[Reflections and the next steps – 16/10 and onwards 3](#_Toc30619176)

[Development plan – 17/10 and onwards 6](#_Toc30619177)

[Phase 0 : Vocabulary 7](#_Toc30619178)

[Phase 1 : Retrieving and storing KB & Dictionary data for the Vocabulary 9](#_Toc30619179)

[How to deal with plural and inflected forms? 11](#_Toc30619180)

[Hypotheses 11](#_Toc30619181)

[Using the Vocabulary to retrieve KB data. 12](#_Toc30619182)

[Phrases 13](#_Toc30619183)

[Phase 2: Corpus-based word embeddings 14](#_Toc30619184)

[Embeddings dimension 14](#_Toc30619185)

[From DistilBERT 15](#_Toc30619186)

[Phase 2b: Definitions and Examples 300d embeddings 16](#_Toc30619187)

[Step 2c: Recap of the architecture 18](#_Toc30619188)

[Step 2d: Implementing the alternatives for Definitions & words embeddings 19](#_Toc30619189)

[Phase 3: Graph Neural Network 20](#_Toc30619190)

[Step3a: In the background – retrieving the input data 20](#_Toc30619191)

[Step3b: Not so fast – i.e. Emerging and throwback issues 21](#_Toc30619192)

[Step 3c: Exploring the alternatives for GNN 22](#_Toc30619193)

[Step 3d: (R)GCNs – (relational) Graph Convolutional Network 24](#_Toc30619194)

[Step 3e: PyTorch Geometric 25](#_Toc30619195)

[Directions from the meeting on 18/11 26](#_Toc30619196)

[Phase 4: GNN Implementation and Training 27](#_Toc30619197)

[Step 4a: The parameters of the Data graph object 27](#_Toc30619198)

[Step 4b: Sense-labeled datasets: SemCor + MASC 28](#_Toc30619199)

[Step 4b: More sense-labeled datasets, aligned to WordNet 30](#_Toc30619200)

[Step 4d: Reviewing the Input Data and its Retrieval 31](#_Toc30619201)

[Phase 5: The sense-labeled datasets and the GNN 32](#_Toc30619202)

[Step 5.1: UFSAC datasets 32](#_Toc30619203)

[Step 5.2: Iterator to read UFSAC’s xml files 33](#_Toc30619204)

[Step 5.3: Rel-GCN: Input graph 33](#_Toc30619205)

[Step 5.4: Rel-GCN: Structure and training 35](#_Toc30619206)

[5.4b : Hypothesis for input and output when training 36](#_Toc30619207)

[Step 5.4: Rel-GCN: multiple outputs, loading KB data 37](#_Toc30619208)

[Sense output and loss 37](#_Toc30619209)

[Loading the KB data into the graph 37](#_Toc30619210)

[Step 5.5: Using the SenseLabeledCorpus(es) for training: 39](#_Toc30619211)

[Phase 6: GNN structure and training 41](#_Toc30619212)

[Step 6.1: Segment Training 42](#_Toc30619213)

[Issues 42](#_Toc30619214)

[Time analysis 43](#_Toc30619215)

[Finding the graph segment - v.2 44](#_Toc30619216)

[Step 6.2: Batch training and Validation check 44](#_Toc30619217)

[Preliminary experiments: learning with no batches. 44](#_Toc30619218)

[Batches mechanism 46](#_Toc30619219)

[PHASE 7: Experiments and hyperparameters 47](#_Toc30619220)

[Step 7.1: Only the training loss on small dataset 47](#_Toc30619221)

[Step 7.2: Too slow – save graphArea data in advance 48](#_Toc30619222)

[Indexing 49](#_Toc30619223)

[Step 3: Graph discrepancy 49](#_Toc30619224)

# Reflections and the next steps – 16/10 and onwards

Add-on: are there any related papers incoming from EMNLP 2019?

* “Improved Word Sense Disambiguation Using Pre-Trained Contextualized Word Representations” by Christian Hadiwinoto, Hwee Tou Ng and Wee Chung Gan
* “Challenging Supervised Word Sense Disambiguation with Lexical-Semantic Combinations” by Marco Maru, Federico Scozzafava, Federico Martelli and Roberto Navigli
* “Towards Zero-shot Language Modelling” by Edoardo Maria Ponti, Ivan Vulić, Ryan Cotterell, Roi Reichart and Anna Korhonen
* “MultiFiT: Efficient Multi-lingual Language Model Fine-tuning” y Julian Eisenschlos, Sebastian Ruder, Piotr Czapla, Marcin Kadras, Sylvain Gugger and Jeremy Howard
* “GlossBERT: BERT for Word Sense Disambiguation with Gloss Knowledge” by Luyao Huang, Chi Sun, Xipeng Qiu and Xuanjing Huang
* **Read** references on multi-sense embeddings & language models  
  90%+ done.   
  A quick overview:
  + J. Resinger & R.J. Mooney 2010: represent the words & contexts with TF-IDF / χ-square features, and then senses = centroids of the context clusters
  + Eric H. Huang et al. 2012: local+global(document) FF-NN architecture. Single-prototype phase: Quasi-LM, I see the whole document, but the objective is a score on the next word.  
    Then, cluster the contexts with spherical k-means.
  + X. Chen et al. 2014:   
    Single-prototype phase: Skip-gram  
    Senses from WordNet: definitions and examples form ‘glosses’; we take the average of the candidate vectors in glosses to get the sense embeddings.  
    WSD is done using the cosine-sim between context window and sense.
  + S. Rothe & H. Schutze 2015: AutoExtend  
    Embeddings for synsets and lemmas, using an AutoEncoder. Exploits the linearity assumption of a synset being constructed as the sum of lemmas.  
    Method: word embeddings (possibly pre-trained) -> encode -> synset embeddings -> decode -> re-create word embeddings
  + A. Neelakantan et al., 2015: MSSG, Multi-Sense Skip-Gram model  
    Single-prototype phase: initial global (Skip-Gram) word vectors.  
    2 variants. One with a fixed number of senses, the other that determines the senses based on the locations of the context embeddings (NP-MSSG)
  + S.K. Jauhar et al., 2015: Interacts with an ontology, uses context-independent sense proportions, considering the senses as latent variables
  + J. Li & D. Jurafsky 2015:   
    Uses CRP, Chinese Restaurant Process to create and populate new senses with contexts.  
    WSD can be either greedy or expectation.
  + H. Shi et al., 2016: Exploration of Pseudo Multi-Sense
  + D.Kartsaklis et al., 2018: Mapping text to KB entities  
    Only relevant here because it uses a (Q,K,V) attention mechanism on a fixed number of vectors (k=3) to choose the senses (with soft-attention) and update them.
  + M.T. Pilehvar et al., 2017: Given a text T, turn it into a graph (S,E) of all possible Senses + relation Edges.  
    At each iteration: select the sense s\_ij with the highest degree in the graph.
  + I. Iacobacci and R. Navigli, 2019: LSTMembed  
    1. Obtain a sense-labeled corpus, using the Babelfly instrument   
    2. BiLSTM over the training text > 1 layer > Training objective: minimize the cosine distance from a pre-trained embedding
  + A. Panigrahi et al., 2019: Word2Sense: interpretable embeddings  
    The dimensions have meaning:   
    A sense is a concept, represented by a set of similar words in the vocabulary. Like topics, senses are distributions over the vocabulary.   
    k senses = k dimensions; a word w will have a score on each sense.  
    (3000 -> JS-merge into ¾ -> truncate to 75 eventually)
  + S. Kumar et al., 2019: EWISE  
    Encode context with BiLSTM, then project…   
    Training objective: minimize the cosine distance from the pre-existing sense embeddings, that were created independently, either from pre-trained models, or from KB (e.g. WordNet) resources
* **Read** on evaluation measures for multi-sense embeddings & language models  
    
  In the works I consulted (most of which were mentioned in the previous section), the evaluation measures that were used most often and/or most promisingly were:
  + SCWS (Stanford's Contextual Word Similarities).  
    Better than basic word similarity tasks such as WS-353 or RG-65; it also allows us to apply the WSD mechanism
  + {Downstream tasks: PoS-tagging, SentEval}
  + Downstream task: STS, Semantic Textual Similarity (STS12, STS13, STS14, STS15, STS16, STS-B), and also the SICK relatedness (SICK-R) and SICK-E (where the classes are ‘entailment’, ‘contradiction’, and ‘neutral’)
  + Word Sense Disambiguation: in particular, SemEval 2013 and 2015
  + Finally, Downstream task: Language model  
    WikiText-2 as a small, pocket-sized development set.  
    WikiText-103 as a larger dataset. We should obtain comparable/acceptable perplexity.

Note: it is also opportune to study the impact of dimensionality   
(e.g. 300, 384, 768)

* **A baseline** Multi-Sense LM  
  What models are going to compare our results with?   
  With different models, depending on the selected task. For some of them, When operating on well-known datasets, we do not need to train & implement anything, we can just compare result scores.  
  It is also opportune to consider models that are not too unorthodox (e.g. Pseudo Multi-sense, or interpretable Word2Sense) and not too old (Huang et al. 2012, FF-NN + context-based) .  
  Hypothesis regarding the models:
  + X. Chen et al. 2014. It uses Skip-gram > candidate words in the glosses > vectors for senses.  
    It is an approach relatively close to mine, but without any GNN.
  + A. Neelakantan et al., 2015: MSSG, Multi-Sense Skip-Gram. Here the senses are obtained from the contexts. A generic Multi-sense instrument, examined often and good for comparisons.
  + Transformer-XL, 2019. It already has perplexity values on WT-2 and more importantly WT-103. I even introduced it as a baseline (and a future feature) for Write-Assistant.
  + Possibility: The (pre-trained, unmodified) BERT.  
      
    We mention again here the (candidate) evaluation tasks:
  + SCWS for word similarity.(low priority)
  + **STS (Semantic Textual Similarity on sentences)**
  + Language Modeling perplexity, on WT-2 and WT-103
  + WSD

Given the evaluation tasks, there are already results, there should be no need to train anything anew.

# Development plan – 17/10 and onwards

* **Develop** the Graph-Based Multi-Sense Word Embeddings:
  + Step 0: define a vocabulary
  + Observation: our dictionary-resources system can only deal directly with base, non-inflected forms of words (e.g. only ‘plant’, not ‘plants’ or ‘planted’)
    - Lemmatization for inflected forms when training, to include them in the processing
    - We can worry about this later, in the next phase. Several alternative solutions are possible:
      * Inflected forms can be adjusted using a standard corpus-based approach
      * We can add singular / plural / past edges to the graph
      * We may also concatenate dict-vecs for base forms, and context-based vecs for inflected forms
    - **Different inflected forms distinct for each sense are confirmed as valid. I can recognize “moved to Denmark “ vs. “moved a cup” by operating on a sense annotated corpus.**
  + Preliminary distributional embeddings from training corpus, in order to provide for words without a dictionary definition (for, of, etc.). This also initializes the 'move.global' vectors.
  + Placing in the multi-dimensional space the definitions, and the examples. Sentence embedding based on BERT (average over the word tokens of the 2nd-to-last layer)
  + What about the embedding dimension? I can try:  
    - the original bert-base-uncased size (768)
  + **\* there must be a Bert-small with ~300 dim., and also UlmFit**   
    - PPA+PCA+PPA from V.Raunak et al. 2019 (768 / 2 = 384, or even to 300)  
    - PCA to 300
  + Consider the senses (move.v.1, ..., move.n.3) from the KB sources. How to initialize them? Alternatives:   
    1) on move.global, plus a random perturbation  
    2) Average of all definitions and examples for that particular sense
  + Build Graph Neural Network (Graph Attention Net, or DCGCN)
  + *Language Modeling objective on the training set  
    What if I predict the right word, but the wrong sense?.  
    Look into sense-annotated corpuses / Use example sentences (a held-out portion) / Look into sense-annotation instruments that are not this one itself*
  + Evaluate (see previous discussion)
  + The evaluation should be comparable with:  
    - SoA models on downstream tasks (eg. LSTMs for LMs on WT-2, we should not be too far off)  
    - contextual Transformer embeddings.  
    - And also old Multi-sense embeddings (MSSG, C.R.P.)

# Phase 0 : Vocabulary

Our vocabulary must be useful for Language Modeling.

* Word-level or subword level? Word level.  
  BERT uses internally subword level, but we employ it only to produce sentence embeddings for defs & examples.
* Keep punctuation or remove it? Keep it. Later on, possibly, a version for Write Assistant may remove it.  
  Punctuation should be processed as separate tokens.
* Considering the end task, obviously: keep stopwords.
* Lower-case or not?  
  An interesting possibility is *truecasing*: lower-case only the words at the beginning of a sentence, thus leaving acronyms (USA, CAT) and proper names (Marco Polo) with the correct starting letter.
* Numbers: keep all vs. replace all with <num> vs. keep only 1-digit and 4-digits   
  The last alternative appears to me as the most appropriate
* Non-latin words: if a word contains a character that is non-latin and not even in the Greek alphabet, such as “マギカ”, we replace the word with the <unk> token.

Example line from WikiText-2 (training set):

Father <unk> joined the observatory in 1889 after it was upgraded with more modern instruments . The same year , a second tower was built some 400 metres ( **1 @,@ 300** ft ) away from the main Gregorian Tower , overlooking the Vatican Gardens behind St. Peter 's Basilica on the **south @-@ west** border . It was built to a diameter of 17 metres ( 56 ft ) with a lower wall thickness of **4 @.@ 5 metres** ( 15 ft ) , which could bear the load of a 13 inch photographic <unk> , newly procured from Paris . ………’

Note: It is also necessary to reconvert some of the symbols found in the WikiText datasets, in particular:

‘@-@’ 4 ‘@.@’ 5 metres

Ampersand symbols taken away. Sentence-level tokenization:

'Father <unk> joined the observatory in 1889 after it was upgraded with more modern instruments .',

"The same year , a second tower was built some 400 metres ( **1,300** ft ) away from the main Gregorian Tower , overlooking the Vatican Gardens behind St. Peter 's Basilica on the **south-west** border .",

'It was built to a diameter of 17 metres ( 56 ft ) with a lower wall thickness of **4.5 metres** ( 15 ft ) , which could bear the load of a 13 inch photographic <unk> , newly procured from Paris .'

Without starting and ending spaces (the effect is seen elsewhere, 2 sentences before here), and lowercasing the first word in a sentence …

Tokens:

'father', '<', 'unk', '>', 'joined', 'the', 'observatory', 'in', '1889', 'after', …

Problem: I am decomposing the Unknown token, ‘<unk>’. …

Recap of the steps to process the text of a line in order to build a vocabulary that will be appropriate for a Language Model downstream task:

1. It is necessary to reconvert some of the symbols found in the WikiText datasets, in particular: ‘@-@’, ‘@.@’ etc.
2. nltk's tokenizer - sentence tokenizer
3. Eliminate outer spaces, and true-case the sentences.
4. Split into word tokens, and reunite the lls
5. Recreate <unk> from '<','unk','>'
6. There are word-tokens with non-Latin and non-Greek characters, such as: 匹, 枚, マギカ , etc. We replace those words with <unk>
7. Numbers. All decimals, together with all numbers that are not 1 digit (basic) or 4 digits (years), get replaced with <num>

Later on, these steps could be used to re-write the corpus and save it to disk for other uses.

The vocabulary should be created at the start of everything. Before retrieving the data from the KB sources.

With min\_count = 1, I get a vocabulary of 32,156 from WikiText-2.

Dataset statistics: |V| = 33,278 (close enough)

Emergent problem:

The titles and subsection-titles of the Wikipedia articles are presented in-between ‘=’ signs.

= Valkyria Chronicles III =

…

= = Gameplay = =

…

We thus add a regex check in the vocabulary processing. to remove the ‘ = ‘ segments from the vocabulary creation process.

Executing our pre-processing, we now get:

Vocabulary created, after processing 2,089,445 tokens [statistics= 2,088,628]

32013 rows x 2 columns – (33,278)

{What about normalizing acronyms? Not necessary, especially if we wish to be able to face datasets that have not been pre-processed in the same way}

I decide to set min\_count=2 on the training set of WikiText-2,

finally obtaining 31628 words.

# Phase 1 : Retrieving and storing KB & Dictionary data for the Vocabulary

Reviewing the mini-vocabulary data that we extracted from the KBs with the current code.

Observation: Denominated antonyms have repetitions.

Such as:

move verb.7 refrain

move verb.7 refrain

move verb.7 refrain

move verb.3 stay

light adj.10 heavy

light adj.10 heavy

etc…

* Double-check their creation

Denominated definitions and Denominated examples are both ok.

Denominated synonyms contain duplicates as well.

0 wide adj.4 wide-eyed

0 wide adj.4 wide-eyed

1 wide adv.1 widely

0 wide adj.4 wide-eyed

1 wide adv.1 widely

2 wide adv.2 astray

0 wide adj.4 wide-eyed

1 wide adv.1 widely

2 wide adv.2 astray

…

The Processed Synonyms and Processed Antonyms, that contain the bn\_id as they are before the sense denomination step, already contain the duplicates.

Synonyms and Antonyms (i.e. just after retrievel, before processing) contain no duplicates.

* Reason found: when extending the dataframe, I add it repeatedly, including again the previous elements.  
  It can be seen in the indices, for instance: 0,1,01,2,0,1,2,3,4,0,1,2,3,4,5.

Status: Adjusted.

We get back to reviewing the Input Data that we collected and prepared.

* Definitions:
  + denominated\_definitions.h5, table with fields: word, sense, definitions.
    - 2 light adj.2 having colors relatıvely near white.  
      3 light adj.2 (used of color) having a relatively small amount of coloring agent
    - 6 light adj.1 low in degree or quantity or number (e.g. of rain, snow, accent).
    - 7 light adj.1 not great in degree or quantity or number
    - 9 light noun.1 a small, reusable, handheld device for creating fire.
  + vocabulary\_table.sql, also used for the examples.  
    The database contains:   
    word, sense, vocab\_index, start\_defs, end\_defs, start\_examples, end\_examples  
    The last 4 fields are used to select the appropriate rows in the matrix of embeddings for definitions / examples.
  + vectorized\_definitions.npy
* Examples:
  + denominated\_examples.h5
    - 2 light adj.4 a light load
    - 3 light adj.4 my bag was much lighter once i had dropped off the books.
    - 20 light adj.1 a light sentence
    - …
    - 25 light adj.1 light smoke from the chimney
    - 47 light verb.1 we lit the fire to get some heat.
  + vocabulary\_table.sql, as stated
  + vectorized\_examples.npy
* Synonyms:
  + denominated\_synonyms:
    - move noun.1 motileness
    - move noun.1 freeswimmer
    - move noun.1 motility
    - move noun.2 movement
    - move noun.2 motion
    - light verb.1 kindle
    - light verb.1 ignite
    - light verb.2 illumine
    - light verb.2 illuminate
    - light verb.2 illume
    - light noun.2 sunlight
* Antonyms:
  + denominated\_antonyms.h5
    - wide adj.1 narrow
    - plant verb.1 abolish
    - move verb.2 stay

## How to deal with plural and inflected forms?

Example: what about ‘moves’ (that can be either the verb or the plural), and ‘moved’?

‘moved’ on WordNet :

Redirects to the verb,

1. (v) travel, go, move, locomote

2. (v) move, displace

…

and to one adjective sense:

(adj) moved, affected, stirred, touched

‘moves’ brings to the main page for ‘move’, with 5 noun senses and 16 verb senses.

‘moved’ in BabelNet: using BN.retrieve\_DESA(‘moved')

Out[7]:

({'bn:00096594a': ['Being excited or provoked to the expression of an emotion']},

{'bn:00096594a': ['Too moved to speak',

"Very touched by the stranger's kindness"]},

{'bn:00096594a': ['moved', 'affected', 'stirred', 'touched']},

{'bn:00096594a': ['unmoved']})

‘moves’ gives nothing,   
BN.retrieve\_DESA('moves')

Out[8]: ({}, {}, {}, {})

### Hypotheses

1. Different inflected forms distinct for each sense are confirmed as valid. I can recognize “moved to Denmark “ vs. “moved a cup” by operating on a sense annotated corpus.
2. The Inflected form should be different for each sense, but the transformation verb-to-past or noun-to-plural should not change. The ideal would be to learn a transformation… although it would probably be simpler (and also, not requiring a sense-annotated corpus) to register the translation between ‘move.global’ and ‘moves.global’, and apply this translation starting from each sense.

We can introduce new edge types in the GNN: Plural (for Noun senses), VerbDeclination (for Verb senses).

### Using the Vocabulary to retrieve KB data.

We extracted a vocabulary of 31628 words from WikiText-2, setting min\_count=2.

Problem: BabelNet currently allows me to send 5000 queries a day.

It will be necessary to:

1. insert an Append setting to all operations in PrepareInput.prepare(), or even make it the default setting and add a reset() command.
2. Correction: the Append mode has to start all the way back in GetInputData.

Then:

Track the number of BabelNet requests in GetInputData, or just pick a random number like 3000?

Tracking is better. A global variable in Utils can do it.

If I use this choice… then I must save in a txt file the vocabulary index I processed last.

And then, start the processing from the next one.

This is now implemented. Current threshold before stopping and saving the index : 4950.

“Retrieving Multisense data for word: III”…

Some words have no definitions/data in any of the KB sources.

We must recognize this and move on, before we attempt to access a dictionary field that does not exist.

Retrieving Multisense data for word: 2011

BN\_request\_sender.requests\_counter= 1

({}, {}, {}, {})

({}, {})

{}

// When all dictionaries are empty, we should move on, without attempting to postprocess and compute BERT embeddings for the definitions. In fact, it would be opportune to exclude such terms from the vocabulary\_chunk sent to postprocessing.

Retrieving Multisense data for word: for

BN\_request\_sender.requests\_counter= 1

({}, {}, {}, {})

({}, {})

{}

Retrieving Multisense data for word: Raven

BN\_request\_sender.requests\_counter= 1

({'bn:00086765v': ['Eat greedily'], 'bn:00091936v': ['Prey on or hunt for', 'To prey on or hunt for.'], 'bn:00022930n': ['Large black bird with a straight bill and long wedge-shaped tail', 'A raven is one of several larger-bodied species of the genus Corvus.', 'The common raven, also known as the northern raven, is a large all-black passerine bird.', 'A bird that is usually black, very rarely white', 'A large black bird, similar to the crow, but larger.'], 'bn:00092393v': ['Feed greedily'], 'bn:00092392v': ['Obtain or seize by violence']}, {'bn:00086765v': ['He devoured three sandwiches'…},

{'bn:00086765v': ['devour', 'guttle', 'raven', 'pig', 'devour', 'guttle'], 'bn:00091936v': ['raven', 'prey', 'predate', 'predate', 'prey', 'raven', 'predate'], 'bn:00022930n': ['raven', 'corvus\_corax', 'common\_raven', 'raven', 'common\_raven', 'corvus\_corax', 'raven', 'raven', 'common\_raven', 'northern\_raven', 'raven', 'common\_raven', 'corbie', 'northern\_raven', 'raven', 'ravens', 'canary\_islands\_raven', 'common\_raven', 'corbies', 'corvus\_corax', 'corvus\_corax\_varius', 'corvus\_tingitanus', 'northern\_raven', 'northern\_raven', 'raen', 'western\_raven', 'western\_raven', 'corbie', 'corbie', 'raven', 'ravin', 'raven'…)

{}

Observation: Raven appears to be in Camelcase, despite our truecasing operation on the vocabulary.

Where does it appear in the training set?

Calamity Raven, in the game Valkyria Chronichles III; also, Big Raven.

This highlights another issue:

### Phrases

As of now, we have no phrase aggregation at all in the vocabulary over the training set.

“New York” would be processed as “New” and “York”, just like “Calamity” “Raven”.

The gensim instrument for phrases may come in handy.

Note: In BabelNet, ‘New\_York’ redirects to New York and is still ok.

In WordNet, ‘New York’ is ok, ‘New\_York’ has no correspondences.

What about the parameter values for gensim phrases?

Default: min\_count=5, threshold=10.0

We consider that even in the training corpus of WikiText-2 we get:

United States – 783 results

New York – 509 results

The parameters can be proportional to the training corpus’ size. Hypothesis:

min\_freq = cubic root of total\_num\_tokens // 2 (e.g. cbrt(2.2\*106) / 2 = 65)

phrases\_score\_threshold = min\_freq \* 2 (130)

It is probably better to just train them based on the phrases we get.

For WikiText-2, min\_freq = 40 and phrases\_score\_threshold = 120 is appropriate.

At this point, there are 2 possibilities:

1. Fast & scalable, less accurate for the subsequent Language Model training & task: just add the phrases to our vocabulary.
2. More time-consuming, more precise: re-process the corpus, joining into a\_b all the bigram occurrences.

Choosing: (2)

Add-on:

And what about tri-gram expressions? Such as ‘Salt Lake City’ or ‘World War II’?

Make 2 passes for Phrases: one with min\_freq = 40 and phrases\_score\_threshold = 120, the other one with min\_freq = 40 and phrases\_score\_threshold = 150

Note: It would appear that the benefit from a 2nd pass with the given parameters is very limited (e.g. +30 phrases). It can be removed.

# Phase 2: Corpus-based word embeddings

This is a preparatory phase: given the training corpus, I need to obtain old-style 1-to-1 word embeddings.

This is necessary for 2 reasons:

* The words like “for”, “and”, “of”, and so on that do not have any definition whether in BabelNet or in WordNet.
* To initialize the ‘global’ elements in our graph. They have 3 roles:
  + They are connected to the senses with the ‘*sc’ (senseChildren*) edges
  + They are connected to other ‘w.global’ nodes through the *‘syn’* and *‘ant’ (synonyms* and *antonyms)* edge types
  + They can be used as an initialization anchor for the senses (with a random perturbation)

There are several alternatives for it:

* BERT with the words in absolute isolation. Possibly, small-BERT / distilBert
* UlmFit.
* Skip-Gram.
* Fast-Text (should be slightly better than Skip-Gram, by empirical evidence)

And the dimensionality should be the same one used for definitions and examples, since they end up in the same embedding space.

For the corpus-based word embeddings, as a first choice I can use:

Fast-Text (wiki-news-300d-1M.vec.zip): 1 million word vectors trained on Wikipedia 2017, UMBC webbase corpus and statmt.org news dataset.

## Embeddings dimension

In the spirit of building a low-resource model, and also to have a direct comparison with some past Multi-Sense methods (e.g. MSSG by Neelakantan et al. 2015), it would be opportune to set the embeddings dimensionality to d=300 (at least as an alternative).

d=300 allows us to operate directly with the FastText pre-trained single-prototype word embeddings.

However, this poses a problem later on, when dealing with definitions and examples embeddings (as specified in the module EmbedWithBERT.py). The embedding dimension of bert-base-uncased, etc., and even of DistilBERT that uses one layer every two of the original model, is 768.

Idea for dimensionality reduction:

Add to BERT a last-layer that leads from 768 dimensions to 300, and train it on a Language Model task on the chosen training set. Then, to get the sentence embedding use the same approach considered previously: average each token, producing one 300-length vector.

I can either:

1. add a last linear layer, and fine-tune the whole BERT or DistilBERT architecture on a Language Model
2. or: get the pretrained 768-embedding of the definition/example, and just train the last additional layer (less accurate, but simpler and not requiring more resources than it is feasible)

{Emerging issue: in assign\_senses\_to\_word(word, input\_dbs, output\_dbs), there is a KeyError: 'No object named antonyms in the file' if the word had no antonyms…}

{Updated Tensorflow in Conda-MacOs to 2.0.0 to use SummaryWriter}

## From DistilBERT

Currently: implementing the Finetuning of DistilBERT on a standard Language Model task (on WikiText-2)

{RuntimeError: index out of range: Tried to access index 512 out of table with 511 rows

while training, at outputs = model(inputs, labels)

>

In TextDataset block\_size must be 510 not 512, because this is \*before\* adding [CLS] and [SEP], we would end up out of bounds at 514}

* Status: I have written the code that finetunes DistilBERT on a standard Language Model instead of a masked LM. In the next step, I will add a final linear layer that goes from token dimensionality 768 to 300.
* There were 2 main changes on DistilBERT:
  + Turn all the relevant script arguments into function parameters, and use the default values for the others (gradient accumulation, distributed-hardware training etc.)
  + Introduce the function mask\_last\_token(inputs, tokenizer). Instead of having a Masked Language Model, we train only to predict the last word in a sentence, in a sequential way.
  + ***Course corrections***: I can also state that, since my objective is the quality of representations for definitions and examples, I may as well predict a MLM.  
    My code can also be changed, I can go and use DistilBertForMaskedLM
* Question: does my modified code produce sensible results, comparable with the original examples?

On **WikiText-2**, the SoA is:

Mogrifier LSTM + dynamic eval : Validation perplexity = 40.2, Test perplexity = 38.6

FRAGE + AWD-LSTM-MoS + dynamic eval : V.p.= 40.85 , T.p.= 39.14

On **WikiText-103**, the SoA is:

Transformer-XL + RMS dynamic eval : Valid.P. = 15.8, Test P.= 16.4

Running on the Lambda machine – waiting to see the results (if it is not too much for the hardware… and it is, if I attempt to finetune the whole model, even with DistilBERT. I must use the standard finetuning approach of adding a last layer and modifying only that one).

## Phase 2b: Definitions and Examples 300d embeddings

(old) Objective:

Finetune DistilBERT, adding a last linear layer that goes from dimensionality 768 to 300, in order to use it to obtain 300-d sentence embeddings for Dictionay Definitions and Examples

{Error: outputs = model(inputs, labels)

… in \_\_call\_\_

result = self.forward(\*input, \*\*kwargs)

**TypeError: forward() takes 2 positional arguments but 3 were given**

Let us try with:

outputs = model((inputs, labels))

* result = self.forward(\*input, \*\*kwargs)
  + attention\_mask = torch.ones\_like(input\_ids) # (bs, seq\_length)

TypeError: ones\_like() received an invalid combination of arguments - got (tuple), but expected one of:

\* (Tensor input, torch.dtype dtype, torch.layout layout, torch.device device, bool pin\_memory, bool requires\_grad)

\* (Tensor input, bool requires\_grad)

softmax() is not the right function, I should be looking for crossEntropyLoss() and similar…

}

* Hypothesis:
  + use distilBERT to encode the input window in: 768 x N dimensions
  + Operate with a learned linear layer, that goes 786d x N> 300d x N
  + Use the 300d x N embedding to predict Masked Language Model  
    (note: it’s complicated – it can not be only one layer, I need to keep the words coherent, and I should mask the same positions – there are 2 alternatives to solve this:
    - Modifying BertForMLM
    - Changing the layer, that will produce directly a sentence embedding with (768 x N) > 300. Must change the objective then?, from MLM
  + When the added linear layer has been optimized, apply the sequence:
    - 768 x N (even if we have only |s| words), distilBert embedding for all the words in the sentence
    - 300d x N, after applying the linear layer
    - Get the average over the N slots, in 300d

Observation:

The DistilBERT tokenizer does not include phrases – and it is named ‘distilbert-base-uncased’.

Therefore… I should exclude phrase pre-processing from the training corpus, or maybe use an alternative version of it without it.

Observation:

It is possible that even a dimensionality reduction would \*not\* mean that the Bert-originated sentence embeddings be in the same embedding space as the pre-trained 300d FastText word embeddings.

* **Hypothesis: AutoEncoder approach:**The objective is to create a 768 > 300 Neural Network that holds in 300 dimensions as much information as possible from the sentence embedding from (d)BERT, that was computed averaging over the last layer on (768 x N)  
  + Read a line (768 x |s|). Pad it to (768 x N).
  + Obtain the line embedding, averaging the last layer on (768 x N)
  + Apply linear layer: 768 > 300
  + Decoder part: try to reconstruct the window embedding of 768.

Writing…

{Runtime Error: CUDA out of memory…

torch.cuda.empty\_cache() necessary?

It helps, but the main issue was not using torch.no\_grad()}

Now the Autoencoder is made of Encoder + Decoder where each of them has 2 linear layers:

768 > 534 > 300 , and 300 > 534 > 768

Currently:

* read lines from WT-2 train
* pad or cut to a 256-tokens window
* obtain DistilBERT embeddings
* Train the AutoEncoder to encode in 300 dimensions and then reconstruct.  
  Optimized on the cosine-distance reconstruction loss on the WikiText-2 validation set. Use early-stopping.

**Hypothesis 2) Simple vector average from FastText’s pre-trained 300d vectors**

We recall the previous criticism:

“It is possible that even a dimensionality reduction would \*not\* mean that the Bert-originated sentence embeddings be in the same embedding space as the pre-trained 300d FastText word embeddings.”

This means there are 3 alternatives:

1. Use distilBERT for everything, including single words (taken from the vocabulary of the training set), to produce embeddings of d=768
2. Use distilBERT + encoding for everything, to get to 300d
3. Obtain the sentence embeddings for Definitions and Examples by averaging the FastText pre-trained 300d vectors

## Step 2c: Recap of the architecture

* In CreateGraphInput.py, we have the function exe(do\_reset), that deletes or empties the archive files of the previous run.

* From Vocabulary.Phrases :  
  setup\_phrased\_corpus(initial\_corpus\_fpath, phrased\_corpus\_fpath, min\_freq, score\_threshold)  
    
  It creates and saves a Phrases model, and a corpus with merged bigrams.  
  Is it useful?   
  - Possibly for the FastText vectors   
  *- Not* for DistilBERT that uses its own tokenizer  
  - For BabelNet requests, yes. I need to be able to search “New York” or “United States”  
    
  The output would be located at: TextCorpuses/phrased\_corpus.txt  
    
  We consider that:  
  The creation of embeddings for Definitions & Examples & basic words will use the standard, unmodified corpus to create the vocabulary.

The vocabulary for BabelNet requests be the one from the phrased corpus.

*However*, For the sake of   
1) Building a Version 1.0 ;   
2) Simplicity and streamlining ;   
--> we ignore Phrases for now

* From Vocabulary.Vocabulary:  
  get\_vocabulary\_df(vocabulary\_h5\_filepath, corpus\_txt\_filepath, min\_count)  
    
  The phrased vocabulary is at: Vocabulary/ vocabulary\_phrased.h5  
  The standard, non-phrased vocabulary is at: Vocabulary/vocabulary\_from\_WikiText-2.h5
* From GetInputData.RetrieveInputData: continue\_retrieving\_data()  
    
  We create a BabelNetRequestSender object.  
  We use the vocabulary index file to remember where we were in the vocabulary, since we have a daily limit on the number of requests.  
    
  The chain of core function invocations is : GetInputData.GetWordData.getAndSave\_multisense\_data(word, BN\_request\_sender, open\_storage\_files) >  
   GetInputData.GetWordData.retrieve\_word\_multisense\_data(BN\_request\_sender, target\_word) >   
    
   BabelNet.retrieve\_DESA(BN\_request\_sender, target\_word),  
   WordNet.retrieve\_SA\_bySenses(target\_word, bn\_dicts[0]),  
   OmegaWiki.retrieve\_S(target\_word, bn\_dicts[0])  
    
  The output of this step is: the archives definitions.h5, examples.h5, antonyms.h5, synonyms.h5
* From PrepareGraphInput.PrepareInput: prepare(vocabulary)  
    
  We recall here the phases of pre-processing:  
  # Phase 1 - Preprocessing: eliminating quasi-duplicate definitions and examples, and lemmatizing synonyms & antonyms  
  # Phase 2 - Selecting, sorting and naming (noun.1, verb.4, etc.) the senses of each word   
  # Phase 3 - Create the Vocabulary table with the correspondences (wordSense, integer index).   
  *# Phase 4 - get the sentence embeddings for definitions and examples, e.g. using BERT, and store them*

## Step 2d: Implementing the alternatives for Definitions & words embeddings

I will leave the AutoEncoder aside, for now.

The 2 main competing alternatives are:

1. Use distilBERT for everything, including single words (taken from the vocabulary of the training set), to produce embeddings of d=768
2. Obtain the sentence embeddings for Definitions and Examples by averaging the FastText pre-trained 300d vectors

Note: When creating the single prototype vectors, I do not need to store the word together with the vectors, because they are created (using either FastText or DistilBERT) from the vocabulary.

Thus they are in the same order as in the vocabulary (that can also be used to define their numerical index)

# Phase 3: Graph Neural Network

Now that:

1. we are retrieving Defs, Exs., Syns., Ants. from dictionary sources
2. we compute sentence embeddings for Defs. and Exs.
3. we compute single-prototype word vectors

While computation and BabelNet requests proceed in the background (highlighting a number of bug fixes to make), we move on to the Graph Neural Network.

## Step3a: In the background – retrieving the input data

* ~~Starting with a limit of 200 for the number of BabelNet requests  
  Resetting, and computing also the single prototype embeddings  
  Current vocabulary index = 8~~
* ~~Proceeding with no reset, no SPVs, and 4000 requests~~
* After adjusting the opening and closing of HDF5 archives between runs, restarting with 200 requests. Current vocabulary index = 9
* Proceeding with +1000… index: 59
* Downloading partial results
* Proceeding with +1200… index: 131
* ~~Proceeding with +1200… index: 198 –~~ ValueError due to column size in storing to HDF5
* Proceeding with +1200… index: 198
* Proceeding with +1200… index: 243
* Proceeding with +800… index: 279
* +2000… index: 366
* +2000… index: 446

2 major problems emerging:

1. In the current version, I reset and replace the SQL table with the indices of the sense vectors in their matrix
2. If more than 7K BabelNet Requests, using 2 days, only manage to retrieve multisense data for ~450 words, then a vocabulary of 25-30K words can never be retrieved in sufficient time

Therefore, it is necessary to review the Input retrieval process, and possibly:

1. drop Antonyms, in order not to request the edges of a synset and thus decrease the total number of requests
2. review the requests sent. Is it possible to select only some of the synsets based on their introduction?
3. possibly, drop BabelNet entirely and just use the WordNet API.

## Step3b: Not so fast – i.e. Emerging and throwback issues

When retrieving dicionary data,  
“Japan” has 3 definitions:  
- “lacquerware decorated and varnished in the japanese manner with a glossy durable black lacquer”

- “lacquer with a durable glossy black finish, originally from the orient”

- “coat with a lacquer, as done in japan”

Where did the actual Japan (i.e. the land) go?

“A string of more than 3,000 islands to the east of Asia extending 1,300 miles between the Sea of Japan and the western Pacific Ocean”

We excluded it because it is a Named Entity.

We must include Named Entities, and not only Concepts, in the case where the word is a Capitalized one.

A simple symbol like ‘=” has definitions and senses:

“a sign indicating that the quantities on either side are equal”

“the equals sign or equality sign is a mathematical symbol used to indicate equality.”

…

And ‘)’ too:

“either of two punctuation marks ([ or ]) used to enclose textual material”

“a bracket is a tall punctuation mark typically used in matched pairs within text, to set apart or interject other text.”

…

And synonyms too:

‘=’: {equal-sign, equals-sign}

‘)’: {parenthesis, ], bracket, \rangle,…}

All of this is going to interfere with the usage of the symbols in Language Modeling / word prediction.

Therefore: we do not send BabelNet requests for any token of length == 1.

Must close the h5 files that were open in writing or in append

ValueError: The file 'InputData/denominated\_definitions.h5' is already opened, but in read-only mode. Please close it before reopening in append mode.

In RQD.eliminate\_duplicates\_in\_word…:

word ==''s '

^

SyntaxError: invalid syntax

Adjusted the requests made to HDF5 and did put that in Utils

ValueError: Trying to store a string with len [838] in [examples] column but

this column has a limit of [512]!

In “GetWordData.py", line 101, in store\_data\_to\_hdf5

## Step 3c: Exploring the alternatives for GNN

The representation of the graph (e.g. with an adjacency matrix) is related to its role as the input of the Graph Neural Network.

There are different tools to create manually a GNN, as well as several pre-made implementations of a GAT…

There are different versions of Graph Neural Network, too.

GraphSAGE and Graph Attention networks can be 2 good instruments for our Version 1.0. In comparison, DC-GCN is more refined but more complicated.

Note: GraphSAGE samples a fixed limited number of neighbours for each node; when dealing with a variable number of synonyms and antonyms and definitions and examples, this is suboptimal.

Let us recall Graph ATtention networks. From my own paper-maps:

* Method:
  1. Shared linear transformation on every node,
  2. Non-normalized attention coefficients for neighbours:
  3. Our attention mechanism uses a 1-layer FF-NN:
  4. Get the normalized coefficients using softmax:
  5. Finally, compute the new state of the node:
* Multi-head attention was found beneficial to extend and stabilize the method.  
  It uses either *concatenation* of the attention heads, or *averaging* (n: applying *ρ* later, over the average)

Problem:

The basic version of GATs as-is does not consider the existence of multiple types of edges.



2 alternatives emerge:

1. Use an implementation of R-GCN: Relational Graph Convolutional Networks  
     
   In R-GCN, the update to a node state will be:  
      
   A Layer update evaluates the above equation in parallel for every node.  
     
   >> Note: the R-GCN proposal contains regularization tools like basis decomposition and block-diagonal decomposition
2. In GATs, use different linear transformations for the different relations.  
   *(note: I may also want to read the paper “Relational Graph Attention Networks”)*  
   2 neighbours i and j will be connected by a specific relation, like ‘definition’ or ‘synonym’.
   1. On every node, use a specific linear transformation for a given kind of edge: , ,
   2. The non-normalized attention coefficients for neighbours, using different weights matrices depending on the edge:
   3. Our attention mechanism uses a 1-layer FF-NN:  
      It is possible to use the same vector ***a***, or different vectors for each edge type, because the next step will take care of the normalization through softmax:
   4. Get the normalized coefficients using softmax:
   5. Finally, compute the new state of the node:

## Step 3d: (R)GCNs – (relational) Graph Convolutional Network

The update rule to the node state for basic GCNs is:

We recall that, like other types of networks, it can be viewed as a case of the message-passing framework:

The update-&-propagation model for the R-GCN is the following:

“ where Nir denotes the set of neighbor indices of node i under relation r.   
cir is a problem-specific normalization constant that can either be learned or chosen in advance (such as cir = |Nir| )

Different from regular GCNs, we introduce relation-specific transformations, i.e. depending on the type and direction of an edge.

To ensure that the representation of a node at layer (l + 1) can also be informed by the corresponding representation at layer (l), we add a single self-connection () of a special relation type 0 to each node in the data.

A neural network layer update consists of evaluating in parallel for every node in the graph.”

“ A central issue with applying the update to *highly multi-relational data* is the rapid growth in number of parameters with the number of relations in the graph.

In practice this can easily lead to overfitting on rare relations and to models of very large size.

To address this issue, we introduce two separate methods for regularizing the weights of R-GCN-layers: basis and block-diagonal-decomposition.

*Both decompositions reduce the number of parameters needed to learn for highly multi-relational data (such as realistic knowledge bases).*”

**However,** I do not operate with highly multi-relational data. My graph is very far from being like Knowledge Bases, with countless entities and different relations.

I have 5 edge types:

* sc: senseChildren
* syn: synonyms
* ant: antonyms
* def: definitions from KBs
* ex: examples form KBs

Therefore, I do not need to deal with the complex decompositions.

## Step 3e: PyTorch Geometric

PyTorch Geometric has an already-made implementation of a R-GCN layer:

***class*RGCNConv(*in\_channels*, *out\_channels*, *num\_relations*, *num\_bases*, *root\_weight=True*, *bias=True*, *\*\*kwargs*)**

Note: Edge type needs to be a one-dimensional torch.long tensor which stores a relation identifier ∈{0,…,|R|−1} for each edge.

We examine the parameters:

* in\_channels (int) – Size of each input sample.  
  The number of dimensions of -s and .  
  300 for the FastText-source version, and 768 for the DistilBERT-source version.
* out\_channels (int) – Size of each output sample.  
  In our Version 1.0, the dimensions should not change: 300 and 768 respectively.  
  Possible addendum: go from 768 to 300 for the variant from DistilBERT.
* num\_relations (int) – Number of relations.  
  5 : sc, syn, ant, def, ex
* num\_bases (int) – Number of bases used for basis-decomposition.  
  I wouldn’t need the basis decomposition, since I am operating with few relations.  
  Let us review it here:  
  each weights matrix for a relation r on layer l, , is defined as follows:

as a linear combination of a lower number of common Bases *Vb*, where only the coefficients depend on *r*.

This is, in fact, a hyperparameter. I wil start by setting it to 5: each *Vb* can adapt to one .

* root\_weight (bool, optional) – If set to False, the layer will not add transformed root node features to the output. (default: True)
* bias (bool, optional) – If set to False, the layer will not learn an additive bias. (default: True)

### Directions from the meeting on 18/11

1. *Must research sense-labeled datasets, and find some good candidates*
2. *D,E,SC edges can be unidirectional. Syn and Ant should be bidirectional*
3. *Implement RCGN*
4. *Continue background data retrieval*

# Phase 4: GNN Implementation and Training

## Step 4a: The parameters of the Data graph object

torch\_geometric.data is the object that holds a graph

|  |  |
| --- | --- |
| **Parameters:** | * **x** (Tensor, optional) – Node feature matrix with shape **[num\_nodes, num\_node\_features]**. (default: [**None**](https://docs.python.org/3/library/constants.html#None))  The nodes are: all the single-prototype embeddings + all the sense embeddings (n: that as of now must still be initialized) + all the sentence embeddings from the definitions and examples. All the nodes must have an index, that will go from 0 to num\_nodes -1, and they must have the same dimensionality, num\_node\_features. * **edge\_index** (LongTensor, optional) – Graph connectivity in COO format with shape **[2, num\_edges]**. (default: [**None**](https://docs.python.org/3/library/constants.html#None)) It consists of a vector of Sources and a vector of Destinations. However, we can also operate with a list of S-D tuples as long as we add t().contiguous()) It will be necessary to review the tables and write down the connections between the node indices (i.e. in a new module). A non-directional edge goes in both directions. * **edge\_attr** (Tensor, optional) – Edge feature matrix with shape **[num\_edges, num\_edge\_features]**. (default: [**None**](https://docs.python.org/3/library/constants.html#None))  Our edges have no features. However, we have different types of edges. The RCGN example uses: out = model(data.edge\_index, data.edge\_type, data.edge\_norm) to call forward(self, edge\_index, edge\_type, edge\_norm) we must store the edge types 1:1 in a way that is not edge\_attr … “The data object is not restricted to these attributes and can be extended by any other additional data.” * **y** (Tensor, optional) – Graph or node targets with arbitrary shape. (default: [**None**](https://docs.python.org/3/library/constants.html#None))  The Training Problem. Our objective is: Language Model on a Sense-labeled corpus. Which means: using a softmax on an appropriate Set of nodes in the graph (it may be: all of them, or instead single-prototype for stopwords and separate senses for the other words), in order to predict the next word at sense-level granularity. * **pos** (Tensor, optional) – Node position matrix with shape **[num\_nodes, num\_dimensions]**. (default: [**None**](https://docs.python.org/3/library/constants.html#None)) * **norm** (Tensor, optional) – Normal vector matrix with shape **[num\_nodes, num\_dimensions]**. (default: [**None**](https://docs.python.org/3/library/constants.html#None)) * **face** (LongTensor, optional) – Face adjacency matrix with shape **[3, num\_faces]**. (default: [**None**](https://docs.python.org/3/library/constants.html#None)) These are not used. |

## Step 4b: Sense-labeled datasets: SemCor + MASC

From ai.googleblog.com: *“A Large Corpus for Supervised Word-Sense Disambiguation”*

“… we’re happy to announce the release of word-sense annotations on the popular MASC and SemCor datasets, manually annotated with senses from the NOAD.   
We’re also releasing mappings from the NOAD senses to English Wordnet, which is more commonly used by the research community.   
This is one of the largest releases of fully sense-annotated English corpora.”

There are XML files for both SemCor and Masc, with the format:

<?**xml** version="1.0" encoding="UTF-8"?>

<!**DOCTYPE** SimpleWsdDoc SYSTEM "simple-wsd-doc.dtd">

<**SimpleWsdDoc** name="/written/email/49059.txt">

<**word** *text*="To" break\_level="LINE\_BREAK"/>

<word text=":" break\_level="NO\_BREAK"/>

<word text="j.kaminski@enron.com" break\_level="SPACE\_BREAK"/>

<**word** *text*="Subject"

*lemma*="subject" *pos*="NOUN"

*sense*="/dictionary/sense/en\_us\_NOAD3e\_2012/m\_en\_us1295267.001"

*break\_level*="LINE\_BREAK"/>

<word text=":" break\_level="NO\_BREAK"/>

…

<word text="Pennsylvania" break\_level="SPACE\_BREAK"/>

<word text="Avenue" break\_level="SPACE\_BREAK"/>

<**word** *text*="comes"

*lemma*="come" *pos*="VERB" *sense*="/dictionary/sense/en\_us\_NOAD3e\_2012/m\_en\_us1234657.001" *break\_level*="SPACE\_BREAK"/>

Exploring the lxml and lxml.etree library…

I observe that some words in a sentence are \*not\* sense-annotated.

e.g.: 223,375 // 2 =~ 111,596 senses for 493,594 words in all masc/written

703,248 senses for 1,864,130 words in all semcor

Among the attributes of the **word** elements:

1. *text* contains the word
2. *break\_level* contains the preceding delimiter; e.g. “Early in November” --> NO\_BREAK, SPACE\_BREAK, SENTENCE\_BREAK  
     
   For sense-annotated words:
3. *lemma* executes lemmatization: clouds -> cloud, lifted -> lift
4. *pos*: e.g. NOUN, VERB, ADJ
5. *sense*: the NOAD sense, e.g.: "/dictionary/sense/en\_us\_NOAD3e\_2012/m\_en\_us1268424.001"  
   It can be mapped to a WordNet sense, through either algorithmic\_map.txt or manual\_map.txt

Hypothesis for storing the XML files from SemCor and Masc in one place, where they can be consulted:

HDF5 archive, that also contains rows for the space (break) characters, with columns: text, lemma (even if unused in the current version), pos, WordNetSense (a search operation must be executed on the given mapping).   
The words that do not have a value for some columns can store “EMPTY”.

Currently storing in HDF5.

I need to iterate over the rows when I read the corpus.

Maybe a SQL database with sqlite3 could be simpler/faster/smaller?

Leaving that question aside for a moment?

**Must write to SQL.**

There are many \*ERROR\* fields in the hdf5 currently.

observation: Some NOAD senses are not present in the mapping to WordNet senses, but they are few compared to the total number of senses in a document, even if this phenomenon is more prominent in the MASC corpus, not so much in SemCor.

In SQL, it would be better to store each document as a separate table, so that for example SemCor can be split into training ,validation and test sets.

## Step 4b: More sense-labeled datasets, aligned to WordNet

Observation: Semcor and MASC do not have the entirety of their words annotated with senses.

“There **were** a few miles of note on Thursday. Those responsible **included** Stardel…”

“and most of our **readers** also **realize** that a **large** portion of this expenditures **go**…”

If my aim is to train a GNN to predict word after word, i.e. like a RNN, not like a Transformer, could I find all-words sense-annotated corpuses? And they should be WordNet senses…

Currently reviewing: “UFSAC: Unification of Sense Annotated Corpora and Tools”

“Our work consists in gathering all English corpora sense annotated with WordNet, and convert all of them to a unified format…”

The work contains several corpora that may be of use:

* *SemCor*, the subset of the Brown Corpus
* The *OMSTI* (One Million Sense-Tagged Instances) (Taghipour and Ng, 2015), a corpus of approximately one million words sense annotated with WordNet 3.0
* The *MASC* (Manually Annotated Sub-Corpus), the version given in the article of (Yuan et al., 2016), annotated with NOAD but with corresponding WordNet 3.0 sense keys
* The *Ontonotes 5.*0 (Hovy et al., 2006), annotated with WordNet 3.0.
* The corpora of the WSD evaluation campaigns *SemEval-SensEval*: SensEval 2 (using WordNet 1.7), SensEval 3 (WN 1.7.1), SemEval 2007 (WN 2.1), SemEval 2013 (WN 3.0) and SemEval 2015 (WN 3.0).

“Sense annotations have been converted, when necessary, from their original WordNet sense key to the last version of WordNet (3.0) thanks to conversion tables.

However, because some senses have been dropped from the old versions of WordNet, some sense annotations have not been converted. In any case, the original sense annotations are always kept alongside the converted sense annotation.”

On the UFSAC File format:

organized as: Corpus > Document > Paragraph > Sentence > Word

The statistics of the corpuses: again, not 100% of words are annotated:

|  |  |  |
| --- | --- | --- |
| Corpus | Words – total | Words – annotated |
| *SemCor* | 778,587 | 229,517 |
| *OMSTI* | 35,843,024 | 920,794 |
| *MASC* | 596,333 | 114,950 |
| *Ontonotes 5.*0 | 435,340 | 52,263 |
| *SemEval + SensEval* | etc. | etc. |

Therefore, it is necessary to train the GNN while only part of the words in the corpus we are reading through are annotated. We will examine how to solve this problem later on.  
  
In the meantime, it is necessary either to request fewer definitions or to cut away some sources, because BabelNet is currently too slow.

## Step 4d: Reviewing the Input Data and its Retrieval

From [Step3a (In the background – retrieving the input data)](#_Step3a:_In_the) , 2 problems emerged.

The first only needs a bugfix, but the second requires rethinking the input retrieval process:

1. In the current version, I reset and replace the SQL table with the indices of the sense vectors in their matrix
2. If more than 7K BabelNet Requests, using 2 days,   
   only manage to retrieve multisense data for ~450 words,   
   then a vocabulary of 25-30K words can never be retrieved in sufficient time.

There are several possible alternatives to alleviate problem 2):

1. drop Antonyms, in order not to request the edges of a synset and thus decrease the total number of requests
2. review the requests sent. Is it possible to select only some of the synsets based on their introduction?
3. possibly, drop BabelNet entirely and just use the WordNet API.

Hypothesis 1:

I start from – and use only – WordNet senses.

BabelNet is dropped entirely.

work in progress… need to unpack elems list in GetWordData.py …

unpacking done. Now I should postprocess and eliminate duplicates, in particular, the target word itself should not be a part of the synonyms.

Thanks to using the WordNet senses, it is not necessary to go through Sense Denomination &co.

RemoveQuasiDuplicates has a problem –

The passed where expression: word == 'III'

contains an invalid variable reference

The Vocabulary chunk has the fields: word, frequency

Now the definitions.h5 tablr has the columns sense\_wn\_id, definitions

We must search if a word is in the sense\_wn\_id.

The same modifications must also be applied to the LemmatyzeNyms part.

**Note**:

with the current code, plural forms (e.g. ‘predecessors’) have 0 synset results.

And gerundive forms (e.g. ‘serving’) too.

**Hypothesis**:

In Pass Number 2: lemmatize to find the basic form > get the vector translation between basic and derived form in the single-prototype version > use it to obtain locations for the plural forms of the senses.

**Observation A**:

some words (e.g. ‘developed’) have synsets as adjectives, but they are also the form of a verb (or viceversa).

**Observation B:**

some words have important roles in language as stopwords, but they can also be used as adjectives or in other roles. Example: ‘in’ : 'in.s.01/02/03' and ‘'in.r.01’.

# Phase 5: The sense-labeled datasets and the GNN

## Step 5.1: UFSAC datasets

One of the directives of the Joint Meeting of 22/11 was:  
- Include more sense-labeled training data: not only SemCor and MASC, but also the rest of UFSAC

We review here the corpuses of the UFSAC compilation (*Unification of Sense Annotated Corpora and Tools*, by L. Vial et al., 2018):

* *SemCor*, subset of the Brown Corpus
* The *OMSTI* (One Million Sense-Tagged Instances) (Taghipour and Ng, 2015), a corpus of approximately one million words, sense-annotated with WordNet 3.0
* The *MASC* (Manually Annotated Sub-Corpus), in the version of Yuan et al., 2016, annotated with NOAD but with corresponding WordNet 3.0 sense keys
* The *Ontonotes 5.*0 (Hovy et al., 2006), annotated with WordNet 3.0.
* The corpora of the WSD evaluation campaigns *SemEval-SensEval*: SensEval 2 (using WordNet 1.7), SensEval 3 (WN 1.7.1), SemEval 2007 (WN 2.1), SemEval 2013 (WN 3.0) and SemEval 2015 (WN 3.0).
* The *DSO* (Defence Science Organisation) (Ng and Lee, 1997), a non-free corpus, that is focused on 121 nouns and 70 verbs among the most frequently used and the most ambiguous words in English and have been annotated in various contexts – originally with WordNet 1.5.
* The *WordNet Gloss Tag*, a corpus which consists of all definitions of WordNet (Miller, 1995) with every words sense annotated since version 3.0.
* *Train-O-Matic* by T.Pasini and R.Navigli (2017), which is entirely automatically generated.

|  |  |  |  |
| --- | --- | --- | --- |
| **Corpus** | **Words – total** | **Words – annotated** | **Comment** |
| *SemCor* | 778,587 | 229,517 | file semcor.xml |
| *OMSTI* | 35,843,024 | 920,794 | file omsti.xml |
| *MASC* | 596,333 | 114,950 | file masc.xml |
| *Ontonotes 5.*0 | 435,340 | 52,263 | (code to convert the original data) must still download the data from the site where I am registered |
| *SemEval + SensEval* | 30,912 | 9,761 | file raganato\_ALL.xml |
| *DSO* | 5,317,184 | 176,915 | (code to convert the original data) as above, must still download from the site of the L.D.C. |
| *WordNet GlossTag* | 1,634,691 | 496,776 | file wngt.xml |
| *Train-O-Matic* | ? | ? | file trainomatic.xml However, it is automatically annotated, not manually-> excluded |

First task ToDo: write an interface to iterate (with an Iterator, there is no need to save the data elsewhere) through the xml-s.

It should add EOS (End-of-Sentence) when reading.

## Step 5.2: Iterator to read UFSAC’s xml files

A generator function (dataset\_generator(xml\_fpath)) yields the attribute dictionary of each <word> element in the given XML file.

A <sentence> yields an EOS.

I copy and split the elements of the XML trees, into Training, Validation and Test as 80%-10%-10%

It is necessary to use copy.deepcopy(elem) in order to include the <word>s, since they are subelements of <sentence>s and/or <document>s.

## Step 5.3: Rel-GCN: Input graph

We start from the input graph.

(Taking up again the information from Step 4a: The parameters of the Data graph object)

**x** (Tensor, optional) – Node feature matrix with shape **[num\_nodes, num\_node\_features]**. (default: [**None**](https://docs.python.org/3/library/constants.html#None))  
The nodes are: all the sense embeddings + all the single-prototype embeddings + all the sentence embeddings from the definitions and examples.All the nodes must have an index, that will go from 0 to num\_nodes -1, and the same dimensionality, num\_node\_features.

**edge\_index** (LongTensor, optional) – Graph connectivity in COO format with shape **[2, num\_edges]**. (default: [**None**](https://docs.python.org/3/library/constants.html#None))  
It consists of a vector of Sources and a vector of Destinations.  
However, we can also operate with a list of S-D tuples if we apply t().contiguous())  
  
A non-directional edge goes in both directions. Definition, Example, SenseChild edges can be unidirectional. Synonym and Antonym should be bidirectional.

It is opportune to define a Procedure to load the vectors into X and also register the connections.

The node indices will be:

* sense = [0,se) ; also found in the column vocab\_index of the indices\_table.sql
* single prototype = [se, se+sp) ;
* definitions = [se+sp, se+sp+d) ;
* examples = [se+sp+d, e==num\_nodes)

The Procedure to set up both **x** and **edge\_index** can be   
(note: it may be modified, especially if we want to separate the 2 concerns (a) Initialize submatrices and (b) Set up connections):

* read the archive of processed definitions. We encounter the sense\_wn\_id (e.g. active.n.03), and the def. text
* consider the database indices\_table.sql. It has the columns: word\_sense, vocab\_index (from 0 to se), start\_defs, end\_defs, start\_examples, end\_examples.
* load (for instance) vectorized\_FastText\_definitions.npy. Even with > 20K vectors, it is very light at ~58MB
* Use [start\_defs, end\_defs) to extract the vector(s) for the definition of a sense, and append that vector to X\_defs   
  (n: Using WordNet as a source, we always have 1 definition)
* Register the connections: add a tuple   
  (source=definition\_index=sp+def, target=sense)
* Do the same for examples: get the examples' vectors for that sense from [start\_examples, end\_examples) in indices\_table.sql, and get the corresponding rows of vectorized\_FastText\_examples.npy.
* Append vectors to submatrix X\_exs (to be put together later), and add the connections [d, e==num\_nodes) ->[0,se)

Then, what about the connections between the indices of a word *w* and the indices of the senses, *w.n.1* etc.?

There is a vocabulary of single tokens / words, that we gathered from the WikiText-2 training corpus at the start: vocabulary-fromWikiText-2.h5.

The single-prototype vocabulary has the columns word and frequency, and the index is the one that the token has in the hdf5 archive.

**Observation**: it is opportune to use as the vocabulary source the sense-labeled corpus itself. Read through it while ignoring the senses, instead adding tokens for the surface\_form-s of the <word>s encountered.

The last connections are those for synonyms and antonyms: they go from global (a.k.a. single-prototype) to global.

The archives processed\_synonyms.h5 and processed\_antonyms.h5 have the columns sense\_wn\_id and synonyms/antonyms.

However, we intend to cut off the sense, parse the word, and make connections between Global nodes. They must also be bidirectional.

Currently building an example graph, to develop our GNN on it, and worry about data collection from the proper (large-scale) KB data later.

How to specify the edge type?

The RCGN example uses:  
out = model(data.edge\_index, data.edge\_type, data.edge\_norm)  
to call forward(self, edge\_index, edge\_type, edge\_norm)

“…The data object is not restricted to these attributes and can be extended by any other additional data.”  
Moreover, from the documentation for the RGCNConv layer:

“Edge type needs to be a one-dimensional :obj:`torch.long` tensor which stores a relation id”

Edge type identifier: integer [0, |R|-1) . def=0, ex=1, sc=2, syn=3, ant=4

## Step 5.4: Rel-GCN: Structure and training

**Objective**:

For training with partially sense-labeled data:

we can use 2 heads (or a stack) that predict 2 outputs: word, and sense (sense is activated when needed).

Which means:

1. we always predict the next word, as a ‘global’ single-prototype node in the graph
2. if there is a sense labelling, head n.2 predicts the sense (it could be opportune if we restricted the choices among the senses of the predicted next word. Alternatively, we could just use a softmax over all the sense nodes).

The update-&-propagation model for the R-GCN is the following:

Reviewed and updated from [Step 3e: PyTorch Geometric](#_Step_3e:_PyTorch) :

Parameters of a RGCNConv layer:

* in\_channels (int) – Size of each input sample.  
  We process 1 word at a time, LSTM-like. in\_channels = the dimensions of -s and i.e. 300 for the FastText-source version, and 768 for the DistilBERT-source version.
* out\_channels (int) – Size of each output sample.  
  The number of classes.For the global/single-prototype head, |
* num\_relations (int) – Number of relations. 5 : def, ex, sc, syn, ant
* num\_bases (int) – Number of bases used for basis-decomposition.  
  I wouldn’t need the basis decomposition, since I am operating with few relations.  
  Each weights matrix for a relation r on layer l, , is defined as follows:

as a linear combination of a lower number of common Bases *Vb*, where only the coefficients depend on *r*. This is, in fact, a hyperparameter. I will start by setting it to |R|==5

* root\_weight (bool, optional) – If set to False, the layer will not add transformed root node features to the output. (default: True)
* bias (bool, optional) – If set to False, the layer will not learn an additive bias. (default: True)

5.4b : Hypothesis for input and output when training:

1. In a Language Model task, we iterate over the corpus and receive as input the i-th word, having to predict the next one, the (i+1)-th.  
   This mechanism is the one also used by RNNs / LSTMs  
   However, given how RGCNConv and GCNConv layers work, it may be necessary to send as input the entire data.x, and then predict a softmax over the nodes’ indices for i+1…
2. Input: *i*, index of the i-th word
3. Output:  
   Consider the solution, y=*i+1*
   * if *i+1*  [range of global/single-prototype indices):  
      use only the single-prototype head, that computes a softmax over [se,se+**sp**)
   * if *i+1*  [range of sense indices):  
      a) use the single-prototype head, to attempt to guess the next word correctly  
      regardless of the sense  
      b) use the sense-level head, computing a softmax over [0,**se**) to try to guess  
      the next sense

Alternatives:

* I could use 2 separate layers, one that gives me logits on [se, se+sp) and the other on [0,se)
* or I could use 1 layer that gives logits on the whole range [0, se+sp)
* **or** I could use a 2-layer architecture: 1 layer to construct a representation at level L+1, and then 2 ways: one that goes xL+1 -> [0,se) and the other that goes xL+1 -> [se, se+sp). And the second loss has a modifying effect on the GNN only when we have a label

Following the example for rgcn.py, it would seem that I have to send as parameter to forward(…) the entire graph with data.x, and not only 1 node.

## Step 5.4: Rel-GCN: multiple outputs, loading KB data

2 RGCNConv layers, followed by log-softmax > nll\_loss.

The input of the GNN is the entire data.x graph, following the example.

Output: the probabilities for each node in the graph to be followed by a specific global word.  
In the pocket example: n=55 nodes, with 5 global nodes   
> matrix of [55,5] probabilities, and then logprobabilities   
> extract the row of the current word (at index *i* in the text). The meaning of the probabilities over the global nodes should be “the probability of having the global node *y’* as the next word in *i+1*”

> use nll\_loss comparing with the target, the actual global word in i+1

**Next Tasks**:

1. Adding the parallel sense output:  
   when the target is in the range of senses, we need to:  
   - extract the index of the global word the sense corresponds to  
   - evaluate 2 losses on 2 outputs: the global head and the sense head
2. Writing the LoadingData architecture

### Sense output and loss

The training dataset reads an element from the sense-labeled corpus.

If the next word is not sense-labeled, we send the objective (global index, -1).

If the word is sense-labeled, the label y will be (global index, sense index).

The loss\_global and the loss\_sense are summed up at each step. If there was no sense label, the latter will be ==0.

Loading the KB data into the graph (and taking care of indices etc.):

Partly taken and expanded from ([5.4b](#loadKBintoGraph)).

**The features’ matrix X:**

X (Tensor, optional) – Node feature matrix with shape [num\_nodes, num\_node\_features].

The nodes are: all the *sense embeddings*, all the *single-prototype embeddings*, all the sentence embeddings from the *definitions,* all the sentence embeddings from the *examples*.

All the nodes must have an index, that will go from 0 to num\_nodes -1, and the same dimensionality, num\_node\_features.

The node index will be the row number in the matrix X.

Steps and sources (currently operating with the FastText embeddings, for lower dimension and simplicity):

1. Vectors of word senses’ definitions:   
   load the matrix from InputData/ vectorized\_FastText\_definitions.npy  
   the indices for a particular word sense (move.n.2, etc. will be found in InputData/indices\_table.sql > columns: start\_defs, end\_defs)
2. Vectors of word senses’ examples:   
   as above, load the matrix from InputData/ vectorized\_FastText\_examples.npy
3. Senses’ placement in the multidimensional space:
   1. Random initialization
   2. Average of the vectors of definitions and examples for a sense. It is necessary to iterate through InputData/indices\_table.sql, select the corresponding vectors of definitions and examples, and average
4. Vectors of the global/single-prototype words:  
   Load InputData/SinglePrototypes\_withFastText.npy.   
   The correspondences vector-word can be found in the vocabulary file created previously, for instance Vocabulary/vocabulary\_from\_WikiText-2.h5.  
    *Note: However*, it seems opportune at this stage to create the vocabulary from the sense-labeled corpus itself.   
   It is necessary to use the generator over the training set, and collect the 'surface\_form'-s of the tokens’ dictionaries.   
   We examine here the pre-processing steps that were used for the vocabulary from WikiText-2, and whether they are necessary or not for the vocabulary from the sense-labeled corpus:
   * + - 1. No: Reconverting some of the symbols found in the WikiText datasets, in particular: ‘@-@’4 ‘@.@’ 5 metres.   
            And the title signs, such as in ' = = = Title = = = \n'
         2. Yes: Decode HTML symbols. E.g. &apos;s into ‘s, &quot; into “  
            note: current choice: leave <eos> out of the vocabulary
         3. No: nltk's tokenizer - sentence tokenizer
         4. No: Eliminate outer spaces, and true-case the sentences.
         5. No: Split into word tokens, and reunite the lls
         6. No: recreate <unk> from '<','unk','>'
         7. Yes: There are word-tokens with non-Latin and non-Greek characters, such as: 匹, 枚, マギカ , etc. We replace those words with <unk>
         8. Yes: Numbers. All decimals, together with all numbers that are not 1 digit (basic) or 4 digits (years), get replaced with <num>
         9. Yes: replace ‘\_’ with ‘ ‘. The former is used in phrases.

Observation: the vocabulary from the SLC has pre-made phrases, that at times may be excessive? e.g.: a deep-water port, secrete digestive juices

May have to drop those that do not contain a Cased word

**The description of edges, edge\_index**

Graph connectivity in COO format, with shape [2, num\_edges]. We can operate with a list of S-D tuples, adding t().contiguous()

To set up the different kinds of edges, we have to consult different sources:

* def: definitions --> senses : [se+sp, se+sp+d) -> [0,se)  
  and ex: examples --> senses : [se+sp+d, e==num\_nodes) -> [0,se)
  + Iterate over: InputData/indices\_table.sql, with the columns word\_sense, vocab\_index, start\_defs, end\_defs, start\_examples, end\_examples
  + We have the sources for our edges:  
    [se+sp +start\_defs, se+sp +start\_defs +end\_defs) for defs, and   
    [se+sp+d +start\_examples, se+sp+d +start\_examples +end\_examples) for ex
  + The target will be the index of the word sense: 0 + vocab\_index
  + Add the tuples. Eventually, transpose to create edge\_index
* sc: globals -> senses : [se,se+sp) -> [0,se)
  + Iterate over: InputData/indices\_table.sql, and open Vocabulary/vocabulary\_from\_....h5  
    parse with regex the word sense: ‘application.n.03’ -> ‘application’
  + Target: vocab\_index in indices\_table.sql that we are currently iterating over.  
    Source: the index of the row in the vocabulary where word==’application’
* syn, ant: globals -> globals : [se,se+sp) -> [se,se+sp).
  + Read InputData/processed\_synonyms.h5   
    (or InputData/processed\_antonyms.h5)  
    Columns: sense\_wn\_id, synonyms
  + Open also Vocabulary/vocabulary\_from\_..h5
  + Regex-parse the sense wn\_id to get word 1. Word 2 is already in the nyms column.
  + Add both edge (w1, w2) and (w2, w1)

## Step 5.5: Using the SenseLabeledCorpus(es) for training:

We should use the generator from SenseLabeledCorpus.py

read\_split(split\_name) gives a stream of elements:

…

{'surface\_form': 'the', 'lemma': 'the', 'pos': 'DT'}

{'surface\_form': 'exact', 'lemma': 'exact', 'pos': 'JJ'}

{'surface\_form': 'sum', 'lemma': 'sum', 'pos': 'NN', 'wn30\_key': 'sum%1:21:00::', 'id': 'semeval2013.d012.s023.t004'}

…

I have constructed a RGCN in the pocket example (see class NetRGCN).

Its training set is [(10,-1),(11,5),(12,-1),(13,3),(14,-1),…]

(index of the global word in X, index of the sense in X if present else -1)

It is necessary to:

1. do an index lookup for the global from the surface\_form
2. check the presence of the parameter wn30\_key, translate into ‘move.n.2’ and lookup

n: Operating temporarily on a subset of the sense-labeled corpuses (only SemCor)

Observation:

A number of adjustments are needed on the vocabulary from Sense-Labeled Corpus:

* <eos> should be removed, whereas <unk> should be added
* and probably the punctuation signs as well.
* Apply Truecasing:
  + STATES, DISTRICT, COURT, OF, … all capital letters -> lower
  + In the iterator, after we encounter a <sentence> element, the subsequent word at the start of the next sentence should be lowered
* Mr.\_Barcus : keeping only the phrases where at least one character is uppercase is a valid choice; however we should also replace ‘\_’ with ‘ ‘

Note: splitting one token into a list (e.g. splitting phrases) may be a problem when we have a sense key… unless we check that the lemma (as in, the surface form) is contained in the key

(e.g.: wn30\_key="irregularity%1:04:00::")

and what about terms like “boa\_constrictor”?

It seems opportune to avoid splitting phrases, at least in the current version.

Error message:

RuntimeError: CUDA out of memory. Tried to allocate 25.70 GiB (GPU 0; 10.73 GiB total capacity; 1.39 GiB already allocated; 8.42 GiB free; 14.15 MiB cached)

At:

loss = compute\_loss\_iteration(data, model, current\_token\_tpl, next\_token\_tpl)

> predicted\_global\_forEachNode, predicted\_sense\_forEachNode =

model(data.x, data.edge\_index, data.edge\_type)

Problem:

RGCN takes in as input the entire graph – trying to load it into CUDA, even when operating with a subset version of the vocabulary with min\_count=10, fails.

Possible solutions:

1 GPU has 10 GBs memory. I have 4 GPUs.

It is necessary to use DataParallel and distributed training…

However, if the graph must be used in its entirety, even DataParallel would not help, it would only modify batch-processing (that I don’t have yet)

In the meantime:

trying to set min\_count=50 to reduce the size of the vocabulary, and therefore of the graph:

Constructing X, matrix of node features

X\_definitions.shape=torch.Size([4353, 300])

X\_examples.shape=torch.Size([6724, 300])

X\_senses.shape=torch.Size([4353, 300])

X\_globals.shape=torch.Size([1119, 300])

…

Data(edge\_index=[2, 5506888], edge\_type=[5506888], node\_types=[16549], num\_relations=[1], x=[16549, 300])

Returning to min\_count=10, we get:

X\_d: torch.Size([13046, 300])

X\_e: torch.Size([16200, 300])

X\_s: torch.Size([13046, 300])

X\_g: torch.Size([5508, 300])

What if there is an error in the construction of the graph-data?

Checking. Operating with min\_count=10.

Constructing X, matrix of node features

X\_definitions.shape=torch.Size([13046, 300])

X\_examples.shape=torch.Size([16200, 300])

X\_senses.shape=torch.Size([13046, 300])

X\_globals.shape=torch.Size([5508, 300])

def\_edges\_se.\_\_len\_\_()=13046

**exs\_edges\_se.\_\_len\_\_()=22957086 ?**

sc\_edges.\_\_len\_\_()=13046

…

There was a mistake in creating the examples’ edges –

After the correction:

X\_definitions.shape=torch.Size([13046, 300])

X\_examples.shape=torch.Size([16200, 300])

X\_senses.shape=torch.Size([13046, 300])

X\_globals.shape=torch.Size([5508, 300])

def\_edges\_se.\_\_len\_\_()=13046

exs\_edges\_se.\_\_len\_\_()=16200

sc\_edges.\_\_len\_\_()=13046

syn\_edges.\_\_len\_\_()=7800

ant\_edges.\_\_len\_\_()=1648

Out[5]: Data(edge\_index=[2, 51740], edge\_type=[51740], node\_types=[47800], num\_relations=[1], x=[47800, 300])

Even after the correction, I still get:

RuntimeError: CUDA out of memory. Tried to allocate **17.35** GiB (GPU 0; 10.73 GiB total capacity; 247.99 MiB already allocated; 9.57 GiB free; 14.01 MiB cached)

# Phase 6: GNN structure and training

The current RGCN structure, that exceeds GPU memory, has:

* 1 layer from the graph (with nodes with 300 features) into the intermediate representation h=300 (or h=64, it appears it still doesn’t make much difference in terms of memory)
* 2 second layers:
  + going from (graph x h) to (graph x globals), using (log)softmax to predict the probability for each node of being followed by a specific global word
  + going from (graph x h) to (graph x senses), using (log)softmax to predict the probability for each node of being followed by a specific word sense

This architecture is redundant. Is there a way to take only one node at a time as input, and producing as output the probabilities for that one?...

Once I have produced the representation at layer L+1 with RGCNConv, I do not need to have a softmax over every single node of the 47K+ nodes in the graph. I can have the 2 softmax-es (on globals and on senses) only for the chosen node…

At the second level, there are now 2 linear layers, of dimensions [num\_node\_features, vocabulary]. The pocket example still manages to overfit on the training set.

Unfortunately, even with the RGCN + 2 Linear layers instead of the RGCN + 2 RGCNs, the error is the same:

RuntimeError: CUDA out of memory. Tried to allocate 17.35 GiB (GPU 0; 10.73 GiB total capacity; 141.34 MiB already allocated; 9.67 GiB free; 12.66 MiB cached)

## Step 6.1: Segment Training

I should train over only a **segment** (or **batch**, using a potentially ambiguous terminology) of the graph at a time.

Due to the fact that NNs require a fixed input, the shape of the batch/subset\_input\_matrix X should be set in advance.

Example: given that the current node is at index *k*, retrieve N neighbours, at i=1,2,… hops of distance.

It is necessary to write a function that, given a node index, and a data.edge\_index COO matrix of edges, retrieves the neighbours.

We need to define priorities:

* Hop distance *i*
  + definitions
  + examples
  + synonyms
  + antonyms
* Hop distance *i+1*
  + etc…

Copying part of the graph with select\_index, and defining a Variable with torch.no\_grad…

### Issues

**Issue**: Error:

starting\_node\_index=13046

node\_index=13046 -> node\_neighbours\_edges=[]

The feature matrix X of the graph has shape [0, 13046)…

Index error or the <UNK> token?

node\_index=13046 -> node\_neighbours\_edges=[] means that the <UNK> token has no connections to any other node in the graph. Possible way to deal with it: skip the location.

**Issue**:

Must add: the wn30\_key such as ‘say%2:32:00::’ should be transformed not only into ‘say’, but into the full sense ID (that here, as we see, is actually ‘state.v.01’)

We encounter 'friday%1:28:00::' as a wn30 sense key, but (operating with the current min\_count) we do not find 'friday.n.01' in the SQL DB of senses

**Issue**:

device-side assert: Assertion `t >= 0 && t < n\_classes` failed

a label -1 is not allowed, and that is what I am currently using when there is no sense specification.

Observation: if we have no sense label, then the sense\_loss is not included in the backward() computation. Even if the GPU-side code complains that I have a label of (-1) for the sense, it is never used.

--> Therefore, I can put any placeholder label for the sense without influencing the weights&prediction. For instance, turning -1 into 0.

In the end, the problem was that adding the last\_sense\_idx brought the label prediction for globals beyond the range, in [8K, 13K) instead of [0, 5K). We have 5K output classes after all. Fixed.

**Issue**:

Step n.653: (…)

in convert\_tokendict\_to\_tpl

sense\_index\_queryresult = senseindices\_db\_c.execute(query).fetchone()

sqlite3.OperationalError: near "s\_race": syntax error

**Issue**:

Step n.2613:

in convert\_tokendict\_to\_tpl

wordnet\_sense = wn.lemma\_from\_key(token\_dict['wn30\_key']).synset().name()

ValueError: too many values to unpack (expected 2)

Seems due to a wrong format occasionally found in wn\_30 sense keys > adding exception.

### Time analysis

I am still in “Step 4432…” of epoch 1.

Since an iteration appears to take too much time, we do a time analysis. From the log:

t1 - t0 = 0.00374

t2 - t1 = 7e-05

**t3 - t2** = 1.51648

t4 - t3 = 0.00135

t5 - t4 = 0.00089

The line:

loss = compute\_loss\_iteration(data, model, graphbatch\_size, current\_token\_tpl, next\_token\_tpl)

takes 1.5 seconds, and the majority of the time spent in an iteration.

Examining the compute\_loss\_iteration function:

t1 - t0 = 1.52684

t2 - t1 = 0.00045

t3 - t2 = 2e-05

t4 - t3 = 0.0

t5 - t4 = 4e-05

t6 - t5 = 0.00015

The line responsible for the majority of the delay is:

batch\_x, batch\_edge\_index, batch\_edge\_type = GraphSegments.get\_batch\_of\_graph(current\_token\_index, graphbatch\_size, data)

Thus, it is opportune to review/optimize the algorithm that finds the graph segment that we use for RGCN propagation (since we previously saw that we can not load the entire graph without breaking the GPU memory).

### Finding the graph segment - v.2

My Breadth-First-Search algorithm is actually fast, 0.003 or 0.005 seconds.

The main responsible for spending **1.5 seconds** is:

edges\_indices = torch.Tensor([e\_col **for** e\_col **in** range(len(graph.edge\_index[0]))**if** (graph.edge\_index[0][e\_col] **in** batch\_elements\_indices  
 **and** graph.edge\_index[1][e\_col] **in** batch\_elements\_indices)]).to(torch.int64).to(DEVICE)

Modified the check “can this 1-element Tensor be found into the batch\_elements\_indices Tensor?” into “can this integer be found into the set?”, as follows:

edges\_indices = …**if** (graph.edge\_index[0][e\_col].item() **in** set(batch\_elements\_indices\_ls)…

This brings down the time to **0.62 seconds**.

However, maybe it is possible to be faster by just registering the edges I come across in the BFS?

**Confirmed**. Now the average time spent in creating a graph segment is much less:

In the current version, with no batches, we use 1.6/1.7 seconds per 100 iterations.

Even SemCor.xml[training] has 694K words (since the word total is 778,587), which means 11,798 seconds > 196 minutes > 3.27 hours

Which is reasonable. However, it is opportune to use 1/10th of the SemCor dataset (69.4K) as development set for faster attempts. The Vocabulary of globals will be extracted from WikiText-2.

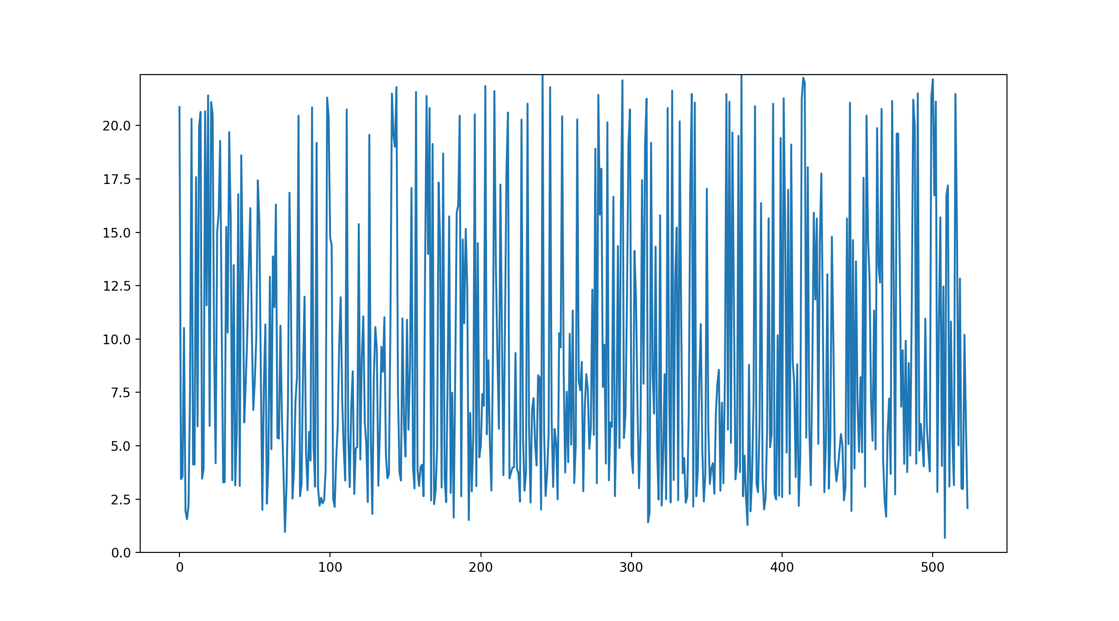
## Step 6.2: Batch training and Validation check

### Preliminary experiments: learning with no batches.

What is the status of the current not-batched mechanism? Does it manage to learn something, to use training to improve the LM prediction?

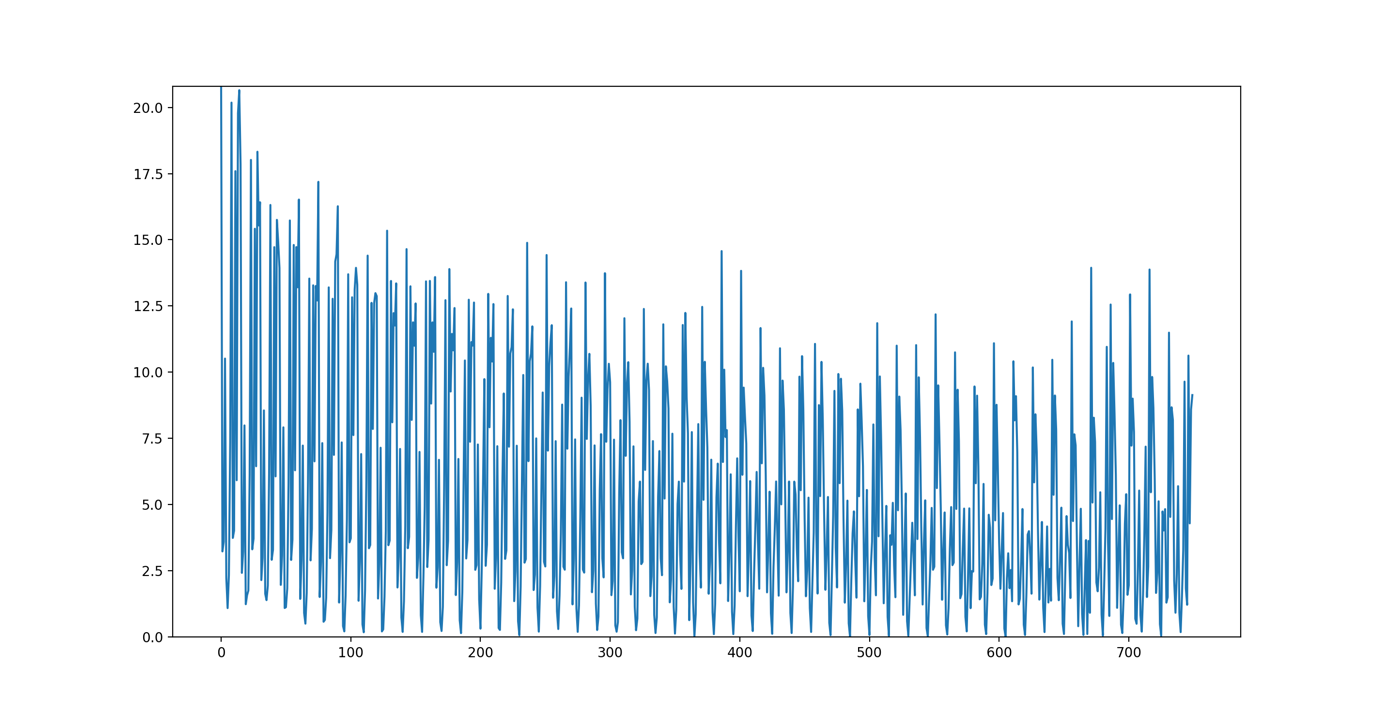
Optimizer currently in use: Adam.

Executing 53.4K steps for 1 epoch, the answer would appear to be “no”:



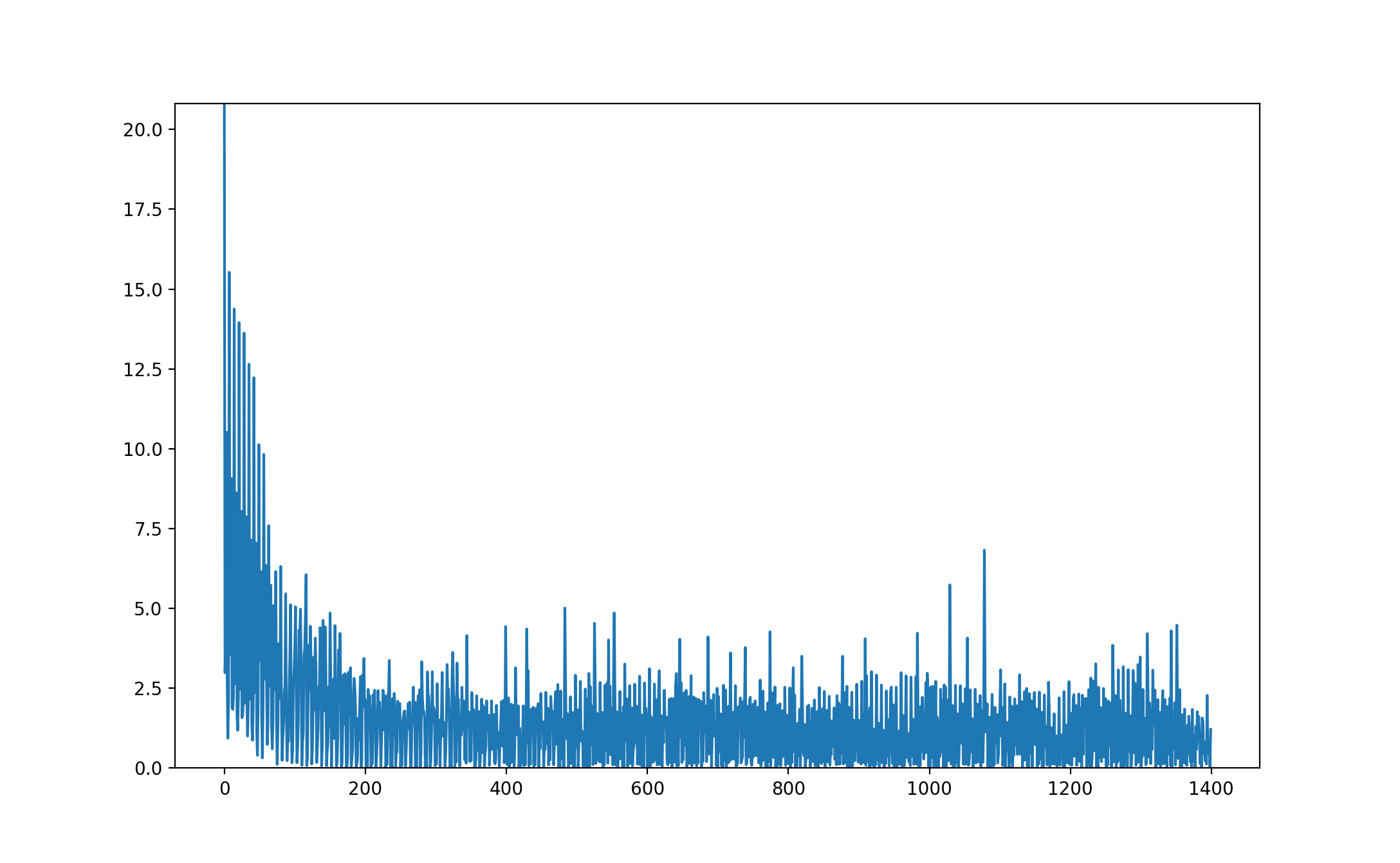
Therefore, we attempt to overfit on a smaller training set:

2K words, that with <unk>s become ~1.5K steps, 50 epochs, graph area size=16.



Marginal improvement. Examining always the same points (i.e. the same (current token, next token) points), the result changes depending on how easy/difficult it is to predict the next token.

Now: only 700 steps, grapharea\_size = 32, and 200 epochs:



It finally manages to overfit on a small training set. Epochs needed: 200K / 700 =~30.

### Batches mechanism

If I had to manually build batching with batch\_size==8:

* we take 8 tuples of indices from the words of the text (sequentially)
* we apply the Breadth-First-Search algorithm to find the graph segment for each starting node (again, sequentially)
* we concatenate/apply the union/join the indices of the nodes and the edges to include, thus finding a wider graph segment that allows the GNN to process the whole batch
* get the model output and compute the nll\_loss, using the batch dimension for tensors
* apply loss.backward() and optimizer.step() on the accumulated loss (probably, accumulation = the sum of the loss)

Problem: the RGCN needs a fixed input size. I should ensure that the BFS on every node eventually returns a graph-area of fixed size.

For instance, instead of a simple extending of lists / union of sets… I could add the i-th element in every adjacency list…

or if there is some overlap between them, I may end up not having enough nodes to add. This can be countered by: get adjacency lists that are 2x the size (2\* (graph\_segment\_size / num\_batch\_nodes)), and then pick the i-th element in every list.

*Or maybe I do not need to modify the model: I can just pass N vectors in parallel to the model during the training iteration, to obtain batch processing*.

As of now, I am not able to use automatic batching without introducing modifications:

The forward function for my RGCN + Linear network is:

forward(self, batch\_x, batch\_edge\_index, batch\_edge\_type)

Automatic batching works as follows: as input, we send Tensors with one dimension that is the batch dimension (usually the first).

from T([5,14,23,76,91,100])

>to: T([[5,14,23,76,91,100],

[20,11,5,89,200,4],)

Dimensions of batch\_x : [grapharea\_size, nodevectors’ dimensionality d]

Dimensions of batch\_edge\_index: [2, any(depending on the edges of the included nodes)]

Dimensions of batch\_edge\_type: [2, any(depending on the edges of the included nodes)]

Therefore, I opt for manual batching: I sequentially call model(…), while *not* resetting with zero\_grad(), in order to accumulate the losses. Then, I concatenate the predictions & labels’ tensors with torch.cat(…) and call the nll\_loss.

(From the Pocket model) With a batch size of 4, I get:

4 tensors of global log-probability predictions, and 4 global class labels: tensor([1, 2, 3, 4])

2 predictions & labels for the senses: tensor([5, 3])

The only remaining point of potential improvement is:

If there is a valid sense label at the current element, I am using only that one to predict, instead of using the global word as well.

However, that may be inevitable given the nature of Language Model prediction? I have to learn to guess the next word from the previous one, while reading sentences.

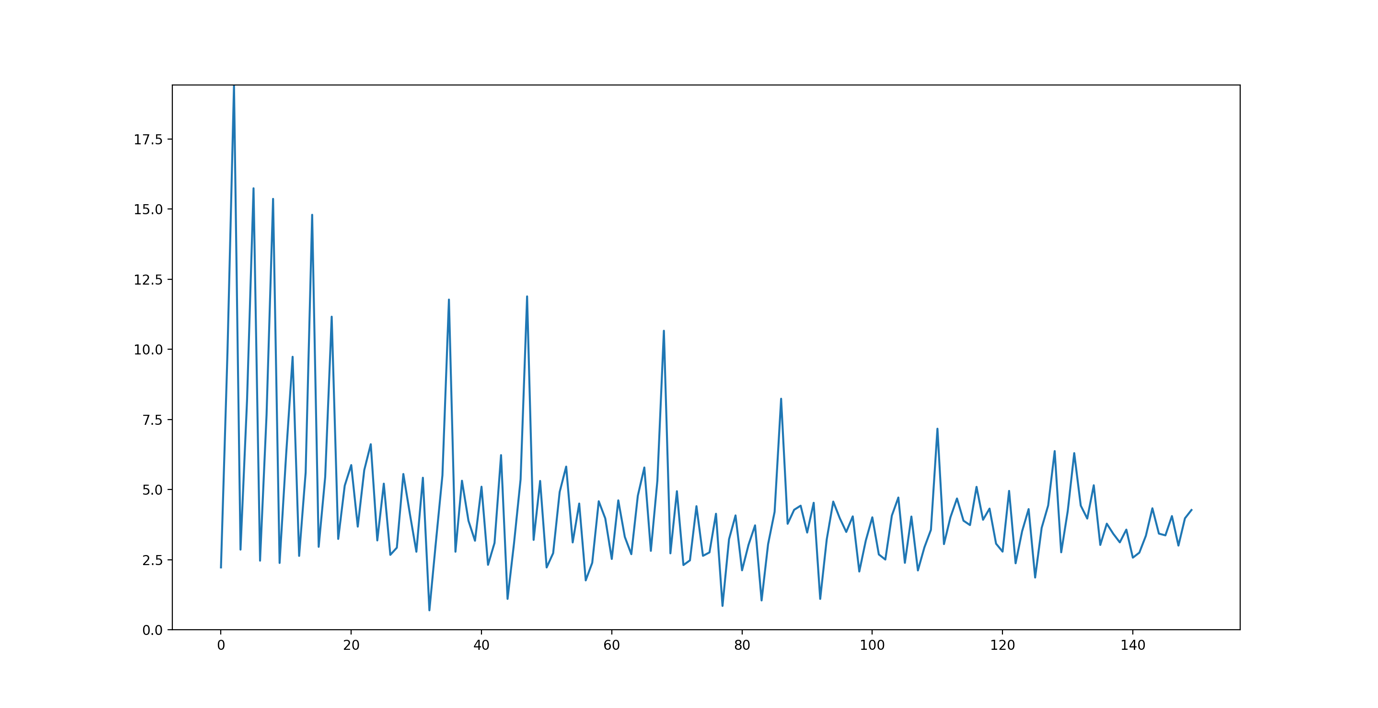
# PHASE 7: Experiments and hyperparameters

## Step 7.1: Only the training loss on small dataset

Steps=756. grapharea\_size = 32, batch\_size = 1. Epochs = 50

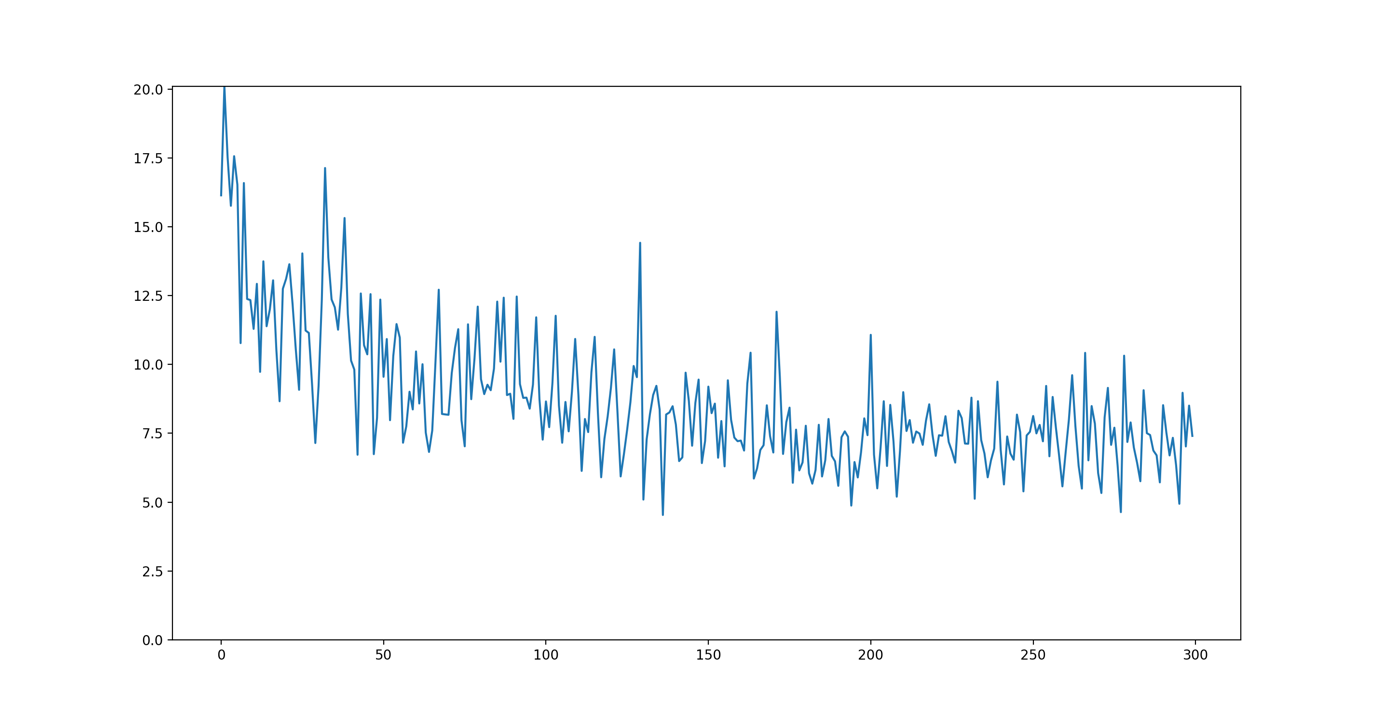
Training set: a fraction of semcor.xml, containing only 756 effective tokens after UNKs

Objective: overfitting on small training set.

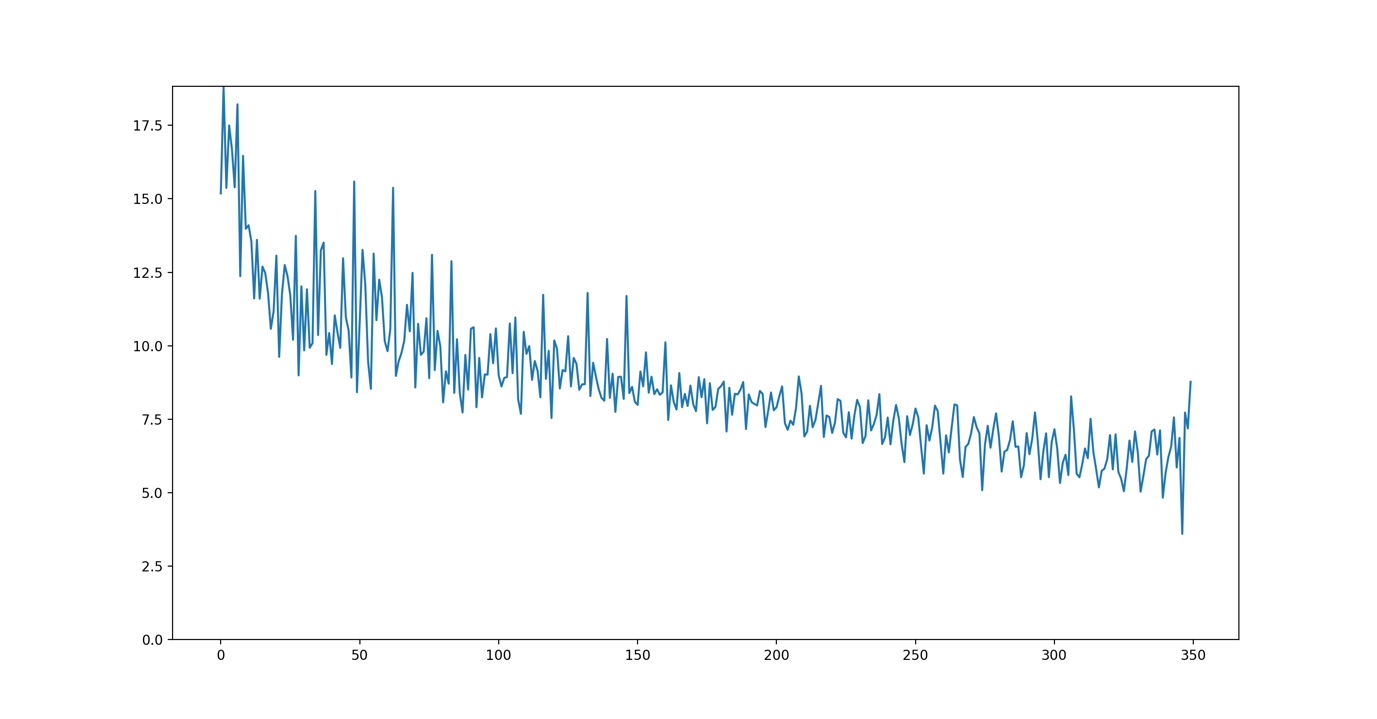


Let us try first with different parameters, and then with a larger training set + a check on the validation accuracy.

Steps=756. grapharea\_size=32, batch\_size=4, num\_epochs=50



Steps=756. grapharea\_size=32, batch\_size=8, num\_epochs=50



Observations:

Batch size=1 is the faster to overfit, but also the least consistent.

Batch size=8 presents a constant, consistent improvement that would need more epochs.

Batch size=4 is in between the two, neither fast nor consistent.

Any further experiments should be improved with a number of modifications:

* Add perplexity (that is simply torch.exp(cross\_entropy\_loss)) as an error measure
* Add a check on the loss&perplexity on the validation set, at the end of each epoch.
* Slightly increase the size of the training set. (It should be still fast enough to be used for development, but also representative enough to allow the model to learn non-insignificant knowledge that should improve the performance on the validation set)

## Step 7.2: Too slow – save graphArea data in advance

{Issue: too slow again. 1 step, that operates on 1 batch, takes from 0.95 to 2.1 seconds.}

In 1 Iteration:

t2 - t1 = 0.7, 0.8 seconds.

loss = select\_and\_process\_batch(…)

Inside the function, for instance:

t1 - t0 = 0.03361

t2 - t1 = 0.82932

t3 - t2 = 0.00536

t2-t1 is:

*# gathering the graph-input for the RGCN layer*

**for** i **in** range(len(input\_indices\_lts) - 1):

…get\_forwardinput\_forelement(global\_idx, sense\_idx, grapharea\_size, data)

which leads to GraphArea.get\_graph\_area(…)

Computing the GraphArea for one node is relatively fast, and there are no major improvements to be found there.

t1 - t0 = 0.01498

t2 - t1 = 0.00421

t3 - t2 = 0.12154

t4 - t3 = 8e-05

Possible solution: pre-create graph-areas for each node, and save them on disk.

Advantage: instead of computing the graph\_area (the adjacent nodes and edges) once for every occurrence of a word in the training corpus, we pre-compute it only once for word.

Note: This decreases the degree of flexibility: I have to save all the adjacent nodes & edges at a given distance/area size for the node *n*. I choose: 8, 16, 32, 64, 128.

What about the file format? SQL vs HDF5 …

Different files/tables can be maintained and updated more easily.

Alternative: a .npy could be lighter. I could load it into memory and search for the node’s row.

Final alternative: a matrix, saved in a Tensor.

### Indexing

The nodes can be stored in order, so I would retrieve directly row *i* for node *i*.

This means there must be pre-set boundaries/indices, specifically:

Given k = graph\_area\_size ,

m = area\_size \* max\_edges\_per\_node :

[0 : k) for the nodes.

[k: k+m) for the sources of edge\_index

[k+m : k+2m) for the targets of edge\_index

[k+2m : k+3m) for edge\_type.

I pre-load the matrix, and when needed during training I access the row *i* for node *i*, and extract the needed data.

## Step 3: Graph discrepancies

The node with index 16309 had no edges at all, something that is not supposed to be possible.

Therefore, I recreated the graph. Instead of 47K nodes, we have now 82K.

Data(edge\_index=[2, 82255], edge\_type=[82255], node\_types=[71882], num\_relations=[1], x=[71882, 300])

Organized in:

X\_definitions.shape=torch.Size([19008, 300])

X\_examples.shape=torch.Size([20191, 300])

X\_senses.shape=torch.Size([19008, 300])

X\_globals.shape=torch.Size([13675, 300])

with edges:

def\_edges\_se.\_\_len\_\_()=19008

exs\_edges\_se.\_\_len\_\_()=20191

syn\_edges.\_\_len\_\_()=21676

ant\_edges.\_\_len\_\_()=2372

Problem: the vocabulary of globals has only 13674 entries.

Time to execute the whole pipeline from CreateGraphInput, with a full reset.

Flag values:

do\_reset=***True***, compute\_single\_prototype=***True***, vocabulary\_from\_senselabeled=**False**

After re-executing the pipepline, the discrepancy is still there.

It is time to examine the source of the data in the graph.

X\_senses = initialize\_senses(X\_definitions, X\_examples, average\_or\_random=True)  
X\_globals = torch.Tensor(np.load(os.path.join(F.FOLDER\_INPUT, F.SPVs\_FASTTEXT\_FILE)))

F.SPVs\_FASTTEXT\_FILE < createGraphInput.exe(…) <

CE.compute\_single\_prototype\_embeddings(vocabulary\_df, spvs\_out\_fpath, method)

<< **for** idx\_word\_freq\_tpl **in** vocabulary\_df.itertuples(): …

X\_globals.shape=torch.Size([13675, 300])

+1 because we add <unk>. This correspondence is correct.

Remember that, when we build the graph in the DefineGraph module:

X = torch.cat([X\_senses, X\_globals, X\_definitions, X\_examples])

*# sense=[0,se) ; single prototype=[se,se+sp) ; definitions=[se+sp, se+sp+d) ; examples=[se+sp+d, e==num\_nodes)*

We proceed to build the graph. The number and types of nodes are the same that we observed previously.

Eventually, the attributes of the object are:

Data(edge\_index=[2, 82255], edge\_type=[82255], node\_types=[71882], num\_relations=[1], x=[71882, 300])

The shape of graph.x is torch.Size([71882, 300])

Now we retrieve the edges and node data we expected.

However, there is a new issue:

RuntimeError: index out of range: Tried to access index 48987 out of table with 31 rows. at: predicted\_globals, predicted\_senses = model(x, edge\_index, edge\_type)

{temp: modified batch\_size in MyRGCN.train(…)}

### SC edges from Globals to Senses

Problem:

Some of the nodes after n=19008 have 0 edges. Why?

We recall that the matrix X is built as:

X = torch.cat([X\_senses, X\_globals, X\_definitions, X\_examples])

And we have: 19008 senses, 13675 globals, 19008 definitions and 20191 examples

Reviewing the sc (SenseChildren) edges:

there are 19008 edges that go from globals to senses, it’s not empty.

However:

*while all senses have a SC edge, not all globals have a SC edge*.

We extracted the globals from WikiText-2.

We have a problem, because some nodes (the globals without sense specifications) are going to be floating around, disconnected from the rest of the graph.

This is also valid for the <unk> token.

Possible solutions:

1. Insert checks in the code, to allow for the processing of nodes with an empty list of edges
2. Add an “artificial” self-loop of type SC to all empty globals.

Given that the RGCN layer throws an error if it operates on a node without an edge, I opt for (2).

Alternatively it could be a new edge type SL (self-loop).