✨How to train a neural coreference model— Neuralcoref 2

Thomas Wolf
Mar 23, 2018 · 8 min read

> ***Links:** Online demo **Github repo**: https://github.com/huggingface/neuralcoref and our previous Medium post.*

The last months have been quite intense at HuggingFace 🤗 with crazy usage growth 🚀 and everybody hard at work to keep up with it 🤯, but we finally managed to free some time and update our open-source library ✨Neuralcoref while publishing the training code at the same time.

Since we launched v1 last summer, more than ten million 💯 coreferences have been resolved on Hugging Face. Also, we are stoked that our library is now used in production by a few other companies and some really smart researchers, and our work was featured in the latest session of Stanford's NLP course! 💪

The training code has been updated to work with the latest releases of both PyTorch (v0.3) and spaCy v2.0 while the pre-trained model only depends on Numpy and spaCy v2.0.

> *This release's major milestone: You will now be able to train ✨Neuralcoref on your own dataset — e.g., another language than English! — provided you have an annotated dataset.*

We have added a special section to the readme about training on another language, as well as detailed instructions on how to get, process and train the model on the English OntoNotes 5.0 dataset.

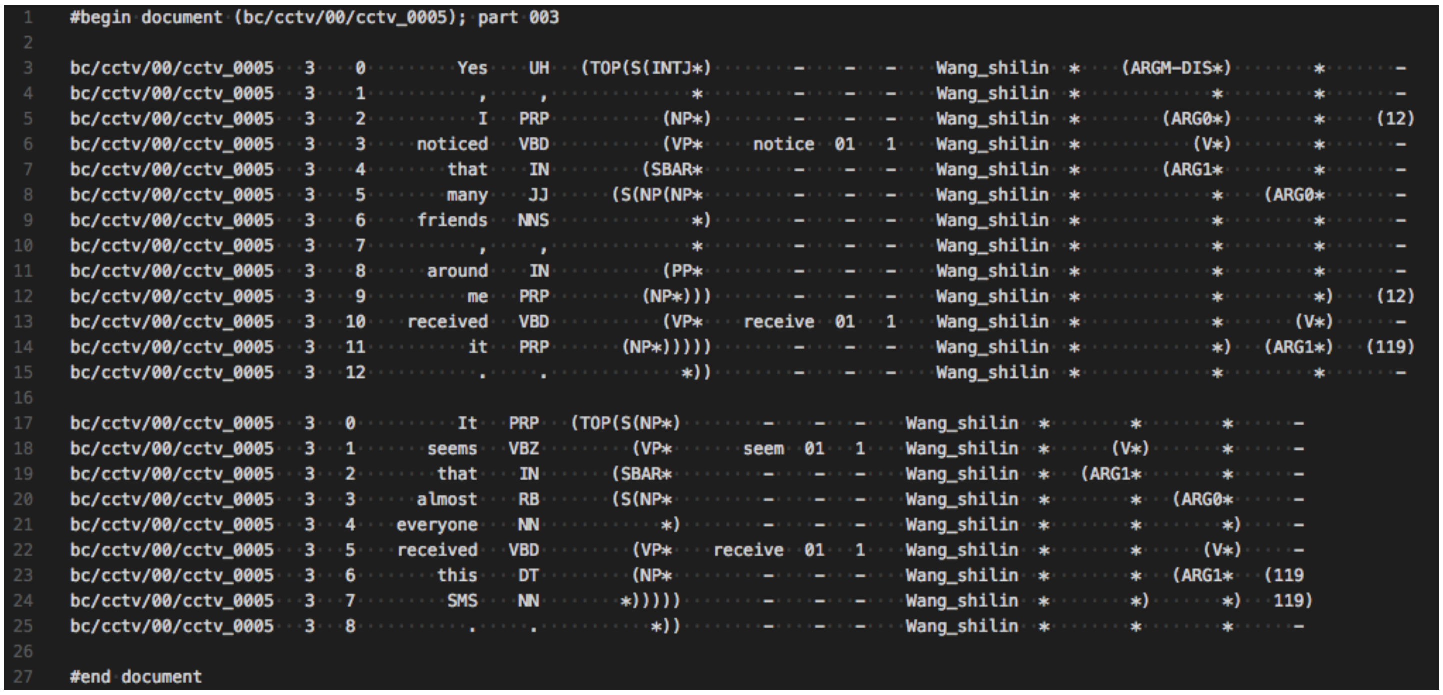
As before, ✨Neuralcoref is designed to strike a good balance between accuracy and speed/simplicity, using a rule-based mention detection module, a constrained number of features and a simple feed-forward neural network that can be implemented easily in Numpy.

In the rest of this blog post, I will describe how the coreference resolution system works and how to train it. Coreference resolution is a rather complicated NLP task 🐉 so bare with me, you won't regret it!

Let's have a quick look at a (public) dataset 📚

A good quality public dataset you can use to train the model on English is the CoNLL 2012 dataset. It is one of the largest freely available dataset with coreference annotations, having about 1.5M+ tokens spanning many fields like newswire, broadcast and telephone conversations as well as web data (blogs, newsgroups …).

In the repo we explain how to download and prepare this dataset if you want to use it. Once you are done with that, a typical CoNLL file will look like this:



```
1   #begin document (bc/cctv/00/cctv_0005); part 003
2
3   bc/cctv/00/cctv_0005  3   0        Yes   UH   (TOP(S(INTJ*)       -   -   -   Wang_shilin  *   (ARGM-DIS*)        *      -
4   bc/cctv/00/cctv_0005  3   1          ,    ,             *       -   -   -   Wang_shilin  *             *        *      -
5   bc/cctv/00/cctv_0005  3   2          I  PRP          (NP*)       -   -   -   Wang_shilin  *        (ARG0*)        *   (12)
6   bc/cctv/00/cctv_0005  3   3    noticed  VBD          (VP*    notice  01   1   Wang_shilin  *           (V*)        *      -
7   bc/cctv/00/cctv_0005  3   4       that   IN        (SBAR*       -   -   -   Wang_shilin  *        (ARG1*         *      -
8   bc/cctv/00/cctv_0005  3   5       many   JJ     (S(NP(NP*       -   -   -   Wang_shilin  *             *   (ARG0*      -
9   bc/cctv/00/cctv_0005  3   6    friends  NNS             *)       -   -   -   Wang_shilin  *             *        *      -
10  bc/cctv/00/cctv_0005  3   7          ,    ,             *       -   -   -   Wang_shilin  *             *        *      -
11  bc/cctv/00/cctv_0005  3   8     around   IN          (PP*       -   -   -   Wang_shilin  *             *        *      -
12  bc/cctv/00/cctv_0005  3   9         me  PRP        (NP*)))       -   -   -   Wang_shilin  *             *       *)   (12)
13  bc/cctv/00/cctv_0005  3  10   received  VBD          (VP*   receive  01   1   Wang_shilin  *             *     (V*)      -
14  bc/cctv/00/cctv_0005  3  11         it  PRP      (NP*)))))       -   -   -   Wang_shilin  *            *)  (ARG1*)  (119)
15  bc/cctv/00/cctv_0005  3  12          .    .           *))       -   -   -   Wang_shilin  *             *        *      -
16
17  bc/cctv/00/cctv_0005  3  0        It  PRP  (TOP(S(NP*)       -   -   -   Wang_shilin  *       *       *    -
18  bc/cctv/00/cctv_0005  3  1     seems  VBZ         (VP*     seem  01   1   Wang_shilin  *    (V*)       *    -
19  bc/cctv/00/cctv_0005  3  2      that   IN       (SBAR*       -   -   -   Wang_shilin  *  (ARG1*       *    -
20  bc/cctv/00/cctv_0005  3  3    almost   RB       (S(NP*       -   -   -   Wang_shilin  *       *  (ARG0*    -
21  bc/cctv/00/cctv_0005  3  4  everyone   NN            *)       -   -   -   Wang_shilin  *       *      *)    -
22  bc/cctv/00/cctv_0005  3  5  received  VBD         (VP*  receive  01   1   Wang_shilin  *       *    (V*)    -
23  bc/cctv/00/cctv_0005  3  6      this   DT         (NP*       -   -   -   Wang_shilin  *       *  (ARG1*  (119
24  bc/cctv/00/cctv_0005  3  7       SMS   NN       *)))))       -   -   -   Wang_shilin  *      *)      *)  119)
25  bc/cctv/00/cctv_0005  3  8         .    .          *))       -   -   -   Wang_shilin  *       *       *    -
26
27  #end document
```

Extract of CoNLL 2012 dataset file "cctv_0005.v4_gold_conll"

This extract contains 2 sentences: "*Yes, I noticed that many friends, around me received it*" and "*It seems that almost everyone received this SMS*"

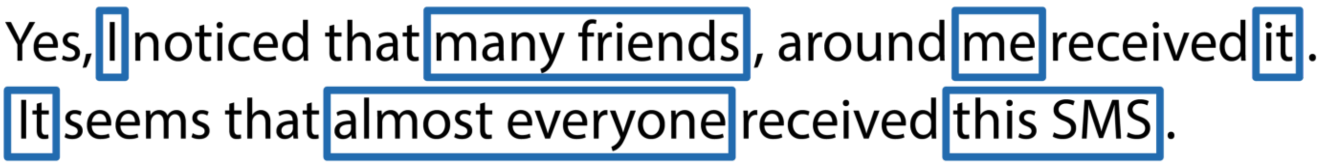
The sentences are tokenized and annotated with the tokens in column 4 and a large number of annotations: POS tags (col 5), parse tree (col 6), verbs lemma (col 7), speaker (col 10) and, what we are especially interested in, **co-reference labels** in the last column (labels 12, 119 on lines 5, 12, 14, 23 & 24). In this extract, "*I*" is annotated as co-referring with "*me*" (they have the same entity label 12) and "*it*" as co-referring with "*this SMS*" (label 119).

You can also notice that *only the mentions that have at least one co-reference are labelled in the dataset* (i.e. at least a pair of mentions referring to the same entity). Single mentions of an entity, with no other mention referring to the same entity, are not labelled.

This is somewhat annoying as it means we cannot fully evaluate (and easily train) the mention identification module of our coreference system (with precision, recall and F1 metrics). However, we can still have a look at the recall of co-referring mentions as we mention in the github repo.

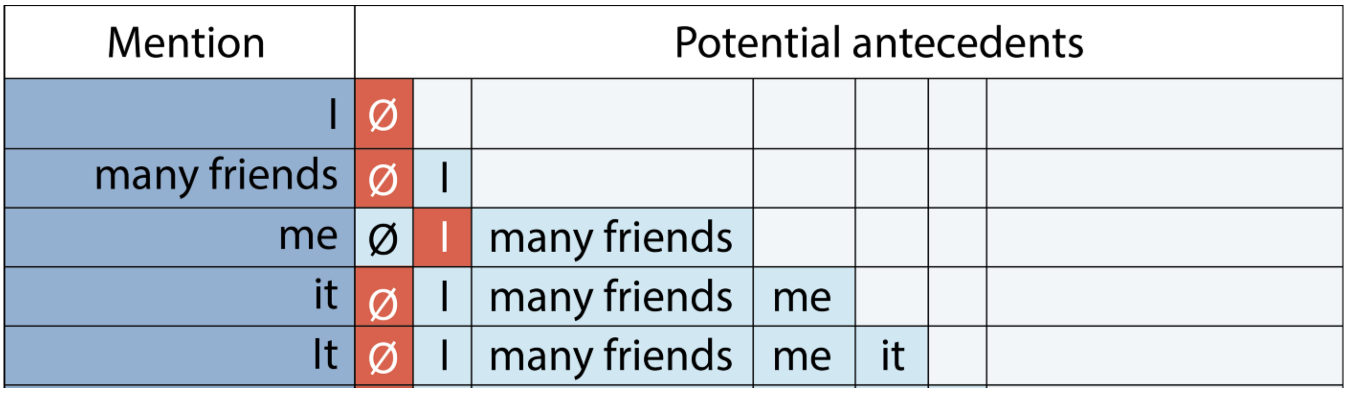
The coreference module workflow 🌊

The first step of our process is to extract potential mentions. ✨Neuralcoref uses a *rule-based mention-extraction* function for this operation and get, in our two-sentences example:

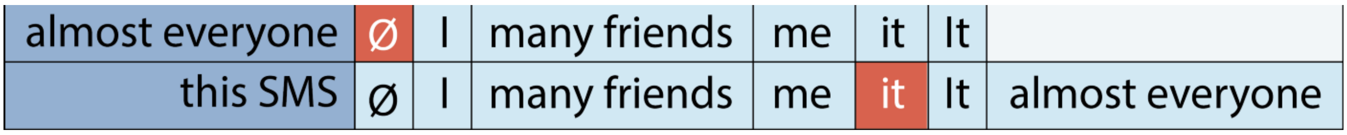

Yes, I noticed that many friends, around me received it. It seems that almost everyone received this SMS.

Depending on the selectivity of the rule-based mention extractor and the parse-tree for our sentence, it may also capture a few bigger mentions like "*many friends, around me received it*" or "*almost everyone received this SMS*". Let's keep only the short mentions here for the sake of simplicity.

Each mention can co-refer with a various number of previous mentions. We can gather all the mentions in a mention-pairs table to highlight the co-referring pairs (in the table Ø means that a mention doesn't corefer with any previous mention).



Mention	Potential antecedents					
I	Ø					
many friends	Ø	I				
me	Ø	I	many friends			
it	Ø	I	many friends	me		
It	Ø	I	many friends	me	it	



almost everyone	∅ 🟥	I	many friends	me	it	It	
this SMS	∅	I	many friends	me	it 🟥	It	almost everyone

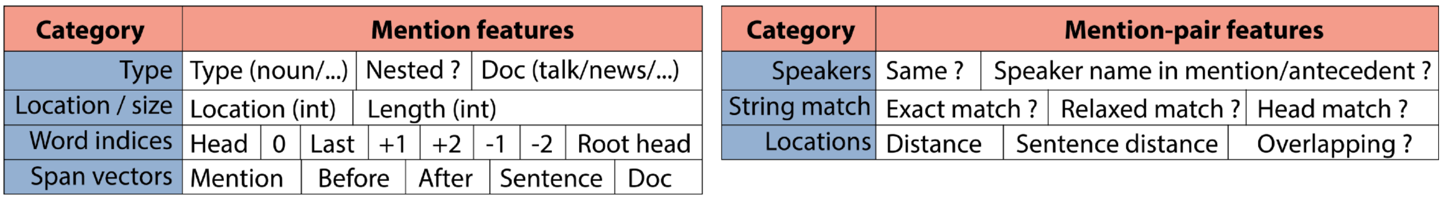
A table of coreferring mention-pairs for our two-sentences example (positive labels in red)

Note that, their may be more than one co-referring antecedent for a given mention (i.e. several red box on a single line on our table), forming clusters of co-referring mentions (co-referrence resolution is a clustering task).

We can already see some of the issues that will arise when training a model on such data, namely that (i) each mention will have a varying number of potential antecedents, complicating the batching (the size of our mention-pairs vector will span all the range from 0 to *N* the total number of mentions in a document), and (ii) the table of mention-pairs will typically scale as *cN* where *c* in the average number of mentions in each document of the dataset, and this can become quite large.

> In practice, our rule-based mention-extractor identifies about 1 million potential mentions on the CoNLL 2012 training set resulting in about 130 millions mention-pairs to train our model on.

Once we have identified potential mentions and labels for them (the red box in our table), we can extract a set of features for each mention and each pair of mentions. Let's see the features we extract:



Category	Mention features							
Type	Type (noun/...)	Nested ?	Doc (talk/news/...)					
Location / size	Location (int)	Length (int)						
Word indices	Head	0	Last	+1	+2	-1	-2	Root head
Span vectors	Mention	Before	After	Sentence	Doc			

Category	Mention-pair features		
Speakers	Same ?	Speaker name in mention/antecedent ?	
String match	Exact match ?	Relaxed match ?	Head match ?
Locations	Distance	Sentence distance	Overlapping ?

Extracted features for mentions and pairs of mentions. Span vectors are pre-computed average of word vectors.

It may seem like we need a lot of features but one of the advantage of ✨Neuralcoref is actually its reduced number of features — some coreference resolution systems uses up to +120 features! Another advantage is that most of these features don't depend on the parser or additional databases (like word gender/number) and they are easy/fast to compute.

Practically, the features are a bunch of *real-valued vectors* (e.g. the span vectors which are average over word vectors and won't be trained), *integers* (e.g. word indices in a dictionary, categorial indices), and *boolean* (e.g. "nested?" indicates whether a pair mention is contained in the other).

The mix of real values, integer and boolean features can give rise to large numpy arrays if we simply gather them in a single array (integer and boolean will be converted to floats). So we store them in separate arrays and build the features arrays on-time while we feed the neural net (see the DataLoader code in dataset.py).

> *We are done with the pre-processing steps. These steps are implemented in conllparser.py and document.py in the code ✨Neuralcoref.*

Now, let's use these features to train our model!

A quick look at the neural net model 🔮

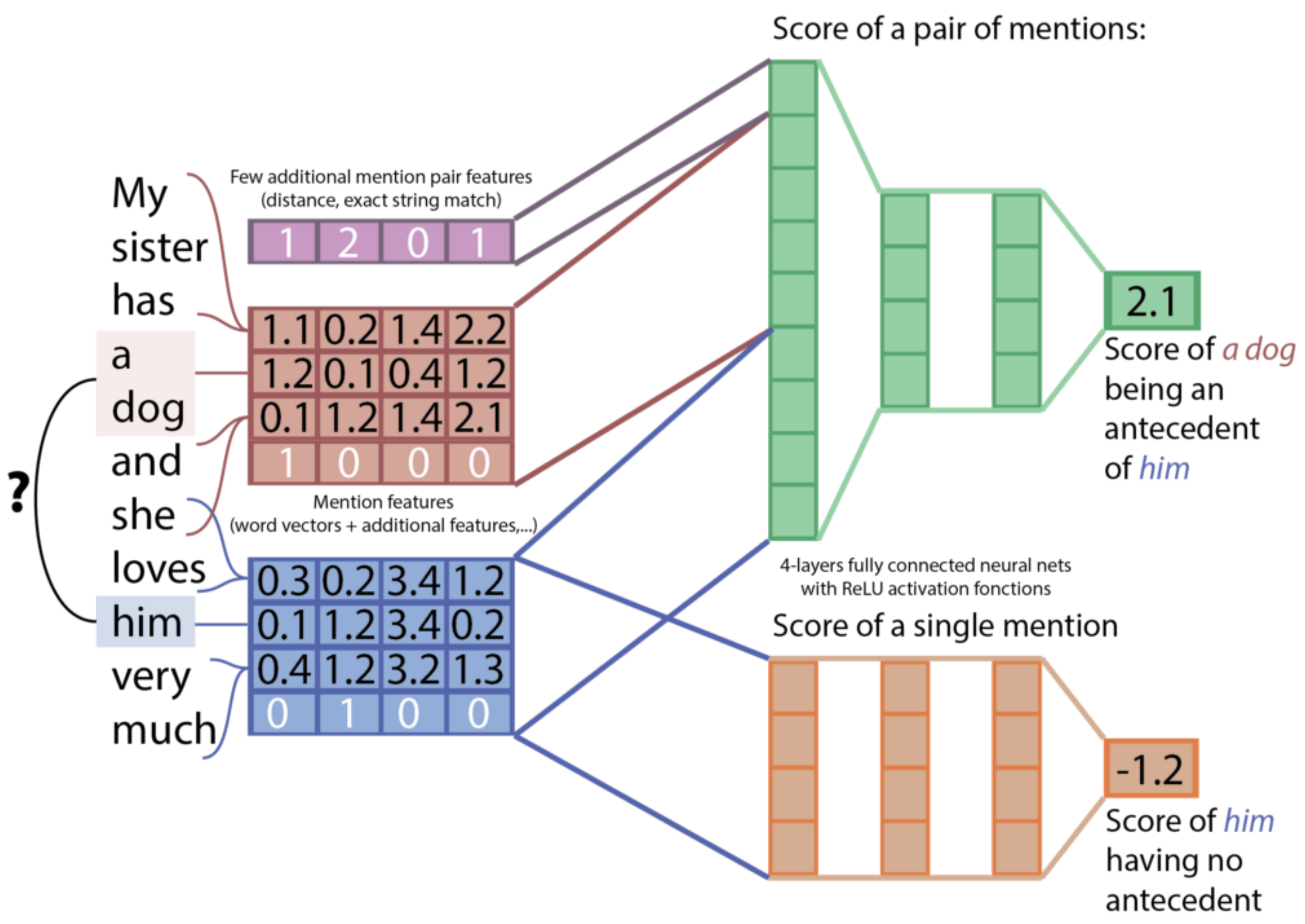
As always, the neural net model is a pleasure to write in pyTorch so I copy it here in full (I just removed weights initialization/loading functions).

```
class Model(nn.Module):
  def __init__(self, vocab_size, embed_dim, H1, H2, H3, pairs_in, single_in, drop=0.5):
    super(Model, self).__init__()
    self.embed = nn.Embedding(vocab_size, embedding_dim)
    self.drop = nn.Dropout(drop)
    self.pairs = nn.Sequential(nn.Linear(pairs_in, H1), nn.ReLU(), nn.Dropout(drop),
                               nn.Linear(H1, H2), nn.ReLU(), nn.Dropout(drop),
                               nn.Linear(H2, H3), nn.ReLU(), nn.Dropout(drop),
                               nn.Linear(H3, 1),
                               nn.Linear(1, 1))
    self.single = nn.Sequential(nn.Linear(single_in, H1), nn.ReLU(), nn.Dropout(drop),
                                nn.Linear(H1, H2), nn.ReLU(), nn.Dropout(drop),
                                nn.Linear(H2, H3), nn.ReLU(), nn.Dropout(drop),
                                nn.Linear(H3, 1),
                                nn.Linear(1, 1))

  def forward(self, inputs, concat_axis=1):
    pairs = (len(inputs) == 8)
    if pairs:
        (spans, words, s_features, a_spans,
         a_words, a_spans, m_words, p_features) = inputs
    else:
        spans, words, s_features = inputs
    embed_words = self.drop(self.embed(words).view(words.size()[0], -1))
```

```
        single_input = torch.cat([spans, embed_words, s_features], 1)
        single_scores = self.single(single_input)
        if pairs:
            btz, n_pairs, _ = a_spans.size()
            a_embed = self.drop(self.embed(a_words.view(btz, -1)).view(btz, n_pairs, -1))
            m_embed = self.drop(self.embed(m_words.view(btz, -1)).view(btz, n_pairs, -1))
            pair_input = torch.cat([a_spans, a_embed, a_spans, m_embed, p_features], 2)
            pair_scores = self.pairs(pair_input).squeeze(dim=2)
            total_scores = torch.cat([pair_scores, single_scores], concat_axis)
        return total_scores if pairs else single_scores
```

The model comprises a common embedding layer, *self.embed,* that transforms *words indices* in word vectors and feed two parallel feed-forward networks:


Score of a pair of mentions:
Few additional mention pair features
(distance, exact string match)
1 2 0 1
My sister has a dog and she loves him very much
?
1.1 0.2 1.4 2.2
1.2 0.1 0.4 1.2
0.1 1.2 1.4 2.1
1 0 0 0
Mention features
(word vectors + additional features,...)
0.3 0.2 3.4 1.2
0.1 1.2 3.4 0.2
0.4 1.2 3.2 1.3
0 1 0 0
2.1
Score of a dog being an antecedent of him
4-layers fully connected neural nets
with ReLU activation fonctions
Score of a single mention
-1.2
Score of him having no antecedent

- *self.single* takes as inputs *word vectors*, *spans* and *additional features* (see above) of a mention and compute the score that it has no other co-referring mention (score of Ø as label),
- *self.pairs* takes as inputs *word vectors*, *spans* and *features* of a mention and an antecedent, together with *pair features,* and compute the score that the pair of mentions are co-referring.

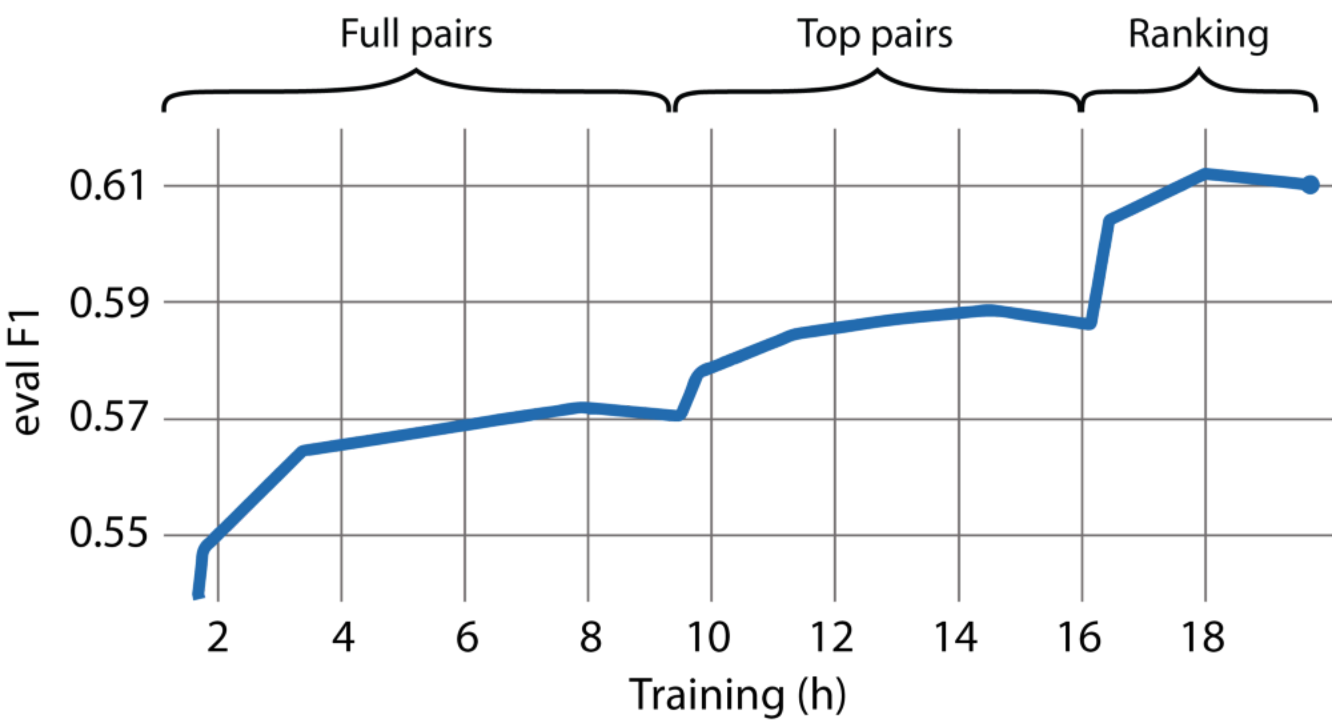
So, how do we train this beauty?

Training the coreference neural net 🌀

First, a word about mini-batching. We talked about the problem of having a varying number of pairs for each mention. One way to use mini-batching in such conditions is to pack mini-batches as follows:

- Sort the mentions in the training set by their number of potential antecedents (the length of each line in our pair table),
- Define a maximum number of pairs P for a mini-batch, and
- Slice the sorted training set in mini-batches of size ≅P, padding the mention-pairs in each mini-batch to the maximum number of pairs in the mini-batch (the longest line, i.e. the last one in our sorted dataset).

In ✨Neuralcoref, this is done by the *dataset.py* module which load and construct a Dataset and a Dataloader with such padded mini-batches.



Example of Neuralcoref evaluation metric during training

Once our mini-batches are ready, we can start training.

The training goes through three successive training phases: *All pairs*, *Top pairs* and *Ranking*.

We set up a very simple scheduling scheme to keep the training fast: each time our metric on the dev set stops increasing, we move on to the next stage.

The first two phases uses probabilistic loss (cross-entropy) while the last phase use a slack-rescaled ranking loss. More precisely, our losses for each training phase look like this:

$$\begin{aligned}
&\text{All pairs loss} && -\sum_{i=1}^{N}\left[\sum_{t\in\mathcal{T}(m_i)}\log p(t,m_i)+\sum_{f\in\mathcal{F}(m_i)}\log(1-p(f,m_i))\right]\\
&\text{Top pairs loss} && -\sum_{i=1}^{N}\left[\max_{t\in\mathcal{T}(m_i)}\log p(t,m_i)+\min_{f\in\mathcal{F}(m_i)}\log(1-p(f,m_i))\right]\\
&\text{Ranking loss} && \sum_{i=1}^{N}\max_{a\in\mathcal{A}(m_i)}\Delta(a,m_i)\left(1+s_m(a,m_i)-\max_{t\in\mathcal{T}(m_i)}s_m(t,m_i)\right)
\end{aligned}$$

$\mathcal{T}$(m) is the set of true antecedents of a mention m, $\mathcal{F}$(m) the false antecedents and $\mathcal{A}$(m) all antecedents (including $\emptyset$).

The *All pairs loss* is a standard cross-entropy loss on the full set of mention-pairs. The *Top pairs* loss is also a cross-entropy but is restricted to the (currently) top-scoring true and false antecedents of a mention. Finally, the *Ranking loss* is a max-margin loss with a slack rescaled cost Δ.

To get more information, you should check the very nice work published in 2016 by Kevin Clark and Christopher Manning (see “Deep Reinforcement Learning for Mention-Ranking Coreference Models” by Kevin Clark and Christopher D. Manning, EMNLP 2016, Improving Coreference Resolution by Learning Entity-Level Distributed Representations by Kevin Clark and Christopher D. Manning, ACL 2016, and the references therein), which our model is an adaptation of.

The full details and more are given in these publications which you should definitely read if you are interested in this model.

> *This training is implemented in learn.py in the code ✨Neuralcoref.*

So I hope this gives you some good intuitions on how this rather uncommon beast works.

Most important, we setup a really nice and fast demo, so don’t hesitate to try the coreference system for yourself!

And don’t hesitate to fork the code and use it in your projects. Hope you liked it and let us know how you use it 🚀

Machine Learning NLP Deep Learning Artificial Intelligence Chatbots