Lab 7- A coreference system using the Keras Functional APIs

March 3

Starting with this lab, we will start to use the Keras functional APIs. The functional APIs give you more flexibility than sequential models, and allow you to create more complex models. But as a consequence, the code will be generally longer than sequential models.

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_keras.py** is the main script you are going to work on.
- The **train/test/dev.english.20sent.jsonlines** documents are the training, testing and development set will be used for training and evaluating the model, which are ready to use.
- **glove.6B.100d.txt** is pre-trained 100-dimensional Glove word embeddings. The original file is large, so we've removed all the words that do not appear in the datasets to make it much smaller.

1. Coref_model_keras.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 and load pre-trained word embeddings.

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 8 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 100:

```
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 = 50
```

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
```

The ratio of positive and negative examples for training

```
self.neg_ratio = 2
```

After set the network parameters, we load the pre-trained word embeddings from the given location by calling the `load_embeddings` method:

```
self.embedding_dict = self.load_embeddings()
```

```
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
```

The pre-trained word embeddings will be later used as the input for our coreference system.

1.2 The build() method builds keras model for our task.

It first creates the input of the model:

```
word_embeddings = Input(shape=(None,None,self.embedding_size,))
```

```
mention_pairs = Input(shape=(None,4,),dtype='int32')
```

The `word_embeddings` contains word embeddings of the tokens in the documents and are stored as a list of sentences padded to the same length. Please note that for keras Input apart of the shape you specified it will always have one more dimension for the batch, so the actual shape for word_embeddings is [batch_size, num_of_sents, max_sent_length, embedding_size], in our case we use a batch size of 1 document.

The mention_pairs input contains a list of anaphora (ana) and antecedent (ant) indices, in the format [ana_start_index, ana_end_index, ant_start_index, ant_end_index]. You will need to use them to find the corresponding LSTM outputs and then feed them into a FFNN to compute `mention_pair_scores`.

After that, it is your first task to create a 2 layer bidirectional LSTMs. The LSTMs takes the `word_embeddings` as the input and output `word_output` for individual tokens in the document. We will come back in the later section to give you more details.

After we get the word_output and before we can use them to represent our mention pairs, we need to convert the sentence level `word_output` into document level. This is because our coreference task is a document level task, hence the document level indices are used in mention_pairs. Here we need to use the `reshape()` method in the `backend (K)` package to reshape the word_output since the `Reshape()` method in keras.layers will always keep the batch dimension unchanged (here we don't want to keep the batch dimension). Also for keras functional APIs the functions need to be always warped layers and they provide the `Lambda` layer to allow us warp our functions easily. `Lambda` layer is mainly used for simple operations like ours, if you want to design more complex operations you could always write a subclass that inherits the Layer class.

```
flatten_word_output=Lambda(lambda x:K.reshape(x,[-1, 2 * self.hidden_size]))(word_output)
flatten_word_output = Dropout(self.hidden_dropout_rate)(flatten_word_output)
```

Then we use the `Lambda` layer again to get the representations for mention pairs (mention_pair_emb) via the `K.gather()` function, and concatenate the word representations of all four indices using the keras `Reshape()` method, here we do want keep the batch dimension (batch size = 1) in order to compare with the gold label.

```
mention_pair_emb = Lambda(lambda x: K.gather(x[0], x[1]))([flatten_word_output, mention_pairs])
ffnn_input = Reshape((-1,8*self.hidden_size))(mention_pair_emb)
```

The next step is to put the ffnn_input into a 2 layer feed-forward neural network (FFNN) and compute the mention pair scores. This will be your task 2, we will give more details in the later section.

Finally, we create the keras `Model()` by specifying the inputs, outputs, loss metrics and optimizer. You can use the `summary()` method to output an overview of our model.

```
self.model = Model(inputs=[word_embeddings,mention_pairs],outputs=mention_pair_scores)
self.model.compile(optimizer='adam',loss='binary_crossentropy',metrics=['accuracy'])
self.model.summary()
```

1.3 The get_feed_dict_list() method creates inputs for the keras model.

This method reads documents from the the json files and returns a list of numpy arrays, which are used as the input for the keras model (`word_emb, mention_pairs, mention_pair_labels`). The method also returns the other document information (`gold_clusters, raw_mention_pairs`) 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']
if len(clusters) == 0:
 continue
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
```

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

```
raw_starts, raw_ends = zip(*gold_mentions)
```

After that, it reads the word embeddings for the sentences in the document and maps the original mention indices into the padded sentences:

```
starts, ends = [],[]
sentences = doc['sentences']
sent_lengths = [len(sent) for sent in sentences]
max_sent_length = max(sent_lengths)
word_emb = np.zeros([1,len(sentences), max_sent_length, self.embedding_size])
raw_pre,padded_pre = 0,0
for i, sent in enumerate(sentences):
```

```
#to associate the gold mention indices with padded sentences
 for s, e in gold_mentions:
   if raw_pre <=s <=e < raw_pre+len(sent):
     starts.append(s-raw_pre+padded_pre)
     ends.append(e-raw_pre+padded_pre)
 raw_pre+= len(sent)
 padded_pre+=max_sent_length
 for j, word in enumerate(sent):
   word_emb[0, i, j] = self.embedding_dict[word.lower()]
```

Next it creates the `mention_pairs` and the gold `mention_pair_labels` for training. For every positive example, we create n (`n = self.neg_ratio`) negative examples. The gold `clusters` and `raw_mention_pairs` are created for the evaluation. Remember in the last step the mention indices were mapped to the padded sentences, but for the evaluation we will need the original indices to compare with the gold clusters.

```
mention_pairs = [[]]
mention_pair_labels = [[]]
raw_mention_pairs = []
if is_training:
 for ana in range(num_mention):
   pos = 1
   s = 0 if ana < self.max_ant else (ana - self.max_ant)
   for ant in range(s,ana):
     l = cluster_ids[ana] == cluster_ids[ant]
     if l:
       pos+=self.neg_ratio
       mention_pairs[0].append([starts[ana],ends[ana],starts[ant],ends[ant]])
       mention_pair_labels[0].append(1)
     elif pos > 0:
       pos -=1
       mention_pairs[0].append([starts[ana],ends[ana],starts[ant],ends[ant]])
       mention_pair_labels[0].append(0)
else:
 for ana in range(num_mention):
   s = 0 if ana < self.max_ant else (ana - self.max_ant)
   for ant in range(s,ana):
     mention_pairs[0].append([starts[ana], ends[ana], starts[ant], ends[ant]])
     raw_mention_pairs.append([(raw_starts[ana],raw_ends[ana]),(raw_starts[ant],
raw_ends[ant])])
mention_pairs,mention_pair_labels = np.array(mention_pairs),np.array(mention_pair_labels)
```

In the end, it stores everything in a list (the `feed_dict_list`):

```
feed_dict_list.append((
  word_emb,
```

```
  mention_pairs,
  mention_pair_labels,
  clusters,
  raw_mention_pairs
 ))
```

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

The requirements will be discussed later.

1.5 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_pairs)
```

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

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

1.6 The batch_generator() method returns a single batch for training.

The method takes the whole training set and shuffle the data. Then each time it generates one training batch in the format required by keras model's `fit_generator` methods.

```
def batch_generator(self, fd_list):
 random.shuffle(fd_list)
 for word_embeddings, mention_pairs, mention_pair_labels, _, _ in fd_list:
  yield [word_embeddings, mention_pairs], mention_pair_labels
```

1.7 The train() method oversees the training process.

It first loads the training/development/testing data:

```
train_fd_list = self.get_feed_dict_list(train_path, is_training=True)
print("Load {} training documents from {}".format(len(train_fd_list), train_path))

dev_fd_list = self.get_feed_dict_list(dev_path)
print("Load {} dev documents from {}".format(len(dev_fd_list), dev_path))

test_fd_list = self.get_feed_dict_list(test_path)
print("Load {} test documents from {}".format(len(test_fd_list), test_path))
```

Then training the model by going through all the training documents a number of times. It also outputs the time usage of the training.

```
start_time = time.time()
for epoch in range(epochs):
 print("\nStarting training epoch {}/{}".format(epoch + 1, epochs))
 epoch_time = time.time()
 self.model.fit_generator(self.batch_generator(train_fd_list), steps_per_epoch=2775)
 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_fd_list)
print("Time used for evaluate on dev set: {}".format(self.time_used(dev_time)))
```

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

```
print("\nEvaluating on test set:")
test_time = time.time()
self.eval(test_fd_list)
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 creates an instance of the coreference scorer:

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

After that, it evaluates the dataset document by document:

```
for word_embeddings, mention_pairs, _, gold_clusters, raw_mention_pairs in eval_fd_list:
 mention_pair_scores = self.model.predict_on_batch([word_embeddings, mention_pairs])
 predicted_antecedents = {}
 best_antecedent_scores = {}
 for (ana, ant), score in zip(raw_mention_pairs, mention_pair_scores[0]):
   if score >= 0.5 and score > best_antecedent_scores.get(ana,0):
     predicted_antecedents[ana] = ant
     best_antecedent_scores[ana] = score

 predicted_mention_pairs = [[k,v] for k,v in predicted_antecedents.items()]

 self.evaluate_coref(predicted_mention_pairs, gold_clusters, coref_evaluator)
```

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 good practice to record the time usage of an individual process, so you always know 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.6B.100d.txt.filtered'
train_path = 'train.english.20sent.jsonlines'
dev_path = 'dev.english.20sent.jsonlines'
test_path = 'test.english.20sent.jsonlines'
```

```
embedding_size = 100
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 `build()` 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 dropout rate of the word embeddings: `self.embedding_dropout_rate`
The dropout rate of the hidden layers: `self.hidden_dropout_rate`
The word embeddings of the document: `word_embeddings`
The size of the hidden layers: `self.hidden_size`

Firstly, you need to remove the batch dimension of the `word_embeddings`, since the LSTM layer takes tensors of rank 3 (3-dimensional). Due to the fact that we are working with the document level task and process one document at a time, the rank of our word embeddings is 4 and with the first dimension (the batch dimension) always equals 1. In order to remove the a dimension that is constantly 1 we can use the backend method `K.squeeze()` and set the `axis=0`. But remember you cannot use the backend method directly you will need to wrap it with a `Lambda` layer, take a look at the code below we used to create the `flatten_word_output` and `mention_pair_emb` or you can find more detailed discussion on how to use the `Lambda` layer at https://keras.io/layers/core/.

Secondly you need to apply a dropout to the word_embeddings using the `Dropout()` layer.

Thirdly, you need to create a two layer bidirectional LSTM (BiLSTMs) by stacking two `LSTM()` layers wrapped with `Bidirectional()` layers. The BiLSTMs need to return the output for all the tokens in the sentences, not just the final one. The output of the BiLSTMs should be called `word_output`. Here is an example of how to create a BiLSTMs using keras: https://keras.io/examples/imdb_bidirectional_lstm/.

Note: Inorder to return the output for all the tokens in the sentences, you will need to set the `return_sequences` attribute to True. The default setting is to return only the final output of the LSTM. And don't forget to apply the `recurrent_dropout`.

The outputs of your LSTMs will be converted to the document level for further use:

```
flatten_word_output=Lambda(lambda x:K.reshape(x,[-1, 2 * self.hidden_size]))(word_output)
```

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 mention pair scores.
The task starts with reshaping the pairwise embeddings to make it a matrix:

```
ffnn_input = Reshape((-1,8*self.hidden_size))(mention_pair_emb)
```

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 (but not the output layer)
3. The outputs are called mention_pair_scores

Tips:
Each hidden layer of the FFNN is a simple `Dense()` with an relu activation function. Layers are simply stacked together the output of the previous layer is the input for the next layer. To apply the dropout you can simply use the `Dropout()` layer. The output layer is slightly different, since it will have an output size of 1. Also in order to compute the binary cross entropy loss we need to give this final layer a sigmoid activation function.

After computing the `mention_pair_scores` you will need to remove the last dimension of it, since the last dimension is always 1. But be careful this time we will need to retain the batch dimension (for compute the loss and training accuracy). Again you can use the `K.squeeze()` method wrapped with a `Lambda` layer.

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_pairs` (the predicted one). The `mention_pairs` contain a list of anaphora and antecedent tuples that are predicted by our model.

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. 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 your system on the data provided. Please make sure all the files are located in the same folder and then type:

```
python coref_model_keras.py
```

to start your training. If all your code is correct your training should finish in about 20 minutes and have CoNLL F1 scores above 50% on both dev and test sets.