[Preliminary information (Datasets, graphs) 4](#_Toc40780897)

[WikiText-2 4](#_Toc40780898)

[WikiText-2 Stats 4](#_Toc40780899)

[Graph 4](#_Toc40780900)

[SemCor 5](#_Toc40780901)

[SemCor stats 5](#_Toc40780902)

[Graph 5](#_Toc40780903)

[Globals - RNN 6](#_Toc40780904)

[Architecture 6](#_Toc40780905)

[Experiments 6](#_Toc40780906)

[Mini-experiment – overfit on fragment 6](#_Toc40780907)

[Experiment – RNN on WT2 8](#_Toc40780908)

[Globals – GRU 9](#_Toc40780909)

[Architecture 9](#_Toc40780910)

[Experiments 9](#_Toc40780911)

[Mini-experiment – overfit GRU on fragment of WT2 9](#_Toc40780912)

[Experiment – GRU on WT2 11](#_Toc40780913)

[Including the graph input 13](#_Toc40780914)

[Review: Graph Attention Networks 13](#_Toc40780915)

[Review: RGAT: Relational Graph-Attention Networks 14](#_Toc40780916)

[Globals & Graph Input – Graph Attention Networks 15](#_Toc40780917)

[GAT, input signals, 4-heads 15](#_Toc40780918)

[Architecture 15](#_Toc40780919)

[Mini-Experiment, GRU-GAT4 on fragment of WT2 16](#_Toc40780920)

[Experiment – GRU-GAT4 on WT2 16](#_Toc40780921)

[Experiment – GAT with 1 head 18](#_Toc40780922)

[The Sense task 20](#_Toc40780923)

[Reflections 20](#_Toc40780924)

[Baseline: Simple GRU on SemCor 20](#_Toc40780925)

[Mini-experiment: overfitting on a fragment of SemCor 20](#_Toc40780926)

[Experiment – GRU on SemCor 22](#_Toc40780927)

[Baseline 2: GRU + GAT4 (current global’s node state) on SemCor 23](#_Toc40780928)

[Mini-Experiment – Overfit on a fragment of SemCor 23](#_Toc40780929)

[Experiment – GRU+GAT4 on SemCor 23](#_Toc40780930)

[Senses: Photo-concat 25](#_Toc40780931)

[Architecture 25](#_Toc40780932)

[Experiment – Globals and senses (GRU + 2xGAT4) on SemCor 26](#_Toc40780933)

[Senses: Shared GRU layer 27](#_Toc40780934)

[Architecture 27](#_Toc40780935)

[Experiment 1 – Senses: Shared GRU Layer 28](#_Toc40780936)

[Senses: Self-attention scores on the senses of the k most likely globals 29](#_Toc40780937)

[Design 29](#_Toc40780938)

[Reviewing self-attention 30](#_Toc40780939)

[Implementation 30](#_Toc40780940)

[Experiments 32](#_Toc40780941)

[Meetings 35](#_Toc40780942)

[IA 35](#_Toc40780943)

[IA, PSK, KF 35](#_Toc40780944)

[Senses Baseline: Operate on the k most likely globals 36](#_Toc40780945)

[Hypothesis 1: FF-NN – from all logits to all logits 37](#_Toc40780946)

[Hypothesis 2: Project embeddings of the first k globals + concat logit + FF-NN 37](#_Toc40780947)

[Model 38](#_Toc40780948)

[Experiment on SemCor 38](#_Toc40780949)

[Senses: self-attention on the k most likely globals 39](#_Toc40780950)

[Design 39](#_Toc40780951)

[Model 39](#_Toc40780952)

[Experiments 40](#_Toc40780953)

[Overfit on mini-dataset 40](#_Toc40780954)

[Experiment 1 – SelfAttention4Senses on SemCor 41](#_Toc40780955)

[Experiment 2 – SelfAttention4Senses on SemCor 42](#_Toc40780956)

[Globals – LSTM 43](#_Toc40780957)

[Design 43](#_Toc40780958)

[Elements in AWD-LSTM 43](#_Toc40780959)

[Hyperpararameters used in AWD-LSTM 44](#_Toc40780960)

[Their Perplexity Results 44](#_Toc40780961)

[Implementation 45](#_Toc40780962)

[Salesforce’s AWD-LSTM 45](#_Toc40780963)

[My own WD-LSTM 45](#_Toc40780964)

[Experiments 46](#_Toc40780965)

[Mini-experiment 1 – overfit LSTM1 on fragment of WT2 46](#_Toc40780966)

[Mini-experiment 2 – overfit LSTM2 on fragment of WT2 47](#_Toc40780967)

[Mini-experiment 3 – overfit LSTM3 on fragment of WT2 48](#_Toc40780968)

[Experiment 1 – LSTM3 on WT2 49](#_Toc40780969)

[Experiment 2 – WD-LSTM3 on WT2 50](#_Toc40780970)

[Experiment 3 – WD-LSTM3 on WT2 51](#_Toc40780971)

[Globals – GRU 52](#_Toc40780972)

[Design 52](#_Toc40780973)

[Experiments 53](#_Toc40780974)

[Mini-experiment 1 – Overfit on fragment of WT-2 53](#_Toc40780975)

[**Experiment 1 – GRU on WT-2** 53](#_Toc40780976)

[Including the Graph Input 55](#_Toc40780977)

[GRU\_GAT 55](#_Toc40780978)

[Model 55](#_Toc40780979)

[Mini-experiment 1 – Overfit on fragment of WT-2 55](#_Toc40780980)

[**Experiment 1 – GRU\_GAT on WT-2** 56](#_Toc40780981)

[WD-LSTM-GAT 59](#_Toc40780982)

[Model 59](#_Toc40780983)

[Experiment 1 – WD-LSTM-GAT 59](#_Toc40780984)

[Including the whole AWD-LSTM implementation 60](#_Toc40780985)

[Input facilities 60](#_Toc40780986)

[Training loop 60](#_Toc40780987)

[Corpus 61](#_Toc40780988)

[Batch & Input data 62](#_Toc40780989)

[Embeddings used by the model 63](#_Toc40780990)

[Hypothesis: use X and the indices of the globals 64](#_Toc40780991)

[Adjusting awd-lstm for the current PyTorch setup (1.4.0) 65](#_Toc40780992)

[WeightDrop 65](#_Toc40780993)

[Optimizer error 66](#_Toc40780994)

[Weights can be flattened in memory 66](#_Toc40780995)

[Torch 0.4.0 and 1.0.0 67](#_Toc40780996)

[Out of Memory 67](#_Toc40780997)

[On the side: batching for GAT 69](#_Toc40780998)

[Unpacking the batchinput\_tensor 70](#_Toc40780999)

[Full batching for GATConv 71](#_Toc40781000)

[Batches and DataParallel 71](#_Toc40781001)

[Meeting with IA, 12/05 72](#_Toc40781002)

[Multi-Sense LM 73](#_Toc40781003)

[Baseline 1: 2 GRUs, shared first layer 73](#_Toc40781004)

[Model 73](#_Toc40781005)

[Mini-experiment on fragment of SemCor 73](#_Toc40781006)

[Experiment –GRUs w/ shared layer on SemCor (fault: not detaching the gru\_senses’ memory) 74](#_Toc40781007)

[Experiment –GRUs w/ shared layer on SemCor 76](#_Toc40781008)

[Baseline 2 – 2GRUs (x3 and x2), no shared layers 77](#_Toc40781009)

[Model 77](#_Toc40781010)

[Experiment – 2 GRUs on SemCor 78](#_Toc40781011)

[Experiment – 2 GRUs on SemCor w/ sense-node input 79](#_Toc40781012)

[Structured prediction – select from K globals 80](#_Toc40781013)

[Design 80](#_Toc40781014)

[Mini-experiment – overfit on fragment of SemCor 81](#_Toc40781015)

[Graph check 82](#_Toc40781016)

[Senses.SelectK retrieval and selection check 83](#_Toc40781017)

[Modifications to SelectK 85](#_Toc40781018)

[Bugs & co 86](#_Toc40781019)

# Preliminary information (Datasets, graphs)

## WikiText-2

### WikiText-2 Stats

Number of tokens in the splits, counted using nltk’s tokenizer:

Training: 2,207,934 tokens (1,774,387 without punctuation)

Validation: 242,478 (186,258)

Test: 277,846 (209,228)

Mini-Dataset – training: 632 tokens (530 without punctuation)

### Graph

After processing WikiText-2 (reading in, getting FastText single-prototype embeddings, retrieving WordNet informations and egdes), the resulting graph will be:

(From the log at DefineGraph.log)

Constructing X, matrix of node features

**X\_definitions**.shape=torch.Size([**28070**, 300])

**X\_examples**.shape=torch.Size([**26544**, 300])

**X\_senses**.shape=torch.Size([**28070**, 300])

**X\_globals**.shape=torch.Size([**31640**, 300])

Defining the edges: def, exs

**def\_edges**\_se.\_\_len\_\_()=**28070**

**exs\_edges**\_se.\_\_len\_\_()=**26544**

Defining the edges: sc

**sc\_edges**.\_\_len\_\_()=**28070**

**#** Currently, since WordNet is our only source, we have the correspondence 1sense-1definition, so the number of SenseChildren edges and the number of Definition edges coincide.

**sc\_edges\_with\_selfloops**.\_\_len\_\_()=**49276**

**#** The Relational Graph Convolutional Network and Graph Attention Network both require that all nodes have at least 1 edge – to satisfy this requirement, we add a self-loop to all the globals that do not have a sense

(example: the stopwords, like ‘for’, ‘and’, ‘of’, etc.)

This way, we also determine that there were 49276 – 28070 **= 21206** globals with no dictionary information, over a total of **31640 (67%).**

syn\_edges.\_\_len\_\_()=**40016 #** synonyms

ant\_edges.\_\_len\_\_()=**3938** **#** antonyms

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

Data(edge\_index=[2, 147844], edge\_type=[**147844**], node\_types=[**114324**], num\_relations=[1], **x=[114324, 300]**)

## SemCor

### SemCor stats

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

Training tokens: 646,038

Validation tokens: 80,760

Mini-dataset: 180 tokens.

### Graph

We gather the WordNet data and build the graph using SemCor’s vocabulary.

Constructing X, matrix of node features

**X\_definitions**.shape=torch.Size([**25986**, 300])

**X\_examples**.shape=torch.Size([**26003**, 300])

**X\_senses**.shape=torch.Size([**25986**, 300])

**X\_globals**.shape=torch.Size([**21988**, 300])

Defining the edges: def, exs

def\_edges\_se.\_\_len\_\_()=25986

exs\_edges\_se.\_\_len\_\_()=26003

Defining the edges: sc

sc\_edges.\_\_len\_\_()=25986

sc\_edges\_with\_selfloops.\_\_len\_\_()=38611

This way, we also determine that there were 38611– 21988**= 16623** globals with no dictionary information, over a total of **21988 (75.6%).**

**syn\_edges**.\_\_len\_\_()=**26222**

**ant\_edges**.\_\_len\_\_()=**3780**

Data(edge\_index=[2, 120602], edge\_type=[**120602**], node\_types=[**99963**], num\_relations=[1], **x=[99963, 300]**)

# Meetings

## IA

* Alternative: simple selection from best globals, no self-attention, also works as estimate [the current simple selection overfits on fragment, but it does not give good results]
* Alternative: proceed in parallel with senses’ model and the selection from globals, and then make the probability derived from both, e.g. with a product
* Additional: when I have stable results, it would be relevant to re-run *everything* using Distil/Al-BERT’s pre-trained embreddings
* For the section meeting, I can present the whole project instead of focusing too much on what I am doing now – possibly also using the poster for the Innovation Foundation
  + It can be 10/11 to 20 minutes
  + I can also explain what different senses are, and show how SemCor works

## IA, PSK, KF

* Transformer-LM: plug in the filter that handles partial words / word completion.
* To improve performance on WikiText-2, I can use:
  + better/larger GRU
  + AWD-LSTM
  + Pre-trained embeddings from DistilBERT instead of FastText
* (How does Perplexity on Senses compare to standard Perplexity:  
  hp: the number of tokens that we compute PPL on. (e.g. PPL on text corpus of 2Mln will be different from PPL on text corpus of 103Mln words)  
  hp2: I could consider all the words that have a sense-specification on SemCor, and then compute the PPL over WikiText-2 only for those words – how does it compare with the standard PPL on WikiText-2?  
  note: PPL is difficult to compare between different datasets and tasks. Since we are in a new task, we can also just use our measure. (WSD uses F1-score))
* Implement structured prediction from globals to senses, as a baseline for the senses part of the task
* (Go on with self-attention for the senses task)

# Attempt to include the whole AWD-LSTM implementation

We move on to reviewing the official SalesForce implementation of AWD-LSTM, to use it as a proper point of comparison.

We use the default parameters for the model: dropout on h2h at p=0.5, dropout on embeddings, etc. etc.

## Input facilities

### Training loop

The trainining loop’s function:

**def** train():  
 *# Turn on training mode which enables dropout.* **if** args.model == **'QRNN'**: model.reset()  
 total\_loss = 0  
 start\_time = time.time()  
 ntokens = len(**corpus**.dictionary)  
 hidden = model.init\_hidden(args.batch\_size)  
 batch, i = 0, 0  
 **while** i < **train\_data**.size(0) - 1 - 1:  
 bptt = args.bptt **if** np.random.random() < 0.95 **else** args.bptt / 2.  
 *# Prevent excessively small or negative sequence lengths* seq\_len = max(5, int(np.random.normal(bptt, 5)))  
 *# There's a very small chance that it could select a very long sequence length resulting in OOM  
 # seq\_len = min(seq\_len, args.bptt + 10)* lr2 = optimizer.param\_groups[0][**'lr'**]  
 optimizer.param\_groups[0][**'lr'**] = lr2 \* seq\_len / args.bptt  
 model.train()  
 data, targets = **get\_batch**(train\_data, i, args, seq\_len=seq\_len)  
  
 *# Starting each batch, we detach the hidden state from how it was previously produced.  
 # If we didn't, the model would try backpropagating all the way to start of the dataset.* hidden = repackage\_hidden(hidden)  
 optimizer.zero\_grad()  
  
 output, hidden, rnn\_hs, dropped\_rnn\_hs = model(data, hidden, return\_h=**True**)  
 raw\_loss = criterion(model.decoder.weight, model.decoder.bias, output, targets)  
  
 loss = raw\_loss  
 *# Activation Regularization* **if** args.alpha: loss = loss + sum(args.alpha \* dropped\_rnn\_h.pow(2).mean() **for** dropped\_rnn\_h **in** dropped\_rnn\_hs[-1:])  
 *# Temporal Activation Regularization (slowness)* **if** args.beta: loss = loss + sum(args.beta \* (rnn\_h[1:] - rnn\_h[:-1]).pow(2).mean() **for** rnn\_h **in** rnn\_hs[-1:])  
 loss.backward()  
  
 *# `clip\_grad\_norm` helps prevent the exploding gradient problem in RNNs / LSTMs.* **if** args.clip: torch.nn.utils.clip\_grad\_norm\_(params, args.clip)  
 optimizer.step()  
  
 total\_loss += raw\_loss.data  
 optimizer.param\_groups[0][**'lr'**] = lr2  
 **if** batch % args.log\_interval == 0 **and** batch > 0:  
 cur\_loss = total\_loss.item() / args.log\_interval  
 elapsed = time.time() - start\_time  
 print(**'| epoch {:3d} | {:5d}/{:5d} batches | lr {:05.5f} | ms/batch {:5.2f} | '  
 'loss {:5.2f} | ppl {:8.2f} | bpc {:8.3f}'**.format(  
 epoch, batch, len(train\_data) // args.bptt, optimizer.param\_groups[0][**'lr'**],  
 elapsed \* 1000 / args.log\_interval, cur\_loss, math.exp(cur\_loss), cur\_loss / math.log(2)))  
 total\_loss = 0  
 start\_time = time.time()  
 *###* batch += 1  
 i += seq\_len

### Corpus

ntokens = len(**corpus**.dictionary)

How is a corpus produced?

**if** os.path.exists(fn):  
 print(**'Loading cached dataset...'**)  
 corpus = torch.load(fn)  
**else**:  
 print(**'Producing dataset...'**)  
 corpus = data.Corpus(args.data)  
 torch.save(corpus, fn)

Let us examine the Corpus class in the …/awd-lstm-lm/data.py module:

**class** Corpus(object):  
 **def** \_\_init\_\_(self, path):  
 self.dictionary = Dictionary()  
 self.train = self.tokenize(os.path.join(path, **'train.txt'**))  
 self.valid = self.tokenize(os.path.join(path, **'valid.txt'**))  
 self.test = self.tokenize(os.path.join(path, **'test.txt'**))  
  
 **def** tokenize(self, path):  
 *"""Tokenizes a text file."""* **assert** os.path.exists(path)  
 *# Add words to the dictionary* **with** open(path, **'r'**) **as** f:  
 tokens = 0  
 **for** line **in** f:  
 words = line.split() + [**'<eos>'**]  
 tokens += len(words)  
 **for** word **in** words:  
 self.dictionary.add\_word(word)  
  
 *# Tokenize file content* **with** open(path, **'r'**) **as** f:  
 ids = torch.LongTensor(tokens) // This tensor, stored for instance in  
 // self.train, holds the whole corpus as indices  
 token = 0  
 **for** line **in** f:  
 words = line.split() + [**'<eos>'**]  
 **for** word **in** words:  
 ids[token] = self.dictionary.word2idx[word]  
 token += 1  
  
 **return** ids

In a folder, we have train.txt, valid.txt, and test.txt.

For every line, it splits on whitespace and <eos>.

Then, it adds to the dictionary, with:

self.dictionary.add\_word(word)ids[token] = self.dictionary.word2idx[word]

The dictionary works as follows:

**class** Dictionary(object):  
 **def** \_\_init\_\_(self):  
 self.word2idx = {}  
 self.idx2word = []  
 self.counter = Counter()  
 self.total = 0  
  
 **def** add\_word(self, word):  
 **if** word **not in** self.word2idx:  
 self.idx2word.append(word)  
 self.word2idx[word] = len(self.idx2word) - 1  
 token\_id = self.word2idx[word]  
 self.counter[token\_id] += 1  
 self.total += 1  
 **return** self.word2idx[word]  
  
 **def** \_\_len\_\_(self):  
 **return** len(self.idx2word)

### Batch & Input data

data, targets = **get\_batch**(train\_data, i, args, seq\_len=seq\_len)

…

output, hidden, rnn\_hs, dropped\_rnn\_hs = model(data, hidden, return\_h=**True**)

In get\_batch():

**def** get\_batch(source, i, args, seq\_len=**None**, evaluation=**False**):  
 seq\_len = min(seq\_len **if** seq\_len **else** args.bptt, len(source) - 1 - i)  
 data = **source[i:i+seq\_len]**  
 target = source[i+1:i+1+seq\_len].view(-1)  
 **return** data, target

The training data were obtained in main.py as follows:

**train\_data = batchify(corpus.train, args.batch\_size, args)**  
val\_data = batchify(corpus.valid, eval\_batch\_size, args)  
test\_data = batchify(corpus.test, test\_batch\_size, args)

We recall that corpus.train is:

self.train = self.tokenize(os.path.join(path, **'train.txt'**))

What is done in batchify?

**def** batchify(data, bsz, args):  
 *# Work out how cleanly we can divide the dataset into bsz parts.* nbatch = data.size(0) // bsz  
 *# Trim off any extra elements that wouldn't cleanly fit (remainders).* data = data.narrow(0, 0, nbatch \* bsz)  
 *# Evenly divide the data across the bsz batches.* data = data.view(bsz, -1).t().contiguous()  
 **if** args.cuda:  
 data = data.cuda()  
 **return** data

Example of batchify:

data = tensor([ 10, 20, 30, 40, 50, 60, 70, 80, 90, 100, 110, 120, 130, 140,

150, 160, 170, 180])

After batchify with bsz=4, we cut off any remainder and distribute the data in batches:

data= tensor([[ 10, 50, 90, 130],

[ 20, 60, 100, 140],

[ 30, 70, 110, 150],

[ 40, 80, 120, 160]])

with seq\_len=3, in get\_batch(), that has source[i:i+seq\_len],

i=0: tensor([[ 10, 50, 90, 130],

[ 20, 60, 100, 140],

[ 30, 70, 110, 150]])

i=1: tensor([[ 20, 60, 100, 140],

[ 30, 70, 110, 150],

[ 40, 80, 120, 160]])

### Embeddings used by the model

When it is initialized, we have:

self.encoder = nn.Embedding(ntoken, ninp)

We should exclude init\_weights, since it executes:

**def** init\_weights(self):  
 initrange = 0.1  
 self.encoder.weight.data.uniform\_(-initrange, initrange)  
 self.decoder.bias.data.fill\_(0)  
 self.decoder.weight.data.uniform\_(-initrange, initrange)

In the forward() function:

…

emb = embedded\_dropout(self.encoder, input, dropout=self.dropoute **if** self.training **else** 0)

emb = self.lockdrop(emb, self.dropouti) //embedding dropout – no need to change this

…

raw\_output = emb

**for** l, rnn **in** enumerate(self.rnns):  
 current\_input = raw\_output  
 raw\_output, new\_h = rnn(raw\_output, hidden[l])  
 new\_hidden.append(new\_h)  
 raw\_outputs.append(raw\_output)

… etc

…

We should initialize the tensor of the torch.nn.Embedding with the vectors from our matrix X.

Regarding torch.nn.Embedding, from the docs:

A simple lookup table that stores embeddings of a fixed dictionary and size.  
This module is often used to store word embeddings and retrieve them using indices. The input to the module is a list of indices, and the output is the corresponding word embeddings.

Moreover, in embedded\_dropout(), the data is obtained as follows:

…

X = torch.nn.functional.embedding(words, masked\_embed\_weight,  
 padding\_idx, embed.max\_norm, embed.norm\_type,  
 embed.scale\_grad\_by\_freq, embed.sparse  
)  
**return** X

On torch.nn.functional.embedding(*input*, *weight*, *padding\_idx=None*, *max\_norm=None*, *norm\_type=2.0*, *scale\_grad\_by\_freq=False*, *sparse=False*)

, from the docs:

A simple lookup table that looks up embeddings in a fixed dictionary and size.  
This module is often used to retrieve word embeddings using indices. The input to the module is a list of indices, and the embedding matrix, and the output is the corresponding word embeddings.  
Among the parameters,  
**input** (*LongTensor*) – Tensor containing indices into the embedding matrix  
**weight** ([*Tensor*](https://pytorch.org/docs/stable/tensors.html#torch.Tensor)) – The embedding matrix with number of rows equal to the maximum possible index + 1, and number of columns equal to the embedding size

## Hypothesis: use X and the indices of the globals

What if I mapped the indices of its default vocabulary to the globals’ indices, and then sent as matrix of embeddings the X matrix that we have from the graph?

X contains the pre-trained embeddings from FastText as a starting point. Then, one can add the GAT in parallel.

It is possible to assign to the encoder.weights.data:

embeddings.weight.data = t2

Their initialization is similar:

self.encoder.weight.data.uniform\_(-initrange, initrange)

And I could adjust the input indices, mapping them onto X (globals), before passing them to the model.

This leaves open the question of how to get the input indices in the first place.

I need to read the corpus and translate it into indices. I could also use the H5 archives to read in the globals’ indices (still without the shift they have in X) directly, so I would need no mapping.

Before I return onto this, however, it is necessary to modify whatever is necessary to get the awd-lstm model to work with PyTorch 1.4.0.

## Adjusting awd-lstm for the current PyTorch setup (1.4.0)

The command for Word level WikiText-2 (WT2) with LSTM:

python main.py --epochs 750 --data data/wikitext-2 --save WT2.pt --dropouth 0.2 --seed 1882

We add:

../../../TextCorpuses/wikitext-2/

The model is initialized as:

Loading cached dataset...

Applying weight drop of 0.5 to weight\_hh\_l0

Applying weight drop of 0.5 to weight\_hh\_l0

Applying weight drop of 0.5 to weight\_hh\_l0

[WeightDrop(

(module): LSTM(400, 1150)

), WeightDrop(

(module): LSTM(1150, 1150)

), WeightDrop(

(module): LSTM(1150, 400)

)]

Using []

### WeightDrop

The error:

AttributeError: 'LSTM' object has no attribute 'weight\_hh\_l0'

follows.

We encounter a bug for PyTorch 1.4.0:

The root cause of this bug is that \_apply function for rnn module does not work correctly after the parameters of rnn have been renamed, like in WeightDrop module here, or after applying weight\_norm, so a minimum script to trigger an error cause by buggy \_apply is this.

I can use my own code for WeightDrop, that operates as follows:

* **get** the model.\_parameters Ordered Dictionary
* apply dropout on specified parameter
* **set** named parameter in \_parameters
* replace the module's forward with our forward

To make it pickle-able, we have to define a Class with a call() method, at the top level of the module.

Now it appears to start correctly:

Loading cached dataset...

[LSTM(400, 1150), LSTM(1150, 1150), LSTM(1150, 400)]

Using []

Args: Namespace(alpha=2, batch\_size=80, beta=1, bptt=70, clip=0.25, cuda=True, data='../../../TextCorpuses/wikitext-2/', dropout=0.4, dropoute=0.1, dropouth=0.2, dropouti=0.65, emsize=400, epochs=750, log\_interval=200, lr=30, model='LSTM', nhid=1150, nlayers=3, nonmono=5, optimizer='sgd', resume='', save='WT2.pt', seed=1882, tied=True, wdecay=1.2e-06, wdrop=0.5, when=[-1])

Model total parameters: 33556078

| epoch 1 | 200/ 372 batches | lr 30.00000 | ms/batch 149.23 | loss 7.80 | ppl 2436.31 | bpc 11.250

### Optimizer error

However, after a while we encounter an error:

Saving model (new best validation)

| epoch 36 | 200/ 372 batches | lr 30.00000 | ms/batch 150.58 | loss 4.36 | ppl 78.27 | bpc 6.290

-----------------------------------------------------------------------------------

| end of epoch 36 | time: 64.84s | valid loss 4.53 | valid ppl 93.16 | valid bpc 6.542

-----------------------------------------------------------------------------------

| epoch 37 | 200/ 372 batches | lr 30.00000 | ms/batch 145.52 | loss 4.33 | ppl 75.67 | bpc 6.242

-----------------------------------------------------------------------------------

| end of epoch 37 | time: 64.81s | valid loss 4.52 | valid ppl 92.13 | valid bpc 6.526

-----------------------------------------------------------------------------------

Saving model (new best validation)

| epoch 38 | 200/ 372 batches | lr 30.00000 | ms/batch 148.20 | loss 4.31 | ppl 74.15 | bpc 6.212

-----------------------------------------------------------------------------------

| end of epoch 38 | time: 64.82s | valid loss 4.53 | valid ppl 93.22 | valid bpc 6.543

-----------------------------------------------------------------------------------

Switching to ASGD

| epoch 39 | 200/ 372 batches | lr 30.00000 | ms/batch 149.06 | loss 4.28 | ppl 72.17 | bpc 6.173

Traceback (most recent call last):

File "main.py", line 245, in <module>

prm.data = optimizer.state[prm]['ax'].clone()

KeyError: 'ax'

We check that the ‘ax’ parameter actually exists in the optimizer (apparently it does in Adam, not in ASGD), and add some other lines from the GitHub fix at Issue #70.

### Weights can be flattened in memory

/pytorch/aten/src/ATen/native/cudnn/RNN.cpp:1266: UserWarning: RNN module weights are not part of single contiguous chunk of memory. This means they need to be compacted at every call, possibly greatly increasing memory usage. To compact weights again call flatten\_parameters().

In the meantime, we obtain:

end of epoch 196 | time: 65.87s | valid loss 4.39 | valid ppl 80.42 | valid bpc 6.329

We proceed to:

end of epoch 570 | time: 66.15s | valid loss 4.37 | valid ppl 78.72 | valid bpc 6.299

Eventually,

RuntimeError: CUDA out of memory. Tried to allocate 814.00 MiB (GPU 0; 31.75 GiB total capacity; 27.49 GiB already allocated; 619.56 MiB free; 30.10 GiB reserved in total by PyTorch)

### Torch 0.4.0 and 1.0.0

Let us then try to use the original PyTorch 0.4.0

When we install it, the torch module inside Python has torch.version.cuda=8.0.61.

Time to try and install PyTorch 1.0.0 with cuda 10.0…

Discussion:

* Have anyone checked fast ai implementaion for pytorch 1.0 ?  
  <https://github.com/fastai/fastai/blob/master/fastai/text/models/awd_lstm.py>
* You probably already know this by now, but just for everyone else who sees this: the fastai implementation works for PyTorch 1.0.
* You are right, it works, but it cannot reproduce the numbers in the paper either. I think that boat has sailed with Pytorch 0.4; at least until someone does a full hyperparameter search for 1.0.

There is a port of AWD-LSTM for PyTorch 1.0.0 at:

<https://github.com/manuvn/lpRNN-awd-lstm-lm>

### Out of Memory

Using the version I adapted to PyTorch 1.4.0 on the Lambda machine,

we arrive at:

end of epoch 149 | time: 85.88s | valid loss 4.40 | valid ppl 81.24 | valid bpc 6.344

-----------------------------------------------------------------------------------------

Saving Averaged!

| epoch 150 | 200/ 372 batches | lr 30.00000 | ms/batch 192.23 | loss 3.90 | ppl 49.39 | bpc 5.626

-----------------------------------------------------------------------------------

| end of epoch 150 | time: 85.90s | valid loss 4.40 | valid ppl 81.23 | valid bpc 6.344

-----------------------------------------------------------------------------------------

Saving Averaged!

| epoch 151 | 200/ 372 batches | lr 30.00000 | ms/batch 194.87 | loss 3.90 | ppl 49.16 | bpc 5.619

-----------------------------------------------------------------------------------

| end of epoch 151 | time: 85.93s | valid loss 4.40 | valid ppl 81.22 | valid bpc 6.344

-----------------------------------------------------------------------------------------

Saving Averaged!

| epoch 152 | 200/ 372 batches | lr 30.00000 | ms/batch 192.51 | loss 3.89 | ppl 49.02 | bpc 5.615

-----------------------------------------------------------------------------------

| end of epoch 152 | time: 85.85s | valid loss 4.40 | valid ppl 81.21 | valid bpc 6.344

then we get:

Traceback (most recent call last):

File "main.py", line 240, in <module>

train()

File "main.py", line 204, in train

loss.backward()

File "/home/andrealk3/.local/lib/python3.6/site-packages/torch/tensor.py", line 195, in backward

torch.autograd.backward(self, gradient, retain\_graph, create\_graph)

File "/home/andrealk3/.local/lib/python3.6/site-packages/torch/autograd/\_\_init\_\_.py", line 99, in backward

allow\_unreachable=True) # allow\_unreachable flag

RuntimeError: CUDA out of memory. Tried to allocate 804.00 MiB (GPU 0; 10.73 GiB total capacity; 7.87 GiB already allocated; 634.62 MiB free; 9.28 GiB reserved in total by PyTorch)

**A profiling method**

“In python, you can use the garbage collector’s book-keeping to print out the currently resident Tensors. Here’s a snippet that shows all the currently allocated Tensors:

*# prints currently alive Tensors and Variables*

**import** torch

**import** gc

**for** obj **in** gc.get\_objects():

**try**:

**if** torch.is\_tensor(obj) **or** (hasattr(obj, 'data') **and** torch.is\_tensor(obj.data)):

print(type(obj), obj.size())

**except**:

**pass**

“Thanks! Seems to work with a try: except block around it (some objects like shared libraries throw exception when you try to do hasattr on them)”

Then, from the discussion: <https://discuss.pytorch.org/t/how-to-debug-causes-of-gpu-memory-leaks/6741/10>

**Hypothesis: Variable-sized batches**

“To comment on your question, do you use variable-sized batches as input? In that case, **that might be caused by memory fragmentation (storages need to be re-allocated)…**”

“Something to consider with variable sized batches is that pytorch allocates memory for the batches as needed and doesn’t free them inline because the cost of calling garbage collection during the training loop is too high. **With variable batch sizes this can lead to multiple instances of the same buffer for the batch in memory**.

**If you make sure that your variably sized batches start with the largest batch then the initial memory allocated will be large enough to hold all batches** and you won’t have crazy memory growth. The natural instinct of most programmers is to do the opposite if they’re ordering, which means that the same buffer gets allocated multiple times over the course of training and never gets freed. Even if it’s random there’s still a lot of unnecessary allocation going on.

I ran into this with a language model with a random backprop through time window in it’s batching and was able to reduce the memory requirements by an order magnitude by forcing the first batch to be the largest.”

**Hypothesis: Variables not freed**

“ I must have figured out the source of the leak by the way.

It was due to the fact that significant portion of the code like variable allocation and intermediate computations was located within a single python function scope,

so I suspect that those intermediate variable were not marked as free even though they were not used anywhere further.

Putting a lot of del's kind of helped, but just isolating each individual step of computation into a separate function call so that all intermediate variable are automatically freed in the end of scope seems to be a better solution.

Does that sound reasonable in context of pytorch?”

“I’m happy you’ve resolved your memory issue - it’s a very useful observation you’ve made and it’s good it’s now here in public.

Indeed**, Python’s lack of block scoping can sometimes delay object destruction unnecessarily long.**

Actually, I’ve used dels myself recently for releasing buffers at the end of each iteration in a loop processing variable-sized data.

”

“Update 2:  
Finally I solved the memory problem! **I realized that in each iteration I put the input data in a new tensor, and pytorch generates a new computation graph. That causes the used RAM to grow forever. Then I use a placeholder tensor and copy the data to this tensor, and the RAM always stays at a low level** ”

## On the side: batching for GAT

Handling the GATs in batches instead of a for cycle, to make it faster for GRU\_GAT and also for awd-lstm when it will be added.

In a forward() call,

edge\_index\_g is a tensor of torch.Size([2, 8]), [ [sources],[destinations] ]  
(in this example: tensor([[1, 2, 3, 3, 3, 3, 3, 3],

[0, 0, 0, 4, 3, 3, 3, 3]]) )

GATConv uses the propagate() call of the MessagePassing base class…

*:obj:`edge\_index` holds the indices of a general (sparse)*

*assignment matrix of shape :obj:`[N, M]`.*

*If :obj:`edge\_index` is of type :obj:`torch.LongTensor`, its*

*shape must be defined as :obj:`[2, num\_messages]`,* ***where***

***messages from nodes in :obj:`edge\_index[0]` are sent to***

***nodes in :obj:`edge\_index[1]****`*

*(in case :obj:`flow="source\_to\_target"`).*

*If :obj:`edge\_index` is of type*

*:obj:`torch\_sparse.SparseTensor`,* ***its sparse indices***

***:obj:`(row, col)` should relate to :obj:`row = edge\_index[1]`***

***and :obj:`col = edge\_index[0]`****.*

*Hence, the only difference between those formats is that we*

*need to input the \*transposed\* sparse adjacency matrix into*

*:func:`propagate`.*

Since handling torch-sparse tensors is not obvious, I will execute a time analysis for the GRU\_GAT’s forward to confirm what is the slowest step.

note: operating on the mini-dataset.

**Time Analysis**:

* t1 - t0 = 2e-05
* **t2 - t1 = 0.04155** : **for** padded\_sequence **in** sequences\_in\_the\_batch\_ls: etc.
* t3 - t2 = 0.00089
* t4 - t3 = 0.00534

Inside:

* **t1 - t0 = 0.00431** :

padded\_sequence = padded\_sequence.squeeze()  
padded\_sequence = padded\_sequence.chunk(chunks=padded\_sequence.shape[0], dim=0)  
sequence\_lts = [Common.unpack\_input\_tensor(sample\_tensor, self.N) **for** sample\_tensor **in** padded\_sequence]

* **t2 - t1 = 0.00585** :  
  **for** ((x\_indices\_g, edge\_index\_g, edge\_type\_g), (x\_indices\_s, edge\_index\_s, edge\_type\_s)) **in** sequence\_lts: etc.

* t3 - t2 = 2e-05

### Unpacking the batchinput\_tensor

batchinput\_tensor has shape= (4,8,1150)

By executing

torch.chunk(batchinput\_tensor, chunks=batchinput\_tensor.shape[0], dim=0) ,

we obtain sequences\_in\_the\_batch\_ls, a list of 4 Tensors.

Then, we state:

**for** padded\_sequence **in** sequences\_in\_the\_batch\_ls:

squeeze()

padded\_sequence = padded\_sequence.chunk(chunks=padded\_sequence.shape[0], dim=0)

In the current example, I get 8 Tensors of shape (1,1150).

On these 8 Tensors, we execute:

sequence\_lts = [Common.unpack\_input\_tensor(sample\_tensor, self.N) **for** sample\_tensor **in** padded\_sequence]

There is no parallel map in Pytorch (unless I used DataParallel - maybe). However, I can see how numpy.apply\_along\_axis works. Maybe the parallelism is enough to offset the problem of moving to CPU and back to GPU.

After moving the unpacking of the input tensor to numpy, let us do a **Time Analysis** of the forward() call, operating on the whole WT-2 with batch\_size=40 and seq\_len=35.

With NumPy:

It can be either:

* t1 - t0 = 2.26592
* t2 - t1 = 1.80152
* t3 - t2 = 0.00162
* t4 - t3 = 0.02588

or:

* t1 - t0 = 4.09457
* t2 - t1 = 1.79964
* t3 - t2 = 0.00163
* t4 - t3 = 0.02692

Iteration time in the Training loop=

With the previous handling of torch.Tensors:

* t1 - t0 = 0.00032
* t2 - t1 = 5.08109
* t3 - t2 = 0.00162
* t4 - t3 = 0.02639

Iteration time in the Training loop=

The improvement is inconsistent. It can be practically negligible.

Let us examine the whole iteration to see the most time-expensive steps:

* t1 - t0 = 9.97688 # batch\_input, batch\_labels = train\_dataiter.\_\_next\_\_()
* t2 - t1 = 4.46099 # loss\_global, loss\_sense = compute\_model\_loss(model, batch\_input, batch\_labels, verbose)
* t3 - t2 = 0.02187 #
* t4 - t3 = 5.59976 # loss.backward()
* t5 - t4 = 0.00148 #

With the standard GRU, what are the times for an iteration in the training loop?

* t1 - t0 = 10.47801
* t2 - t1 = 2.72076
* t3 - t2 = 0.0008
* t4 - t3 = 1.22589
* t5 - t4 = 0.04444

The NumPy unpacking is marginally better.

### Full batching for GATConv

Now, would it be possible to handle the GATConv in a parallel way?

I can not operate directly with edge\_index tensors of different dimensions. For instance, stacking works with (2,8) and (2,8), not with (2,8) and (2,6).

However… I could pad edge\_index with self\_loop edges. They are removed by the MessagePassing module internally.

I know that the first index in the x\_indices is the node itself…

Yet another problem: I use the x\_indices to get the x (k,300) matrix that gets passed to the GAT network…

I can not stack them as they are. Because there is a different number of indices for each node.

Inside the GAT, x is used as:

x = torch.matmul(x, self.weight)

so I could pad it with zeros…

However, defining the inputs as 3D tensors with an additional batch dimension does not work.

One would have to use a Batch object, that reunites a list of graphs…

### Batches and DataParallel

Question: maybe I can reconcile batches and DataParallel…

”As we understood in this topic DataParallel does not work correct with RNN hidden states, when batch\_first=True. It is because batch\_first option affects only input and output of RNN, but not hidden states…

I can specify in DataParallel that the dimension along which the input is distributed is dim=1... note: this may end up being inconsistent with the input if we still serve it as (batch, sequence).

Alternatively, we can change the settings: batch\_first=True in the LSTM to receive (bsz, seq\_len, hid), and scatter on dim=0 in DataParallel.

Operating with the distributed\_batch\_size in the forward() instead of the original batch size.

However, this way I have multiple different hidden states for every copy of the model that resides on a different GPU.

Maybe:

“The hidden is being created for the entire model input (10000 in my case) where dataparallel is dividing that input by GPU count (4 in my case) to spread the load. Maybe we can also wrap the hidden input tensor with dataparallel so its also distributed correctly?”

Let us try PyTorch 1.5.0:

In the discussion “ [**GRU model learns very slowly when using DataParallel with multiple GPUs**](https://github.com/pytorch/pytorch/issues/33238) ”, someone reports that “I did some debugging, and the issue seems to be related to the caching of the flattened weights in the RNN introduced in PR [#27399](https://github.com/pytorch/pytorch/pull/27399). [on PyTorch 1.4.0]”. “This has been fixed in version 1.5.0, closing issue.”

Eventually installed PyTorch 1.5.0. and the corresponding torch-geometric.

Confirmed that DataParallel(MyGRU) and DataParallel(MyGRU\_GAT) have the same descent values when overfitting on fragment of WT-2 (see above in previous mini-experiments)

# Meeting with IA, 12/05

(Reordered) list:

* Rerun MultiSense Evaluation, using the new GRU in the architecture
  + select k=few eg. 5 globals, consider their senses, and choose among them.  
      
    question: I still need to open up to the senses’ logits; I would only get a distribution over the e.g. 20 senses from the k globals. Should I copy paste the logits over the senses’ logits, keeping everything at 0?   
    I can multiply the senses’logits (obtained, for instance, with a standard GRU side-architecture) per what I get from here. Everything not selected would be x0.
  + Compare with alternatives (e.g. Multi-sense alternatives)
* Standard Language Modeling:
  + AWD-LSTM implementation
  + Mogrifier LSTM – search for PyTorch implementation

# Multi-Sense LM

## Baseline 1: 2 GRUs, shared first layer

### Model

DataParallel(

(module): GRU\_base(

(maingru\_ls): ModuleList(

(0): GRU(300, 1150)

(1): GRU(1150, 1150)

(2): GRU(1150, 1150)

)

(gru\_senses): GRU(1150, 1150)

(linear2global): Linear(in\_features=1150, out\_features=21988, bias=True)

(linear2senses): Linear(in\_features=1150, out\_features=25986, bias=True))

)

Parameters:

('module.X', torch.Size([99963, 300]), torch.float32, True)

('module.select\_first\_indices', torch.Size([6]), torch.int64, False)

('module.embedding\_zeros', torch.Size([1, 300]), torch.float32, False)

('module.memory\_hn', torch.Size([3, 4, 1150]), torch.float32, False)

('module.memory\_hn\_senses', torch.Size([1, 4, 1150]), torch.float32, False)

('module.maingru\_ls.0.weight\_ih\_l0', torch.Size([3450, 300]), torch.float32, True)

('module.maingru\_ls.0.weight\_hh\_l0', torch.Size([3450, 1150]), torch.float32, True)

('module.maingru\_ls.0.bias\_ih\_l0', torch.Size([3450]), torch.float32, True)

('module.maingru\_ls.0.bias\_hh\_l0', torch.Size([3450]), torch.float32, True)

('module.maingru\_ls.1.weight\_ih\_l0', torch.Size([3450, 1150]), torch.float32, True)

('module.maingru\_ls.1.weight\_hh\_l0', torch.Size([3450, 1150]), torch.float32, True)

('module.maingru\_ls.1.bias\_ih\_l0', torch.Size([3450]), torch.float32, True)

('module.maingru\_ls.1.bias\_hh\_l0', torch.Size([3450]), torch.float32, True)

('module.maingru\_ls.2.weight\_ih\_l0', torch.Size([3450, 1150]), torch.float32, True)

('module.maingru\_ls.2.weight\_hh\_l0', torch.Size([3450, 1150]), torch.float32, True)

('module.maingru\_ls.2.bias\_ih\_l0', torch.Size([3450]), torch.float32, True)

('module.maingru\_ls.2.bias\_hh\_l0', torch.Size([3450]), torch.float32, True)

('module.gru\_senses.weight\_ih\_l0', torch.Size([3450, 1150]), torch.float32, True)

('module.gru\_senses.weight\_hh\_l0', torch.Size([3450, 1150]), torch.float32, True)

('module.gru\_senses.bias\_ih\_l0', torch.Size([3450]), torch.float32, True)

('module.gru\_senses.bias\_hh\_l0', torch.Size([3450]), torch.float32, True)

('module.linear2global.weight', torch.Size([21988, 1150]), torch.float32, True)

('module.linear2global.bias', torch.Size([21988]), torch.float32, True)

('module.linear2senses.weight', torch.Size([25986, 1150]), torch.float32, True)

('module.linear2senses.bias', torch.Size([25986]), torch.float32, True)

Number of trainable parameters=114.04M, where softmax=29.90M + 25.28M=55.19M, embeddings=29.99M, core = 28.86M

### Mini-experiment on fragment of SemCor

|  |  |  |
| --- | --- | --- |
| **Architecture** | **Input Signals** | **Hyperparameters** |
| For the globals:  **GRU** w/ 3 layers, 1150x3 (layer 1 shared).  Followed by **FF-NN** to logits | 1. The word embedding of the current global   (d=300) | batch\_size=4 |
| For the senses:  **1 shared layer** of the main GRU > **1-layer GRU** -> **FF-NN** to logits | TBPTT length=8 |
|  | learning rate=10^(**-3**) |

|  |  |  |
| --- | --- | --- |
| *Epoch* | *Training perplexity on globals* | *Training perplexity on senses* |
| 1 | 4918.23 | 10418.61 |
| 2 | 105.08 | 134.04 |
| 3 | 77.19 | 68.57 |
| 4 | 65.73 | 56.92 |
| 5 | 66.21 | 54.71 |
| 10 | 62.92 | 51.64 |
| 30 | 62.92 | 19.82 |
| 50 | 56.17 | 4.42 |
| 75 | 37.24 | 1.44 |
| 100 | 23.13 | 1.1 |
| 150 | 9.02 | 1.03 |
| 200 | 3.57 | 1.01 |

### Experiment –GRUs w/ shared layer on SemCor (fault: not detaching the gru\_senses’ memory)

|  |  |  |
| --- | --- | --- |
| **Architecture** | **Input Signals** | **Hyperparameters** |
| For the globals:  **GRU** w/ 3 layers, 1150x3 (layer 1 shared).  Followed by **FF-NN** to logits | 1. The word embedding of the current global   (d=300) | batch\_size=40 |
| For the senses:  **1 shared layer** of the main GRU > **1-layer GRU** -> **FF-NN** to logits | TBPTT length=35 |
|  | learning rate=0.5\*10^(**-4**) |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| *Epoch* | *Training perplexity on globals* | *Training perplexity on senses* |  | *Validation perplexity on globals* | *Validation perplexity on senses* |
| 1 | 971.96 | 5169.47 |  | 490.31 | 835.0 |
| 2 | 540.3 | 3432.66 |  | 347.28 | **804.95** |
| 3 | 418.94 | 3129.2 |  | 315.93 | 831.75 |
| 4 | 373.93 | 2947.42 |  | 299.93 | 829.01 |
| 5 | 344.06 | 2821.13 |  | 289.18 | 886.89 |
| 6 | 321.49 | 2768.63 |  | 283.92 | 1034.11 |
| 7 | 302.41 | 3269.6 |  | 278.56 | 914.73 |
| 8 | 285.49 | 2665.07 |  | 270.99 | 1807.85 |
| 9 | 270.04 | 2528.11 |  | 267.79 | 1112.29 |
| 10 | 256.36 | 2502.85 |  | 262.02 | 970.6 |
| 11 | 243.94 | 2391.66 |  | 263.61 | 1200.81 |
| 12 | 232.49 | 2373.34 |  | 258.92 | 1399.38 |
| 13 | 221.78 | 2252.45 |  | 252.47 | 2179.9 |
| 14 | 211.73 | 2222.98 |  | 251.66 | 1309.55 |
| 15 | 202.54 | 2070.02 |  | 250.55 | 2772.74 |
| 16 | 193.66 | 1950.52 |  | 247.19 | 2650.16 |
| 17 | 185.68 | 1891.93 |  | 248.58 | 3095.87 |
| 18 | 177.93 | 1835.95 |  | 247.61 | 2480.83 |
| 19 | 170.97 | 1809.4 |  | 245.7 | 1232.44 |
| 20 | 164.49 | 1725.47 |  | 245.27 | 2917.69 |
| 21 | 158.3 | 1641.77 |  | 244.83 | 1680.78 |
| 22 | 152.17 | 1589.13 |  | 245.41 | 2118.22 |
| 23 | 146.72 | 1569.21 |  | 244.53 | 2307.36 |
| 24 | 140.69 | 1497.27 |  | 245.31 | 1977.77 |
| 25 | 134.79 | 1427.68 |  | 244.14 | 3117.87 |
| 26 | 129.49 | 1348.3 |  | 243.1 | 2807.55 |
| 27 | 124.34 | 1311.9 |  | **242.35** | 2739.16 |
| 28 | 119.55 | 1289.07 |  | 243.71 | 3153.67 |
| 29 | 115.16 | 1264.97 |  | 244.83 | 3012.75 |
| 30 | 110.88 | 1209.65 |  | 245.65 | 3195.36 |

|  |  |  |
| --- | --- | --- |
| **Architecture** | **Input Signals** | **Hyperparameters** |
| For the globals:  **GRU** w/ 3 layers, 1150x3 (layer 1 shared).  Followed by **FF-NN** to logits | 1. The word embedding of the current global   (d=300) | batch\_size=40 |
| For the senses:  **1 shared layer** of the main GRU > **1-layer GRU** -> **FF-NN** to logits | TBPTT length=35 |
|  | **learning rate=10^(-5)** |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| *Epoch* | *Training perplexity on globals* | *Training perplexity on senses* |  | *Validation perplexity on globals* | *Validation perplexity on senses* |
| 1 | 1612.97 | 7651.04 |  | 522.04 | 1105.27 |
| 2 | 736.76 | 3579.03 |  | 510.6 | 1007.8 |
| 3 | 725.9 | 3486.44 |  | 510.74 | 983.15 |
| 4 | 723.02 | 3457.18 |  | 511.13 | **981.36** |
| 5 | 721.14 | 3443.05 |  | 508.15 | 991.13 |
| 6 | 720.06 | 3430.39 |  | 510.9 | 1001.09 |
| 7 | 719.66 | 3422.33 |  | 511.41 | 991.3 |
| 8 | 719.17 | 3410.85 |  | 508.73 | 1013.25 |
| 9 | 718.67 | 3410.02 |  | 511.87 | 1021.22 |
| 10 | 718.51 | 3407.2 |  | 508.25 | 1028.66 |
| 20 | 401.88 | 3258.57 |  | 309.43 | 1013.5 |
| 30 | 326.42 | 3116.9 |  | 274.1 | 1151.66 |
| 40 | 296.26 | 2955.15 |  | 267.23 | 1646.8 |
| 50 | 269.84 | 2696.74 |  | **263.68** | 1682.31 |
| 60 | 247.95 | 2453.59 |  | 264.04 | 2097.26 |

Observations:

a learning rate of 10^(-5) is too low. 10^(-4) is fast. Grid search suggests 0.5\*10^(-4)

I did not detach the senses’ memory!

### Experiment –GRUs w/ shared layer on SemCor

|  |  |  |
| --- | --- | --- |
| **Architecture** | **Input Signals** | **Hyperparameters** |
| For the globals:  **GRU** w/ 3 layers, 1150x3 (layer 1 shared).  Followed by **FF-NN** to logits | 1. The word embedding of the current global   (d=300) | batch\_size=40 |
| For the senses:  **1 shared layer** of the main GRU > **1-layer GRU** -> **FF-NN** to logits | TBPTT length=35 |
|  | learning rate=**0.5\*10^(-4)** |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| *Epoch* | *Training perplexity on globals* | *Training perplexity on senses* |  | *Validation perplexity on globals* | *Validation perplexity on senses* |
| 1 | 974.01 | 5184.52 |  | 495.82 | 835.31 |
| 2 | 543.19 | 3431.93 |  | 343.02 | **809.3** |
| 3 | 420.57 | 3115.14 |  | 316.4 | 830.73 |
| 4 | 375.15 | 2931.76 |  | 300.37 | 849.11 |
| 5 | 345.35 | 2806.64 |  | 289.39 | 919.22 |
| 6 | 322.07 | 2755.25 |  | 283.68 | 1344.15 |
| 7 | 302.68 | 2966.38 |  | 275.61 | 914.12 |
| 8 | 285.5 | 2683.41 |  | 269.84 | 1144.84 |
| 9 | 269.57 | 2614.12 |  | 266.23 | 1091.96 |
| 10 | 255.34 | 2522.42 |  | 260.59 | 1562.29 |
| 11 | 242.45 | 2525.91 |  | 262.01 | 1201.69 |
| 12 | 230.17 | 2335.81 |  | 255.8 | 3216.74 |
| 13 | 218.73 | 2203.31 |  | 249.31 | 1609.48 |
| 14 | 208.14 | 2198.41 |  | 248.36 | 2093.41 |
| 15 | 198.53 | 2077.28 |  | 247.43 | 1538.86 |
| 16 | 189.53 | 2036.01 |  | 242.89 | 2184.96 |
| 17 | 181.32 | 2029.97 |  | 243.98 | 1695.24 |
| 18 | 173.23 | 1921.61 |  | 241.94 | 2423.03 |
| 19 | 165.53 | 1774.17 |  | 241.25 | 2554.37 |
| 20 | 158.48 | 1748.05 |  | 241.11 | 1825.42 |
| 21 | 152.1 | 1682.55 |  | 240.31 | 2533.01 |
| 22 | 146.39 | 1696.8 |  | 239.08 | 1760.2 |
| 23 | 141.34 | 1531.6 |  | 239.15 | 2551.63 |
| 24 | 136.85 | 1468.94 |  | 240.69 | 3231.04 |
| 25 | 132.03 | 1453.37 |  | 239.0 | 2714.26 |
| 26 | 126.83 | 1346.67 |  | 238.2 | 3285.9 |
| 27 | 122.56 | 1297.54 |  | **237.42** | 2934.18 |
| 28 | 118.1 | 1255.5 |  | 238.0 | 2678.2 |
| 29 | 113.91 | 1216.16 |  | 239.22 | 3284.78 |
| 30 |  |  |  |  |  |

## Baseline 2 – 2GRUs (x3 and x2), no shared layers

### Model

DataParallel(

(module): GRU\_base2(

(maingru\_ls): ModuleList(

(0): GRU(300, 1150)

(1): GRU(1150, 1150)

(2): GRU(1150, 1150)

)

(gru\_senses): GRU(300, 1150, num\_layers=2)

(linear2global): Linear(in\_features=1150, out\_features=21988, bias=True)

(linear2senses): Linear(in\_features=1150, out\_features=25986, bias=True)

)

)

Number of trainable parameters=119.05M, where softmax=29.90M + 25.28M=55.19M, embeddings=29.99M, core = 33.87M

**Variant**: Including the sense node-state among the input signals:

DataParallel(

(module): GRU\_base2(

(maingru\_ls): ModuleList(

(0): GRU(600, 1150)

(1): GRU(1150, 1150)

(2): GRU(1150, 1150)

)

(gru\_senses): GRU(600, 1150, num\_layers=2)

(gat\_senses): GATConv(300, 75, heads=4)

(linear2global): Linear(in\_features=1150, out\_features=21988, bias=True)

(linear2senses): Linear(in\_features=1150, out\_features=25986, bias=True)

)

)

Number of trainable parameters=121.21M, where core=36.03M

### Experiment – 2 GRUs on SemCor

|  |  |  |
| --- | --- | --- |
| **Architecture** | **Input Signals** | **Hyperparameters** |
| For the globals:  **GRU** w/ 3 layers, 1150x3 (layer 1 shared).  Followed by **FF-NN** to logits | 1. The word embedding of the current global   (d=300) | batch\_size=40  TBPTT length=35 |
| For the senses:  **GRU** w/ 2 layers -> **FF-NN** to logits | learning rate=**0.5\*10^(-4)** |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| *Epoch* | *Training perplexity on globals* | *Training perplexity on senses* |  | *Validation perplexity on globals* | *Validation perplexity on senses* |
| 1 | 967.94 | 5149.14 |  | 494.29 | 828.62 |
| 2 | 527.79 | 3413.73 |  | 344.56 | **809.02** |
| 3 | 415.68 | 3060.8 |  | 314.78 | 876.36 |
| 4 | 373.02 | 2898.12 |  | 299.13 | 960.85 |
| 5 | 343.53 | 2741.66 |  | 286.95 | 1717.02 |
| 6 | 319.55 | 2697.53 |  | 279.82 | 906.0 |
| 7 | 298.75 | 2779.89 |  | 272.17 | 1035.97 |
| 8 | 280.14 | 2865.53 |  | 263.82 | 1036.54 |
| 9 | 263.37 | 2729.85 |  | 259.78 | 831.5 |
| 10 | 248.22 | 2986.64 |  | 253.18 | 1417.8 |
| 11 | 234.43 | 2479.41 |  | 252.78 | 1582.18 |
| 12 | 221.68 | 2397.1 |  | 246.32 | 1551.16 |
| 13 | 209.56 | 2695.38 |  | 239.0 | 881.0 |
| 14 | 198.55 | 2565.65 |  | 237.06 | 1204.42 |
| 15 | 188.63 | 2354.02 |  | 235.9 | 1339.91 |
| 16 | 179.54 | 2433.03 |  | 232.66 | 1072.53 |
| 17 | 171.34 | 2335.96 |  | 233.08 | 2045.94 |
| 18 | 163.51 | 2525.17 |  | 232.08 | 1157.81 |
| 19 | 156.38 | 2600.38 |  | 231.01 | 1704.28 |
| 20 | 149.69 | 2189.19 |  | 231.73 | 4222.49 |
| 21 | 143.11 | 2137.6 |  | 230.99 | 3757.52 |
| 22 | 136.94 | 2096.51 |  | 229.7 | 2110.82 |
| 23 | 131.19 | 1997.07 |  | **228.25** | 2498.45 |
| 24 | 125.67 | 1997.78 |  | 229.65 | 2904.29 |
| 25 | 120.53 | 1954.38 |  | 229.96 | 2206.7 |
| 26 | 115.94 | 1863.62 |  | 230.31 | 3183.38 |
| 27 | 111.5 | 1911.42 |  | 231.0 | 2748.2 |
| 28 | 107.35 | 1749.79 |  | 233.52 | 2585.29 |
| 29 | 103.34 | 1804.05 |  | 234.73 | 2605.26 |
| 30 | 99.47 | 1669.6 |  | 236.99 | 3488.63 |

The Valid-PPL on globals and senses is (228, 809). Better than the architecture that shares a layer, that reaches (237, 809). So there is no reason to share the first layer between the GRUs.

### Experiment – 2 GRUs on SemCor w/ sense-node input

|  |  |  |
| --- | --- | --- |
| **Architecture** | **Input Signals** | **Hyperparameters** |
| **GraphAttentionNetwork** to get the current sense’s node-state (when present. Otherwise, all 0s) | 1. The word embedding of the current global 2. The node state of the current sense   (d=600) | batch\_size=40 |
| For the globals:  **GRU** w/ 3 layers, 1150x3 (layer 1 shared).  Followed by **FF-NN** to logits | TBPTT length=35 |
| For the senses:  **GRU** w/ 2 layers -> **FF-NN** to logits | learning rate=**0.5\*10^(-4)** |
|  |  |

(starting the experiment to check where a bug is in SelectK – GruBase2 is ok, it’s due to DataParallel…)

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| *Epoch* | *Training perplexity on globals* | *Training perplexity on senses* |  | *Validation perplexity on globals* | *Validation perplexity on senses* |
| 1 |  |  |  |  |  |
| 2 |  |  |  |  |  |
| 3 |  |  |  |  |  |
| 4 |  |  |  |  |  |
| 5 |  |  |  |  |  |
| 6 |  |  |  |  |  |
| 7 |  |  |  |  |  |
| 8 |  |  |  |  |  |
| 9 |  |  |  |  |  |
| 10 |  |  |  |  |  |
| 11 |  |  |  |  |  |
| 12 |  |  |  |  |  |
| 13 |  |  |  |  |  |
| 14 |  |  |  |  |  |
| 15 |  |  |  |  |  |
| 16 |  |  |  |  |  |
| 17 |  |  |  |  |  |
| 18 |  |  |  |  |  |
| 19 |  |  |  |  |  |
| 20 |  |  |  |  |  |
| 21 |  |  |  |  |  |
| 22 |  |  |  |  |  |
| 23 |  |  |  |  |  |
| 24 |  |  |  |  |  |
| 25 |  |  |  |  |  |
| 26 |  |  |  |  |  |
| 27 |  |  |  |  |  |
| 28 |  |  |  |  |  |
| 29 |  |  |  |  |  |
| 30 |  |  |  |  |  |

## Structured prediction – select from K globals

### Design

Select k=5 globals, consider their senses, and choose among them.  
  
I still need to open up to the senses’ logits.

I get a distribution over the e.g. 20 senses from the k=5 globals.

I can multiply the senses’logits (obtained, for instance, with a standard GRU side-architecture) per what I get from here. Everything not selected would be x0.

Possibility: an alternative version may include the sense input.

For the sake of speed when retrieving the senses of the selected k globals:

we pass the graph\_area\_matrix as a parameter, to get the indices of the neighbouring nodes (since we are starting from globals, they will be either senses through the *sc* edges or other globals throught *synonyms/antonyms*.). We filter the neighbours to keep only the nodes whose index is in the senses’ range.

We obtain batch\_size\*sequence\_len\*k (e.g 4x8x5=160) tensors of variable size, containing the indices of the senses.

For every tensor containing the senses, we can apply X.index\_select(…) and get the sense embeddings.

We should get a probability distribution over these senses.

Idea #1: do not retrieve the embeddings. Assign 1 to the selected senses’indices and 0 to all others. Multiply per the softmax from the “other line”(the GRU). Then possibly scale up the probabilities that survived the filter so that they sum up to 1.

Problem: the nll\_loss works with log\_softmax, not with softmax.

Hypothesis a): assign 10^(-10) to the all the not-selected senses, instead of exactly 0 that would break the softmax and nll\_loss numerically. Then, rescale the softmax values that came from the relevant logits so that they sum up to 1.

**Hypothesis b)**: mask out the logits we don’t care for, and apply the softmax over the selected senses. Then, we will have to “make space” for all the 10^(-10) values, so we will have to subtract a small quantity δ from the selected values that we computed.

DataParallel(

(module): SelectK(

(maingru\_ls): ModuleList(

(0): GRU(300, 1150)

(1): GRU(1150, 1150)

(2): GRU(1150, 1150)

)

(gru\_senses): GRU(300, 1150, num\_layers=2)

(linear2global): Linear(in\_features=1150, out\_features=21988, bias=True)

(linear2senses): Linear(in\_features=1150, out\_features=25986, bias=True)

)

)

Number of parameters: 119.05M, where core=33.87M

### Mini-experiment – overfit on fragment of SemCor

|  |  |  |
| --- | --- | --- |
| **Architecture** | **Input Signals** | **Hyperparameters** |
| For the globals:  **GRU** w/ 3 layers, 1150x3 (layer 1 shared).  Followed by **FF-NN** to logits | 1. The word embedding of the current global   (d=300) | batch\_size=40  TBPTT length=35 |
| For the senses:  **GRU** w/ 2 layers -> **FF-NN** to logits  - > applying the softmax of the senses of the k most likely globals | learning rate=**0.5\*10^(-4)** |

I do not manage to overfit. Sense loss stuck at 13.82, sense perplexity at 10^(6).

Incidentally, I set all the senses that do not belong to the k globals to 10^(-6).

When we are at the start, the first predicted globals are random, and do not provide a useful indication.

However, in later stages of the training they will. Here, the possibility of a correct sense prediction depends on the quality of the prediction of globals.

Let us use k=5 and wait until the globals’ part of the model starts overfitting…

We have an error due to the retrieval of globals and indices.

If my current most likely global is ‘cotton’, the senses I retrieve should not contain ‘factor.n.2’.

AD.get\_node\_data(self.grapharea\_matrix\_lil, 8977, self.N, features\_mask=(**True**,**False**,**False**))

-> tensor([ 8977, 56951, 83334, 32288, 8972, 8973, 8974, 8976, 8975, 35033,

34527])

We should remember that we are retrieving nodes 2 steps away, in a maximum number of 32.

Through the following edges, we can get the following node types:

sc -> senses; sc + def -> definitions; sc + ex -> examples; syn, ant -> globals; syn/ant + sc -> senses.

Since: EP.get\_globalword\_fromindex(32288-self.last\_idx\_senses) = ‘factor’ {global}

we are retrieving its senses.

However, the synonyms of ‘cotton’ , in the processed\_synonyms.h5 archive, are:

6386 cotton\_fiber

6387 cotton\_wool

6388 cotton\_plant

It is time to apply a sanity check to the graph, and the nodes we retrieve from the grapharea\_matrix.

## Graph check

Node: 8977 in the vocabulary of globals (cotton)

GA.get\_indices\_area\_toinclude(graph\_dataobject.edge\_index, graph\_dataobject.edge\_type, 8977, area\_size=32, max\_hops=1)

We get: ([8977(n: index of the global itself here), 56951, 83334, 32288],)

Remember that the X matrix has the sections:

**senses**: (0, 25986). **globals**: (25986, 47974). definitions: (47974, 73960). examples: (73960, 120602)

56951-47974=definition 8977. factor.v.03 consider as relevant when making a decision

83334-73960=example 9374. factor.v.03 You must factor in the recent developments

32288-25986=global 6302. factor 35

Why am I retrieving ‘factor’? What is the mistake here?

Let us add self.last\_idx\_senses to the node number I am retrieving…

This time,

GA.get\_indices\_area\_toinclude(graph\_dataobject.edge\_index, graph\_dataobject.edge\_type, 8977+25986{=34963}, 32, 1)

node\_indices\_ls, all\_edges\_retrieved\_ls =

([**34963**, 5673, 5672, 5671, 5674, *5670*], [57659, 57660, 57661, 57662, 57663])

**34963-**25986: global 8977 (‘cotton’)

Senses: 5670 to 5674: ﻿costume.v.02, cotton.n.01, cotton.n.02, cotton.n.03, cotton.n.04.

*Note*: it should be 5675, cotton.v.01 not 5670-costume.v.02. Must add 1. When I create the graph, maybe?

From the indices\_table.db, I get:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| word\_sense | vocab\_index | start\_defs | end\_defs | start\_examples | end\_examples |
| cotton.n.01 | 5670 | 5670 | 5671 | 5821 | 5821 |
| cotton.n.02 | 5671 | 5671 | 5672 | 5821 | 5821 |
| cotton.n.03 | 5672 | 5672 | 5673 | 5821 | 5821 |
| cotton.n.04 | 5673 | 5673 | 5674 | 5821 | 5821 |
| cotton.v.01 | 5674 | 5674 | 5675 | 5821 | 5822 |

When I execute get\_node\_edges, I get:

<class 'list'>: [(34963, 5673, 2), (34963, 5672, 2), (34963, 5671, 2), (34963, 5674, 2), (34963, 5670, 2)]

Let us review the most likely global and the senses that we retrieve for it in the SelectK architecture.

### Senses.SelectK retrieval and selection check

In Epoch 1, in the very first batch, at the start of the training process on mini-fragment:

(the initial most likely globals depend on the random initialization of the main GRU)

**INFO : sample: 0; k\_globals=['microorganisms']**

*INFO : sample: 1; k\_globals=['unscrupulous']*

…

**INFO : sample: 16; k\_globals=['shrink']**

*INFO : sample: 17; k\_globals=['unscrupulous']*

…

**# sample 0 is missing, since the plural form ‘microorganisms’ has no senses**.

*INFO : Sample: 1; selected\_senses=['unscrupulous.a.01']*

…

**INFO : Sample: 16; selected\_senses=['shrink.v.03', 'shrink.v.05', 'shrink.v.04', 'reduce.v.15', 'reduce.v.02', 'reduce.v.14', 'reduce.v.11', 'reduce.v.20', 'reduce.v.09', 'reduce.v.08', 'reduce.v.05', 'reduce.v.04', 'reduce.v.03', 'reduce.v.01', 'reduce.v.13', 'reduce.v.06', 'reduce.v.18']**

*INFO : Sample: 17; selected\_senses=['unscrupulous.a.01']*

From Epoch 2 to epoch… 161 and counting (globals train-PPL @161: 25.97, descending…) … epoch :

The most likely global is either ‘”’,‘the’ or ‘<unk>’ that have no senses.

I do not manage to get relevant globals to be the first one.

I use the verbose log of the predictions every 10 epochs, thus training faster (on GPUs) for 300 epochs.

Training, end of epoch 289. Global step n.1734. Time = 1180.27.

INFO : Perplexity: Globals perplexity=1.08 Sense perplexity=1383.87

Training, end of epoch 290. Global step n.1740. Time = 1182.03. The training losses are:

INFO : Perplexity: Globals perplexity=1.1 Sense perplexity=1979.8

|  |  |  |
| --- | --- | --- |
| Globals predicted | Senses predicted | Comment |
| Label: the next global is: said(from 1)  INFO : Label: the next sense is: state.v.01(from 22232)  INFO : The top- 5 predicted globals are:  INFO : Word: said ; probability = 92.4%  INFO : Word: further ; probability = 2.03%  INFO : Word: to ; probability = 1.34%  INFO : Word: thanks ; probability = 0.72%  INFO : Word: the ; probability = 0.54%  INFO : The top- 5 predicted senses are:  INFO : |  | Since “said” has no senses to select, we do not find any.  (We find only “say” in the vocabulary) |
| Label: the next global is: Friday(from 2)  INFO : Label: the next sense is: None(from -1)  INFO : The top- 5 predicted globals are:  INFO : Word: Friday ; probability = 82.86%  INFO : Word: it ; probability = 3.96%  INFO : Word: conducted ; probability = 2.76%  INFO : Word: , ; probability = 1.7%  INFO : Word: Only ; probability = 1.64%  INFO : |  |  |
| Label: the next global is: an(from 3)  INFO : Label: the next sense is: None(from -1)  INFO : The top- 5 predicted globals are:  INFO : Word: an ; probability = 91.28%  INFO : Word: the ; probability = 4.16%  INFO : Word: which ; probability = 1.6%  INFO : Word: . ; probability = 0.66%  INFO : Word: evidence ; probability = 0.36%  INFO : |  |  |
| Label: the next global is: investigation(from 4)  INFO : Label: the next sense is: probe.n.01(from 17882)  INFO : The top- 5 predicted globals are:  INFO : Word: investigation ; probability = 89.53%  INFO : Word: considering ; probability = 1.04%  INFO : Word: a ; probability = 0.97%  INFO : Word: won ; probability = 0.88%  INFO : Word: Only ; probability = 0.82% | INFO : The top- 5 predicted senses are:  INFO : Sense: probe.n.01 ; probability = 11.23%  INFO : Sense: investigate.v.01 ; probability = 10.91%  INFO : Sense: investigation.n.02 ; probability = 10.91%  INFO : Sense: investigate.v.02 ; probability = 10.86%  INFO : Sense: probe.n.03 ; probability = 10.82% |  |
| of Atlanta s |  |  |
| Label: the next global is: recent(from 8)  INFO : Label: the next sense is: late.s.03(from 13363)  INFO : The top- 5 predicted globals are:  INFO : Word: recent ; probability = 80.44%  INFO : Word: charge ; probability = 5.42%  INFO : Word: registration ; probability = 3.93%  INFO : Word: act ; probability = 1.94%  INFO : Word: handful ; probability = 1.29% | INFO : The top- 5 predicted senses are:  INFO : Sense: late.a.01 ; probability = 12.44%  INFO : Sense: late.r.03 ; probability = 12.3%  INFO : Sense: late.a.06 ; probability = 12.21%  INFO : Sense: late.s.03 ; probability = 12.18%  INFO : Sense: late.r.01 ; probability = 12.17% | we do not manage to focus on the correct sense… we are just picking the senses of the most likely global. |
|  |  |  |

Training epoch n.295: Perplexity: Globals perplexity=1.09 Sense perplexity=1712.68

Training epoch n.296: Perplexity: Globals perplexity=1.08 Sense perplexity=1776.37

Training epoch n.297: Perplexity: Globals perplexity=1.07 Sense perplexity=1786.04

Training epoch n.298: Perplexity: Globals perplexity=1.06 Sense perplexity=2161.16

Training epoch n.299: Perplexity: Globals perplexity=1.06 Sense perplexity=1197.16

Training epoch n.300: Perplexity: Globals perplexity=1.08 Sense perplexity=1712.68

### Modifications to SelectK

When the globals have already gone into overfit on the fragment, the sense perplexity is oscillating around 1100 and 2100, but it’s still very high.

We can try several modifications.

#1: Since “said” has no senses to select, we do not find any, even if there is a sense label – here, state.v.01

We can lemmatize ‘said’ (into ‘say’), and get the senses again.

#2: It may happen that even the lemmatized form has no senses or does not change. This happens with phrases, like ‘full of’.

We may decide to send a tensor of full-zeros, and ignore that sense label. This may or may not require modyfing the condition for ignoring the label, from a tensor with [0] to a full-zeros… it does not, because we do not explicitly ignore that label, we just exploit the fact that a uniform full-zero logsoftmax gives no gradient in the nll\_loss.

Working example of lemmatization:

sample\_k\_indices=[32671] -> [‘arches’] -> [‘arch’(6687+25986=32673)] -> we get sense\_neighbours\_t = tensor([ 1135, 1136, 1134, 1131, 1130, 1133, 1129, 1132, 25588])

The corresponding senses we extract are:

0 = {str} 'arch.s.03'

1 = {str} 'arch.v.01'

2 = {str} 'arch.s.02'

3 = {str} 'arch.n.03'

4 = {str} 'arch.n.02'

5 = {str} 'arch.s.01'

6 = {str} 'arch.n.01'

7 = {str} 'arch.n.04'

8 = {str} 'wicked.a.01'

We are picking, erroneously, one of arch’s syonyms:

arch.n.01 arch

arch.n.02 arch

arch.n.03 arch

arch.n.03 archway

arch.n.04 arch

arch.s.01 patronize

arch.s.01 patronise

arch.s.01 arch

arch.s.01 condescend

arch.s.02 arch

arch.s.03 pixilated

arch.s.03 puckish

arch.s.03 prankish

arch.s.03 arch

arch.s.03 wicked

arch.s.03 impish

arch.s.03 implike

arch.s.03 mischievous

arch.v.01 arch

arch.v.01 arc

arch.v.01 curve

Where does 25588 come from? It’s on the 2nd hop.

In order to avoid pulling the senses of synonyms (and antonyms), we need to use a 1-hop graph and graph area matrix for this particular use case.

Later on I will send the grapharea\_matrix with 1 hop specifically for this purpose while using the 2-hops version for the model input. For now, I can just use that one as the current grapharea matrix of the experiment

### Bugs & co

Current problems:

segfault only on GPU

git version

The segfault:

lib/python3.6/site-packages/tables/.libs/**libhdf5**-933c8d2d.so.103.0.0

In particular, we get:

Thread 38 "python3" received signal SIGSEGV, Segmentation fault.

[Switching to Thread 0x7ffd7b7ad700 (LWP 25678)]

0x00007fff53a8aab8 in H5C\_protect ()

from /home/andrealk3/venvs/torch15/lib/python3.6/site-packages/tables/.libs/libhdf5-933c8d2d.so.103.0.0

Problem:

we moved the model to DataParallel. HDF5 does not reliably allow for multithread access, or multiprocess for that matter.

New error: corrupted double-linked list.

Hypothesis 1:

“Some googling told me that apparently this one is due to a kernel debugging option:

CONFIG\_DEBUG\_STACK\_USAGE

and that the message is generally benign.”

Hypothesis 2:

“The two likely causes I can see are:

1) Writing into a block after it is freed.

2) A buffer overrun in a memory block into an adjacent freed block.”

On Cheetah we do not get this error. It is due to DataParallel-ism and replication.

Parallel reading on HDF5 strikes again. From the stacktrace:

k\_globals\_words = [EP.get\_globalword\_fromindex(global\_relative\_idx) for global\_relative\_idx in k\_globals\_relative\_indices]

File "/home/andrealk3/ALMA\_PhD/Code/TASK1\_Gbwe/GNN/ExplorePredictions.py", line 19, in get\_globalword\_fromindex

globals\_vocabulary\_df = pd.read\_hdf(globals\_vocabulary\_fpath, mode='r')

…’

tables.exceptions.HDF5ExtError: Problems reading records.

### Mini-experiment – overfit SelectK on a fragment of SemCor

# GRUs on WT-2

Aims: - verify that the new, unified, flag-based GRU model works.’

- try again to get results on WT-2, this time using the *correct* vocabulary of globals  
 instead of the one from SemCor

## Experiments

### Mini-experiment 1 – Overfit on fragment of WT-2

|  |  |  |
| --- | --- | --- |
| **Architecture** | **Input Signals** | **Hyperparameters** |
| **GRU** with 3 layers (1150 x3) | 1) The word embedding of the current global (d=300) | batch\_size=4 |
|  | TBPTT length=8 |
| Followed by linear2Globals **FF-NN** | learning rate=10^(-4) |

|  |  |
| --- | --- |
| **Epoch** | **Training perplexity** |
| 1 |  |
| 2 |  |
| 3 |  |
| 4 |  |
| 5 |  |
| 10 |  |
| 30 |  |
| 50 |  |
| 75 |  |
| 100 |  |
| 125 |  |
| 150 |  |