NLP HW 3
Word Sense Disambiguation

Karim Ghonim - Matricola: 1774086

12 Sep 2019

1 Introduction

The purpose of this report is to introduce the reasoning and intuition behind my proposed solution for creating a word sense disambiguation system. Inspiration was drawn from "Neural Sequence Learning Models for Word Sense Disambiguation" by Raganato, Delli Bovi and Navigli in order to create a baseline for the experiments showcased here, modify as well as build upon it, and evaluate it in terms of performance and the results achieved.

2 Dataset

SemCor is a semantically annotated english corpus with all the sense annotations drawn from WordNet 3.0 sense inventory, it also provides the POS tag and lemma for each instance. It is a manually annotated corpus created from texts from Brown corpus. Semcor was used on its own as the training dataset since it being manually annotated ensures that the models are trained on clean data and that annotation errors are kept to a minimum. Semcor consists of $\approx$ 37 thousand sentences containing $\approx$ 26 thousand unique fine grained senses, 151 domains, 45 lexnames, and $\approx$ 36 thousand lemma-pos pairings. The iterparse function from the *lxml.etree* library was used in order to iterate through and parse the dataset.

The framework provides five evaluation datasets, SensEval-2, SensEval-3, SemEval-2007, SemEval-2013 and SemEval-2015. SE07 was used as a dev set during the training process, while all five were used as test sets in order to judge the performance of the models and the experiments in comparison to the baseline.

2.1 Preprocessing

No data-cleaning procedures such as removing punctuations, stop-words were carried out in the final experiments as they impaired the model's performance. My hypothesis for this is that they help the model distinguish some senses at times according to their relative position from a punctuation or a stop-word. Other than lowercasing all tokens and lemmas used during the training process for consistency, all preprocessing implemented was in order to fight the problem of data sparsity. WordNet synsets were considered instead of sense IDs in order to limit the number of classes, with the mappings provided used to generate the fine-grained (babelnet) and coarse-grained (Domains and LexNames) labels.

Each <**lemma**>_<**pos**> pair was assigned a list of candidate synsets for both the fine-grained and coarse-grained cases, with the lemma preferred to the token as <**running**>_<**verb**> and <**run**>_<**verb**> should both have the same possible senses, while the <pos> was included in order to restrict the possible synsets even more. For example, the pair <**plant**>_<**noun**> has ['bn:00035324n', 'bn:00046568n'] as candidate synsets, while <**plant**>_<**verb**> has ['bn:00085671v', 'bn:00087554v', 'bn:00091692v'] as candidate synsets, therefore <**plant**> would have five possible synsets, making it more ambiguous and difficult for the model to learn and predict the correct synset. Restricting the possible synsets (classes) by integrating the POS tag was possible as it's available for each instance thanks to the Raganato et al. format.

The original number of classes for the fine-grained synsets was $\approx$ 50 thousand, consisting of all possible synsets and words. This was cut down in half by mapping all the words with no meaning to the label $< WRD >$ rather than giving each its own index, ending up with $\approx$ 26 thousand classes. The reason behind this is that I wanted to train my network on predicting the senses only and have it completely ignore the words with no meaning. These two methods improved the performance of the model as well as its training time.

3 Network architecture

3.1 Embeddings

Two different approaches were taken in order to provide the embeddings for each instance. The first involved training the embeddings for every <**lemma**>_<**pos**> pair from scratch. This incorporates the information about each instance's POS tag to the next layer of the network, as <**bank**>_<**verb**> and <**bank**>_<**noun**> will both have their own separate embedding, leading to more instance-specific, task-oriented embeddings that are optimized to perform well at the required task.

The second approach uses "Embeddings from Language Models" (ELMo) which provides pre-trained contextualized embeddings. In this case, for each instance the model provides the contextual embedding of the instance's token, as providing the lemma only without the POS tag meant losing information. I opted not to retrain the embeddings and use them as is in order to provide a comparison between using contextual and task specific embeddings.

3.2 Baseline

The Bidirectional LSTM Tagger architecture consisting of one BiLSTM was implemented and used as the baseline for comparison of the suggested models and extensions as previously mentioned.

3.3 Self-attention

The self attention layer was trained to capture the most informative part of the sentence and condense it in a "context vector", as it has the ability to look at past and future information, thus guiding the network to focus on the elements that prove more helpful in providing the correct prediction. The attention layer was added prior to the softmax layer for each prediction, fine-grained or coarse-grained, as well as part of the Seq-to-Seq model's encoder.

3.4 Seq-to-Seq

A complete encoder-decoder model was implemented. The encoder consists of a BiLSTM and an attention layer, with the "context vector" passed on to the decoder instead of the hidden state of the BiLSTM layer after reading the entire sequence. This was offered as a second option in the paper, as it showed an improvement in performance.

3.5 Multitask learning

Multi-task learning helps improve the main task by making use of the training signals in related tasks and exploiting their commonalities. Inspiration was drawn from the multi-task learning architecture suggested in the paper but with several changes implemented. Two auxiliary tasks were considered, being coarse-grained LexNames and Domains rather than Part-of-speech, as the POS tag is provided as part of the LexName, and adding it to the model did not achieve much improvement, if any. A hierarchical model was implemented with the idea that each coarse-grained label -being a Domain or a LexName- is simply related senses grouped together. The first BiLSTM layer passes its output to two attention-softmax pairings for each of the coarse-grained labels, as well as a second BiLSTM layer, who's focus will be refining this output solely focusing on the fine-grained prediction, also through an attention-softmax pair. This lead to having three attention-softmax pairings each focused on their own respective task, while improving each prediction recursively, with the intuition being that if the first section learns to extract the features for a group of senses, the second finer one can only improve as a result.

3.6 Masks

In order to introduce the ideas presented in the preprocessing section to the implementation of the model, two masks were created. The first is the infinity mask, which is a 3D mask (batch_size, max_length, class_number). For each instance an array with the size of the classes in question is created. This array is filled with a "0" for every candidate sense for the said instance's <**lemma**>_<**pos**> pair, and " $-\infty$" for the rest. This mask is added to the logits before being passed to the cross entropy softmax layer, thus setting the probability of all classes except those for the respective candidate senses to "0", ensuring that the prediction will be one of the candidate senses, as well as training only for these senses and not updating unrelated weights. In the case of multi-task learning, an infinity mask is generated for each task separately.

The second is an instance mask, having the same shape of the input batch (batch_size, max_length), filled with a "1" for instances and a "0" for words without meaning respectively. This mask is then multiplied element-wise by the loss matrix in order to nulify or cancel out any loss related to a word without a meaning, not an instance, hence forcing the network to only learn from instances and completely ignore any weight updates that could have been caused by a meaningless word. This mask acts as a back up for the infinity mask.

4 Experiments

The performance of the mentioned architectures was tested using all five evaluation datasets. MFS backoff strategy was implemented so as to provide a prediction for every lemma that was never encountered during training. The domain of any given synset missing from the mapping was set to "factotum". In order to test the model for the multi-task implementation, the F1 score was calculated using the predictions provided by the network and not the provided mappings. Several optimizers were tested with Adam and a learning rate of 10^{-3} providing the best and most consistent results overall. A grid-search was performed to find the best values of the hyperparameters such as batch-size, embedding size (first embeddings case), max-length, and hidden size, with the best values provided in Table 1. Drop-out rates between 0.1 to 0.4 were tested in an attempt to avoid overfitting -which seems to occur due to the small size of the training dataset, specially for the more complex models.

5 Results

All results and graphs of the mentioned experiments are provided in the tables below. The best performing model was the Seq-to-Seq model with self-attention multi-task learning, with the contextual embeddings provided by ELMo. This proves that with more reasonable thought put into the structure of the network -even with a small dataset- reaonable results can be achieved. It is the author's conclusion that with more computing resources in order to perform a larger grid-search and train the model on even more data, improved results are attainable.

Embedding Size	Batch Size	Hidden Size	Max Length	Drop out	Learning Rate	Epochs
200	128	256	60	0.2	10^{-3}	15

Table 1: Hyperparameters

Architecture	SE07	SE2	SE3	SE13	SE15
BLSTM	49.1	52.8	51.6	49.5	49.8
BLSTM + Att	51.1	61.1	57.9	55.1	57.1
BLSTM + Att + Dom + Lex	56.4	63.9	59.6	58.8	58.2
Seq2Seq	49.8	51.4	53.3	51.3	51.4
Seq2Seq + Att	51.3	62.4	61.6	57.6	58.1
Seq2Seq + Att + Dom + Lex	58.2	65.1	65.3	61.9	61.3

Table 2: F-scores (%) Fine-Grained WSD - Trained Embeddings

Architecture	SE07	SE2	SE3	SE13	SE15
BLSTM	52.4	48.3	49.2	48.3	48.4
BLSTM + Att	58.7	62.4	55.1	61.1	59.9
BLSTM + Att + Dom + Lex	62.2	68.6	67.4	66.8	63.7
Seq2Seq	58.1	51.9	52.3	52.2	50.1
Seq2Seq + Att	61.6	65.5	66.4	61.3	61.8
Seq2Seq + Att + Dom + Lex	64.0	70.0	69.1	67.0	65.4

Table 3: F-scores (%) Fine-Grained WSD - ELMo

Architecture	SE07	SE2	SE3	SE13	SE15
BLSTM + Att + Dom + Lex	77.1	85.3	82.5	75.5	74.9
Seq2Seq + Att + Dom + Lex	80.2	87.1	85.1	78.3	78.6

Table 4: F-scores (%) Coarse-Grained WSD - Domains - Trained Embeddings

Architecture	SE07	SE2	SE3	SE13	SE15
BLSTM + Att + Dom + Lex	83.3	87.7	84.9	78.8	79.3
Seq2Seq + Att + Dom + Lex	90.1	92.6	89.1	80.5	84.8

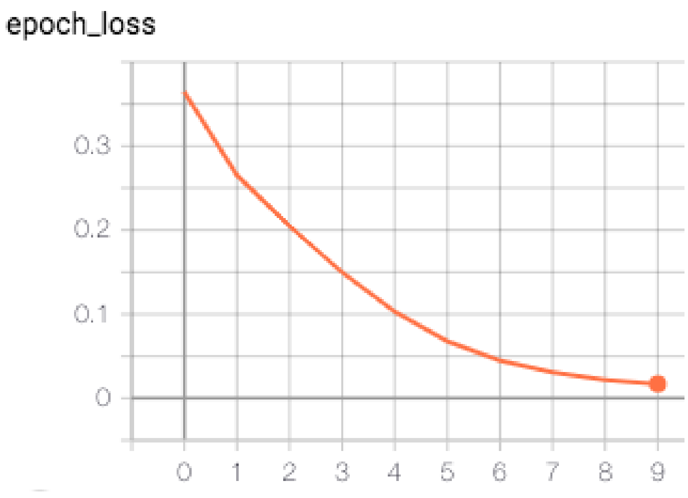
Table 5: F-scores (%) Coarse-Grained WSD - Domains - ELMo

Architecture	SE07	SE2	SE3	SE13	SE15
BLSTM + Att + Dom + Lex	71.2	81.3	79.3	74.1	74.6
Seq2Seq + Att + Dom + Lex	75.7	82.1	82.2	76.7	76.9

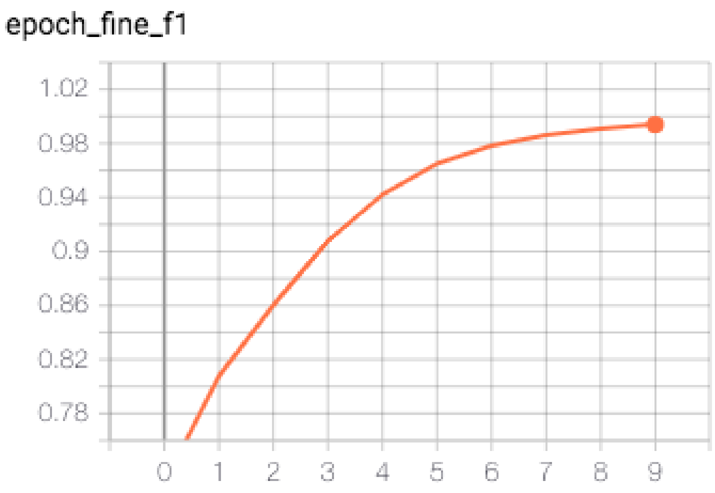
Table 6: F-scores (%) Coarse-Grained WSD - LexNames - Trained Embeddings

Architecture	SE07	SE2	SE3	SE13	SE15
BLSTM + Att + Dom + Lex	74.8	83.8	82.3	77.3	77.5
Seq2Seq + Att + Dom + Lex	79.1	87.0	84.1	78.7	80.2

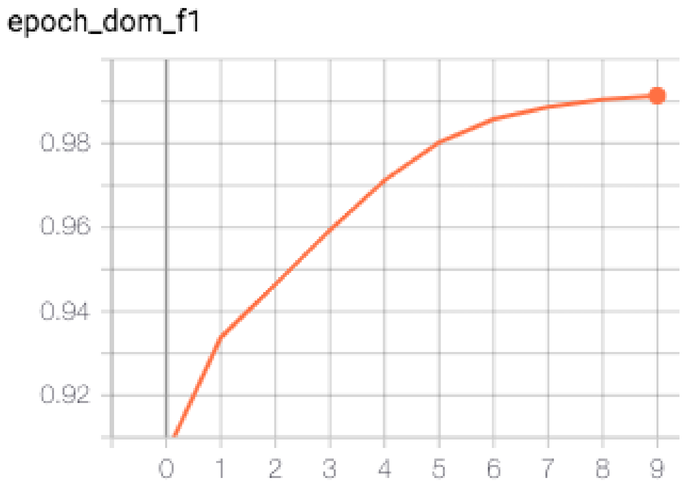
Table 7: F-scores (%) Coarse-Grained WSD - LexNames - ELMo



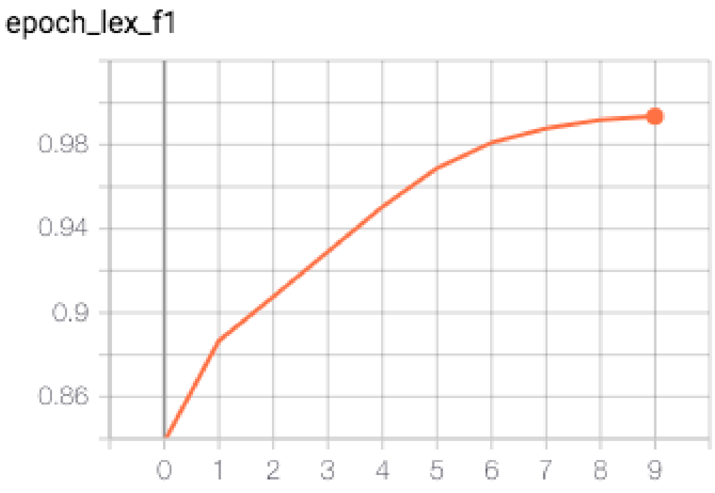
(a) Training loss



(b) Fine-grained F-Score

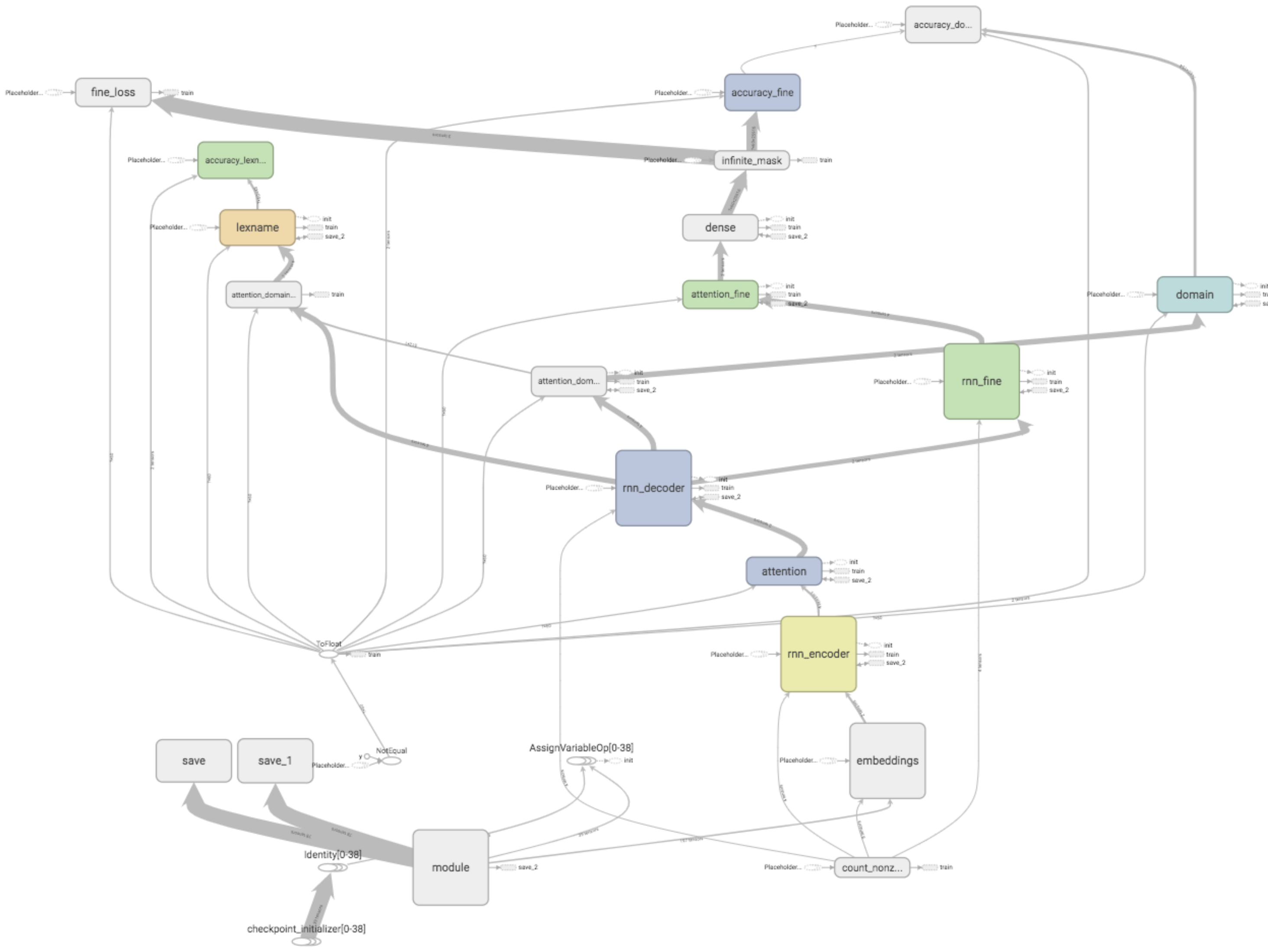


(a) Coarse-grained F-Score - Domains



(b) Coarse-grained F-Score - LexName

Figure 3: Submitted Model Graph



Placeholder...
accuracy_do...
fine_loss
train
accuracy_fine
accuracy_lexn...
infinite_mask
lexname
init
train
save_2
dense
attention_domain...
attention_fine
domain
attention_dom...
rnn_fine
rnn_decoder
attention
rnn_encoder
ToFloat
save
save_1
y
NotEqual
AssignVariableOp[0-38]
embeddings
Identity[0-38]
module
count_nonz...
checkpoint_initializer[0-38]