列（Packed sequence对象）**

**---rnn前向压缩计算**

**---数据填充pad\_packed**

**理想的方式是可以输入变长的数据即再封装一层**，但是现在还不可以，需要手动处理。为了批量的处理数据，必须让同一个batch的数据等长即：

[batch\_size, seq\_len, hidden\_size] # batch\_first 为true

但是这样做的话会带来错误的数据：由公式可以看出

h\_t = f(W\_{ih} x\_t + b\_{ih} + W\_{hh} h\_{(t-1)} + b\_{hh})

就算x padding的数据为0， 但是由于隐层数据的存在，最终h{t-1}会和权重相乘，在padding为0的位置产生数据，所以解决办法就是：告诉rnn数据在每一个step的实际长度，然后在前向传播的过程中直接跳过padding的位置，即压缩计算，得到压缩的结果后，将数据解压即padding回原来的shape。

PackedSequence里的data和batch\_size其实就是把我们的输入数据重新整理打包成data，同时根据我们传入的 seqlist 计算batch\_size，然后，RNN会根据batch\_size从打包好的data里面取数据，然后一遍遍的执行 forward 函数。

最后要注意一点，因为pack\_padded\_sequence把输入数据按照 seq\_lenseq\_len 从大到小重新排序了，所以后面在计算 loss 的时候，要么把output的顺序重新调整回去，要么把 target 数据的顺序也按照新的 seq\_lenseq\_len 重新排序。代码中已经调整回原来的顺序了。

当 target 是 label 时，调整起来还算方便，但如果 target 也是序列类型的数据，可能会多点体力活。例如：

len=3245

idx\_sort = 3201 -->5432

idx\_unsort = 2310(长度为3的，长度为2的，长度为4的长度为5的)刚好对应原来的顺序

**为什么要倒排：**

这是与seq的计算原理相配合的，首先rnn在内存中的存放形式是同一step的数据放在一块，接着是下一step的数据，这样方便并行数据传输。由于数据是padding的，为了数据采集时候规避填充的数据采用的方法是：根据传入的长度指示计算出每一步实际所需采样的数据，例如序列长度为35,26,17,17,17,13,13,13

在0-12step： batchsize重设为8（没变）

在13-16step：batchsize重设为5

在17-26step：batchsize重设为2

在27-34step：batchsize重设为1

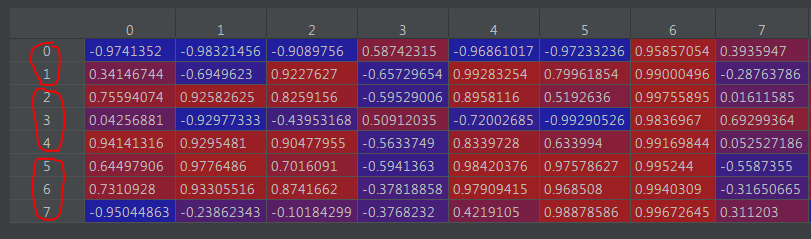
数据还是padding后的张量，只不过采集的时候忽略了某些数据，也就是说batch中的数据是分批结束的。

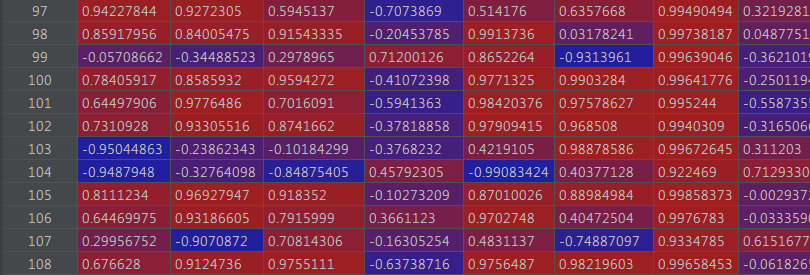
关于隐层与输出：前向传播后，返回y\_packed（packedSequence对象），h\_n

·y\_packed：是前向传播时计算的直接结果，继续上面的例子，那么结果为151\*50的张量，151=35+26+（17+13）\*3，另外带有batch\_size属性，为后面可能需要的y\_padding做好准备。

·h\_n：（num\_layers，batch\_size，embedding\_size）想其他hidden\_state一样，隐层仍然是最后一步所有层的输出，低层在低位，高层在高位。由于有些序列提前结束，所以最后一步是有效数据的最后一步。

数据实例：上图为最后一层的最后一步输出，下图为压缩序列的输出。由于数据经过了倒排，所以567就是长度为13的数据，234为长度 为17的数据，以长度为13为例，由于每个序列的长度都包含前13步，所以567对应的output为18\*8-1=103，然后101,102,103即为这三个序列的隐层最后输出。





**last\_h = torch.index\_select(h\_n[-1], dim=0, index=idx\_unsort) #** 求出最后一层的隐层输出

output的填充：由于y\_packed是压缩后的结果，现在把他重新padding回（batch，seq，hidden\_size）

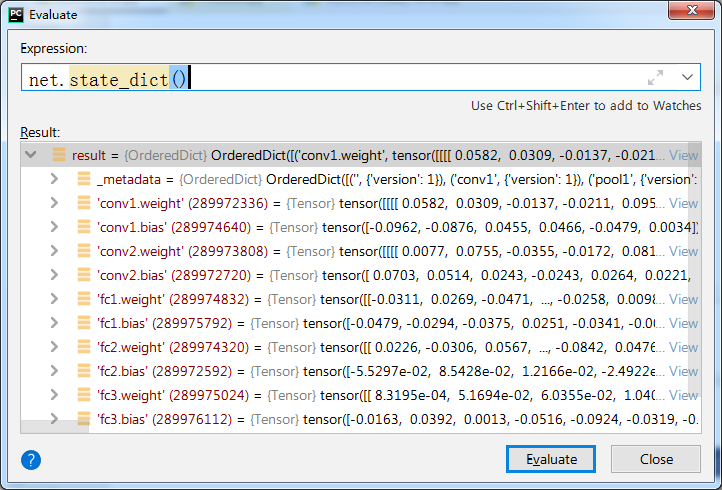
**output = torch.index\_select(y\_sort, dim=0, index=idx\_unsort)**

至此，完工。

##### ##### .Finetune

**模型状态字典-state\_dict**

返回一个order\_dict是一个有序的字典，是各个网络的weight和bias对应的张量。



**示例代码：**

pretrained\_model = **torch.load**('weights/Rnn\_params.pkl')

net\_state\_dict = net**.state\_dict()**

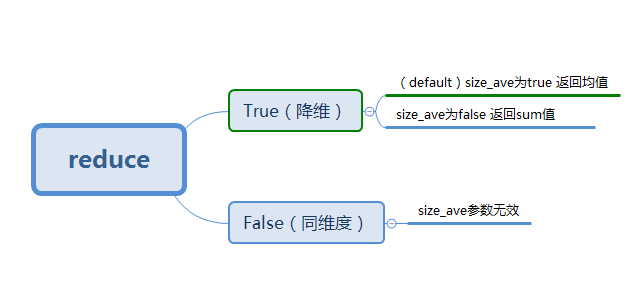
backup\_dict = {k:v for k,v in pretrained\_model.items() if k in net\_state\_dict.keys()}

*.keys可以省略*

net\_state\_dict**.update**(backup\_dict)

net.**load\_state\_dict**(net\_state\_dict)

#### ####、损失函数



**weight参数可以设置不同类别的权重。**

##### #####、L1loss：差的绝对值

##### #####、MSEloss：差的平方

##### #####、CrossEntropyLoss：分布交叉熵

原始的L=-sigma（y \* log y\_hat）等价于L= -log （y\_hat[class]）

实际上加上了softmax equal to -log（soft\_max(y\_hat) [class]）

CrossEntropyLoss（y\_hat, target）y\_hat 是矩阵，shape=（batch\_size, classifications\_num）

target是索引号对应的一维数组， Note：target是类别号码-->索引，不是被实际逼近的值。

例如三分类问题：最后预测值经过softmax函数后为[a,b,c],如果label为2，表示希望b逼近1，而不是2.

**参数：**

**weight** = torch.from\_numpy(np.array([0.6, 0.2, 0.2])).float()，常用语类别不平衡问题，类型必须是float，就是将默认weight为1 的结果，在分别乘以一个weight系数。而且不要求系数和归一。

**reduce**：对于批量样本，每一个sample都有一个loss，默认返回一个降维的实数。当为False是返回和label同纬度发tensor。

**ignore\_index**：忽略的类别，如果某个类别被忽略，那么将其相关的loss=0

ignore\_index = 1: tensor([1.0986, 0.0000, 1.0986], grad\_fn=<NllLossBackward>)

ignore\_index = 2: tensor([1.0986, 1.0986, 0.0000], grad\_fn=<NllLossBackward>)

##### #####.NLLLoss

常用于多分类任务，但是input在输入NLLLoss()之前，需要对input进行log\_softmax函数激活，即将input转换成概率分布的形式，并且取对数。其实这些步骤在CrossEntropyLoss中就有，如果不想让网络的最后一层是log\_softmax层的话，就可以采用CrossEntropyLoss完全代替此函数。

NLLLoss --> log\_softmax（softmax -->log）= 激活交叉熵

##### #####.PoissonNLLLoss

主要用于Target服从泊松分布的数据集。具体没有细看。

##### #####. KLDivLoss ：散度损失

KL散度：计算拟合与实际分布之间的KL散度，计算公式为：

Ln = yn \* （logyn - xn），这与分布之间的散度公式是不一样的，因为：

Ln = yn \* （logyn - logxn） # 真正的散度公式

所以，在torch中要先计算预测值的对数然后再feed进损失函数中。

一般的做法: soft-log --> KLDloss

降维为True， 平均为True， 然后对Batch求平均。

##### #####.BCELoss 二分类交叉熵

是交叉熵在二分类背景下的特例，就是将sigma函数换成了0,1两种情况。计算公式为：



但是实际上，由于是二分类所以：两个yn相加为1，两个Xn相加为1.所以就简写成了上式。数据要经过softmax处理再feed进来。

实际上是Ln即以自然对数为底计算的、拟合值和目标值要在[0,1 ]之间，但是不要求归一。

##### #####.BCEWithLogisLoss 带sigmoid的二分类交叉熵

在7之前加上sigmoid层

##### #####.MarginRankingLoss

运算是针对每一个维度进行的，如果两个值的差距在margin范围之内（当y等于1时），其损失为0， 否则为一个正数。计算公式为：



x1, x2是两个张量， y可以取任何数，不限制于+-1.

参数：

margin：

传统的reduce和size\_average不鼓励使用， 使用reduction：none（不降维）， sum， mean

#### ####、优化器

##### #####.SGD：随机梯度下降

(self, params, lr=required, momentum=0, dampening=0,  
 weight\_decay=0, nesterov=False)

现在，SGD通常是指Mini-Batch方法，而不是早期单Example的方法。

封装了momentum（指数平均加权）和nesterov（预判下一次的梯度，防止冲过头），其中momentum为float，一般设置为0.9。 nesterov为bool。

weight\_decay：权重衰减，L2正则化系数。

阻尼参数与nesterov只能使能一个。

*v\_{t+1} = mu \* v\_{t} + g\_{t+1}   
p\_{t+1} = p\_{t} - lr \* v\_{t+1}*

p：参数

g:：梯度

mu：动量系数

v:速度-->拟合的梯度值

与其他框架的区别，对历史信息乘了学习率。

##### #####..ASGD:随机平均梯度下降

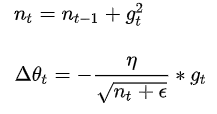
模型并行的分布到不同的机器中，然后参数平均。待学习......

<https://blog.csdn.net/xbinworld/article/details/74781605>

##### #####..AdaGrad

AdaGrad思路基本是借鉴L2 Regularizer，不过此时调节的不是WW，而是GradientGradient:

Adagrad其实是对学习率进行了一个约束。即：



此处，对gt从1到t进行一个递推形成一个约束项regularizer

特点：

·前期[公式]较小的时候， regularizer较大，能够放大梯度

·后期[公式]较大的时候，regularizer较小，能够约束梯度

·适合处理稀疏梯度

##### #####..Adadelta

不懂

##### #####..RMSprop

adgrad进阶版本，用窗口值取代所有的历史信息，防止学习衰减到0

##### #####..Adam

(self, params, lr=1e-3, betas=(0.9, 0.999), eps=1e-8,  
 weight\_decay=0, amsgrad=False

实现 Adam(Adaptive Moment Estimation))优化方法。Adam 是一种自适应学习率的优化方法，Adam 利用梯度的一阶矩估计和二阶矩估计动态的调整学习率。Adam 是结合了 Momentum 和 RMSprop，并进行了偏差修正。

第一个β是动量参数，第二个是RMS参数。lr默认为0.001，没有正则化。

##### ##### 属性：

###### ###### param\_groups：参数组，type=list

定义optimizer是的参数列表，对于整体的网络，可以单独设置优化器，同一个优化器也可以对不同的参数制定不同的参数。

参数组的设置：

optim.SGD( [ {'params': w1, 'lr': 0.1}, {'params': w2, 'lr': 0.001}] )，以字典列表的形式设置两组参数。

##### #####.方法：

###### ######1.state\_dict（）

{

**'state'**: packed\_state,

**'param\_groups'**: param\_groups, # 参数列表

}

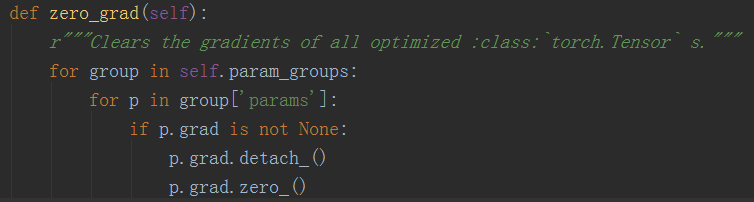
关于state\_dict：

1、params\_groups：与属性之param\_groups（字典列表）相比，将每一个字典其分为两部分，params（model的参数）和其他，其他的参数仍然以键值对的方式保存，将params键对应的value换成变量的实际地址列表（id函数）

2、packed\_state：对象的属性值相关的字典key为地址（针对张量属性），value为原先的value，但是debug的时候看到是空的。

这是优化器的state\_dict，model也有自己的dtate\_dict，而且字典的内容完全不一样，不管是优化器还是自定义的module都是继承了module类。

###### ######2.zero\_grad（）



一个因为param\_groups是字典列表，所以group就是一个dict，然后dict内部有两种变量：params和其他，针对params是一个列表，针对其中的每一个变量，如果有梯度，就把他置位。

What is detach\_（）?从计算图中分离，使之成为叶子结点，不可求导。

###### ######3.add\_params\_group（）

**optimizer\_1 = optim.SGD([w1, w2], lr=0.1)**

# 增加一个参数组

**optimizer\_1.add\_param\_group({'params': w3, 'lr': 0.001, 'momentum': 0.8, 'initial\_lr':0.1})**

可以看到，参数组是一个list，一个元素是一个dict，每个dict中都有lr, momentum等参数，这些都是可单独管理，单独设定，十分灵活！

###### ######4.steps（）

优化器初始化的时候，第一参数为指定的需要优化的参数params，然后执行steps函数时，会执行：p.data.add\_(-group['lr'], d\_p) # d\_p为数据的梯度值

#### ####、学习率

##### #####1.学习率衰减方法：

##### #####1. StepLR

等间隔调整，参数：

gaamma：衰减系数

step\_size: 以epoch为步进单位，每次将learining\_rate调整为gamma倍。

last\_epoch：默认情况下为-1，表示以optimizer的lr为衰减的初始值，如果设置为其他的值，可以使得学习率的衰减曲线移动，但是要在optimizer中包含属性initial\_lr属性（属性添加），否则会报错。

##### #####2.MultiStepLR

按照设定的间隔调整学习率，适合后期观察，定制学习率。参数：

milesones：list定义epoch调整的时机。

gamma：衰减系数，默认0.1

##### #####3.ExponentialLR

指数衰减：Lr = Lr \* gamma \*\*epoch，参数：

gammma、last\_epoch

##### #####4.CosineLingLR

##### #####5.RedeceLROnPlateau

plateau：高原稳定器，停滞期

当某个指标不在变化的时候调整学习率。参数：

mode：两种模式，min/max如果是min表示数据不在减小的时候调整，max相反。

factor：等价于gamma

patience：忍受该指标多少个epoch不变化后调整

verbose（唠叨的）：是否打印学习率调整信息。

threshhold：未知

cooldown：冷却时间int，

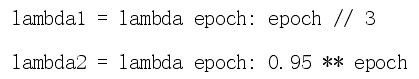
min\_lr：学习率下限int/list取决于参数组

eps：float衰减下限

##### #####6.LambdaLR：自定义

参数：

lr\_lambda：函数或函数列表[lambda1，lambda2]



##### #####2.学习率定制

学习率是优化器的参数，在学习优化器时，一个重要的想法就是：优化器是一个计算单元，它根据损失函数反向传播时参数产生的梯度对参数进行更新，优化器用参数组的概念来管理参数，可以为不同的层（积木）配置不同的学习率，momentum等参数。

某一层定制：

**示例代码：**

*# 将fc3层的参数从原始网络参数中剔除*

ignored\_params = list(**map**(**id**, net.fc3**.parameters()**)) # 可迭代对象

base\_params = **filter**(lambda p: **id(p)** not in ignored\_params, **net.parameters()**)

*# 为fc3层设置需要的学习率*

optimizer = optim.SGD(params=[ {'params': base\_params},

**{'params': net.fc3.parameters(), 'lr': lr\_init\*10}**], lr\_init, momentum=0.9, weight\_decay=1e-4)

**fc3两个学习率？**

# optimizer的add\_param\_group函数在设置优化器参数组的时候，会首先把除去params参数的部分统一加入到default数组，这些参数会包含到每一个参数组中。

# 设置每一个参数组的时候会调用setdefault(name, default)函数，只有那么没在param\_group中出现而又在default中的参数才会被设定，所以 fc3中的学习率不会变

# **优化器的params参数**：可以是model的parameters（）函数生成的迭代对象**列表**，也可以是源码中的其他形式，例如具体的张量

# 一个参数组：[{params:[w1, w2...]}}]

# 两个参数组：[{params=...，lr=0.1}，{params=...，lr=0.01}] 字典的形式表达参数组

##### #####2.学习率衰减定制

学习率调整，是针对learning\_rate做工作，可以使用简单的steplr，maxsteplr等方式，使得学习率在若干个epoch后衰减gamma倍。

可以用LambdaLR进行学习率是衰减个性化配置：以有两个参数组为例，当然也可以是一个参数组：

**lambda1** = lambda epoch: epoch // 3

**lambda2** = lambda epoch: 0.95 \*\* epoch

scheduler = torch.optim.lr\_scheduler.LambdaLR(optimizer, lr\_lambda=[**lambda1,**

**lambda2**])

### 2.3.2 tensorflow

### 2.3.3 pandas

### 2.3.4 numpy

### 2.3.5 matplotlib