[Preliminary information (Datasets, graphs) 4](#_Toc44947587)

[WikiText-2 4](#_Toc44947588)

[WikiText-2 Stats 4](#_Toc44947589)

[Graph 4](#_Toc44947590)

[Meetings 5](#_Toc44947591)

[IA 5](#_Toc44947592)

[IA, PSK, KF 5](#_Toc44947593)

[Attempt to include the whole AWD-LSTM implementation 6](#_Toc44947594)

[Input facilities 6](#_Toc44947595)

[Training loop 6](#_Toc44947596)

[Corpus 7](#_Toc44947597)

[Batch & Input data 8](#_Toc44947598)

[Embeddings used by the model 9](#_Toc44947599)

[Hypothesis: use X and the indices of the globals 10](#_Toc44947600)

[Adjusting awd-lstm for the current PyTorch setup (1.4.0) 11](#_Toc44947601)

[WeightDrop 11](#_Toc44947602)

[Optimizer error 12](#_Toc44947603)

[Weights can be flattened in memory 12](#_Toc44947604)

[Torch 0.4.0 and 1.0.0 13](#_Toc44947605)

[Out of Memory 13](#_Toc44947606)

[Meeting with IA, 12/05 14](#_Toc44947607)

[GRU\_s on WT-2 – bugs in the model were not solved 16](#_Toc44947608)

[Model Verification and Modification 22](#_Toc44947609)

[The loss when operating on senses 22](#_Toc44947610)

[Considerations 23](#_Toc44947611)

[Graph retrieval 24](#_Toc44947612)

[Boundaries of graph node types 24](#_Toc44947613)

[Analysis… 24](#_Toc44947614)

[Senses’ +1 displacement 26](#_Toc44947615)

[Adding lemmatization for the global node 27](#_Toc44947616)

[GRU\_s on WT-2 – version 2 28](#_Toc44947617)

[Baseline GRU 28](#_Toc44947618)

[Mini-experiment – Overfit on fragment of WT-2 28](#_Toc44947619)

[Experiment – GRU on WT-2 29](#_Toc44947620)

[Model 29](#_Toc44947621)

[Experiment 29](#_Toc44947622)

[Experiment – GRU\_GAT on WT-2 30](#_Toc44947623)

[Model 30](#_Toc44947624)

[Experiment 31](#_Toc44947625)

[LSTM on WT-2 33](#_Toc44947626)

[Model 33](#_Toc44947627)

[Experiments 35](#_Toc44947628)

[Mini-experiment: LSTM on fragment of WT-2 35](#_Toc44947629)

[LSTM w/dropout on WT-2 35](#_Toc44947630)

[LSTM\_GAT w/dropout on WT-2 36](#_Toc44947631)

[Verifiying node retrieval from the graph for WikiText-2 38](#_Toc44947632)

[AWD-LSTM on WT-2 39](#_Toc44947633)

[Baseline AWD-LSTM 39](#_Toc44947634)

[Model 39](#_Toc44947635)

[Experiment: unmodified AWD-LSTM on WT-2 39](#_Toc44947636)

[Hypotheses for CUDAOutOfMemory 40](#_Toc44947637)

[Further runs to try to solve CUDAOutOfMemory 41](#_Toc44947638)

[Analysis of tensors in GPU memory 42](#_Toc44947639)

[AWD-LSTM : Experiment 1 on WT-2 46](#_Toc44947640)

[Re-run, port hyperparams 46](#_Toc44947641)

[Port for PyTorch 1.2.0 by @ahmetumutdurmus 47](#_Toc44947642)

[Port for PyTorch 1.2.0 by @mourga 48](#_Toc44947643)

[Synthesis 48](#_Toc44947644)

[Mogrifier LSTM on WT-2 [not viable] 50](#_Toc44947645)

[Model 50](#_Toc44947646)

[AWD-LSTM, version 2.1 54](#_Toc44947647)

[Perplexity results and alternatives 54](#_Toc44947648)

[@mourga’s port for PyTorch 1.2.0 54](#_Toc44947649)

[AWD-LSTM Experiment : Modified port on WikiText-2 55](#_Toc44947650)

[Adding upon AWD-LSTM 56](#_Toc44947651)

[Using the same Vocabulary 56](#_Toc44947652)

[Using the pre-trained FastText embeddings 59](#_Toc44947653)

[Idea #1: Add projected input signal to 1st layer output 59](#_Toc44947654)

[Issues: 61](#_Toc44947655)

[Idea #2: Concatenate the input signal to the word embedding 63](#_Toc44947656)

[Comparison of the methods of inclusion of FastText embeddings into AWD-LSTM 64](#_Toc44947657)

[LSTM on WT-2, version 2.1 66](#_Toc44947658)

[Experiment 2.1.1. 66](#_Toc44947659)

[Model 66](#_Toc44947660)

[Run 66](#_Toc44947661)

[Experiment 2.1.2 67](#_Toc44947662)

[Faster computations 69](#_Toc44947663)

[Experiments 71](#_Toc44947664)

[Model 71](#_Toc44947665)

[Experiment 2.1.3 71](#_Toc44947666)

[Comparison between LSTM architectures 72](#_Toc44947667)

[**Aside:** Note on the inclusion of lemmatization 72](#_Toc44947668)

[Meeting with IA, 23/07 73](#_Toc44947669)

[AWD-LSTM, version 3 73](#_Toc44947670)

[AWD\_modified 73](#_Toc44947671)

[Embeddings 73](#_Toc44947672)

[Forward() call and embeddings update 74](#_Toc44947673)

[Results and combining softmaxes 76](#_Toc44947674)

[AWD-LSTM, ensemble model 77](#_Toc44947675)

[AWD-LSTM, base 77](#_Toc44947676)

[AWD-LSTM, FastText embeddings 77](#_Toc44947677)

[Combining 2 models 78](#_Toc44947678)

[Weighted average of the result logits 78](#_Toc44947679)

[Ensemble 1.0 , experiments 80](#_Toc44947680)

[a=1 80](#_Toc44947681)

# Preliminary information (Datasets, graphs)

## WikiText-2

### WikiText-2 Stats

(note: I have updated the statistics, given that now we use the full vocabulary of WikiText-2 of 33,278 tokens, instead of the reduced version I previously used with |V|=31640)

Number of tokens in the splits:

Training: 2,051,910 tokens ; Validation: 213,886 ; Test: 241,211

Mini-Dataset – training: 598 tokens

### 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

INFO : X\_**definitions**.shape=torch.Size([**28141**, 300])

INFO : X\_**examples**.shape=torch.Size([**26593**, 300])

INFO : X\_**senses**.shape=torch.Size([**28141**, 300])

INFO : X\_**globals**.shape=torch.Size([**33278**, 300])

Defining the edges: def, exs

INFO : def\_edges\_se.\_\_len\_\_()=**28141**

INFO : exs\_edges\_se.\_\_len\_\_()=**26593**

Defining the edges: sc

**sc\_edges**.\_\_len\_\_()= **28141**

**#** 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\_\_()= **50938**

**#** 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 50938– 28141**= 22797** globals with no dictionary information, over a total of **33278 (68.5%).**

syn\_edges.\_\_len\_\_()=**42024 #** synonyms

ant\_edges.\_\_len\_\_()=**3942 #** antonyms

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

Data(edge\_index=[2, 151638], edge\_type=[151638], node\_types=[116153], num\_relations=[1], x=[116153, 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.

However, we saw previously with the GRU that we went from 163.7 to 159.7.

So either I handle the recurrent states incorrectly with the d500@L3 setup, or the dropout has a negative effect.

# 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, (for instance, all 900, with a dropout of 0.1 for regularization).

Eventually, I pick the following reasonable choice: 1000 > 1000 > 500, with dropout p=0.2.

INFO : DataParallel(

(module): LSTM(

(dropout): Dropout(p=0.2, inplace=False)

(main\_lstm\_ls): ModuleList(

(0): LSTM(300, 1000)

(1): LSTM(1000, 1000)

(2): LSTM(1000, 500))

(linear2global): Linear(in\_features=500, out\_features=31640, bias=True) ))

INFO : Parameters:

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

('module.select\_first\_indices', torch.Size([1000]), torch.float32, False)

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

('module.memory\_hn', torch.Size([3, 4, 1000]), torch.float32, False)

('module.memory\_cn', torch.Size([3, 4, 1000]), torch.float32, False)

('module.memory\_hn\_senses', torch.Size([3, 4, 500]), torch.float32, False) # unused for now

('module.memory\_cn\_senses', torch.Size([3, 4, 500]), torch.float32, False) # unused for now

('module.main\_lstm\_ls.0.weight\_ih\_l0', torch.Size([4000, 300]), torch.float32, True)

('module.main\_lstm\_ls.0.weight\_hh\_l0', torch.Size([4000, 1000]), torch.float32, True)

('module.main\_lstm\_ls.0.bias\_ih\_l0', torch.Size([4000]), torch.float32, True)

('module.main\_lstm\_ls.0.bias\_hh\_l0', torch.Size([4000]), torch.float32, True)

('module.main\_lstm\_ls.1.weight\_ih\_l0', torch.Size([4000, 1000]), torch.float32, True)

('module.main\_lstm\_ls.1.weight\_hh\_l0', torch.Size([4000, 1000]), torch.float32, True)

('module.main\_lstm\_ls.1.bias\_ih\_l0', torch.Size([4000]), torch.float32, True)

('module.main\_lstm\_ls.1.bias\_hh\_l0', torch.Size([4000]), torch.float32, True)

('module.main\_lstm\_ls.2.weight\_ih\_l0', torch.Size([2000, 1000]), torch.float32, True)

('module.main\_lstm\_ls.2.weight\_hh\_l0', torch.Size([2000, 500]), torch.float32, True)

('module.main\_lstm\_ls.2.bias\_ih\_l0', torch.Size([2000]), torch.float32, True)

('module.main\_lstm\_ls.2.bias\_hh\_l0', torch.Size([2000]), torch.float32, True)

('module.linear2global.weight', torch.Size([31640, 500]), torch.float32, True)

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

INFO : Number of trainable parameters=66.37M, where core= 66.37 – 15.85(softmax) – 34.30(embeddings) = 16.22M

## Experiments

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

Hyperparameters: \_batchPerSeqlen32\_area32\_lr0.0001\_epochs150

|  |  |  |
| --- | --- | --- |
| **Architecture** | **Input Signals** | **Hyperparameters** |
| model: LSTM | 1. The word embedding of the current global   (d=300) | batch\_size=4 |
| LSTM with 3 layers (1000, 1000, 500). Dropout on them with p=0.2 | TBPTT length=8 |
|  | learning rate=10^(-4) |
| Followed by linear2Globals **FF-NN** |  | grapharea=32, hops=1 |

|  |  |
| --- | --- |
| **Epoch** | **Training perplexity** |
| 1 | 30696.26 |
| 2 | 10134.27 |
| 3 | 1192.55 |
| 10 | 113.27 |
| 50 | 104.13 |
| 100 | 103.28 |
| 150 | 37.5 |
| 200 | 14.25 |
| 250 | 6.43 |
| 300 | 3.12 |

due to the dropout, we do not expect a 100% overfit.

### LSTM w/dropout on WT-2

|  |  |  |
| --- | --- | --- |
| **Architecture** | **Input Signals** | **Hyperparameters** |
| model: LSTM | 1. The word embedding of the current global   (d=300) | batch\_size=40 |
| LSTM with 3 layers (1000, 1000, 500). Dropout on them with p=0.2 | TBPTT length=35 |
|  | learning rate=5\*10^(-5) |
| Followed by linear2Globals **FF-NN** |  | grapharea=32, hops=1 |

|  |  |  |
| --- | --- | --- |
| *Epoch* | *Training perplexity* | *Validation perplexity* |
| 1 | 1124.66 | 686.21 |
| 2 | 905.69 | 679.67 |
| 3 | 678.99 | 443.61 |
| 4 | 517.21 | 371.13 |
| 5 | 439.43 | 332.15 |
| 6 | 388.27 | 305.7 |
| 7 | 351.18 | 286.98 |
| 8 | 322.27 | 269.74 |
| 9 | 298.73 | 257.34 |
| 10 | 278.88 | 246.39 |
| 11 | 261.8 | 236.06 |
| 12 | 246.95 | 227.59 |
| 13 | 233.6 | 219.7 |
| 14 | 221.6 | 213.34 |
| 15 | 210.82 | 207.89 |
| 16 | 200.97 | 202.56 |
| 17 | 191.89 | 197.09 |
| 18 | 183.45 | 192.43 |
| 19 | 175.9 | 189.04 |
| 20 | 168.79 | 186.1 |
| 21 | 162.43 | 182.83 |
| 22 | 156.5 | 180.44 |
| 23 | 151.06 | 178.31 |
| 24 | 145.77 | 175.77 |
| 25 | 140.98 | 174.12 |
| 26 | 136.55 | 172.33 |
| 27 | 132.19 | 170.6 |
| 28 | 128.06 | 169.38 |
| 29 | 124.31 | 168.21 |
| 30 | 120.7 | 167.12 |
| 31 | 117.29 | 166.49 |
| 32 | 114.01 | 165.87 |
| 33 | 110.85 | 165.06 |
| 34 | 107.88 | 164.63 |
| 35 | 105.13 | 164.63 |
| 36 | 102.37 | **163.27** |
| 37 | 99.82 | 163.48 |
| 38 | 97.31 | 163.57 |
| 39 | 94.93 | 163.64 |
| 40 | 92.56 | 164.2 |

Valid PPL on WT-2 = 163.3

### LSTM\_GAT w/dropout on WT-2

|  |  |  |
| --- | --- | --- |
| **Architecture** | **Input Signals** | **Hyperparameters** |
| model: LSTM | 1. The word embedding of the current global 2. The node-state of the current global   (d=600) | batch\_size=40 |
| **LSTM** with 3 layers (1000, 1000, 500). Dropout on them with p=0.2 | TBPTT length=35 |
| **GAT** for the state of the current global’s node (4 heads x d75) | learning rate=5\*10^(-5) |
| Followed by linear2Globals **FF-NN** |  | grapharea=32, hops=1 |

|  |  |  |
| --- | --- | --- |
| *Epoch* | *Training perplexity* | *Validation perplexity* |
| 1 | 1118.68 | 638.85 |
| 2 | 693.5 | 466.68 |
| 3 | 551.84 | 396.17 |
| 4 | 477.94 | 356.3 |
| 5 | 427.52 | 328.42 |
| 6 | 389.93 | 308.25 |
| 7 | 359.94 | 292.59 |
| 8 | 335.64 | 277.85 |
| 9 | 314.83 | 267.68 |
| 10 | 297.25 | 258.35 |
| 11 | 282.04 | 249.88 |
| 12 | 268.54 | 242.35 |
| 13 | 256.7 | 235.38 |
| 14 | 245.53 | 230.11 |
| 15 | 235.3 | 224.78 |
| 16 | 226.21 | 220.38 |
| 17 | 217.6 | 214.92 |
| 18 | 209.44 | 210.52 |
| 19 | 201.94 | 206.97 |
| 20 | 194.91 | 203.73 |
| 21 | 188.37 | 200.42 |
| 22 | 182.23 | 197.79 |
| 23 | 176.46 | 195.38 |
| 24 | 170.97 | 192.89 |
| 25 | 165.94 | 190.56 |
| 26 | 161.15 | 188.84 |

too slow, and not good. The 1000>1000>500 LSTM without GAT on the global node-states was

|  |  |
| --- | --- |
| 168.79 | 186.1 |

at Epoch 20.

### Verifiying node retrieval from the graph for WikiText-2

Since the results-in-progress for the LSTM-GAT+dropout seem underwhelming, let us verify the correctness of the retrieval of node neighbours for the WikiText-2 dataset.

Let us pick the global n.124, ‘edition’.

DG.get\_graph\_dataobject():

The graph has 28070 senses, and 59710-28070=31640 globals.

We previously determined – see the Stats section at the start of this document – that:

**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])

We expect ‘edition’ to correspond to the 28070+124= 28194th row of X.

Let us gear the local CPU project for WT-2, from SemCor, checking that all the relevant files are identical to the remote WT-2 version.

GA.get\_indices\_area\_toinclude(graph\_dataobject.edge\_index, graph\_dataobject.edge\_type, node\_index=28070+124, area\_size = 32, max\_hops=1):

Nodes: 28194, 8658, 8659, 8660, 31559

28194=124th global, ‘edition’

8658 = 8659th sense, ‘edition.n.01’

8659 = 8660th sense, ‘edition.n.02’

8660 = 8661th sense, ‘edition.n.03’

31559 = 3489th global, ‘version’ (evidently a synonym)

# 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 1 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

From nlpprogress.com, we gather that the Perplexity values for AWD-LSTM on WikiText-2 are supposed to be: Valid PPL=68.6, Test PPL=65.8

### Re-run, port hyperparams

Let us try the hyperparameters used in the port for PyTorch 1.2.0 by @ahmetumutdurmus on GitHub.

python

main.py

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

--save model.tar

--layer\_num 3 🡪 nlayers 3

--embed\_size 400 🡪 emsize

--hidden\_size 1150 🡪 nhid

--lstm\_type pytorch

--w\_drop 0.65 🡪 wdrop (from 0.5)

--dropout\_i 0.4 🡪 dropouti (from 0.65)

--dropout\_l 0.3 🡪 dropouth (from 0.2 in the original command)

--dropout\_o 0.4 🡪 dropout

--dropout\_e 0.1 🡪 dropoute

--winit 0.1

--batch\_size 80

--bptt 70

--ar 2 🡪 alpha

--tar 1 🡪 beta

--weight\_decay 1.2e-6

--epochs 750

--lr 30

--max\_grad\_norm 0.25 🡪 clip

--non\_mono 5 🡪 nonmono

--device gpu 🡪 (auto)

--log 50 🡪 log\_interval (from 200)

Using the port’s hyperparameters in my version, I still get:

| end of epoch 750 | time: 64.75s | valid loss 4.37 | **valid ppl 78.76** | valid bpc 6.299

==================================================================================

| End of training | test loss 4.32 | **test ppl 75.46** | test bpc 6.238

instead of 68.8, 65.8.

### Port for PyTorch 1.2.0 by @ahmetumutdurmus

I just git-clone the PyTorch 1.2.0 port. The author states

“I was able to replicate the results presented in the paper to a fairly reasonable degree (± 1.0 PPL due to random initiation). …”

I added a parallel ported-awd-lstm folder, out of Task1\_Gbwe

I get:

UserWarning: The .grad attribute of a Tensor that is not a leaf Tensor is being accessed. Its .grad attribute won't be populated during autograd.backward(). If you indeed want the gradient for a non-leaf Tensor, use .retain\_grad() on the non-leaf Tensor. If you access the non-leaf Tensor by mistake, make sure you access the leaf Tensor instead. See github.com/pytorch/pytorch/pull/30531 for more informations.

warnings.warn("The .grad attribute of a Tensor that is not a leaf Tensor is being accessed. Its .grad "

Location:

File "main.py", line 140, in train

norm = nn.utils.clip\_grad\_norm\_(model.parameters(), args.max\_grad\_norm)

File "/home/andrealk3/venvs/torch15/lib/python3.6/site-packages/torch/nn/utils/clip\_grad.py", line 24, in clip\_grad\_norm\_

parameters = list(filter(lambda p: p.grad is not None, parameters))

File "/home/andrealk3/venvs/torch15/lib/python3.6/site-packages/torch/nn/utils/clip\_grad.py", line 24, in <lambda>

parameters = list(filter(lambda p: p.grad is not None, parameters))

Sometimes this kind of error ("The .grad attribute of a Tensor that is not a leaf Tensor is being accessed.”) is not necessarily relevant. For a first attempt, I will just suppress the warning and execute the command.

The best validation PPL was only **80.6**, reached early at Epoch 100. Test set perplexity of best model: **77.2**.

### Port for PyTorch 1.2.0 by @mourga

This time I git-clone and execute the port of AWD-LSTM for PyTorch 1.2.0 by @mourga.

Without doing rnn.flatten\_parameters() before the layer call, I get the errors:

/pytorch/aten/src/ATen/native/cudnn/RNN.cpp:1269: 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().

This means that the unmodified version is slower and clogs up the GPUs too. However, I wanted to try it out without changing \*anything\*.

Continued the training all the way to epoch 549. We see that:

…

| epoch 500 | 800/ 932 batches | lr 30.00000 | ms/batch 99.60 | loss 3.98 | ppl 53.39 | bpc 5.739

| end of epoch 500 | time: 99.80s | valid loss 4.32 | valid ppl 74.92 | valid bpc 6.227

…

| epoch 549 | 800/ 932 batches | lr 30.00000 | ms/batch 95.17 | loss 3.99 | **ppl 54.00** | bpc 5.755

| end of epoch 549 | time: 99.80s | valid loss 4.32 | **valid ppl 74.84** | valid bpc 6.226

### Synthesis

So far, we have had:

With my fixes from the original AWD-LSTM:

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

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

My version, using the hyperparameters of the port for PyTorch 1.2.0 by @ahmetumutdurmus:

| end of epoch 750 | time: 64.75s | valid loss 4.37 | **valid ppl 78.76** |

==================================================================================

| End of training | test loss 4.32 | **test ppl 75.46** | test bpc 6.238

The git-clone of the port by @ahmetumutdurmus:

The best validation PPL was only **80.6**, reached early at Epoch 100.

Test set perplexity of best model: **77.2**.

Port for PyTorch 1.2.0 by @mourga:

| end of epoch 549 | time: 99.80s | valid loss 4.32 | **valid ppl 74.84** | valid bpc 6.226

It’s worth setting aside 1 GPU and running the 1.2.0 port by @mourga, adding flatten\_parameters() before a layer’s forward call…

… of course, in order to be able to do so, I should find a way to solve the non-flat parameters problem…

… that the author herself couldn’t solve. The original authors (S.Merity et al.) used a hack-y fix that worked for PyTorch 1.4.0.

Time to focus on the senses’ task…

however, I can still use the AWD-LSTM as a baseline, given that it’s close enough to the original results and I made a reasonable attempt to reach them.

# Mogrifier LSTM on WT-2 [not viable]

## Model

The LSTM’s input *x* is gated conditioned on the output of the previous step *hprev*.

Next, the gated input is used in a similar manner to gate the output of the previous time step. After several rounds of this mutual gating, the last updated *x* and *hprev* are fed to an LSTM.

With *r*=5:

For i in [1,3,5]:

For i in [2,4]:

*r*=0 recovers a standard LSTM.

Settings and hyperparameters from the paper:

*r*=5.

For word-level tasks, BPTT window size is set to 70 and batch size to 64.

Optimization is performed with Adam with L = 0, a setting that resembles RMSProp without momentum.

Gradients are clipped to norm 10.

We switch to averaging weights similarly to Merity et al. (2017) after a certain number of checkpoints with no improvement in validation cross-entropy or at 80% of the training time at the latest.

We found no benefit to using two-step finetuning.

More from deepmind/lamb @ GitHub:

[ ('config\_version', 5),

# data

('file\_encoding', 'utf-8'),

('word\_based', True), # Whether to do word or character based modelling.

('episodic', False), # If true, iterate over examples (lines in the data files) in random order. If false, iterate mostly sequentially carrying over model from the previous example to the next.

# model

('num\_params', 35000000), # An upper bound on the total number of trainable parameters over all parts of the model (including the recurrent cell and input/output embeddings). If this is set to a meaningful value (i.e. not -1, the default), then hidden\_size is set to the largest possible value such that the parameter budget is not exceeded.

('share\_input\_and\_output\_embeddings', True), # Whether the input and output embeddings are the same matrix (transposed) or independent (the default)

('input\_embedding\_size', -1), # The length of the vector that represents an input token. If -1 (the default), then it's determined by input\_embedding\_ratio

('output\_embedding\_size', -1),

('input\_embedding\_ratio', 0.3530770457779424), # If input\_embedding\_size is not specified (i.e. -1), then it's set to round(input\_embedding\_ratio\*hidden\_size).

('output\_embedding\_ratio', -1.0),

('mos\_num\_components', 2), # See Breaking the softmax bottleneck. The default of 0 turns this feature off.

('token\_dropout', 0.0),

('embedding\_dropout', 0.0),

('input\_dropout', 0.6090979517941943), # The dropout rate (here and elsewhere, 0 means deterministic operation) for the input to the first layer (i.e. just after the input embeddings). This drops out individual elements of the embedding vector.

('output\_dropout', 0.34845530389157287), # The dropout rate for just after the cell output.

('downprojected\_output\_dropout', -1.0), # The dropout rate for the projection of the cell output. Only used if output\_embedding\_size is different from hidden\_size or if mos\_num\_components is not 1. Defaults to output\_dropout if set to -1.

('shared\_mask\_dropout', False),

('embed\_once', True), # Whether to compute the logits from the cell output in a single operation or per time step. The single operation is faster but uses more GPU memory.

# cell

('model', 'lstm'),

('num\_layers', 2),

('residual\_connections', False),

('lstm\_skip\_connection', True), # If true, for multi-layer (num\_layers>1) LSTMs, the output is computed as the sum of the outputs of the individual layers

('feature\_mask\_rounds', 0), # When feature\_mask\_rounds is 0, there is no extra gating in the LSTM.

('feature\_mask\_rank', 0),

('feature\_mask', False),

('sparsity\_ratio', -1.0),

('overlay\_rank', -1),

('hidden\_size', [-1]),

('hidden\_size\_multiplier', 1.0),

('layer\_norm', False), # Whether to perform Layer Normalization

('activation\_fn', 'tf.tanh'), # The non-linearity for the update candidate ('j') and the output ('o') in an LSTM

('tie\_forget\_and\_input\_gates', False),

('cap\_input\_gate', True),

('trainable\_initial\_state', False), # Whether the initial state of the recurrent cells is allowed to be learnt or is set to a fixed zero vector. In non-episodic mode, this switch is forced off.

('inter\_layer\_dropout', 0.09075401405970591), # The input dropout for layers other than the first one.

('state\_dropout', 0.2714030562283111), # This is the dropout rate for the recurrent state from the previous time step ('h' in an LSTM)

('state\_dropout\_flip\_rate', 0.0),

('update\_dropout', 0.0),

('cell\_clip', -1.0),

# objective

('model\_average', 'arithmetic'),

('num\_training\_samples', 1),

('l2\_penalty', 0.00023063627783021125),

('l1\_penalty', 0.0),

('activation\_norm\_penalty', 0.0),

('drop\_state\_probability', 0.01), # In non-episodic mode, model state is carried over from batch to batch. Not feeding back the state with drop\_state\_probability encourages the model to work well starting from the zero state which brings it closer to the test regime.

# initialization

('embedding\_init\_factor', 1.0),

('scale\_input\_embeddings', False),

('cell\_init\_factor', 1.0),

('forget\_bias', 1.0),

('output\_init\_factor', 1.0),

# schedule

('steps\_per\_turn', 200), # The number of optimization steps between two successive evaluations.

('print\_training\_stats\_every\_num\_steps', 200),

('turns', 1000),

# optimization

('optimizer\_type', 'rmsprop'), # RMSPROP is actually Adam with beta1=0.0

('rmsprop\_beta2', 0.999),

('rmsprop\_epsilon', 1e-08),

('adam\_beta1', 0.9),

('adam\_beta2', 0.999),

('adam\_epsilon', 1e-08),

('batch\_size', 64),

('accum\_batch\_size', -1), # The number of examples that are fed to the network at the same time. Set this to a divisor of batch\_size to reduce memory usage at the cost of possibly slower training. Using accum\_batch\_size does not change the results.

('max\_grad\_norm', 10.0),

('max\_time\_steps', 70),

('trigger\_averaging\_turns', 50), # The number of turns of no improvement on the validation set, after which weight averaging is turned on. Weight averaging is a trivial generalization of the idea behind Averaged SGD: it keeps track of the average weights, updating the average after each optimization step. Weight averaging does not affect training directly, only through evaluation.

('trigger\_averaging\_at\_the\_latest', 800),

# learning rate

('learning\_rate', 0.003183909546336849),

('learning\_rate\_decay', 1.0),

('learning\_rate\_decay\_burn\_in\_steps', 0),

('drop\_learning\_rate\_turns', -1),

('drop\_learning\_rate\_multiplier', 1.0),

('drop\_learning\_rate\_at\_the\_latest', -1),

# early stopping

('early\_stopping\_turns', -1),

('early\_stopping\_rampup\_turns', 0),

('early\_stopping\_worst\_xe\_target', ''),

('early\_stopping\_slowest\_rate', 0.0),

# cross-validation

('crossvalidate', False),

('crossvalidation\_folds', 10),

('crossvalidation\_rounds', 1),

# evaluation

('max\_training\_eval\_batches', 20),

('max\_eval\_eval\_batches', -1),

('max\_test\_eval\_batches', -1),

('min\_non\_episodic\_eval\_examples\_per\_stripe', 100),

('eval\_on\_test', False),

('eval\_method', 'deterministic'),

('num\_eval\_samples', 0),

('eval\_softmax\_temperature', -0.8),

('eval\_softmax\_temperature\_estimation\_num\_tokens', 50000),

('eval\_power\_mean\_power', 1.0),

('eval\_dropout\_multiplier', 1.0),

('validation\_prediction\_file', ''),

('dyneval', False),

('dyneval\_learning\_rate', 0.001),

('dyneval\_decay\_rate', 0.02),

('dyneval\_epsilon', 1e-05),

# experiments

# checkpoints

('save\_checkpoints', True),

# misc

('seed', 1),

('swap\_memory', True),

('log\_device\_placement', False),

('summary\_flush\_secs', 120),

]

The settings are many as they are refined, I can not replicate this on my own.

The repo at <https://github.com/deepmind/lamb> uses Tensorflow as a framework, I have done everything in PyTorch.

I can try out a Mogrifier LSTM baseline of my own, modifying the PyTorch implementation.

In particular, I have to turn a MogrifierLSTMCell from <https://github.com/fawazsammani/mogrifier-lstm-pytorch> into a LSTM network.

The gating should be done in parallel for the batch elements…

# AWD-LSTM, version 2.1

## Perplexity results and alternatives

Original results of AWD-LSTM on WikiText-2, from the paper by S.Merity et al., 2017:

|  |  |
| --- | --- |
| Valid PPL= 68.6 | Test PPL=65.8 |

So far, we have had:

With my fixes from the original AWD-LSTM:

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

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

My version, using the hyperparameters of the port for PyTorch 1.2.0 by @ahmetumutdurmus:

| end of epoch 750 | time: 64.75s | valid loss 4.37 | **valid ppl 78.76** |

==================================================================================

| End of training | test loss 4.32 | **test ppl 75.46** | test bpc 6.238

The git-clone of the port by @ahmetumutdurmus:

The best validation PPL was only **80.6**, reached early at Epoch 100.

Test set perplexity of best model: **77.2**.

Port for PyTorch 1.2.0 by @mourga:

| end of epoch 549 | time: 99.80s | valid loss 4.32 | **valid ppl 74.84** | valid bpc 6.226

The valid ppl seems interesting. However, it’s buggy, throwing a UserWarning: need to flatten parameters, and clogging the memory and the GPU.

I can still use the AWD-LSTM as a baseline, given that it’s close enough to the original results and I made a reasonable attempt to reach them.

## @mourga’s port for PyTorch 1.2.0

We must fix:

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().

Into the model’s forward():

Every time we pass through:

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

…

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

…

When we invoke, inside rnn.py:

result = \_VF.lstm(input, hx, self.\_flat\_weights, self.bias, self.num\_layers,

self.dropout, self.training, self.bidirectional, self.batch\_first)

Hypothesis: the WeightDrop wrapper calls

self.\_setweights()

>>>> self.module.flatten\_parameters() <<<<

return self.module.forward(\*args)

if setweights changes the parameters, then any attempt to flatten\_parameters should follow it, not precede it…

Even that does not solve the problem.

As an alternative:

I will insert into the @mourga port my own version of WeightDrop, and see whether that solves the problem while managing to keep the PPL improvements.

However, before trying out my alternative, I can follow Hypothesis #2 and correct the effect of the model setup written by the original authors:

**def** \_setup(self):  
 *# Terrible temporary solution to an issue regarding compacting weights re: CUDNN RNN* **if** issubclass(type(self.module), torch.nn.RNNBase):  
 self.module.flatten\_parameters = self.widget\_demagnetizer\_y2k\_edition

So, I re-enabled the original flatten\_parameters() function of the RNN module. I still do not know whether that causes bugs or memory problems.

Directive: run again an experiment on WikiText-2. If it doesn’t work, I will be justified in using my own version.

### AWD-LSTM Experiment : Modified port on WikiText-2

| epoch 1 | 800/ 932 batches | lr 30.00000 | ms/batch 86.48 | loss 6.52 | ppl 681.67 | bpc 9.413

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

| end of epoch 1 | time: 86.54s | valid loss 5.99 | valid ppl 398.30 | valid bpc 8.638

…

| end of epoch 750 | time: 89.32s | valid loss 4.31 | valid ppl 74.66 | valid bpc 6.222

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

Saving Averaged!

==================================================================

| End of training | test loss 4.32 | test ppl 75.51 | test bpc 6.239

The modified version of the port for PyTorch 1.2.0 works without bugs. The results it gives are in line with the best obtained so far, so we use this as our baseline and starting point for the evaluations on AWD-LSTM.

## Adding upon AWD-LSTM

### Using the same Vocabulary

To make a comparison under the same conditions, it is necessary to modify the creation of the vocabulary that we use for the globals, i.e. for our models and for the graph.

We should use the same method employed in AWD-LSTM, which is found in data.py:

**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), str(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)  
  
…

Thanks to implementing this, we have vocabulary\_df = 33,278, which is exactly the official vocabulary size for WikiText-2.

We can also explore what happens if we do not exclude the long tail in the vocabulary of SemCor.

With a min\_count=2, we have only 21988globals**.**

Without a barrier on the minimum frequency, and extracting words from all the 3 splits (training, validation, test) we get:

53138 globals

… (the rest of the discussion is in the senses’ document).

In the meantime, we re-build the vocabulary and word embeddings for the globals’ part.

How to match the original vocabulary of WikiText-2, containing its own symbols like @-@, to the vocabulary of FastText embeddings?

We could keep the vocabulary and make the pre-processing in VocabUtils.process\_slc\_token(token\_dict) mandatory… or we could modify the vocabulary H5 archive splitting “word” into 2 columns, “raw\_word” and “refined\_word”.B

VocabUtils.process\_slc\_token(token\_dict) is invoked in several places:

* when creating the graph:
  + get\_additional\_edges\_sensechildren\_from\_slc
  + get\_edges\_nyms
* In NumericalIndices.py:
  + in convert\_tokendict\_to\_tpl() # the internal function to: translate the word (and if present, the sense) into numerical indices.  
    The chain of calls is: Dataloader > TextDataset > NI.get\_tokens\_tpls() > NI. convert\_tokendict\_to\_tpl()
* Vocabulary. build\_vocabulary\_from\_senselabeled()

We observe that even in DataLoading. standardtextcorpus\_generator(split\_name) we have:

**for** token **in** line:  
 token\_dict ={**'surface\_form'**:token} *# to use the same refinement as the tokens from sense-labeled corpus* **yield** token\_dict

What about the word embeddings and connecting to FastText?

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

**for** idx\_word\_freq\_tpl **in** vocabulary\_df.itertuples():  
 word = idx\_word\_freq\_tpl[1]  
  
 **if** method == Method.DISTILBERT:  
 word\_vector = EDB.compute\_sentence\_dBert\_vector(distilBERT\_model, distilBERT\_tokenizer, word).squeeze().numpy()  
 **else**: *# i.e. elif method == Method\_for\_SPV.FASTTEXT:* word\_vector = fasttext\_vectors[word]  
  
 word\_vectors\_lls.append(word\_vector)  
 i = i+1  
  
embds\_nparray = np.array(word\_vectors\_lls)

we could insert process\_slc\_token here. This way, every single access to the vocabulary would have to pass through this. We would not see @-@, but only –.

What steps should we take then?

(n: renaming process\_slc\_token(…) into process\_word\_token(…))

Example line:

“

Homarus gammarus is a large <unk> , with a body length up to 60 centimetres ( 24 in ) and weighing up to 5 – 6 kilograms ( 11 – 13 lb ) , although the lobsters caught in lobster pots are usually 23 – 38 cm ( 9 – 15 in ) long and weigh 0 @.@ 7 – 2 @.@ 2 kg ( 1 @.@ 5 – 4 @.@ 9 lb ) .

”

If I process token-by-token after a split on whitespace, I will have “… 0 . 7 – 2 . 2 kg ( 1 . 5 – 4 . 9 lb ) .”

**Possibility 1**:

avoid using the FastText embeddings as input embeddings. Let a GRU/LSTM/AWD-LSTM train their own embeddings from a vocabulary with no changes.

The only way we use these pre-trained embeddings is to create the senses’ locations. The globals are part of the graph, but they are not used on their own. (and where does that leave the globals’ task?)

**Possibility 2**:

Keep the FastText embeddings for globals in the pipeline as they are. When we transform something like ‘0 @.@ 7’ into ‘0 . 7’, this should still be different from the ‘. <eos>’

And what about ‘3 @,@ 462 t’?

A standard language model would read those tokens as: [‘3’,’@,@’,’462’,’t’].

Idea:

the sense-labeled corpus does not have this issue, we can just pre-process in any way we wish without the need to keep as close to an unprocessed reading as possible.

In a standard text corpus that we read for the global task, we can proceed as follows:

* read the token as it is.
* If we do not find its original form (e.g. ‘@-@’) in our FastText-compatible vocabulary, we apply pre-processing and try to find ‘-‘.
* If even ‘-‘ is not present in our vocabulary, we use <unk>.

We need a FastText-compatible vocabulary in order to retrieve the word embeddings for ‘,’ and ‘-‘. (So, token preprocessing is necessary when creating the vocabulary).

When we read a standard text corpus, we try to get the raw token’s embedding. If not found, preprocess token (e.g. ‘@-@’ into ‘-‘) and try again. If not found, use <unk>.

Example text, nonprocessed tokens:

['Homarus', 'gammarus', 'is', 'a', 'large', '<unk>', ',', 'with', 'a', 'body', 'length', 'up', 'to', '60', 'centimetres', '(', '24', 'in', ')', 'and', 'weighing', 'up', 'to', '5', '–', '6', 'kilograms', '(', '11', '–', '13', 'lb', ')', ',', 'although', 'the', 'lobsters', 'caught', 'in', 'lobster', 'pots', 'are', 'usually', '23', '–', '38', 'cm', '(', '9', '–', '15', 'in', ')', 'long', 'and', 'weigh', '0', '@.@', '7', '–', '2', '@.@', '2', 'kg', '(', '1', '@.@', '5', '–', '4', '@.@', '9', 'lb', ')', '.', '<eos>']

['Homarus', 'gammarus', 'is', 'a', 'large', '<unk>', ',', 'with', 'a', 'body', 'length', 'up', 'to', '60', 'centimetres', '(', '24', 'in', ')', 'and', 'weighing', 'up', 'to', '5', '–', '6', 'kilograms', '(', '11', '–', '13', 'lb', ')', ',', 'although', 'the', 'lobsters', 'caught', 'in', 'lobster', 'pots', 'are', 'usually', '23', '–', '38', 'cm', '(', '9', '–', '15', 'in', ')', 'long', 'and', 'weigh', '0', '.', '7', '–', '2', '.', '2', 'kg', '(', '1', '.', '5', '–', '4', '.', '9', 'lb', ')', '.', '<eos>']

Once we inserted this adjustment (html.unescape + convert\_symbols) in the vocabulary creation, let us examine the resulting WikiText-2 vocabulary and compare it with the nonprocessed one:

<eos> frequency=44836 in both

[original]: @-@ f.=20884

[adjusted]: - f.=21166

33275 instead of 33278 words. The difference is negligible.

***However*, we just observed that the FastText vectors also contain vectors for symbols found in WikiText-2:** like @-@, @,@ @.@

**This means that any adjustment is ultimately unnecessary.** For the globals’ task, we can read the WikiText-2 text as it is and even symbols like @-@ will have their own global vector.

We re-compute the globals’ pipeline.

## Using the pre-trained FastText embeddings

Before adding any graph information, we just switch to using the pre-trained FastText embeddings instead of those created by the AWD-LSTM itself.

It is necessary to modify:

The self.encoder in the AWD model object, that has shape=(33278, 400)

It is created as: self.encoder = nn.Embedding(ntoken, ninp)

and it is initialized as:

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

Remember that we have 2 alternatives:

* replacing the embeddings that the AWD-LSTM trains with the FastText embeddings
* Including the FastText embeddings, the graph matrix X and the GNNs *in parallel*

The second alternative is actually more sensible, and does not throw out of the window the embeddings that the AWD-LSTM manages to train.

However, it requires a moment of thought regarding how to modify the architecture.

Among the hyperparameters, args.tied=True.

The original structure is:

[WeightDrop( (module): LSTM(400, 1150)

), WeightDrop( (module): LSTM(1150, 1150)

), WeightDrop( (module): LSTM(1150, 400)

)] followed by a self.decoder= nn.Linear(nhid, ntoken)

a.k.a. 400 -> 1150 -> 1150 -> 400 > 33278

### Idea #1: Add projected input signal to 1st layer output

Keep the tied weights 400->1150 and 1150->400.

However, add *in parallel* to the first layer a Graph Attention Network that provides a d=300 input (the node-state of the current global), and sum the 2 input signals…

We will need to add the graph\_dataobject to the parameters of the AWD model…

done. We also add a variant\_flags\_dict to determine which options we use in the model…

We chose to include the graph-KB/FastText input signal in parallel to the main word embeddings, that the AWD-LSTM should still be able to read and create as usual.

I can project the d300 into 1150 and then *add* it to the output of the 1st layer.

Considering that in the forward() call, the shape of the input is ([72, 32]):

given that among the hyperparameters we have:

**'batch\_size'**:32, *# (orig. 80)***'bptt'**:70,

To find out how to pick the current word, we need to consider the dimensions of the input:

emb.shape = torch.Size([72, 32, 400])

…

raw\_output = emb

…

current\_input = raw\_output

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

then

raw\_output.shape=torch.Size([72, 32, 1150])

so we can set up a temporary version that uses a double for-cycle, and then worry about making it faster with Tensor operations.

Using the steps: for cycle > append to list > torch.cat > Linear( we could also choose to use a LSTM here)

we obtain additional\_contribution\_01.shape= torch.Size([32, 72, 1150])

Let us try summing this quantity to the the output of the 1st layer, and then proceed as previously.

**Review of the modification:**

**for** t **in** range(seq\_len):  
 **for** b **in** range(bsz):  
 **if not** (self.variant\_flags\_dict[**'include\_globalnode\_input'**]):  
 currentword\_embedding = self.X.index\_select(dim=0, index=input[t,b])additional\_input\_signal\_ls.append(currentword\_embedding)  
additional\_input\_signal = torch.cat(additional\_input\_signal\_ls, dim=0) additional\_contribution = self.P(additional\_input\_signal)

…

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

raw\_output, new\_h = rnn(raw\_output, hidden[l])  
*# my modification (insertion of the additional input signal):***if** l== 0:  
 raw\_output = raw\_output + additional\_contribution\_01  
new\_hidden.append(new\_h)

**Experiment:**

Starting epoch 1

| epoch 1 | 200/ 932 batches | lr 30.00000 | ms/batch 2307.79 | loss 8.04 | ppl 3092.42 | bpc 11.595

| epoch 1 | 400/ 932 batches | lr 30.00000 | ms/batch 2313.04 | loss 7.04 | ppl 1146.07 | bpc 10.162

| epoch 1 | 600/ 932 batches | lr 30.00000 | ms/batch 2323.63 | loss 6.82 | ppl 911.50 | bpc 9.832

| epoch 1 | 800/ 932 batches | lr 30.00000 | ms/batch 2346.57 | loss 6.59 | ppl 724.30 | bpc 9.500

| end of epoch 1 | time: **2223**.39s | valid loss 6.13 | valid ppl 459.05 | valid bpc 8.843

| end of epoch 2 | time: 2223.36s | valid loss 5.75 | valid ppl 315.74 | valid bpc 8.303

| end of epoch 3 | time: 2223.41s | valid loss 5.52 | valid ppl 250.07 | valid bpc 7.966

…

| end of epoch 10 | time: 2223.42s | valid loss 4.99 | valid ppl 147.48 | valid bpc 7.204

…

| end of epoch 30 | time: 2224.42s | valid loss 4.75 | valid ppl 115.97 | valid bpc 6.858

| end of epoch 31 | time: 2223.69s | valid loss 4.76 | valid ppl 116.39 | valid bpc 6.863

| end of epoch 32 | time: 2223.61s | valid loss 4.76 | **valid ppl 116.71** | valid bpc 6.867  
Switching to ASGD

…

| end of epoch 50 | time: 2225.96s | valid loss 4.60 | valid ppl 98.99 | valid bpc 6.629

…

| end of epoch 64 | time: 2226.38s | valid loss 4.59 | valid ppl 98.10 | valid bpc 6.616

Comparing it briefly with the original, AWD-LSTM-only:

| end of epoch 1 | time: 86.54s | valid loss 5.99 | valid ppl 398.30 | valid bpc 8.638

…

| end of epoch 32 | time: 86.77s | valid loss 4.60 | **valid ppl 99.17** | valid bpc 6.632

Switching to ASGD

…

| end of epoch 64 | time: 89.62s | valid loss 4.40 | valid ppl 81.43 | valid bpc 6.347

…

| end of epoch 750 | time: 89.32s | valid loss 4.31 | **valid ppl 74.66** | valid bpc 6.222

### Issues:

We are notably slower, 2.2K seconds =~37minutes for 1 epoch as opposed to ~86seconds.

The validation PPL is also worse.

We should: use batch processing to make it faster + change how we include the FastText input + verify the input indices & words of our vocabulary.’

**Verify input indices & vocabulary**

At the start, we have input[0:5,0:4]=

tensor([[ 0, 8043, 652, 37],

[ 1, 1238, 2349, 17],

[ 2, 61, 1986, 8597],

[ 3, 15, 665, 115],

[ 4, 135, 3932, 17]])

We consult our vocabulary and obtain:

|  |  |  |  |
| --- | --- | --- | --- |
| <eos> | surprising | home | and |
| = | authority | 1933 | the |
| Valkyria | " | she | acclaim |
| Chronicles | . | made | from |
| III | It | port | the |

It corresponds to different places in the wiki.train.tokens file. However, a more thorough check can be made if we add a debug-print of the entire input matrix using our vocabulary:

['<eos>', '=', 'Valkyria', 'Chronicles', 'III', '=', '<eos>', '<eos>', 'Senjō', 'no', 'Valkyria', '3', ':', '<unk>', 'Chronicles', '(', 'Japanese', ':', '戦場のヴァルキュリア3', ',', 'lit', '.', 'Valkyria', 'of', 'the', 'Battlefield', '3', ')', ',', 'commonly', 'referred', 'to', 'as', 'Valkyria', 'Chronicles', 'III', 'outside', 'Japan', ',', 'is', 'a', 'tactical', 'role', '@-@', 'playing', 'video', 'game', 'developed', 'by', 'Sega', 'and', 'Media.Vision', 'for', 'the', 'PlayStation', 'Portable', '.', 'Released', 'in', 'January', '2011', 'in', 'Japan', ',', 'it', 'is', 'the', 'third', 'game', 'in', 'the', 'Valkyria']

['surprising', 'authority', '"', '.', 'It', 'is', 'Balaguer', 'who', 'guides', 'much', 'of', 'the', 'action', 'in', 'the', 'last', 'sections', 'of', 'the', 'book', '.', '<eos>', '<eos>', '=', '=', '=', '<unk>', '=', '=', '=', '<eos>', '<eos>', 'The', 'storyline', 'concerning', 'the', 'assassination', 'primarily', 'follows', 'the', 'four', 'conspirators', 'who', 'directly', 'participate', 'in', 'Trujillo', "'s", 'death', '.', 'Antonio', 'Imbert', '<unk>', 'is', 'one', 'of', 'the', 'few', 'conspirators', 'who', 'survives', 'the', 'violent', '<unk>', 'that', 'follow', 'Trujillo', "'s", 'assassination', '.', 'Imbert', 'is']

['In', '1933', 'she', 'made', 'port', 'visits', 'in', 'Turkey', ',', 'Greece', 'and', 'Italy', '.', '<eos>', 'She', 'was', 'refitted', 'before', 'Operation', 'Barbarossa', ',', 'probably', 'about', '1940', ',', 'her', 'catapult', 'was', 'removed', ',', 'and', 'her', 'anti', '@-@', 'aircraft', 'armament', 'was', 'greatly', 'increased', '.', 'Her', 'four', '76', '@.@', '2', 'mm', '<unk>', 'AA', 'guns', 'were', 'exchanged', 'for', 'four', 'Italian', '<unk>', 'twin', 'gun', '50', '@-@', 'caliber', '100', 'mm', '(', '3', '@.@', '9', 'in', ')', 'AA', 'mounts', 'and', 'she']

['and', 'the', 'acclaim', 'from', 'the', 'film', 'launched', 'his', 'career', '.', '<eos>', '<eos>', '=', '=', '=', '=', 'Boogie', 'Nights', '=', '=', '=', '=', '<eos>', '<eos>', 'Anderson', 'began', 'working', 'on', 'the', 'script', 'for', 'his', 'next', 'feature', 'film', 'during', 'his', 'troubles', 'with', 'Hard', 'Eight', ',', 'completing', 'the', 'script', 'in', '1995', '.', 'The', 'result', 'was', 'Anderson', "'s", 'breakout', 'for', 'the', 'drama', 'film', 'Boogie', 'Nights', '(', '1997', ')', ',', 'which', 'is', 'based', 'on', 'his', 'short', 'The', 'Dirk']

['and', 'career', 'of', 'Norman', 'Finkelstein', ',', 'released', 'in', '2009', 'and', 'directed', 'by', 'David', '<unk>', 'and', 'Nicolas', '<unk>', '.', 'It', 'has', 'been', 'screened', 'in', 'Amsterdam', '<unk>', ',', 'in', 'Toronto', 'Hot', '<unk>', 'and', 'in', 'more', 'than', '40', 'other', 'national', 'and', 'international', 'venues', ',', 'it', 'received', 'a', 'freshness', 'rating', 'of', '100', '%', 'on', 'film', 'review', 'aggregator', 'Rotten', 'Tomatoes', '.', 'The', 'same', 'year', 'Finkelstein', 'appeared', 'in', 'Defamation', '(', 'Hebrew', ':', '<unk>', ';', 'translit', '.', '<unk>', ')']

['but', 'is', 'itself', 'single', 'sentence', 'spaced', '.', 'The', 'German', 'language', 'manual', '<unk>', 'des', '<unk>', 'für', 'Deutsche', '<unk>', '(', '"', '<unk>', 'of', 'the', 'Council', 'for', 'German', '<unk>', '"', ')', '(', '2006', ')', 'does', 'not', 'address', 'sentence', 'spacing', '.', 'The', 'manual', 'itself', 'uses', 'one', 'space', 'after', 'terminal', 'punctuation', '.', 'Additionally', ',', 'the', '<unk>', ',', 'the', 'German', 'language', 'dictionary', 'most', 'commonly', 'used', 'in', 'Germany', ',', 'indicates', 'that', 'double', 'sentence', 'spacing', 'is', 'an', 'error', '.', '<eos>']

['turn', 'towards', 'the', 'French', 'and', 'for', 'each', 'of', 'his', 'vessels', 'to', 'rake', 'and', 'engage', 'their', 'immediate', 'opponent', '.', 'This', 'unexpected', 'order', 'was', 'not', 'understood', 'by', 'all', 'of', 'his', 'captains', ',', 'and', 'as', 'a', 'result', 'his', 'attack', 'was', 'more', '<unk>', 'than', 'he', 'intended', '.', 'Nevertheless', ',', 'his', 'ships', 'inflicted', 'a', 'severe', 'tactical', 'defeat', 'on', 'the', 'French', 'fleet', '.', 'In', 'the', 'aftermath', 'of', 'the', 'battle', 'both', 'fleets', 'were', 'left', 'shattered', ';', 'in', 'no', 'condition']

[',', 'allowing', 'Mega', 'Man', 'and', 'Bass', 'to', 'best', 'King', 'in', 'battle', 'afterwards', '.', 'King', 'questions', 'why', 'they', 'fight', 'so', 'hard', 'for', 'humans', 'when', 'robots', 'are', 'the', 'superior', 'species', '.', 'The', 'pair', 'explains', 'that', 'humans', 'are', 'the', 'ones', 'who', 'created', 'robots', 'in', 'the', 'first', 'place', ',', 'which', '<unk>', 'King', '.', 'The', 'villain', 'reveals', 'that', 'his', 'creator', 'is', 'Dr.', 'Wily', ',', 'who', 'then', 'appears', 'on', 'a', 'video', 'monitor', '.', 'When', 'King', 'asks', 'the', 'evil']

We print every time-sequence in the batch. While the sequences are internally consistent, and our vocabulary retrieves the words correctly, they do not appear to be contiguous…

There is still the question of whether this method of creating the batching and sending the input is also the one that worked in the original model in PyTorch 0.4.

I did not modify anything in this regard. Neither presumably did the author of the port.

We can assume that she would have obtained much worse PPL results, instead of being close and comparable?

We can check this in a moment of ‘downtime’. For now, we verified that our vocabulary is actually identical to the one used by AWD-LSTM.

**Improve speed with tensor operations and/or batch processing**

Using

time\_t\_word\_embeddings = self.X.index\_select(dim=0, index=input[t, :])

we iterate across the time instants, collect the embeddings across the batches, and then torch.cat them and – as previously – reshape/view > project to 1150 > add to output of 1st layer.

It is necessary to:

start the experiment again, and verify that:

1. we get a speed improvement
2. the perplexity results are similar / close enough.

We get:

| epoch 1 | 200/ 932 batches | lr 30.00000 | ms/batch 153.39 | loss 8.07 | ppl 3185.16 | bpc 11.637

| epoch 1 | 400/ 932 batches | lr 30.00000 | ms/batch 154.54 | loss 7.02 | ppl 1120.63 | bpc 10.130

| epoch 1 | 600/ 932 batches | lr 30.00000 | ms/batch 155.84 | loss 6.73 | ppl 837.46 | bpc 9.710

| epoch 1 | 800/ 932 batches | lr 30.00000 | ms/batch 157.78 | loss 6.49 | ppl 657.67 | bpc 9.361

| end of epoch 1 | time: 154.22s | valid loss 6.04 | valid ppl 417.82 | valid bpc 8.707

In the baseline:

ms/batch ~86s (so we managed to bring it in the same order of magnitude, at ~155s instead of 2200s)

epoch 1 training ppl: 3401.69, 1162.06, 844.84, 681.67

epoch 1 valid ppl: 398.30

In the previous attempt of this variant, we got:

epoch 1 training ppl: 3092.42, 911, 724.30

epoch 1 valid ppl 459.05

The values are all comparable, and the random initialization even gives slightly better results compared to the previous attempt.

The batching on FastText embedding now works (the next problem will be using batches on GAT…)

### Idea #2: Concatenate the input signal to the word embedding

d=400 ++ d=300 == d=700

Consequently, we change the network structure into:

700 -> 1150 -> 1150 -> 400 > 33278

Objectives:

1. When adding the pre-trained FastText embedding, we should have comparable PPL
2. When including the GAT global node-state, we should improve the PPL

**Experiment a)**

Core of the model:

[WeightDrop(

(module): LSTM(700, 1150)

), WeightDrop(

(module): LSTM(1150, 1150)

), WeightDrop(

(module): LSTM(1150, 400)

)]

| end of epoch 1 | time: 149.24s | valid loss 6.02 | valid ppl 410.60 | valid bpc 8.682

…

| end of epoch 10 | time: 151.22s | valid loss 4.77 | valid ppl 118.09 | valid bpc 6.884

…

| end of epoch 32 | time: 157.07s | valid loss 4.48 | **valid ppl 88.24** | valid bpc 6.463

…

| end of epoch 67 | time: 157.06s | valid loss 4.46 | **valid ppl 86.67** | valid bpc 6.438

| end of epoch 68 | time: 157.17s | valid loss 4.46 | valid ppl 86.68 | valid bpc 6.438

…

| end of epoch 150 | time: 159.76s | valid loss 4.51 | valid ppl 91.09 | valid bpc 6.509

…

and then the Valid PPL gets worse

## Comparison of the methods of inclusion of FastText embeddings into AWD-LSTM

**Baseline:**

@ epoch 32:

| epoch 32 | 800/ 932 batches | lr 30 | ms/batch 85.71 | loss 4.46 | **ppl 86.35** | bpc 6.432|

end of epoch 32 | time: 86.77s | valid loss 4.60 | **valid ppl 99.17** | valid bpc 6.632

@ best:

| epoch 750 | 800/ 932 batches | lr 30 | ms/batch 88.04 | loss 3.96 | **ppl 52.45** | bpc 5.713|

end of epoch 750 | time: 89.32s | valid loss 4.31 | **valid ppl 74.66** | valid bpc 6.222

**Project 300 > 1150 and add to the output of the 1st layer**:

@epoch 32:

800/ 932 batches , train ppl **69.09** || valid ppl **116.71**

@ best, epoch 64:

800/932 batches, train ppl **57.59** || valid ppl **98.10**

**Concatenate 300++400 and operate with 700>1150>1150>400** :

@epoch 32:

800/ 932 batches , train ppl **51.15** || valid ppl **88.24**

@ best:

800/932 batches, train ppl **37.36** || valid ppl **86.67**

Concatenating the FastText embedding to the awd-lstm’s embedding gives the most interesting result:

we overfit faster, and then we are unable to learn further.

It is worth checking whether things improve when we apply the same dropout used on the original embeddings.

We also remember that even a slightly-worse result is acceptable, the real experiment is the one that uses the GAT and global node-state.

Although self.dropoute=0.1, would it really have such a strong impact?

I could also not use tied weights…

# LSTM on WT-2, version 2.1

## Experiment 2.1.1.

### Model

INFO : DataParallel(

(module): RNN(

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

(main\_rnn\_ls): ModuleList(

(0): LSTM(300, 1024)

(1): LSTM(1024, 1024)

(2): LSTM(1024, 512)

)

(linear2global): Linear(in\_features=512, out\_features=33278, bias=True))

INFO : Parameters:

INFO : ('module.X', torch.Size([116153, 300]), torch.float32, True)

('module.select\_first\_indices', torch.Size([1024]), torch.float32, False)

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

('module.memory\_hn', torch.Size([3, 40, 1024]), torch.float32, False)

('module.memory\_cn', torch.Size([3, 40, 1024]), torch.float32, False)

('module.memory\_hn\_senses', torch.Size([2, 40, 1024]), torch.float32, False)

('module.memory\_cn\_senses', torch.Size([2, 40, 1024]), torch.float32, False)

('module.main\_rnn\_ls.0.weight\_ih\_l0', torch.Size([4096, 300]), torch.float32, True)

('module.main\_rnn\_ls.0.weight\_hh\_l0', torch.Size([4096, 1024]), torch.float32, True)

('module.main\_rnn\_ls.0.bias\_ih\_l0', torch.Size([4096]), torch.float32, True)

('module.main\_rnn\_ls.0.bias\_hh\_l0', torch.Size([4096]), torch.float32, True)

('module.main\_rnn\_ls.1.weight\_ih\_l0', torch.Size([4096, 1024]), torch.float32, True)

('module.main\_rnn\_ls.1.weight\_hh\_l0', torch.Size([4096, 1024]), torch.float32, True)

('module.main\_rnn\_ls.1.bias\_ih\_l0', torch.Size([4096]), torch.float32, True)

('module.main\_rnn\_ls.1.bias\_hh\_l0', torch.Size([4096]), torch.float32, True)

('module.main\_rnn\_ls.2.weight\_ih\_l0', torch.Size([2048, 1024]), torch.float32, True)

('module.main\_rnn\_ls.2.weight\_hh\_l0', torch.Size([2048, 512]), torch.float32, True)

('module.main\_rnn\_ls.2.bias\_ih\_l0', torch.Size([2048]), torch.float32, True)

('module.main\_rnn\_ls.2.bias\_hh\_l0', torch.Size([2048]), torch.float32, True)

('module.linear2global.weight', torch.Size([33278, 512]), torch.float32, True)

('module.linear2global.bias', torch.Size([33278]), torch.float32, True)

INFO : Number of trainable parameters=68.895M, where embeddings=34.846M, softmax=17.072M, core=16.977M

### Run

|  |  |  |
| --- | --- | --- |
| **Architecture** | **Input Signals** | **Hyperparameters** |
| model: LSTM | 1. The word embedding of the current global   (d=300) | batch\_size=40 |
| **LSTM** with 3 layers (1024, 1024, 512). | TBPTT length=70 |
| Followed by linear2Globals **FF-NN** | learning rate=5\*10^(-5) |

INFO : Hyperparameters: \_batchPerSeqlen2800\_area32\_lr5e-05\_epochs100

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| *Epoch* |  | | | | | | |
|  | *Globals: Training* | | *Globals: Validation* | | | | |
|  | *PPL* | *Correct / total* | | *Top-10 / total* | *PPL* | *Correct / total* | *Top-10 / total* |
| 1 | 1686.84 | / **2052400** | | / 2052400 | 976.84 | / **215600** | / 215600 |
| 2 | 1251.78 |  | |  | 976.18 |  |  |
| 3 | 1237.52 |  | |  | 973.27 |  |  |
| 4 | 1062.72 |  | |  | 724.81 |  |  |
| 5 | 825.32 |  | |  | 618.31 |  |  |
| 6 | 711.48 |  | |  | 545.97 |  |  |
| 7 | 633.01 |  | |  | 495.14 |  |  |
| 8 | 576.04 |  | |  | 460.83 |  |  |
| 9 | 530.21 |  | |  | 430.51 |  |  |
| 10 | 490.31 |  | |  | 402.47 |  |  |
| 11 | 455.68 |  | |  | 380.2 |  |  |
| 12 | 426.28 |  | |  | 362.82 |  |  |
| 13 | 401.12 |  | |  | 346.83 |  |  |
| 14 | 379.37 |  | |  | 332.84 |  |  |
| 15 | 360.39 |  | |  | 321.92 |  |  |
| 16 | 343.56 |  | |  | 313.52 |  |  |
| 17 | 328.58 |  | |  | 304.11 |  |  |
| 18 | 314.81 |  | |  | 297.62 |  |  |
| 19 | 302.33 |  | |  | 290.95 |  |  |
| 20 | 290.98 |  | |  | 282.8 |  |  |
| 21 | 280.45 |  | |  | 277.07 |  |  |
| 22 | 270.67 |  | |  | 272.12 |  |  |
| 23 | 261.49 |  | |  | 268.16 |  |  |
| 24 | 252.92 |  | |  | 264.01 |  |  |
| 25 | 244.89 |  | |  | 258.93 |  |  |
| 26 | 237.24 |  | |  | 253.9 |  |  |
| 27 | 230.06 |  | |  | 248.81 |  |  |
| 28 | 223.3 |  | |  | 246.42 |  |  |
| 29 | 217.11 |  | |  | 244.43 |  |  |
| 30 | 211.44 |  | |  | 242.95 |  |  |

and so on and so forth.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 43 | 155.17 |  |  | 221.02 |  |

With batch\_size=40 and seq\_len=70, the learning rate value of 5\*10^(-5) is too low. We increase it to 10^(-4) and repeat the experiment.

(note: the values 40x70 were taken from the original 2017 AWD-LSTM paper.)

## Experiment 2.1.2

|  |  |  |
| --- | --- | --- |
| **Architecture** | **Input Signals** | **Hyperparameters** |
| model: LSTM | 1. The word embedding of the current global   (d=300) | batch\_size=40 |
| **LSTM** with 3 layers (1024, 1024, 512). | TBPTT length=70 |
| Followed by linear2Globals **FF-NN** | learning rate=10^(-4) |

INFO : Hyperparameters: \_batchPerSeqlen2800\_area32\_lr5e-05\_epochs100

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| *Epoch* |  | | | | | | |
|  | *Globals: Training* | | *Globals: Validation* | | | | |
|  | *PPL* | *Correct / total* | | *Top-10 / total* | *PPL* | *Correct / total* | *Top-10 / total* |
| 1 | 1551.72 | / **2052400** | | / 2052400 | 988.3 | / **215600** | / 215600 |
| 2 | 1259.24 |  | |  | 986.31 |  |  |
| 3 | 1246.74 |  | |  | 983.5 |  |  |
| 4 | 1128.83 |  | |  | 715.87 |  |  |
| 5 | 725.18 |  | |  | 514.36 |  |  |
| 6 | 546.81 |  | |  | 421.14 |  |  |
| 7 | 451.13 |  | |  | 369.84 |  |  |
| 8 | 388.96 |  | |  | 337.94 |  |  |
| 9 | 345.15 |  | |  | 312.57 |  |  |
| 10 | 310.64 |  | |  | 291.86 |  |  |
| 11 | 282.7 |  | |  | 274.72 |  |  |
| 12 | 259.62 |  | |  | 261.94 |  |  |
| 13 | 239.22 |  | |  | 250.07 |  |  |
| 14 | 221.67 |  | |  | 240.3 |  |  |
| 15 | 206.38 |  | |  | 231.51 |  |  |
| 16 | 192.78 |  | |  | 226.07 |  |  |
| 17 | 180.84 |  | |  | 221.36 |  |  |
| 18 | 169.89 |  | |  | 220.82 |  |  |
| 19 | 160.46 |  | |  | 217.09 |  |  |
| 20 | 151.53 |  | |  | 211.36 |  |  |
| 21 | 143.4 |  | |  | 208.17 |  |  |
| 22 | 135.72 |  | |  | 203.61 |  |  |
| 23 | 129.47 |  | |  | 201.19 |  |  |
| 24 | 123.17 |  | |  | 198.82 |  |  |
| 25 | 117.68 |  | |  | 195.98 |  |  |
| 26 | 112.3 |  | |  | 194.81 |  |  |
| 27 | 107.11 |  | |  | 193.27 |  |  |
| 28 | 102.55 |  | |  | 192.0 |  |  |
| 29 | 98.5 |  | |  | 190.14 |  |  |
| 30 | 94.56 |  | |  | 189.04 |  |  |
| 31 | 90.69 |  | |  | 189.33 |  |  |
| 32 | 87.12 |  | |  | 189.39 |  |  |
| 33 | 83.82 |  | |  | 190.11 |  |  |
| 34 | 80.65 | 551203 | | 1109417 | **188.48** | 48929 | 106197 |
| 35 | 77.5 |  | |  | 188.94 |  |  |
| 36 | 74.53 |  | |  | 190.59 |  |  |
| 37 | 71.46 |  | |  | 191.91 |  |  |

The result is relatively disappointing compared to previous experiments.

An LSTM with 1000>1000>500, dropout p=0.2, batch\_size=40, seq\_len=35, lr=5e-5 previously got **163.27** Valid PPL.

***However***, at the time we operated with a vocabulary without the long tail (*min. frequency=2*), that had only 31K globals instead of 33.2K. Thus, we can assume the task was easier.

For the next experiment, we change:

the seq\_len from 70 to 35 and thus decrease the learning rate from e-4 to 5e-5, because there is a chance the better results of the other setting were also due to this.

The structure of the network, going to 1024 > 1024 > 1024, because in the past a 1150x3 GRU got to 163.7 Valid-PPL and then the GRU\_GAT improved to 159.7.

However, in the meantime, it can be useful to speed up the computation, following the methods I have seen in awd\_lstm\_lm.

## Faster computations

Currently the input in the RNN (and also in the Senses.SelectK model elsewhere) is:

**for** padded\_sequence **in** sequences\_in\_the\_batch\_ls:  
 padded\_sequence = padded\_sequence.squeeze(dim=0)  
 padded\_sequence = padded\_sequence.chunk(chunks=padded\_sequence.shape[0], dim=0)  
 sequence\_lts = [unpack\_input\_tensor(sample\_tensor, self.grapharea\_size) **for** sample\_tensor **in** padded\_sequence]

**for** ((x\_indices\_g, edge\_index\_g, edge\_type\_g),   
 (x\_indices\_s, edge\_index\_s, edge\_type\_s)) **in** sequence\_lts:

*# Input signal n.1: the embedding of the current (global) word* currentword\_embedding = self.X.index\_select(dim=0, index=x\_indices\_g[0])  
  
 *# Input signal n.2: the node-state of the current global word* **if** self.include\_globalnode\_input:

etc…

We examine sequence\_lts.

In our example, we have distributed\_batch\_size=2 and we have set sequence\_length=3.

sequences\_in\_the\_batch\_ls is a list containing 2 tensors. (this batch contains 2 sequences, because batch\_size=2).

sequence\_lts has 3 elements, because seq\_len=3.

each one of them is a tuple of 2 input tuples, that contain x\_indices, edge\_index, edge\_type for globals and senses.

Let us review the process that starts from the batchinput\_tensor, and develop from there:

batchinput\_tensor.shape=torch.Size([2, 3, 1150]) ( (bsz, sqlen, dims), as it is before the inversion)

time\_instants = 3 tensors of (2,1,1150)

for every time instant: (**for** batch\_elements\_at\_t **in** time\_instants):

batch\_elems\_at\_t.shape=torch.Size([2, 1150])

elems\_at\_t\_ls[0].shape=torch.Size([1, 1, 1150])

elems\_at\_t\_ls[1].shape=torch.Size([1, 1, 1150])

After we unpack, we obtain:

t\_input\_signals\_lts (temporary name):

list of length 2.

Each of the 2 elements is a tuple containing the 2 input tuples for globals and senses.

When we operate with the FastText embeddings alone, we have:

batch\_input\_signals.shape = (3,2,300) = (seq\_len, bsz, dim)

Further notes, on the possibility of batching in pytorch-geometric & GNNs:

conv = GCNConv(in\_channels, out\_channels, node\_dim=1)

conv(x, edge\_index) # here, x is a tensor of size [batch\_size, num\_nodes, num\_features]

x\_padded.size=(2,32,300)

t\_edgeindex\_g\_size = (2, 2, 180)

These are passed on to GATConv as x and edge\_index…

after remove\_self\_loops, edge\_index.shape=(2,0)

after add\_self\_loops, edge\_index.shape= torch.Size([2, 32])

Then,

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

gives us x.shape=torch.Size([2, 32, 300])

In the propagate() function: in the message() function: when computing the attention coefficients:

x\_j.shape=torch.Size([64, 4, 75])

with a structure like:

tensor([[[-0.0611, 0.0256, -0.0906, ..., -0.0397, 0.0553, 0.0074],

[ 0.1738, -0.0726, -0.1257, ..., -0.0007, -0.0911, 0.1103],

[-0.0553, 0.0032, 0.0947, ..., 0.0507, 0.0373, -0.0074],

[ 0.0390, 0.0184, -0.0235, ..., 0.0960, -0.0880, -0.0353]],

[[ 0.0000, 0.0000, 0.0000, ..., 0.0000, 0.0000, 0.0000],

[ 0.0000, 0.0000, 0.0000, ..., 0.0000, 0.0000, 0.0000],

[ 0.0000, 0.0000, 0.0000, ..., 0.0000, 0.0000, 0.0000],

[ 0.0000, 0.0000, 0.0000, ..., 0.0000, 0.0000, 0.0000]], …….

Shapes:

alpha: (64,4)

edge\_index\_i: 32

size\_i=32

In:

alpha = softmax(alpha, edge\_index\_i, size\_i)

…

return torch.ops.torch\_scatter.scatter\_max(src, index, dim, out, dim\_size)

RuntimeError: The expanded size of the tensor (64) must match the existing size (32) at non-singleton dimension 0. Target sizes: [64, 4]. Tensor sizes: [32, 1]

x\_j = x\_j.view(-1, self.heads, self.out\_channels)

becomes of torch.Size([128, 4, 75])

Should I unify the edge\_index for the batch?

The example on the PyTorch-Geometric forum operated with different node features but the same graph structure, and it said:

Alternative (1) is Replicating your edge\_index by stacking them diagonally, *e.g.*, via:

batch\_edge\_index = Batch.from\_data\_list([Data(edge\_index=edge\_index)] \* batch\_size)

Currently using the torch\_geometric Batch facilities.

## Experiments

### Model

Version 2 of the model, 1024x3, that uses the new batching.

### Experiment 2.1.3

|  |  |  |
| --- | --- | --- |
| **Architecture** | **Input Signals** | **Hyperparameters** |
|  | 1. The word embedding of the current global   (d=300) | batch\_size=40 |
| LSTM with 3 layers (1024, 1024, **1024**). | TBPTT length=**35** |
| Followed by linear2Globals FF-NN | learning rate=**5\*10^(-5)** |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| *Epoch* |  | | | | | | |
|  | *Globals: Training* | | *Globals: Validation* | | | | |
|  | *PPL* | *Correct / total* | | *Top-10 / total* | *PPL* | *Correct / total* | *Top-10 / total* |
| 1 | 1492.12 | 109729 / **2052400** | | 593954 / 2052400 | 986.09 | 10507 / **214200** | 69320 / 214200 |
| … |  |  | |  |  |  |  |
| 10 | 220.67 | 410531 | | 919447 | 259.06 | 44184 | 98587 |
| … |  |  | |  |  |  |  |
| 15 | 147.48 |  | |  | 218.06 |  |  |
| 16 | 137.79 |  | |  | 214.01 |  |  |
| 17 | 129.29 |  | |  | 209.41 |  |  |
| 18 | 121.58 |  | |  | 205.5 |  |  |
| 19 | 114.34 |  | |  | 202.54 |  |  |
| 20 | 107.98 |  | |  | 199.9 |  |  |
| 21 | 102.15 |  | |  | 197.71 |  |  |
| 22 | 96.72 |  | |  | 195.59 |  |  |
| 23 | 91.78 |  | |  | 196.58 |  |  |
| 24 | 87.09 |  | |  | 195.61 |  |  |
| 25 | 82.79 |  | |  | 194.59 |  |  |
| 26 | 78.82 |  | |  | 194.14 |  |  |
| 27 | 75.11 |  | |  | 194.29 |  |  |
| 28 | 71.61 | 542757 | | 1137084 | **192.92** | 48004 | 105263 |
| 29 | 68.29 |  | |  | 193.31 |  |  |
| 30 | 65.18 |  | |  | 193.87 |  |  |
| 31 | 62.25 |  | |  | 194.34 |  |  |

### Comparison between LSTM architectures

Using as input signal only the FastText word embedding of the current global,

The LSTM with 3 layers (1024, 1024, 512), bsz=40 x seq\_len=70, lr=10^(-4) gets:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| *Epoch* | *PPL* | *Correct / total* | *Top-10 / total* | *PPL* | *Correct / total* | *Top-10 / total* |
| 34 | 80.65 | 551,203 | 1,109,417 | **188.48** | 48,929 | 106,197 |

The LSTM with 3 layers (1024, 1024, 1024), bsz=40 x seq\_len=35, lr=5e-5 gets:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| *Epoch* | *PPL* | *Correct / total* | *Top-10 / total* | *PPL* | *Correct / total* | *Top-10 / total* |
| 28 | 71.61 | 542,757 | 1,137,084 | **192.92** | 48,004 | 105,263 |

Architecture #1 performs slightly better, so we will use that one for comparisons with the inclusion of the graph input (LSTM+GAT), and for any GRU experiments.

### **Aside:** Note on the inclusion of lemmatization

‘as’, the 23rd global, gets lemmatized as ‘a’, which is clearly incorrect.

Possible way to solve it: if a word has edges that are *not* self-loops, do not lemmatize it.

‘is’ gets lemmatized as ‘be’, and this is correct. The edge\_index is tensor([[0], [0]],)

Double-checking Lemmatization:

word=no ; lemmatized\_word= no

INFO : word has edges that are not all self-loops

INFO : \*\*\*

word=Chronicles ; lemmatized\_word= Chronicles

INFO : \*\*\*

word=is ; lemmatized\_word= be

INFO : Getting the data for the lemmatized word

INFO : \*\*\*

word=Battlefield ; lemmatized\_word= Battlefield

INFO : \*\*\*

word=referred ; lemmatized\_word= refer

INFO : Getting the data for the lemmatized word

INFO : \*\*\*

ord=follows ; lemmatized\_word= follow

INFO : Getting the data for the lemmatized word

INFO : \*\*\*

word=gameplay ; lemmatized\_word= gameplay

INFO : \*\*\*

word=the ; lemmatized\_word= the

INFO : \*\*\*

etc.. Correct, as far as we can see.

Meeting with IA, 23/07:

1. Add a dummy sense label. This will be used by the GRU\_senses.
2. As a consequence, I must compute 2 perplexities on the GRU\_senses: one for all the words in the document, and another only for the words that have multiple senses.
3. For the AWD-LSTM, I can try using 2 models in parallel, one from the d400 embeddings and and one from the d300 FastText embeddings. Each one has tied weights, identical hyperparameters etc.  
   Then, as a transfer learning method: use the weighted, learned average of the softmax (or logits, depending on how it works numerically) coming from the 2 models.  
   This weight can be 1 number, influenced by… yet another AWD-LSTM? With which input? the 400 or the 300? Possibly the concatenation of the last encoding 400+300, with a 1-layer LSTM…

# AWD-LSTM, version 3

## AWD\_modified

### Embeddings

We have an embeddings matrix for: senses, globals, definitions, examples. For the GAT, we index\_select some of its rows.

self.X = Parameter(graph\_dataobj.x.clone().detach(), requires\_grad=**True**)

The original AWD uses an

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

initialized with

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

We should tie the look-up table in the Embedding object to the globals’ part of the matrix X, namely X[self.last\_idx\_senses:self.last\_idx\_globals,:].

We observe that the encoder and decoder weights are tied as:

**if** tie\_weights:self.decoder.weight = self.encoder.weight

Maybe self.X.index\_select(index=torch.tensor(list(range(self.last\_idx\_senses,self.last\_idx\_globals))), dim=0)

?

However,

torch.index\_select(*input*, *dim*, *index*, *out=None*) → Tensor

Returns a **new** tensor which indexes the input tensor along dimension dim using the entries in index which is a *LongTensor*.

Possible solution:

manually modify the look-up table, so that it picks in the globals’ section of the matrix X.

Examining the steps in **embed\_regularize.py**:

def embedded\_dropout(embed, words, dropout=0.1, scale=**None**):

where:

embed=Embedding(33278, 400), words=tensor of torch.Size([72, 32]) (numerical indices)

mask= tensor of torch.Size([33278, 400]) with a lot of 1.111

masked\_embed\_weight = mask \* embed.weight

followed by:

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

so that eventually X is a tensor of torch.Size([72, 32, 400])

how many of these parameters are actually in use?

embed.max\_norm=None

embed.**norm\_type**=2.0

embed.scale\_grad\_by\_freq=False

embed.sparse=False

**Idea #1**:

When I use the FastText Embeddings alone, I can just “load” the globals from X into a nn.Embedding object. Then, the nn.Embedding will be updated through the gradient, and the X matrix will be unused.

When I include the graph input: it applies a GraphAttentionNetwork on (sample\_x, sample\_edge\_index), as we did in the other models

– brief review –

After we got (x\_indices\_g, edge\_index\_g, edge\_type\_g) either from the input itself or AD.get\_node\_data(…) after lemmatization,

we proceeded

x = self.X.index\_select(dim=0, index=x\_indices\_g.squeeze())  
x\_attention\_state = self.gat\_globals(x, edge\_index\_g)  
currentglobal\_node\_state = x\_attention\_state.index\_select(dim=0,  
 index=self.select\_first\_indices[0].to(torch.int64))

– end of brief review –

After this computation, that uses the matrix X, X has its gradient up-to-date; the nn.Embeddings do not. So, it is necessary to copy-paste (must keep the gradient) the resulting node embeddings (the x\_attention\_state for all the globals involved) into the 2nd part of the tensors in the nn.Embedding object, that is of shape [33278,600].

We can use a masked\_scatter, that updates a row {rewrites the FastText embedding | writes the nodestate}

### Forward() call and embeddings update

We may iterate over the time steps, in order to proceed using batch-parallelism and thus be faster. **for** t **in** range(seq\_len): …

For instance:

input.shape= torch.Size([65, 32]) #(numerical indices)

input[t,:] has 32 elements

Let us check the retrieval of the global node-states:

word=’<eos>’

x\_indices\_g = tensor([ 0, 61419, 49095]) # does not change after lemmatization.

sample\_x=tensor of torch.Size([3, 300])

we get a mini sample\_graph

I am retrieving the nodes with a nested 2x for cycle. This may be slow. However, the GAT is executed for all the batch elements at that time instant

**{Idea**:

I could build a **pre-made table/lts of the lemmatized forms** of each word, so I don’t have to do that at runtime.

The insertion point for this would be the moment of vocabulary creation.

**}**

The batch\_graph has num\_graphs=2304 (=72\*32)

The resulting x\_attention\_states are of torch.Size([9581, 300]). The size is not fixed because these are all the globals made of (current global + neighbours).

At this point, we could collect all the globals’ indices and then make a scatter-update on the 2nd part [:,300:600] of the nn.Embeddings, so that the nn.Embeddings will then be used for the LSTMs.

Question: **and what if some of the globals appear twice?** Even without considering the neighbours, we may have the current\_global appear more than once in the current batch. I would have 2 attention states…

choice 1) assign the average ; choice 2) assign the last index, that comes later in the time sequence.

Following the logic of this architecture used in other models, I do not need the neighbours. I just need to transform the d300 input (FastText embedding) into d600 (current global FastText embedding ++ current global node-state).

currentglobal\_node\_states has torch.Size([2304, 300])

Then at

self.encoder.weight.scatter\_(dim=0, index=currentglobal\_nodestate\_update, src=currentglobal\_nodestate\_update)

we get

{RuntimeError}a leaf Variable that requires grad is being used in an in-place operation.

Following what I did for the hidden states of the RNN, I try to use:

self.encoder.weight.data.scatter\_(dim=0, index=current\_global\_indices, src=currentglobal\_nodestate\_update.clone())

but there is a dimensions mismatch because current\_global\_indices has shape ([2304]) and currentglobal\_nodestate\_update has shape [2304,600].

Maybe I can proceed:

(sort the current\_global\_indices)

make a bool tensor, with True if index in current\_global\_indices

expand it to 2304, 600.

Use it as 2D mask

### Results and combining softmaxes

The model’s forward returns result, hidden.

result.shape=2304, 300

hidden = list of 3 tuples – 1 tuple for each layer, containing the 2 tensors hn and cn.

Which have shapes ([1, 32, 1150]), ([1, 32, 1150]), ([1, 32, 300])

Let us now review

raw\_loss = criterion(model.decoder.weight, model.decoder.bias, output, targets)

to understand how the output of the 2 models can be mixed.

In splitcross.py:

split\_targets, split\_hiddens = self.split\_on\_targets(hiddens, targets)

where hiddens.shape=torch.Size([2304, 300]) and targets.shape=torch.Size([2304])

In this particular case, we have no modifications to the shapes occurring in the function. It’s because self.splits= [0, 100 000 000]

*# We only add the tombstones if we have more than one split*

…

combo.shape=torch.Size([2304, 300])

head\_weight.shape=torch.Size([33278, 300]) # (this comes from the decoder)

all\_head\_res = torch.nn.functional.linear(combo, head\_weight, bias=head\_bias)

all\_head\_res.shape=torch.Size([2304, 33278])

Then we finally have:

softmaxed\_all\_head\_res = torch.nn.functional.log\_softmax(all\_head\_res, dim=-1)

What follows is:

softmaxed\_head\_res = softmaxed\_all\_head\_res[running\_offset:running\_offset + len(split\_hiddens[idx])]  
entropy = -torch.gather(softmaxed\_head\_res, dim=1, index=split\_targets[idx].view(-1, 1))

…

total\_loss = entropy.float().sum() **if** total\_loss **is None else** total\_loss + entropy.float().sum()

# AWD-LSTM, ensemble model

ToDo list of tasks:

* start experiment on AWD-LSTM (AWD, original unchanged model) to be sure that the performance/PPL is the same we hade previously
* double-check code on AWD\_modified, and run an experiment that initializes the nn.Embeddings as the FastText embeddings with d300. The performance should be comparable or slightly worse than the self-trained d400.
* Examine the loss function. Implement and evaluate 2 alternative ways of mixing the 2 models AWD and AWD\_modified:  
  a) learned weighted average of the result logits of the 2 models  
  b) modify the compute-loss function and use the average of the (log)Probabilities from the (log)Softmax  
  1 layer (Linear or LSTM) from the 2 concatenated encodings of the last layer of the models can be the source for the scalar value of the average-gate.
* Modify the creation of the vocabulary, to add a column ‘lemmatized\_form’, so the lemmatization step to get the global node-state will be faster.
* Modify the creation of the Graph. Add a dummy sense label, and a corresponding edge type maybe (ds, dummySense). This will be used by the GRU\_senses.
* Compute 2 perplexities on the GRU\_senses: one for all the words in the document, and another only for the words that have multiple senses

### AWD-LSTM, base

| epoch 1 | 800/ 932 batches | lr 30.00000 | ms/batch 81.28 | loss 6.52 | ppl 681.67

| end of epoch 1 | time: 81.79s | valid loss 5.99 | valid ppl 398.30 | valid bpc 8.638

| end of epoch 2 | time: 83.27s | valid loss 5.62 | valid ppl 274.66 | valid bpc 8.101

The PPL in the original modified\_port baseline run was:

| epoch 1 | 800/ 932 batches | lr 30.00000 | ms/batch 86.48 | loss 6.52 | ppl 681.67

| end of epoch 1 | time: 86.54s | valid loss 5.99 | valid ppl 398.30 | valid bpc 8.638

| end of epoch 2 | time: 88.61s | valid loss 5.62 | valid ppl 274.66 | valid bpc 8.101

We get exactly the same results (as the random seed is fixed), the baseline is unchanged by the modifications we made to the code. We can proceed.

### AWD-LSTM, FastText embeddings

We compare it to the baseline, as follows:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Epoch | 300d w/ FastText embeddings | | 400d baseline | |
|  | Train PPL @ 800 | Valid PPL | Train PPL @ 800 | Valid PPL |
| 1 | 632.58 | 357.27 | 681.67 | 398.30 |
| 10 | 143.68 | 119.30 | 143.38 | 123.30 |
| 20 | 108.26 | 100.67 | 101.21 | 101.22 |
| 32 | 93.34 | 96.54 | 86.35 | 99.17 |
| 50 | 84.40 | 84.57 | 75.39 | 82.81 |
| 100 | 74.15 | 81.16 | 64.86 | 79.20 |
| 200 | 68.73 | 78.72 | 55.94 | 75.60 |
| 500 | 63.49 | 77.20 | 53.39 | 74.92 |
| 750 | 62.41 | 77.03 | 52.45 | 74.66 |
| Test set | 78.03 | | 75.51 | |

We can observe that:

at the start of training, the pre-trained FastText embeddings already hold some knowledge and reach a better performance faster.

Later on, the adjustments on embeddings and encoding of d=400 overtake the d=300.

The next question is:

if we learn to apply a weighted average of the 2 models, can we do better?

## Combining 2 models

### Weighted average of the result logits

We create a new version of the model, AWD\_ensemble.

Currently, it is initialized by taking as parameters an AWD\_base and an AWD\_modified model.

Which parameters do we need to copy or modify?

criterion = SplitCrossEntropyLoss(**model.ninp**, splits=splits, verbose=**False**)

SplitCrossEntropyLoss is initialized with:

\_\_init\_\_(self, hidden\_size, splits, verbose=**False**)

I may cause a mistake on the hidden\_size

The final encoding of the AWD\_base has d=400, in the AWD\_modified with FastText embeddings it is d=300.

It is not possible to sum them up element by element. The sum must be over the logits or if possible the (log)probabilities.

Logging, we print:

Using splits []

Which means that for WikiText-2 we do not have splits – the reasoning is that with a relatively low number of tokens and small vocabulary there is no necessity to split the softmax.

model.ninp is not required elsewhere.

I decide to put the value in the AWD\_base as a placeholder.

More relevant: both in the train() and the evaluate() functions, we find:

hidden = model.init\_hidden(batch\_size)

…

raw\_loss = criterion(model.decoder.weight, model.decoder.bias, output, targets)

The core steps of the criterion, which is the SplitCrossEntropyLoss, can be found in:

**def** forward(self, weight, bias, hiddens, targets, verbose=**False**):

…

all\_head\_res = torch.nn.functional.linear(combo, head\_weight,   
 bias=head\_bias)  
 softmaxed\_all\_head\_res =   
 torch.nn.functional.log\_softmax(all\_head\_res, dim=-1)

the criterion is invoked as:

criterion(model.decoder.weight, model.decoder.bias, output, targets)

**Idea #1**:

modify heavily the loss function.

Add an ensemble flag, or make an alternative version of the function.

Take in 2 sets of: weight, bias, hidden (aka a model’s output logits), and also the weights of a matrix that goes from the last layer’s encoding into a contribution to the weighted average coefficient.

Then execute:

(a \* logProbs\_base) + (b\* logProbs\_modified)

The alternative version (**Idea #2**) would be to have a weighted sum of the logits across the vocabulary and then apply the log-softmax.

In the meantime, I have to adjust the model.init\_hidden(). The hidden state of the ensemble will be a tuple…

After the nn.Linear, the logits: all\_head\_res.shape= torch.Size([2304, 33278])

After the logsoftmax, softmaxed\_all\_head\_res.shape = torch.Size([2304, 33278])

In **Version 1**, we have 1 LSTM layer that goes from the last layers of the AWD-LSTM (d400 ++ d300) into a scalar between 0 and 1.

This is **a**, in: a \* (logProbs\_base) + ((1-a)\* logProbs\_modified).

The output of the last layers is retrieved as?

Considering that

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

we may also use the output instead of the result. The result is viewed as (2304 x 400) instead of (70x32x400)

using the ensemble model, the result of the forward() call was:

result, hidden, raw\_outputs, outputs

assigned as:

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

What we pass on to the criterion.forward\_ensemble is the output, a.k.a. the in-model result, 2304x400.

Operating with:

a\_out, a\_hidden = ensemble\_model.C(last\_layers\_concat)

a\_out.shape=(72,32,1).

here, min=0.0152 and max=0.5717 , but the tanh range is (-1,1). We may have to squash it into (0,1) either by logistic function or by simple scaling.

a\_hidden is made of h\_ and c\_, as usual.

Then, we must replicate the entropy and loss.

softmaxed\_all\_head\_res.shape= torch.Size([2304, 33278])

When the idx of the split is 0:

softmaxed\_head\_res.shape= torch.Size([2304, 33278]) (identical, since we have no splits here).

This is followed by:

entropy = -torch.gather(softmaxed\_head\_res, dim=1, index=split\_targets[idx].view(-1, 1))

of shape= torch.Size([2304, 33278])

where

targets.shape=2304 and

split\_targets[idx].view(-1, 1).shape= torch.Size([2304, 1])

since it is equivalent to: targets.view(-1, 1)

## Ensemble 1.0 , experiments

Now that we have in place the version 1.0 of the Ensemble model, that uses the architecture:

concatenated last layers d400++d300 > 1-layer LSTM > coefficient **a** >   
a\*logsoftmax(base AWD-LSTM) + (1-a)\*logsoftmax(modified AWD-LSTM)

it is opportune to debug it / check whether it works as intended.

### a=1

* **If we force a=1, do we manage to recover the exact results of the baseline AWD-LSTM?**

We get:

| epoch 1 | 600/ 932 batches | lr 30.00000 | ms/batch 158.37 | loss 6.76 | ppl 863.82 |

| epoch 1 | 800/ 932 batches | lr 30.00000 | ms/batch 160.30 | loss 6.53 | ppl **687.72** |

| end of epoch 1 | time: 161.72s | valid loss 6.11 | valid ppl **449.28** | valid bpc 8.811  
…

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

| end of epoch 750 | time: 164.01s | valid loss 4.31 | valid ppl **74.12** | valid bpc 6.212

Saving Averaged!

| End of training | test loss 4.34 | test ppl **76.99** | test bpc 6.267

The port baseline was:

| epoch 1 | 600/ 932 batches | lr 30.00000 | ms/batch 85.07 | loss 6.74 | ppl 844.84

| epoch 1 | 800/ 932 batches | lr 30.00000 | ms/batch 81.28 | loss 6.52 | ppl **681.67**

| end of epoch 1 | time: 81.79s | valid loss 5.99 | valid ppl **398.30** | valid bpc 8.638

…

| end of epoch 750 | time: 89.32s | valid loss 4.31 | valid ppl **74.66** | valid bpc 6.222

Saving Averaged!

| End of training | test loss 4.32 | test ppl **75.51** | test bpc 6.239

Validation PPL: 74.12 vs 74.66, slightly better.

Test PPL: 76.99 vs 75.51, slightly worse.

Why the difference? we set a=1s for: a\*logsoftmax(base AWD) + (1-a)\*logsoftmax(modified AWD)

Maybe it’s due to the activation regularization, that depends on the output of the layers of both models.

If we use module&time activation regularization only for the original model, can we have the performance of the original base AWD-LSTM again?

No, it doesn’t seem to be the case.

Let us compare the output, at iteration 0, of the unchanged AWD-LSTM with the ensemble model, before the first application of loss.backwards() and the gradient.

output\_base (of size (2304,400)):

output\_base[0:2,0:5] =

tensor([[ 0.0000, -0.0020, 0.0080, 0.0013, 0.0064],

[ 0.0000, -0.0022, 0.0068, 0.0000, 0.0076]],

From the ensemble, output[0][0:2,0:5]=

tensor([[ 0.0080, -0.0000, 0.0000, 0.0000, 0.0078],

[ 0.0000, -0.0032, 0.0075, 0.0005, 0.0071]], device='cuda:0',

grad\_fn=<SliceBackward>)

**Idea #2**:

instead of modifying the whole code of the main.py and the training loop, I can execute the 2 models sequentially (or in parallel, if we have > 1 GPU) by refactoring part of the training iteration, until we obtain output, hidden, rnn\_hs, dropped\_rnn\_hs.

This way, we do not create any ensemble model object. We expect the output of AWD\_base and AWD\_modified to be the usual one. Then, we separately define a 400+300 -> 1 LSTM that will handle the “a\*logsoftmax(AWD\_base) + (1-a)\*logsoftmax(AWD\_modified)” .

The ensemble\_criterion already written will be unchanged.