[Current Next Steps 3](#_Toc52655101)

[University side 3](#_Toc52655102)

[Ordbogen side 5](#_Toc52655103)

[Preliminary information (Datasets, graphs) 6](#_Toc52655104)

[SemCor – current 6](#_Toc52655105)

[SemCor stats (80-10-10, lowercased) 6](#_Toc52655106)

[Graph (min\_freq=2, with dummySense) 6](#_Toc52655107)

[LM\_TransformerXL 7](#_Toc52655108)

[Training.py, train() 7](#_Toc52655109)

[train.py -> get\_lm\_corpus() 7](#_Toc52655110)

[get\_lm\_corpus() -> Corpus.\_\_init\_\_() 8](#_Toc52655111)

[Mini-experiments and checks 10](#_Toc52655112)

[Simple GRUs 10](#_Toc52655113)

[With freezing 10](#_Toc52655114)

[SelectK 11](#_Toc52655115)

[With freezing 11](#_Toc52655116)

[Part 1 13](#_Toc52655117)

[WikiText-2 (standard LM) 13](#_Toc52655118)

[Experiment 13](#_Toc52655119)

[2 GRUs on SemCor 14](#_Toc52655120)

[Simple GRUs 14](#_Toc52655121)

[GNN Input Signal 15](#_Toc52655122)

[2GRUs + GAT200 15](#_Toc52655123)

[2GRUs based only on GAT, d=150x2 16](#_Toc52655124)

[SelectK 18](#_Toc52655125)

[Reinserting the SelectK method 18](#_Toc52655126)

[Checking code & output 19](#_Toc52655127)

[K=1 21](#_Toc52655128)

[K=10 22](#_Toc52655129)

[K=1, with freezing 23](#_Toc52655130)

[K=10, with freezing 24](#_Toc52655131)

[Time analysis of SelectK and modification 25](#_Toc52655132)

[Issue: SelectK10 with freezing, Invalid Index 27](#_Toc52655133)

[Reasoning on other methods 31](#_Toc52655134)

[Variants of Structured prediction 31](#_Toc52655135)

[Variants of General prediction 32](#_Toc52655136)

[Rank by context similarity 32](#_Toc52655137)

[Design & implementation 32](#_Toc52655138)

[Checks, Mini-experiments and analysis of predictions 34](#_Toc52655139)

[Checks 34](#_Toc52655140)

[Part 2 – Same experiments as before, now reading SemCor with the correct lowercasing 36](#_Toc52655141)

[2 GRUs on SemCor 36](#_Toc52655142)

[Simple GRUs 36](#_Toc52655143)

[GNN Input Signal 36](#_Toc52655144)

[2GRUs + GAT300 36](#_Toc52655145)

[SelectK 36](#_Toc52655146)

[K=1 36](#_Toc52655147)

[K=10 36](#_Toc52655148)

[K=1, with freezing 36](#_Toc52655149)

[K=10, with freezing 36](#_Toc52655150)

[Context Similarity – continuation 37](#_Toc52655151)

[K=1, C=10 37](#_Toc52655152)

[K=10, C=10 37](#_Toc52655153)

# Current Next Steps

## University side

Step 1: re-run the pipeline for SemCor, and try to replicate exactly the old result with 185 Valid-PPL. This would demonstrate that the architecture still works as intended.

Status: done,

the old result was so good because the SemCor corpus had been **lowercased**, and 2/3/n digit numbers had been turned into <num>.

we got 219.4 because we used an LSTM where a GRU works better. See [Experiment](#_Experiment).

This validates the architecture (i.e. it’s not bugged, as we also see from WikiText-2). However, when we try experiments on SemCor’s 650K training tokens, we get very high PPL values, like 445 Valid-PPL at [Experiment C](#_Experiment_C).

We recall that Penn-Treebank has 929K training tokens with heavy pre-processing, namely “…lower-cased, numbers replaced with N, punctuation removed. The vocabulary is the most frequent 10k words”. WikiText-2 has ~2 million training tokens.

Since the RNN architecture works as intended and is not bugged, we try to reach better performance by including *more* sense-labeled data, see Step 3.

Step 2: try a 1024>1024>512 architecture on WT-2, with no dropout, to try to bring the Valid-PPL well under 200.

status:

done. The first experiment on WT-2 is the one with a 1150>1150>400 LSTM that (mistakenly) applies a standard dropout of p=0.1. See [Experiment A](#_Experiment_A), it reaches 200.5 Validation PPL.

Moreover, [Experiment B](#_Experiment_B), using a 1024>1024>512 GRU with no dropout, reaches **186.4** Valid-PPL

Step 3: gather enough sense-labeled data to obtain a corpus of size comparable to WT-2, to try to obtain a similar Valid-PPL.

I need to review the other Sense-Labeled Corpuses. The ones eligible for inclusion are those labelled with **WordNet 3.0** senses, i.e.:

SemCor (228k annotated / 778k words), OMSTI (1 million / 35 millions), MASC (115K / 596K). Ontonotes would be too, but they are saved in a complex acess-DB format. + SemEval 2013 (8.3K words), SemEval 2015(2.6K)

The current expanded SLC contains SemCor + MASC + the first 300 MB of text of OMSTI (for the sake of speed in the current experiments, we did not include it in its entirety).

training tokens: 4,875,767. Validation tokens: 605,366

The first tentative [Exp A](#_Exp_A) reaches 158 Valid-PPL, but there is a chance I am reading one of the subcorpuses twice so it may be bugged. It must be checked and re-done.

Step 4: continue adjusting the code and running mini-experiments, to prepare the freezing mechanism.

Using the flags predict\_senses=**True** in training\_setup() and with\_freezing=**True** in training\_loop(), we apply the current version of the freezing mechanism:

* we set p.requires\_grad=**False** on all the parameters of the globals’ GRU and softmax
* We recreate the optimizer, to eliminate any remaining momentum
* We activate the senses’ GRU with model\_forParameters.predict\_senses = **True**.
* We set after\_freezing\_flag = **True**, so the loss will be computed as   
  loss = loss\_sense instead of loss = loss\_global + loss\_sense

Despite all of this, the globals’ loss still changes in the epochs after the freezing point.

ToDo:

build a small toy-model, that takes in the same input and labels, and examine the freezing mechanism.

Step 5: Finish the freezing mechanism – a very minor oscillation due to the embeddings is still ok

status: done. The freezing mechanism now works, and without any oscillation, as we can see in [Full RNN (RNNs, embeddings, GATs)](#_Full_RNN_(RNNs,).

Step 6: Use lowercased SemCor, for the sake of brevity and development. Execute Senses-with-RNN experiments, with and without the freezing mechanism

Step 7a:

Set up and run, indicatively, the following experiments:

* On SemCor alone, and on pretrained on WT-2+SemCor
* With and without freezing
* With and without GAT-global-nodestate
* Standard GRU for globals and senses
* SelectK(1,5,10,50), with and without freezing
* Attention mechanism for senses

This would mean a total of: 2x2x2x3 = 24 experiments

(alternatives that work better get priority when it comes to computational resources)

Step 7b:

Start the write-up of the paper. Proceed with the structure of the experiments and the description of the methods, and also abstract + background

Step 8a:

* Write, check and complete the code for the remaining methods

Step 9a:

* I could maybe eliminate the dummySenses of all the words where the lemmatized form has a sense: plurals, verbs, et cetera.  
  note: Lemmatization is useful in a number of cases (‘sports‘, ‘blacks’, ‘was’ etc.), however, ‘its’ gets lemmatized into ‘it’. In some cases, we may not want lemmatization. -> check, search, and ask how to handle lemmatization.
* Sometimes in the vocabulary we have the non-lemmatized form (‘eyelids’) but we do not have the lemmatized form (‘eyelid’).   
  As a consequence, when we retrieve the WordNet data we should do it not only for the initial vocabulary, but also for the vocabulary that includes all of its lemmatized forms.

## Ordbogen side

Information 1:

The Desktop version is currently put aside and not developed. The version currently developed is the web browser add-on version.

The GrammaTip team is also working on a text editor.

Information 2:

You can (ideally) reliably convert to & fro Pytorch and Tensorflow (see instrument: Neural Network Exchange)

Information 3:

Instrument for deployment that could be used: MLFlow.

Task 4:

Look at the transformer-LM code. Can we train it as expected? Can we operate on both Danish and English?

# Preliminary information (Datasets, graphs)

## SemCor – current

### SemCor stats (80-10-10, lowercased)

Note: currently I am also including punctuation in the vocabulary from SLC.

Training tokens: 646,038

Validation tokens: 80,760

|Vocabulary|= 22,782

### Graph (min\_freq=2, with dummySense)

INFO : Constructing X, matrix of node features

INFO : X\_senses.shape=torch.Size([**41206**, 200]) # senses with data: 30445

INFO : X\_globals.shape=torch.Size([**22782**, 200])

INFO : X\_definitions.shape=torch.Size([30446, 200])

INFO : X\_examples.shape=torch.Size([27921, 200])

INFO : E\_embeddings.shape=torch.Size([22782, 200])

Graph ranges: [0, senses, 41206, globals, 63988, defs, 94434, examples, 122355]

INFO : Defining the edges: def, exs

INFO : def\_edges\_se.\_\_len\_\_()=30446

INFO : exs\_edges\_se.\_\_len\_\_()=27921

INFO : Defining the edges: sc

INFO : Reading the sense-labeled corpus, to create the connections between globals and the senses that belong to other words.

INFO : Generator over subcorpus at TextCorpuses/My Sense-Labeled Corpus/Training/semcor.xml

INFO : sc\_edges\_with\_external.\_\_len\_\_()=7044

INFO : sc\_edges.\_\_len\_\_()=48250

INFO : get\_edges\_selfloops>max\_sense=41205

INFO : []

INFO : len(edges\_ls)==0

INFO : sc\_edges\_with\_selfloops.\_\_len\_\_()= 48250

INFO : Defining the edges: syn, ant

INFO : syn\_edges.\_\_len\_\_()= 19804 (fewer, it was 24272…(?)(is this relevant or only a modification from an older version)

INFO : ant\_edges.\_\_len\_\_()=4070

INFO : Pre-computing and saving graphArea matrix, with area\_size=32

~~D~~ata(edge\_index=[2, **130491**], edge\_type=[130491], node\_types=[122355], num\_relations=[1], x=[**122355**, 100]) ; file size =~151MB

# LM\_TransformerXL

## Training.py, train()

corpus\_preparator = CorpusPreparator(overwrite=**False**)

corpus\_preparator.prepare\_corpus(wiki\_dump\_file\_danish, output\_dir=danish\_data\_dir)

*"""  
 Create\_text\_from\_wiki\_dump(path\_to\_wiki\_dump, destination\_folder\_name), turns a .bz2 compressed archive  
 into multiple folders (e.g. AA, AB, etc.) with files wiki\_00, wiki\_01, etc.. Each files has multiple <doc>s  
"""*

Then,

os.chdir(pytorch\_dir)  
*# the parameters are modified from: the default ones + run\_wt103\_base.sh*bash\_command = [**'python3'**,  
 **'train.py'**, …

### train.py -> get\_lm\_corpus()

args = parse\_arguments()

…

corpus = get\_lm\_corpus(args.data, args.data\_set)

In data\_utils, **get\_lm\_corpus():**

get\_lm\_corpus(data\_dir: str, data\_set)

*"""  
Factory method for creating a Corpus instance. Once a my\_corpus has been created, it is saved in 'cache.pt' in 'data\_dir'  
Next time this method is called, the my\_corpus is not recreated but loaded.***:param** *data\_dir: Where to look for cache.pt***:param** *data\_set: What type of data\_set is to be loaded.***:return***: An instance of Corpus  
"""*

**if** os.path.exists(corpus\_cache\_file):

corpus = torch.load(corpus\_cache\_file)

**else**:

**if** data\_set **in** [**'wt103'**, **'wt2'**]:  
 corpus = Corpus(data\_dir, data\_set, special=[**'<eos>'**], lower\_case=**False**)

torch.save(corpus, corpus\_cache\_file)

**return** corpus

### get\_lm\_corpus() -> Corpus.\_\_init\_\_()

*"""  
Constructor for Corpus. Sets up the Vocabulary of the corpus***:param** *data\_dir: The folder in which to look for train.txt, test.txt and valid.txt***:param** *data\_set: What type of data\_set (corpus) is to be loaded.  
"""*

self.vocab = Vocabulary(special, min\_freq, max\_size, lower\_case, delimiter, vocab\_file)

**Vocabulary.\_\_init\_\_()**

self.special = special  
self.min\_freq = min\_freq  
self.max\_size = max\_size  
self.lower\_case = lower\_case  
self.delimiter = delimiter  
self.vocab\_file = vocab\_file  
self.idx2sym = []  
self.sym2idx = OrderedDict()  
self.counter = Counter()

**if** self.data\_set **in** [**'ptb'**, **'wt2'**, **'enwik8'**, **'text8'**]:  
 self.vocab.count\_file(os.path.join(data\_dir, **'train.txt'**))  
 self.vocab.count\_file(os.path.join(data\_dir, **'valid.txt'**))  
 self.vocab.count\_file(os.path.join(data\_dir, **'test.txt'**))  
**elif** self.data\_set == **'wt103'**:  
 self.vocab.count\_file(os.path.join(data\_dir, **'train.txt'**))

**def count\_file(self, path, verbose=True, add\_eos=False):**

**with** open(path, **'r'**, encoding=**'utf-8'**) **as** f:  
 **for** idx, line **in** enumerate(f):

symbols = self.tokenize(line, add\_eos=add\_eos)  
self.counter.update(symbols)  
sents.append(symbol)

self.vocab.build\_vocab()

**def build\_vocab(self):**

(n: generally we don’t have a vocab\_file if we are not on 1BW)

**for** sym **in** self.special:  
 self.add\_special(sym)  
  
**for** sym, cnt **in** self.counter.most\_common(self.max\_size):  
 **if** cnt < self.min\_freq: **break** self.\_add\_symbol(sym)

(where adding a symbol means:

self.idx2sym.append(sym)  
self.sym2idx[sym] = len(self.idx2sym) - 1)

**if** self.data\_set **in** [**'ptb'**, **'wt2'**, **'wt103'**]:  
 self.train =self.vocab.encode\_file(train\_file,ordered=**True**, verbose=verbose)  
 self.valid =self.vocab.encode\_file(valid\_file, ordered=**True**, verbose=verbose)  
 self.test =self.vocab.encode\_file(test\_file, ordered=**True**, verbose=verbose)

**def encode\_file(self, path, ordered=False, verbose=False,  
 add\_eos=True, add\_double\_eos=False):**

**with** open(path, **'r'**, encoding=**'utf-8'**) **as** f:  
 **for** idx, line **in** enumerate(f):  
 **if** verbose **and** idx > 0 **and** idx % 500000 == 0:  
 print(**' line {}'**.format(idx))  
 symbols = self.tokenize(line, add\_eos=add\_eos, add\_double\_eos=add\_double\_eos)  
 encoded.append(self.convert\_to\_tensor(symbols))

**if** self.dataset == **"wt103"**:  
 self.cutoffs = [0, 20000, 40000, 200000] + [len(self.vocab)]

Again on get\_lm\_corpus(data\_dir: str, data\_set)…

# Mini-experiments and checks

This section is for the mini-experiment to check the functionality of the various architectures

## Simple GRUs

### With freezing

INFO : RNN(

(dropout): Dropout(p=0, inplace=False)

(main\_rnn\_ls): ModuleList(

(0): GRU(300, 1024)

(1): GRU(1024, 1024)

(2): GRU(1024, 512)

)

(senses\_rnn\_ls): ModuleList(

(0): GRU(300, 1024)

(1): GRU(1024, 1024)

(2): GRU(1024, 512)

)

(linear2global): Linear(in\_features=512, out\_features=22782, bias=True)

(linear2senses): Linear(in\_features=512, out\_features=41206, bias=True)

)

INFO : Parameters:

INFO : ('E', torch.Size([22782, 300]), torch.float32, True)

('memory\_hn', torch.Size([3, 2, 1024]), torch.float32, False)

('memory\_hn\_senses', torch.Size([3, 2, 1024]), torch.float32, False)

('main\_rnn\_ls.0.weight\_ih\_l0', torch.Size([3072, 300]), torch.float32, True)

…

('linear2global.bias', torch.Size([22782]), torch.float32, True)

('linear2senses.weight', torch.Size([41206, 512]), torch.float32, True)

('linear2senses.bias', torch.Size([41206]), torch.float32, True)

INFO : Number of trainable parameters=65.13M

|  |  |  |  |
| --- | --- | --- | --- |
| *Epoch* | *Training* | | |
|  | Globals | Senses | Multi-senses |
| 1 | 22831.58 | 1.0 | 1.0 |
| … |  |  |  |
| 5 | 8954.96 | 1.0 | 1.0 |
| 6 | 3708.13 | 40549.28 | 41154.67 |
| 7 | **3702.04** | 37411.89 | 39028.87 |
| 8 | **3702.04** | 33370.31 | 36221.34 |

## SelectK

### With freezing

INFO : Model:

INFO : SelectK(

(dropout): Dropout(p=0, inplace=False)

(main\_rnn\_ls): ModuleList(

(0): GRU(300, 1024)

(1): GRU(1024, 1024)

(2): GRU(1024, 512)

)

(senses\_rnn\_ls): ModuleList(

(0): GRU(300, 1024)

(1): GRU(1024, 1024)

(2): GRU(1024, 512)

)

(linear2global): Linear(in\_features=512, out\_features=22782, bias=True)

(linear2senses): Linear(in\_features=512, out\_features=41206, bias=True)

)

INFO : Number of trainable parameters=65.13M

model.K=1

|  |  |  |  |
| --- | --- | --- | --- |
| *Epoch* | *Training* | | |
|  | Globals | Senses | Multi-senses |
| 1 | 22831.58 | 1.0 | 1.0 |
| … |  |  |  |
| 5 | 8954.96 | 1.0 | 1.0 |
| 6 | 3708.13 | 39810810.26 | 100000092.9 |
| 7 | **3702.04** | 39810810.26 | 100000092.9 |
| 8 | **3702.04** | 39810810.26 | 100000092.9 |

# Part 1

## WikiText-2 (standard LM)

Standard Language Model task, i.e. globals only.

Objective: train a model on WikiText-2, that will be stored and used as a pretrained base to operate on SemCor.

### Experiment

|  |  |  |
| --- | --- | --- |
| **Architecture** | **Input Signals** | **Hyperparameters** |
| Globals: **Main GRU** with 3 layers (1024>1024>512) | 1) The word embedding of the current global (d=300) | batch\_size=40  TBPTT length=35 |
| Senses:nothing. We create a GRU, but we do not use it. | grapharea=32, hops=1  learning rate=0.00005 |
| *Later on we may also add other mechanisms and parameters to the model object.* |  |

Number of trainable parameters=78.65M. Expecting a Valid-PPL in the range 185-190

|  |  |  |
| --- | --- | --- |
| *Epoch* | *Training PPL, globals* | *Validation PPL, globals* |
| 1 | 1435.47 | 704.11 |
| 2 | 676.57 | 445.01 |
| 3 | 481.45 | 358.82 |
| 4 | 391.48 | 313.99 |
| 5 | 336.84 | 285.54 |
| 6 | 298.43 | 265.36 |
| 7 | 269.1 | 250.15 |
| 8 | 245.64 | 238.23 |
| 9 | 226.35 | 228.74 |
| 10 | 210.11 | 221.11 |
| 11 | 196.2 | 214.92 |
| 12 | 184.1 | 209.79 |
| 13 | 173.43 | 205.52 |
| 14 | 163.9 | 201.92 |
| 15 | 155.32 | 198.86 |
| 16 | 147.54 | 196.2 |
| 17 | 140.42 | 193.93 |
| 18 | 133.87 | 192.02 |
| 19 | 127.83 | 190.42 |
| 20 | 122.22 | 189.08 |
| 21 | 116.99 | 188.01 |
| 22 | 112.11 | 187.16 |
| 23 | 107.52 | 186.52 |
| 24 | 103.21 | 186.07 |
| 25 | 99.14 | 185.74 |
| 26 | 95.3 | 185.57 |
| 27 | 91.65 | **185.53**  'correct\_g': 47667,  'top\_k\_g': 104562, 'tot\_g': 212800 |
| 28 | 88.2 | 185.68 |

We saved the model @epoch 28. We store it, to be used later as a pretrained model.

What could be the next steps?

* Using 2 simple GRUs on SemCor, providing a baseline for the sense prediction
* Adding the GAT nodestate of the global should provide additional information, and ideally improve both the standard LM and the senses’ prediction.  
  From this, 2 experiments would follow: Simple GRUs + GAT input, with and without freezing.
* Using the pretrained WT-2 model? This should improve the performance of the standard LM task on globals…  
  However, it is likely that using the pretrained WT-2 is more significant on methods that actually rely on the globals’ performance (SelectK).
* Then, moving on to the other mechanisms

## 2 GRUs on SemCor

### Simple GRUs

|  |  |  |
| --- | --- | --- |
| **Architecture** | **Input Signals** | **Hyperparameters** |
| Globals: **Main GRU** with 3 layers (1024>1024>512) | 1) The word embedding of the current global (d=300) | batch\_size=32  TBPTT length=35 |
| Senses: **Senses’** **GRU** with 3 layers, as above. |  | grapharea=32, hops=1  learning rate=0.00005 |

INFO : Model:

INFO : DataParallel(

(module): RNN(

(dropout): Dropout(p=0, inplace=False)

(main\_rnn\_ls): ModuleList(

(0): GRU(300, 1024)

(1): GRU(1024, 1024)

(2): GRU(1024, 512)

)

(senses\_rnn\_ls): ModuleList(

(0): GRU(300, 1024)

(1): GRU(1024, 1024)

(2): GRU(1024, 512)

)

(linear2global): Linear(in\_features=512, out\_features=24122, bias=True)

(linear2senses): Linear(in\_features=512, out\_features=41534, bias=True)

))

Number of trainable parameters=65.13M,

where embeddings=22782x300=6.84M, softmax=22782x513+41206x513=32.83M , core=25.46M (i.e. 12.73M x 2)

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| *Epoch* | *Training* | | | *Validation* | | |
|  | Globals | Senses | Multi-senses | Globals | Senses | Multi-senses |
| 1 | 624.45 | 943.69 | 11541.31 | 507.95 | 646.48 | 11236.47 |
| 2 | 358.54 | 593.88 | 8199.44 | 403.42 | **632.07**  **'correct\_all\_s': 11761, 'top\_k\_all\_s': 32999, 'tot\_all\_s': 80640** | 10391.08 |
| 3 | 291.33 | 586.87 | 8037.58 | 327.9 | 636.47 | 10305.85  'correct\_multi\_s': 0, 'top\_k\_multi\_s': 1897, 'tot\_multi\_s': 29575 |
| … |  |  |  |  |  |  |
| 17 | 114.24 | 547.9 | 7021.49 | **224.22** | 697.34 | 11324.48 |
| 18 | 110.53 | 529.92 | 6545.85 | 224.28 | 739.26 | 14473.89 |
| 19 | 107.04 | 517.93 | 6322.95 | 224.55 | 719.44 | 12236.07 |

(top\_k==top\_10)

‘correct\_all\_s’: 11761, always. 'top\_k\_all\_s': 32999 @ epoch 2. 'tot\_all\_s': 80640

'correct\_multi\_s': 0, always. 'top\_k\_multi\_s': 1897 @epoch 3 'tot\_multi\_s': 29575

## GNN Input Signal

### 2GRUs + GAT200

|  |  |  |
| --- | --- | --- |
| **Architecture** | **Input Signals** | **Hyperparameters** |
| Globals: **Main GRU** with 3 layers (1024>1024>512) | 1) The word embedding of the current global *wi* (d=300)  2) The state vector of the global node *I*, obtained applying a GNN (GAT) on the dictionary graph. (d=**200**). | batch\_size=**32**  TBPTT length=35 |
| Senses: **GRU** with 3 layers, as above. | grapharea=32, hops=1  learning rate=0.00005 |
| **GNN**: GAT, with input\_d=200 and output=100x2 |

INFO : DataParallel(

(module): RNN(

(dropout): Dropout(p=0, inplace=False)

(gat\_globals): GATConv(200, 100, heads=2)

(main\_rnn\_ls): ModuleList(

(0): GRU(500, 1024) (1): GRU(1024, 1024) (2): GRU(1024, 512) )

(senses\_rnn\_ls): ModuleList(

(0): GRU(500, 1024) (1): GRU(1024, 1024) (2): GRU(1024, 512) )

(linear2global): Linear(in\_features=512, out\_features=22782, bias=True)

(linear2senses): Linear(in\_features=512, out\_features=41206, bias=True) ))

N. of parameters=90.87M

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| *Epoch* | *Training* | | | *Validation* | | |
|  | Globals | Senses | Multi-senses | Globals | Senses | Multi-senses |
| 1 | 598.7 | 927.2 | 11685.38 | 428.84 | 645.24 | 11085.52 |
| 2 | 323.63 | 593.62 | 8195.95 | 355.01 | **630.39** | 10221.31 |
| 3 | 262.49 | 587.24 | 8052.54 | 309.42 | 636.85 | 10305.34 |
| … |  |  |  |  |  |  |
| 17 | 112.68 | 559.39 | 7334.36 | **223.8** | 708.85 | 11857.94 |
| 18 | 109.13 | 545.7 | 6959.92 | 223.89 | 711.23 | 11795.71 |
| 19 | 105.78 | 533.85 | 6683.94 | 224.17 | 716.01 | 12018.15 |

**Ongoing comparison table**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Method | correct\_all\_s ( / 80640) | top\_k\_all\_s | correct\_multi\_s ( / 29575) | top\_k\_multi\_s | Senses’ PPL |
| Simple GRUs | 11761 @ | 32999 @ epoch 2 | 0 | 1897 @ epoch 3 | 632.07 |
| **(GRUs, concat input GAT 200)** | **11761 always** | **32989  @ epoch 13** | **0 always** | **1826 @ epoch 13** | **630.39** |

### 2GRUs *based only on* GAT, d=150x2

|  |  |  |
| --- | --- | --- |
| **Architecture** | **Input Signals** | **Hyperparameters** |
| Globals: **Main GRU** with 3 layers (1024>1024>512) | 1) The state vector of the global node *I*, obtained applying a GNN (GAT) on the dictionary graph. (d=150). | batch\_size=**32**  TBPTT length=35 |
| Senses: **GRU** with 3 layers, as above. | grapharea=32, hops=1  learning rate=0.00005 |
| **GNN**: GAT, input\_d=150, output=150x2 |

Parameters=77.52M

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| *Epoch* | *Training* | | | *Validation* | | |
|  | Globals | Senses | Multi-senses | Globals | Senses | Multi-senses |
| 1 |  |  |  |  |  |  |
| 2 |  |  |  |  |  |  |
| 3 |  |  |  |  |  |  |
| … |  |  |  |  |  |  |
| 10 |  |  |  |  |  |  |
| … |  |  |  |  |  |  |
| 15 |  |  |  |  |  |  |
| … |  |  |  |  |  |  |
| 18 |  |  |  |  |  |  |
| … |  |  |  |  |  |  |
| 22 |  |  |  |  |  |  |
| 23 |  |  |  |  |  |  |

# SelectK

## Reinserting the SelectK method

We paste the latest version of the GRU in the Senses.py module, in the class SelectK; review it, and examine the insertion of the previously developed code for SelectK.

In \_\_init\_\_():

* init\_model\_parameters(self, data, grapharea\_size, grapharea\_matrix,…
* self.E = Parameter(embeddings\_matrix.clone().detach(), requires\_grad=**True**)
* **if** include\_globalnode\_input:  
   self.X = Parameter(data.x.clone().detach(), requires\_grad=**True**)
* *Utilities* (from the RNN, unchanged. Focusing on main elements and changes)

In the forward():

* *# -------------------- Input --------------------  
  # Input signal n.1: the embedding of the current (global) word*t\_word\_embeddings = self.E.index\_select(dim=0, index=t\_current\_globals\_indices)
* *# Input signal n.2: the node-state of the current global word - now with graph batching***if** self.include\_globalnode\_input:  
   t\_g\_nodestates = run\_graphnet(t\_input\_lts, batch\_elems\_at\_t,t\_globals\_indices\_ls, CURRENT\_DEVICE, self)  
   currentglobals\_nodestates\_ls.append(t\_g\_nodestates)
* *# ------------------- Senses -------------------  
  # line 1: GRU for senses + linear FF-NN to logits.***if** self.predict\_senses:  
   task\_2\_out = rnn\_loop(batch\_input\_signals, model=self)  
   task2\_out = task\_2\_out.reshape(distributed\_batch\_size \* seq\_len, task\_2\_out.shape[2])  
    
   logits\_sense = self.linear2senses(task2\_out)
* *# line 2: select senses of the k most likely globals*k\_globals\_indices = logits\_global.sort(descending=**True**).indices[:, 0:self.k]
* senses\_softmax = torch.ones((distributed\_batch\_size \* seq\_len, self.last\_idx\_senses)).to(CURRENT\_DEVICE)
* senses\_softmax = 10 \*\* (-8) \* senses\_softmax *# base probability value for non-selected senses*
* sample\_k\_indices\_lls\_relative = k\_globals\_indices.tolist()
* **for** s **in** range(distributed\_batch\_size \* seq\_len):
* k\_globals\_vocab\_indices = sample\_k\_indices\_in\_vocab\_lls[s]
* k\_globals\_words > k\_globals\_lemmatized > lemmatized\_indices
* **if** sense\_neighbours\_t.shape[0] == 0: *# no senses found, even lemmatizing. Ignore current entry* senses\_softmax[s] = torch.tensor(1 / self.last\_idx\_senses).to(CURRENT\_DEVICE)  
   **continue**
* *# standard procedure: get the logits of the senses of the most likely globals,  
  # apply a softmax only over them, and then assign an epsilon probability to the other senses*
* sample\_logits\_senses = logits\_sense.index\_select(dim=0, index=self.select\_first\_indices[**s**].to(torch.int64)).squeeze()
* logits\_selected\_senses = sample\_logits\_senses.index\_select(dim=0, index=sense\_neighbours\_t)
* softmax\_selected\_senses = tfunc.softmax(input=logits\_selected\_senses, dim=0)
* **for** i **in** range(len(sense\_neighbours\_t)):  
   i\_senseneighbours\_mask[s,sense\_neighbours\_t[i]]=**True**
* quantity\_to\_subtract\_from\_selected = epsilon \* (self.last\_idx\_senses - len(sense\_neighbours\_t))  
   softmax\_selected\_senses = subtract\_probability\_mass\_from\_selected(softmax\_selected\_senses, quantity\_to\_subtract\_from\_selected)  
   senses\_softmax[s].masked\_scatter\_(mask=i\_senseneighbours\_mask[s].data.clone(), source=softmax\_selected\_senses)
* predictions\_senses = torch.log(senses\_softmax)

### Checking code & output

k\_globals\_indices = logits\_global.sort(descending=**True**).indices[:, 0:self.K]

with batch size =2, seq\_len=5 (so 10 samples), and self.K=1, we get:

tensor([[ 846], dallas\_county

[ 846],

[ 846],

[ 846],

[ 846],

[6356], desert

[6356],

[6356],

[ 846],

[ 846]])

The random initial weights of the NN determine the most likely globals.

Turned into a LLS by

sample\_k\_indices\_in\_vocab\_lls = k\_globals\_indices.tolist()

Then, a for cycle for each element in the sample:

**for** s **in** range(distributed\_batch\_size \* seq\_len):

lemmatized\_indices\_in\_X # [42052], i.e. 42052-41206=846, ok

sense\_neighbours\_t=tensor([35377]), i.e. dallas\_county.dummySense.01

sample\_logits\_senses= logits\_sense.index\_select(dim=0, index=self.select\_first\_indices[s].to(torch.int64)).squeeze()

tensor of torch.Size([41206])

logits\_selected\_senses = sample\_logits\_senses.index\_select(dim=0, index=sense\_neighbours\_t), tensor([**0.0071**], grad\_fn=<IndexSelectBackward>)

softmax\_selected\_senses = tfunc.softmax(input=logits\_selected\_senses, dim=0), tensor([**1.**], grad\_fn=<SoftmaxBackward>)

It seems ok. Let us check with a mini-experiment.

INFO : Hyperparameters: \_batchPerSeqlen20\_area32\_lr0.0001\_epochs400

|  |  |  |  |
| --- | --- | --- | --- |
| *Epoch* | *Validation == Training* | | |
|  | Globals | Senses | Multi-senses |
| 1 | 22385.71 | 54117076.14 | 100000216.32 |
| 2 | 21683.67 | 39810810.26 | 100000216.32 |
| 10 | 3425.86 | 39810797.6 | 100000216.32 |
| 100 | 22.61 | 21544393.04 | 100000216.32 |
| 200 | 11.39 | 1000001.78 | 100000216.32 |
| 300 | 15.07 | 8576976.16 | 100000216.32 |
| 350 | 3.14 | 29.46 | 84.33 |
| 375 | 2.51 | 11.96 | 9.69 |
| 400 | 2.07 'correct\_g': 60,  'top\_k\_g': 60,  'tot\_g': 60 | 8.59 'correct\_all\_s': 53,  'top\_k\_all\_s': 53,  'tot\_all\_s': 60, | 4.38  'correct\_multi\_s': 23, 'top\_k\_multi\_s': 23, 'tot\_multi\_s': 25 |

**Analysis of predictions**

|  |  |
| --- | --- |
| Label: the next global is: <unk>(from 22213)  INFO : Label: the next sense is: group.n.01(from 12610) | INFO : The top- 2 predicted globals are:  INFO : Word: <unk> ; p=75.8%  INFO : Word: in ; p=5.27%  INFO : The top- 2 predicted senses are:  INFO : Sense: <unk>.dummySense.01 ; p = 100.0% |
| Label: the next global is: said(from 1)  INFO : Label: the next sense is: state.v.01(from 26023) | INFO : The top- 2 predicted globals are:  INFO : Word: said ; p=77.17%  INFO : Word: charge ; p=5.01%  INFO : The top- 2 predicted senses are:  INFO : Sense: state.v.01 ; p = 99.72% |
| Label: the next global is: investigation(from 4)  INFO : Label: the next sense is: probe.n.01(from 21009) | INFO : The top- 2 predicted globals are:  INFO : Word: investigation ; p=19.57%  INFO : Word: further ; p=7.53%  INFO : The top- 2 predicted senses are:  INFO : Sense: probe.n.01 ; p = 99.74% |

### K=1

|  |  |  |
| --- | --- | --- |
| **Architecture** | **Input Signals** | **Hyperparameters** |
| Globals’ prediction: **Main GRU** with 3 layers (1024>1024>512), > FF-NN to logits | 1) The word embedding of the current global (d=300) | batch\_size=32  TBPTT length=35 |
|  | grapharea=32, hops=2 |
| **Senses GRU > FF-NN** as above, we apply the softmax on the logits of the senses of the most likely ***k=1*** global | learning rate=5\*10^(-5) |

Iteration time=2.42032

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| *Epoch* | *Training* | | | *Validation* | | |
|  | Globals | Senses | Multi-senses | Globals | Senses | Multi-senses |
| 1 | 624.38 | 9737949.89 | 100000225.42 | 505.97 | 6811371.12 | 100000213.73 |
| 2 | 355.62 | 7917699.78 | 100000225.42 | 401.09 | 3939463.62 | 100000213.73 |
| 3 | 291.1 | 3501053.57 | 99993328.86 | 329.69 | 2344336.18 | 98872662.2 |
| … |  |  |  |  |  |  |
| 17 | 114.35 | 1011201.83 | 54664394.58 | **224.6** | 1205637.81 | 64870456.66 |
| 18 | 110.68 | 978485.48 | 53698253.4 | 224.63 | 1201188.8 | 64652532.94 |
| 19 | 107.22 | 949394.0 | 52872507.1 | 224.83 | 1197689.61  **'correct\_all\_s': 19020, 'top\_k\_all\_s': 19483, 'tot\_all\_s': 80640** | 63979598.63  **'correct\_multi\_s': 391, 'top\_k\_multi\_s': 776, 'tot\_multi\_s': 29575** |

Globals’ valid PPL=224.6. Senses’ valid PPL = not significant, as a consequence of our manipulation of the softmax.

**Ongoing comparison table**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Method | correct\_all\_s ( / 80640) | top\_k\_all\_s | correct\_multi\_s ( / 29575) | top\_k\_multi\_s | Senses’ PPL |
| Simple GRUs | 11761 always | 32999 @ epoch 2 | 0 | 1897 @ epoch 3 | 632.07 |
| (GRUs, concat input GAT 200) | 11761 always | 32989  @ epoch 13 | 0 always | 1826 @ epoch 13 | 630.39 |
| **SelectK1** | **19020 @ epoch 19** | **19483 @ epoch 19** | **391 @ epoch 19** | **776 @ epoch 19** | not significant |

We have potentially better prediction capabilities, but they are hampered by the fact that if the global was wrong in the first place, there is no way to retrieve the correct sense.

We check what happens if:

1. we choose among the first k=10 globals, instead of k=1
2. we use the freezing mechanism: first optimize the globals’ prediction as much as possible, freeze the globals, and then optimize the senses
3. we use the model pre-trained on WikiText-2

### K=10

|  |  |  |
| --- | --- | --- |
| **Architecture** | **Input Signals** | **Hyperparameters** |
| Globals’ prediction: **Main GRU** with 3 layers (1024>1024>512), > FF-NN to logits | 1) The word embedding of the current global (d=300) | batch\_size=32  TBPTT length=35 |
|  | grapharea=32, hops=2 |
| **Senses GRU > FF-NN** as above, we apply the softmax on the logits of the senses of the most likely ***k=10*** globals | learning rate=5\*10^(-5) |

Iteration time=**10.9849**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| *Epoch* | *Training* | | | *Validation* | | |
|  | Globals | Senses | Multi-senses | Globals | Senses | Multi-senses |
| 1 | 624.41 | 232776.94 | 50544848.48 | 506.54 | 125516.81 | 56235815.16 |
| 2 | 356.82 | 174996.61 | 38769545.17 | 402.31 | 143309.47 | 43468423.83 |
| 3 | 291.37 | 159397.18 | 30476220.94 | 328.54 | 106346.9 | 30927979.83 |
| … |  |  |  |  |  |  |
| 17 | 114.0 | 52349.62 | 11347551.11 | **223.87** | 62021.0 | 16886190.92 |
| 18 | 110.28 | 50948.26 | 10947177.92 | 223.88 | 61762.45 | 16818079.97 |
| 19 | 106.78 | 49513.74 | 10583746.73 | 224.06 | 61606.69 | 16575703.33 |

**Ongoing comparison table**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Method | correct\_all\_s ( / 80640) | top\_k\_all\_s | correct\_multi\_s ( / 29575) | top\_k\_multi\_s | Senses’ PPL |
| Simple GRUs | 11761 @ | 32999 @ epoch 2 | 0 | 1897 @ epoch 3 | 632.07 |
| (GRUs, concat input GAT 200) | 11761 always | 32989  @ epoch 13 | 0 always | 1826 @ epoch 13 | 630.39 |
| SelectK1 | 19020 @ epoch 19 | 19483 @ epoch 19 | 391 @ epoch 19 | 776 @ epoch 19 | not significant |
| **SelectK10** | **11761 @ epoch 1** | **32807 @epoch 1** | **118 @epoch 17** | **1139  @ epoch 1,  930  @ epoch 19** | n.s. |

What happens if we use the freezing mechanism? We optimize the Senses’ GRU in the second phase, when the globals’ GRU and the word embeddings have been fixed after reaching the lowest Globals’ Valid-PPL possible

### K=1, with freezing

|  |  |  |
| --- | --- | --- |
| **Architecture** | **Input Signals** | **Hyperparameters** |
| Globals: **Main GRU** with 3 layers (1024>1024>512) | 1) The word embedding of the current global (d=300) | batch\_size=32  TBPTT length=35 |
| **Senses GRU > FF-NN** as above, we apply the softmax on the logits of the senses of the most likely ***k=1*** global | grapharea=32, hops=1 |
| **Freezing:** in 1st part, update only globals. Then freeze global’s network + embeddings, and update senses | learning rate=5\*10^(-5) |
| dropout=none |

INFO : Number of trainable parameters=65.13M ;

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| *Epoch* | *Training* | | | *Validation* | | |
|  | Globals | Senses | Multi-senses | Globals | Senses | Multi-senses |
| 1 | 624.38 | 1.0 | 1.0 | 505.97 | 1.0 | 1.0 |
| 2 | 355.62 | 1.0 | 1.0 | 401.09 | 1.0 | 1.0 |
| … |  |  |  |  |  |  |
| 17 | 114.35 |  |  | **224.6** |  |  |
| 18 | 110.68 |  |  | 224.64 |  |  |
| 19 | 107.22 |  |  | 224.83 |  |  |
| **20** | 107.33 | 958992.42 | 54611460.56 | **224.73** | 1195787.65 | 63848283.01 |
| 21 | 107.33 | 958262.02 | 54472873.79 | 224.73 | 1195662.11 | 63829131.42 |
| 22 | 107.33 |  |  | 224.73 | 1195649.17 | 63826308.62 |
| 23 | 107.33 |  |  | 224.73 | 1195650.56 | 63826723.48 |
| 24 |  |  |  |  | 1195648.55 | 63827419.42 |
| 25 |  |  |  |  | 1195652.92 | 63828294.68 |

**Ongoing comparison table**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Method | correct\_all\_s ( / 80640) | top\_k\_all\_s | correct\_multi\_s ( / 29575) | top\_k\_multi\_s | Senses’ PPL |
| Simple GRUs | 11761 always | 32999 @ epoch 2 | 0 always | 1897 @ epoch 3 | 632.07 |
| (GRUs, c GAT 200) | 11761 always | 32989  @ epoch 13 | 0 always | 1826 @ epoch 13 | 630.39 |
| SelectK1 | 19020 @ epoch 19 | 19483 @ epoch 19 | 391 @ epoch 19 | 776 @ epoch 19 | n.s. |
| **SelectK1 + freeze** | **19050 @ epoch 24** | **19486 always** | **412 @ epoch 24** | **776 always** | n.s. |
| SelectK10 | 11761 @ epoch 1 | 32807 @epoch 1 | 118 @epoch 17 | 1139  @ epoch 1,  930  @ epoch 19 | n.s. |

### K=10, with freezing

|  |  |  |
| --- | --- | --- |
| **Architecture** | **Input Signals** | **Hyperparameters** |
| Globals: **Main GRU** with 3 layers (1024>1024>512) | 1) The word embedding of the current global (d=300) | batch\_size=32  TBPTT length=35 |
| **Senses GRU > FF-NN** as above, we apply the softmax on the logits of the senses of the most likely ***k=10*** globals | grapharea=32, hops=1 |
| **Freezing:** in 1st part, update only globals. Then freeze global’s network + embeddings, and update senses | learning rate=5\*10^(-5) |
| dropout=none |

INFO : Number of trainable parameters=65.13M ;

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| *Epoch* | *Training* | | | *Validation* | | |
|  | Globals | Senses | Multi-senses | Globals | Senses | Multi-senses |
| 1 | 624.38 | 1.0 | 1.0 | 505.97 | 1.0 | 1.0 |
| 2 | 355.62 |  |  | 401.09 |  |  |
| 3 | 291.1 |  |  | 329.69 |  |  |
| … |  |  |  |  |  |  |
| 17 | 114.35 |  |  | **224.6** |  |  |
| 18 | 110.68 |  |  | 224.64 |  |  |
| 19 | 107.22 |  |  | 224.83 |  |  |
| **20** | 107.33 | 50620.09 | 11034134.6 | **224.73** | 60932.99 'correct\_all\_s': 4212, 'top\_k\_all\_s': 28863  'tot\_all\_s': 80640 | 16716243.89  'correct\_multi\_s': 66, 'top\_k\_multi\_s': 948, 'tot\_multi\_s': 29575 |
| 21 | ” | 50257.84 | 11064336.11 | ” | 60848.46 **'correct\_all\_s': 4357, 'top\_k\_all\_s': 28889,** | 16738649.66 'correct\_multi\_s': 68, 'top\_k\_multi\_s': 939, |
| 22 | ” | 50198.14 | 11068236.97 | ” | 60827.74 'correct\_all\_s': 4320, 'top\_k\_all\_s': 28783 | 16748540.46 'correct\_multi\_s': 78, **'top\_k\_multi\_s': 945** |
| 23 | ” |  |  | ” | 60819.22  'correct\_all\_s': 4274, 'top\_k\_all\_s': 28761, | 16751168.01  'correct\_multi\_s': 88, 'top\_k\_multi\_s': 940 |
| 24 | ” |  |  | ” | 60815.52  'correct\_all\_s': 4284, 'top\_k\_all\_s': 28769 | 16753247.06  'correct\_multi\_s': 98, 'top\_k\_multi\_s': 940 |
| 25 |  |  |  |  | 60820.2  'correct\_all\_s': 4291, 'top\_k\_all\_s': 28757 | 16757746.84  **'correct\_multi\_s': 102**, 'top\_k\_multi\_s': 934 |

**Ongoing comparison table**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Method | correct\_all\_s ( / 80640) | top\_k\_all\_s | correct\_multi\_s ( / 29575) | top\_k\_multi\_s | Senses’ PPL |
| Simple GRUs | 11761 always | 32999 @ epoch 2 | 0 always | 1897 @ epoch 3 | 632.07 |
| (GRUs, c GAT 200) | 11761 always | 32989  @ epoch 13 | 0 always | 1826 @ epoch 13 | 630.39 |
| SelectK1 | 19020 @ epoch 19 | 19483 @ epoch 19 | 391 @ epoch 19 | 776 @ epoch 19 | n.s. |
| SelectK1 + freeze (e20) | 19050 @ epoch 24 | 19486 always | 412 @ epoch 24 | 776 always | n.s. |
| SelectK10 | 11761 @ epoch 1 | 32807 @ epoch 1 | 118 @epoch 17 | 1139  @ epoch 1,  930  @ epoch 19 | n.s. |
| **SelectK10 +freeze (e20)** | **4357 @ epoch 21** | **28889 @ epoch 21** | **945 @ epoch 22** | **102 @ epoch 25** | n.s |

### Time analysis of SelectK and modification

Why does K=10 cause an iteration time of ~10 seconds? Is there a way to circumvent this?

Time analysis in the forward() call:

INFO : First part

INFO : \*\*\* Chronometer:

INFO : t1 - t0 = 0.17128

INFO : Loop on samples

INFO : \*\*\* Chronometer:

INFO : t1 - t0 = 0.00436

INFO : t2 - t1 = 0.0002

INFO : t3 - t2 = 0.00154

…

INFO : Loop on samples

INFO : \*\*\* Chronometer:

INFO : t1 - t0 = 0.00261

INFO : t2 - t1 = 7e-05

INFO : t3 - t2 = 0.00096

…

INFO : Loop on samples

INFO : \*\*\* Chronometer:

INFO : t1 - t0 = 0.00251

INFO : t2 - t1 = 7e-05

INFO : t3 - t2 = 0.00089

where a loop on samples is executed batch\_size x sequence\_len times, i.e. 32\*35=1120 times.

~0.003 \* 1120 = 3.36s

Temporarily using the mini-experiment fragment dataset.

After the 1st epoch,

with bsz=2 and seq\_len=5 (10 elements)

Loop time =~ t2 - t1 = 0.03882

I could try to write sample-parallel code for the loop…

* k\_globals\_indices is a tensor of shape (bsz\*seq\_len, K), e.g. (12,10)
* we wish to have k\_globals\_lemmatized, currently done with:  
  [self.vocabulary\_lemmatizedList[idx] for idx in k\_globals\_vocab\_indices]
* …

I don’t think I can speed up the points where I am applying a function.

But I may speed up the subsequent steps, namely

sense\_neighbours\_t = get\_senseneighbours\_of\_k\_globals(self, lemmatized\_indices\_in\_X)  
  
*# standard procedure: get the logits of the senses of the most likely globals,  
# apply a softmax only over them, and then assign an epsilon probability to the other senses*sample\_logits\_senses = logits\_sense.index\_select(dim=0, index=self.select\_first\_indices[s].to(torch.int64)).squeeze()  
logits\_selected\_senses = sample\_logits\_senses.index\_select(dim=0, index=sense\_neighbours\_t)  
softmax\_selected\_senses = tfunc.softmax(input=logits\_selected\_senses, dim=0)  
  
**for** i **in** range(len(sense\_neighbours\_t)):  
 i\_senseneighbours\_mask[s,sense\_neighbours\_t[i]]=**True**quantity\_to\_subtract\_from\_selected = epsilon \* (self.last\_idx\_senses - len(sense\_neighbours\_t))  
softmax\_selected\_senses = subtract\_probability\_mass\_from\_selected(softmax\_selected\_senses, quantity\_to\_subtract\_from\_selected)  
senses\_softmax[s].masked\_scatter\_(mask=i\_senseneighbours\_mask[s].data.clone(), source=softmax\_selected\_senses)

We get the k most likely globals for each sample, lemmatize them, and collect them into lemmatized\_indices\_in\_X\_lls.

We can make it into a tensor, where every row contains 10 globals.

For **each** row,

For **each** of these 10 globals, we want to retrieve the neighbours.

We map -1 to extract correctly from the sparse matrix, and then we filter on < last\_sense\_idx, to finally obtain the “sense-neighbours of the 10 globals”, where we apply the softmax on.

On a row,

self.grapharea\_matrix[torch.tensor(lemmatized\_indices\_in\_X\_lls)[0], 0:32]

this extracts the (raw, +1) neighbours for all the k(10) globals.

This subtracts 1 from the tensor of neighbours:

self.grapharea\_matrix[torch.tensor(lemmatized\_indices\_in\_X\_lls)[0], 0:32].todense() -1

Then a tensor index could exclude both -1 and the elements > last\_idx\_senses.

No, a tensor index can not be applied directly on 2D tensors.

However, since I am operating on the 10 globals of the same sample and I have already retrieved the neighbours from the grapharea\_matrix rows, I can just flatten the tensor (it doesn’t matter which global the adjacent senses come from)

And besides, this is a dense matrix NOT a tensor

So something simpler, like

sample\_neighbours[sample\_neighbours > 0]

works.

sense\_neighbours\_t\_new[0,0:5] == [el.item() **for** el **in** sense\_neighbours\_t\_old[0:5]]

[all True]

We thus applied a minor modification. We review that we still have the same results on SelectK1, and we are going to check whether this bring any significant improvement on SelectK10 when we try the experiment with freezing.

Checking results on SelectK1 miniexperiment with freezing:

|  |  |  |  |
| --- | --- | --- | --- |
| *Epoch* | *Training* | | |
|  | Globals | Senses | Multi-senses |
| 1 | 22831.58 | 1.0 | 1.0 |
| … |  |  |  |
| 5 | 8954.96 | 1.0 | 1.0 |
| 6 | **3708.13** | **39810810.26** | **100000092.9** |
| 7 | **3702.04** | 39810810.26 | 100000092.9 |
| 8 | 3702.04 | 39810810.26 | 100000092.9 |

identical.

### Issue: SelectK10 with freezing, Invalid Index

The mini-experiment does not appear to show any problem.

Where was the original problem, and can we replicate it and log it?

Epoch n.20:

IndexError: Caught IndexError in replica 0 on device 0.

…

TypeError: can't convert cuda:0 device type tensor to numpy. Use Tensor.cpu() to copy the tensor to host memory first.

During handling of the above exception, another exception occurred:

in get\_senseneighbours\_of\_k\_globals

sample\_neighbours\_section = (model.grapharea\_matrix[sample\_k\_indices, 0:32].todense()) -1

raise IndexError('invalid index')

Executing mini-experiment on remote with GPUs…

If I don’t manage to correct the mistake, I will rollback to the previous method to get the senseneighbours of globals (even if it is slower)

“

Epoch n.3:

sample\_k\_indices=tensor([41224, 41206, 41230, 63419, 41238, 41250, 41211, 41257, 41227, 41220],

device='cuda:0')

Traceback (most recent call last):

File "/home/andrealk3/venvs/torch15/lib/python3.6/site-packages/scipy/sparse/\_index.py", line 158, in \_asindices

x = np.asarray(idx)

return self.numpy()

TypeError: can't convert cuda:0 device type tensor to numpy. Use Tensor.cpu() to copy the tensor to host memory first.

During handling of the above exception, another exception occurred:

Traceback (most recent call last):

…

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

File "/home/andrealk3/ALMA\_PhD/Code/TASK1\_Gbwe/NN/Models/Senses.py", line 172, in forward

sense\_neighbours\_t = torch.tensor(get\_senseneighbours\_of\_k\_globals(self, sample\_k\_indices\_lemmatized))\

File "/home/andrealk3/ALMA\_PhD/Code/TASK1\_Gbwe/NN/Models/Senses.py", line 20, in get\_senseneighbours\_of\_k\_globals

sample\_neighbours\_section = (model.grapharea\_matrix[sample\_k\_indices, 0:32].todense()) -1

File "/home/andrealk3/venvs/torch15/lib/python3.6/site-packages/scipy/sparse/\_index.py", line 35, in \_\_getitem\_\_

row, col = self.\_validate\_indices(key)

File "/home/andrealk3/venvs/torch15/lib/python3.6/site-packages/scipy/sparse/\_index.py", line 139, in \_validate\_indices

row = self.\_asindices(row, M)

File "/home/andrealk3/venvs/torch15/lib/python3.6/site-packages/scipy/sparse/\_index.py", line 160, in \_asindices

raise IndexError('invalid index')

IndexError: invalid index

“

Question on the side:

If I don’t operate with freeze, but I immediately optimize both the globals’ and the senses’ part, do I still encounter the problem with SelectK10?

Apparently, yes.

It just doesn’t happen on the local copy.

SelectK1 does not have this problem,

sample\_k\_indices=tensor(42052, device='cuda:1')

**What happens with SelectK1?**

INFO : k\_globals\_vocab\_indices=[846]

INFO : sample\_k\_indices\_lemmatized=tensor(42052, device='cuda:0')

INFO : sample\_k\_indices=tensor(42052, device='cuda:0')

So many times. Why?

It’s normal: the random initialization of the globals’ GRU means that it is skewed towards predicting some random words.

Because of the steps:

k\_globals\_indices = logits\_global.sort(descending=**True**).indices[:, 0:self.K]

sample\_k\_indices\_in\_vocab\_lls = k\_globals\_indices.tolist()

**for** s **in** range(distributed\_batch\_size \* seq\_len):

k\_globals\_vocab\_indices = sample\_k\_indices\_in\_vocab\_lls[s]

…

sense\_neighbours\_t =  
 torch.tensor(get\_senseneighbours\_of\_k\_globals(self,  
 sample\_k\_indices\_lemmatized))\   
 .squeeze(dim=0).to(torch.int64).to(CURRENT\_DEVICE)

In which:

**def** get\_senseneighbours\_of\_k\_globals(model, sample\_k\_indices):  
  
 sample\_neighbours\_section = (model.grapharea\_matrix[sample\_k\_indices,   
 0:32].todense()) -1  
 sample\_neighbours =   
 sample\_neighbours\_section[sample\_neighbours\_section > 0]  
 sense\_neighbours = sample\_neighbours[sample\_neighbours <   
 model.last\_idx\_senses]  
 **return** sense\_neighbours

SelectK1 does not have a problem because we always select **1** row of the grapharea\_matrix, then get the neighbours section [0:32], turn todense(), filter on >0 and < last\_idx\_senses.

There are therefore 2 alternatives:

1. move the sample\_k\_indices to cpu(), and operate on the grapharea\_matrix on the cpu.  
   sample\_neighbours\_section = (model.grapharea\_matrix[sample\_k\_indices**.cpu()**, 0:32].todense()) -1
2. Move the sparse grapharea\_matrix on GPU… however, is this viable? A Parameter can be moved on multiple GPUs via DataParallel, but a Parameter needs a Tensor…  
   Maybe yes, if we use torch.sparse torch**.**sparse**.**FloatTensor and similar

Currently, for the sake of simplicity, we simply move the sample\_k\_indices to cpu() and then the sense\_neighbours back to GPU. If it is even partially faster, it’s ok. If it’s slower, we must try something else.

# Reasoning on other methods

## Variants of Structured prediction

I partially re-insert here, and modify, the old reflections from DeveloperDiary\_3:

How could I make it so that the prediction of the next global influences or restricts the prediction of the next sense?

Hypothesis: I consider the max first k=1,10,…,100 logits of the globals. The predicted sense must be found among the senses of these words.

What do I need to implement this hypothesis:

retrieve the sense indices of the senses of the k likely globals.

Then, either:

1. (SelectK) use the senses’ GRU as normal; assign to ~0 all the softmax values of senses that are *not* of the likely globals; apply softmax on the senses of the selected K globals, predict
2. consider the first… n= e.g. 5\*k senses (they must be a fixed number).   
   Retrieve their embeddings from the graph matrix X.   
   This would give us a matrix of n (e.g.50) x (d=300).   
   We have a soft-classification task, where we need to choose one of the rows as the correct one, with a probability distribution over them.
   1. Use a self-attention score, where we have:  
      a query (e.g. the average of the last C words of the context, a standard attention projection from the last words of the context, or a state of the globals’ GRU, or the state of its own 1-layer GRU)   
      and key vectors (the embeddings of the senses)  
      => we compute to get the self-attention logits, and the softmax to have a probability distribution over the senses of the selected K globals.   
      We assign it, while keeping the other senses’ softmax at 10-8=~0, as usual
   2. Use cosine similarity between any of the senses and the entity C. Scale it between 0 and 1 or apply the softmax

In both (A) and (B), instead of having to create a probability distribution over the whole set of 40K+ senses, the distribution p would be over a small number of candidates, determined by the globals I am currently predicting.

Note: since the meaning of Perplexity is “on average, how many alternatives we are undecided on” and the values of Valid PPL on globals oscillate around ~224 if we train on SemCor alone, it would probably be better to select K=~200 globals if we wished to “shift” the difficulty of predicting from the globals’ task to the senses’ task.

After all, in Structured prediction, if we are not predicting the correct global, then we have no chance whatsoever of predicting the correct sense.

## Variants of General prediction

We try now to find alternative means to have a probability distribution over the whole set of 40K+ senses, that are not the GRU.

If we had a cosine similarity comparison between \*all\* the senses in the graph matrix X and the entity C (average of last words / attention mechanisn / GRU state as it may be),   
would it be fast enough?

# Rank by context similarity

## Design & implementation

* We keep a running average of the last C word embeddings we encounter (in the global task).
* For every sample s in batch\_size \* sequence\_length,   
   Individuate the K most likely predicted globals, select their senses  
   Compute the cosine similarity between the selected senses and the running average  
   of the word embeddings up to s  
   Rank the selected senses based on the cosine similarity

for batch\_elements\_at\_t in time\_instants: …

t\_word\_embeddings.shape=(2,300)

self.context\_embs\_avg.shape=(2,5,300)

Then, we should still use the artificial softmax used for SelectK: assign ε to everything else, and 1 to the most likely sense (ranking, i.e. we need to pick the one with the maximum cosine similarity to the running average of the context)

Issue: *This model can work only with senses that have the same embedding dimension as the globals*. Currently I have senses of d=200 after PCA, and embeddings – as usual – of dim=300.

Must rebuild the graph using the senses as they are, without PCA reduction.

Preparing to run mini-experiment…

Note: when I add the sense embeddings into the running average of the context, I should keep their gradient.

Checking the softmax handling in SelectK:

sample\_logits\_senses.shape = 41206

logits\_selected\_senses.shape=27 (it could be anything)

softmax\_selected\_senses = tfunc.softmax(input=logits\_selected\_senses, dim=0); shape=27

# With batch\_size=3 and sequence\_length=4,

senses\_softmax.shape=(12, 41206)

i\_senseneighbours\_mask.shape=(12, 41206)

# Then,

senses\_softmax[s].masked\_scatter\_(mask=i\_senseneighbours\_mask[s].data.clone(), source=softmax\_selected\_senses)

**Issue**:

Warning The .grad attribute of a Tensor that is not a leaf Tensor is being accessed.

Probably because I should index\_select the word embeddings and just assign a tensor on the fly? I have the problem of keeping the running average.

The self.context\_embs\_avg … should be like the memories of the RNN? a Parameter with requires\_grad=False?

If it is not a Parameter, then we have a problem: we can not distribute it on the replicated models on the different GPUs. So it should be a Parameter…

However, here we are computing the running average of the word embeddings we encounter…

And this running average should encompass the previous C words

Let’s keep requires\_grad = True

This seems to have eliminated the problem.

Regarding the formula of the running average:

l1 = [0,10,30,100,200,5000]

[l1[i]/3 + (l1[i+1])/3 + (l1[i+2])/3 for i in [0,1,2,3]]=

[13.34, 46.67, 110.0, 1766.67]

At some point we have to drop the most ancient values…

i.e. we have to store the last C, and then compute the average on them

Therefore, we need the 4 dimensions:

self.context\_running\_embeddings =

Parameter(torch.zeros(

size=(batch\_size, self.seq\_len, self.C, self.dim\_embs)) .to(torch.float32), requires\_grad=**True**)

We can use a Tensor as a queue by doing:

torch.cat((t1[1:], torch.tensor([40]) )).

For us,

torch.cat((self.context\_running\_embeddings[:, i, 1:, :], t\_word\_embeddings.unsqueeze(dim=1).clone()), dim=1)

going from (3,9,300) to (3,10,300)

Then, we will have to use

torch.mean(self.context\_running\_embeddings, dim=2)

to get the running average. We compare it with our senses’ embeddings.

cosine\_sim\_scores = self.cosine\_sim(torch.mean(self.context\_running\_embeddings, dim=2)[b,l].unsqueeze(dim=0), *# running average* t\_sense\_embeddings)

However, the previous **Issue**,

Warning The .grad attribute of a Tensor that is not a leaf Tensor is being accessed.

returns.

“It means that you are accessing the .grad field of a Tensor for which PyTorch will never populate the .grad field.

Note that, leaf Tensors will have their .grad field populated.

So if this warning happens, it means that you think something a leaf tensor but actually it isn’t. And this usually happens if you perform operations on a Tensor that requires gradients.”

So let us set requires\_grad=False again for

self.context\_running\_embeddings

and also assigning with clone().detach()

Now the issue has disappeared. However, we are utterly unable to learn – the senses’ embeddings have *no gradient*.

Let us keep this as the version 1.0.

## Checks, Mini-experiments and analysis of predictions

### Checks

I would like to print the word embeddings we encounter (and also, their coordinates in the batch x sequence\_length structure. It may be that we need to extract – and select the senses for – a lower number of globals than expected?

(It would seem this is not the case. The batches on each GPU handle the elements sequentially, each one is a different segment)

Second objective: checking that the computation of the average of embeddings is correct.

With batch\_size=2 (i.e. 2 elements per GPU(2)) and sequence\_length=5:

Device cuda:0

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Device \ Time | t0 | t1 | t2 | t3 | t4 |
| cuda:0 | [22213] | 22213 | 222213 | 222213 | 3 |
| cuda:1 | 4 | 5 | 222213 | 5178 | 8 |

vocabulary indices -> words:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Device \ Time | t0 | t1 | t2 | t3 | t4 |
| cuda:0 | <unk> | <unk> | <unk> | <unk> | an |
| cuda:1 | investigation | of | <unk> | s | recent |

Major problem on SemCor: why do I have all of these <unk>?

Review the DataLoader:

Adjusted lowercasing to True when invoking process\_word\_token().

What now? re-check:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Device \ Time | t0 | t1 | t2 | t3 | t4 |
| cuda:0 | 0 | 22213 | 1 | 2 | 3 |
| cuda:1 | 4 | 5 | 6 | 5178 | 98 |

vocabulary indices -> words:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Device \ Time | t0 | t1 | t2 | t3 | t4 |
| cuda:0 | the | <unk> | said | Friday | an |
| cuda:1 | investigation | of | atlanta | s | recent |

NOW it is correct.

# Part 2 – Same experiments as before, now reading SemCor with the correct lowercasing

## 2 GRUs on SemCor

Status check: the current vocabulary is lowercased

(It could be simpler to just re-write everything in SemCor, lowercasing it…)

The indices\_table.sql is also already lowercased.

### Lowercase check

d=300 > 1024 > 1024 > 512

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| *Epoch* | *Training* | | | *Validation* | | |
|  | Globals | Senses | Multi-senses | Globals | Senses | Multi-senses |
| 1 | 1007.94 | 1286.26 | 11204.69 | 818.94 | 1018.17 | 10363.95 |
| … |  |  |  |  |  |  |
| 17 | 170.95 | 690.03 | 6222.34 | 365.4 | 1180.8 | 11171.08 |

Hypothesis: we removed reading so many upper-cased words as <unk>. Thus, we have removed the artificial “easier” reading of the corpus as “mostly UNK”.

It may very well be possible that, even lowercased, training on SemCor *alone* means that 600K unprocessed tokens are simply not enough to get a good performance.

In which case, how does the pre-trained model we got from WikiText-2 fare?

We should first create a new pre-trained model

## GNN Input Signal

### 2GRUs + GAT300

## SelectK

### K=1

### K=10

### K=1, with freezing

### K=10, with freezing

# Context Similarity – continuation

### K=1, C=10

ContextSim(

(main\_rnn\_ls): ModuleList(

(0): GRU(300, 1024)

(1): GRU(1024, 1024)

(2): GRU(1024, 512)

)

(cosine\_sim): CosineSimilarity()

(linear2global): Linear(in\_features=512, out\_features=22782, bias=True)

)

)

Number of trainable parameters=68.02M,

where: word embeddings= 6.84M , graph=36.71M , softmax=11.69M, GRU=12.8M

K=1 (number of most likely globals, from them we select the candidate senses),

C=10 (width of the running average of the context)

note: The senses’ embeddings do not move and there is no learning architecture for the senses. We just consider the senses of the most likely K globals and compute the cosine similarity with the last C words in the context (with the FastText word embeddings).

|  |  |  |  |
| --- | --- | --- | --- |
| *Epoch* | *Training* | | |
|  | Globals | Senses | Multi-senses |
| 1 |  |  |  |
| 2 |  |  |  |
| … |  |  |  |
| 100 |  |  |  |
| 200 |  |  |  |
| 300 |  |  |  |
| 400 |  |  |  |
| 500 |  |  |  |
| 600 |  |  |  |

### K=10, C=10

K=10: number of most likely globals, from them we select the candidate senses

C=10: width of the running average of the context

|  |  |  |  |
| --- | --- | --- | --- |
| *Epoch* | *Training* | | |
|  | Globals | Senses | Multi-senses |
| 1 |  |  |  |
| 2 |  |  |  |
| … |  |  |  |
| 100 |  |  |  |
| 200 |  |  |  |
| 300 |  |  |  |
| 400 |  |  |  |
| 500 |  |  |  |
| 600 |  |  |  |

# Protocol for experiments

* Is the vocabulary lowercased?
* Is the indices\_table.sql lowercased?
* What are we reading from the DataLoader? i.e. what is in the batches that the Neural Network reads? Are we reading all <UNK>s?

# Re-writing SemCor as lowercased