

**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

# 一、数据

## 1、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

## 2、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

# 二、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。

## 1、CNN

## 2、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

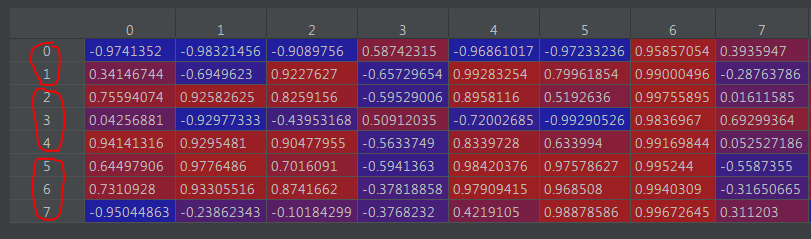
数据还是padding后的张量，只不过采集的时候忽略了某些数据，也就是说batch中的数据是分批结束的。

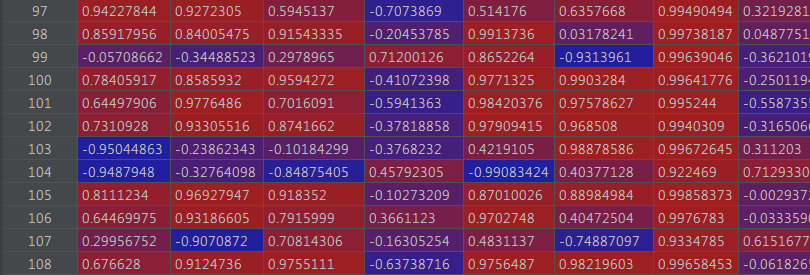
关于隐层与输出：前向传播后，返回y\_packed（packedSequence对象），h\_n

·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）

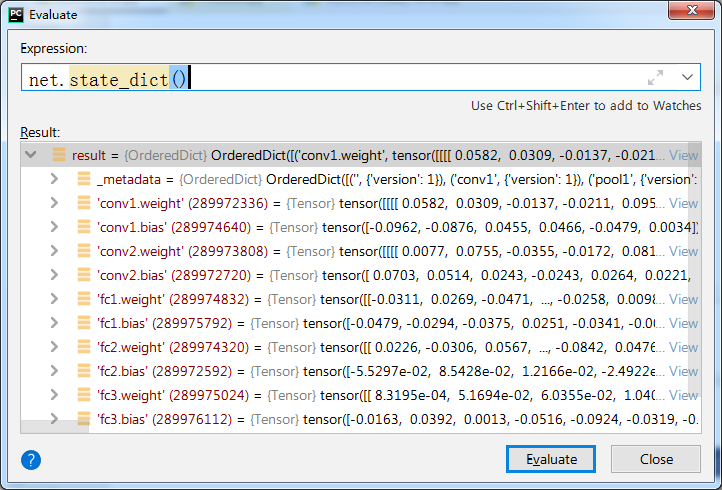
**output = torch.index\_select(y\_sort, dim=0, index=idx\_unsort)**

至此，完工。

## 3.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()**

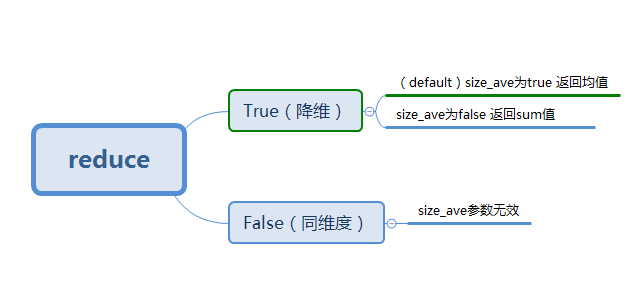
backup\_dict = {k:v for k,v in pretrained\_model.items() if k in net\_state\_dict.keys()}

*.keys可以省略*

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

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

# 三、损失函数



**weight参数可以设置不同类别的权重。**

## 1、L1loss：差的绝对值

## 2、MSEloss：差的平方

## 3、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>)

## 4.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

## 5.PoissonNLLLoss

主要用于Target服从泊松分布的数据集。具体没有细看。

## 6. 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。

## 7.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。

## 8.BCEWithLogisLoss 带sigmoid的二分类交叉熵

在7之前加上sigmoid层

## 9.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

# 四、优化器

## 1.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。

## 2.ASGD:随机平均梯度下降

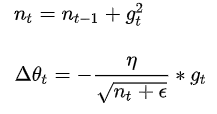
模型并行的分布到不同的机器中，然后参数平均。待学习......

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

## 3.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

## 4.Adadelta

不懂

## 5.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

## 6.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

}

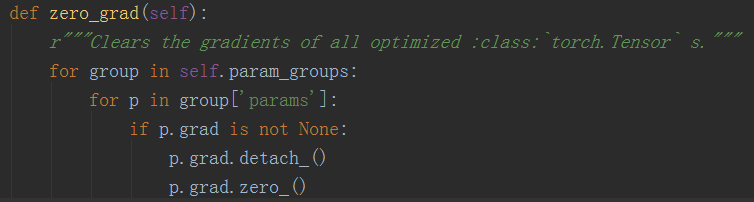
关于state\_dict：

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

# 五、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，

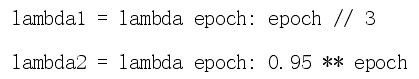
min\_lr：学习率下限int/list取决于参数组

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**])