Lab 7- A coreference system using the Tensorflow low-level APIs

February 26

Starting with this lab, we will start to use the Tensorflow low-level APIs. The low-level APIs give you more flexibility than high-level APIs such as Keras, and allow you to understand how the system works in more detail. But as a consequence, the code will be generally longer than Keras.

In this lab, we are going to build a coreference system based on the mention-ranking algorithm proposed by Lee et al (2017). You will get part of the code required to build the system, and you are required to fill three code blocks. Hints will be provided to guide you through.

In total, you will be given two python files (*.py), three JSON files (*.jsonlines) and one embedding file (*.txt).

- The script **metric.py** is used to compute the CoNLL scores; you don't need to change it.
- **coref_model.py** is the main script you are going to work on.
- The **train/test/dev.english.jsonlines** documents are the training, testing and development set will be used for training and evaluating the model, which are ready to use.
- **glove.840B.300d.txt** is pre-trained 300-dimensional Glove word embeddings. The original file is very large, so we've removed all the words that do not appear in the datasets to make it much smaller.

1. Coref_model.py, step by step

The script defines a single class (`CorefModel`) which contains all the elements needed for a simple coreference system.

1.1 The __init__ () method : initialize the network parameters.

Here we hardcoded them. For a real system, usually, the parameters will be stored in a configuration file, as there will be many of them.

In our case, we have in total 7 parameters:

The path to the pre-trained word embeddings:
```
self.embedding_path = embedding_path
```

The dimension of the pretrained embeddings, which in our case is 300:

```
self.embedding_size = embedding_size
```

A dropout rate for word embeddings (this is usually larger than the hidden dropout rate for the neural networks).

```
self.embedding_dropout_rate = 0.5
```

The maximum number of candidate antecedents we will give to each of the candidate mentions.

```
self.max_ant = 250
```

The size of the hidden layer, include both LSTM and feedforward NN

```
self.hidden_size = 150
```

The number of hidden layers used for the feedforward NN

```
self.ffnn_layer = 2
```

The dropout rate for the hidden layers of LSTM and feedforward NN

```
self.hidden_dropout_rate = 0.2
```

1.2 The build() method builds a tensorflow graph for our task.

This method first loads the pre-trained word embeddings from the given location by calling the `load_embeddings` method:

```
self.embedding_dict = self.load_embeddings(self.embedding_path, self.embedding_size)

def load_embeddings(self, path, size):
 print("Loading word embeddings from {}...".format(path))
 embeddings = collections.defaultdict(lambda: np.zeros(size))
 for line in open(path):
   splitter = line.find(' ')
   emb = np.fromstring(line[splitter + 1:], np.float32, sep=' ')
   assert len(emb) == size
   embeddings[line[:splitter]] = emb
 print("Finished loading word embeddings")
 return embeddings
```

It then creates placeholders, which are the inputs of the tensorflow graph:

```
self.word_embeddings = tf.placeholder(tf.float32, shape=[None, None,self.embedding_size])
self.sent_lengths = tf.placeholder(tf.int32, shape=[None])
self.mention_starts = tf.placeholder(tf.int32, shape=[None])
self.mention_ends = tf.placeholder(tf.int32, shape=[None])
self.mention_cluster_ids = tf.placeholder(tf.int32, shape=[None])
self.is_training = tf.placeholder(tf.bool, shape=[])
```

After that, the method calls the `get_predictions_and_loss` method to create the rest of the tensorflow graph. The `get_predictions_and_loss` method is the one you will mainly working on; we will come back to this method in the later sections for more details.

```
self.predictions, self.loss = self.get_predictions_and_loss(
 self.word_embeddings, self.sent_lengths, self.mention_starts, self.mention_ends,
 self.mention_cluster_ids, self.is_training)
```

Finally, the method defines the training mechanism and initializes global variables of our model.

```
trainable_params = tf.trainable_variables()
gradients = tf.gradients(self.loss, trainable_params)
gradients, _ = tf.clip_by_global_norm(gradients,5.0)
optimizer = tf.train.AdamOptimizer()
self.train_op = optimizer.apply_gradients(zip(gradients,trainable_params))
self.sess = tf.Session()
self.sess.run(tf.global_variables_initializer())
```

1.3 The shape() method helps us get the n-th dimension of a given tensor.

In tensorflow there are two ways of getting a tensor's shape. The `Tensor.get_shape()[n]` method can get a predefined dimension of the tensor, such as the last dimension of the word_embeddings (300) or the dimension of the hidden layer (for both LSTM and FFNN is 150). The second method (`tf.shpe()[n]`) returns the dynamic size of the tensor, such as the number of sentences or the number of mentions. Those sizes are not fixed across different documents but are dynamic.

1.4 The get_feed_dict_list() method creates inputs for the tensorflow graph.

This method reads documents from the the json files and return a list of `feed_dict` elements, which are used as the input for the tensorflow graph. The method also returns the gold clusters of each documents which are used for evaluation.

Each of the lines in the json files contains information for a single document. The "`doc_key`" stores the name of the document; the "`sentences`" points you to tokenized sentences of the document; the "`clusters`" element stores the coreference clusters. Each of the clusters contains a number of mentions, each of the mentions has a start and an end indices which link back to the sentences.

For each document, the method first assigns each mention a `cluster_id` according to the clusters it belongs to:

```
clusters = doc['clusters']
gold_mentions = sorted([tuple(m) for cl in clusters for m in cl])
gold_mention_map = {m:i for i,m in enumerate(gold_mentions)}
cluster_ids = np.zeros(len(gold_mentions))
for cid, cluster in enumerate(clusters):
 for mention in cluster:
   cluster_ids[gold_mention_map[tuple(mention)]] = cid + 1
```

It then splits the mentions into two arrays, one representing the start indices, and the other for the end indices:

```
starts, ends = [], []
if len(gold_mentions) > 0:
 starts, ends = zip(*gold_mentions)
starts, ends = np.array(starts), np.array(ends)
```

After that, it reads the word embeddings for the sentences in the document:

```
sentences = doc['sentences']
sent_lengths = [len(sent) for sent in sentences]
max_sent_length = max(sent_lengths)
word_emb = np.zeros([len(sentences),max_sent_length,self.embedding_size])
for i, sent in enumerate(sentences):
 for j, word in enumerate(sent):
   word_emb[i,j] = self.embedding_dict[word]
```

In the end, it creates the feed_dict:

```
fd = {}
fd[self.word_embeddings] = word_emb
fd[self.sent_lengths] = np.array(sent_lengths)
fd[self.mention_starts] = starts
fd[self.mention_ends] = ends
fd[self.mention_cluster_ids] = cluster_ids
fd[self.is_training] = is_training
```

1.5 The get_predicted_clusters() method is the third code block you are asked to fill.

The requirements will be discussed later.

1.6 The evaluate_coref() method updates the coreference scorer.

The method first creates `gold_clusters` and `mention_to_gold` which are required by the coreference scorer. It is important that both clusters and mentions should be tuples in order to be used by the scorer. `Mention_to_gold` is the map from mention to clusters.

```
gold_clusters = [tuple(tuple(m) for m in gc) for gc in gold_clusters]
mention_to_gold = {}
for gc in gold_clusters:
 for mention in gc:
   mention_to_gold[mention] = gc
```

It then creates predicted clusters by calling the `get_predicted_clusters` method.

```
predicted_clusters, mention_to_predicted = self.get_predicted_clusters(mention_starts,
mention_ends, predicted_antecedents)
```

After obtaining both gold and predicted clusters, the method updates the scorer.

```
evaluator.update(predicted_clusters, gold_clusters, mention_to_predicted, mention_to_gold)
```

1.7 The train() method oversees the training process.

It first loads the training data:

```
train_fd_list = self.get_feed_dict_list(train_path, True)
```

Then training the model by go through the all the training documents a number of times. It also outputs the average loss and time usage of the training.

```
for epoch in xrange(epochs):
 print("Starting training epoch {}/{}".format(epoch+1,epochs))
 epoch_time = time.time()
 losses = []
 for i, (fd, _) in enumerate(train_fd_list):
   _,loss = self.sess.run([self.train_op,self.loss], feed_dict=fd)
   losses.append(loss)
   if i>0 and i%200 == 0:
     print("[{}]: loss:{:.2f}".format(i,sum(losses[i-200:])/200.0))
 print("Average epoch loss:{}".format(sum(losses)/len(losses)))
 print("Time used for epoch {}: {}".format(epoch+1, self.time_used(epoch_time)))
```

After each epoch, the model evaluates on the development set. Normally, the model will be written to the disk if a better dev score is obtained. Here we didn’t do that to simplify the code for lab use.

```
dev_time = time.time()
print("Evaluating on dev set after epoch {}/{}:".format(epoch+1,epochs))
self.eval(dev_path)
print("Time used for evaluate on dev set: {}".format(self.time_used(dev_time)))
```

After finished all the training epochs, it evaluates on the final test set.

```
print("Evaluating on test set:")
test_time = time.time()
self.eval(test_path)
print("Time used for evaluate on test set: {}".format(self.time_used(test_time)))
```

1.8 The eval() method runs a test on the given dataset.

The method first reads the dataset:

```
eval_fd_list = self.get_feed_dict_list(path, False)
```

Then, it creates an instance of the coreference scorer:

```
coref_evaluator = metrics.CorefEvaluator()
```

After that, it evaluates the dataset document by document:

```
for fd, clusters in eval_fd_list:
 mention_starts,mention_ends = fd[self.mention_starts],fd[self.mention_ends]
 antecedents, mention_pair_scores = self.sess.run(self.predictions, fd)
 predicted_antecedents = []
 for i, index in enumerate(np.argmax(mention_pair_scores, axis=1) - 1):
   if index < 0:
     predicted_antecedents.append(-1)
   else:
     predicted_antecedents.append(antecedents[i, index])
self.evaluate_coref(mention_starts,mention_ends,predicted_antecedents,clusters,coref_eval
uator)
```

In the end, it gets the scores from the coreference scorer.

```
p, r, f = coref_evaluator.get_prf()
print("Average F1 (py): {:.2f}%".format(f * 100))
print("Average precision (py): {:.2f}%".format(p * 100))
print("Average recall (py): {:.2f}%".format(r * 100))
```

1.9 The time_used() method outputs the time differences between the current time and the input time.

It is always a good practice to record the time usage of individual process, so you always known which part is most expensive to run.

```
curr_time = time.time()
used_time = curr_time-start_time
m = used_time // 60
s = used_time - 60 * m
return "%d m %d s" % (m, s)
```

1.10 The __main__ method starts the training.

It also configures the model by providing the locations of all the files needed for the model.

```
embedding_path = 'glove.840B.300d.txt.filtered'
train_path = 'train.english.30sent.jsonlines'
dev_path = 'dev.english.jsonlines'
test_path = 'test.english.jsonlines'
embedding_size = 300
model = CorefModel(embedding_path,embedding_size)
model.build()
model.train(train_path,dev_path,test_path,5)
```

2 Task 1: Create a bidirectional LSTM

For task 1 we will be working at the beginning of the `get_predictions_and_loss()` method. The task is to create a bidirectional LSTM to encode the sentences from both directions, which provides context information to the coreference system.

The variables you will be using are:
The keep probability (1-dropout_rate) of the word embeddings: `embedding_keep_prob`
The keep probability of the hidden layers: `hidden_keep_prob`
The word embeddings of the document: `word_embeddings`
The length of individual sentences: `sent_lengths`
The size of the hidden layers: `self.hidden_size`

Firstly, you need to apply a dropout to the word_embeddings using the **tf.nn.dropout()** method.

Secondly, you need to create two LSTMs, one for forward LSTM and one for backward LSTM. The basic LSTM available from tensorflow is the **tf.nn.rnn_cell.LSTMCell()**, you can also apply dropouts to the LSTMs by using the **tf.nn.rnn_cell.DropoutWrapper()**.

Thirdly, you need to feed the LSTMs the word embeddings and retrieve the outputs. The easiest way to do this is to use the build-in **tf.nn.bidirectional_dynamic_rnn()** method. The method does all the reversing for you and returns the pairwise outputs of for/backward LSTMs. Please name the outputs of the for/backward LSTMs `output_for` and `output_rev`.

The outputs of your LSTMs will be concatenated for further use:

```
word_output = tf.concat([output_for, output_rev], axis=-1)
```

`word_output` at the moment is grouped by sentences, but we have the mentions with word indices of the documents. In order to match the indices between `word_output` and mentions, we need to flatten `word_output` and remove the padding words.

```
num_sents = self.shape(word_embeddings, 0)
max_sent_length = self.shape(word_embeddings, 1)
word_seq_mask = tf.sequence_mask(sent_lengths, max_sent_length)
flatten_word_seq_mask = tf.reshape(word_seq_mask, [num_sents * max_sent_length])
flatten_word_output = tf.reshape(word_output, [num_sents * max_sent_length, 2 * self.hidden_size])
flatten_word_output = tf.nn.dropout(flatten_word_output, hidden_keep_prob)
flatten_word_output = tf.boolean_mask(flatten_word_output, flatten_word_seq_mask, axis=0)
```

Then, the mention embeddings can be created by concatenating `word_output` at the positions of mention's start and end indices:

```
mention_starts_emb = tf.gather(flatten_word_output,mention_starts)
mention_ends_emb = tf.gather(flatten_word_output,mention_ends)
mention_emb = tf.concat([mention_starts_emb,mention_ends_emb],axis=1)
```

In order to do coreference, we also need to create the candidate antecedents. Here we give each of the candidate mentions a maximum 250 candidate antecedents (candidate mentions that before the current mention).

```
num_mention = self.shape(mention_emb, 0)
max_ant = tf.minimum(num_mention,self.max_ant)
antecedents = tf.expand_dims(tf.range(num_mention),1) \
          - tf.tile(tf.expand_dims(tf.range(max_ant)+1, 0), [num_mention, 1])
antecedents_mask = antecedents >= 0
antecedents = tf.maximum(antecedents, 0)
antecedents_emb = tf.gather(mention_emb, antecedents)
```

After that, we concatenate the mentions embeddings with the antecedent embeddings to create the mention pair embeddings.

```
tiled_mention_emb = tf.tile(tf.expand_dims(mention_emb, 1), [1,max_ant,1])
mention_pair_emb = tf.concat([tiled_mention_emb, antecedents_emb], 2)
```

The pairwise embeddings will be used by task 2 to compute mention pair scores.

3 Task 2: Create a multilayer feed-forward neural network to compute the mention-pair scores

This part of the script will develop code to compute the s_a(i,j) score for the mention pairs used as part of the computation of the overall s(i,j) score (Note: you don't need to compute s(i) and s(j) because we're using gold.)

The task starts with reshaping the pairwise embeddings to make the embeddings as a metric:

```
ffnn_input = tf.reshape(mention_pair_emb,[num_mention*max_ant, 8 * self.hidden_size])
```

Then you are required to create a FFNN that contains 2 hidden layers and an output layer. The outputs of the FFNN are mention_pair_scores. Here are some requirements:

1. The hidden layers need to have a size of self.hidden_size
2. You need to apply dropout to all the hidden layers
3. The outputs are mention_pair_scores

Tips:
Each layer of the FFNN is a simple matrix operation: xw + b where x is the input, w is the weights and b is the bias. Let d_x be the shape of the last dimension of the input matrix and d_o is the last dimension of the output of the layer. Then w should be a d_x * d_o matrix and b is a d_o dimensional vector. You could use the **tf.get_variable()** method to create weights and bias as needed. The matrix operation can be done by using **tf.nn.xw_plus_b()**.

After getting the mention_pair_scores from you, we need to further process them by setting the padding antecedents to -inf and adding a dummy_scores to handle cases in which a mention does not coreference with any of the candidate antecedents (discourse new, not a mention etc).

```
mention_pair_scores = tf.reshape(mention_pair_scores,[num_mention,max_ant])
mention_pair_scores += tf.log(tf.to_float(antecedents_mask))
dummy_scores = tf.zeros([num_mention,1])
```

```
mention_pair_scores = tf.concat([dummy_scores,mention_pair_scores], 1)
```

Then, we calculate the train labels for the mentions pairs (the train labels tell you a mention pair is corefence or not):

```
antecedents_cluster_ids = tf.gather(mention_cluster_ids,antecedents) +
tf.to_int32(tf.log(tf.to_float(antecedents_mask)))
mention_pair_labels = tf.logical_and(
 tf.equal(antecedents_cluster_ids, tf.expand_dims(mention_cluster_ids, 1)),
 tf.greater(antecedents_cluster_ids, 0))
dummy_labels = tf.logical_not(tf.reduce_any(mention_pair_labels,1,keepdims=True))
mention_pair_labels = tf.concat([dummy_labels,mention_pair_labels],1)
```

Finally, we compute the loss. Here we use the loss function of the state-of-the-art corefence system (Lee et al 2018):

```
gold_scores = mention_pair_scores + tf.log(tf.to_float(mention_pair_labels))
marginalized_gold_scores = tf.reduce_logsumexp(gold_scores,1)
log_norm = tf.reduce_logsumexp(mention_pair_scores,1)
loss = log_norm - marginalized_gold_scores
loss = tf.reduce_sum(loss)
```

4 Task 3: Form the predicted clusters.

The last task is to form the predicted clusters. You will need to finish the `get_predicted_clusters()` method.

The inputs of the method are mention_starts and mention_ends, and the indices of predicted antecedents. All three inputs have the same size: the i_th mention and it's antecedents are (mention_starts[i], mention_ends[i]), (mention_starts[predicted_antecedents[i]], mention_ends[predicted_antecedents[i]]) respectively. Please note the predicted_antecedents can be -1 if a mention was predicted as discourse new or not a mention.

The required outputs are predicted_clusters and mention_to_predicted. The predicted_clusters are coreference clusters that have at least two mentions. In CoNLL coreference, singleton mentions and non-referring expressions are not annotated. There are different ways to form the clusters, for example, you could create a new cluster whenever predicted_antecedents[n] = -1 and then remove all the clusters that have a size of one afterwards. Since the mention_to_predicted can be easily created from the predicted_clusters, it is advisable to create the predicted_clusters first and follow the code in evaluate_coref() method to create the mention_to_predicted from predicted_clusters.

5 Run your code.

If you finished all three tasks, congratulations! you’ve made a coreference system. The next step is to train you system on the data provided. Please make sure all the files are located in the same folder. You should first load your tensorflow environment, then type:

```
python coref_model.py
```

o start your training. If all your code is correct your training should finish in about 20 minutes and have CoNLL F1 scores above 65% on both dev and test sets.