[Preliminary information (Datasets, graphs) 3](#_Toc42333462)

[WikiText-2 3](#_Toc42333463)

[WikiText-2 Stats 3](#_Toc42333464)

[Graph 3](#_Toc42333465)

[Meetings 4](#_Toc42333466)

[IA 4](#_Toc42333467)

[IA, PSK, KF 4](#_Toc42333468)

[Attempt to include the whole AWD-LSTM implementation 5](#_Toc42333469)

[Input facilities 5](#_Toc42333470)

[Training loop 5](#_Toc42333471)

[Corpus 6](#_Toc42333472)

[Batch & Input data 7](#_Toc42333473)

[Embeddings used by the model 8](#_Toc42333474)

[Hypothesis: use X and the indices of the globals 9](#_Toc42333475)

[Adjusting awd-lstm for the current PyTorch setup (1.4.0) 10](#_Toc42333476)

[WeightDrop 10](#_Toc42333477)

[Optimizer error 11](#_Toc42333478)

[Weights can be flattened in memory 11](#_Toc42333479)

[Torch 0.4.0 and 1.0.0 12](#_Toc42333480)

[Out of Memory 12](#_Toc42333481)

[Meeting with IA, 12/05 13](#_Toc42333482)

[GRU\_s on WT-2 – bugs in the model were not solved 15](#_Toc42333483)

[Model Verification and Modification 21](#_Toc42333484)

[The loss when operating on senses 21](#_Toc42333485)

[Considerations 22](#_Toc42333486)

[Graph retrieval 23](#_Toc42333487)

[Boundaries of graph node types 23](#_Toc42333488)

[Analysis… 23](#_Toc42333489)

[Senses’ +1 displacement 25](#_Toc42333490)

[Adding lemmatization for the global node 26](#_Toc42333491)

[GRU\_s on WT-2 – version 2 27](#_Toc42333492)

[Baseline GRU 27](#_Toc42333493)

[Mini-experiment – Overfit on fragment of WT-2 27](#_Toc42333494)

[Experiment – GRU on WT-2 27](#_Toc42333495)

[Model 28](#_Toc42333496)

[Experiment 28](#_Toc42333497)

[Experiment – GRU\_GAT on WT-2 29](#_Toc42333498)

[Model 29](#_Toc42333499)

[Experiment 30](#_Toc42333500)

[LSTM on WT-2 31](#_Toc42333501)

[Model 31](#_Toc42333502)

[AWD-LSTM on WT-2 33](#_Toc42333503)

[Baseline AWD-LSTM 33](#_Toc42333504)

[Model 33](#_Toc42333505)

[Experiment: unmodified AWD-LSTM on WT-2 33](#_Toc42333506)

[Hypotheses for CUDAOutOfMemory 34](#_Toc42333507)

[Further runs to try to solve CUDAOutOfMemory 35](#_Toc42333508)

[Analysis of tensors in GPU memory 36](#_Toc42333509)

# 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]**)

# 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. (or 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? The corpus also contains ids = torch.LongTensor(tokens) )

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 #(but the argument dropouth is 0.2…?)

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>

**Hypotheses: Variable-sized batches, Variables not freed**

(See later on)

# 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

# GRU\_s on WT-2 – bugs in the model were not solved

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

**GRU baseline**

Hyperparameters: \_batchPerSeqlen32\_area32\_lr0.0001\_epochs200

Model:

DataParallel(

(module): GRU\_base2(

(maingru\_ls): ModuleList(

(0): GRU(300, 1150)

(1): GRU(1150, 1150)

(2): GRU(1150, 1150)

)

(linear2global): Linear(in\_features=1150, out\_features=31640, bias=True)

)

)

Parameters:

('module.X', torch.Size([114324, 300]), torch.float32, True)

('module.select\_first\_indices', torch.Size([6]), torch.float32, 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([2, 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.linear2global.weight', torch.Size([31640, 1150]), torch.float32, True)

('module.linear2global.bias', torch.Size([31640]), torch.float32, True)

Number of trainable parameters=91.61M

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

|  |  |  |
| --- | --- | --- |
| **Architecture** | **Input Signals** | **Hyperparameters** |
| model: GRUbase2 | 1) The word embedding of the current global (d=300) | batch\_size=4 |
| **GRU** with 3 layers (1150 x3) | TBPTT length=8 |
| Followed by linear2Globals **FF-NN** | learning rate=10^(-4) |
|  |  | grapharea=32, hops=2 |

|  |  |  |  |
| --- | --- | --- | --- |
| having a torch.nn.ModuleList as the main GRU | | having a torch.nn.GRU with 3 layers as the main GRU | |
| **Epoch** | **Training perplexity** | **Epoch** | **Training perplexity** |
| 1 | 21225.43 | 1 | 19812.91 |
| 2 | 676.41 | 2 | 658.97 |
| 3 | 164.19 | 3 | 161.62 |
| 4 | 114.26 | 4 | 114.18 |
| 5 | 106.45 | 5 | 106.13 |
| 10 | 103.9 | 10 | 103.88 |
| 30 | 100.38 | 30 | 100.36 |
| 50 | 103.95 | 50 | 103.88 |
| 75 | 50.13 | 75 | 54.49 |
| 100 | 1.5 | 100 | 14.05 |
| 125 | 1.19 | 125 | 2.74 |
| 150 | 1.09 | 150 | 1.42 |
|  |  | 175 | 1.23 |
|  |  | 200 | 1.17 |

**Predictions**:

|  |  |  |
| --- | --- | --- |
| Label: the next global is: of(from 37) | INFO : The top- 5 predicted globals are:  INFO : Word: , ; probability = 30.93%  INFO : Word: and ; probability = 21.57%  INFO : Word: on ; probability = 15.78%  INFO : Word: **of ; probability = 6.86%**  INFO : Word: up ; probability = 1.67% |  |
| Label: the next global is: the(from 31) | INFO : Word: **the ; probability = 43.55%**  INFO : Word: <unk> ; probability = 30.32%  INFO : Word: effective ; probability = 10.05%  INFO : Word: Vice ; probability = 2.65%  INFO : Word: two ; probability = 2.41% |  |
| Label: the next global is: 5th(from 1367) | INFO : Word: Grand ; probability = 35.39%  INFO : Word: <unk> ; probability = 14.6%  INFO : Word: British ; probability = 7.63%  INFO : Word: High ; probability = 6.23%  INFO : Word: vanguard ; probability = 6.14% | maybe because ‘5th’ is not in the WT-2 vocabulary? |
| Label: the next global is: Battle(from 301) | INFO : Word: **Battle ; probability = 88.74%**  INFO : Word: Squadron ; probability = 2.36%  INFO : Word: , ; probability = 1.15%  INFO : Word: battleships ; probability = 1.09%  INFO : Word: British ; probability = 0.95% |  |
| Label: the next global is: Squadron(from 3815) | INFO : Word: **Squadron ; probability = 98.58%**  INFO : Word: of ; probability = 0.36%  INFO : Word: , ; probability = 0.34%  INFO : Word: Battle ; probability = 0.18%  INFO : Word: were ; probability = 0.07% |  |
| Label: the next global is: fired(from 2726) | INFO : Word: **fired ; probability = 42.36%**  INFO : Word: , ; probability = 18.18%  INFO : Word: . ; probability = 9.15%  INFO : Word: approaching ; probability = 7.1%  INFO : Word: for ; probability = 3.1% | quite low. |
| Label: the next global is: on(from 122) | INFO : Word: **on ; probability = 94.87%**  INFO : Word: . ; probability = 1.23%  INFO : Word: fired ; probability = 0.91%  INFO : Word: a ; probability = 0.52%  INFO : Word: and ; probability = 0.51% |  |
| Label: the next global is: the(from 31) | INFO : Word: **the ; probability = 96.7%**  INFO : Word: Markgraf ; probability = 0.72%  INFO : Word: a ; probability = 0.42%  INFO : Word: by ; probability = 0.39%  INFO : Word: at ; probability = 0.28% |  |
| Label: the next global is: leading(from 764) | INFO : Word: **leading ; probability = 89.1%**  INFO : Word: of ; probability = 3.19%  INFO : Word: Scouting ; probability = 0.99%  INFO : Word: German ; probability = 0.88%  INFO : Word: , ; probability = 0.8% |  |
| … |  |  |

**Experiment – baseline GRU on WT-2**

|  |  |  |
| --- | --- | --- |
| **Architecture** | **Input Signals** | **Hyperparameters** |
| model: GRUbase2 | 1) The word embedding of the current global (d=300) | batch\_size=40 |
| **GRU** with 3 layers (1150 x3) | TBPTT length=35 |
| Followed by linear2Globals **FF-NN** | learning rate=5\*10^(-5) |
|  |  | grapharea=32, hops=2 |

|  |  |  |
| --- | --- | --- |
| *Epoch* | *Training perplexity* | *Validation perplexity* |
| 1 | 980.12 | 510.7 |
| 2 | 533.51 | 364.06 |
| 3 | 397.31 | 307.28 |
| 4 | 326.09 | 277.26 |
| 5 | 281.21 | 259.88 |
| 6 | 248.77 | 248.28 |
| 7 | 223.57 | 240.2 |
| 8 | 203.15 | 231.76 |
| 9 | 185.98 | 226.92 |
| 10 | 171.43 | 222.4 |
| 11 | 158.79 | 217.97 |
| 12 | 147.7 | 214.07 |
| 13 | 137.84 | 210.25 |
| 14 | 129.01 | 208.36 |
| 15 | 121.11 | 206.09 |
| 16 | 113.89 | 204.26 |
| 17 | 107.25 | 201.24 |
| 18 | 101.14 | 199.73 |
| 19 | 95.5 | 198.94 |
| 20 | 90.28 | 198.72 |
| 21 | 85.5 | **198.26** |
| 22 | 81.04 | 198.89 |
| 23 | 76.93 | 199.67 |
| 24 | 73.12 | 200.12 |
| 25 |  |  |

**GRU + graph input**

Hyperparameters: \_batchPerSeqlen32\_area32\_lr0.0001\_epochs200

Model:

DataParallel(

(module): GRU\_base2(

(main\_gru): GRU(600, 1150, num\_layers=3)

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

(linear2global): Linear(in\_features=1150, out\_features=31640, bias=True)

)

)

Parameters:

('module.X', torch.Size([114324, 300]), torch.float32, True)

('module.select\_first\_indices', torch.Size([6]), torch.float32, 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([2, 4, 1150]), torch.float32, False)

('module.main\_gru.weight\_ih\_l0', torch.Size([3450, 600]), torch.float32, True)

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

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

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

('module.main\_gru.weight\_ih\_l1', torch.Size([3450, 1150]), torch.float32, True)

('module.main\_gru.weight\_hh\_l1', torch.Size([3450, 1150]), torch.float32, True)

('module.main\_gru.bias\_ih\_l1', torch.Size([3450]), torch.float32, True)

('module.main\_gru.bias\_hh\_l1', torch.Size([3450]), torch.float32, True)

('module.main\_gru.weight\_ih\_l2', torch.Size([3450, 1150]), torch.float32, True)

('module.main\_gru.weight\_hh\_l2', torch.Size([3450, 1150]), torch.float32, True)

('module.main\_gru.bias\_ih\_l2', torch.Size([3450]), torch.float32, True)

('module.main\_gru.bias\_hh\_l2', torch.Size([3450]), torch.float32, True)

('module.gat\_globals.weight', torch.Size([300, 300]), torch.float32, True)

('module.gat\_globals.att', torch.Size([1, 4, 150]), torch.float32, True)

('module.gat\_globals.bias', torch.Size([300]), torch.float32, True)

('module.linear2global.weight', torch.Size([31640, 1150]), torch.float32, True)

('module.linear2global.bias', torch.Size([31640]), torch.float32, True)

Number of trainable parameters=92.73M

**Mini-experiment on fragment of WT-2**

|  |  |  |
| --- | --- | --- |
| **Architecture** | **Input Signals** | **Hyperparameters** |
| model: GRUbase2 | 1. The word embedding of the current global 2. The node-state of the current global   (d=600) | batch\_size=4 |
| **GAT** for the global node | TBPTT length=8 |
| **GRU** with 3 layers (1150 x3) | learning rate=10^(-4) |
| Followed by linear2Globals **FF-NN** |  | grapharea=32, hops=2 |

|  |  |
| --- | --- |
| **Epoch** | **Training perplexity** |
| 1 | 19187.21 |
| 2 | 603.54 |
| 3 | 165.15 |
| 4 | 115.35 |
| 5 | 106.69 |
| 10 | 103.95 |
| 30 | 100.4 |
| 50 | 84.1 |
| 75 | 6.68 |
| 100 | 1.45 |
| 125 | 1.22 |
| 150 |  |
| 175 |  |
| 200 |  |

**Experiment – GRU + GAT global node input on WT-2**

|  |  |  |
| --- | --- | --- |
| **Architecture** | **Input Signals** | **Hyperparameters** |
| model: GRUbase2 | 1. The word embedding of the current global 2. The node state of the current global   (d=600) | batch\_size=40 |
| **GAT** for the global node | TBPTT length=35 |
| **GRU** with 3 layers (1150 x3) | learning rate=5\*10^(-5) |
| Followed by linear2Globals **FF-NN** |  | grapharea=32, hops=2 |

|  |  |  |
| --- | --- | --- |
| *Epoch* | *Training perplexity* | *Validation perplexity* |
| 1 | 917.77 | 437.5 |
| 2 | 467.87 | 328.78 |
| 3 | 364.01 | 290.23 |
| 4 | 309.4 | 268.26 |
| 5 | 272.61 | 254.68 |
| 6 | 244.56 | 245.6 |
| 7 | 222.06 | 239.36 |
| 8 | 203.33 | 232.52 |
| 9 | 187.27 | 228.59 |
| 10 | 173.38 | 224.73 |
| 11 | 158.79 |  |
| 12 | 150.07 | 216.69 |
| 13 | 140.18 | 212.62 |
| 14 | 131.22 | 210.16 |
| 15 | 123.13 | 207.87 |
| 16 | 115.7 | 205.8 |
| 17 | 108.88 | 203.11 |
| 18 | 102.66 | 201.59 |
| 19 | 96.95 | 200.72 |
| 20 | 91.67 | 200.51 |
| 21 | 86.81 (baseline: 85.5) | **199.87** (baseline: 198.26) |
| 22 | 82.21 | 200.34 |
| 23 |  |  |
| 24 |  |  |
| 25 |  |  |

# Model Verification and Modification

## The loss when operating on senses

We launch GRUbase2 on a fragment of SemCor, and examine what happens when we compute the loss and after we

call loss.backward()

We set batch\_size=1, sequence\_length=1…

batch\_labels = tensor([[ 2, -1]])

Shape of the batch\_input tensor: torch.Size([1, 1, 1150])

The model, at the start of the forward:

(objective: the weights of the gru\_senses should *not* change)

self.memory\_hn.shape=(3,1,600)

At the start, it’s a tensor made entirely of zeros.

self.memory\_hn.nonzero()=tensor([],…)

The same for self.memory\_hn\_senses. Shape=(3,1,600), .nonzero()=tensor([])

We examine

self.main\_gru.weight\_ih\_l1.mean()=tensor(-4.3736e-05, grad\_fn=<MeanBackward0>)

self.main\_gru.weight\_hh\_l2.mean()=tensor(-8.4784e-06, grad\_fn=<MeanBackward0>)

, that we expect to change, and

self.gru\_senses.weight\_ih\_l1.mean()=tensor(-1.2207e-05, grad\_fn=<MeanBackward0>)

self.gru\_senses.weight\_hh\_l2.mean()=tensor(-2.3060e-05, grad\_fn=<MeanBackward0>)

that we expect to remain the same.

x\_indices\_g= tensor([ 1, 47975, 34783, 0, 32616])

edge\_index\_g= tensor([[1, 2, 2, 2, 2, 4, 2],

[0, 3, 0, 2, 2, 2, 4]])

edge\_type\_g= tensor([0, 2, 2, 3, 3, 3, 3])

x\_indices\_s= tensor([22232, 70206, 96175, 96176, 96177, 96178, 26158, 22228, 22229, 22230, 22227, 22231, 36926, 26852, 28113, 34096, 29324, 29292, 37538, 26756,

30394, 29242

edge\_index\_s= tensor([[ 1, 2, 3, 4, 5, 6, 6, 6, 6, 6, 6, 17, 6, 19, 6, 15, 6, 6,

12, 6, 6, 6, 6, 6, 6, 6, 18, 6, 16, 6, 20, 6, 17, 6, 6, 6,

6, 6, 6, 6, 13, 6, 14, 21, 6],

[ 0, 0, 0, 0, 0, 10, 7, 8, 9, 11, 0, 6, 17, 6, 19, 6, 15, 12,

6, 6, 6, 6, 6, 6, 6, 18, 6, 16, 6, 20, 6, 17, 6, 6, 6, 6,

6, 6, 6, 13, 6, 14, 6, 6, 21]])

edge\_type\_s= tensor([0, 1, 1, 1, 1, 2, 2, 2, 2, 2, 2, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3,

3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3])

We compute the logits for both globals and senses, but only the predictions\_globals should encounter a meaningful label in the next step…

loss\_global= tensor(10.0134, grad\_fn=<NllLossBackward>)

Then, we have:

loss\_sense = tfunc.nll\_loss(predictions\_senses, batch\_labels\_senses, ignore\_index=-1)

with predictions\_senses.shape=(1,25986) and batch\_labels\_senses=tensor([-1])

loss\_sense= tensor(0., grad\_fn=<NllLossBackward>)

Then, in the training loop we execute:

batch\_sense\_tokens = (batch\_labels.t()[1][batch\_labels.t()[1]!=-1].shape[0])  
sum\_epoch\_loss\_sense = sum\_epoch\_loss\_sense + loss\_sense.item() \* batch\_sense\_tokens

epoch\_senselabeled\_tokens = epoch\_senselabeled\_tokens + batch\_sense\_tokens

And

loss = loss\_global + loss\_sense

loss.backward()

Now,

model.main\_gru.weight\_ih\_l1.mean()=tensor(-4.3115e-05, grad\_fn=<MeanBackward0>)

(from -4.3736e-05)

model.gru\_senses.weight\_ih\_l1.mean()=tensor(-1.2207e-05, grad\_fn=<MeanBackward0>)

(from -1.2207e-05) # unchanged, as expected.

### Considerations

The gru\_senses does not “see” anything related to the processing of the text word-after-word, as instead the main\_gru does.

It sees the current “snapshot” of input signals (whether that be the word embedding alone or input from the graph). gru\_senses adjusts itself and then waits “dormant” for a number of words, until the next sense label comes along.

Thus, gru\_senses has no understanding of the text / of the language. It proceeds as follows:

Label: the next global is: <unk>(from 21987) (from: Fulton\_County\_Grand\_Jury)

Label: the next sense is: group.n.01(from 10898)

INFO : batch\_labels=tensor([[21987, 10898]])

INFO : loss\_sense=tensor(10.1451, grad\_fn=<NllLossBackward>)

Label: the next global is: said(from 1)

INFO : Label: the next sense is: state.v.01(from 22232)

INFO : batch\_labels=tensor([[ 1, 22232]])

INFO : loss\_sense=tensor(10.1698, grad\_fn=<NllLossBackward>)

Label: the next global is: Friday(from 2)

INFO : Label: the next sense is: None(from -1)

INFO : batch\_labels=tensor([[ 2, -1]])

INFO : loss\_sense=tensor(0., grad\_fn=<NllLossBackward>)

Label: the next global is: an(from 3)

INFO : Label: the next sense is: None(from -1)

INFO : batch\_labels=tensor([[ 3, -1]])

INFO : loss\_sense=tensor(0., grad\_fn=<NllLossBackward>)

Label: the next global is: investigation(from 4)

INFO : Label: the next sense is: probe.n.01(from 17882)

INFO : batch\_labels=tensor([[ 4, 17882]])

INFO : loss\_sense=tensor(10.2071, grad\_fn=<NllLossBackward>)

## Graph retrieval

### Boundaries of graph node types

(n: node types’ boundaries are at: 25986, 47974, 73960)

**Senses: (0, 25986). Globals: (25986, 47974). Definitions: (47974, 73960). Examples: (73960, end)**

### Analysis…

Let us examine:

x\_indices\_g= tensor([ 1, 47975, 34783, 0, 32616])

# Likely mistake: the 1st global has not been made a relative index to get the correct row from the X matrix.

x\_indices\_s= tensor([22232, 70206, 96175, 96176, 96177, 96178, 26158, 22228, 22229, 22230, 22227, 22231, 36926, 26852, 28113, 34096, 29324, 29292, 37538, 26756,

30394, 29242])

When I am starting to read the SemCor fragment, how do I retrieve nodes?

The DataLoader calls:

get\_forwardinput\_forelement(global\_idx, sense\_idx, grapharea\_matrix, area\_size)

with:

global\_idx=0, sense\_idx=-1, grapharea\_matrix=…, area\_size=32

**if** (sense\_idx == -1): we use all torch.zeros(…).

Let us see the globals:

area\_x\_indices\_global, edge\_index\_global, edge\_type\_global = AD.get\_node\_data(grapharea\_matrix, global\_idx, area\_size)

Inside that function:

*# Accessing sparse matrix. Everything was shifted +1, so now: we ignore 0 ; we shift -1; we get the data*nodes\_ls =list(map(**lambda** value: value - 1, filter(**lambda** num: num != 0, grapharea\_matrix[i, 0:k].todense().tolist()[0])))

and nodes\_ls=<class 'list'>: [0.0, 47974.0, 34783.0, 1.0, 32616.0].

Which nodes did we get from the matrix?

0: 0th sense

47974: 47974-47974=0th definition

34783: 34783-25986= 8797th global

1: 1st sense

32616: 32616-25986=6630th global

Let us review what we get from the graph, instead of the graph matrix. Starting node: 0.

GA.get\_indices\_area\_toinclude(graph\_dataobject.edge\_index, graph\_dataobject.edge\_type, node\_index=0, area\_size=32, max\_hops=1):

nodes\_queue\_at\_current\_level=[0]

node\_edges=<class 'list'>: [(47974, 0, 0), (34783, 0, 2)]

nodes\_queue\_at\_next\_hop=we add 47974 and 34783

In the end, node\_indices\_ls, all\_edges\_retrieved\_ls=([0, 47974, 34783], [0, 51989])

Issue #1: if we are extracting the 1st global, we should not operate on index 0, but on the index (start\_of\_globals+0).

Examining: NI.get\_tokens\_tpls(…)

When we are in the dataloader’s \_\_getitem\_\_(self, index):

current\_token\_tpl=<class 'tuple'>: (0, -1)

next\_token\_tpl=<class 'tuple'>: (21987, 10898)

So the 2nd is correct, but the 1st is not.

What happens when token\_dict={'surface\_form': 'said', 'lemma': 'say', 'pos': 'VBD', 'wn16\_key': 'say%2:32:00::', 'wn30\_key': 'say%2:32:00::'}?

wordnet\_sense=state.v.01 ; From querying the SQL indices\_table.db, we get: sense\_index\_queryresult=<class 'tuple'>: (22232,)

For the global,

global\_absolute\_index = Utils.select\_from\_hdf5(globals\_vocabulary\_h5, **'vocabulary'**, [**'word'**], [word]).index[0]

I took out the addition of the last\_sense\_index…

global\_index = global\_absolute\_index *# + last\_idx\_senses; do not add this to globals, or we go beyond the n\_classes*

For the label purpose, the consideration is correct. However, it is necessary to add that term for the purpose of retrieving graph data and rows of X.

Therefore, in the current version, global\_index=1.

next\_token\_tpl=<class 'tuple'>: (1, 22232)

Whereas the current\_token\_tpl was correct: <class 'tuple'>: (21987, 10898)

Then we go on:

{'surface\_form': 'Friday', 'lemma': 'friday', 'pos': 'NNP', 'wn16\_key': 'friday%1:28:00::', 'wn30\_key': 'friday%1:28:00::'}

The sense is friday.n.01… The global\_absolute\_index is 2.

So we have:

current\_token\_tpl=<class 'tuple'>: (1, 22232) , next\_token\_tpl=<class 'tuple'>: (2, -1)

the current tuple, in the meantime, leads us to get\_node\_data with i=1.

nodes=tensor([ 1(sense!), 47975(def), 34783(global), 0(sense), 32616(global)])

edgeindex.T=tensor([[1, 0],

[2, 3], [2, 0],

[2, 2], [2, 2], [4, 2], [2, 4]])

edgetype=tensor([0, 2, 2, 3, 3, 3, 3]) (edge types are: 0=defs, 1=examples, 2=SenseChildren, 3=synonyms, 4=antonyms). (n: considering that here we may be from the point of view of the sense=1, instead of the global 25987)

What are the nodes we got?

“1”: It should have been the global: said, but this is sense n.1=ab.n.04

47975: 47975-47974= definition n. 1= from ab.n.04, the blood group whose red cells carry both the A and B antigens

34783: 34783-25986=8797th global= 8797 word=ab frequency=3

0: Sense n.0 = ab.n.02

32616: 32616-25986= 6630th global= 6630 word=abdominal frequency=3

While the token\_tuple that is also used for the labels and can use the “absolute” global index, the retrieval of data from the graph must use the **relative global index** (i.e. + last\_sense\_idx).

If I retrieve what I should, i.e. the node index 25986+1, from the graph:

nodes, edges=([25987], [77976])

The only node is the global n.1 in the vocabulary of globals:

“said”.

Since it’s not “say”, it does not have any connection in the graph. We should apply lemmatization before attempting node retrieval.

“say” is the global n. 2127.

If we execute get\_indices\_area\_toinclude(edge\_index, edge\_type, node\_index*=25986+2127=28113*, area\_size, max\_hops):

Using the node itself (during the 1st iteration, at hop=1)

node\_edges=<class 'list'>: [

(28113, 20173, 2),

(28113, 20178, 2),

(28113, 20177, 2),

(28113, 20175, 2),

(28113, 20176, 2),

(28113, 20174, 2),

(26631, 28113, 3), (28113, 26631, 3),

(28113, 26158, 3), (26158, 28113, 3),

(28113, 29596, 3), (29596, 28113, 3)

(28113, 43985, 3), (43985, 28113, 3),

(28113, 26599, 3), (26599, 28113, 3),

(28113, 28113, 3)]

Senses: say.n.01, say.v.07, say.v.08, say.v.09, say.v.10, say.v.11

***~~Note~~***~~: I am still with a displacement of 1. This displacement could be due to <unk>… I will have to check sense 20173 and what I retrieve from it…~~

Global synonyms – to find them, subtract 25968:

[645:order, 172:state, 3610:suppose, 17999:aforesaid, 613:read]

Moreover, now that we are able to make the correct request to the graph, I decide to operate with 1 hop of distance as the starting hyperparameter.

### Senses’ +1 displacement

Lastly, we review the senses’ retrieval:

when we have global n.1 (‘said’), we will operate with:

area\_x\_indices\_sense: tensor([

22232: sense ~~state.n.06 (or so we get from consulting indices\_table.sql)~~ state v.01,

70206: 22232nd definition (state.v.01 – express in words),

96175: 22215th example (state.v.01 – He said that he wanted to marry her),

96176: 22216th example (state.v.01 – tell me what is bothering you),

96177: “” state your opinion, 96178: “” state your name,

26158: 172nd global “state”,

22228: sense – from consulting: state.n.02,

22229: sense – from consulting: state.n.03,

22230: sense – from consulting: state.n.04,

22227: sense – from consulting: state.n.01,

22231: sense – from consulting: state.n.06,

36926: 10940th global “province”,

26852: 866th global “tell”, # I still have 2 hops in the graph area in this particular experiment, so it’s: sense->global->synonym.

28113: 2127th global “say”,

34096: 8110th global “express”,

etc. 29324, 29292, 37538, 26756, 30394, 29242])

This may be just due to the row-counting in indices\_table.sql.

The rows that we visualize in the table go from 1 to 25986.

When my sense-node index is 0, I actually refer to row 1 in the table.

(I adjust the meaning of the indices above)

## Adding lemmatization for the global node

If we encounter a word like ‘said’, it would be informative to include the graph-input-signal from the global node ‘say’.

# GRU\_s on WT-2 – version 2

Now that we retrieve nodes from the graph correctly – adding the number of senses to obtain the relative global index – we apply again the GRUs on WikiText-2.

Even when we used the word embedding alone, we retrieved the i-th sense instead of the i-th global.

The impact on the definition of the graph\_area was even greater.

Note – maybe I should modify this to follow more closely AWD-LSTM:

Using an LSTM instead of a GRU, with a structure:

WeightDrop(

(module): LSTM(400, 1150)

), WeightDrop(

(module): LSTM(1150, 1150)

), WeightDrop(

(module): LSTM(1150, 400)

)]

This, however, is not for this experiment.

## Baseline GRU

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

|  |  |  |
| --- | --- | --- |
| **Architecture** | **Input Signals** | **Hyperparameters** |
| model: GRUbase2 | 1) The word embedding of the current global (d=300) | batch\_size=4 |
| **GRU** with 3 layers (1150 x3) | TBPTT length=8 |
| Followed by linear2Globals **FF-NN** | learning rate=10^(-4) |
|  |  | grapharea=32, hops=1 |

|  |  |
| --- | --- |
| **Epoch** | **Training perplexity** |
| 1 | 23356.04 |
| 2 | 710.8 |
| 3 | 165.06 |
| 4 | 115.16 |
| 5 | 106.56 |
| 10 | 103.86 |
| 30 | 90.56 |
| 50 | 18.31 |
| 75 | 1.6 |
| 100 | 1.22 |
|  |  |
|  |  |

## Experiment – GRU on WT-2

### Model

Hyperparameters: \_batchPerSeqlen1400\_area32\_lr5e-05\_epochs100

Model:

DataParallel(

(module): GRU\_base2(

(main\_gru): GRU(300, 1150, num\_layers=3)

(linear2global): Linear(in\_features=1150, out\_features=31640, bias=True)

)

)

Parameters:

('module.X', torch.Size([114324, 300]), torch.float32, True)

('module.select\_first\_indices', torch.Size([6]), torch.float32, False)

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

('module.memory\_hn', torch.Size([3, 40, 1150]), torch.float32, False)

('module.memory\_hn\_senses', torch.Size([3, 40, 1150]), torch.float32, False)

('module.main\_gru.weight\_ih\_l0', torch.Size([3450, 300]), torch.float32, True)

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

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

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

('module.main\_gru.weight\_ih\_l1', torch.Size([3450, 1150]), torch.float32, True)

('module.main\_gru.weight\_hh\_l1', torch.Size([3450, 1150]), torch.float32, True)

('module.main\_gru.bias\_ih\_l1', torch.Size([3450]), torch.float32, True)

('module.main\_gru.bias\_hh\_l1', torch.Size([3450]), torch.float32, True)

('module.main\_gru.weight\_ih\_l2', torch.Size([3450, 1150]), torch.float32, True)

('module.main\_gru.weight\_hh\_l2', torch.Size([3450, 1150]), torch.float32, True)

('module.main\_gru.bias\_ih\_l2', torch.Size([3450]), torch.float32, True)

('module.main\_gru.bias\_hh\_l2', torch.Size([3450]), torch.float32, True)

('module.linear2global.weight', torch.Size([31640, 1150]), torch.float32, True)

('module.linear2global.bias', torch.Size([31640]), torch.float32, True)

Number of trainable parameters=91.61M

### Experiment

|  |  |  |
| --- | --- | --- |
| **Architecture** | **Input Signals** | **Hyperparameters** |
| model: GRUbase2 | 1) The word embedding of the current global (d=300) | batch\_size=40 |
| **GRU** with 3 layers (1150 x3) | TBPTT length=35 |
| Followed by linear2Globals **FF-NN** | learning rate=  a) 5\*10^(-5) |
|  |  | grapharea=32, hops=**1** |

a)

|  |  |  |
| --- | --- | --- |
| *Epoch* | *Training perplexity* | *Validation perplexity* |
| 1 | 917.58 | 444.2 |
| 2 | 451.33 | 307.77 |
| 3 | 333.13 | 260.04 |
| 4 | 273.15 | 233.46 |
| 5 | 233.87 | 216.22 |
| 6 | 205.2 | 204.28 |
| 7 | 183.27 | 196.26 |
| 8 | 165.72 | 188.44 |
| 9 | 151.08 | 183.93 |
| 10 | 138.69 | 179.87 |
| 11 | 127.92 | 176.08 |
| 12 | 118.48 | 173.24 |
| 13 | 110.07 | 170.46 |
| 14 | 102.54 | 169.14 |
| 15 | 95.77 | 167.83 |
| 16 | 89.6 | 166.66 |
| 17 | 83.95 | 164.9 |
| 18 | 78.76 | 164.16 |
| 19 | 73.98 | **163.77** |
| 20 | 69.56 | 163.98 |
| 21 | 65.52 | 163.82 |
| 22 | 61.74 | 164.49 |
| 23 | 58.24 | 165.57 |
| 24 |  |  |
| 25 |  |  |

## Experiment – GRU\_GAT on WT-2

### Model

DataParallel(

(module): GRU\_base2(

(main\_gru): GRU(**600**, 1150, num\_layers=3)

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

(linear2global): Linear(in\_features=1150, out\_features=31640, bias=True)

))

Parameters:

('module.X', torch.Size([114324, 300]), torch.float32, True)

('module.select\_first\_indices', torch.Size([6]), torch.float32, False)

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

('module.memory\_hn', torch.Size([3, 40, 1150]), torch.float32, False)

('module.memory\_hn\_senses', torch.Size([3, 40, 1150]), torch.float32, False)

('module.main\_gru.weight\_ih\_l0', torch.Size([3450, 600]), torch.float32, True)

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

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

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

('module.main\_gru.weight\_ih\_l1', torch.Size([3450, 1150]), torch.float32, True)

('module.main\_gru.weight\_hh\_l1', torch.Size([3450, 1150]), torch.float32, True)

('module.main\_gru.bias\_ih\_l1', torch.Size([3450]), torch.float32, True)

('module.main\_gru.bias\_hh\_l1', torch.Size([3450]), torch.float32, True)

('module.main\_gru.weight\_ih\_l2', torch.Size([3450, 1150]), torch.float32, True)

('module.main\_gru.weight\_hh\_l2', torch.Size([3450, 1150]), torch.float32, True)

('module.main\_gru.bias\_ih\_l2', torch.Size([3450]), torch.float32, True)

('module.main\_gru.bias\_hh\_l2', torch.Size([3450]), torch.float32, True)

('module.gat\_globals.weight', torch.Size([300, 300]), torch.float32, True)

('module.gat\_globals.att', torch.Size([1, 4, 150]), torch.float32, True)

('module.gat\_globals.bias', torch.Size([300]), torch.float32, True)

('module.linear2global.weight', torch.Size([31640, 1150]), torch.float32, True)

('module.linear2global.bias', torch.Size([31640]), torch.float32, True)

Number of trainable parameters=92.73M

### Experiment

Hyperparameters: \_batchPerSeqlen1400\_area32\_lr5e-05\_epochs100

|  |  |  |
| --- | --- | --- |
| **Architecture** | **Input Signals** | **Hyperparameters** |
| model: GRUbase2 | 1. The word embedding of the current global 2. The node state of the current global   (d=600) | batch\_size=40 |
| **GAT** for the global node | TBPTT length=35 |
| **GRU** with 3 layers (1150) | learning rate=5\*10^(-5) |
| Followed by linear2Globals **FF-NN** |  | grapharea=32, hops=1 |

a)

|  |  |  |
| --- | --- | --- |
| *Epoch* | *Training perplexity* | *Validation perplexity* |
| 1 | 846.51 | 400.6 |
| 2 | 415.18 | 290.41 |
| 3 | 312.81 | 249.34 |
| 4 | 259.61 | 225.86 |
| 5 | 223.77 | 210.13 |
| 6 | 196.83 | 198.9 |
| 7 | 175.88 | 190.97 |
| 8 | 159.01 | 183.3 |
| 9 | 144.89 | 178.81 |
| 10 | 132.93 | 174.89 |
| 11 | 122.53 | 171.08 |
| 12 | 113.39 | 168.39 |
| 13 | 105.24 | 165.68 |
| 14 | 97.91 | 164.44 |
| 15 | 91.33 | 163.04 |
| 16 | 85.29 | 161.92 |
| 17 | 79.77 | 160.48 |
| 18 | 74.69 | 159.93 |
| 19 | 70.01 | **159.68** |
| 20 | 65.66 | 160.13 |
| 21 | 61.66 | 160.27 |
| 22 |  |  |
| 23 |  |  |
| 24 |  |  |
| 25 |  |  |

Improvement compared to GRU without GAT and the global’s node-state: from 163.8 to 159.7 Valid PPL.

# LSTM on WT-2

## Model

We may (**or may not**, as specified later) use the same network architecture as the one used in AWD-LSTM – albeit without WeightDrop or any special modification.

From:

[WeightDrop(

(module): LSTM(400, 1150)

), WeightDrop(

(module): LSTM(1150, 1150)

), WeightDrop(

(module): LSTM(1150, 400)

)]

Since for now we are operating with FastText embeddings, the initial dimension will be 300 instead of 400. We may use 768 later, when using pretrained BERT embeddings.

And if we add other input signals? Then the initial dimension d will increase accordingly (e.g. from 300 to 600)

… **however**, the original AWD-LSTM also has a decoder= Linear(in\_features=1150, out\_features=33278, bias=True)

So why do we have 400 as the output dimension of the last RNN layer? I must check this out.

raw\_outputs = []

outputs = []

for l, rnn in enumerate(self.rnns):

current\_input = raw\_output

rnn.flatten\_parameters()

raw\_output, new\_h = rnn(raw\_output, hidden[l])

new\_hidden.append(new\_h)

raw\_outputs.append(raw\_output)

if l != self.nlayers - 1:

#self.hdrop(raw\_output)

raw\_output = self.lockdrop(raw\_output, self.dropouth)

outputs.append(raw\_output)

…

hidden = new\_hidden

output = self.lockdrop(raw\_output, self.dropout)

outputs.append(output)

result = output.view(output.size(0)\*output.size(1), output.size(2))

return result, hidden, [#and possibly] raw\_outputs, outputs

# where:

output.shape=torch.Size([71, 80, 400])

result.shape=torch.Size([5680, 400])

raw\_outputs is a list of 3 tensors, of sizes (71,80,1150), (71,80,1150), (71,80,400)

In splitcross.py (approximate softmax followed by CrossEntropyLoss):

weight.shape=(33278,400)

bias.shape=(33278,)

hiddens.shape=(5680,400)

targets.shape=(5680,)

and decoder and encoder weight are tied.

This confirms that the decoder to the logits for the softmax expands from 400, and a separate decode was used only in the version without tied weights, and can be considered a leftover from it.

This confirms that the structure used in AWD-LSTM was:

LSTM\_L0: (400,1150) > LSTM\_L1: (1150,1150) > LSTM\_L2: (1150,400) > Linear (400, |V|)

**However**, they narrowed the final dimension to 400 because they have tied weights.

We do not need to lose information.

We can use either 1150 > 1150 > 575,

or maybe something different, taking into account the fact that we do not have DropConnect.

Since our aim is only to have a as-good-as-possible baseline for GRU+GAT,

we can also choose 3 layers, all 900, with a dropout of 0.1 for regularization.

## Experiments

### Mini-experiment: LSTM on fragment of WT-2

# AWD-LSTM on WT-2

## Baseline AWD-LSTM

### Model

We use all the default hyperparameters for the Averaged-SGD WeightDropped LSTM.

Minor necessary modifications:

* weight\_drop.py, since Pytorch 0.4 and >=1.0 handle the renaming of parameters used in the DropConnect differently.  
  *AttributeError: 'LSTM' object has no attribute 'weight\_hh\_l0'*Instead of renaming, our code in the class ForwardWithDrop uses get and \_\_setitem\_\_
* torch.cuda.empty\_cache in the training loop to avoid a RuntimeError: CudaOutOfMemory

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

### Experiment: unmodified AWD-LSTM on WT-2

In the 1st run, it goes all the way to:

| **epoch 147** | 200/ 372 batches | lr 30.00000 | ms/batch 196.17 | loss 3.96 | **ppl 52.30** | bpc 5.709

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

| end of epoch 147 | time: 81.43s | valid loss 4.39 | **valid ppl 80.83** | valid bpc 6.337

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

Until it meets:

Traceback (most recent call last):

File "main.py", line 241, in <module>

train()

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

**loss.backward()**

File "/home/andrealk3/venvs/torch15/lib/python3.6/site-packages/torch/tensor.py", line 198, in backward

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

File "/home/andrealk3/venvs/torch15/lib/python3.6/site-packages/torch/autograd/\_\_init\_\_.py", line 100, in backward

allow\_unreachable=True) # allow\_unreachable flag

**RuntimeError: CUDA out of memory. Tried to allocate 814.00 MiB (GPU 0; 10.73 GiB total capacity; 8.07 GiB already allocated; 798.62 MiB free; 9.11 GiB reserved in total by PyTorch)** (malloc at /pytorch/c10/cuda/CUDACachingAllocator.cpp:289)

frame #0: c10::Error::Error(c10::SourceLocation, std::string const&) + 0x46 (0x7fe378e91536 in /home/andrealk3/venvs/torch15/lib/python3.6/site-packages/torch/lib/libc10.so)

…

I already tried uncommenting the line:

*# 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)

## Hypotheses for CUDAOutOfMemory

**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** ”

### Further runs to try to solve CUDAOutOfMemory

**2nd run:**

**Hypothesis tested**: “Python’s lack of block scoping can sometimes delay object destruction unnecessarily long.”

**Modifications**: I added the

training\_loop\_iteration(i, hidden, total\_loss, batch, start\_time)

function to encapsulate an iteration of the training loop.

**Result**: none. We still encounter CUDA-OOM after 147 epochs, after the model has switched to the ASGD optimizer at epoch 39.

| epoch 147 | 200/ 372 batches | lr 30.00000 | ms/batch 151.36 | loss 3.96 | ppl 52.30 | bpc 5.709

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

| end of epoch 147 | time: 63.96s | valid loss 4.39 | valid ppl 80.83 | valid bpc 6.337

RuntimeError: CUDA out of memory. Tried to allocate 814.00 MiB (GPU 0; 10.73 GiB total capacity; 8.07 GiB already allocated; 550.62 MiB free; 9.35 GiB reserved in total by PyTorch)

**3rd run:**

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

Moreover:

total\_loss **+=** loss

Here, total\_loss is accumulating history across your training loop, since loss is a differentiable variable with autograd history.

You can fix this by writing total\_loss += float(loss) instead.

**Modifications**:

* I add:  
  **del** data  
  **del** targets  
  at the end of the (original) training loop.
* I add one of the versions of the profiling code for GPU memory leaks mentioned in the discussion:  
  <https://discuss.pytorch.org/t/how-to-debug-causes-of-gpu-memory-leaks/6741/11>  
  At the end of each epoch (after we call train() in the for-cycle in main.py), we empty the cache and print the tensors currently alive.
* I turn   
  total\_loss += raw\_loss.data  
  into   
  total\_loss += raw\_loss.data.clone().detach()  
  following the PyTorch FAQs for Cuda-OutOfMemory at <https://pytorch.org/docs/stable/notes/faq.html#my-model-reports-cuda-runtime-error-2-out-of-memory>

**Result**:

It still goes OOM, just a few epochs later:

@: | end of epoch 150 | time: 64.33s | valid loss 4.39 | valid ppl 80.79 | valid bpc 6.336

Traceback (most recent call last):

File "main.py", line 243, in <module>

train()

File "main.py", line 205, in train

loss.backward()

File "/home/andrealk3/venvs/torch15/lib/python3.6/site-packages/torch/tensor.py", line 198, in backward

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

File "/home/andrealk3/venvs/torch15/lib/python3.6/site-packages/torch/autograd/\_\_init\_\_.py", line 100, in backward

allow\_unreachable=True) # allow\_unreachable flag

RuntimeError: CUDA out of memory. Tried to allocate 782.00 MiB (GPU 0; 10.73 GiB total capacity; 8.09 GiB already allocated; 778.62 MiB free; 9.13 GiB reserved in total by PyTorch) (malloc at /pytorch/c10/cuda/CUDACachingAllocator.cpp:289)

### Analysis of tensors in GPU memory

It is also necessary to examine what is the name of each tensor…

At the end of Epoch 1:

| epoch 1 | 200/ 372 batches | lr 30.00000 | ms/batch 148.75 | loss 7.81 | ppl 2469.21 | bpc 11.270

(<class 'torch.nn.parameter.Parameter'>, torch.Size([4600, 1150]))

(<class 'torch.nn.parameter.Parameter'>, torch.Size([4600, 1150]))

(<class 'torch.nn.parameter.Parameter'>, torch.Size([1600, 400]))

(<class 'torch.Tensor'>, torch.Size([26107, 80]))

(<class 'torch.Tensor'>, torch.Size([21764, 10]))

(<class 'torch.Tensor'>, torch.Size([245569, 1]))

(<class 'torch.Tensor'>, torch.Size([2088628]))

(<class 'torch.Tensor'>, torch.Size([217646]))

(<class 'torch.Tensor'>, torch.Size([245569]))

(<class 'torch.nn.parameter.Parameter'>, torch.Size([33278]))

(<class 'torch.nn.parameter.Parameter'>, torch.Size([1600]))

(<class 'torch.nn.parameter.Parameter'>, torch.Size([1600]))

(<class 'torch.nn.parameter.Parameter'>, torch.Size([1600, 400]))

(<class 'torch.nn.parameter.Parameter'>, torch.Size([1600, 1150]))

(<class 'torch.nn.parameter.Parameter'>, torch.Size([4600]))

(<class 'torch.nn.parameter.Parameter'>, torch.Size([4600]))

(<class 'torch.nn.parameter.Parameter'>, torch.Size([4600, 1150]))

(<class 'torch.nn.parameter.Parameter'>, torch.Size([4600, 1150]))

(<class 'torch.nn.parameter.Parameter'>, torch.Size([4600]))

(<class 'torch.nn.parameter.Parameter'>, torch.Size([4600]))

(<class 'torch.nn.parameter.Parameter'>, torch.Size([4600, 1150]))

(<class 'torch.nn.parameter.Parameter'>, torch.Size([4600, 400]))

(<class 'torch.nn.parameter.Parameter'>, torch.Size([33278, 400]))

/home/andrealk3/venvs/torch15/lib/python3.6/site-packages/torch/distributed/distributed\_c10d.py:102: UserWarning: torch.distributed.reduce\_op is deprecated, please use torch.distributed.ReduceOp instead

warnings.warn("torch.distributed.reduce\_op is deprecated, please use "

Named parameters of the model:

('encoder.weight', torch.Size([33278, 400]), torch.float32, True)

('rnns.0.weight\_ih\_l0', torch.Size([4600, 400]), torch.float32, True)

('rnns.0.weight\_hh\_l0', torch.Size([4600, 1150]), torch.float32, True)

('rnns.0.bias\_ih\_l0', torch.Size([4600]), torch.float32, True)

('rnns.0.bias\_hh\_l0', torch.Size([4600]), torch.float32, True)

('rnns.1.weight\_ih\_l0', torch.Size([4600, 1150]), torch.float32, True)

('rnns.1.weight\_hh\_l0', torch.Size([4600, 1150]), torch.float32, True)

('rnns.1.bias\_ih\_l0', torch.Size([4600]), torch.float32, True)

('rnns.1.bias\_hh\_l0', torch.Size([4600]), torch.float32, True)

('rnns.2.weight\_ih\_l0', torch.Size([1600, 1150]), torch.float32, True)

('rnns.2.weight\_hh\_l0', torch.Size([1600, 400]), torch.float32, True)

('rnns.2.bias\_ih\_l0', torch.Size([1600]), torch.float32, True)

('rnns.2.bias\_hh\_l0', torch.Size([1600]), torch.float32, True)

('decoder.bias', torch.Size([33278]), torch.float32, True)

Number of trainable parameters=33556078

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

| end of epoch 1 | time: 62.78s | valid loss 6.42 | valid ppl 616.89 | valid bpc 9.269

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

After switching to ASGD, we have more tensors that eventually fill up the memory:

| epoch 52 | 200/ 372 batches | lr 30.00000 | ms/batch 151.99 | loss 4.15 | ppl 63.61 | bpc 5.991

(<class 'torch.nn.parameter.Parameter'>, torch.Size([4600, 1150]))

(<class 'torch.nn.parameter.Parameter'>, torch.Size([4600, 1150]))

(<class 'torch.nn.parameter.Parameter'>, torch.Size([1600, 400]))

(<class 'torch.nn.parameter.Parameter'>, torch.Size([4600, 1150]))

(<class 'torch.nn.parameter.Parameter'>, torch.Size([4600, 1150]))

(<class 'torch.nn.parameter.Parameter'>, torch.Size([1600, 400]))

(<class 'torch.nn.parameter.Parameter'>, torch.Size([4600, 1150]))

(<class 'torch.nn.parameter.Parameter'>, torch.Size([4600, 1150]))

(<class 'torch.nn.parameter.Parameter'>, torch.Size([1600, 400]))

(<class 'torch.nn.parameter.Parameter'>, torch.Size([4600, 1150]))

(<class 'torch.nn.parameter.Parameter'>, torch.Size([4600, 1150]))

(<class 'torch.nn.parameter.Parameter'>, torch.Size([1600, 400]))

(<class 'torch.nn.parameter.Parameter'>, torch.Size([4600, 1150]))

(<class 'torch.nn.parameter.Parameter'>, torch.Size([4600, 1150]))

(<class 'torch.nn.parameter.Parameter'>, torch.Size([1600, 400]))

(<class 'torch.nn.parameter.Parameter'>, torch.Size([4600, 1150]))

(<class 'torch.nn.parameter.Parameter'>, torch.Size([4600, 1150]))

(<class 'torch.nn.parameter.Parameter'>, torch.Size([1600, 400]))

(<class 'torch.nn.parameter.Parameter'>, torch.Size([4600, 1150]))

(<class 'torch.nn.parameter.Parameter'>, torch.Size([4600, 1150]))

(<class 'torch.nn.parameter.Parameter'>, torch.Size([1600, 400]))

(<class 'torch.nn.parameter.Parameter'>, torch.Size([4600, 1150]))

(<class 'torch.nn.parameter.Parameter'>, torch.Size([4600, 1150]))

(<class 'torch.nn.parameter.Parameter'>, torch.Size([1600, 400]))

(<class 'torch.nn.parameter.Parameter'>, torch.Size([4600, 1150]))

(<class 'torch.nn.parameter.Parameter'>, torch.Size([4600, 1150]))

(<class 'torch.nn.parameter.Parameter'>, torch.Size([1600, 400]))

(<class 'torch.nn.parameter.Parameter'>, torch.Size([4600, 1150]))

(<class 'torch.nn.parameter.Parameter'>, torch.Size([4600, 1150]))

(<class 'torch.nn.parameter.Parameter'>, torch.Size([1600, 400]))

(<class 'torch.nn.parameter.Parameter'>, torch.Size([4600, 1150]))

(<class 'torch.nn.parameter.Parameter'>, torch.Size([4600, 1150]))

(<class 'torch.nn.parameter.Parameter'>, torch.Size([1600, 400]))

(<class 'torch.nn.parameter.Parameter'>, torch.Size([4600, 1150]))

(<class 'torch.nn.parameter.Parameter'>, torch.Size([4600, 1150]))

(<class 'torch.nn.parameter.Parameter'>, torch.Size([1600, 400]))

(<class 'torch.Tensor'>, torch.Size([33278, 400]))

(<class 'torch.Tensor'>, torch.Size([4600, 400]))

(<class 'torch.Tensor'>, torch.Size([4600, 1150]))

(<class 'torch.Tensor'>, torch.Size([4600]))

(<class 'torch.Tensor'>, torch.Size([4600]))

(<class 'torch.Tensor'>, torch.Size([4600, 1150]))

(<class 'torch.Tensor'>, torch.Size([4600, 1150]))

(<class 'torch.Tensor'>, torch.Size([4600]))

(<class 'torch.Tensor'>, torch.Size([4600]))

(<class 'torch.Tensor'>, torch.Size([1600, 1150]))

(<class 'torch.Tensor'>, torch.Size([1600, 400]))

(<class 'torch.Tensor'>, torch.Size([1600]))

(<class 'torch.Tensor'>, torch.Size([1600]))

(<class 'torch.Tensor'>, torch.Size([33278]))

(<class 'torch.nn.parameter.Parameter'>, torch.Size([4600, 1150]))

(<class 'torch.nn.parameter.Parameter'>, torch.Size([4600, 1150]))

(<class 'torch.nn.parameter.Parameter'>, torch.Size([1600, 400]))

(<class 'torch.Tensor'>, torch.Size([26107, 80]))

(<class 'torch.Tensor'>, torch.Size([21764, 10]))

(<class 'torch.Tensor'>, torch.Size([245569, 1]))

(<class 'torch.Tensor'>, torch.Size([2088628]))

(<class 'torch.Tensor'>, torch.Size([217646]))

(<class 'torch.Tensor'>, torch.Size([245569]))

(<class 'torch.nn.parameter.Parameter'>, torch.Size([33278]))

(<class 'torch.nn.parameter.Parameter'>, torch.Size([1600]))

(<class 'torch.nn.parameter.Parameter'>, torch.Size([1600]))

(<class 'torch.nn.parameter.Parameter'>, torch.Size([1600, 400]))

(<class 'torch.nn.parameter.Parameter'>, torch.Size([1600, 1150]))

(<class 'torch.nn.parameter.Parameter'>, torch.Size([4600]))

(<class 'torch.nn.parameter.Parameter'>, torch.Size([4600]))

(<class 'torch.nn.parameter.Parameter'>, torch.Size([4600, 1150]))

(<class 'torch.nn.parameter.Parameter'>, torch.Size([4600, 1150]))

(<class 'torch.nn.parameter.Parameter'>, torch.Size([4600]))

(<class 'torch.nn.parameter.Parameter'>, torch.Size([4600]))

(<class 'torch.nn.parameter.Parameter'>, torch.Size([4600, 1150]))

(<class 'torch.nn.parameter.Parameter'>, torch.Size([4600, 400]))

(<class 'torch.nn.parameter.Parameter'>, torch.Size([33278, 400]))

(<class 'torch.nn.parameter.Parameter'>, torch.Size([4600, 1150]))

(<class 'torch.nn.parameter.Parameter'>, torch.Size([4600, 1150]))

(<class 'torch.nn.parameter.Parameter'>, torch.Size([1600, 400]))

(<class 'torch.nn.parameter.Parameter'>, torch.Size([1600, 400]))

(<class 'torch.nn.parameter.Parameter'>, torch.Size([4600, 1150]))

(<class 'torch.nn.parameter.Parameter'>, torch.Size([4600, 1150]))

(<class 'torch.Tensor'>, torch.Size([33278, 400]))

(<class 'torch.Tensor'>, torch.Size([4600, 400]))

(<class 'torch.Tensor'>, torch.Size([4600]))

(<class 'torch.Tensor'>, torch.Size([4600]))

(<class 'torch.Tensor'>, torch.Size([4600, 1150]))

(<class 'torch.Tensor'>, torch.Size([4600]))

(<class 'torch.Tensor'>, torch.Size([4600]))

(<class 'torch.Tensor'>, torch.Size([1600, 1150]))

(<class 'torch.Tensor'>, torch.Size([1600]))

(<class 'torch.Tensor'>, torch.Size([1600]))

(<class 'torch.Tensor'>, torch.Size([33278]))

Named parameters of the model:

('encoder.weight', torch.Size([33278, 400]), torch.float32, True)

('rnns.0.weight\_ih\_l0', torch.Size([4600, 400]), torch.float32, True)

('rnns.0.weight\_hh\_l0', torch.Size([4600, 1150]), torch.float32, True)

('rnns.0.bias\_ih\_l0', torch.Size([4600]), torch.float32, True)

('rnns.0.bias\_hh\_l0', torch.Size([4600]), torch.float32, True)

('rnns.1.weight\_ih\_l0', torch.Size([4600, 1150]), torch.float32, True)

('rnns.1.weight\_hh\_l0', torch.Size([4600, 1150]), torch.float32, True)

('rnns.1.bias\_ih\_l0', torch.Size([4600]), torch.float32, True)

('rnns.1.bias\_hh\_l0', torch.Size([4600]), torch.float32, True)

('rnns.2.weight\_ih\_l0', torch.Size([1600, 1150]), torch.float32, True)

('rnns.2.weight\_hh\_l0', torch.Size([1600, 400]), torch.float32, True)

('rnns.2.bias\_ih\_l0', torch.Size([1600]), torch.float32, True)

('rnns.2.bias\_hh\_l0', torch.Size([1600]), torch.float32, True)

('decoder.bias', torch.Size([33278]), torch.float32, True)

Number of trainable parameters=33556078

It is necessary to rewrite the code of main.py in such a way as to make it launchable by Python console, so we can debug and see what are the tensors that fill up the memory.

Just after

| epoch 43 | 200/ 372 batches | lr 30.00000 | ms/batch 154.68 | loss 4.22 | ppl 68.24 | bpc 6.093

Variables that we see in the debugger:

* best\_val\_loss
* corpus:
  + dictionary (33278)
  + test (tensor 245569)
  + train (tensor 2088628)
  + valid (tensor 217646)
* epoch = 43
* eval\_batch\_size=10
* model: among other things, contains
  + Encoder: embeddings table (33278 x 400)
  + a ModuleList of 3 RNNs: ModuleList( (0): LSTM(400, 1150) , (1): LSTM(1150, 1150), (2): LSTM(1150, 400))
  + decoder: Linear(in\_features=1150, out\_features=33278, bias=True)
* The **optimizer**, most likely responsible for the OOM  
  ASGD (Parameter Group 0  
   alpha: 0.75  
   lambd: 0.0  
   lr: 30  
   t0: 0  
   weight\_decay: 1.2e-06)
  + param\_groups = list:1 [ dict:6{‘lr’:30, the hyperparameters above etc., ‘params’: list:14 [tensors]
  + state = defaultdict:23 {Parameter containing Tensor, Parameter containing Tensor, etc etc.}

In the next epoch, we have again:

* optimizer
  + defaults (its default settings)
  + param\_groups = list containing 1 dictionary, that has the hyperparameters ‘lr’, ‘alpha’ etc. and ‘params’=a list of 14 Parameters w/Tensors
  + state= defaultdict with 26 elements

Epoch 44: state=defaultdict with 29 elements

Currently looking at the port for Torch 1.2.0 made by user mourga on GitHub…

Not only we use torch.cuda.empty\_cache(), but we also add his modifications, namely:

**for** prm **in** model.parameters():  
 **if** prm **in** optimizer.state.keys():  
 tmp[prm] = prm.data.detach()  
 prm.data = optimizer.state[prm][**'ax'**].detach()

and

**for** prm **in** model.parameters():  
 **if** prm **in** tmp.keys():  
 prm.data = tmp[prm].detach()  
 prm.requires\_grad = **True**

in the main.py training loop.

## AWD-LSTM : Experiment on WT-2

Using all the default hyperparameters, and the command

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

| epoch 1 | 200/ 372 batches | lr 30.00000 | ms/batch 142.41 | loss 7.81 | ppl 2469.21 | bpc 11.270

| end of epoch 1 | time: 60.71s | valid loss 6.42 | valid ppl 616.89 | valid bpc 9.269

| epoch 5 | 200/ 372 batches | lr 30.00000 | ms/batch 147.09 | loss 5.81 | ppl 332.91 | bpc 8.379

| end of epoch 5 | time: 62.77s | valid loss 5.41 | valid ppl 223.52 | valid bpc 7.804

| epoch 10 | 200/ 372 batches | lr 30.00000 | ms/batch 145.91 | loss 5.24 | ppl 189.41 | bpc 7.565

| end of epoch 10 | time: 62.74s | valid loss 4.99 | valid ppl 146.23 | valid bpc 7.192

| epoch 50 | 200/ 372 batches | lr 30.00000 | ms/batch 151.32 | loss 4.17 | ppl 64.58 | bpc 6.013

| end of epoch 50 | time: 63.30s | valid loss 4.43 | valid ppl 83.96 | valid bpc 6.392

| epoch 100 | 200/ 372 batches | lr 30.00000 | ms/batch 151.84 | loss 4.01 | ppl 54.93 | bpc 5.780

| end of epoch 100 | time: 63.53s | valid loss 4.40 | valid ppl 81.66 | valid bpc 6.352

| epoch 200 | 200/ 372 batches | lr 30.00000 | ms/batch 148.23 | loss 3.91 | ppl 50.01 | bpc 5.644

| end of epoch 200 | time: 63.24s | valid loss 4.39 | valid ppl 80.25 | valid bpc 6.327

| epoch 500 | 200/ 372 batches | lr 30.00000 | ms/batch 148.74 | loss 3.85 | ppl 47.03 | bpc 5.556

| end of epoch 500 | time: 63.28s | valid loss 4.37 | valid ppl 78.92 | valid bpc 6.302

| epoch 750 | 200/ 372 batches | lr 30.00000 | ms/batch 152.06 | loss 3.82 | ppl **45.41** | bpc 5.505

| end of epoch 750 | time: 63.40s | valid loss 4.36 | valid ppl **78.49** | valid bpc 6.294

| End of training | test loss 4.32 | test ppl **74.98** | test bpc 6.228