Assignment 1

Files

data/

data_utils.py: Helper functions or classes used in data processing.

process.py: Process a raw dataset into a sample file.

model/

config.py: Define configuration parameters.

dataset.py: Define the format of samples used in the model.

evaluate.py: Evaluate the loss in the dev set.

model.py: Define the model.

predict.py: Generate a summary.

test_vocab.py: Testing vocab.

train.py: Train the model.

utils.py: Helper functions or classes used for the model.

vocal.py: Define the vocabulary object.

saved_model/

Save the trained model objects here.

files/

Place data files here.

runs/

Save logs here for TensorboardX.

TO-DO list:

必备资料

为了完成以下任务，我们需要逐步熟悉、掌握Pytorch框架，所以请大家在完成每一个模块时先查阅一下Pytorch的API文档，弄清楚要使用的模块是做什么的以及如何使用。此外，模型的具体实现，需要参见论文 *Get To The Point: Summarization with Pointer-Generator Networks*。

模块1: 词典处理

model/vocab.py:

任务1: 完成add_words函数。

向词典里加入一个新词，需要完成对word2index、index2word和word2count三个变量的更新。

model/dataset.py:

任务2: 完成PairDataset类中的build_vocab函数。

需要实现控制数据集词典的大小（从config.max_vocab_size）读取这一参数。建议使用python的collection模块中的Counter来做，这个数据类型跟dict很像，但有两个好处：

1. 加入新的key时不需要判断是否存在，会自动将其对应的值初始化为0。
2. 可以通过most_common函数来获取数量最多的k个key。

Counter的用法详见https://docs.python.org/2/library/collections.html。

model/utils.py:

任务3: 完成source2ids函数。

当我们训练好模型要对测试集进行测试时，测试集中的样本往往会包含OOV tokens。这个函数需要你将在词典中的token映射到相应的index，对于oov tokens则需要记录下来并返回。

任务4: 完成outputids2words函数。

与任务3相反的过程，将token id映射到对应的词，并输出。

测试

```
python test_vocab.py
```

示例输出

```
Reading dataset ../files/train.txt... 43995 pairs.
idx:  5
token:  的
vocab size:  20004
```

模块2: 模型搭建

model/model.py:

任务1: 完成Encoder。

1. 定义embedding层和BiLSTM层。

2. 实现前向传导（输入输出详见代码）。

   你可能用到的Pytorch模块：

   nn.Embedding

   nn.LSTM

任务2: 完成Decoder。

1. 定义embedding层和LSTM层（单向）；定义两个线性层（前馈层）W1和W2。
2. 实现前向传导。具体实现参见论文中的公式(4)和(5)。代码中会给出每一个步骤的提示。

   你可能用到的Pytorch模块：

   nn.Embedding

   nn.LSTM

   nn.Linear

   Tensor.view

   torch.cat

   nn.functional.softmax

任务3: 完成Attention。

1. 定义三个线性层Wh、Ws和v。维度详见论文中的公式(1)和(2)。
2. 定义前向传导。

   a. 处理decoder的隐状态h和c，将二者拼接得到s_t，并处理成合理的shape。

   b. 参考论文中的公式(1)和(2)，实现attention weights的计算。

   c. 由于训练过程中会对batch中的样本进行padding，对于进行了padding的输入我们需要把填充的位置的attention weights给过滤掉（padding mask），然后对剩下位置的attention weights进行归一化。

   d. 根据论文中的公式(3)计算context vector (hint: 可以使用torch.bmm)。

   你可能用到的Pytorch模块：

   nn.Linear

   Tensor.expand_as

   Tensor.view

   Tensor.squeeze

   torch.cat

   nn.functional.tanh

   nn.functional.softmax

   torch.bmm

任务4: 完成整个model的前向传导。

1. 对输入序列x进行处理，对于oov的token，需要将他们的index转换成<UNK> token (hint: 可以使用torch.where)。
2. 生成输入序列x的padding mask (hint: 可以使用torch.ne)。
3. 得到encoder的输出和隐状态，并对隐状态进行降维后作为decoder的初始隐状态。
4. 对于每一个time step，以输入序列y的y_t作为输入，y_t+1作为target，计算attention，然后用decoder得到p_vocab，找到target对应的词在p_vocab中对应的概率target_probs (hint: 可以使用torch.gather)，然后计算time step t的损失（NLL loss，详见论文公式(6)）。然后加上padding mask。
5. 计算整个序列的平均loss，详见论文公式(7)。
6. 计算整个batch的平均loss并返回。

   你可能用到的Pytorch模块：

   torch.full

   torch.ne

   Tensor.view

   Tensor.squeeze

   Tensor.unsqueeze

   torch.cat

   torch.gather

   torch.log

   torch.sum

   torch.mean

任务5 (optional)：

由于我们的encoder用了BiLSTM，而decoder用的是单向的LSTM，使用encoder的输出作为decoder初始隐状态时，需要对encoder的隐状态进行降维。实现的方式可以有多种，可以对两个方向的隐状态简单相加，也可以定义一个前馈层来做这个事。这里我们用一个ReduceState的模块以简单相加的形式来实现，具体见代码。你们可以对这个模块做一些更多的操作，设计自己觉得更为合理的方式来对encoder的隐状态进行处理。

```
class ReduceState(nn.Module):
    """
    Since the encoder has a bidirectional LSTM layer while the decoder has a
    unidirectional LSTM layer, we add this module to reduce the hidden states
    output by the encoder (merge two directions) before input the hidden
states
    nto the decoder.
    """
    def __init__(self):
        super(ReduceState, self).__init__()

    def forward(self, hidden):
```

```
        """The forward propagation of reduce state module.

        Args:
            hidden (tuple):
                Hidden states of encoder,
                each with shape (2, batch_size, hidden_units).

        Returns:
            tuple:
                Reduced hidden states,
                each with shape (1, batch_size, hidden_units).
        """
        h, c = hidden
        h_reduced = torch.sum(h, dim=0, keepdim=True)
        c_reduced = torch.sum(c, dim=0, keepdim=True)
        return (h_reduced, c_reduced)
```

测试

需完成模块3后进行测试。

模块3: 训练

model/train.py

任务1: 实现训练过程。

模板如下：

```
optimizer = None  # Choose an optimizer from torch.optim
batch_losses = []
for batch in batches:
    model.train()  # Sets the module in training mode.
    optimizer.zero_grad()  # Clear gradients.
    batch_loss = model(**params)# Calculate loss for a batch.
    batch_losses.append(loss.item())
    batch_loss.backward()  # Backpropagation.
    optimizer.step()  # Update weights.
epoch_loss = torch.mean(batch_losses)
```

需要注意的是，论文里的优化方式和训练参数未必适合我们的数据集，大家可以自己尝试选择合适的优化方式和训练参数，推荐使用Adam作为optimizer。

任务2: 在适当的位置实现梯度剪裁。

Hint: 参见torch.nn.utils模块下的clip_grad_norm_函数。

任务3: 用TensorboardX记录训练过程的损失并实现可视化。

Hint: 创建一个SummaryWriter对象，调用add_scalar函数来记录损失，记得写完要调用close函数。详见[TensorboardX](TensorboardX)文档。

测试

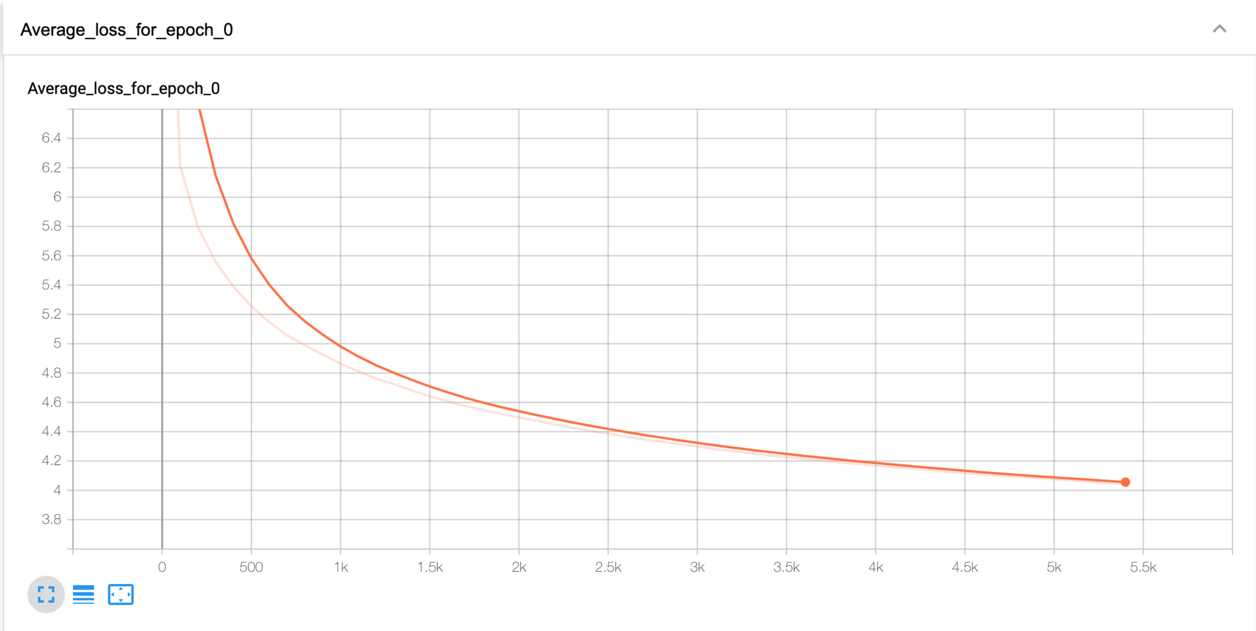
```
python train.py
```

示例输出

terminal

```
Reading dataset ../files/train.txt... 43995 pairs.
Reading dataset ../files/dev.txt... 4999 pairs.
loading model
loading data
initializing optimizer
  0%|                                  | 0/8 [00:00<?, ?it/s]
  0%|                                  | 0/687 [00:00<?, ?it/s]
Epoch 0:  12%|                          | 1/8 [40:06<4:40:47, 2406.73s/it, Loss=4.06]
validating                              | 147/5500 [01:03<38:57,  2.29it/s]
training loss:4.062740321939642 validation loss:3.6831318472440424| 5500/5500 [40:06<00:00,  2.52it/s]
Epoch 1:  25%|                          | 2/8 [1:21:46<4:03:28, 2434.79s/it, Loss=3.42]
validating                              | 0/687 [00:00<?, ?it/s]
training loss:3.4210317802429198 validation loss:3.543324131614123| 5500/5500 [40:13<00:00,  2.47it/s]
Epoch 2:  38%|                          | 3/8 [2:03:23<3:24:27, 2453.45s/it, Loss=3.15]
validating                              | 0/687 [00:00<?, ?it/s]
training loss:3.1451434763128105 validation loss:3.5226738311541386| 5500/5500 [40:10<00:00,  2.63it/s]
Epoch 3:  50%|                          | 4/8 [2:44:59<2:44:24, 2466.10s/it, Loss=2.93]
validating                              | 0/687 [00:00<?, ?it/s]
training loss:2.9329414658979935 validation loss:3.55842999731883| 5500/5500 [40:09<00:00,  2.47it/s]
Epoch 4:  62%|                          | 5/8 [3:26:35<2:03:45, 2475.13s/it, Loss=2.75]
validating                              | 0/687 [00:00<?, ?it/s]
training loss:2.7542431769587776 validation loss:3.6269769985706377| 5500/5500 [40:10<00:00,  2.60it/s]
Epoch 5:  75%|                          | 6/8 [4:08:08<1:22:40, 2480.50s/it, Loss=2.6]
validating                              | 0/687 [00:00<?, ?it/s]
training loss:2.6023435665043917 validation loss:3.7135685151204085| 5500/5500 [40:08<00:00,  2.57it/s]
Epoch 6:  88%|                          | 7/8 [4:49:42<41:24, 2484.50s/it, Loss=2.47]
validating                              | 0/687 [00:00<?, ?it/s]
training loss:2.4746298182444137 validation loss:3.8049495743635373| 5500/5500 [40:09<00:00,  2.61it/s]
Epoch 7: 100%|                          | 8/8 [5:31:12<00:00, 2486.25s/it, Loss=2.36]
validating                              | 0/687 [00:00<?, ?it/s]
training loss:2.3638342743786898 validation loss:3.904374156624843| 5500/5500 [40:05<00:00,  2.66it/s]
100%|                                   | 624/624 [01:25<00:00,  7.41it/s]
```

tensorboard



如果确认前面的模块实现无误，可以训练你的模型，模型需要长时间运行，为了防止其意外中断，可以使用以下命令，让模型在后台运行，同时又可以实时看到输出。

```
nohup python train.py 2>&1 | tee out/train.out &
```

执行后会生成一个文件train.out，训练过程的输出可以在train.out中查看。

模块4: 解码

model/predict.py:

任务1: 实现Greedy search。

这一块比较简单，跟着代码的提示，用encoder编码输入，传递每一个time step的信息给decoder，计算attention，得到decoder的p_vocab，根据p_vocab选出概率最大的词作为下一个token。

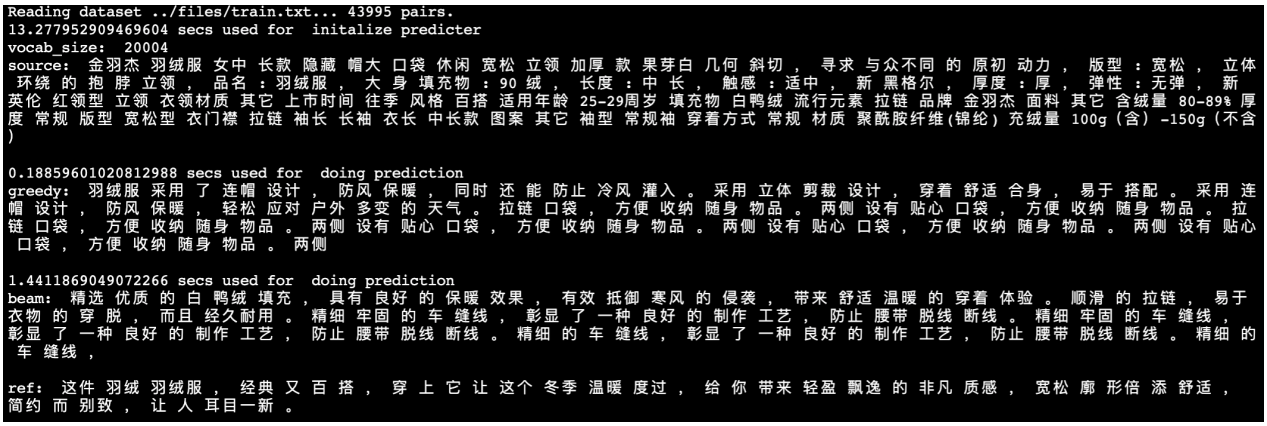
任务2: 实现Beam search。

1. 看懂Beam这一个类需要传递的变量（实现在model/utils.py中）。
2. 完成best_k函数。这里做的事情与greedy search很接近，不过要选出最好的k个token，然后扩展出k个新的beam容器。
3. 完成beam search函数。初始化encoder、attention和decoder的输入，然后对于每一个decode step，对于现有的k个beam，我们分别利用best_k函数来得到各自最佳的k个extended beam，也就是每个decode step我们会得到k*k个新的beam，然后只保留分数最高的k个，作为下一轮需要扩展的k个beam。为了只保留分数最高的k个beam，我们可以用一个堆(heap)来实现，堆的中只保存k个节点，根结点保存分数最低的beam，python实现堆的方法详见https://docs.python.org/2/library/heapq.html。
4. Hint: 用heapq模块时，各种操作（push, pop）需要比较堆中元素的大小，如果以tuple的形式来存储，会从tuple的第一个位置开始比较，如果第一个位置的值相同，会继续比较后面的位置的值。所以建议以（分数，对象id，对象）三元组的形式来存储，其中对象id的作用是快速break ties。

测试

```
python predict.py
```

示例输出




```
python rouge_eval.py
```

示例输出

```
rouge-1
f: 19.2766766683626
p: 13.915999999999986
r: 31.78550522979301
rouge-2
f: 3.3795789869785264
p: 2.4121212121212348
r: 5.734313237506023
rouge-l
f: 14.511206665736337
p: 13.57712270176535
r: 16.627146722660004
```

####