**实验十一 神经网络自动摘要**

1. 实验目的
2. 熟悉神经网络自动摘要的基本模型。
3. 掌握Tensorflow的基本用法，学会使用框架进行Attention机制、Sequence to Sequence等算法的实现。
4. 深刻理解深度学习中的经典调参技术。
5. 实验要求
6. 进行神经网络自动摘要模型的设计与分析。
7. 实现神经网络自动摘要模型，并在CNN/Daily Mails数据集上进行训练和评估。
8. 实验原理

经典神经网络自动摘要算法的基本结构是序列到序列的注意力模型即（Sequence-to-sequence attentional model），模型结构如图1所示。可参考附件文章“Get To The Point: Summarization with Pointer-Generator Networks”

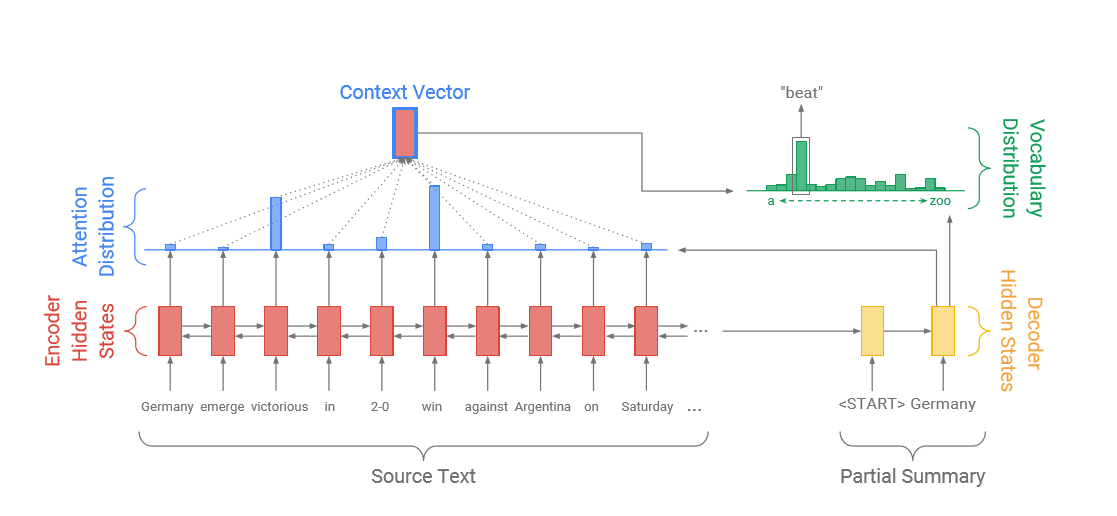


图1 经典神经网络自动摘要算法的基本结构

首先，将文章（article）中的单词一个一个的输入编码器（encoder）中，一般的encoder是一个双向LSTM或GRU，由此产生一系列的encoder状态。在每一个时间step中，decoder(一个单向的LSTM或GRU)，接收来自前一个step的单词输入，（注意：在训练过程中，这个单词是从标准的reference summary中得到，而在测试过程中，这个单词是由decoder产生的），并得到一个decoder的隐藏层状态（）。

此外，该模型还涉及到注意力机制（attention mechanism），注意力机制在本模型的计算方法如下：

、、、是可学习的参数。注意力可以看作是decoder过程中的对原句子的概率分布，表示此时应该着重关注原句子的哪一部分。注意力分布用来对原句子的隐层状态加权求和得到上下文向量（context vector）。

最后，context vector与decoder 状态进行连接，通过线性层产生词汇表分布，得到predict words。在摘要生成部分，为了简化候选摘要的筛选，使用beam search 算法选择最终摘要。

模型使用负对数似然作为loss函数，计算神经网络参数的梯度，进行网络的训练。

1. 实验所用工具及数据集
2. 实验工具

Tensorflow，Pyrouge

1. 数据集

名称：non-anonymized version of the CNN / Daily Mail summarization dataset，下载地址：https://pan.baidu.com/s/1gj4korvqZ3HJD5wyqKDBnw，提取码: gfub。

注意：可以直接下载别人处理过的数据，也可以下载未经处理的数据然后按照相关介绍自行处理。

数据集基本构成：

* 训练集：287226对数据，每对数据由一个article和一个abstract构成，用来训练模型。
* 测试集：11490对数据，用来测试模型性能。
* 验证集：13368对数据，用来进行模型的early stopping判断和超参数的调节。

1. 实验步骤与方法
2. 数据预处理

处理方法及数据集下载地址：https://github.com/abisee/cnn-dailymail

1. 模型主要模块的实现

根据实验原理部分的内容，模型实验的主要模块分为encoder，decoder，attention mechanism，beam search几个算法以及rouge score的计算（使用pyrouge进行计算即可），具体实现方法可以参考下面的内容。

1. sequence-to-sequence部分

Encoder:

1. with tf.variable\_scope('seq2seq'):
2. # Some initializers
3. self.rand\_unif\_init = tf.random\_uniform\_initializer(-hps.rand\_unif\_init\_mag, hps.rand\_unif\_init\_mag, seed=123)
4. self.trunc\_norm\_init = tf.truncated\_normal\_initializer(stddev=hps.trunc\_norm\_init\_std)
6. # Add embedding matrix (shared by the encoder and decoder inputs)
7. with tf.variable\_scope('embedding'):
8. embedding = tf.get\_variable('embedding', [vsize, hps.emb\_dim], dtype=tf.float32, initializer=self.trunc\_norm\_init)
9. **if** hps.mode=="train": self.\_add\_emb\_vis(embedding) # add to tensorboard
10. emb\_enc\_inputs = tf.nn.embedding\_lookup(embedding, self.\_enc\_batch) # tensor with shape (batch\_size, max\_enc\_steps, emb\_size)
11. emb\_dec\_inputs = [tf.nn.embedding\_lookup(embedding, x) **for** x **in** tf.unstack(self.\_dec\_batch, axis=1)] # list length max\_dec\_steps containing shape (batch\_size, emb\_size)
13. # Add the encoder.
14. enc\_outputs, fw\_st, bw\_st = self.\_add\_encoder(emb\_enc\_inputs, self.\_enc\_lens)
15. self.\_enc\_states = enc\_outputs
17. # Our encoder is bidirectional and our decoder is unidirectional so we need to reduce the final encoder hidden state to the right size to be the initial decoder hidden state
18. self.\_dec\_in\_state = self.\_reduce\_states(fw\_st, bw\_st)

Decoder:

1. with tf.variable\_scope('decoder'):
2. decoder\_outputs, self.\_dec\_out\_state, self.attn\_dists, self.p\_gens, self.coverage = self.\_add\_decoder(emb\_dec\_inputs)
4. # Add the output projection to obtain the vocabulary distribution
5. with tf.variable\_scope('output\_projection'):
6. w = tf.get\_variable('w', [hps.hidden\_dim, vsize], dtype=tf.float32, initializer=self.trunc\_norm\_init)
7. w\_t = tf.transpose(w)
8. v = tf.get\_variable('v', [vsize], dtype=tf.float32, initializer=self.trunc\_norm\_init)
9. vocab\_scores = [] # vocab\_scores is the vocabulary distribution before applying softmax. Each entry on the list corresponds to one decoder step
10. **for** i,output **in** enumerate(decoder\_outputs):
11. **if** i > 0:
12. tf.get\_variable\_scope().reuse\_variables()
13. vocab\_scores.append(tf.nn.xw\_plus\_b(output, w, v)) # apply the linear layer
15. vocab\_dists = [tf.nn.softmax(s) **for** s **in** vocab\_scores] # The vocabulary distributions. List length max\_dec\_steps of (batch\_size, vsize) arrays. The words are in the order they appear in the vocabulary file.

2)attention mechanism

1. **def** attention(decoder\_state, coverage=None):
2. """Calculate the context vector and attention distribution from the decoder state.
3. Args:
4. decoder\_state: state of the decoder
5. coverage: Optional. Previous timestep's coverage vector, shape (batch\_size, attn\_len, 1, 1).
6. Returns:
7. context\_vector: weighted sum of encoder\_states
8. attn\_dist: attention distribution
9. coverage: new coverage vector. shape (batch\_size, attn\_len, 1, 1)
10. """
11. with variable\_scope.variable\_scope("Attention"):
12. # Pass the decoder state through a linear layer (this is W\_s s\_t + b\_attn in the paper)
13. decoder\_features = linear(decoder\_state, attention\_vec\_size, True) # shape (batch\_size, attention\_vec\_size)
14. decoder\_features = tf.expand\_dims(tf.expand\_dims(decoder\_features, 1), 1) # reshape to (batch\_size, 1, 1, attention\_vec\_size)
16. **def** masked\_attention(e):
17. """Take softmax of e then apply enc\_padding\_mask and re-normalize"""
18. attn\_dist = nn\_ops.softmax(e) # take softmax. shape (batch\_size, attn\_length)
19. attn\_dist \*= enc\_padding\_mask # apply mask
20. masked\_sums = tf.reduce\_sum(attn\_dist, axis=1) # shape (batch\_size)
21. **return** attn\_dist / tf.reshape(masked\_sums, [-1, 1]) # re-normalize
23. **if** use\_coverage **and** coverage **is** **not** None: # non-first step of coverage
24. # Multiply coverage vector by w\_c to get coverage\_features.
25. coverage\_features = nn\_ops.conv2d(coverage, w\_c, [1, 1, 1, 1], "SAME") # c has shape (batch\_size, attn\_length, 1, attention\_vec\_size)
27. # Calculate v^T tanh(W\_h h\_i + W\_s s\_t + w\_c c\_i^t + b\_attn)
28. e = math\_ops.reduce\_sum(v \* math\_ops.tanh(encoder\_features + decoder\_features + coverage\_features), [2, 3])  # shape (batch\_size,attn\_length)
30. # Calculate attention distribution
31. attn\_dist = masked\_attention(e)
33. # Update coverage vector
34. coverage += array\_ops.reshape(attn\_dist, [batch\_size, -1, 1, 1])
35. **else**:
36. # Calculate v^T tanh(W\_h h\_i + W\_s s\_t + b\_attn)
37. e = math\_ops.reduce\_sum(v \* math\_ops.tanh(encoder\_features + decoder\_features), [2, 3]) # calculate e
39. # Calculate attention distribution
40. attn\_dist = masked\_attention(e)
42. **if** use\_coverage: # first step of training
43. coverage = tf.expand\_dims(tf.expand\_dims(attn\_dist,2),2) # initialize coverage
45. # Calculate the context vector from attn\_dist and encoder\_states
46. context\_vector = math\_ops.reduce\_sum(array\_ops.reshape(attn\_dist, [batch\_size, -1, 1, 1]) \* encoder\_states, [1, 2]) # shape (batch\_size, attn\_size).
47. context\_vector = array\_ops.reshape(context\_vector, [-1, attn\_size])
49. **return** context\_vector, attn\_dist, coverage

3)损失计算

1. **if** hps.mode **in** ['train', 'eval']:
2. # Calculate the loss
3. with tf.variable\_scope('loss'):
4. **if** FLAGS.pointer\_gen:
5. # Calculate the loss per step
6. # This is fiddly; we use tf.gather\_nd to pick out the probabilities of the gold target words
7. loss\_per\_step = [] # will be list length max\_dec\_steps containing shape (batch\_size)
8. batch\_nums = tf.range(0, limit=hps.batch\_size) # shape (batch\_size)
9. **for** dec\_step, dist **in** enumerate(final\_dists):
10. targets = self.\_target\_batch[:,dec\_step] # The indices of the target words. shape (batch\_size)
11. indices = tf.stack( (batch\_nums, targets), axis=1) # shape (batch\_size, 2)
12. gold\_probs = tf.gather\_nd(dist, indices) # shape (batch\_size). prob of correct words on this step
13. losses = -tf.log(gold\_probs)
14. loss\_per\_step.append(losses)
16. # Apply dec\_padding\_mask and get loss
17. self.\_loss = \_mask\_and\_avg(loss\_per\_step, self.\_dec\_padding\_mask)
19. **else**: # baseline model
20. self.\_loss = tf.contrib.seq2seq.sequence\_loss(tf.stack(vocab\_scores, axis=1), self.\_target\_batch, self.\_dec\_padding\_mask) # this applies softmax internally
22. tf.summary.scalar('loss', self.\_loss)
24. # Calculate coverage loss from the attention distributions
25. **if** hps.coverage:
26. with tf.variable\_scope('coverage\_loss'):
27. self.\_coverage\_loss = \_coverage\_loss(self.attn\_dists, self.\_dec\_padding\_mask)
28. tf.summary.scalar('coverage\_loss', self.\_coverage\_loss)
29. self.\_total\_loss = self.\_loss + hps.cov\_loss\_wt \* self.\_coverage\_loss
30. tf.summary.scalar('total\_loss', self.\_total\_loss)

4）beam search部分

1. **def** run\_beam\_search(sess, model, vocab, batch):
2. """Performs beam search decoding on the given example.
3. Args:
4. sess: a tf.Session
5. model: a seq2seq model
6. vocab: Vocabulary object
7. batch: Batch object that is the same example repeated across the batch
8. Returns:
9. best\_hyp: Hypothesis object; the best hypothesis found by beam search.
10. """
11. # Run the encoder to get the encoder hidden states and decoder initial state
12. enc\_states, dec\_in\_state = model.run\_encoder(sess, batch)
13. # dec\_in\_state is a LSTMStateTuple
14. # enc\_states has shape [batch\_size, <=max\_enc\_steps, 2\*hidden\_dim].
16. # Initialize beam\_size-many hyptheses
17. hyps = [Hypothesis(tokens=[vocab.word2id(data.START\_DECODING)],
18. log\_probs=[0.0],
19. state=dec\_in\_state,
20. attn\_dists=[],
21. p\_gens=[],
22. coverage=np.zeros([batch.enc\_batch.shape[1]]) # zero vector of length attention\_length
23. ) **for** \_ **in** range(FLAGS.beam\_size)]
24. results = [] # this will contain finished hypotheses (those that have emitted the [STOP] token)
26. steps = 0
27. **while** steps < FLAGS.max\_dec\_steps **and** len(results) < FLAGS.beam\_size:
28. latest\_tokens = [h.latest\_token **for** h **in** hyps] # latest token produced by each hypothesis
29. latest\_tokens = [t **if** t **in** range(vocab.size()) **else** vocab.word2id(data.UNKNOWN\_TOKEN) **for** t **in** latest\_tokens] # change any in-article temporary OOV ids to [UNK] id, so that we can lookup word embeddings
30. states = [h.state **for** h **in** hyps] # list of current decoder states of the hypotheses
31. prev\_coverage = [h.coverage **for** h **in** hyps] # list of coverage vectors (or None)
33. # Run one step of the decoder to get the new info
34. (topk\_ids, topk\_log\_probs, new\_states, attn\_dists, p\_gens, new\_coverage) = model.decode\_onestep(sess=sess,
35. batch=batch,
36. latest\_tokens=latest\_tokens,
37. enc\_states=enc\_states,
38. dec\_init\_states=states,
39. prev\_coverage=prev\_coverage)
41. # Extend each hypothesis and collect them all in all\_hyps
42. all\_hyps = []
43. num\_orig\_hyps = 1 **if** steps == 0 **else** len(hyps) # On the first step, we only had one original hypothesis (the initial hypothesis). On subsequent steps, all original hypotheses are distinct.
44. **for** i **in** range(num\_orig\_hyps):
45. h, new\_state, attn\_dist, p\_gen, new\_coverage\_i = hyps[i], new\_states[i], attn\_dists[i], p\_gens[i], new\_coverage[i]  # take the ith hypothesis and new decoder state info
46. **for** j **in** range(FLAGS.beam\_size \* 2):  # for each of the top 2\*beam\_size hyps:
47. # Extend the ith hypothesis with the jth option
48. new\_hyp = h.extend(token=topk\_ids[i, j],
49. log\_prob=topk\_log\_probs[i, j],
50. state=new\_state,
51. attn\_dist=attn\_dist,
52. p\_gen=p\_gen,
53. coverage=new\_coverage\_i)
54. all\_hyps.append(new\_hyp)
56. # Filter and collect any hypotheses that have produced the end token.
57. hyps = [] # will contain hypotheses for the next step
58. **for** h **in** sort\_hyps(all\_hyps): # in order of most likely h
59. **if** h.latest\_token == vocab.word2id(data.STOP\_DECODING): # if stop token is reached...
60. # If this hypothesis is sufficiently long, put in results. Otherwise discard.
61. **if** steps >= FLAGS.min\_dec\_steps:
62. results.append(h)
63. **else**: # hasn't reached stop token, so continue to extend this hypothesis
64. hyps.append(h)
65. **if** len(hyps) == FLAGS.beam\_size **or** len(results) == FLAGS.beam\_size:
66. # Once we've collected beam\_size-many hypotheses for the next step, or beam\_size-many complete hypotheses, stop.
67. **break**
69. steps += 1
71. # At this point, either we've got beam\_size results, or we've reached maximum decoder steps
73. **if** len(results)==0: # if we don't have any complete results, add all current hypotheses (incomplete summaries) to results
74. results = hyps
76. # Sort hypotheses by average log probability
77. hyps\_sorted = sort\_hyps(results)
79. # Return the hypothesis with highest average log prob
80. **return** hyps\_sorted[0]

（5）模型评估，计算rouge score

1. **def** rouge\_eval(ref\_dir, dec\_dir):
2. """Evaluate the files in ref\_dir and dec\_dir with pyrouge, returning results\_dict"""
3. r = pyrouge.Rouge155()
4. r.model\_filename\_pattern = '#ID#\_reference.txt'
5. r.system\_filename\_pattern = '(\d+)\_decoded.txt'
6. r.model\_dir = ref\_dir
7. r.system\_dir = dec\_dir
8. logging.getLogger('global').setLevel(logging.WARNING) # silence pyrouge logging
9. rouge\_results = r.convert\_and\_evaluate()
10. **return** r.output\_to\_dict(rouge\_results)

13. **def** rouge\_log(results\_dict, dir\_to\_write):
14. """Log ROUGE results to screen and write to file.
15. Args:
16. results\_dict: the dictionary returned by pyrouge
17. dir\_to\_write: the directory where we will write the results to"""
18. log\_str = ""
19. **for** x **in** ["1","2","l"]:
20. log\_str += "\nROUGE-%s:\n" % x
21. **for** y **in** ["f\_score", "recall", "precision"]:
22. key = "rouge\_%s\_%s" % (x,y)
23. key\_cb = key + "\_cb"
24. key\_ce = key + "\_ce"
25. val = results\_dict[key]
26. val\_cb = results\_dict[key\_cb]
27. val\_ce = results\_dict[key\_ce]
28. log\_str += "%s: %.4f with confidence interval (%.4f, %.4f)\n" % (key, val, val\_cb, val\_ce)
29. tf.logging.info(log\_str) # log to screen
30. results\_file = os.path.join(dir\_to\_write, "ROUGE\_results.txt")
31. tf.logging.info("Writing final ROUGE results to %s...", results\_file)
32. with open(results\_file, "w") as f:
33. f.write(log\_str)