[Preliminary information (Datasets, graphs) 4](#_Toc40189079)

[WikiText-2 4](#_Toc40189080)

[WikiText-2 Stats 4](#_Toc40189081)

[Graph 4](#_Toc40189082)

[SemCor 5](#_Toc40189083)

[SemCor stats 5](#_Toc40189084)

[Graph 5](#_Toc40189085)

[Globals - RNN 6](#_Toc40189086)

[Architecture 6](#_Toc40189087)

[Experiments 6](#_Toc40189088)

[Mini-experiment – overfit on fragment 6](#_Toc40189089)

[Experiment – RNN on WT2 8](#_Toc40189090)

[Globals – GRU 9](#_Toc40189091)

[Architecture 9](#_Toc40189092)

[Experiments 9](#_Toc40189093)

[Mini-experiment – overfit GRU on fragment of WT2 9](#_Toc40189094)

[Experiment – GRU on WT2 11](#_Toc40189095)

[Including the graph input 13](#_Toc40189096)

[Review: Graph Attention Networks 13](#_Toc40189097)

[Review: RGAT: Relational Graph-Attention Networks 14](#_Toc40189098)

[Globals & Graph Input – Graph Attention Networks 15](#_Toc40189099)

[GAT, input signals, 4-heads 15](#_Toc40189100)

[Architecture 15](#_Toc40189101)

[Mini-Experiment, GRU-GAT4 on fragment of WT2 16](#_Toc40189102)

[Experiment – GRU-GAT4 on WT2 16](#_Toc40189103)

[Experiment – GAT with 1 head 18](#_Toc40189104)

[The Sense task 20](#_Toc40189105)

[Reflections 20](#_Toc40189106)

[Baseline: Simple GRU on SemCor 20](#_Toc40189107)

[Mini-experiment: overfitting on a fragment of SemCor 20](#_Toc40189108)

[Experiment – GRU on SemCor 22](#_Toc40189109)

[Baseline 2: GRU + GAT4 (current global’s node state) on SemCor 23](#_Toc40189110)

[Mini-Experiment – Overfit on a fragment of SemCor 23](#_Toc40189111)

[Experiment – GRU+GAT4 on SemCor 23](#_Toc40189112)

[Senses: Photo-concat 25](#_Toc40189113)

[Architecture 25](#_Toc40189114)

[Experiment – Globals and senses (GRU + 2xGAT4) on SemCor 26](#_Toc40189115)

[Senses: Shared GRU layer 27](#_Toc40189116)

[Architecture 27](#_Toc40189117)

[Experiment 1 – Senses: Shared GRU Layer 28](#_Toc40189118)

[Senses: Self-attention scores on the senses of the k most likely globals 29](#_Toc40189119)

[Design 29](#_Toc40189120)

[Reviewing self-attention 30](#_Toc40189121)

[Implementation 30](#_Toc40189122)

[Experiments 32](#_Toc40189123)

[Meetings 35](#_Toc40189124)

[IA 35](#_Toc40189125)

[IA, PSK, KF 35](#_Toc40189126)

[Senses Baseline: Operate on the k most likely globals 36](#_Toc40189127)

[Hypothesis 1: FF-NN – from all logits to all logits 37](#_Toc40189128)

[Hypothesis 2: Project embeddings of the first k globals + concat logit + FF-NN 37](#_Toc40189129)

[Model 38](#_Toc40189130)

[Experiment on SemCor 38](#_Toc40189131)

[Senses: self-attention on the k most likely globals 39](#_Toc40189132)

[Design 39](#_Toc40189133)

[Model 39](#_Toc40189134)

[Experiments 40](#_Toc40189135)

[Overfit on mini-dataset 40](#_Toc40189136)

[Experiment 1 – SelfAttention4Senses on SemCor 41](#_Toc40189137)

[Experiment 2 – SelfAttention4Senses on SemCor 42](#_Toc40189138)

[Globals – LSTM 43](#_Toc40189139)

[Design 43](#_Toc40189140)

[Elements in AWD-LSTM 43](#_Toc40189141)

[Hyperpararameters used in AWD-LSTM 44](#_Toc40189142)

[Their Perplexity Results 44](#_Toc40189143)

[Implementation 45](#_Toc40189144)

[Salesforce’s AWD-LSTM 45](#_Toc40189145)

[My own WD-LSTM 45](#_Toc40189146)

[Experiments 46](#_Toc40189147)

[Mini-experiment 1 – overfit LSTM1 on fragment of WT2 46](#_Toc40189148)

[Mini-experiment 2 – overfit LSTM2 on fragment of WT2 47](#_Toc40189149)

[Mini-experiment 3 – overfit LSTM3 on fragment of WT2 48](#_Toc40189150)

[Experiment 1 – LSTM3 on WT2 49](#_Toc40189151)

[Experiment 2 – WD-LSTM3 on WT2 50](#_Toc40189152)

[Experiment 3 – WD-LSTM3 on WT2 51](#_Toc40189153)

[Globals – GRU 52](#_Toc40189154)

[Design 52](#_Toc40189155)

[Experiments 53](#_Toc40189156)

[Mini-experiment 1 – Overfit on fragment of WT-2 53](#_Toc40189157)

[**Experiment 1 – GRU on WT-2** 53](#_Toc40189158)

[Including the Graph Input 55](#_Toc40189159)

[GRU\_GAT 55](#_Toc40189160)

[Model 55](#_Toc40189161)

[Mini-experiment 1 – Overfit on fragment of WT-2 55](#_Toc40189162)

[**Experiment 1 – GRU\_GAT on WT-2** 56](#_Toc40189163)

[WD-LSTM-GAT 58](#_Toc40189164)

[Model 58](#_Toc40189165)

[Experiment 1 – WD-LSTM-GAT 58](#_Toc40189166)

[Including the whole AWD-LSTM implementation 59](#_Toc40189167)

[Input facilities 59](#_Toc40189168)

[Training loop 59](#_Toc40189169)

[Corpus 60](#_Toc40189170)

[Batch & Input data 61](#_Toc40189171)

[Embeddings used by the model 62](#_Toc40189172)

[Hypothesis: use X and the indices of the globals 63](#_Toc40189173)

[Adjusting awd-lstm for the current PyTorch setup (1.4.0) 64](#_Toc40189174)

[WeightDrop 64](#_Toc40189175)

[Optimizer error 65](#_Toc40189176)

[Weights can be flattened in memory 65](#_Toc40189177)

[Torch 0.4.0 and 1.0.0 66](#_Toc40189178)

[Out of Memory 66](#_Toc40189179)

[On the side: batching for GAT 68](#_Toc40189180)

[Unpacking the batchinput\_tensor 69](#_Toc40189181)

[Full batching for GATConv 70](#_Toc40189182)

[Batches and DataParallel 70](#_Toc40189183)

[Meeting with IA, 12/05 71](#_Toc40189184)

# Preliminary information (Datasets, graphs)

## WikiText-2

### WikiText-2 Stats

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

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

Validation: 242,478 (186,258)

Test: 277,846 (209,228)

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

### Graph

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

(From the log at DefineGraph.log)

Constructing X, matrix of node features

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

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

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

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

Defining the edges: def, exs

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

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

Defining the edges: sc

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

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

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

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

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

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

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

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

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

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

## SemCor

### SemCor stats

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

Training tokens: 646,038

Validation tokens: 80,760

Mini-dataset: 180 tokens.

### Graph

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

Constructing X, matrix of node features

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

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

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

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

Defining the edges: def, exs

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

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

Defining the edges: sc

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

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

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

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

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

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

# Globals - RNN

## Architecture

The first baseline is a simple Recurrent Neural Network.

* X= input embeddings.
  + w(t)= the current global word
* x(t) = w(t)++s(t-1).   
  input\_x = torch.cat([currentword\_embedding, self.memory\_context])
* New state: use the matrix W. E.g.: (300+512) > 512. s(t) = W\*x(t)
* Save s(t) in the memory buffer
* Obtain the logits, using the matrix linear2global: 1024 -> |Vocab|
* Apply softmax

## Experiments

### Mini-experiment – overfit on fragment

|  |  |  |
| --- | --- | --- |
| **Architecture** | **Input Signals** | **Hyperparameters** |
| Simple RNN | 1) The word embedding of the current global | batch\_size=4 |
| 2 layers: 600,300 | TBPTT length=16 |
|  | learning rate=0.001 |
|  |  |

{len(train\_dataloader)=9 (tokens>=~9\*4\*16>=576, we know they are 632)}

|  |  |
| --- | --- |
| **Epoch** | **Training perplexity** |
| 1 | 79684.59 |
| 2 | 23370.42 |
| 3 | 11611.48 |
| 4 | 616.67 |
| 5 | 381.21 |
| 10 | 91.15 |
| 30 | 2.86 |
| 50 | 1.24 |
| 100 | 1.12 |

Doubt: why do I have 79K perplexity after the 1st epoch, when my vocabulary of globals is only **31640**?

In the 1st epoch, while 7 out of 9 batches have loss around 9 or 10, the last 2 have loss 26 and 65 respectively. Same for the 1st batch of the 2nd epoch (25.6).

Probably due to extreme values (0 or inf) in the network.

Nevertheless, this artefact disappears during the rest of the training.

**Reviewing the predictions:**

Label: the next global is: **less**(from 1575)

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

INFO : The top- 5 predicted globals are:

INFO : **Word: less ; probability = 99.9%**

INFO : Word: the ; probability = 0.1%

INFO : Word: this ; probability = 0.01%

INFO : Word: during ; probability = 0.0%

INFO : Word: followed ; probability = 0.0%

INFO :

Label: the next global is: **than**(from 167)

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

INFO : The top- 5 predicted globals are:

INFO : **Word: than ; probability = 100.0%**

INFO : Word: Kurfürst ; probability = 0.0%

INFO : Word: a ; probability = 0.0%

INFO : Word: were ; probability = 0.0%

INFO : Word: period ; probability = 0.0%

INFO :

Label: the next global is: **half**(from 904)

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

INFO : The top- 5 predicted globals are:

INFO : **Word: half ; probability = 99.99%**

INFO : Word: a ; probability = 0.0%

INFO : Word: Cruiser ; probability = 0.0%

INFO : Word: draw ; probability = 0.0%

INFO : Word: at ; probability = 0.0%

INFO :

Label: the next global is: **an**(from 158)

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

INFO : The top- 5 predicted globals are:

INFO : **Word: an ; probability = 100.0%**

INFO : Word: sisters ; probability = 0.0%

INFO : Word: began ; probability = 0.0%

INFO : Word: a ; probability = 0.0%

INFO : Word: resulted ; probability = 0.0%

INFO :

Label: the next global is: **hour**(from 2780)

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

INFO : The top- 5 predicted globals are:

INFO : **Word: hour ; probability = 99.97%**

INFO : Word: fired ; probability = 0.03%

INFO : Word: in ; probability = 0.0%

INFO : Word: on ; probability = 0.0%

INFO : Word: Beatty ; probability = 0.0%

INFO :

Label: the next global is: later(from 678)

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

INFO : The top- 5 predicted globals are:

INFO : **Word: later ; probability = 99.93%**

INFO : Word: their ; probability = 0.05%

INFO : Word: fired ; probability = 0.01%

INFO : Word: crew ; probability = 0.0%

INFO : Word: duel ; probability = 0.0%

INFO:

The predictions appear to be correct – a simple RNN manages to overfit on a fragment of WikiText-2.

### Experiment – RNN on WT2

(from the previous series)

|  |  |  |
| --- | --- | --- |
| **Architecture** | **Input Signals** | **Hyperparameters** |
| Simple RNN | 1) The word embedding of the current global | batch\_size=40 |
| 2 layers: 512,256 | TBPTT length=32 |
|  | learning rate=10^(-4) |
|  |  |

Number of trainable parameters=43.042M

INFO : Parameters:

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

('module.memory\_h1', torch.Size([1, 512]), False), ('module.memory\_h2', torch.Size([1, 256]), False),

('module.W1.weight', torch.Size([512, 812]), True), ('module.W1.bias', torch.Size([512]), True),

('module.W2.weight', torch.Size([256, 768]), True), ('module.W2.bias', torch.Size([256]), True), ('module.linear2global.weight', torch.Size([31640, 256]), True), ('module.linear2global.bias', torch.Size([31640]), True)]

34.30M embeddings, 8.1M softmax, 0.62M core

|  |  |  |
| --- | --- | --- |
| *Epoch* | *Training perplexity* | *Validation perplexity* |
| 1 | 1034.0 | 472.72 |
| 2 | 548.1 | 385.2 |
| 3 | 438.11 | 345.09 |
| 4 | 374.13 | 318.25 |
| 5 | 329.48 | 302.44 |
| 6 | 296.01 | 293.35 |
| 7 | 269.14 | 288.09 |
| 8 | 247.38 | 285.5 |
| 9 | 228.91 | 283.75 |
| 10 | 213.34 | 283.78 |
| 11 | 199.86 | 283.23 |
| 12 | 188.19 | 284.34 |
| 13 | 177.92 | 283.85 |
| 14 | 168.97 | 286.75 |
| 15 | 161.08 | 283.29 |
| 16 | 154.02 | 280.45 |
| 17 | 147.79 | 280.46 |
| 18 | 141.92 | 285.7 |

In the “Restricted Recurrent Neural Networks” paper, that we are using as point of comparison, the validation performance of an RNN model on WT-2 is in the range (320,250), with the best value they obtained being 253.1 comparable to our value of 280.45.

# Globals – GRU

## Architecture

Reviewing the formula for the GRU:

* + Update gate:
  + Reset gate:
  + Proposed new-state:
  + Final updated state:

## Experiments

### Mini-experiment – overfit GRU on fragment of WT2

|  |  |  |
| --- | --- | --- |
| **Architecture** | **Input Signals** | **Hyperparameters** |
| GRU RNN  (see the full GRU formula) | 1) The word embedding of the current global | batch\_size=4 |
| 2 layers: 300,300 | TBPTT length=16 |
|  | learning rate=0.001 |
|  |  |

{len(train\_dataloader)=9 (tokens>=~9\*4\*16>=576, we know they are 632)}

|  |  |
| --- | --- |
| **Epoch** | **Training perplexity** |
| 1 | 20311.33 |
| 2 | 514.01 |
| 3 | 134.27 |
| 4 | 109.9 |
| 5 | 101.61 |
| 10 | 96.02 |
| 30 | 88.64 |
| 50 | 56.54 |
| 75 | 11.12 |
| 100 | 2.37 |

**Reviewing the predictions:**

Label: the next global is: **s**(from 1829)

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

INFO : The top- 5 predicted globals are:

INFO : **Word: s ; probability = 42.57%**

INFO : Word: two ; probability = 7.06%

INFO : Word: ordered ; probability = 6.17%

INFO : Word: were ; probability = 4.18%

INFO : Word: and ; probability = 3.71%

INFO :

Label: the next global is: **crew**(from 3618)

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

INFO : The top- 5 predicted globals are:

INFO : **Word: crew ; probability = 34.44%**

INFO : Word: the ; probability = 12.77%

INFO : Word: British ; probability = 3.65%

INFO : Word: time ; probability = 2.89%

INFO : Word: by ; probability = 2.64%

INFO :

Label: the next global is: **spotted**(from 3845)

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

INFO : The top- 5 predicted globals are:

INFO : **Word: spotted ; probability = 55.95%**

INFO : Word: Tiger ; probability = 3.76%

INFO : Word: his ; probability = 3.61%

INFO : Word: torpedo ; probability = 2.35%

INFO : Word: of ; probability = 2.16%

INFO :

Label: the next global is: **both**(from 81)

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

INFO : The top- 5 predicted globals are:

INFO : **Word: both ; probability = 48.73%**

INFO : Word: that ; probability = 5.94%

INFO : Word: from ; probability = 4.72%

INFO : Word: was ; probability = 4.53%

INFO : Word: due ; probability = 3.38%

INFO :

Label: the next global is: **the**(from 31)

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

INFO : The top- 5 predicted globals are:

INFO : **Word: the ; probability = 94.16%**

INFO : Word: by ; probability = 0.78%

INFO : Word: steaming ; probability = 0.41%

INFO : Word: i ; probability = 0.21%

INFO : Word: followed ; probability = 0.2%

INFO :

In the overfitting experiment, the predictions are ok.

### Experiment – GRU on WT2

(from the previous series)

|  |  |  |
| --- | --- | --- |
| **Architecture** | **Input Signals** | **Hyperparameters** |
| GRU RNN  (see the full GRU formula) | 1) The word embedding of the current global | batch\_size=80 |
| 2 layers: 300,300 | TBPTT length=35 |
| Pretrained embeddings from: FastText (d=300) | learning rate=10^(-5) |
|  |  |

[('module.X', torch.Size([114324, 300]), True), ('module.select\_first\_node', torch.Size([1]), False),

('module.memory\_h1', torch.Size([1, 300]), False), ('module.memory\_h2', torch.Size([1, 300]), False), ('module.U\_z\_1.weight', torch.Size([300, 300]), True), ('module.W\_z\_1.weight', torch.Size([300, 300]), True), ('module.U\_r\_1.weight', torch.Size([300, 300]), True), ('module.W\_r\_1.weight', torch.Size([300, 300]), True), ('module.U\_1.weight', torch.Size([300, 300]), True), ('module.W\_1.weight', torch.Size([300, 300]), True), ('module.U\_z\_2.weight', torch.Size([300, 300]), True), ('module.W\_z\_2.weight', torch.Size([300, 300]), True), ('module.U\_r\_2.weight', torch.Size([300, 300]), True), ('module.W\_r\_2.weight', torch.Size([300, 300]), True), ('module.U\_2.weight', torch.Size([300, 300]), True), ('module.W\_2.weight', torch.Size([300, 300]), True), ('module.linear2global.weight', torch.Size([31640, 300]), True), ('module.linear2global.bias', torch.Size([31640]), True)]

Number of trainable parameters=44.90M

34.30M embeddings, 9.52M softmax, 1.08M core

|  |  |  |
| --- | --- | --- |
| *Epoch* | *Training perplexity* | *Validation perplexity* |
| 1 | 1226.69 | 681.65 |
| 2 | 694.85 | 444.03 |
| 3 | 522.39 | 377.5 |
| 4 | 450.1 | 341.81 |
| 5 | 406.0 | 319.07 |
| 6 | 374.83 | 302.5 |
| 7 | 350.41 | 289.09 |
| 8 | 330.49 | 278.24 |
| 9 | 313.65 | 269.1 |
| 10 | 298.89 | 261.25 |
| 11 | 286.02 | 254.58 |
| 12 | 274.56 | 248.86 |
| 13 | 264.47 | 243.84 |
| 14 | 255.26 | 239.41 |
| 15 | 246.88 | 235.47 |
| 16 | 239.29 | 231.94 |
| 17 | 232.18 | 228.78 |
| 18 | 225.76 | 226.07 |
| 19 | 219.9 | 223.64 |
| 20 | 214.4 | 221.48 |
| 21 | 209.27 | 219.62 |
| 22 | 204.49 | 217.89 |
| 23 | 200.09 | 216.31 |
| 24 | 195.9 | 214.96 |
| 25 | 191.87 | 213.62 |
| 26 | 188.1 | 212.35 |
| 27 | 184.61 | 211.15 |
| 28 | 181.25 | 210.06 |
| 29 | 178.03 | 208.99 |
| 30 | 174.86 | 208.09 |
| 31 | 171.93 | 207.27 |
| 32 | 169.15 | 206.45 |
| 33 | 166.5 | 205.68 |
| 34 | 163.81 | 205.07 |
| 35 | 161.3 | 204.35 |
| 36 | 158.91 | 203.68 |
| 37 | 156.63 | 203.03 |
| 38 | 154.5 | 202.49 |
| 39 | 152.32 | 201.94 |
| 40 | 150.23 | 201.48 |
| 41 | 148.17 | 201.03 |
| 42 | 146.31 | 200.55 |
| 43 | 144.47 | 200.21 |
| 44 | 142.68 | 200.0 |
| 45 | 140.97 | 199.49 |
| 46 | 139.21 | 199.32 |

Operating with a (complete) Gated Recurrent Unit, we obtain a Validation Perplexity of 199.3 on WikiText-2.

# Including the graph input

The graph:



(note: the figure also higlights the problem of dealing with inflected forms. We ignore that for now. We could create a new edge type.)

The idea is to Add **in parallel** the input from the graph, as well as the one from the senses:  
current global’s word embedding || global’s node-state || sense’s node state.

What could we use to obtain node states from the graph?

## Review: Graph Attention Networks

The steps of Graph Attention Networks, originally proposed by Velickovic et al. (2017), are the following:

* Shared linear transformation on every node,   
  This can be modified, using a different weight matrix for each node type: definitions, examples, senses, synonyms, antonyms (and globals?)
* Non-normalized attention coefficients for neighbours:   
  The attention mechanism uses a 1-layer FF-NN:
* Get the normalized coefficients using softmax over the neighbourhood:
* Finally, compute the new state of the node:

Multi-head attention (either through averaging or through concatenating) was found beneficial to extend and stabilize the method.

## Review: RGAT: Relational Graph-Attention Networks

* Starting point: a graph with |R| relation types and N nodes.
* We operate with |R|=5 intermediate representations for the nodes, 1 for each relation.  
  In practice, they are summed up to obtain the node rep at the next layer, .
* Attention mechanism, part 1:  
  We must compute the logit scores , from node *i* to *j* under relation *r.* Self-attention is used.
  + We obtain queries and keys of dimension d=1 as follows:  
     and
  + Then, we create either Additive logits: or Multiplicative
* Attention mechanism, part 2:  
  Either WithIn-Relation G.AT. (independent probablity distribution for each relation r)  
  or Across-Relation G.AT. = 1 probability distribution. The sum to 1 across all the neighbouring nodes and relations
* Propagation rule:

Found to be comparable with RGCNs, but not significantly better.

For inductive tasks, ARGAT with multiplicative logits fared slightly better

# Globals & Graph Input – Graph Attention Networks

Note: Originally, I tried replacing the global’s word embedding with the *node state* of the global node from the graph, obtained with a GNN.

It seems that *replacing* the word embedding with the graph signal from a 2-hops neighbourhood introduces noise. Validation perplexity obtained on WikiText-2: 225.8 > 199.

No improvement on GRU\_GNN.

2 modifications must be applied:

* Using (concatenated) **multi-head attention** in the GAT
* Changing altogether the inclusion of the graph-influenced input: instead of **replacing** the pretrained word embedding, it should be **added in parallel**.

## GAT, input signals, 4-heads

### Architecture

Input signals:

1. the current global word
2. the node-state of the word in the KB-graph, obtained applying the GNN
3. the node-state of the current sense (if present)

We concatenate the input signals and send them as input to the 1st GRU layer*.*

Operating with no senses, total dimensions of the concatenated input: 600.

The hidden state at GRU\_h1 will also have increased dimensionality, 600 (from 300).

Parameters:

[('module.X', torch.Size([114324, 300]), True), ('module.select\_first\_node', torch.Size([1]), False), ('module.nodestate\_zeros', torch.Size([1, 300]), False),

('module.memory\_h1', torch.Size([1, 600]), False), ('module.memory\_h2', torch.Size([1, 300]), False), ('module.gat.weight', torch.Size([300, 300]), True), ('module.gat.att', torch.Size([1, 4, 150]), True), ('module.gat.bias', torch.Size([300]), True),

('module.U\_z\_1.weight', torch.Size([600, 600]), True), ('module.W\_z\_1.weight', torch.Size([600, 600]), True), ('module.U\_r\_1.weight', torch.Size([600, 600]), True), ('module.W\_r\_1.weight', torch.Size([600, 600]), True), ('module.U\_1.weight', torch.Size([600, 600]), True), ('module.U\_1.bias', torch.Size([600]), True), ('module.W\_1.weight', torch.Size([600, 600]), True), ('module.W\_1.bias', torch.Size([600]), True), ('module.U\_z\_2.weight', torch.Size([300, 300]), True), ('module.W\_z\_2.weight', torch.Size([300, 600]), True), ('module.U\_r\_2.weight', torch.Size([300, 300]), True), ('module.W\_r\_2.weight', torch.Size([300, 600]), True), ('module.U\_2.weight', torch.Size([300, 300]), True), ('module.U\_2.bias', torch.Size([300]), True), ('module.W\_2.weight', torch.Size([300, 600]), True), ('module.W\_2.bias', torch.Size([300]), True), ('module.linear2global.weight', torch.Size([31640, 300]), True), ('module.linear2global.bias', torch.Size([31640]), True)]

Trainable parameters=46.883M ; 34.30M embeddings, 9.52M softmax, 3.06M core

### Mini-Experiment, GRU-GAT4 on fragment of WT2

|  |  |  |
| --- | --- | --- |
| **Architecture** | **Input Signals** | **Hyperparameters** |
| GRU + GAT with 4 attention heads | 1) Current global’s word embedding  2) current global’s node-state | batch\_size=4 |
| The GRU has 2 layers: 300, 300 | TBPTT length=8 |
| Pretrained embeddings from: FastText | learning rate=10^(-3) |
|  | graph\_area=32 |

|  |  |
| --- | --- |
| **Epoch** | **Training perplexity** |
| 1 | 16303.75 |
| 2 | 437.81 |
| 3 | 130.15 |
| 4 | 109.63 |
| 5 | 101.48 |
| 10 | 95.21 |
| 30 | 61.0 |
| 50 | 7.62 |
| 75 | 1.45 |
| 100 | 1.11 |

### Experiment – GRU-GAT4 on WT2

|  |  |  |
| --- | --- | --- |
| **Architecture** | **Input Signals** | **Hyperparameters** |
| GRU + GAT with 4 attention heads | 1) Current global’s word embedding  2) current global’s node-state | batch\_size=40 |
| The GRU has 2 layers: 600, 300 | TBPTT length=35 |
| Pretrained embeddings from: FastText | learning rate=0.5\* 10^(-4) |
|  | graph\_area=32 |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| *Epoch* | *Run 1* | | *Run 2* | |
|  | *Train PPL* | *Valid PPL* | *Train PPL* | *Valid PPL* |
| 1 | 1158.57 | 666.68 |  |  |
| 2 | 675.32 | 438.89 |  |  |
| 3 | 514.35 | 376.3 |  |  |
| 4 | 450.62 | 344.82 |  |  |
| 5 | 411.91 | 323.98 |  |  |
| 6 | 384.14 | 308.58 |  |  |
| 7 | 362.51 | 296.37 |  |  |
| 8 | 344.84 | 286.66 |  |  |
| 9 | 329.53 | 278.12 |  |  |
| 10 | 316.27 | 270.79 |  |  |
| 11 | 304.32 | 264.27 |  |  |
| 12 | 293.48 | 258.2 |  |  |
| 13 | 283.52 | 252.67 |  |  |
| 14 | 274.34 | 247.66 |  |  |
| 15 | 265.88 | 243.3 |  |  |
| 16 | 258.1 | 239.37 |  |  |
| 17 | 250.94 | 235.71 |  |  |
| 18 | 244.27 | 232.26 |  |  |
| 19 | 238.02 | 229.14 |  |  |
| 20 | 232.13 | 226.21 |  |  |
| 21 | 226.74 | 223.62 |  |  |
| 22 | 221.61 | 221.24 |  |  |
| 23 | 216.83 | 218.96 |  |  |
| 24 | 212.31 | 216.91 |  |  |
| 25 | 208.05 | 215.11 |  |  |
| 26 | 204.0 | 213.33 |  |  |
| 27 | 200.18 | 211.71 |  |  |
| 28 | 196.53 | 210.42 |  |  |
| 29 | 193.09 | 209.1 |  |  |
| 30 | 189.71 | 207.78 |  |  |
| 31 | 186.61 | 206.7 |  |  |
| 32 | 183.59 | 205.57 |  |  |
| 33 | 180.7 | 204.68 |  |  |
| 34 | 177.88 | 203.91 |  |  |
| 35 | 175.27 | 203.04 |  |  |
| 36 | 172.72 | 202.27 |  |  |
| 37 | 170.28 | 201.66 |  |  |
| 38 | 167.89 | 200.95 |  |  |
| 39 | 165.57 | 200.27 |  |  |
| 40 | 163.42 | 199.83 |  |  |
| 41 | 161.19 | 199.24 |  |  |
| 41 | 159.17 | 198.72 |  |  |
| 43 | 157.1 | 198.31 |  |  |
| 44 | 155.16 | 197.9 |  |  |
| 45 | 153.26 | 197.44 |  |  |
| 46 | 151.4 | 197.06 |  |  |
| 47 | 149.6 | 196.7 |  |  |
| 48 | 147.92 | 196.36 |  |  |
| 49 | 146.17 | 195.99 |  |  |
| 50 | 144.51 | 195.46 |  |  |
| 51 | 142.86 | 194.95 |  |  |
| 52 | 141.34 | 194.48 |  |  |
| 53 | 139.82 | 194.06 |  |  |
| 54 | 138.34 | 193.6 |  |  |
| 55 | 136.91 | 193.16 |  |  |
| 56 | 135.46 | 192.92 |  |  |
| 57 | 134.1 | 192.5 |  |  |
| 58 | 132.79 | 192.33 |  |  |
| 59 | 131.49 | 192.01 |  |  |
| 60 | 130.16 | 191.86 |  |  |

191.8 Validation Perplexity on WikiText-2, better than the 199 obtained with a GRU that used only the Word Embedding as an Input Signal.

### Experiment – GAT with 1 head

|  |  |  |
| --- | --- | --- |
| **Architecture** | **Input Signals** | **Hyperparameters** |
| GRU + GAT | 1) Current global’s word embedding  2) current global’s node-state | batch\_size=40 |
| GRU: 2 layers: 600, 300 | TBPTT length=35 |
| GAT: 1 attention head | learning rate=0.5\* 10^(-4) |
|  | graph\_area=32 |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| *Epoch* | *Run 1* | | *Run 2* | |
|  | Train PPL | Valid PPL | Train PPL | Valid PPL |
| 1 | 1121.7 | 585.29 |  |  |
| 2 | 645.82 | 442.49 |  |  |
| 3 | 531.05 | 395.01 |  |  |
| 4 | 470.75 | 359.85 |  |  |
| 5 | 425.67 (411.91) | 333.01 (323.98) |  |  |
| 6 | 391.9 | 313.56 |  |  |
| 7 | 365.68 | 299.06 |  |  |
| 8 | 344.83 | 288.12 |  |  |
| 9 | 327.42 | 278.83 |  |  |
| 10 | 312.78 (316.27) | 271.01 (270.79) |  |  |
| 11 | 299.8 | 264.19 |  |  |
| 12 | 288.27 | 258.05 |  |  |
| 13 | 277.72 | 252.5 |  |  |
| 14 | 268.3 | 247.34 |  |  |
| 15 | 259.63 | 243.05 |  |  |
| 16 | 251.68 | 238.9 |  |  |
| 17 | 244.33 | 235.22 |  |  |
| 18 | 237.52 | 231.76 |  |  |
| 19 | 231.29 | 228.76 |  |  |
| 20 | 225.35 (232.13) | 225.83 (226.21) |  |  |
| 21 | 219.98 | 223.45 |  |  |
| 22 | 214.9 | 221.34 |  |  |
| 23 | 209.97 | 219.27 |  |  |
| 24 | 205.46 | 217.36 |  |  |
| 25 | 201.1 | 215.71 |  |  |
| 26 | 196.97 | 214.14 |  |  |
| 27 | 193.09 | 212.92 |  |  |
| 28 | 189.32 | 211.9 |  |  |
| 29 | 185.78 | 210.78 |  |  |
| 30 | 182.33 (189.71) | 209.9 (207.78) |  |  |
| 31 | 179.06 | 208.99 |  |  |
| 32 | 175.96 | 208.21 |  |  |
| 33 | 172.89 | 207.44 |  |  |
| 34 | 169.98 | 206.91 |  |  |
| 35 | 167.2 | 206.42 |  |  |
| 36 | 164.49 | 205.98 |  |  |
| 37 | 161.94(170.28) | 205.39(201.66) |  |  |

Conclusion: operating with only 1 head is not as effective as having multiple attention heads in the GAT.  
From the descent, we can reasonably expect that (GAT-1head > simple GRU). However, (GAT-4heads > GAT-1head).

# The Sense task

## Reflections

The starting point is that the original idea was suboptimal:

It was:   
(1st part, GRU, GNNs etc.) > representation > in parallel, 1 FF-NN to the globals’ logits and   
 || 1 FF-NN to the senses’ logits

Preliminary experiments on SemCor indicated that when the perplexity on globals was decreasing, the PPL on senses ended up increasing.

The reason was that I was trying to adjust 1 encoding (the representation built with GNNs & co.) to do 2 tasks: predictions on globals and predictions on senses.

## Baseline: Simple GRU on SemCor

**Parameters:**

[('module.X', torch.Size([99963, 300]), True), ('module.select\_first\_node', torch.Size([1]), False), ('module.embedding\_zeros', torch.Size([1, 300]), False), ('module.memory\_h1', torch.Size([1, 300]), False), ('module.memory\_h2', torch.Size([1, 300]), False), ('module.U\_z\_1.weight', torch.Size([300, 300]), True), ('module.W\_z\_1.weight', torch.Size([300, 300]), True), ('module.U\_r\_1.weight', torch.Size([300, 300]), True), ('module.W\_r\_1.weight', torch.Size([300, 300]), True), ('module.U\_1.weight', torch.Size([300, 300]), True), ('module.U\_1.bias', torch.Size([300]), True), ('module.W\_1.weight', torch.Size([300, 300]), True), ('module.W\_1.bias', torch.Size([300]), True), ('module.U\_z\_2.weight', torch.Size([300, 300]), True), ('module.W\_z\_2.weight', torch.Size([300, 300]), True), ('module.U\_r\_2.weight', torch.Size([300, 300]), True), ('module.W\_r\_2.weight', torch.Size([300, 300]), True), ('module.U\_2.weight', torch.Size([300, 300]), True), ('module.U\_2.bias', torch.Size([300]), True), ('module.W\_2.weight', torch.Size([300, 300]), True), ('module.W\_2.bias', torch.Size([300]), True), ('module.linear2global.weight', torch.Size([21988, 300]), True), ('module.linear2global.bias', torch.Size([21988]), True)]

Number of trainable parameters=37.69M, where 6.62M softmax, 29.99M embeddings, ~1.08M core

### Mini-experiment: overfitting on a fragment of SemCor

(Mini-Dataset – training: 180 tokens)

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Training hyperparameters** | | **Input signals & GNN** | | **Architecture** | |
| batch\_size | 4 | grapharea | I: 32n | II: 181e | GRU layers | 2 |
| learning rate | 0.001 | Input signals | 1) current word embedding | hidden dim.s | 300,300 |
| TBPTT length | 16 |  |  |  |

|  |  |
| --- | --- |
| **Epoch** | **Training perplexity** |
| 1 | 20725.76 |
| 5 | 96.59 |
| 10 | 46.49 |
| 20 | 45.48 |
| 30 | 45.26 |
| 50 | 39.78 |
| 100 | 24.66 |
| 150 | 15.71 |
| 200 | 3.16 |

**Reviewing the predictions:**

Label: the next global is: **recent**(from 8)

INFO : Label: the next sense is: late.s.03(from 13363)

INFO : The top- 5 predicted globals are:

INFO : **Word: recent ; probability = 16.48%**

INFO : Word: produced ; probability = 13.0%

INFO : Word: . ; probability = 10.6%

INFO : Word: relative ; probability = 6.38%

INFO : Word: s ; probability = 5.68%

INFO :

Label: the next global is: **primary election**(from 9)

INFO : Label: the next sense is: primary.n.01(from 17809)

INFO : The top- 5 predicted globals are:

INFO : **Word: primary election ; probability = 20.99%**

INFO : Word: was ; probability = 14.71%

INFO : Word: " ; probability = 10.42%

INFO : Word: reports ; probability = 4.05%

INFO : Word: of ; probability = 3.89%

INFO :

Label: the next global is: **produced**(from 10)

INFO : Label: the next sense is: produce.v.04(from 17913)

INFO : The top- 5 predicted globals are:

INFO : **Word: produced ; probability = 17.9%**

INFO : Word: recent ; probability = 10.71%

INFO : Word: . ; probability = 9.93%

INFO : Word: relative ; probability = 6.46%

INFO : Word: s ; probability = 5.18%

INFO :

Overfitting the GRU on fragment of SemCor: confirmed.

### Experiment – GRU on SemCor

(Run 1 is from the previous series)

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Training hyperparameters** | | **Input signals & GNN** | | **Architecture** | |
| batch\_size | 40 | grapharea | I: 32n | II: 181e | GRU layers | 2 |
| learning rate | 1\* 10^(-4) | Input signals | 1) current word embedding | hidden dim.s | 300,300 |
| TBPTT length | 35 |  | Dropout | 0.1 on GRU W1, W2 |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| *Epoch* | *Run 1* | | *Run 2* | |
|  | Train PPL | Valid PPL | Train PPL | Valid PPL |
| 1 | 1160.8 | 512.56 | 1152.3 | 500.47 |
| 2 | 592.36 | 371.88 | 576.01 | 362.94 |
| 3 | 447.17 | 318.74 | 437.35 | 311.46 |
| 4 | 389.24 | 293.26 | 378.46 | 288.02 |
| 5 | 352.05 | 277.62 | 344.2 | 274.89 |
| 6 | 325.85 | 266.19 | 321.18 | 264.72 |
| 7 | 306.34 | 256.95 | 303.29 | 255.9 |
| 8 | 290.63 | 249.21 | 287.73 | 247.86 |
| 9 | 276.94 | 242.02 | 274.08 | 240.81 |
| 10 | 264.88 | 235.99 | 262.45 | 235.38 |
| 11 | 254.07 | 230.65 | 252.32 | 230.51 |
| 12 | 244.18 | 226.36 | 243.29 | 226.65 |
| 13 | 235.07 | 222.76 | 234.99 | 223.46 |
| 14 | 226.92 | 219.69 | 227.56 | 220.68 |
| 15 | 219.65 | 217.09 | 220.81 | 218.17 |
| 16 | 212.96 | 214.94 | 214.5 | 216.15 |
| 17 | 207.14 | 212.94 | 208.8 | 214.22 |
| 18 | 201.55 | 211.29 | 203.23 | 212.76 |
| 19 | 196.51 | 209.76 | 198.2 | 211.32 |
| 20 | 192.0 | 208.54 | 193.51 | 210.15 |
| 21 | 187.67 | 207.39 | 189.17 | 208.9 |
| 22 | 183.68 | 206.49 | 184.99 | 208.06 |
| 23 | 179.96 | 205.57 | 181.3 | 207.21 |
| 24 | 176.36 | 204.91 | 177.64 | 206.52 |
| 25 | 172.92 | 204.41 | 174.17 | 206.15 |
| 26 | 169.72 | 203.82 | 171.0 | 205.55 |
| 27 | 166.71 | 203.58 | 167.96 | 205.45 |
| 28 | 163.69 | 203.23 | 164.88 | 205.22 |
| 29 | 160.78 | 203.03 | 162.02 | 204.95 |
| 30 | 157.97 | 202.63 | 159.26 | 204.86 |
| 31 | 155.36 | 202.61 |  |  |
| 32 | 152.77 | 202.4 |  |  |
| 33 | 150.34 | 202.49 |  |  |
| 34 | 147.87 | 202.42 |  |  |
| 35 | 145.6 | 202.34 |  |  |
| 36 | 143.37 | 202.53 |  |  |
| 37 | 141.23 | 202.76 |  |  |

With the given hyperparameters (that are not necessarily optimal – we are running a comparison with the inclusion of the graph input and the addition of senses) we obtain a min. Valid-PPL of 202 on SemCor, with a vocabulary of globals of |V|=21,988

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

We examine the performance of the same model that brough us improvements on WikiText-2 compared to the simple GRU – i.e. the model that includes the node-state of the current global, obtained from the graph by applying a GAT.

**Parameters:**

[('module.X', torch.Size([99963, 300]), True), ('module.select\_first\_node', torch.Size([1]), False), ('module.nodestate\_zeros', torch.Size([1, 300]), False), ('module.memory\_h1', torch.Size([1, 600]), False), ('module.memory\_h2', torch.Size([1, 300]), False), ('module.gat.weight', torch.Size([300, 300]), True), ('module.gat.att', torch.Size([1, 4, 150]), True), ('module.gat.bias', torch.Size([300]), True), ('module.U\_z\_1.weight', torch.Size([600, 600]), True), ('module.W\_z\_1.weight', torch.Size([600, 600]), True), ('module.U\_r\_1.weight', torch.Size([600, 600]), True), ('module.W\_r\_1.weight', torch.Size([600, 600]), True), ('module.U\_1.weight', torch.Size([600, 600]), True), ('module.U\_1.bias', torch.Size([600]), True), ('module.W\_1.weight', torch.Size([600, 600]), True), ('module.W\_1.bias', torch.Size([600]), True), ('module.U\_z\_2.weight', torch.Size([300, 300]), True), ('module.W\_z\_2.weight', torch.Size([300, 600]), True), ('module.U\_r\_2.weight', torch.Size([300, 300]), True), ('module.W\_r\_2.weight', torch.Size([300, 600]), True), ('module.U\_2.weight', torch.Size([300, 300]), True), ('module.U\_2.bias', torch.Size([300]), True), ('module.W\_2.weight', torch.Size([300, 600]), True), ('module.W\_2.bias', torch.Size([300]), True), ('module.linear2global.weight', torch.Size([21988, 300]), True), ('module.linear2global.bias', torch.Size([21988]), True)]

Number of trainable parameters=39.67M, where 6.62M softmax, 29.99M embeddings, ~3.06M core

### Mini-Experiment – Overfit on a fragment of SemCor

### Experiment – GRU+GAT4 on SemCor

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Training hyperparameters** | | **Input signals & GNN** | | **Architecture** | |
| batch\_size | 40 | grapharea | I: 32n | II: 181e | GRU layers | 2 |
| learning rate | 1\* 10^(-4) | Input signals | 1) current word embedding | hidden dim.s | 600,300 |
| TBPTT length | 35 | 2) current word node-state | Dropout | 0.1 on GRU W1, W2 |
|  |  | GNN | GAT, 4 heads |  |  |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| *Epoch* | *Run 1* | | *Run 2* | |
|  | *Training PPL* | *Validation PPL* | *Training PPL* | *Validation PPL* |
| 1 | 1097.37 | 515.14 |  |  |
| 2 | 604.26 | 363.72 |  |  |
| 3 | 442.33 | 314.53 |  |  |
| 4 | 384.1 | 290.12 |  |  |
| 5 | 348.03 | 274.99 |  |  |
| 6 | 321.38 | 262.19 |  |  |
| 7 | 299.82 | 251.3 |  |  |
| 8 | 281.92 | 243.1 |  |  |
| 9 | 266.89 | 236.23 |  |  |
| 10 | 254.27 | 230.96 |  |  |
| 11 | 243.59 | 226.13 |  |  |
| 12 | 234.27 | 222.1 |  |  |
| 13 | 225.63 | 218.67 |  |  |
| 14 | 217.94 | 215.57 |  |  |
| 15 | 210.91 | 212.6 |  |  |
| 16 | 204.41 | 210.59 |  |  |
| 17 | 198.69 | 208.53 |  |  |
| 18 | 193.23 | 207.21 |  |  |
| 19 | 188.31 | 205.75 |  |  |
| 20 | 183.83 | 204.55 |  |  |
| 21 | 179.44 | 203.63 |  |  |
| 22 | 175.5 | 202.84 |  |  |
| 23 | 171.81 | 202.28 |  |  |
| 24 | 168.29 | 201.88 |  |  |
| 25 | 164.92 | 201.69 |  |  |
| 26 | 161.75 | 201.46 |  |  |
| 27 | 158.71 | 201.5 |  |  |
| 28 | 155.78 | 201.52 |  |  |
| 29 | 152.78 | 201.72 |  |  |
| 30 | 150.03 (157.97) | 201.55 (202.63) |  |  |

Training on the SemCor dataset, and evaluating the perplexity on the global words, we observe that including the KB input only gives a marginal improvement (201.5 vs 202.3).

## Senses: Photo-concat

### Architecture

Consideration: It would be useful to   
Idea: take a “photo” of the encoding produced by the 2-layer GRU. We copy the value, with no gradient.

The input from the current word can be given as the concatenation of (global) Word Embedding + (global) Node State + (sense) Node State.

Context data + current word data are then passed on to a FF-NN.



Parameters:

[('module.X', torch.Size([99963, 300]), True), ('module.select\_first\_node', torch.Size([1]), False), ('module.nodestate\_zeros', torch.Size([1, 300]), False), ('module.memory\_h1', torch.Size([1, 600]), False), ('module.memory\_h2', torch.Size([1, 300]), False), ('module.gat\_globals.weight', torch.Size([300, 300]), True), ('module.gat\_globals.att', torch.Size([1, 4, 150]), True), ('module.gat\_globals.bias', torch.Size([300]), True), ('module.gat\_senses.weight', torch.Size([300, 300]), True), ('module.gat\_senses.att', torch.Size([1, 4, 150]), True), ('module.gat\_senses.bias', torch.Size([300]), True), ('module.U\_z\_1.weight', torch.Size([600, 600]), True), ('module.W\_z\_1.weight', torch.Size([600, 900]), True), ('module.U\_r\_1.weight', torch.Size([600, 600]), True), ('module.W\_r\_1.weight', torch.Size([600, 900]), True), ('module.U\_1.weight', torch.Size([600, 600]), True), ('module.U\_1.bias', torch.Size([600]), True), ('module.W\_1.weight', torch.Size([600, 900]), True), ('module.W\_1.bias', torch.Size([600]), True), ('module.U\_z\_2.weight', torch.Size([300, 300]), True), ('module.W\_z\_2.weight', torch.Size([300, 600]), True), ('module.U\_r\_2.weight', torch.Size([300, 300]), True), ('module.W\_r\_2.weight', torch.Size([300, 600]), True), ('module.U\_2.weight', torch.Size([300, 300]), True), ('module.U\_2.bias', torch.Size([300]), True), ('module.W\_2.weight', torch.Size([300, 600]), True), ('module.W\_2.bias', torch.Size([300]), True), ('module.linear2global.weight', torch.Size([21988, 300]), True), ('module.linear2global.bias', torch.Size([21988]), True), ('module.linear2sense.weight', torch.Size([25986 , 1200]), True), ('module.linear2sense.bias', torch.Size([25986]), True)]

Number of trainable parameters=71.51M, where Softmax=6.62M+31.21M, Embeddings=29.99M, core=3.69M (from the 3.06M with 1 GAT)

### Experiment – Globals and senses (GRU + 2xGAT4) on SemCor

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Training hyperparameters** | | **Input signals & GNN** | | **Architecture** | |
| batch\_size | 40 | grapharea | I: 32n | II: 181e | GRU layers | 2 |
| learning rate | 1\* 10^(-4) | Input signals | 1) current word embedding | hidden dim.s | 600,300 |
| TBPTT length | 35 | 2) current word node-state | Dropout | 0.1 on GRU W1, W2 |
| GNN | 2 GATs, 4 heads |  | 3) current *sense* node-state |  |  |

We also have sense prediction, that works using just a FF-NN on the concatenated input: copied h2 from the GRU || word embedding || global word node-state || sense node-state ||

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| *Epoch* | *Training perplexity* | | *Validation perplexity* | |
|  | Globals | Senses | Globals | Senses |
| 1 | 1104.34 | 5368.16 | 510.79 | 849.74 |
| 2 | 587.02 | 2973.06 | 353.61 | 758.61 |
| 3 | 417.42 | 2212.39 | 289.25 | 693.2 |
| 4 | 342.46 | 1815.9 | 261.3 | 675.57 |
| 5 | 301.03 | 1591.29 | 245.63 | 672.9 |
| 6 | 274.34 | 1432.27 | 235.33 | 675.28 |
| 7 | 255.2 | 1305.72 | 227.81 | 681.36 |
| 8 | 240.04 | 1199.06 | 222.46 | 687.76 |
| 9 | 227.35 | 1107.27 | 216.99 | 699.28 |
| 10 | 216.69 | 1025.38 | 213.57 | 712.9 |
| 11 | 207.53 | 951.97 | 210.25 | 726.85 |

It caused early stopping. The senses use a FF-NN that takes in as input: h2 ++ (word ++ global node ++ sense node). They depend on h2, and as h2 is tuned and modified only depending on the globals’ loss.

Maybe if h1 was shared (with gradient, not only as a copy) between the two tasks, and then we had a 2nd separate GRU layer?

## Senses: Shared GRU layer

### Architecture



**Parameters:**

[('module.X', torch.Size([99963, 300]), True), ('module.select\_first\_node', torch.Size([1]), False), ('module.nodestate\_zeros', torch.Size([1, 300]), False), ('module.memory\_h1', torch.Size([1, 600]), False), ('module.memory\_h2', torch.Size([1, 300]), False), ('module.memory\_h2b', torch.Size([1, 300]), False), ('module.gat\_globals.weight', torch.Size([300, 300]), True), ('module.gat\_globals.att', torch.Size([1, 4, 150]), True), ('module.gat\_globals.bias', torch.Size([300]), True), ('module.gat\_senses.weight', torch.Size([300, 300]), True), ('module.gat\_senses.att', torch.Size([1, 4, 150]), True), ('module.gat\_senses.bias', torch.Size([300]), True), ('module.U\_z\_1.weight', torch.Size([600, 600]), True), ('module.W\_z\_1.weight', torch.Size([600, 900]), True), ('module.U\_r\_1.weight', torch.Size([600, 600]), True), ('module.W\_r\_1.weight', torch.Size([600, 900]), True), ('module.U\_1.weight', torch.Size([600, 600]), True), ('module.U\_1.bias', torch.Size([600]), True), ('module.W\_1.weight', torch.Size([600, 900]), True), ('module.W\_1.bias', torch.Size([600]), True), ('module.U\_z\_2.weight', torch.Size([300, 300]), True), ('module.W\_z\_2.weight', torch.Size([300, 600]), True), ('module.U\_r\_2.weight', torch.Size([300, 300]), True), ('module.W\_r\_2.weight', torch.Size([300, 600]), True), ('module.U\_2.weight', torch.Size([300, 300]), True), ('module.U\_2.bias', torch.Size([300]), True), ('module.W\_2.weight', torch.Size([300, 600]), True), ('module.W\_2.bias', torch.Size([300]), True), ('module.U\_z\_2b.weight', torch.Size([300, 300]), True), ('module.W\_z\_2b.weight', torch.Size([300, 600]), True), ('module.U\_r\_2b.weight', torch.Size([300, 300]), True), ('module.W\_r\_2b.weight', torch.Size([300, 600]), True), ('module.U\_2b.weight', torch.Size([300, 300]), True), ('module.U\_2b.bias', torch.Size([300]), True), ('module.W\_2b.weight', torch.Size([300, 600]), True), ('module.W\_2b.bias', torch.Size([300]), True), ('module.linear2global.weight', torch.Size([21988, 300]), True), ('module.linear2global.bias', torch.Size([21988]), True), ('module.linear2sense.weight', torch.Size([25986, 300]), True), ('module.linear2sense.bias', torch.Size([25986]), True)]

Number of trainable parameters=48.93M, where Softmax = 6.62M + 7.82M, Embeddings=29.99M, core=4.5M   
(2 GATs, and 2 second layers of the GRU for the 2 different tasks of globals and senses)

### Experiment 1 – Senses: Shared GRU Layer

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Training hyperparameters** | | **Input signals & GNN** | | **Architecture** | |
| batch\_size | 40 | grapharea | I: 32n | II: 181e | GRU layers | 2 |
| learning rate | 1\* 10^(-4) | Input signals | 1) current word embedding | hidden dim.s | 600,300 |
| TBPTT length | 35 | 2) current word node-state | Dropout | 0.1 on GRU W1, W2 |
| GNN | 2 GATs, 4 heads |  | 3) current *sense* node-state |  |  |

**Run n. 1:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| *Epoch* | *Training perplexity* | | *Validation perplexity* | |
|  | Globals | Senses | Globals | Senses |
| 1 | 1106.28 | 5481.8 | 497.85 | 906.28 |
| 2 | 543.93 | 3233.42 | 346.43 | 792.47 |
| 3 | 413.44 | 2691.96 | 297.21 | 746.68 |
| 4 | 352.45 | 2291.23 | 272.99 | 708.21 |
| 5 | 315.81 | 2028.78 | 258.67 | 683.04 |
| 6 | 291.14 | 1839.55 | 249.37 | 662.51 |
| 7 | 272.99 | 1693.47 | 242.19 | 651.76 |
| 8 | 258.16 | 1572.48 | 237.35 | 643.23 |
| 9 | 245.72 | 1469.71 | 233.13 | 641.04 |
| 10 | 235.15 | 1378.23 | 229.57 | 642.25 |
| 11 | 225.58 | 1298.84 | 225.68 | 644.98 |
| 12 | 217.04 | 1229.62 | 222.35 | 647.24 |
| 13 | 209.12 | 1163.16 | 219.15 | 650.73 |
| 14 | 202.15 | 1103.04 | 216.29 | 654.98 |
| 15 | 195.66 | 1049.54 | 214.35 | 662.43 |
| 16 | 189.7 | 998.4 | 212.92 | 667.79 |

The perplexity on globals with all 3 input signals was interesting, and it looked on track to overtake previous results before the early stopping was triggered.

It seems that even the current architecture, that shares 1 layer of the GRU between the Global and the Sense tasks and then uses GRU Layer 2 > FF-NN to logits for each, is not good at dealing with senses.

Alternatives:

* 1. Separate the 2 tasks entirely: use 2 GRUs, 1 for globals and 1 for senses
  2. Use the prediction of the next global to influence or restrict the prediction of the next sense.

Of these, **(b)** is the one that warrants exploration the most.

## Senses: Self-attention scores on the senses of the k most likely globals

### Design

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

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

What do I need to implement this hypothesis:

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

Then, either:

1. use the senses architecture as normal; assign to 0 all the logits of senses that are not of the likely globals; softmax&predict
2. consider the first… n=e.g.5\*k senses. Select their embeddings. This would give us a matrix of n (e.g.500) x 300. We have a soft-classification task, where we need to choose one of the rows as the correct one, with a probability distribution over them.  
   To do this, I need several pieces of information:
   * The embeddings of the first n senses, as stated
   * The preceding context, after which we are making the prediction. I could copy h2 from the globals’GRU again?
   * The current token. I may bring the whole input (global word embedding || global node state || sense node state), or only part of it.

The main change of (B) is that, instead of having to create a probability distribution over the whole set of 25K+ senses, the distribution p would be over a small number of candidates, determined by the globals I am currently predicting.

Given: n embeddings of d=300; context information; current token;

how to obtain a probability distribution p over the n sense embeddings, to determine the next sense?

Possibility: Use self-attention.

### Reviewing self-attention

We start with N inputs.

Every input must have three representations: key, query, and value. They do not necessarily have the same number of dimensions as the input (e.g. from 512 to 64, or from 300 to 100, thus allowing for multi-headed self-attention if needed).

We need to use 3 projection matrices: Wq, Wk, Wv.

Their operate on: [n x (d x dq)], [n x (d x dk)], [n x (d x dv)].

The self-attention logit score of word2 from the point of view of word1, we proceed as follows:   
q1\*k2, the dot product of the point of view’s query vector with the key vector of the word we’re scoring.

We obtain: s11= q1\*k1, s12= q1\*k2, s13= q1\*k3, s14= q1\*k4, …

Then, we divide: s11/sqrt(dk) , by the square root of the dimension of the key vectors, and we apply the softmax over the window to obtain the self-attention score.

The next step is to multiply the self-attention score per the Value vector at that position.

The final step is to sum up the value vectors in the window, weighted by the scores from position 1.

I see how self-attention can be useful for the senses’ architecture:

The query q1 will always be the context.

The key will be projected from the candidate senses that we have selected. Likewise the value.

I could also use a simpler alternative, i.e. only the first part of the self-attention mechanism:

s11= q1\*k1, s12= q1\*k2,… etc. where q<- context and k <- sense embedding,  
followed by /sqrt(dk) and the softmax.

### Implementation

**1)** Given the logits of the globals for the current prediction, select the indices of those globals with the greatest k=100 logits.

**2)** The GNN also has a reference to the graph object. We want to retrieve the indices of the senses of the k=100 most likely globals.  
We need the indices of the immediate neighbours. So, n\_hops=1.

We do not need to check the edge\_index and edge\_type. Instead: does the index of the neighbour falls into the range of the senses in the embeddings matrix X, [0, last\_idx\_senses]?

**3)** Once wehave the indices of the senses of the most-likely-globals, we call X.index\_select(…) to retrieve their embeddings.

**4)** Multiply self.memory\_h2 x self.Wq to obtain the query (in this version, the query of the self-attention mechanism is always the same, derived from the current context).

Multiply self.likely\_senses\_embs x self.Wk, to project the senses embeddings into the keys of the self-attention mechanism (currently, we also change the dimensionality from 300 to 150).

*note/issue*: A global has, at minimum, 0 senses (not even 1), e.g. ‘for’ ,’of’ and any other stopwords that we do not find in the KB/dictionary.

Thus, in the self-attention mechanism, we may have: keys.shape=torch.Size([92, 100])

To solve it: pad it with 0s? But some scores are <0… I can index manually.

*note/issue*: at start, the matrix Wk has values ranging from 10^-38 to 10^38, and thus the matrix multiplication gives NaN. It is necessary to adjust its initialization.

**5)** Multiply query x keys, and divide by sqrt(dk) in order to obtain the attention logits.

**6)** Apply softmax on the <= k attention logits. We have now a probability distribution over the senses of the most likely globals.

**7)** We have to assign the self-attention scores to the selected senses, using their indices. All other senses in the vocabulary must have probability =0.

*Issue*:

I am assigning manually 0 to the sample’s predicted senses… however, in the globals and in the previous senses architecture, this element is not obtained with a softmax but with a tf.nn.functional.log\_softmax:

From the docs of pytorch, on tfunc.log\_softmax:  
“While mathematically equivalent to log(softmax(x)), doing these two operations separately is slower, and numerically unstable.  This function uses an alternative formulation to compute the output and gradient correctly.”

From the docs of pytorch, on torch.nn.LogSoftmax:

“[This function] returns a Tensor of the same dimension and shape as the input with values in the range [-inf, 0)”  
Hypothesis: I can simply assign -inf to “zero out” the senses that do not come from the most likely predicted globals?

No. loss\_sense=inf

Among the options of tfunc.nll\_loss(…):

**ignore\_index** (*python:int, optional*) – Specifies a target value that is ignored and does not contribute to the input gradient. When size\_average is True, the loss is averaged over non-ignored targets. **Default: -100**

### Experiments

The preliminary experiment on a fragment of SemCor does not even manage to overfit on a small training set:

Training, end of epoch 300. Global step n.1200. Time = 47826.22. ¨

The training losses are:

Losses: Globals loss=4.19 Sense loss=61.69 Total loss=65.876

Perplexity: Globals perplexity=66.0

Sense perplexity=6.1632970018788446**e+26**

It is necessary to find another architecture.

# 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)

# Senses Baseline: Operate on the k most likely globals

After we have the logits from the globals’ prediction, we can sort them descendingly and obtain the indices of the ***k* most likely globals**.

Then… for each of the *k* globals we retrieve the indices of the neighbours, and filter out those that are not in the range of senses?

~~Question: do we have a guarantee on how many senses we retrieve this way?~~

~~A global may have multiple senses in the dictionary, or only one, or even 0 (see: all the stopwords: ‘for’, ‘and’, ‘of’ etc.).~~

~~We do not have a guarantee.~~

~~It would be opportune to retrieve, let’s say, at most~~ *~~3\*k~~* ~~senses and have a mechanism that also works if we have strictly less than~~ ***~~3\*k~~* ~~senses~~**~~, by ignoring all 0/empty values.~~

~~Possible answer: create the logits for 3k senses. If we got less than 3k senses from the k most likely globals, fill with random sense indices. Apply log-softmax. Assign the log-softmax values to the output – in nll\_loss we can ignore part of the values.~~

Idea:

Filter the most likely k globals.

Send the logits over k most likely globals. k can be fixed or dynamic.

* + - look up: structured prediction.

There is a network that takes in input from the globals’ prediction and opens up to the senses’ logits over the whole vocabulary.

Hypotheses:

* take in the entirety of the globals’ logits, and have a FF-NN lead to the senses’ logits.  
  The dimensions of the Linear network would be: 21.9K x 25.9K =~ 567M parameters -> no.
* take in the embeddings of the k most likely globals, and then lead to the senses’ logits. (k=50 x d=300) -> 25.9K = 388M parameters, no.
* Make a structure with a globals-to-senses correspondence. If 1 (input) neuron == 1 global and 1(output, logit) neuron == 1 sense… I would need to create this structure beforehand, from the graph\_dataobject… wouldn’t it just return the most likely sense?
* represent the indices of the globals in 1-hot encoding… No. This would be worse than the first alternative that uses the logits as they are.
* Forcefully project the globals’ embeddings to a lower dimensionality, let’s say 300 -> 50. Then, (d=50++logit)\*k > senses’ vocabulary, which means (51\*50)\*25.9K =~67M
* Maybe, if I chose fewer globals? (20x300) x 25.9K = 155..., it could be viable.
* Passing on to the senses’ network: the globals’ projected embeddings, and their logits. I can choose relatively few globals and add an intermediate FF-NN.  
  (300 x 100) + (30x(100+1) x 25.9K = 6000 + 77.9M = 78.5M.

## Hypothesis 1: FF-NN – from all logits to all logits

Take in the entirety of the globals’ logits, and have a FF-NN lead to the senses’ logits.  
The dimensions of the Linear network would be: 21.9K x 25.9K =~ 567M parameters

*However,* we wish to limit the update to the logits of the first most likely *k* globals.

The globals’ logits, sorted descendingly:

tensor([tensor([0.0969, 0.0945, 0.0936, 0.0922, 0.0922, 0.0914, 0.0893, 0.0891, 0.0891,

0.0864,…, 0.0855, 0.0851, 0.0851,x -0.0871, -0.0877, -0.0887],)

And if we executed tfunc.softmax(logits\_global)?

We get a Tensor of 21988, that currently (in iteration number 1, just after the initialization) ranges from 5.0074e-05 to 4.1589e-05.

I can zero out all the softmax values that do not come from the most likely globals.

RunTimeError: CUDA out of memory. We can not have 600+M parameters.

## Hypothesis 2: Project embeddings of the first k globals + concat logit + FF-NN

* Select the first k globals with the greatest logits
* Extract their embeddings that have d=300, and use a FF-NN P to project them down to d\_p=50
* For each global, concatenate (projected embedding ++ logit value). We obtain a matrix PL of 20x51
* Flatten PL into a vector of d=1020. Apply a FF-NN from it to the ~26K senses’ logits

If I operate with the globals alone, that use the GRU-GAT4, I manage to overfit on a fragment of SemCor.

If I include this mechanism for the senses, I do not:

Perplexity: Globals perplexity=45.32 Sense perplexity=37.46

Let us try without the projection, simply concatenating the embeddings and the logits.

… More or less no improvement.

And what if I detached and clone-copied properly the embeddings of the globals, so that the sense task can not modify them?

I still need to project, or I run into CUDAoutOfMemory.

If I project with a higher dimension, like to d=100?

Considerably more interesting, to the point of being valid:

After 200 epochs at lr=0.001, with k=10 and dp=100,

Perplexity: Globals perplexity=1.39 Sense perplexity=4.59

After 500 epochs at lr=0.0001, with k=10 and dp=150?

It does NOT work: Perplexity: Globals perplexity=45.3 Sense perplexity=36.59

After 300 epochs at lr=0.001, with k=10 and dp=150,

Perplexity: Globals perplexity=1.01 Sense perplexity=1.06

### Model

(module): ProjectK(

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

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

(U\_z\_1): Linear(in\_features=600, out\_features=600, bias=False)

(W\_z\_1): Linear(in\_features=900, out\_features=600, bias=False)

(U\_r\_1): Linear(in\_features=600, out\_features=600, bias=False)

(W\_r\_1): Linear(in\_features=900, out\_features=600, bias=False)

(U\_1): Linear(in\_features=600, out\_features=600, bias=True)

(W\_1): Linear(in\_features=900, out\_features=600, bias=True)

(dropout): Dropout(p=0.1, inplace=False)

(U\_z\_2): Linear(in\_features=300, out\_features=300, bias=False)

(W\_z\_2): Linear(in\_features=600, out\_features=300, bias=False)

(U\_r\_2): Linear(in\_features=300, out\_features=300, bias=False)

(W\_r\_2): Linear(in\_features=600, out\_features=300, bias=False)

(U\_2): Linear(in\_features=300, out\_features=300, bias=True)

(W\_2): Linear(in\_features=600, out\_features=300, bias=True)

(linear2global): Linear(in\_features=300, out\_features=21988, bias=True)

(P): Linear(in\_features=300, out\_features=150, bias=False)

(projs2senselogits): Linear(in\_features=1510, out\_features=25986, bias=True) )

)

### Experiment on SemCor

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Training hyperparameters** | | **Input signals & GNN** | | **Architecture** | |
| batch\_size | 40 | grapharea | I: 32n | II: 181e | GRU layers | 2 |
| learning rate | 1\* 10^(-4) | Input signals | 1) current word embedding | hidden dim.s | 600,300 |
| TBPTT length | 35 | 2) current word node-state | Senses | take the first k=20 globals, project to d=150, concat logit, apply FF-NN |
| GNN | 2 GATs, 4 heads |  | 3) current *sense* node-state |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| *Epoch* | *Training perplexity* | | *Validation perplexity* | |
|  | Globals | Senses | Globals | Senses |
| 1 | 1116.99 | 5935.0 | 512.91 | 900.14 |
| 2 | 566.75 | 3449.23 | 355.98 | 852.81 |
| 3 | 424.93 | 3327.42 | 306.61 | 849.9 |
| 4 | 366.87 | 3219.58 | 281.45 | **893.32** |
| 5 | 329.19 | 3060.49 | 265.64 | 925.83 |
| 10 | 237.56 | 2258.25 | 229.99 | 916.87 |
| 15 | 198.91 | 1844.59 | 218.56 | 911.01 |
| 20 | 172.93 | 1613.02 | 212.85 | 944.02 |
| 26 | 150.58 | 1435.92 | 209.93 | 1045.72 |
| 50 | 101.09 | 1150.71 | 226.89 | 1635.5 |

We see that the ProjectK method (k globals -> project embeddings down to d=150 -> apply FF-NN to the logits) does not work, and in fact compares unfavourably with previous experiments (see the 642 Vald PPL obtained by sharing the 1st layer of a GRU)

# Senses: self-attention on the k most likely globals

## Design

Instead of trying to use as prediction the self-attention scores derived by queries and values on the senses of the likely globals,

I can operate with the full self-attention mechanism:

* **query**: **the input and/or the context**. Project it with Wq
* **keys**: either **the k likely globals**, or their senses. Project them with Wk
* **values**: **the k likely globals/**senses. Project them with Wv.

We computed self-attention scores from sofmax(Q\*K/sqrt(d\_k)).

Our result can be Σ(score\*value).

Finally, we use this result as the input of a FF-NN that leads to the senses’ logits.

## Model

(module): SelfAttK(

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

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

(U\_z\_1): Linear(in\_features=600, out\_features=600, bias=False)

(W\_z\_1): Linear(in\_features=900, out\_features=600, bias=False)

(U\_r\_1): Linear(in\_features=600, out\_features=600, bias=False)

(W\_r\_1): Linear(in\_features=900, out\_features=600, bias=False)

(U\_1): Linear(in\_features=600, out\_features=600, bias=True)

(W\_1): Linear(in\_features=900, out\_features=600, bias=True)

(dropout): Dropout(p=0.1, inplace=False)

(U\_z\_2): Linear(in\_features=300, out\_features=300, bias=False)

(W\_z\_2): Linear(in\_features=600, out\_features=300, bias=False)

(U\_r\_2): Linear(in\_features=300, out\_features=300, bias=False)

(W\_r\_2): Linear(in\_features=600, out\_features=300, bias=False)

(U\_2): Linear(in\_features=300, out\_features=300, bias=True)

(W\_2): Linear(in\_features=600, out\_features=300, bias=True)

(linear2global): Linear(in\_features=300, out\_features=21988, bias=True)

(mySelfAttention): SelfAttention(

(Wq\_ls): ModuleList(

(0): Linear(in\_features=900, out\_features=150, bias=False)

(1): Linear(in\_features=900, out\_features=150, bias=False)

(2): Linear(in\_features=900, out\_features=150, bias=False)

(3): Linear(in\_features=900, out\_features=150, bias=False))

(Wk\_ls): ModuleList(

(0): Linear(in\_features=300, out\_features=150, bias=False)

(1): Linear(in\_features=300, out\_features=150, bias=False)

(2): Linear(in\_features=300, out\_features=150, bias=False)

(3): Linear(in\_features=300, out\_features=150, bias=False))

(Wv\_ls): ModuleList(

(0): Linear(in\_features=300, out\_features=150, bias=False)

(1): Linear(in\_features=300, out\_features=150, bias=False)

(2): Linear(in\_features=300, out\_features=150, bias=False)

(3): Linear(in\_features=300, out\_features=150, bias=False)

))

(linear2senses): Linear(in\_features=600, out\_features=25986, bias=True)))

56.81M parameters, where:

Embeddings=29.99M ; GRU-GAT4(globals)=3.69M; globals’ softmax=6.59M ;

Self-attention4Senses=0.9M; senses’softmax=15.59M

## Experiments

### Overfit on mini-dataset

**Mini-experiment n.1**

Objective: Overfit on small training dataset (fragment of SemCor)

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Training hyperparameters** | | **Input signals & GNN** | | **Architecture** | |
| batch\_size | 4 | grapharea | I: 32n | II: 181e | GRU layers | 2 |
| learning rate | 1\* 10^(-3) | Input signals | 1) current word embedding | hidden dim.s | 600,300 |
| TBPTT length | 16 | 2) current word node-state | Senses self-attention | Q=Input signals. K,V=k globals.  4 heads(150). |
| GNN | 2 GATs, 4 heads(75) |  | 3) current *sense* node-state | Senses | select k globals > Self-attention > FF-NN to logits |

|  |  |  |
| --- | --- | --- |
| *Epoch* | *Training perplexity* | |
|  | Globals | Senses |
| 1 | 19949.4 | 25724.72 |
| 5 | 86.97 | 6254.64 |
| 10 | 46.06 | 38.52 |
| 50 | 37.9 | 8.77 |
| 100 | 28.25 | 3.72 |
| 200 | 1.11 | 1.68 |

Self-attention mechanism on the k most likely globals, operating with 4 Self-attention heads, of dimension 150, concatenated and used as input for the FF-NN to the logits.

Overfitting on fragment: confirmed.

### Experiment 1 – SelfAttention4Senses on SemCor

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Training hyperparameters** | | | | **Input signals & GNN** | | | | **Architecture** | | |
| batch\_size | | 40 | | grapharea | I: 32n | II: 181e | | | GRU layers | 2 | |
| learning rate | | 1\* 10^(-4) | | Input signals | 1) current word embedding | | | hidden dim.s | 600,300 | |
| TBPTT length | | 35 | | 2) current word node-state | | | Senses self-attention | Q=Input signals. K,V=k globals.  4 heads(150). | |
| GNN | | 2 GATs, 4 heads(75) | |  | 3) current *sense* node-state | | | Senses | select k=**20** globals > Self-attention > FF-NN to logits | |
| *Epoch* | *Training perplexity* | | | | | *Validation perplexity* | | | |
|  | Globals | | Senses | | | Globals | Senses | | |
| 1 | 1102.24 | | 6459.86 | | | 509.92 | 915.97 | | |
| 2 | 584.56 | | 3280.37 | | | 354.48 | 813.53 | | |
| 3 | 419.8 | | 3039.59 | | | 294.14 | 754.32 | | |
| 4 | 351.11 | | 2540.84 | | | 268.19 | 702.9 | | |
| 5 | 309.74 | | 2215.51 | | | 252.11 | 677.11 | | |
| 6 | 282.58 | | 2055.59 | | | 240.86 | 674.35 | | |
| 7 | 262.72 | | 1907.36 | | | 232.3 | 658.85 | | |
| 8 | 247.06 | | 1763.76 | | | 226.13 | **653.31** | | |
| 9 | 234.11 | | 1651.74 | | | 221.02 | 660.12 | | |
| 10 | 223.34 | | 1552.69 | | | 216.96 | 672.16 | | |
| 11 | 213.92 | | 1461.75 | | | 213.29 | 684.77 | | |
| 12 | 205.52 | | 1384.63 | | | 210.6 | 698.59 | | |
| 13 | 197.92 | | 1311.55 | | | 208.43 | 714.57 | | |
| 14 | 191.08 | | 1247.08 | | | 206.46 | 729.69 | | |
| 15 | 184.86 | | 1190.14 | | | 205.09 | 751.59 | | |
| 16 | 179.05 | | 1138.66 | | | 204.13 | 766.07 | | |
| 17 | 173.95 | | 1094.81 | | | 203.21 | 786.61 | | |
| 18 | 168.91 | | 1050.6 | | | 202.68 | 808.82 | | |
| 19 | 164.35 | | 1012.83 | | | 202.39 | 835.68 | | |
| 20 | 160.2 | | 978.71 | | | 201.96 | 857.45 | | |
| 24 | 145.57 | | 863.83 | | | **201.19** | 947.67 | | |
| 25 | 142.33 | | 840.49 | | | 201.37 | 973.09 | | |
| 30 | 128.23 | | 742.04 | | | 203.09 | 1104.24 | | |

### Experiment 2 – SelfAttention4Senses on SemCor

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Training hyperparameters** | | **Input signals & GNN** | | **Architecture** | |
| batch\_size | 40 | grapharea | I: 32n | II: 181e | GRU layers | 2 |
| learning rate | 1\* 10^(-4) | Input signals | 1) current word embedding | hidden dim.s | 600,300 |
| TBPTT length | 35 | 2) current word node-state | Senses self-attention | Q=Input signals. K,V=k globals.  **2** heads(**300**). |
| GNN | 2 GATs, 4 heads(75) |  | 3) current *sense* node-state | Senses | select k=**10** globals > Self-attention > FF-NN to logits |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| *Epoch* | *Training perplexity* | | *Validation perplexity* | |
|  | Globals | Senses | Globals | Senses |
| 1 | 1097.78 | 6185.59 | 516.04 | 909.92 |
| 2 | 658.27 | 3298.34 | 390.96 | 848.34 |
| 3 | 454.04 | 3116.59 | 309.53 | 788.37 |
| 4 | 368.13 | 2712.82 | 277.21 | 729.81 |
| 5 | 322.18 | 2292.11 | 260.11 | 690.64 |
| 6 | 292.09 | 2063.6 | 249.21 | 672.24 |
| 7 | 270.47 | 1919.19 | 240.78 | **671.64** |
| 8 | 253.61 | 1808.5 | 234.19 | 673.24 |
| 9 | 239.91 | 1707.75 | 228.42 | 682.51 |
| 10 | 228.56 | 1621.85 | 223.41 | 701.12 |
| 15 | 189.62 | 1336.58 | 207.01 | 750.16 |
| 20 | 164.56 | 1147.05 | 200.74 | 820.51 |
| 25 | 146.27 | 1012.65 | 198.75 | 892.74 |
| 26 | 143.21 | 990.99 | **198.57** | 908.62 |
| 30 | 132.02 | 913.03 | 199.44 | 983.56 |
| 35 | 120.57 | 832.45 | 201.82 | 1095.15 |
| 40 | 110.95 | 762.55 | 205.71 | 1214.21 |
| 50 | 95.27 | 650.43 | 214.13 | 1470.57 |

# Globals – LSTM

## Design

Purposes:

- to obtain better Perplexity results on WikiText-2

- to confirm that including KB input brings an improvement, even in better models.

Idea on how to do so:

follow AWD-LSTM by S.Merity et al., 2017.

It is the basis for all high-performing models on WikiText-2, as we can see in <https://nlpprogress.com/>.

It has Valid-PPL=60.0 and Test-PPL=57.3, whereas there is a number of best-models that go from 58.9 to 44.8, but all of them are additions and modifications over AWD-LSTM.

### Elements in AWD-LSTM

On the side: I observe their statement:

“A naïve application of dropout (Srivastava et al., 2014) to an RNN’s hidden state is ineffective as it disrupts the RNN’s ability to retain long term dependencies”

and reduce the dropout on the GRU’s hidden states from 0.1 to 0.01

Reasoning on optimization methods:

“The choice of the optimizer is even more important in the context of regularized models since such strategies, especially the use of dropout, can impede the training process.

Stochastic gradient descent (SGD), and its variants such as Adam (Kingma & Ba, 2014) and RMSprop (Tieleman & Hinton, 2012) are amongst the most popular training methods.

These methods iteratively reduce the training loss through scaled (stochastic) gradient steps. In particular, Adam has been found to be widely applicable despite requiring less tuning of its hyperparameters.

In the context of word-level language modeling, past work has empirically found that SGD outperforms other methods in not only the final loss but also in the rate of convergence. This is in agreement with recent evidence pointing to the insufficiency of adaptive gradient methods (Wilson et al., 2017).”

**Averaged SGD**:

“ASGD carries out iterations similar to SGD, but instead of returning the last iterate as the solution, returns an average of the iterates past a certain, tuned, threshold T”

**DropConnect**:

“It does not require any modifications to an RNN’s formulation.

As the dropout operation is applied once to the weight matrices, before the forward and backward pass, the impact on training speed is minimal and any standard RNN implementation can be used”

“By performing DropConnect on the hidden-to-hidden weightmatrices [Ui,Uf ,Uo,Uc] within the LSTM, we can prevent overfitting from occurring on the recurrent connections”

“As the same weights are reused over multiple timesteps, the same individual dropped weights remain dropped for the entirety of the forward and backward pass. The result is similar to variational dropout, which applies the same dropout mask to recurrent connections within the LSTM by performing dropout on ht−1, except that the dropout is applied to the recurrent weights.”

Other elements are:

Variable length BPPT , Variational dropout, Embedding dropout, reducing word vector size.

**Weight tying** :

“It shares the weights between the embedding and softmax layer, substantially reducing the total parameter count in the model.”

+ **Activation Regularization** and **Temporal Activation Regularization**:

“AR penalizes activations that are significantly larger than 0 as

a means of regularizing the network. αL2(m⊙ ht)”

“Using the notation from AR, TAR is defined as

β L2(ht − ht+1)

where β is a scaling coefficient. As in Merity et al. (2017), the AR and TAR loss are only applied to the output of the final RNN layer as opposed to being applied to all layers.”

### Hyperpararameters used in AWD-LSTM

“All experiments use a **three-layer** LSTM model with **1150** units in the hidden layer and an embedding of size **400**.”

“For training the models, we use the NT-ASGD algorithm discussed in the previous section for **750** epochs with L equivalent to one epoch and n = 5.”

Batch size of **80** for WT2 and **40** for PTB.

“Empirically, we found relatively large batch sizes (e.g., 40-80) performed better than

smaller sizes (e.g., 10-20) for NT-ASGD”

“We use a random BPTT length which is **N(70, 5)** with probability

**0.95** and **N(35, 5)** with probability 0.05.

The values used for dropout on the word vectors [0.4], the output between LSTM layers, [0.3], the output of the final LSTM layer [0.4], and embedding dropout [0.1]

were (0.4, 0.3, 0.4, 0.1) respectively.”

### Their Perplexity Results

**Word level WikiText-2 (WT2) with LSTM**

The instruction below trains a PTB model that without finetuning achieves perplexities of approximately **68.7 / 65.6** (validation / testing), with finetuning achieves perplexities of approximately 67.4 / 64.7, and with the continuous cache pointer augmentation achieves perplexities of approximately 52.2 / 50.6.

**Word level WikiText-2 (WT2) with QRNN**

The instruction below will a QRNN model that without finetuning achieves perplexities of approximately 69.3 / 66.8 (validation / testing), with finetuning achieves perplexities of approximately 68.5 / 65.9, and with the continuous cache pointer augmentation achieves perplexities of approximately 53.6 / 52.1. Better numbers are likely achievable but the hyper parameters have not been extensively searched.

## Implementation

### Salesforce’s AWD-LSTM

Let us try to use the original Salesforce’s AWD-LSTM implementation, by S.Merity et al.

It works on PyTorch 0.4 – hopefully a few modifications should be enough to use the RNNModel in model.py ….

Using their default parameters, with nhid=1150, nlayers=3.

ninp=args.emsize – I have to use 300 instead of 400.

This is related to the fact that I am not learning the embeddings, I am using the pre-trained FastText embeddings.

They use: ntokens = len(corpus.dictionary) ; is this the vocabulary size?

Yes, since the usage is:

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

… self.decoder = nn.Linear(nhid, ntoken)

In torch.nn.Embedding(…),  the learnable weights of the module of shape (num\_embeddings, embedding\_dim) are initialized from N(0,1). I should instead write in my embeddings from the X matrix of the graph.

Problem 1: they package the batch differently, directly as a tensor of word indices, and then they call detach() on the hidden states in the main.py>train() function, , i.e. in the training loop.

Problem 2: Their implementation works only with PyTorch 0.4.0, not with anything >=1.4.0.

In our batches we have to send the graph data. Maybe we could recreate the implementation using available pytorch instruments, from LSTM cells to DropConnect?

### My own WD-LSTM

Let us start with recreating an LSTM, using the same hyperparameters used by the AWD-LSTM, as much as possible.

3 layers, each with 1150 hidden units.

Since using LSTMCells is complicated and requires a great level of detail, why not using stacked LSTMS and apply manually DropConnect on each of them?

PyTOrch’s LSTM Applies a multi-layer long short-term memory (LSTM) RNN to an **input sequence**.

i.e. we may not need the for cycle anymore,

**for** ((x\_indices\_g, edge\_index\_g, edge\_type\_g),

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

or – actually – we can keep it only to extract the input\_signals, and then send them in batch form to the LSTM.

**Issue:** Throws SegFault…

Someone reports: when I use DataParallel for LSTM model, it has segmentation fault after some batch. If I remove Dataparallel, it can work well.

**Issue:**

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

“ The \_apply function does not work correctly after the parameters of rnn have been renamed…”

“Unfortunately you can't fix it for pytorch 1.4.0, you need to get nightly package to get a fix.”

Eventually, I implemented manually the Weight-Dropping mechanism: we apply dropout (e.g. with p=0.3) to the specified named parameters of the model, and then apply the original forward() call.

## Experiments

### Mini-experiment 1 – overfit LSTM1 on fragment of WT2

INFO : LSTM(

(lstm): LSTM(300, 1150)

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

INFO : Parameters:

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

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

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

('memory\_cn', torch.Size([1, 1, 1150]), torch.float32, False)

('lstm.weight\_ih\_l0', torch.Size([4600, 300]), torch.float32, True)

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

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

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

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

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

INFO : Number of trainable parameters=77.394M, where embeddings=34.297M, softmax=36.417M, core = 6.68M

|  |  |  |
| --- | --- | --- |
| **Architecture** | **Input Signals** | **Hyperparameters** |
| Using LSTM, with 1 layers: 1150 | 1) The word embedding of the current global. d=300 | batch\_size=4 |
|  | TBPTT length=8 |
| Followed by linear2Globals FF-NN | learning rate=0.0001 |
|  |  |

|  |  |
| --- | --- |
| **Epoch** | **Training perplexity** |
| 1 | 30094.21 |
| 2 | 2451.76 |
| 3 | 115.34 |
| 4 | 73.08 |
| 5 | 66.52 |
| 10 | 64.37 |
| 20 | 60.92 |
| 30 | 55.71 |
| 50 | 31.34 |
| 75 | 13.04 |
| 100 | 4.73 |
| 199 | 1.13 |

### Mini-experiment 2 – overfit LSTM2 on fragment of WT2

Model:

INFO : LSTM(

(lstm): LSTM(300, 1150, num\_layers=2)

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

)

INFO : Parameters:

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

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

('memory\_hn', torch.Size([2, 1, 1150]), torch.float32, False)

('memory\_cn', torch.Size([2, 1, 1150]), torch.float32, False)

('lstm.weight\_ih\_l0', torch.Size([4600, 300]), torch.float32, True)

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

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

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

('lstm.weight\_ih\_l1', torch.Size([4600, 1150]), torch.float32, True)

('lstm.weight\_hh\_l1', torch.Size([4600, 1150]), torch.float32, True)

('lstm.bias\_ih\_l1', torch.Size([4600]), torch.float32, True)

('lstm.bias\_hh\_l1', torch.Size([4600]), torch.float32, True)

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

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

INFO : Number of trainable parameters=87.983M, where embeddings=34.297M, softmax=36.417M, core = 17.269M

|  |  |  |
| --- | --- | --- |
| **Architecture** | **Input Signals** | **Hyperparameters** |
| Using LSTM, with 2 layers: 1150, 1150 | 1) The word embedding of the current global. d=300 | batch\_size=4 |
|  | TBPTT length=8 |
| Followed by linear2Globals FF-NN | learning rate=0.0001 |
|  |  |

|  |  |
| --- | --- |
| **Epoch** | **Training perplexity** |
| 1 | 28306.16 |
| 2 | 811.06 |
| 3 | 114.38 |
| 4 | 74.24 |
| 5 | 67.07 |
| 10 | 64.67 |
| 50 | 64.64 |
| 75 | 30.51 |
| 100 | 8.1 |
| 199 | 1.13 |

### Mini-experiment 3 – overfit LSTM3 on fragment of WT2

Model:

INFO : LSTM(

(lstm): LSTM(300, 1150, num\_layers=3)

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

)

INFO : Parameters:

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

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

('memory\_hn', torch.Size([3, 1, 1150]), torch.float32, False)

('memory\_cn', torch.Size([3, 1, 1150]), torch.float32, False)

('lstm.weight\_ih\_l0', torch.Size([4600, 300]), torch.float32, True)

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

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

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

('lstm.weight\_ih\_l1', torch.Size([4600, 1150]), torch.float32, True)

('lstm.weight\_hh\_l1', torch.Size([4600, 1150]), torch.float32, True)

('lstm.bias\_ih\_l1', torch.Size([4600]), torch.float32, True)

('lstm.bias\_hh\_l1', torch.Size([4600]), torch.float32, True)

('lstm.weight\_ih\_l2', torch.Size([4600, 1150]), torch.float32, True)

('lstm.weight\_hh\_l2', torch.Size([4600, 1150]), torch.float32, True)

('lstm.bias\_ih\_l2', torch.Size([4600]), torch.float32, True)

('lstm.bias\_hh\_l2', torch.Size([4600]), torch.float32, True)

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

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

INFO : Number of trainable parameters=98.572M, where embeddings=34.297M, softmax=36.417M, core = 27.858M

|  |  |  |
| --- | --- | --- |
| **Architecture** | **Input Signals** | **Hyperparameters** |
| Using **LSTM**, with 3 layers: 1150, 1150, 1150 | 1) The word embedding of the current global. d=300 | batch\_size=8 |
|  | TBPTT length=8 |
| Followed by linear2Globals FF-NN | learning rate=0.001 |
|  |  |

|  |  |
| --- | --- |
| **Epoch** | **Training perplexity** |
| 1 | 2737.32 |
| 2 | 89.43 |
| 3 | 65.59 |
| 4 | 66.23 |
| 5 | 63.19 |
| 10 | 59.04 |
| 20 | 63.82 |
| 50 | 58.98 |
| 100 | 63.6 |
| 125 | 27.04 |
| 150 | 4.21 |
| 175 | 1.37 |

### Experiment 1 – LSTM3 on WT2

|  |  |  |
| --- | --- | --- |
| **Architecture** | **Input Signals** | **Hyperparameters** |
| Using **LSTM**, with 3 layers: 1150, 1150, 1150 | 1) The word embedding of the current global (d=300) | batch\_size=40 |
|  | TBPTT length=35 |
| Followed by linear2Globals FF-NN | learning rate=10^(-4) |
|  |  |

|  |  |  |
| --- | --- | --- |
| *Epoch* | *Training perplexity* | *Validation perplexity* |
| 1 | 350.14 | 293.12 |
| 2 | 221.63 | 191.58 |
| 3 | 180.26 | 172.51 |
| 4 | 162.78 | 159.46 |
| 5 | 147.32 | 146.72 |
| 6 | 133.01 | 137.48 |
| 7 | 120.91 | 129.64 |
| 8 | 111.31 | 124.08 |
| 9 | 103.84 | 120.44 |
| 10 | 97.68 | 117.66 |
| 11 | 92.56 | 115.09 |
| 12 | 88.01 | 112.51 |
| 13 | 84.0 | 110.83 |
| 14 | 80.43 | 109.76 |
| 15 | 77.19 | 109.2 |
| 16 | 74.15 | 108.61 |
| 17 | 71.36 | 108.21 |
| 18 | 68.82 | **108.04** |
| 19 | 66.42 | 108.15 |
| 20 | 64.14 | 108.55 |
| 21 | 62.06 | 108.85 |
| 22 | 60.04 | 109.49 |

### Experiment 2 – WD-LSTM3 on WT2

|  |  |  |
| --- | --- | --- |
| **Architecture** | **Input Signals** | **Hyperparameters** |
| Using **LSTM**, with 3 layers: 1150, 1150, 1150 | 1) The word embedding of the current global (d=300) | batch\_size=40 |
| **Weight-Dropping** mechanism with p=0.3 on the h2h LSTM weights | TBPTT length=35 |
| Followed by linear2Globals **FF-NN** | learning rate=10^(-4) |

|  |  |  |
| --- | --- | --- |
| *Epoch* | *Training perplexity* | *Validation perplexity* |
| 1 | 335.67 | 229.01 |
| 2 | 201.36 | 171.46 |
| 3 | 154.75 | 142.04 |
| 4 | 130.75 | 128.38 |
| 5 | 117.13 | 120.68 |
| 6 | 107.62 | 115.52 |
| 7 | 100.1 | 111.2 |
| 8 | 93.75 | 107.68 |
| 9 | 88.21 | 104.89 |
| 10 | 83.34 | 102.3 |
| 11 | 78.99 | 100.21 |
| 12 | 75.13 | **98.29** |
| 13 | process killed – too much load on several GPUs at once |  |

### Experiment 3 – WD-LSTM3 on WT2

|  |  |  |
| --- | --- | --- |
| **Architecture** | **Input Signals** | **Hyperparameters** |
| Using **LSTM**, with 3 layers: 1150, 1150, 1150 | 1) The word embedding of the current global (d=300) | batch\_size=40 |
| **Weight-Dropping** mechanism with p=0.3 on the h2h LSTM weights | TBPTT length=35 |
| Followed by linear2Globals **FF-NN** | learning rate=10^(-**5**) |

|  |  |  |
| --- | --- | --- |
| *Epoch* | *Training perplexity* | *Validation perplexity* |
| 1 | 464.73 | 302.74 |
| 2 | 311.06 | 300.64 |
| 3 | 309.53 | 300.6 |
| 4 | 309.15 | 300.44 |
| 5 | 308.9 | 300.23 |
| 6 | 308.74 | 300.96 |
| 7 | 308.58 | 299.94 |
| 8 | 292.77 | 262.15 |
| 9 | 250.3 | 233.08 |
| 10 | 230.5 | 219.34 |
| 11 | 219.12 | 210.14 |
| 12 | 210.41 | 202.45 |
| 13 | 203.25 | 196.18 |
| 14 | 197.25 | 191.62 |
| 15 | 192.1 | 187.61 |
| 16 | 187.53 | 183.73 |

At this point, I switch to implementing the full AWD-LSTM.

Nevertheless, a learning rate of 10^(-5) is too slow. Maybe 0.5\*10^(-4) would be more opportune.

# Globals – GRU

## Design

Instead of using my own implementation of the GRU, as I have done before, it may be more opportune to use the torch.nn.GRU. This would also confirm and consolidate the GRU results.

Parameters

* **input\_size** – The number of expected features in the input x
* **hidden\_size** – The number of features in the hidden state h
* **num\_layers** – Number of recurrent layers. E.g., setting num\_layers=2 would mean stacking two GRUs together to form a stacked GRU, with the second GRU taking in outputs of the first GRU and computing the final results. Default: 1
* **bias** – If False, then the layer does not use bias weights b\_ih and b\_hh. Default: True
* **batch\_first** – If True, then the input and output tensors are provided as (batch, seq, feature). Default: False
* **dropout** – If non-zero, introduces a Dropout layer on the outputs of each GRU layer except the last layer, with dropout probability equal to dropout. Default: 0
* **bidirectional** – If True, becomes a bidirectional GRU. Default: False

Inputs: input, h\_0

* **input** of shape (seq\_len, batch, input\_size): tensor containing the features of the input sequence. The input can also be a packed variable length sequence. See [torch.nn.utils.rnn.pack\_padded\_sequence()](https://pytorch.org/docs/stable/nn.html#torch.nn.utils.rnn.pack_padded_sequence) for details.
* **h\_0** of shape (num\_layers \* num\_directions, batch, hidden\_size): tensor containing the initial hidden state for each element in the batch. Defaults to zero if not provided. If the RNN is bidirectional, num\_directions should be 2, else it should be 1.

Outputs: output, h\_n

* **output** of shape (seq\_len, batch, num\_directions \* hidden\_size): tensor containing the output features h\_t from the last layer of the GRU, for each t. If a [torch.nn.utils.rnn.PackedSequence](https://pytorch.org/docs/stable/nn.html#torch.nn.utils.rnn.PackedSequence) has been given as the input, the output will also be a packed sequence. For the unpacked case, the directions can be separated using output.view(seq\_len, batch, num\_directions, hidden\_size), with forward and backward being direction 0 and 1 respectively.

Similarly, the directions can be separated in the packed case.

* **h\_n** of shape (num\_layers \* num\_directions, batch, hidden\_size): tensor containing the hidden state for t = seq\_len

Like output, the layers can be separated using h\_n.view(num\_layers, num\_directions, batch, hidden\_size).

INFO : GRU(

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

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

)

INFO : Number of trainable parameters=91.61M, where embeddings=34.30M, softmax=36.42M, core = 20.89M

## Experiments

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

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

|  |  |
| --- | --- |
| **Epoch** | **Training perplexity** |
| 1 | 17366.3 |
| 2 | 456.52 |
| 3 | 99.99 |
| 4 | 71.2 |
| 5 | 66.34 |
| 10 | 64.86 |
| 30 | 63.12 |
| 50 | 34.44 |
| 75 | 5.06 |
| 100 | 1.47 |

### **Experiment 1 – GRU on WT-2**

|  |  |  |
| --- | --- | --- |
| **Architecture** | **Input Signals** | **Hyperparameters** |
| **GRU** with 3 layers (1150 x3) | 1) The word embedding of the current global (d=300) | batch\_size=40 |
|  | TBPTT length=35 |
| Followed by linear2Globals **FF-NN** | learning rate=**0.5 \* 10^(-4)** |

|  |  |  |
| --- | --- | --- |
| *Epoch* | *Training perplexity* | *Validation perplexity* |
| 1 | 319.33 | 220.82 |
| 2 | 186.14 | 159.95 |
| 3 | 146.18 | 136.31 |
| 4 | 127.33 | 124.12 |
| 5 | 115.36 | 116.45 |
| 6 | 106.52 | 111.52 |
| 7 | 99.61 | 107.61 |
| 8 | 93.82 | 104.45 |
| 9 | 88.88 | 102.05 |
| 10 | 84.54 | 99.93 |
| 11 | 80.63 | 98.13 |
| 12 | 77.1 | 96.38 |
| 13 | 73.84 | 94.98 |
| 14 | 70.84 | 94.13 |
| 15 | 68.06 | 93.44 |
| 16 | 65.43 | 92.62 |
| 17 | 62.94 | 91.8 |
| 18 | 60.61 | 91.46 |
| 19 | 58.39 | 91.07 |
| 20 | 56.27 | 91.02 |
| 21 | 54.28 | **90.73** |
| 22 | 52.36 | 90.73 |
| 23 | 50.52 | 91.13 |
| 24 | 48.75 | 91.16 |
| 25 | 47.07 | 91.27 |

# Including the Graph Input

## GRU\_GAT

### Model

We use the new version of the GRU (torch.nn.GRU, 3 layers with 1150 units), plus the Graph Attention Network to obtain the node-state of the current global.

The GAT uses concatenated attention, with 4 heads, and we increase the dimensionality from the previous d/num\_heads (e.g. 300/4=75) to d/sqrt(num\_heads) (300/2=150) -> 600.

INFO : GRU\_GAT(

(gat\_globals): GATConv(300, 150, heads=4)

(gru): GRU(900, 1150, num\_layers=3)

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

)

INFO : Number of trainable parameters=93.85M, where embeddings=34.30M, softmax=36.42M, core = 23.13M

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

|  |  |  |
| --- | --- | --- |
| **Architecture** | **Input Signals** | **Hyperparameters** |
| **GRU** with 3 layers (1150 x3) | 1) The word embedding of the current global  2) Node-state of the current global  (d=300+150\*4 = 900) | batch\_size=4 |
| **GraphAttentionNetwork** with 4 heads @d=150, to get the current global’s node-state | TBPTT length=8 |
| Followed by linear2Globals **FF-NN** | learning rate=10^(-4) |

|  |  |
| --- | --- |
| **Epoch** | **Training perplexity** |
| 1 | 14143.46 |
| 2 | 370.29 |
| 3 | 101.61 |
| 4 | 72.28 |
| 5 | 66.54 |
| 10 | 64.91 |
| 30 | 43.89 |
| 50 | 9.14 |
| 75 | 1.67 |
| 100 | 1.23 |

### **Experiment 1 – GRU\_GAT on WT-2**

|  |  |  |
| --- | --- | --- |
| **Architecture** | **Input Signals** | **Hyperparameters** |
| **GRU** with 3 layers (1150 x3) | 1) The word embedding of the current global  2) Node-state of the current global  (d=300+150\*4 = 900) | batch\_size=40 |
| **GraphAttentionNetwork** with 4 heads @d=150, to get the current global’s node-state | TBPTT length=35 |
| Followed by linear2Globals **FF-NN** | learning rate=0.5\*10^(-4) |

|  |  |  |
| --- | --- | --- |
| *Epoch* | *Training perplexity* | *Validation perplexity* |
| 1 | 301.56 | 200.62 |
| 2 | 171.18 | 150.3 |
| 3 | 138.93 | 131.76 |
| 4 | 122.79 | 121.53 |
|  | Process killed – it was on a separate GPU, so probably it ran out of RAM |  |

|  |  |  |
| --- | --- | --- |
| *Epoch* | *Training perplexity* | *Validation perplexity* |
| 1 | 297.74 | 195.19 |
| 2 | 167.43 | 146.84 |
| 3 | 136.56 | 129.67 |
| 4 | 121.07 | 120.11 |
| 5 | 110.89 | 113.83 |
| 6 | 103.12 | 109.55 |
| 7 | 96.76 | 106.03 |
| 8 | 91.35 | 103.25 |
| 9 | 86.68 | 101.15 |
| 10 | 82.58 | 99.3 |
| 11 | 78.86 | 97.75 |
| 12 | 75.47 | 96.22 |
| 13 | 72.33 | 95.04 |
| 14 | 69.43 | 94.34 |
| 15 | 66.72 | 93.81 |
| 16 | 64.14 | 93.07 |
| 17 | 61.7 | 92.34 |
| 18 | 59.39 | 92.02 |
| 19 | 57.2 | 91.69 |
| 20 | 55.09 | 91.59 |
| 21 | 53.12 | **91.3** |
| 22 | 51.21 | 91.38 |
| 23 | 49.4 | 91.84 |
| 24 | 47.69 | 91.95 |
| 25 | 46.08 | 92.18 |

|  |  |  |
| --- | --- | --- |
| **Architecture** | **Input Signals** | **Hyperparameters** |
| **GRU** with 3 layers (1150 x3) | 1) The word embedding of the current global  2) Node-state of the current global  (d=300+75\*4 = 600) | batch\_size=40 |
| **GraphAttentionNetwork** with 4 heads @d=75, to get the current global’s node-state | TBPTT length=35 |
| Followed by linear2Globals **FF-NN** | learning rate=0.5\*10^(-4) |

GRU\_GAT(

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

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

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

)

Number of trainable parameters=92.73M

|  |  |  |
| --- | --- | --- |
| *Epoch* | *Training perplexity* | *Validation perplexity* |
| 1 | 302.09 | 195.15 |
| 2 | 167.16 | 146.48 |
| 3 | 136.41 | 129.4 |
| 4 | 121.05 | 119.9 |
| 5 | 110.79 | 113.58 |
| 6 | 102.97 (nearly identical to 4h’s 103.12) | 109.21 (nearly identical to 4h’s 109.55) |
| 7 | 96.63 | 105.65 |
| 8 | 91.25 (4h: 91.35) (base GRU: 93.82) | 102.79 (4h: 103.25) (GRU: 104.45) |
| 9 | 86.6 | 100.73 |
| 10 | 82.52 | 98.91 |
| 11 | 78.82 | 97.4 |
| 12 | 75.43 | 95.86 |
| 13 | 72.28 | 94.64 |
| 14 | 69.37 | 93.91 |
| 15 | 66.65 (4h: 66.72) (base GRU: 68.06) | 93.34 (4h: 93.81) (base GRU: 93.44) |
| 16 | 64.07 (base GRU: 65.43) | 92.62 (base GRU: 92.62) |
| 17 | 61.62 | 91.85 |
| 18 |  |  |
| 19 |  |  |
| 20 |  |  |
| 21 |  |  |
| 22 |  |  |
| 23 |  |  |
| 24 |  |  |
| 25 |  |  |

## WD-LSTM-GAT

### Model

We start from the LSTM with Weight-Drop on the hidden-to-hidden weights, that operated only on the Input Signal: current word embedding.

We add the Graph Attention Network to create the Input Signal: node-state of the current global, and examine the LM performance using the same hyperparameters.

INFO : WD\_LSTM\_GAT(

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

(lstm): LSTM(600, 1150, num\_layers=3)

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

)

INFO : Number of trainable parameters=100.04M where embeddings=34.30M, softmax=36.42M, core=29.32M

### Experiment 1 – WD-LSTM-GAT

|  |  |  |
| --- | --- | --- |
| **Architecture** | **Input Signals** | **Hyperparameters** |
| **LSTM** with 3 layers: 1150, 1150, 1150 | 1. The word embedding of the current global 2. Node-state of the current global   (d=600) | batch\_size=40 |
| **Weight-Dropping** mechanism with p=0.3 on the h2h LSTM weights | TBPTT length=35 |
| **GraphAttentionNetwork** to get the word’s node-state |  |
| Followed by linear2Globals **FF-NN** | learning rate=10^(**-5**) |

|  |  |  |
| --- | --- | --- |
| *Epoch* | *Training perplexity* | *Validation perplexity* |
| 1 | 458.52 | 302.77 |
| 2 | 311.01 | 300.69 |
| 3 | 309.51 | 300.63 |
| 4 | 309.15 | 300.46 |
| 5 | 308.92 | 300.23 |
| 6 | 308.38 | 297.3 |
| 7 | 268.64 | 236.43 |
| 8 | 231.99 | 218.58 |

Switching to full AWD-LSTM and GRU

# Including the whole AWD-LSTM implementation

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

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

## Input facilities

### Training loop

The trainining loop’s function:

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

### Corpus

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

How is a corpus produced?

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

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

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

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

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

Then, it adds to the dictionary, with:

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

The dictionary works as follows:

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

### Batch & Input data

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

…

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

In get\_batch():

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

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

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

We recall that corpus.train is:

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

What is done in batchify?

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

Example of batchify:

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

150, 160, 170, 180])

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

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

[ 20, 60, 100, 140],

[ 30, 70, 110, 150],

[ 40, 80, 120, 160]])

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

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

[ 20, 60, 100, 140],

[ 30, 70, 110, 150]])

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

[ 30, 70, 110, 150],

[ 40, 80, 120, 160]])

### Embeddings used by the model

When it is initialized, we have:

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

We should exclude init\_weights, since it executes:

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

In the forward() function:

…

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

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

…

raw\_output = emb

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

… etc

…

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

Regarding torch.nn.Embedding, from the docs:

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

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

…

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

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

, from the docs:

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

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

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

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

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

embeddings.weight.data = t2

Their initialization is similar:

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

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

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

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

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

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

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

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

We add:

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

The model is initialized as:

Loading cached dataset...

Applying weight drop of 0.5 to weight\_hh\_l0

Applying weight drop of 0.5 to weight\_hh\_l0

Applying weight drop of 0.5 to weight\_hh\_l0

[WeightDrop(

(module): LSTM(400, 1150)

), WeightDrop(

(module): LSTM(1150, 1150)

), WeightDrop(

(module): LSTM(1150, 400)

)]

Using []

### WeightDrop

The error:

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

follows.

We encounter a bug for PyTorch 1.4.0:

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

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

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

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

Now it appears to start correctly:

Loading cached dataset...

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

Using []

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

Model total parameters: 33556078

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

### Optimizer error

However, after a while we encounter an error:

Saving model (new best validation)

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

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

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

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

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

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

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

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

Saving model (new best validation)

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

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

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

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

Switching to ASGD

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

Traceback (most recent call last):

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

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

KeyError: 'ax'

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

### Weights can be flattened in memory

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

In the meantime, we obtain:

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

We proceed to:

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

Eventually,

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

### Torch 0.4.0 and 1.0.0

Let us then try to use the original PyTorch 0.4.0

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

Time to try and install PyTorch 1.0.0 with cuda 10.0…

Discussion:

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

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

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

### Out of Memory

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

we arrive at:

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

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

Saving Averaged!

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

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

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

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

Saving Averaged!

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

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

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

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

Saving Averaged!

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

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

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

then we get:

Traceback (most recent call last):

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

train()

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

loss.backward()

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

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

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

allow\_unreachable=True) # allow\_unreachable flag

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

**A profiling method**

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

*# prints currently alive Tensors and Variables*

**import** torch

**import** gc

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

**try**:

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

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

**except**:

**pass**

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

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

**Hypothesis: Variable-sized batches**

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

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

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

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

**Hypothesis: Variables not freed**

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

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

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

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

Does that sound reasonable in context of pytorch?”

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

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

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

”

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

## On the side: batching for GAT

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

In a forward() call,

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

*:func:`propagate`.*

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

note: operating on the mini-dataset.

**Time Analysis**:

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

Inside:

* **t1 - t0 = 0.00431** :

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

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

* t3 - t2 = 2e-05

### Unpacking the batchinput\_tensor

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

By executing

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

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

Then, we state:

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

squeeze()

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

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

On these 8 Tensors, we execute:

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

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

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

With NumPy:

It can be either:

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

or:

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

Iteration time in the Training loop=

With the previous handling of torch.Tensors:

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

Iteration time in the Training loop=

The improvement is inconsistent. It can be practically negligible.

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

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

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

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

The NumPy unpacking is marginally better.

### Full batching for GATConv

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

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

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

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

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

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

Inside the GAT, x is used as:

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

so I could pad it with zeros…

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

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

### Batches and DataParallel

Question: maybe I can reconcile batches and DataParallel…

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

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

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

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

# Meeting with IA, 12/05

(Reordered) list:

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

# Multi-Sense LM

## Structured prediction