[1: Meeting (IA, 24/01/20) 3](#_Toc36025417)

[2: Attempting Batching 3](#_Toc36025418)

[Using DataLoader 4](#_Toc36025419)

[Issues 4](#_Toc36025420)

[Speedup: 6](#_Toc36025421)

[3: Experiments and Hyperparameters - I 7](#_Toc36025422)

[3.1: Full overfit on mini-dataset 7](#_Toc36025423)

[3.1.1. 7](#_Toc36025424)

[3.1.2 8](#_Toc36025425)

[3.1.3 9](#_Toc36025426)

[Next steps 9](#_Toc36025427)

[4: Modifications 10](#_Toc36025428)

[Visualizing predictions – Round 1 10](#_Toc36025429)

[Experiment 10](#_Toc36025430)

[Samples 11](#_Toc36025431)

[Observations 14](#_Toc36025432)

[Phrases 14](#_Toc36025433)

[Punctuation 14](#_Toc36025434)

[Visualizing predictions – Round 2 15](#_Toc36025435)

[Experiment 15](#_Toc36025436)

[Samples 16](#_Toc36025437)

[Observations 18](#_Toc36025438)

[5: Alternative GNNs 19](#_Toc36025439)

[Current version: RGCN 19](#_Toc36025440)

[Observation: basis decomposition included by default 19](#_Toc36025441)

[Manual RGCN 20](#_Toc36025442)

[Split by relation into subgraphs 21](#_Toc36025443)

[Manual RGCN version 1.0 - Experiments 21](#_Toc36025444)

[Time analysis on MyRGCN 22](#_Toc36025445)

[Composing GCNs 22](#_Toc36025446)

[Trainable parameters 23](#_Toc36025447)

[Final experiment – all parameters explicitly included 25](#_Toc36025448)

[On the side: Experiment – Composite RGCN with Leaky ReLU 26](#_Toc36025449)

[6: Memory & Recurrence 27](#_Toc36025450)

[Gated GNNs 27](#_Toc36025451)

[Writing the GCNs+GRU 27](#_Toc36025452)

[Manual GRU on the representation 27](#_Toc36025453)

[Composite GatedGraphConv 29](#_Toc36025454)

[Experiment 29](#_Toc36025455)

[Settings and loss 29](#_Toc36025456)

[Conclusions 30](#_Toc36025457)

[7: Experiments on SemCor.xml – Round 1 31](#_Toc36025458)

[Parameters review 31](#_Toc36025459)

[Experiment 1 32](#_Toc36025460)

[Reviewing the model structure 32](#_Toc36025461)

[Experiment 2 34](#_Toc36025462)

[Modifications: Learning rate 34](#_Toc36025463)

[Input batching 35](#_Toc36025464)

[Time Analysis 37](#_Toc36025465)

[Experiment 3 38](#_Toc36025466)

[Meeting (PSK, 04/03/20) 39](#_Toc36025467)

[8: Loss and model modifications 40](#_Toc36025468)

[On the loss 40](#_Toc36025469)

[Mini-Experiment 40](#_Toc36025470)

[On the speed 41](#_Toc36025471)

[Experiment IV - SemCor 41](#_Toc36025472)

[Observations 42](#_Toc36025473)

[Preparation for WikiText-2 42](#_Toc36025474)

[Vocabulary and graph statistics 42](#_Toc36025475)

[Issue: Batches & TBPTT 43](#_Toc36025476)

[Experiment V – WikiText-2 44](#_Toc36025477)

[Mini Experiment - Overfit 44](#_Toc36025478)

[Variants 44](#_Toc36025479)

[9: Baseline comparisons and reconstructing the model 46](#_Toc36025480)

[RNN 46](#_Toc36025481)

[Experiment: RNN on WikiText-2 46](#_Toc36025482)

[RNN, 2 layers 47](#_Toc36025483)

[GRU + RNN 49](#_Toc36025484)

[Simplified GRU 49](#_Toc36025485)

[Graph GRU 50](#_Toc36025486)

[Full GRU 50](#_Toc36025487)

[10: Including the graph input 53](#_Toc36025488)

[Reflections on previous attempts 53](#_Toc36025489)

[Graph Attention Networks on different kinds of nodes 54](#_Toc36025490)

[Recap of GATs 54](#_Toc36025491)

[On senses 55](#_Toc36025492)

[RGAT: Relational Graph-Attention Networks 55](#_Toc36025493)

[GRU + GATs with different node types 56](#_Toc36025494)

[Architecture 56](#_Toc36025495)

[Input from the graph 56](#_Toc36025496)

[11: Adding the Graph Networks in practice 58](#_Toc36025497)

[GATs with different node types 58](#_Toc36025498)

[Experiment 1 58](#_Toc36025499)

[Examining possible modifications 59](#_Toc36025500)

[Graph\_area hops 59](#_Toc36025501)

[Senses and input signal 59](#_Toc36025502)

[GATs – basic 60](#_Toc36025503)

[Experiment 1 – GAT, 1-hop 60](#_Toc36025504)

[Experiment 2 – GAT, 2-hops 61](#_Toc36025505)

[Experiment 3 – GAT, input signals 62](#_Toc36025506)

[12: Batch normalization 64](#_Toc36025507)

[Introduction 64](#_Toc36025508)

[Method and observations 64](#_Toc36025509)

[Necessity of Batch Renormalization 65](#_Toc36025510)

# 1: Meeting (IA, 24/01/20)

ToDo list:

* complete batching
* lower the learning rate from 0.01 to 10^-3 or -4
* do not plot the loss for each batch, but instead the average training loss over the epoch
* make another overfit test. It should go down all the way to 0
* Include <UNK> in the processing. Initialized as the average of all globals at start
* RRGCN, recurrent RGCN, to add the RNN logic to the Language model task

# 2: Attempting Batching

In the current version, I do not have real, parallel batching.

I am using a for cycle on the elements of the batch:

**for** i **in** range(len(input\_indices\_lts) - 1):  
 (x, edge\_index, edge\_type) = batch\_rgcn\_input\_ls[i]  
  
 predicted\_globals, predicted\_senses = model(x, edge\_index, edge\_type)

…

Proposal: send batch\_rgcn\_input as the input to the model’s forward().

It depends on the input-to-the-forward-call for each node.

As of now, we collect it in a list for all nodes in the batch as follows:

forward\_input\_ls.append((area\_x, edge\_index, edge\_type))

**Problem**: forward\_input\_ls is a tuple of 3 tensors.

The shapes of (area\_x, edge\_index, edge\_type) are, respectively:

torch.Size([32, 300])  
torch.Size([2, 2048])  
torch.Size([2048])

As they are, they can not be stacked.

They could be padded with -1s, and stacked side-by-side:

torch.Size([32, 300]) 🡪 torch.Size([32, 300])  
torch.Size([2, 2048]) 🡪 torch.Size([32, 2048]) 🡪 torch.Size([32, 4396])  
torch.Size([2048]) 🡪 torch.Size([32, 2048])

Then, each element of the batch can be stacked vertically, thus obtaining a batch-dimension that can be used for parallel processing.

With batch\_size=8, obtain: torch.Size([8, 32, 4396])

## Using DataLoader

**Problem**: I am not able to specify that [8, 32, 4396] contains the 8 elements of a batch.

Considering: I have already the graphArea\_matrix, precomputed, that contains the graph-input for any word/node.  
I also have the training dataset.

**Choice**: construct a PyTorch DataLoader, that can take in 1 sample, or possibly more if I specify a batch size > 1.

This will also make the padding mechanism redundant, simplifying part of the code..

This involves the creation of a TextDataser(Dataset) class, that returns the next (X, y) item:  
the 3 input features (x, edge\_index, edge\_type) + the labels of the next token.

### Issues

{

Issues: Error operating on the H5. Possibility of throwing a wrong StopIteration exception.

The vocabulary of globals has the columns ‘word’ and ‘frequency’…

Hypothesis: the error on the vocabulary\_of\_globals.h5 is due to the parallel access of the DataLoader’s num\_workers > 1.

Information that supports the hypothesis: parallel hdf5 is a separate implementation:  
“Starting with version 2.2.0, h5py includes support for Parallel HDF5.  
Parallel HDF5 is a configuration of the HDF5 library which lets you share open files across multiple parallel processes. It uses the MPI (Message Passing Interface) standard for interprocess communication. .. This is accomplished through the mpi4py Python package ...”

Opinion that opposes the hypothesis from StackOverflow:  
“Parallel reads are fine with h5py, no need for the MPI version. But why do you expect a speed-up here? Your job is almost entirely I/O bound, not CPU bound. Parallel processes are not gonna help because the bottleneck is your hard disk, not the CPU. It wouldn't surprise me if parallelization in this case even slowed down the whole reading operation. Other opinions?”

Conclusion: if it is a way to avoid that HDF5 bug, I use 1 worker.}

{

The next issue is due to not being able to pass next\_token\_tuple properly, as an argument to the Dataset’s \_\_getitem\_\_(), when iterating over the DataLoader.

Maybe :

1. I can get the Dataset & DataLoader to return only the input, and just add manually the label. After all, the next\_token\_tuple does not need any processing
2. Modify the TextDataset, to keep a variable for the next token, and return it without any need for input.

}

{

Issue: I have an error because I am still throwing the Utils.MustSkipUNK\_Exception.

However, we added self-loops to the UNK nodes, so they are not disconnected and without edges (that was a cause of error) anymore.

I should review how they are handled, and initialized.

This covers one of the points-of-order from the last meeting, that was:   
“*•Include <UNK> in the processing. Initialized as the average of all globals at start*”

Error: “Raising Utils.MustSkipUNK\_Exception with word= Fulton County Grand Jury”  
I must redirect these cases to the <UNK> token

<unk> is, in fact, already present in the vocabulary\_of\_globals.h5}

{

When sending a batch\_dimension > 1 :

RuntimeError: invalid argument 0: Sizes of tensors must match except in dimension 0. Got 49 and 57 in dimension 2

Problem: the number of edges must be aligned. Considering 2 elements, we have:

area\_x.shape=torch.Size([32, 300])  
edge\_index.shape=torch.Size([2, 49])  
edge\_type.shape=torch.Size([49])

area\_x.shape=torch.Size([32, 300])  
edge\_index.shape=torch.Size([2, 57])  
edge\_type.shape=torch.Size([57])

Possible solutions:

1. Pad the vectors to the same dimension in Dataset, and select relevant elements in the forward()
2. Implement manually the def collate\_fn(data) function passed to the Dataloader. I could also use it as the point where to add padding.

}

Observation:

Apparently, I still need to implement the padding with -1 to a common size, in order to perform a stacking along the batch dimension (0).

What to do about the label tuple, a.k.a. the next token’s tuple e.g (238, 16015) ?

Major problem, that may undermine the concept of using batching with RGCN:  
The core call is: tF.relu(self.conv1(x, edge\_index, edge\_type))  
However, we are not able to send standardised edge\_index, because some nodes may have only few edges. Or few neighbours, in which case the number of rows in x will be << grapharea\_size. **No stacking --> no batching**

Error, in select\_valid\_features:  
return torch.stack(valid\_elems\_all)  
RuntimeError: invalid argument 0: Sizes of tensors must match except in dimension 0. Got 49 and 57 in dimension 1

### Speedup:

With the previous version, a batch size of 32 has an iteration time of 0.73/0.90s

Now, even if we do not have 100% proper batching because the RGCN layer call is not parallelized, thanks to the Dataset + DataLoader, and to the parallel input creation and loss computation, a batch size of 8 has an iteration time of 0.012/0.14s, and a batch of 32 has \_\_\_.

(I consider this an absolute win)

# 3: Experiments and Hyperparameters - I

## 3.1: Full overfit on mini-dataset

### 3.1.1.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| batch size | 8 |  | training tokens | 496 |
| graph\_area | 32 |  | epochs | 100 |
| learning rate | 0.001 |  | final global step | 6200 |
|  |  |  | token-steps | 49600 |



Final epoch nll\_loss= 4.158

However, my objective is to have a *full* overfit, with the loss value approaching 0.

I will try to:

* Increase the number of epochs
* Reduce the number of samples

### 3.1.2

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| batch size | 8 |  | **training tokens** | **176** |
| graph\_area | 32 |  | **epochs** | **300** |
| learning rate | 0.001 |  | final global step | 6600 |
|  |  |  | token-steps | 52800 |



Training, epoch nll\_loss= 2.21

### 3.1.3

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| batch size | 4 |  | **training tokens** | **128** |
| graph\_area | 32 |  | **epochs** | **1000** |
| **learning rate** | **0.001** |  | final global step | 32000 |
|  |  |  | token-steps | 26400 |



Training, epoch nll\_loss= 1.2242

Increasing the learning rate (e.g. 0.005) does not bring any improvement, only a bounce effect.  
The minimum training loss I can currently achieve, even trying to overfit on a very small dataset, is ~1.2

Even using grapharea\_size = 64 does not bring the nll\_loss < 1.

### Next steps

2 directions / modifications are needed:

1. Check the solution-tokens and the predicted globals&senses. The model may not be able to read/predict something, what is it?
2. Extend the vocabulary of globals. Check the current status of the vocabulary of senses as well.

# 4: Modifications

## Visualizing predictions – Round 1

### Experiment

Proceeding in reverse from the numerical indices, I can now log the predicted senses and globals.

I will print the predictions in the last epoch, after the model has stabilized, for instance in

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| batch size | 8 |  | training tokens | 128 |
| graph\_area | 32 |  | epochs | 100 |
| learning rate | 0.003 |  | final global step |  |
|  |  |  | token-steps |  |



Final nll\_loss = 2.20181

### Samples

Sentence, from the start of semcor.xml:

“The Fulton\_County\_Grand\_Jury said Friday an investigation of Atlanta 's recent primary\_election produced &quot no evidence &quot that any irregularities took\_place”

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Global** | **Probability** |  | **Sense** | **Probability** |
| said | Solution |  | state.v.01 | solution |
| , | 38.93% |  | state.v.01 | 95.58% |
| said | 38.03% |  | produce.v.04 | 2.17% |
| <unk> | 11.38% |  | far.r.02 | 0.7% |
| and | 3.21% |  | person.n.01 | 0.46% |
| which | 3.2% |  | mission.n.03 | 0.38% |

|  |  |
| --- | --- |
| Sample | Comment |
| **Label: the next global is: Friday**  INFO : The top- 5 predicted globals are:  **INFO : Word: , ; probability = 41.48%**  **INFO : Word: in ; probability = 33.9%**  **INFO : Word: Friday ; probability = 23.78%**  INFO : Word: . ; probability = 0.37%  INFO : Word: was ; probability = 0.26% | The comma ends up being the first prediction.  Occurrences in the text=3: said Friday said in said , |
| **Label: the next global is: an**  INFO : The top- 5 predicted globals are:  **INFO : Word: an ; probability = 100.0%** | v |
| **Label: the next global is: investigation**  **INFO : Label: the next sense is: probe.n.01**  INFO : The top- 5 predicted globals are:  **INFO : Word: investigation ; probability = 97.0%**  INFO : Word: of ; probability = 2.18%  INFO : Word: by ; probability = 0.34%  INFO : Word: was ; probability = 0.19%  INFO : Word: <unk> ; probability = 0.16%  INFO : The top- 5 predicted senses are:  **INFO : Sense: probe.n.01 ; probability = 98.99%** | v |
| **Label: the next global is: of**  INFO : The top- 5 predicted globals are:  **INFO : Word: <unk> ; probability = 32.29%**  **INFO : Word: of ; probability = 31.81%**  INFO : Word: the ; probability = 11.58%  INFO : Word: said ; probability = 7.53%  INFO : Word: in ; probability = 5.38% | <unk> versus of.  ‘of’ should have been the only reasonable alternative.  Occurrences in the text=1:  ‘investigation of‘  Issue of the globals-prediction system. By the way, what is the node ‘of’ connected to? |
| **Label: the next global is: Atlanta**  INFO : The top- 5 predicted globals are:  **INFO : Word: the ; probability = 35.87%**  INFO : Word: possible ; probability = 17.19%  INFO : Word: such ; probability = 16.8%  INFO : Word: voters ; probability = 14.04%  **INFO : Word: Atlanta ; probability = 12.82%** | We observe that Atlanta is too low after 100 epochs.  Occurrences in the text=5: of Atlanta [Atlanta=1] of the (x2) [the=15] of possible [possible=1] of such [such=1] of voters [voters=1]  All alternatives are represented. It is interesting to notice that words that are more common in language are more prominent even when their frequency in the training text is identical  (possible > such > voters > Atlanta) |
| **Label: the next global is: s**  INFO : The top- 5 predicted globals are:  **INFO : Word: s ; probability = 99.93%** | v |
| **Label: the next global is: recent**  **INFO : Label: the next sense is: late.s.03**  INFO : The top- 5 predicted globals are:  **INFO : Word: recent ; probability = 92.28%**  INFO : Word: . ; probability = 2.38%  INFO : Word: , ; probability = 1.31%  INFO : Word: term ; probability = 1.12%  INFO : Word: <unk> ; probability = 0.94%  INFO : The top- 5 predicted senses are:  **INFO : Sense: late.s.03 ; probability = 96.5%**  INFO : Sense: term.n.02 ; probability = 1.79% | v |
| **Label: the next global is: <unk>**  **INFO : Label: the next sense is: primary.n.01**  INFO : The top- 5 predicted globals are:  **INFO : Word: <unk> ; probability = 99.99%**  INFO : Word: reports ; probability = 0.01%  INFO : Word: in ; probability = 0.0%  INFO : Word: a ; probability = 0.0%  INFO : Word: September ; probability = 0.0%  INFO : The top- 5 predicted senses are:  **INFO : Sense: primary.n.01 ; probability = 99.98%** | primary\_election is evaluated as a <unk>  ISSUE: there is a discrepancy between the phrases in the sense-labeled corpus and a vocabulary of globals.  Maybe it is necessary to try out the vocabulary from SLC. Or to build phrases in the vocabulary from WikiText |
| **Label: the next global is: produced**  **INFO : Label: the next sense is: produce.v.04**  INFO : The top- 5 predicted globals are:  **INFO : Word: which ; probability = 44.2%**  **INFO : Word: produced ; probability = 40.69%**  INFO : Word: , ; probability = 5.12%  INFO : Word: <unk> ; probability = 2.47%  INFO : Word: said ; probability = 2.27%  INFO : The top- 5 predicted senses are:  **INFO : Sense: produce.v.04 ; probability = 94.99%**  INFO : Sense: state.v.01 ; probability = 4.2% | ‘produced’ follows a <unk>.  We have no other information, other than the likelihood of a word to follow <unk> |
| **Label: the next global is: <unk>**  INFO : The top- 5 predicted globals are:  **INFO : Word: <unk> ; probability = 79.9%**  INFO : Word: was ; probability = 5.18%  INFO : Word: the ; probability = 2.58%  INFO : Word: of ; probability = 2.48%  INFO : Word: by ; probability = 2.1% | This is the ‘&quot’ HTML symbol.  It should be turned into “  Then, it will be covered by our decision regarding punctuation. |
| -**Label: the next global is: no**  INFO : The top- 5 predicted globals are:  **INFO : Word: <unk> ; probability = 21.16%**  INFO : Word: The ; probability = 10.93%  INFO : Word: that ; probability = 9.87%  INFO : Word: primary ; probability = 6.34%  INFO : Word: in ; probability = 6.29%  INFO : | I am, apparently, unable to predict this global word after a <unk> |
| **Label: the next global is: evidence**  **INFO : Label: the next sense is: evidence.n.01**  INFO : The top- 5 predicted globals are:  **INFO : Word: the ; probability = 34.81%**  INFO : Word: <unk> ; probability = 23.1%  INFO : Word: handful ; probability = 15.71%  INFO : Word: which ; probability = 13.39%  **INFO : Word: evidence ; probability = 11.14%**  INFO : The top- 5 predicted senses are:  **INFO : Sense: handful.n.01 ; probability = 57.37%**  **INFO : Sense: evidence.n.01 ; probability = 40.85%**  INFO : Sense: potential.a.01 ; probability = 0.75%  INFO : Sense: such.s.01 ; probability = 0.75%  INFO : Sense: conduct.v.01 ; probability = 0.09% | In the text, we have only one “no”, followed by “evidence”.  Why does the global prediction fail?  Rare event here: the sense-prediction system fails. |
| … |  |
| **Label: the next global is: praise**  **INFO : Label: the next sense is: praise.n.01**  INFO : The top- 5 predicted globals are:  **INFO : Word: election ; probability = 24.7%**  **INFO : Word: <unk> ; probability = 23.24%**  INFO : Word: widespread ; probability = 9.07%  INFO : Word: number ; probability = 8.91%  INFO : Word: size ; probability = 8.77%  INFO : The top- 5 predicted senses are:  **INFO : Sense: election.n.01 ; probability = 25.94%**  INFO : Sense: size.n.01 ; probability = 12.53%  INFO : Sense: manner.n.01 ; probability = 11.47%  INFO : Sense: location.n.01 ; probability = 11.43%  **INFO : Sense: praise.n.01 ; probability = 11.37%** | ‘praise’ follows ‘the’.  Since ‘the’ appears many times, we will have a widespread probability of having different next words.  ‘the election’ is found repeatedly, and this is mirrored in the prediction. |
| **Label: the next global is: and**  INFO : Label: the next sense is: None  INFO : tensor([ 41, 25, 65, ..., 2180, 13499, 7587], device='cuda:0')  INFO : The top- 5 predicted globals are:  **INFO : Word: and ; probability = 99.77%**  INFO : Word: . ; probability = 0.14%  INFO : Word: of ; probability = 0.03%  INFO : Word: the ; probability = 0.03%  INFO : Word: October ; probability = 0.02% | v |
| **Label: the next global is: thanks**  **INFO : Label: the next sense is: thanks.n.01**  INFO : The top- 5 predicted globals are:  **INFO : Word: the ; probability = 64.9%**  **INFO : Word: thanks ; probability = 34.91%**  INFO : Word: was ; probability = 0.14%  INFO : Word: and ; probability = 0.03%  INFO : Word: charge ; probability = 0.02%  INFO : The top- 5 predicted senses are:  **INFO : Sense: thanks.n.01 ; probability = 99.96%**  INFO : Sense: mission.n.03 ; probability = 0.03%  INFO : Sense: probe.n.01 ; probability = 0.01%  INFO : Sense: potential.a.01 ; probability = 0.0%  INFO : Sense: such.s.01 ; probability = 0.0% | ‘and’ has n.occurrences=2.  ‘and thanks’, ‘and the’ |
| … |  |
| **Label: the next global is: October**  INFO : Label: the next sense is: None  INFO : The top- 5 predicted globals are:  **INFO : Word: October ; probability = 91.28%**  INFO : Word: and ; probability = 2.84%  INFO : Word: , ; probability = 1.09%  INFO : Word: . ; probability = 1.07%  INFO : Word: to ; probability = 1.0% | v |
| **Label: the next global is: term**  **INFO : Label: the next sense is: term.n.02**  INFO : The top- 5 predicted globals are:  **INFO : Word: term ; probability = 88.13%**  INFO : Word: which ; probability = 6.36%  INFO : Word: recent ; probability = 1.78%  INFO : Word: to ; probability = 1.62%  INFO : Word: October ; probability = 0.77%  INFO : The top- 5 predicted senses are:  **INFO : Sense: term.n.02 ; probability = 97.7%**  INFO : Sense: late.s.03 ; probability = 1.19%  INFO : Sense: produce.v.04 ; probability = 0.79%  INFO : Sense: overall.s.02 ; probability = 0.27%  INFO : Sense: state.v.01 ; probability = 0.04% | v |

### Observations

Since we have no memory for the context, we end up computing the probability of obtaining the next word W(t+1)=*b* after W(t)=*a*.

This probability is influenced by:

1. The number of occurrences of “… *a* *b* …” in the text, like a n-gram language model.
2. The frequency of *a* and *b* in the text
3. The centrality/”mainstream nature” of the possible completions in *b*(e.g. possible > such > voters > Atlanta)

Issues that emerge from this experiment:

* **Context**: we should be able to use as a base more than 1 word. Recurrence / memory mechanism (e.g. RNNs) needed
* **Punctuation**: the ‘&quot’ HTML symbol is seen as a <unk>. It should be turned into “. Then, it will be covered by our decision regarding punctuation
* **Phrases**: there is a discrepancy between the phrases in the sense-labeled corpus and a vocabulary of globals. Maybe it is necessary to try out the vocabulary from SLC. Or to build phrases in the vocabulary from WikiText

## Phrases

We use the vocabulary from the Sense-LabeledCorpus itself instead of the one from WikiText-2.

Now: primary\_election is still not present, most probably because of the low frequency. However,we find took\_place, pointed\_out and others.

## Punctuation

Q: If I take the vocabulary from the SLC, does it mean that the HTML-encoded elements from punctuation (e.g. &quot) are already present?

If so, then all that is needed is to turn them into symbols when printing the predictions.

“ Atlanta’s “

The vocabulary from SLC reads [Atlanta], [‘s]

The RGCN has been reading [Atlanta], [s]

----> I should keep the apostrophe

Examining: MyRGCN.train() -> DataLoading.TextDataset ->

self.generator = SLC.read\_split(self.split\_name) -> SLC. dataset\_generator(xml\_fpath)

Output from the generator on the training dataset:

Out[16]: {'surface\_form': 'produced', 'lemma': 'produce', 'pos': 'VBD', 'wn16\_key': 'produce%2:39:01::', 'wn30\_key': 'produce%2:39:01::'}

gen.\_\_next\_\_()

Out[17]: {'surface\_form': '"', 'pos': "''"}

The SLC-vocabulary does not have anything for [&quot], nor for [“]

It is the vocabulary’s fault.

Commas and punctuation signs are present in the Sense-Labeled Corpus as separate tokens.

Therefore, it makes sense to keep them in the vocabulary, and thus in the graph, as globals.

I have no way to connect them to anything else, though, apart from the self-loop.

They will be placed and then trained based on reading the text corpus – no input from dictionary sources.

With min\_count=10 (from here on it will be changed to min\_count=5), we have a relatively limited number of globals. In the graph:

X\_definitions.shape=torch.Size([13046, 300])

X\_examples.shape=torch.Size([16200, 300])

X\_senses.shape=torch.Size([13046, 300])

X\_globals.shape=torch.Size([5528, 300])

And the graph-

dataObject will be:

Data(edge\_index=[2, 54080], edge\_type=[54080], node\_types=[47820],  
 num\_relations=[1], x=[47820, 300])

## Visualizing predictions – Round 2

### Experiment

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| batch size | 8 |  | training tokens | 128 |
| graph\_area | 32 |  | epochs | 100 |
| learning rate | 0.003 |  | final global step | 1600 |



Training, final nll\_loss= 2.44262

(Since we include the punctuation, we have more symbols to choose from and we need more epochs to overfit)

### Samples

|  |  |
| --- | --- |
| Sample | Comment |
| **Label: the next global is: said**  **INFO : The top- 5 predicted globals are:**  **INFO : Word: , ; probability = 33.88%**  **INFO : Word: said ; probability = 31.93%**  INFO : Word: further ; probability = 9.0%  INFO : Word: <unk> ; probability = 8.13%  INFO : Word: had ; probability = 6.35%  **INFO : Label: the next sense is: state.v.01**  **INFO : The top- 5 predicted senses are:**  **INFO : Sense: state.v.01 ; probability = 82.11%**  INFO : Sense: far.r.02 ; probability = 13.1%  INFO : Sense: person.n.01 ; probability = 3.76%  INFO : Sense: produce.v.04 ; probability = 0.72%  INFO : Sense: late.s.03 ; probability = 0.13% | The comma is the first global predicted. It happened already in the previous version |
| **Label: the next global is: Friday**  **INFO : The top- 5 predicted globals are:**  **INFO : Word: , ; probability = 38.21%**  **INFO : Word: in ; probability = 34.14%**  **INFO : Word: Friday ; probability = 27.1%**  INFO : Word: recent ; probability = 0.14%  INFO : Word: was ; probability = 0.13% | Same as previous version |
| **Label: the next global is: an**  INFO : The top- 5 predicted globals are:  **INFO : Word: an ; probability = 98.09%** | v  Same |
| **Label: the next global is: investigation**  **INFO : Label: the next sense is: None**  **INFO : The top- 5 predicted globals are:**  **INFO : Word: investigation ; probability = 97.76%**  INFO : Word: an ; probability = 0.94%  INFO : Word: jury ; probability = 0.64%  INFO : Word: <unk> ; probability = 0.24%  INFO : Word: of ; probability = 0.22% | The next global is correct, as it was before.  *However*, why are we not recognizing ‘probe.s.01’ as the sense solution, as we did before? |
| **Label: the next global is: of**  **INFO : The top- 5 predicted globals are:**  **INFO : Word: September ; probability = 28.37%**  **INFO : Word: of ; probability = 24.26%**  INFO : Word: jury ; probability = 23.94%  INFO : Word: <unk> ; probability = 22.95%  INFO : Word: investigation ; probability = 0.32% | previously, we predicted the globals: <unk>, of, the, said, in |
| **Label: the next global is: Atlanta**  **INFO : Label: the next sense is: None**  **INFO : The top- 5 predicted globals are:**  INFO : Word: the ; probability = 30.69%  INFO : Word: possible ; probability = 15.35%  INFO : Word: such ; probability = 14.83%  INFO : Word: voters ; probability = 13.66%  INFO : Word: s ; probability = 12.69% | Worse than previously, when it was [the, possible, such, voters, Atlanta]. Probably due to overfitting less? |

|  |  |
| --- | --- |
| **Label: the next global is: s**  **INFO : Label: the next sense is: None**  **INFO : The top- 5 predicted globals are:**  INFO : Word: the ; probability = 30.69%  INFO : Word: possible ; probability = 15.35%  INFO : Word: such ; probability = 14.83%  INFO : Word: voters ; probability = 13.66%  **INFO : Word: s ; probability = 12.69%** | We do not see the apostrophe. Is it not printed, or is it just ignored?  Originally, it was: INFO : Word: s ; probability = 99.93% |
| **Label: the next global is: recent**  **INFO : Label: the next sense is: late.s.03**  INFO : The top- 5 predicted globals are:  **INFO : Word: recent ; probability = 61.05%**  INFO : Word: of this ; probability = 16.86%  INFO : Word: " ; probability = 7.7%  INFO : Word: the ; probability = 2.89%  INFO : Word: of ; probability = 2.27%  **INFO : The top- 5 predicted senses are:**  INFO : Sense: late.s.03 ; probability = 92.13%  INFO : Sense: jury.n.01 ; probability = 2.17%  INFO : Sense: end.n.02 ; probability = 1.37%  INFO : Sense: produce.v.04 ; probability = 1.3%  INFO : Sense: primary.n.01 ; probability = 0.82% | v, as before |
| **Label: the next global is: <unk>**  **INFO : Label: the next sense is: primary.n.01**  **INFO : The top- 5 predicted globals are:**  **INFO : Word: <unk> ; probability = 99.23%**  INFO : Word: " ; probability = 0.28%  INFO : Word: was ; probability = 0.25%  INFO : Word: of this ; probability = 0.09%  INFO : Word: in ; probability = 0.06%  **INFO : The top- 5 predicted senses are:**  **INFO : Sense: primary.n.01 ; probability = 99.73%**  INFO : Sense: happen.v.01 ; probability = 0.14%  INFO : Sense: jury.n.01 ; probability = 0.07%  INFO : Sense: end.n.02 ; probability = 0.03%  INFO : Sense: conduct.v.01 ; probability = 0.01% | As before   primary\_election is evaluated as a <unk> due to low frequency  The primary.n.01 sense is still correct |
| **Label: the next global is: produced**  **INFO : Label: the next sense is: produce.v.04**  **INFO : The top- 5 predicted globals are:**  **INFO : Word: which ; probability = 30.99%**  **INFO : Word: produced ; probability = 28.51%**  INFO : Word: " ; probability = 10.24%  INFO : Word: had ; probability = 6.73%  INFO : Word: <unk> ; probability = 4.33%  **INFO : The top- 5 predicted senses are:**  **INFO : Sense: produce.v.04 ; probability = 65.75%**  INFO : Sense: person.n.01 ; probability = 21.68%  INFO : Sense: far.r.02 ; probability = 8.04%  INFO : Sense: state.v.01 ; probability = 2.76%  INFO : Sense: late.s.03 ; probability = 0.94% | as before: ‘which’ and ‘produced’ are nearly tied as the predicted global. The sense is ok |
| **Label: the next global is: "**  **INFO : Label: the next sense is: None**  **INFO : The top- 5 predicted globals are:**  **INFO : Word: " ; probability = 25.56%**  INFO : Word: of ; probability = 19.37%  INFO : Word: the ; probability = 13.3%  INFO : Word: was ; probability = 7.77%  INFO : Word: end ; probability = 7.51% | New, and correct |
| **Label: the next global is: no**  **INFO : Label: the next sense is: None**  **INFO : The top- 5 predicted globals are:**  INFO : Word: , ; probability = 11.24%  INFO : Word: that ; probability = 10.73%  INFO : Word: in ; probability = 9.91%  INFO : Word: Only ; probability = 9.65%  INFO : Word: for ; probability = 9.6% | As before: I am, apparently, unable to predict this global word after a <unk> |

### Observations

**Why we do not have ‘probe.s.01’ as the sense solution, as we did before?**

Using the SLCvocabulary, the indices\_table.sql file is of size 442 KB instead of 647 KBs.

We retrieve the senses from WordNet, using as a base the vocabulary of globals…

Hypothesis:

Fewer globals -> fewer senses

|  |  |  |
| --- | --- | --- |
| WT-2 Vocab, min\_f=10 | SLC Vocab, min\_f=10 | SLC Vocab, min\_f=5 |
| X\_definitions.shape=  torch.Size([19008, 300])  X\_examples.shape=  torch.Size([20191, 300])  X\_senses.shape=  torch.Size([19008, 300])  X\_globals.shape=  torch.Size([13675, 300]) | X\_definitions.shape=  torch.Size([13046, 300])  X\_examples.shape=  torch.Size([16200, 300])  X\_senses.shape=  torch.Size([13046, 300])  X\_globals.shape=  torch.Size([5528, 300]) | X\_definitions.shape=  torch.Size([17843, 300])  X\_examples.shape=  torch.Size([20088, 300])  X\_senses.shape=  torch.Size([17843, 300])  X\_globals.shape  =torch.Size([9858, 300]) |
| Tot.nodes = 71882 | Tot.nodes =47800 | Tot.nodes = 65632 |

We assume that this solves the problem of not containing “probe”, which prevents us from including ‘probe.s.01’. A later check will follow.

No, we still do not have probe, but we have added other senses. During the development process, I keep it as it is.

# 5: Alternative GNNs

## Current version: RGCN

We recall the rules of the current version, that handles multiple types of edges but has no recurrence:

The update rule to the node state for basic GCNs is:

The update-&-propagation model for the R-GCN is the following:

“ where Nir denotes the set of neighbor indices of node i under relation r.

cir can be =|Nir|

Different from regular GCNs, we introduce relation-specific transformations, i.e. depending on the type and direction of an edge.

To ensure that the representation of a node at layer (l + 1) can also be informed by the corresponding representation at layer (l), we add a single self-connection () of a special relation type 0 to each node in the data.

A neural network layer update consists of evaluating in parallel for every node in the graph.”

### Observation: basis decomposition included by default

The original aim of RGCNs was to deal with Knowledge Bases, that can easily have thousands of relations/edge types.

With this aim in mind, they are meant to have block-diagonal and basis decompositions, to decrease the number of parameters.

The pre-implemented version of RGCN in PyTorch-Geometric includes, mandatorily, a basis-decomposition:

1. each weights matrix for a relation r on layer l, , is defined as follows:

as a linear combination of a lower number of common Bases *Vb*, where only the coefficients depend on *r*.

I set #*Vb*-s = 5, since I do not need this feature.

**Idea:**

I could implement the RGCN formula myself, with the aim of streamlining it, avoiding the basis decomposition, and preparing the ground for the introduction of recurrence.

**Observations on recurrence:**

Regarding recurrence and the forward() call:

I need to include in the input matrix **x** all the nodes that are involved in the batch.

Batch = text window = BPTT window.

**x**: must contain all the area nodes for the words in the batch.

## Manual RGCN

The batchinput\_ls contains elements (x, edge\_index, edge\_type).

When using the pre-made implementation, we have:

rgcn\_conv = self.conv1(x, edge\_index, edge\_type)

x\_Lplus1 = tfunc.relu(rgcn\_conv)

The weight matrices are . We must have:

* 1 W per layer. We need only 1 if we keep the previous architecture, of:  
  input > relu(rgcn\_conv) > representation > 2 linear FF-NNs > (logits\_global, logits\_sense)
* 1 W per edge type
* 1 W for the direct connection from the previous level

It is necessary to split the edge\_index, edge\_type into |R|=5 (def, ex, sc, syn, ant) adjacency matrices.

Then, and +

**Observation**:

The way the product is written in the formulas is actually misleading with respect to the order of factors and the dimensions.

i.e. in

we have that must have the same dimensions as , since it is added to it (before the ReLu that does not change the dimensionality)

W : (d x d) , and is (d x 1) --> the only way to have a result of (d x 1) is to execute the product using as a **column**. We are effectively summing up the values of , depending on the weights.

This is corroborated by the github implementations of GCNs.

Currently, my rgcn\_convolution(H, Ar\_ls, W\_all) uses a for-cycle on the *Ar* matrices which store the neighbourhoods for a given edge type *r*.

For each relation, we execute a standard GCN convolution, divided by a normalization constant . Then, they are summed up and we add the direct connection from the node’s previous layer.

### Split by relation into subgraphs

It is also necessary to implement an instrument that splits the input of the pre-made RGCN call, (x, edge\_index, edge\_type), into K=5 different adjacency matrices.

This can be used both with our manual RGCN and with tools from Pytorch-geometric (e.g library GCNs > sum up the outputs).

x.shape = (32, 300). We have a graph\_area\_size of 32 nodes around the current selected one. d = 300.

edge\_index.shape = (2,45). In general, (2, num\_edges).

It is made of (vector\_of\_sources, vector\_of\_destinations). If we transpose it, we will have the classic form [(source1,dest1),…, ]:

tensor([[ 1, 0],

[20, 7],

[11, 4],

[15, 5], …)

edge\_type.shape = (45). In general, (num\_edges).

e.g.: tensor([0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 2, 2, 2, 2, 2, 2, 2, 2,

2, 2, 2, 2, 3, 3 …

The vector specifying the type of edges. We should use it to perform the split.

Note: I am not using any bias term currently (set to 0s in the GCN convolution).

### Manual RGCN version 1.0 - Experiments

1. The comparison is made with the latest experiment that used the pre-made RGCN and included the punctuation.
   1. It had final nll\_loss= 2.44262 after 100 epochs, and it still needed more epochs to stabilize fully.
   2. At 100 epochs, nll\_loss= 3.7078 . It seems to be worse: the loss descendes more quickly but it encounters a plateau at a higher altitude.  
      Must: double-check the code; experiment with the batch size & learning rate.
2. Num. tokens=128. Batch size=4. Learning rate=0.001. Epochs=250.
   1. Pre-made RGCN: nll\_loss= 1.44663
   2. My RGCN: nll\_loss= 2.64023 . It’s just worse.   
      Adding bias term to GCN-convolutions, and checking dimensions.  
      There is an error when the effective grapharea\_size is < 32, it can not add the bias term.

### Time analysis on MyRGCN

t1 - t0 = 0.87455

t2 - t1 = 0.00921

t3 - t2 = 0.0

Where t1-t0 is: loss = compute\_model\_loss(model, batch\_input, batch\_labels, verbose)

Inside the forward call, for each element in the batch:

t1 - t0 = 0.00191

t2 - t1 = 0.10676

t3 - t2 = 1e-05

t4 - t3 = 0.0001

t5 - t4 = 2e-05

Line: rgcn\_conv = rgcn\_convolution(x, Ar\_ls, self.Wr\_all, self.biasr\_all)

Taking out the normalization constant c\_i\_r:

t1 - t0 = 0.00198

t2 - t1 = 0.00026

t3 - t2 = 1e-05

t4 - t3 = 8e-05

t5 - t4 = 2e-05

* After 50 epochs, PremadeRGCN has nll\_loss= 3.17274
* After 30 epochs, without proper initialization, with the normalization constant, MyRGCN is still at: nll\_loss=16.44657
* After 50 epochs, with proper initialization, without the normalization constant, MyRGCN: nll\_loss= 5.89391

Reworking the normalization constant to be faster, going from 2 for-cycles to tensor operations.

Iteration time becomes 0.03 / 0.04 seconds

* After 50 epochs, with proper initialization, with the normalization constant, MyRGCN: nll\_loss= 5.83789

Currently using batch\_size=8. The iteration time of the premade version is ~0.015 seconds.

## Composing GCNs

It is possible to implement a “hybrid” RGCN, where I split the edge\_index but I use the pre-made standard GCNs to execute the convolution on for each relation.

Note: version 1.0 of it is missing the direct connection from previous layer

* Num. tokens=still 128. Batch size=4. Learning rate=0.001. Epochs=250.  
  Multiple GCNs, version 1.0: nll\_loss= 2.25827  
  Better than my RGCN, but worse than the PremadeRGCN.

## Trainable parameters

Before we can declare the pre-made RGCN to be superior, it is opportune to check the number of trainable parameters of each model.

**Pre-made RGCN:**

Parameters:

[('conv1.basis', torch.Size([5, 300, 300]), True),

('conv1.att', torch.Size([5, 5]), True),

('conv1.root', torch.Size([300, 300]), True),

('conv1.bias', torch.Size([300]), True),

('linear2global.weight', torch.Size([9858, 300]), True),

('linear2global.bias', torch.Size([9858]), True),

('linear2sense.weight', torch.Size([17843, 300]), True),

('linear2sense.bias', torch.Size([17843]), True)]

Number of trainable parameters=8,878,326

**MyRGCN:**

Parameters:

[('linear2global.weight', torch.Size([9858, 300]), True),

('linear2global.bias', torch.Size([9858]), True),

('linear2sense.weight', torch.Size([17843, 300]), True),

('linear2sense.bias', torch.Size([17843]), True)]

Number of trainable parameters=8,338,001

Apparently, all the weights’ matrices W\_r-s have not been included among the parameters. It is necessary to correct this.

Documentation: **torch.nn.Parameter**

“A kind of Tensor that is to be considered a module parameter.

Parameters are Tensor subclasses, that have a very special property when used with Module-s - when they’re assigned as Module attributes they are automatically added to the list of its parameters, and will appear e.g. in parameters() iterator. Assigning a Tensor doesn’t have such effect. …”

**MyRGCN**:

[('Wr\_all', torch.Size([6, 300, 300]), True),

('biasr\_all', torch.Size([6, 32, 300]), True),

('linear2global.weight', torch.Size([9858, 300]), True),

('linear2global.bias', torch.Size([9858]), True),

('linear2sense.weight', torch.Size([17843, 300]), True),

('linear2sense.bias', torch.Size([17843]), True)]

Number of trainable parameters=8,935,601

**CompositeRGCN:**

[('W\_0', torch.Size([300, 300]), True),

('convs\_ls.0.weight', torch.Size([300, 300]), True),

('convs\_ls.0.bias', torch.Size([300]), True),

('convs\_ls.1.weight', torch.Size([300, 300]), True),

('convs\_ls.1.bias', torch.Size([300]), True),

('convs\_ls.2.weight', torch.Size([300, 300]), True),

('convs\_ls.2.bias', torch.Size([300]), True),

('convs\_ls.3.weight', torch.Size([300, 300]), True),

('convs\_ls.3.bias', torch.Size([300]), True),

('convs\_ls.4.weight', torch.Size([300, 300]), True),

('convs\_ls.4.bias', torch.Size([300]), True),

('linear2global.weight', torch.Size([9858, 300]), True),

('linear2global.bias', torch.Size([9858]), True),

('linear2sense.weight', torch.Size([17843, 300]), True),

('linear2sense.bias', torch.Size([17843]), True)]

Number of trainable parameters=8,879,501

**Further observations:**

Logits parameters for globals: 9858\*300 + 9858= 2,967,258

Logits parameters for senses: 17843\*300 + 17843 = 5,370,743

Tot. logits parameters: 8,338,001

PremadeRGCN-specific parameters: 8,878,326 - 8,338,001 = 540,325

MyRGCN-specific parameters: 597,600

CompositeRGCN-specific parameters: 541,500

## Final experiment – all parameters explicitly included

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| batch size | 8 |  | training tokens | 128 |
| graph\_area | 32 |  | epochs | 150 |
| learning rate | 0.002 |  | final global step | 2400 |

* Pre-made RGCN: Final training nll\_loss= 2.146
* MyRGCN: Final training nll\_loss =3.402
* **CompositeRGCN**: Final training nll\_loss= 1.994
  + Using the CompositeRGCN without bias mirrors better the RGCN formula, and the performance is basically identical, at nll\_loss=1.998



The CompositeRGCN replicates the formula for RGCN:

with R GraphConvolutionalNetworks operating on the subgraphs of the different edge types. The output is summed up, and then we employ the W0 matrix for the connection from the previous layer of the node itself.

### On the side: Experiment – Composite RGCN with Leaky ReLU

A consideration on the side, that never occurred before:

if I use ReLU, am I not forcefully cutting to 0 some dimensions of all the entities (senses, definitions, etc.) that were initialized with some negative value among their d=300 dimensions?

It is worthwhile to explore what happens if I set the non-linear function on the rgcn\_conv representation to something else, like LeakyReLU (default negative\_slope=0.01).

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| batch size | 8 |  | training tokens | 128 |
| graph\_area | 32 |  | epochs | 150 |
| learning rate | 0.002 |  | final global step | 2400 |

Previous experiment, with Composite RGCN: final training nll\_loss = 1.994

Composite RGCN with LeakyReLU(negative\_slope=0.01), final training nll\_loss = 1.93342

Composite RGCN with LeakyReLU(negative\_slope=0.1), final training nll\_loss= 1.93183

The LeakyReLu does in fact bring a minor benefit. I would opt for a cautious choice and use the default negative slope of 0.01.

# 6: Memory & Recurrence

## Gated GNNs

“Gated Graph Sequence Neural Networks” by Y.Li et al. (2015-2017) defines the inclusion of Gated Recurrent Units in GNNs. (We only care for the recurrent step that builds the representation, not for the output values/sequences)

It unrolls the recurrence for a fixed number of steps T, and uses BPTT.

* Gated GNN recurrence:
  + + **b**  
    where the matrix A describes the graph structure (e.g. adjacency matrix)  
    i.e. select the hidden states of the neighbours
  + Update gate:
  + Reset gate:
  + New-state:
  + Updated state:

There are 2 possibilities:

1. Save the representation built by the RGCN mechanism (the sum of the output of the GCNs and the previous-layer node-connection), and use it as the input of the reset&update gates. This part should be written manually.
2. On each subgraph from the edge type, instead of using the simple GCNs provided by Pytorch-Geometric, use the **GatedGraphConv** networkclass.  
   This would mean having a greater number of parameters…

## Writing the GCNs+GRU

### Manual GRU on the representation

I decide to have a update\_gate *u*, with an update:

The update\_gate will be updated based on (x, edge\_index, edge\_type), i.e. the input of each batch element

Following (partially) the formula:

where is just the concatenation of the neighbourhood, + **b**

So for us will be the selected graph\_area, in order to operate on fixed input dimensions.

It is necessary to have 2 matrices, update\_gate\_W (32\*300 x ~~1~~ 300) and update\_gate\_U (300 x ~~1~~ 300)

**Note:** I could replace 'update\_gate\_W', that has torch.Size([9600, 1]) since it operates on the concatenation of the graph area, with a GCN on (x, edge\_index).

**Issue:** I am encountering:

RuntimeError: Trying to backward through the graph a second time, but the buffers have already been freed. Specify retain\_graph=True when calling backward the first time.

Hypothesis: the error may be caused by the fact that the model does not keep the intermediate results that are necessary to execute BPTT.

Relevant answer on discuss.pytorch.org:

“Am I right in saying that your training loop doesn’t detach or repackage the hidden state in between batches? If so, then loss.backward() is trying to back-propagate all the way through to the start of time, which works for the first batch but not for the second because the graph for the first batch has been discarded.

If I am right then there are two possible solutions.

1. detach/repackage the hidden state in between batches. There are (at least) three ways to do this.  
   hidden.detach\_()  
   hidden = hidden.detach()  
   hidden = Variable(hidden.data, requires\_grad=True)
2. replace loss.backward() with loss.backward(retain\_graph=True) but know that each successive batch will take more time than the previous one because it will have to back-propagate all the way through to the start of the first batch.”

The error happens after I get through step=1. Therefore, I am trying to BPTT from 1 batch to the previous (I should not be doing that) when the intermediate results have already been lost.

Thus I write: self.memory\_previous\_rgcnconv.detach\_(), executed in the forward() at the start of each batch.

In addition to what has been described, there are **3 variants** that can be considered:

* replace 'update\_gate\_W', that operates on the concatenation of the graph area having torch.Size([9600, 1]), instead using a GCN on (x, edge\_index).
* Instead of using a gate that is a constant, with dim=1, to decide whether to preserve/update the hidden state, use a gate with the same number of dimensions as the hidden state (here, dim=300)
* Follow the formulas mentioned previously in full: Use 2 gates, reset *u* and update *z*, where *r* is used to create the proposed new-state (that has a Tanh on it, although I could apply a LeakyReLU)

Observation: I am operating on the whole grapharea: the rgcn\_conv has dimension torch.Size([1, 32, 300])…

For the update gate, the matrix W is multiplied per the concatenated neighbourhood (for me, the graph\_area). The matrix U should operate only on the current node/word.

### Composite GatedGraphConv

**classGatedGraphConv(out\_channels, num\_layers, aggr='add', bias=True, \*\*kwargs)**

The gated graph convolution operator from the [“Gated Graph Sequence Neural Networks”](https://arxiv.org/abs/1511.05493) paper

Equations:

For every relation *r*  (since we split the graph into subgraphs), there will be:

* Update gate:
* Reset gate:
* Proposed new-state:
* Updated state:

Which means that for every *r* we will have 2 + 2 + 2 = 6 matrices, in total 5\*6=30 matrices.

## Experiment

### Settings and loss

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| batch size | 8 |  | training tokens | 128 |
| graph\_area | 32 |  | epochs | 100 |
| learning rate | 0.002 |  | final global step | 1600 |

* **GRU\_RGCN**: manual GRU with one gate *u* on the rgcn\_conv, dimension 1, loss=2.26
* **Multiple GGCNs**: executing separately the gated convolution for each relation, using the pre-made GatedGraphConv networks. Final training nll\_loss = 2.02091
* **GRU\_RGCN**: manual GRU with one gate *u* on the rgcn\_conv, dimension 300, loss=0.0011
* **GRU\_RGCN\_Wconv**: manual GRU with one gate *u* on the rgcn\_conv, dimension 300, and the W matrix for the update gate is not a matrix that gets multiplies per the concatenated graph\_area, but instead a GCN. Loss= 0.27001   
  (although it bounced back after reaching 0)



### Conclusions

The hidden state, saved in the buffer self.memory\_previous\_rgcnconv, has size 32 x 300.   
We are saving the entire graph\_area, which means: the current node, and the <=32 adjacent nodes.

An update gate of 300 dimensions decides which dimensions to keep and which to discard in the hidden state.

An update gate of 1, that only presents the decision whether to discard or keep, does not manage to overfit on a small training set.

# 7: Experiments on SemCor.xml – Round 1

It is now time to:

* review all the parameters used
* build a graph (and graph-area matrix) from the whole SemCor.xml
* train a Recurrent Graph Neural Network on it, to predict globals and senses for the Language Model task

## Parameters review

CreateGraphInput.exe(…):

* vocabulary\_from\_senselabeled=**True**:  
  In order to have all the phrases that are present in multi-sense corpuses, such as “took\_place”, we get the vocabulary from the training split of our current Sense-Labeled Corpus.
* V.get\_vocabulary\_df(senselabeled\_or\_text=vocabulary\_from\_senselabeled, slc\_split\_name=**'training'**, corpus\_txt\_filepath=vocab\_text\_source, out\_vocabulary\_h5\_filepath=outvocab\_filepath, min\_count=5):
  + slc\_split\_name:  
    later on, I may change the code of the function and operate on a list of splits, that would reasonably be [‘training’, ‘validation’].   
    For now, it stays as it is.
  + min\_count:  
    5. It could be made higher, depending on the total number of tokens in the vocabulary.
* CE.compute\_single\_prototype\_embeddings(  
  vocabulary,   
  os.path.join(F.FOLDER\_INPUT, F.SPVs\_FASTTEXT\_FILE),  
  CE.Method.FASTTEXT):  
  The purpose of this function is: iterate over the vocabulary that we previously built from the training corpus, use either DistilBERT or FastText to compute d=768 or d=300 single-prototype word embeddings.
  + method:  
    Currently FastText. This choice influences the quality of the starting embeddings that are used for globals, and for definitions & examples.  
    Hypothetically: FastText > fewer dimensions > faster, easier training whereas DistilBERT (or AlBERT / any small version of BERT) > better quality.
* Retrieving the d-e-s-a input data from WordNet has a requests\_segment\_size = 50000.  
  I have never encountered problems with the number of requests to WordNet (BabelNet instead is another story…) and until now our vocabularies of globals have been 5/10/20K. However, I increase this to 100,000 just in case.

DefineGraph.get\_graph\_dataobject (new=**False**, method=Method.FASTTEXT)

Training.train(grapharea\_size=32,batch\_size=8,learning\_rate=0.001,num\_epochs=50):

* Obviously, the grapharea\_size, batch\_size, learning\_rate are hyperparameters that should be explored in grid-search.

### Experiment 1

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| batch size | 8 |  | training tokens | 646,032 |
| graph\_area | 32 |  | epochs | 50 |
| learning rate | 0.001 |  | final global step | 242262 |

In the first attempt, it stopped after 3 epochs, due to an oscillation in the validation loss and the early-stopping mechanism.

|  |  |  |
| --- | --- | --- |
| **Epoch** | **Training loss** | **Validation loss** |
| 1 | 12.140 | 7.037 |
| 2 | 12.688 | 6.999 |
| 3 | 12.755 | 7.012 |
| 4 | 12.571 | 7.071 |

(Note: from now on I modify the early-stopping mechanism:

* if the validation loss at the end of an epoch is greater than the previous one for the 2nd time during training, then stop
* every time that the validation loss is better (lower), we save the model

)

As it stands, the model is unable to learn anything. However, an analysis of the model’s structure and parameters can explain why it is so.

### Reviewing the model structure

INFO : [

('W\_0', torch.Size([300, 300]), True), # previous layer, self-connection

('update\_gate\_W', torch.Size([9600, 300]), True), # graph\_area > update\_gate(300)

('update\_gate\_U', torch.Size([300, 300]), True), # current node/word > update\_gate

('convs\_ls.0.weight', torch.Size([300, 300]), True), # graph\_area > rgcn\_conv\_rep

('convs\_ls.1.weight', torch.Size([300, 300]), True), # “

('convs\_ls.2.weight', torch.Size([300, 300]), True), # “

('convs\_ls.3.weight', torch.Size([300, 300]), True), # “

('convs\_ls.4.weight', torch.Size([300, 300]), True), # “

('linear2global.weight', torch.Size([9858, 300]), True), # rgcn\_conv\_rep > globals

('linear2global.bias', torch.Size([9858]), True), # “

('linear2sense.weight', torch.Size([17843, 300]), True), # rgcn\_conv\_rep > senses

('linear2sense.bias', torch.Size([17843]), True)] # “

INFO : Number of trainable parameters=11,848,001

The number of parameters used for the softmax is 8,338,001.

We thus have merely 3,510,000 “effective” parameters in the model.

Comparisons:

The optimized AWD-LSTM (Merity et al., 2017) that was used on WikiText-2 has 33M.

The Transformer-XL (standard) on WikiText-103 has 151M.

Crucially, the entirety of the embeddings of *globals*, *senses*, definitions, examples are currently fixed to their initial default values.

They are not included as a Parameter, they are just the input **x** passed to the GCNs.

The model does not have any way of moving them (i.e. modifying their embeddings) in the multi-dimensional space, it just tries to adapt the GCNs and update gate while having an insufficient number of parameters.

Possible path:

* Load the **x** matrix from the graph.
* The embeddings of definitions and examples should be left as they are, e.g. with a no\_grad specification. They were initialized by sentence embedding with FastText/miniBERT and their location makes sense.
* The embeddings of globals and senses should be a parameter that can be optimized.

In DefineGraph,

X = torch.cat([X\_senses, X\_globals, X\_definitions, X\_examples])

I could specify that, at the end of every forward, the gradient of the X\_definitions and X\_examples is assigned to 0. The example in discuss.pytorch.org/… does it after the backward(), and before the optimizer.step()

As a consequence, in the batch element I should not send the x anymore, but the indices – then we select the x from the Parameter X.

We add to the list of parameters

'X', torch.Size([65632, 300]), True)

Adding 19,689,600 parameters, for a total of 31,537,601

– even if, among the 19M, only the embeddings for senses & globals, 17843 + 9858=27701 x 300 = 8,310,300 are optimized

Although num\_total\_params = 31.5M

(20.2M optimized, among which 11.9M core and 8.3M softmax)

In a mini-experiment,

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| batch size | 8 |  | training tokens | 216 |
| graph\_area | 32 |  | epochs | 100 |
| learning rate | 0.001 |  | final global step | 2700 |

we get: final training nll\_loss= 0.0004

## Experiment 2

We added the X matrix of embeddings as a Parameter of the network, and in every iteration we set the gradient of X\_definitions and X\_examples to 0 before calling optimizer.step()

We thus brought the number of parameters of the model to 31M, where 11.9M are “core” (embeddings + GCNs), 8.3M softmax, and 11.3M kept fixed.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| batch size | | 8 |  | training tokens | | 646,032 |
| graph\_area | | 32 |  | epochs | | 100 |
| learning rate | | **0.001** |  | steps in 1 epoch | | 80754 |
|  | | | | | | |
| Epoch | Training loss | | | | Validation loss | |
| 1 | 12.525 | | | | 7.318 | |
| 2 | 12.342 | | | | 7.834 | |
| 3 | 12.353 | | | | 7.231 | |

Bad. Let us try longer batch\_size and smaller graph\_area:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| batch size | | 16 |  | steps in 1 epoch | | 40377 |
| graph\_area | | 16 |  |  | |  |
|  | | | | | | |
| Epoch | Training loss | | | | Validation loss | |
| 1 | 12.956 | | | | 8.879 | |
| 2 | 13.851 | | | | 8.990 | |
| 3 | 13.837 | | | | 8.780 | |
| 4 | 13.723 | | | | 8.779 | |

Again, we observe no visible improvement.

### Modifications: Learning rate

First: I experiment with a lower and a higher learning rate, trying 0.00001 and 0.01.

**Learning rate = 0.01** -> bad.

Training, epoch nll\_loss= 18.48172

After training 9 epochs, validation nll\_loss= 16.77284

------

Early stopping

**Learning rate = 10-5** -> acceptable.

|  |  |  |
| --- | --- | --- |
| **Epoch** | **Training loss** | **Validation loss** |
| 1 | 11.048 | 6.248 |
| 2 | 10.347 | 6.096 |
| 3 | 10.045 | 6.014 |
| 4 | 9.860 | 5.965 |
| 5 | 9.737 | 5.925 |
| 6 | 9.653 | 5.902 |
| 7 | 9.593 | 5.890 |
| 8 | 9.550 | 5.876 |
| 9 | 9.521 | 5.871 (ppl=354.6 ) |
| 10 | 9.499 | 5.867 |
| 11 | 9.487 | 5.862 |

Stopping for now.

### Input batching

Although it could be better, and the training process could be faster.

Currently, with “batch\_size”=8, iteration time~=0.05s, and 1 epoch on SemCor = 80K steps ~= 4000s ~= 1 hour.

I deem it opportune to pack the input tuples into tensors, in order to implement batching and parallel processing.

Right now what I have been calling “batch\_size” is actually the sequence length; I send 1 sequence at a time to the GRU\_RGCN, that gets processed sequentially.

**Bug**: I should not be sorting the nodes\_ls that I pass to the input, or I lose track of the current node.

current\_token\_tpl=(0, -1)

current\_token\_tpl=(9857, 7607)

current\_token\_tpl=(1, 15301)

current\_token\_tpl=(2, -1)

input\_ls > x\_indices =

[tensor([ 0, 1, 2, 3, 4, 4696, 4697, 4720, 18469, 19191,

…]),

tensor([ 7607, 7608, 7609, 7610, 7611, 12677, 12678, 12679, 12680, 12681,

…]),

tensor([15296, 15297, 15298, 15299, 15300, 15301, 18004, 18530, 18607, 19543,

…]),

tensor([ 0, 1, 2, 3, 4, 4696, 4697, 4720, 18469, 19191,

…])]

Now:

current\_token\_tpl=(0, -1)

current\_token\_tpl=(9857, 7607)

current\_token\_tpl=(1, 15301)

current\_token\_tpl=(2, -1)

[tensor([ 0, 27701, 45544, 21530, 2, 3, 4, 1, 22477, 22476,

…]),

tensor([ 7607, 35308, 18662, 7609, 7610, 7611, 7608, 20594, 35310, 35311,

…]),

tensor([15301, 43002, 62776, 62779, 62778, 62777, 18004, 15297, 15300, 15298,

…]),

tensor([ 2, 27703, 45546, 45548, 45547, 21530, 3, 0, 4, 1,

…])]

The size of the batchinput\_tensor is [4, 8, 416]

Let us move the packing of a batch from the Training module to DataLoading’s collate\_fn…

In my BPTTBatchCollator, having set batch\_size=4, I have 4 elements/input tuples…

Setting it to batch\_size\*seq\_len, i.e. =32

I am appending in input\_lls Tensors of shape (8, 416). batchinput\_tensor is still [4, 8, 416]

Finally, after applying nn.DataParallel properly, I get:

batch\_input.shape=torch.Size([4, 8, 416])

batchinput\_tensor.shape=torch.Size([1, 8, 416])

batchinput\_tensor.shape=torch.Size([1, 8, 416])

batchinput\_tensor.shape=torch.Size([1, 8, 416])

batchinput\_tensor.shape=torch.Size([1, 8, 416])

Adjusting… caused an error on edge\_index and split\_edge\_index…

packing. x\_indices =

tensor([ 0, 27701, 45544, 21530, 2, 3, 4, 1, 22477, 22476,

23569, 27703, 45546, 45548, 45547, 27704, 45550, 45549, 27705, 45551,

27702, 45545, 4697, 4696, 18469, 4720, 25636, 21031, 25376, 19191,

21485, 32398])

Unpacking;

x\_indices =

tensor([0.0000e+00, 2.7701e+04, 4.5544e+04, 2.1530e+04, 2.0000e+00, 3.0000e+00,

4.0000e+00, 1.0000e+00, 2.2477e+04, 2.2476e+04, 2.3569e+04, 2.7703e+04,

4.5546e+04, 4.5548e+04, 4.5547e+04, 2.7704e+04, 4.5550e+04, 4.5549e+04,

2.7705e+04, 4.5551e+04, 2.7702e+04, 4.5545e+04, 4.6970e+03, 4.6960e+03,

1.8469e+04, 4.7200e+03, 2.5636e+04, 2.1031e+04, 2.5376e+04, 1.9191e+04,

2.1485e+04, 3.2398e+04])

packing. Edge\_sources=

tensor([ 1, 20, 11, 15, 18, 31, 2, 21, 12, 14, 13, 17, 16, 19, 3, 3, 3, 3,

3, 8, 8, 9, 10, 3, 10, 3, 10, 3, 8, 3, 9, 24, 8, 26, 9, 29,

10, 27, 10, 28, 10, 30, 10, 30, 10])

Unpacking.

edge\_sources=

tensor([0.0000e+00, 2.7701e+04, 4.5544e+04, 2.1530e+04, 2.0000e+00, 3.0000e+00,

4.0000e+00, 1.0000e+00, 2.2477e+04, 2.2476e+04, 2.3569e+04, 2.7703e+04,

4.5546e+04, 4.5548e+04, 4.5547e+04, 2.7704e+04, 4.5550e+04, 4.5549e+04,

2.7705e+04, 4.5551e+04, 2.7702e+04, 4.5545e+04, 4.6970e+03, 4.6960e+03,

1.8469e+04, 4.7200e+03, 2.5636e+04, 2.1031e+04, 2.5376e+04, 1.9191e+04,

2.1485e+04, 3.2398e+04, 1.0000e+00, 2.0000e+01, 1.1000e+01, 1.5000e+01,

1.8000e+01, 3.1000e+01, 2.0000e+00, 2.1000e+01, 1.2000e+01, 1.4000e+01,

1.3000e+01, 1.7000e+01, 1.6000e+01])

packing. Edge\_destinations=

tensor([ 0, 7, 4, 5, 6, 22, 0, 7, 4, 4, 4, 5, 5, 6, 0, 7, 4, 5,

6, 23, 22, 25, 10, 10, 3, 10, 3, 8, 3, 9, 3, 8, 24, 9, 26, 10,

29, 10, 27, 10, 28, 10, 30, 10, 30])

packing. in\_tensor=…

**Bug**: tensors on different GPUs…

Now, iteration time =~0.19s, having sequence\_length=8 x batch\_size=4

Using memory on all 4 GPUs, and utilization ~ 21%-15%-15%-15%

### Time Analysis

On CPU:

t1 - t0 = 0.06577

t2 - t1 = 1.05096

t3 - t2 = 0.2414

t4 - t3 = 0.0

On GPU:

**t1 - t0** = 0.16963. loss = compute\_model\_loss(model, batch\_input, batch\_labels, verbose)

t2 - t1 = 0.02485

t3 - t2 = 0.00191

t4 - t3 = 0.0

E.g.:

Current device : 1 ; t1 - t0=0.0677s ; t2 - t1=0.0102s ; t3 - t2=0.0118

Current device : 2 ; t1 - t0=0.0675s ; t2 - t1=0.0103s ; t3 - t2=0.0117

Current device : 1 ; t05 - t0=0.0001s ; t1 - t05=0.0688s ; t2 - t1=0.083s ; t3 - t2=0.0042

batchinput\_ls = [unpack\_input\_tensor(batchinput\_tensor.squeeze(), self.N)

for batchinput\_tensor in paddedtensors\_ls]

As it stands, I am unable to improve speed further. The time saved by parallelism is mostly consumed by unpacking each tensor in the BPTT sequence using a for cycle.

And the BPTT sequence itself requires a for cycle (this can not be avoided).

Current device : 0 ; t05 - t0=0.0009s ; t1 - t05=0.0678s ; t2 - t1=0.1029s ; t3 - t2=0.0078s ; t3 - t1=0.1107

Nevertheless, the time spent remains the same (~0.20s on 4x8, equivalent to 0.05 on 8) and now I have ironed out more bugs (e.g. wrong specification of the current node in the graphArea) and implemented batching.

Mini-experiments indicate that it is opportune to increase the learning rate.

## Experiment 3

Preliminary check: on a mini dataset, after as few as 20 epochs we have Training nll\_loss= 0.924 (with a high validation nll\_loss= 12.116 , since we are ust overfitting on a mini-training set).

Then we experiment on the whole SemCor.xml dataset:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| batch size | 4 |  | TBPTT length | 8 |
| graph\_area | 32 |  | training tokens | 646,032 |

Note: in the current version, I am keeping definitions and examples fixed.

Learning rate: 10^-5

|  |  |  |
| --- | --- | --- |
| **Epoch** | **Training loss** | **Validation perplexity** |
| 1 | 13.771, | 551.379, |
| 2 | 12.868, | 453.712, |
| 3 | 12.496, | 408.852, |
| 4 | 12.208, | 380.825, |
| 5 | 11.988, | 363.539, |
| 6 | 11.811, | 352.696, |
| 7 | 11.664, | 346.226, |
| 8 | 11.535, | 342.033, |
| 9 | 11.421, | 338.957, |
| 10 | 11.319 | **336.441** |

Learning rate: 10^-4

|  |  |  |
| --- | --- | --- |
| **Epoch** | **Training loss** | **Validation perplexity** |
| 1 | 12.681, | 360.179, |
| 2 | 11.473, | **338.949**, |
| 3 | 11.021 | , 351.235, |
| 4 | 10.773 | 370.273 |

Learning rate: 10^-3

|  |  |  |
| --- | --- | --- |
| **Epoch** | **Training loss** | **Validation perplexity** |
| 1 | 13.284, | 693.253, |
| 2 | 13.66, | 702.136, |
| 3 | 13.216, | 613.846 |
| 4 | 14.129 | 807.694 |

Learning rate: 10^-6.

|  |  |  |
| --- | --- | --- |
| **Epoch** | **Training loss** | **Validation perplexity** |
| 1 | 17.582 | 4627.64 |
| 2 | 14.218 | 703.26 |
| 3 | 13.395 | 667.00 |
| 4 | 13.353 | 647.24 |
| 5 | 13.304 | 629.13 |
| 6 | 13.264 | 613.39 |

Learning rate: 10^-5, TBPTT sequence length=16

|  |  |  |
| --- | --- | --- |
| **Epoch** | **Training loss** | **Validation perplexity** |
| 1 | 14.394 | 4715.59 |
| 2 | 13.24 | 3673.27 |
| 3 | 12.931 | 3183.24 |
| 4 | 12.662 | 2847.03 |
| 5 | 12.427 | 2631.09 |
| 6 | 12.233 | 2489.7 |
| 7 | 12.072 | 2390.94 |

# Meeting (PSK, 04/03/20)

ToDo list:

* Add the separate recording of Perplexity on Globals and Perplexity on Senses.  
  [done]
* Turn offline the sense-head of the network[done], and train on standard text such as WT-2[in progress]
* Use DistilBERT initialization instead of FastText, d=768 instead of 300
* How can we compute perplexity in a meaningful way on our Sense-Labeled Datasets, such as SemCor.xml ? [see (1)]
* Explore how to speed up the forward()  
  [marginal improvement, could not speed up more]
* Link the globals to Wikipedia (allowing us to use Linked WikiText-2). It can be tested separately or added on the top of the rest.

# 8: Loss and model modifications

## On the loss

Until now, we have summed up loss\_globals + loss\_sense, and then divided by the num\_steps\_in\_epoch.

However, this brings an erroneous contribution from the senses: their loss is always =0 when there is no sense.

Moreover, the perplexity values must be computed separately for globals and senses, otherwise they make no sense.

It is necessary to keep separate statistics. The loss on senses should be computed/normalized using the number of steps that actually had a sense label.

### Mini-Experiment

n: We recall that the vocabulary sizes on SemCor, with mincount=5, are 9858 globals, 17843 senses

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| batch size | 4 |  | TBPTT length | 8 |
| graph\_area | 32 |  | training tokens | 128 |
| learning rate | **0.001** |  |  |  |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Epoch | Globals loss | Globals perplexity | Senses loss | Senses perplexity |
| 1 | 9.15 | 9411.859 | 9.782 | 17703.679 |
| 10 | 3.327 | 27.851 | 2.812 | 16.651 |
| 60 | 0.12 | 1.127 | 0.026 | 1.026 |



## On the speed

Is there any way to have a faster forward(), instead of using 0.20s on 4x8 tokens?

The for-cycle inside the TBPTT is inevitable…

It uses (x, edge\_index, edge\_type)

The input elements are unpacked from the sample’s padded tensor, that holds all the tokens in a TBPTT sequence, using a for cycle (slow!).

Yet another time-analysis is in order:

t1 - t0 = **0.0651**

in the for-cycle:

t3 - t2 = 6e-05

t4 - t3 = 0.0

t5 - t4 = **0.00288**

t6 - t5 = 0.00019

t7 - t6 = **0.00707**

t8 - t7 = 0.00021

t9 - t8 = 0.00018

t10 - t9 = 0.00027

t11 - t10 = 0.00032

t12 - t11 = 0.00031

After modifying the tensor manipulation to extract the BPTT elements’ (x, edge\_index, edge\_type) from the padded tensor…

I have brought it down slightly, from ~0.20 to ~0.18, but no further luck.

## Experiment IV - SemCor

This time, definitions and examples in the graph are also mobile, all the 31.5M parameters of this version of the model are trainable.

We try again on the whole SemCor.xml.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| batch size | 4 |  | TBPTT length | 8 |
| graph\_area | 32 |  | training tokens | 646,032 |
| learning rate | 10^-5, 10^-4 |  |  |  |

Learning rate 10^-4

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Epoch | Training PPL - globals | Training PPL - senses | Validation PPL - globals | Validation PPL - senses |
| 1 | 272.946 | 2386.962 | 190.998 | 535.629 |
| 2 | 168.527 | 1088.187 | 178.238 | 551.442 |
| 3 | 143.069 | 776.163 | 182.761 | 624.04 |
| 4 | 130.885 | 635.672 | 191.264 | 671.432 |

Learning rate 10^-5

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Epoch | Training PPL - globals | Training PPL - senses | Validation PPL - globals | Validation PPL - senses |
| 1 | 506.604 | 3836.833 | 268.118 | 688.785 |
| 2 | 307.202 | 2517.478 | 223.133 | 613.214 |
| 3 | 255.317 | 1977.409 | 202.439 | 585.717 |
| 4 | 224.237 | 1650.956 | 189.25 | 575.543 |
| 5 | 204.316 | 1451.378 | 182.048 | 572.392 |
| 6 | 190.629 | 1305.307 | 178.124 | 571.38 |
| 7 | 180.098 | 1190.058 | 175.421 | 574.382 |
| 8 | 171.431 | 1093.681 | 173.388 | 577.23 |

### Observations

From this and previous experiments, we can surmise that LR=10^-4 is the best choice when we operate with batch\_size=4 and TBPTT=8.

It is possible that the prediction of globals and the one of senses are at odds.

We try to build a representation using the RGCN, and then this same representation get used for the 2 FF-NNs that give the logits for the softmax on globals and senses.

Maybe if we have only one objective we will have a better representation?

Predicting globals-only would still include in the input the definitions and examples and senses in the graph.

It is necessary to make a comparison.

## Preparation for WikiText-2

I intend to examine the perplexity of the model on WikiText-2.

The vocabulary will be created from it (training + validation splits), and then I will retrieve definitions & examples & etc. and initialize the senses. As stated above, the graph will have the same kinds of elements, but the output will be only-globals (classic LM).

### Vocabulary and graph statistics

From WT-2 Train + Validation, without eliminating any rare words, len(vocab.keys()) = 44069.

In the statistics, the vocabulary size from WT-2 is 33,278. Can I obtain the same?

43232 from the training set alone.

If I operate with min\_count=2, I obtain 31640 from the Training set alone. Close enough.

(n: Must also double-check how globals are initialized.

They are loaded as single-prototypes, i.e. from the FastText/miniBERT embeddings.)

**Nodes** in the graph:

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])

So tot\_nodes = 114,324

**Edges**:

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

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

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

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

syn\_edges.\_\_len\_\_()=40016

ant\_edges.\_\_len\_\_()=3938

edge\_index=[2, 147844].

**Parameters** of the GRU\_RGCN:

47,330,840 -> **47.3M** in total, where 114324 \* 300= 34.3M embeddings and 9.5M softmax

(previously, on SemCor, we had 31.5M total: 20.2M optimized, among which 19.6M embeddings and 8.3M softmax)

**Vocabulary**:

from WikiText-2 Training with min\_count=2 : 31640 globals, 0 senses

(from SemCor with min\_count=5, it was: 9858 globals, 17843 senses)

Issue: Batches & TBPTT (speed, buggy split\_edge\_index)

When I have batch\_size > n\_gpus, split\_edge\_index causes an error…

With batch\_size = 8:

batchinput\_tensor.shape = torch.Size([2, 8, 512])

sequenceinput\_lts is a tuple containing the input elements. Example:

Element 0: tensor, of shape torch.Size([2, 8, 32])

I have to unpack the batch size dimension. In this case, the parallelism is determined only by the number of GPUs.

*Additional note*: the K1 / sequence length can be slightly randomized, and not always coincide with the TBPTT length.

*Additional idea:* I could try a Graph Attention Network, or potentially even a Graph Transformer…

## Experiment V – WikiText-2

### Mini Experiment - Overfit

We verify that our GRU\_RGCN without the Sense head is capable of overfitting on a small dataset of 41\*32=1312 tokens.

Training, end of epoch 100. Global step n.4100. …

The training losses are:

Globals loss=0.88(and still descending) Globals perplexity=2.41

### Variants

Training tokens of WikiText-2 ‘s training split: 30961\*8\*8=1,981,504.

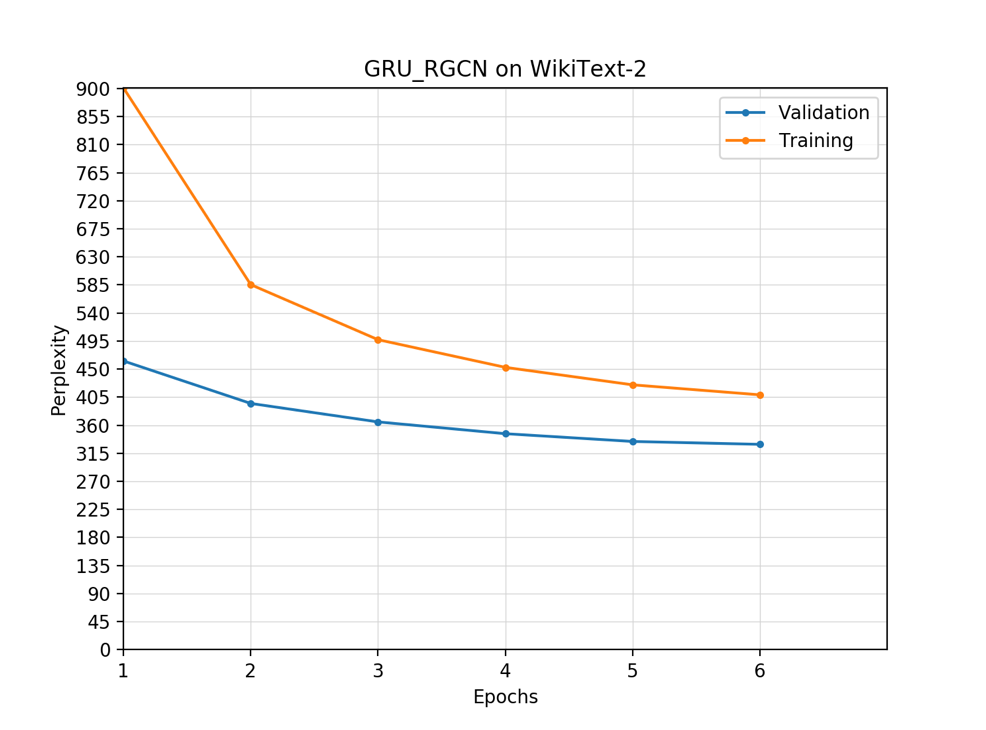
Validation split: 25829 \* 8 = 206,632

**Variant 1:**

(Training time for 1 epoch: 28072s = 7h47min52s )

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| batch\_size | 4 |  | TBPTT length | 8 (currently, fixed) |
| graph area | 32 |  | learning rate | 10^-5 |

|  |  |  |
| --- | --- | --- |
| *Epoch* | *Training perplexity* | *Validation perplexity* |
| 1 | 901.24 | 463.01 |
| 2 | 585.46 | 394.86 |
| 3 | 497.09 | 365.16 |
| 4 | 452.71 | 346.26 |
| 5 | 424.67 | 333.82 |
| 6 | 408.64 | 329.18 |

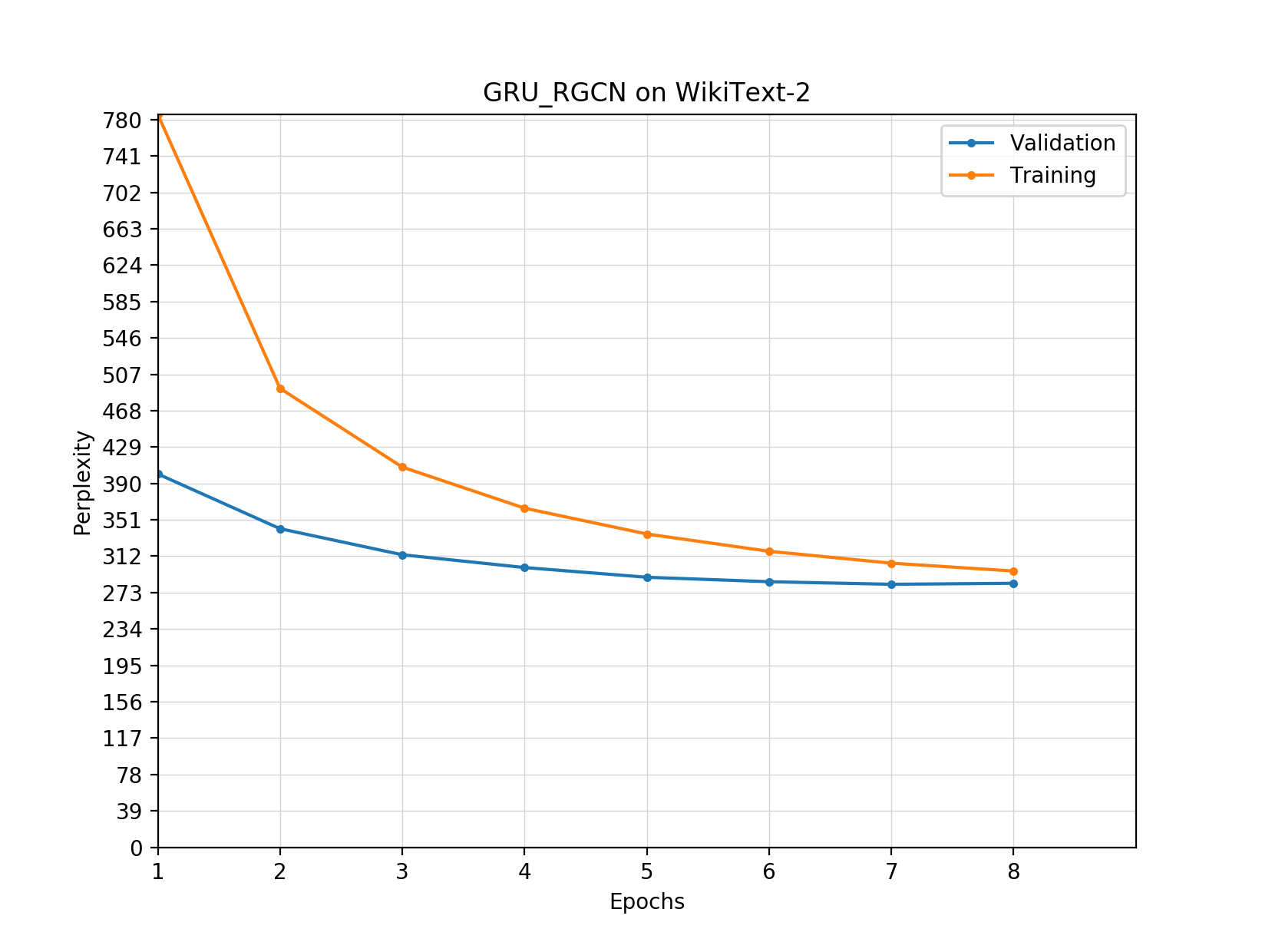


**Variant 2:**

( Training time for 1 epoch: 23354s = 6h29min14s )

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| batch\_size | 8 |  | TBPTT length | 8 |
| graph area | 32 |  | learning rate | 2 \* 10^-5 |

|  |  |  |
| --- | --- | --- |
| *Epoch* | *Training perplexity* | *Validation perplexity* |
| 1 | 785.75 | 400.19 |
| 2 | 491.92 | 341.51 |
| 3 | 407.55 | 313.57 |
| 4 | 363.49 | 299.86 |
| 5 | 335.83 | 289.46 |
| 6 | 317.29 | 284.67 |
| 7 | 304.56 | 281.90 |
| 8 | 296.13 | 282.98 |



# 9: Baseline comparisons and reconstructing the model

## RNN

First, 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: (300+1024) > 1024. s(t) = W\*x(t)
* Save s(t) in the memory buffer
* Obtain the logits, using the matrix linear2global: 1024 > |Vocab|
* Apply softmax

Parameters:

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

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

('module.memory\_context', torch.Size([1, 1024]), False),

('module.W', torch.Size([1324, 1024]), True),

('module.linear2global.weight', torch.Size([31640, 1024]), True), ('module.linear2global.bias', torch.Size([31640]), True)]

Number of trainable parameters=68.08 M ;   
34.30M embeddings, 32.43M softmax, 1.36M core (W)

### Experiment: RNN on WikiText-2

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| batch\_size | 4 |  | learning rate | 2\* 10^-5 |
| graph area | (1) |  | layers | 1 |
| TBPTT length | 16 |  | hidden state dim. | 1024 |

|  |  |  |
| --- | --- | --- |
| *Epoch* | *Training perplexity* | *Validation perplexity* |
| 1 | 4537.27 | 575.64 |
| 2 | 480.73 | 477.45 |
| 3 | 417.72 | 440.58 |
| 4 | 380.26 | 412.07 |
| 5 | 355.23 | 403.53 |
| 6 | 334.45 | 405.07 |
| 7 | 318.7 | 379.69 |
| 8 | 305.62 | 377.73 |

Hypothesis: I do not have better results here because of:

1. Learning rate too low
2. The limited expressiveness of only 1 layer with 1024 units (just a linear function)

Conclusion: Extend to a 2-layer network, and then insert a GRU on both hidden states.

### RNN, 2 layers

Word embeddings X. Current word w(t).

1st layer, hidden state h1: h1(t) = w(t) ++ h1(t-1)

2nd layer, hidden state h2: h2(t) = h1(t) ++ h2(t)

Parameters:

[…,

('module.memory\_h1', torch.Size([1, 1024]), False),

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

('module.W1.weight', torch.Size([1024, 1324]), True), ('module.W1.bias', torch.Size([1024]), True),

('module.W2.weight', torch.Size([512, 1536]), True), ('module.W2.bias', torch.Size([512]), True),

('module.linear2global.weight', torch.Size([31640, 512]), True),

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

Number of trainable parameters=52,67M

34.30M embeddings, 16.20M softmax, 2.14M core

In order to be faster, I select a development set: I keep ½ of the lines in WT-’s Training (18357 from 36717) and 1/2 of the lines in WT-2’s Validation datasets (1878 from 3758).

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| batch\_size | 4 |  | learning rate | 10^(-4) |
| graph area | (1) |  | layers | 2 |
| TBPTT length | 16 |  | hidden state dim. | 1024, 512 |

|  |  |  |
| --- | --- | --- |
| *Epoch* | *Training perplexity* | *Validation perplexity* |
| 1 | 772.54 | 439.55 |
| 2 | 415.94 | 394.01 |
| 3 | 353.99 | 378.8 |
| 4 | 318.83 | 377.79 |
| 5 | 293.37 | 384.56 |
| 6 | 256.66 | 379.46 |
| 7 | 240.24 | 368.12 |
| 8 | 227.2 | 371.45 |
| 9 | 216.09 | 374.55 |
| 10 | 206.19 | 380.7 |

The validation perplexity is still bad, and comparable to the previous attempt.

It is necessary to check: what are the **parameters** of RNNs used on WikiText-2?

And: how could I forget to use an **activation function** after the 1st hidden layer?

I examine “Restricted Recurrent Neural Networks” by E.Diao et al. 2019, paper that presents a compression method for RNNs, by tying the weights of the input and hidden state matrices.

While I do not care for Restricted RNNs, I wish to check the parameters used:

“A typical neural network architecture used in language modeling consists of an embedding layer, recurrent layers and a softmax layer.

Both embedding layer and softmax layer are fully connected neural networks.

We experiment with three recurrent layers with 200 hidden units and 200 embedding

size.

We use cosine annealing learning rate (…) [I use Adam, they use SGD]

We use a batch size 80 and 35 Back Propagation Through Time (BPTT) length for both datasets.”

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| batch\_size | 40 |  | learning rate | 10^(-4) |
| graph area | (1) |  | layers | 2 |
| TBPTT length | 32 |  | hidden state dim. | 512, 256 |
| Dropout | 0.1 on W1, W2 |  |  |  |

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.

## GRU + RNN

### Simplified GRU

Building upon the previous architecture of 2-layers RNN, we add our slightly simplified version of the Gated Recurrent Unit (possibly comparing it with the full version).

Gate:

where is just the concatenation of the neighbourhood, + **b**

So, if we decide to make a graph-aware version, will be the selected graph\_area, in order to operate on fixed input dimensions.

With a graph-unaware version, we shall have the gates for the 2 hidden states, where the first depends on the current word, and the second depends on the lower-level hidden state:

Same parameters as the ones used for the 2-layer RNN:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| batch\_size | 40 |  | learning rate | 10^(-4) |
| graph area | (1) |  | layers | 2 |
| TBPTT length | 32 |  | hidden state dim. | 512, 256 |
| Dropout | 0.1 on W1, W2 |  | GRU | 2 gates, d=512, 256 |

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

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

('module.U\_1.weight', torch.Size([512, 300]), True), ('module.U\_1.bias', torch.Size([512]), True),

('module.W1.weight', torch.Size([512, 300]), True), ('module.W1.bias', torch.Size([512]), True),

('module.U\_2.weight', torch.Size([256, 512]), True), ('module.U\_2.bias', torch.Size([256]), True),

('module.W2.weight', torch.Size([256, 512]), True), ('module.W2.bias', torch.Size([256]), True), ('module.linear2global.weight', torch.Size([31640, 256]), True), ('module.linear2global.bias', torch.Size([31640]), True)]

Number of trainable parameters=~43.0M

34.30M embeddings, 8.1 softmax, 0.57M core

|  |  |  |
| --- | --- | --- |
| *Epoch* | *Training perplexity* | *Validation perplexity* |
| 1 | 1215.96 | 581.35 |
| 2 | 674.9 | 460.08 |
| 3 | 526.82 | 400.75 |
| 4 | 445.9 | 365.8 |
| 5 | 396.09 | 347.05 |
| 6 | 360.72 | 335.25 |
| 7 | 332.75 | 323.45 |
| 8 | 308.79 | 314.49 |
| 9 | 288.17 | 308.27 |
| 10 | 270.67 | 305.31 |
| 11 | 255.71 | 302.5 |
| 12 | 242.68 | 302.13 |

Worse results than the concatenation of input + memory done in a standard RNN.

### Graph GRU

Let us try with the GRU used in Gated Graph Neural Networks:

* + + **b**  
    where the matrix A describes the graph structure (e.g. adjacency matrix)  
    i.e. select the hidden states of the neighbours
  + Update gate:
  + Reset gate:
  + New-state:
  + Updated state:

For now, we still ignore the adjacent nodes and the graph.

|  |  |  |
| --- | --- | --- |
| *Epoch* | *Training perplexity* | *Validation perplexity* |
| 1 | 29512.47 | 27394.45 |
| 2 | 25785.2 | 23958.27 |
| 3 | 22677.28 | 21010.18 |
| … | … | … |
| 12 | 7909.65 | 7123.25 |

Too slow. Let us review the number of parameters and turn up the learning rate?

Number of trainable parameters=43.61M

34.30M embeddings, 8.1M softmax, ~1.21M core

Or instead, review the GRU formula, and adjust the number of hidden units as well, to 300.

### Full GRU

Reviewing the formula for GRU:

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

It is equivalent to the one used in Gated GNNs, just replacing the neighbourhood with the input (word)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| batch\_size | 40 |  | learning rate | 10^(-4) |
| graph area | **(1)** |  | layers | 2 |
| TBPTT length | 32 |  | hidden state dim. | 300,300 |
| Dropout | None |  | GRU | full, on both |

[('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 | 913.5 | 444.28 |
| 2 | 499.33 | 350.37 |
| 3 | 406.51 | 311.24 |
| 4 | 358.76 | 290.77 |
| 5 | 328.82 | 278.75 |
| 6 | 309.39 | 271.62 |
| 7 | 289.83 | 257.56 |
| 8 | 268.3 | 248.41 |
| 9 | 252.36 | 241.79 |
| 10 | 239.39 | 236.73 |
| 11 | 228.23 | 232.46 |

Process killed. I have no regularization, and I overfit on the training set quite soon.

Trying again with batchsize=80, TBPTT=35, lr=0.0001, **dropout** on the W\_1, W\_2 matrices

|  |  |  |
| --- | --- | --- |
| *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.

It is now necessary to include in the network the graph input from WordNet.

# 10: Including the graph input

## Reflections on previous attempts

The GRU\_RGCN that we employed previously did not give great results on WikiText-2, with the best validation perplexity being **281.9**.

(On 4 GeForce RTX GPUs, 1 epoch on WT-2 takes 6-7 hours)

Operating on a vocabulary of **31.5K**, it had **47.3M** parameters in total, among which: embeddings=34.3M, softmax=9.5M, rest of the model (GCNs & co.)=3.5M



Maybe we need a different, possibly simpler & faster architecture than the GRU\_RGCN.

An architecture where, ideally, the sense prediction and the globals’ prediction do not counter each other.

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

## Graph Attention Networks on different kinds of nodes

### Recap of GATs

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.  
  
Considerations

Problem: in a graph area, we do not have necessarily the same number of nodes of a given type.

Answer: This should not matter. We just execute if . The resulting dimensionality can be either still *d=D=300*, or possibly *d=D/k*, where *k* would be the number of attention heads to concatenate.

should go: 2\*d -> 1, in order to apply and have the attention logit score,

Then: softmax over neighbourhood > weighted sum > non-linearity > new state of the node.

*Note:* The graph\_area=k includes at most k neighbouring nodes of the current word/node, and part of their neighbours inevitably get cut off.

Therefore, I should compute the new state only for the current/central node.

*Note II*: Moreover, this would make it viable to operate on graph areas of variable size, taking in exactly all the adjacent nodes of the current node.

### On senses

It seems a better idea to either:

* put the sense head consecutively after the globals’ head, instead of in parallel
* Restrict the sense prediction among the senses of the *k* (eg. *k*=10) most likely globals, or possibly, given the necessity of operating with fixed dimensions, among the *n* senses that have been ‘extracted’ from the most likely globals.

## 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

## GRU + GATs with different node types

### Architecture

In the GRU\_RNN, the input that we are feeding to the 1st gates z\_1 and r\_1 and to the proposed new-state is simply the word vector x(t) of the global.

However, it would be opportune to leverage the graph structure of (senseChildren > definitions||examples) || synonyms || antonyms.

The input to the 1st GRU layer should be the new state of the current word’s node according to the GAT.

We use the mechanism of the Graph Attention networks, modified in order to have different matrices for the different kinds of nodes:

* Shared linear transformation on every node,
* Non-normalized attention coefficients for neighbours:
* Normalized coefficients using softmax over the neighbourhood:
* New state of the node:

+ Multi-head attention (either through averaging or concatenating) was found beneficial.

### Input from the graph

This time, I need only the nodes that are directly adjacent to the central/current-word node.

I also foresee a sense architecture that will work as follows:

Input: global , sense. Both vectors and neighbourhood graphs.

Use GATs on both the global node (as usual, to predict the next global), and the sense node.

Then: There should be a layer/memory that takes in both the sense input and the global input.

The final head can work 2 ways: parallel output of next\_global and next\_sense, or sequentially choosing the sense among the ones of the k=10 most likely globals.

I decide to change the creation of the GraphArea matrix, so that only immediate neighbours of each node are recorded.

Currently, changed with a flag. Depending on results, I may keep only the retrieval of immediate neighbours and discard the graph-area that allows for a hop > 1.

**Note**: I do not need to extract the type of the adjacent nodes from the edge: the graph\_dataobj.node\_types attribute is a Tensor of shape torch.Size([114324]) with elements tensor([0, 0, 0, ..., 3, 3, 3]).

The node types are determined when we build the graph, in the following order:

**X\_definitions** = load\_senses\_elements(method, Utils.DEFINITIONS)  
**X\_examples** = load\_senses\_elements(method, Utils.EXAMPLES)  
**X\_senses** = initialize\_senses(X\_definitions, X\_examples, average\_or\_random=**True**)  
**X\_globals** = torch.tensor(np.load(os.path.join(F.FOLDER\_INPUT, single\_prototypes\_file))).to(torch.float32)

**However**, an adjacent global can be either a synonym or an antonym.

The edge types are: 0=definitions, 1=examples, 2=senseChildren, 3=synonyms, 4=antonyms.

A case can be made for the removal of antonyms:

In GraphAttentionNetworks, the new node state is computed with an attention on the state of the neighbours: However, I do not strictly need to remove antonyms: their projection matrix can multiply their coordinates by -1, or make them negative in general, to “push” away the node from it.

Implementing…

grapharea\_matrix.shape=(114324, 256)

**Issue**:

Edge\_index and edge\_type are wrong,

e.g tensor([ 56, 59766, 87829, 43741], device='cuda:3') has

edge\_index = tensor([[ 56, 59766, 87829],

[ 56, 59766, 87829]], device='cuda:3')

and edge\_type= tensor([ 56, 59766, 87829], device='cuda:3')

The grapharea\_matrix is correct: we have [1,2,3]->[0,0,0] and [0,1,2]

Error located in unpack\_input\_tensor(…)

node\_projections\_matrices.shape=(5,300,300)

neighbours\_nodetypes=[0,2,1]

projected\_neighbours.shape=(300,3)

currentword\_embedding.shape=(1,300)

Must create node\_pairs using torch.cat

Note: maybe an initialization of the matrices with relatively small vaues could be beneficial? Idk.

**Issue:**

node\_attention\_state = (sum([attention\_coefficients\_alpha[i]\*projected\_neighbours[:,i] for i in range(projected\_neighbours.shape[1])])).unsqueeze(0)

**IndexError**: invalid index of a 0-dim tensor. Use tensor.item() to convert a 0-dim tensor to a Python number

We should always have a non-zero number of projected neighbours, because one of them should be the current node itself.

Reviewing graph edges…

In DefineGraph, we have sc\_edges.extend(edges\_selfloops)

From the log:

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

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

Error due to:

current\_node\_index = tensor(30337, device='cuda:0')

INFO : projected\_neighbours.shape=torch.Size([300, 1])

INFO : attention\_coefficients\_alpha=tensor(**1.,** device='cuda:0', grad\_fn=<SoftmaxBackward>)

# 11: Adding the Graph Networks in practice

## GATs with different node types

### Experiment 1

Parameters:

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

('module.select\_first\_node', torch.Size([1]), False), ('module.node\_types', torch.Size([114324]), False), ('module.node\_projections\_matrices', torch.Size([5, 300, 300]), True),

('module.memory\_h1', torch.Size([1, 300]), False), ('module.memory\_h2', torch.Size([1, 300]), False),

('module.A.weight', torch.Size([1, 600]), True),

('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([31640, 300]), True), ('module.linear2global.bias', torch.Size([31640]), True)]

Number of trainable parameters=45.35M

Sotfmax=9.52M. Embeddings=34.3M. Core=1.52M

Keeping the hyperparameters that brought the 2-layers GRU to have 199 validation perplexity:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| batch\_size | 80 |  | learning rate | 10^(-4) |
| graph area | <= 64, 1-hop |  | layers | 2 |
| TBPTT length | 35 |  | hidden state dim. | 300,300 |
| Dropout | 0.1 on GRU W1,W2 |  | GRU | full, on both |
| Attention | 1 node, manual on different node types |  |  |  |

|  |  |  |
| --- | --- | --- |
| *Epoch* | *Training perplexity* | *Validation perplexity* |
| 1 | 1216.73 | 683.59 |
| 2 | 893.43 | 683.42 |
| 3 | 885.82 | 685.73 |
| 4 | 882.44 | 687.92 |
| 5 | 879.94 | 690.7 |

Possibilities:

* Changing hyperparameters: batch size to 20, and learning rate to 10^(-5)
* Adding Multi-head attention
* Changing architecture:  
  currently we have:   
  GATN on current node and immediate neighbours ->   
  GRU gates and hidden state 1 ->  
  GRU gates and hidden state 2 ->  
  FF-NN to logits (as usual)

## Examining possible modifications

### Graph\_area hops

Currently, I am including in the graph\_area only the immediate neighbours of the current node.

Let us consider the case:   
starting from the current global.  
At 1-hop: synonyms, antonyms, senses.

It can be considered opportune to adjust the senses based on definitions and examples.   
This would necessitate 2-hops, i.e. applying the GAT on the 2-hops graph\_area that includes not only synonyms, antonyms and senses, but also:  
- the senses’ definitions and examples  
- the nyms’ senses

**Issue**: the grapharea\_matrix

old\_graphareamatrix.shape=(114324, 6176)

Remember that the nodes’ indices are absolute, in order to index\_select on X, while the edges are relative to the local area.

In the current implementation of the Queue, I can not control the hops, only the grapharea-size or the number of neighbours of the original node that I add.

Must fix: GraphArea.get\_indices\_area\_toinclude(edge\_index, edge\_type, node\_index, area\_size, max\_hops)

Edges added per area: int(area\_size \*\* 1.5)

64 > 512; 32 -> 181.

### Senses and input signal

For predicting the next token, the input and starting point should not include only the current global. If we are operating on a Sense-Labeled Corpus, we should be able to read the sense at the same time.

The sense signal should give a contribution to the prediction whenever possible.

We do not necessarily have the sense specification for subsequent words. Thus, we should not keep a continuous GRU on it.

As we process new words, the input from the last sense should be less important / decay over time…

Hypothesis: (hidden\_state\_1\_from\_globals || hidden\_state\_from\_sense) \* W --> hidden\_state\_1 [final dimensionality]

whenever there is no sense input, the hidden\_state\_from\_sense will be zeros, giving no contribution.

However, the input from the senses is a part of the overall question:

how to organize the input from additional sources (KB graph, senses) in such a way that we provide useful information to the LM task (instead of adding only noise to the word prediction that simply uses the word embedding alone)?

As of now, trying to replace the word embedding with the new node state from the GAT over the graph area only introduced noise:

Alternatives:

1. ~~Still~~ **~~replace~~** ~~the word embedding with the current word’s node state from the graph. Try out RGCNs and multi-head attention.  
   2 sources: projection from global’s node-state || sense’s node state~~
2. 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

## GATs – basic

### Experiment 1 – GAT, 1-hop

Instead of using the manually built Graph Attention Network, that has different projection matrices for the different kinds of nodes, we use the standard pytorch-geometric GAT, that does not differentiate between the different kinds of nodes and edges, since its forward(…) call takes only (x, edge\_index)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| batch\_size | 80 |  | learning rate | 10^(-4) |
| graph area | <= 64, **1-hop** |  | layers | 2 |
| TBPTT length | 35 |  | hidden state dim. | 300,300 |
| Dropout | 0.1 on GRU W1,W2 |  | GRU | full, on both layers |
| Attention | 1 node, standard GAT, 1 head |  |  |  |

|  |  |  |
| --- | --- | --- |
| *Epoch* | *Training perplexity* | *Validation perplexity* |
| 1 | 1170.98 | 546.51 |
| 2 | 635.45 | 445.05 |
| 3 | 541.61 | 405.25 |
| 4 | 492.43 | 380.18 |
| 5 | 458.04 | 360.84 |
| 6 | 429.71 | 343.27 |
| 7 | 404.53 | 327.45 |
| 8 | 382.11 | 313.68 |
| 9 | 361.51 | 301.11 |
| 10 | 343.23 | 290.26 |
| 11 | 327.46 | 281.13 |
| 12 | 313.46 | 273.15 |
| 13 | 301.18 | 266.26 |
| 14 | 290.22 | 260.46 |
| 15 | 280.34 | 255.26 |

Stopped. We can expect a final validation PPL around 220 at best, not improving on the 199 PPL of a 2-layer GRU.

It appears that by replacing the current word’s global Fastext embedding with a standard GAT on a 1-hop graph\_area we have just added noise.

### Experiment 2 – GAT, 2-hops

Standard pytorch-geometric GAT.

Using a graph\_area that has the nodes up to 2 hops away, as the input to the GAT.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| batch\_size | **40** |  | learning rate | **0.5 \* 10^(-4)** |
| graph area | <= 64, **2-hops** |  | layers | 2 |
| TBPTT length | 35 |  | hidden state dim. | 300,300 |
| Dropout | 0.1 on GRU W1,W2 |  | GRU | full, on both layers |
| Attention | standard GAT, 1 head |  |  |  |

|  |  |  |
| --- | --- | --- |
| *Epoch* | *Training perplexity* | *Validation perplexity* |
| 1 | 1146.51 | 556.16 |
| 2 | 633.23 | 447.54 |
| 3 | 542.92 | 409.26 |
| 4 | 497.75 | 385.16 |
| 5 | 465.67 | 366.44 |
| 6 | 439.49 | 349.93 |
| 7 | 416.21 | 334.62 |
| 8 | 394.79 | 320.98 |
| 9 | 375.71 | 309.83 |
| 10 | 359.18 | 300.34 |
| 11 | 345.31 | 292.59 |
| 12 | 333.27 | 285.74 |
| 13 | 322.73 | 279.47 |
| 14 | 313.1 | 274.0 |
| 15 | 304.34 | 269.09 |
| 16 | 296.31 | 264.49 |
| 17 | 288.68 | 260.21 |
| 18 | 281.67 | 256.35 |
| 19 | 275.17 | 252.73 |
| 20 | 268.93 | 249.5 |
| 21 | 263.24 | 246.28 |
| 22 | 257.8 | 243.47 |
| 23 | 252.63 | 240.79 |
| 24 | 247.8 | 238.28 |
| 25 | 243.22 | 235.97 |
| 26 | 238.8 | 233.87 |
| 27 | 234.67 | 231.97 |
| 28 | 230.75 | 230.23 |
| 29 | 227.07 | 228.54 |
| 30 | 223.54 | 227.12 |
| 31 | 220.24 | 225.8 |

It seems that *replacing* the word embedding with the graph signal from a 2-hops neighbourhood introduces noise. 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**.

### Experiment 3 – GAT, input signals

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).

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Training hyperparameters** | | **Input signals & GNN** | | **Architecture** | |
| batch\_size | 80 | grapharea | 64 | GRU layers | 2 |
| learning rate | 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\*150 |  |  |

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

**Issue**:

The process is killed because, operating with a larger model, I am breaking memory. This happens mostly because of the grapharea\_matrix, which is a dense matrix where the majority of the elements are -1.

grapharea\_matrix.shape=(114324, 1600) with a total of 182,918,400 elements.

**Memory profiling**:

(Using an excel doc.)

Last snapshot before process was killed – during validation:

INFO : Partition of a set of 1255508 objects. Total size = 1.634Million bytes. Of which ~1,469Million are for the numpy grapharea\_matrix.

Possible route: use a sparse pandas Dataframe. To actually benefit, we should specify the compressed value is not 0 but -1. The datatype should not be Sparse[float64, 0.0] but Sparse[float64, -1].

It becomes too slow if I do this. Originally, 1 iteration for a batch of 80 elements used ~6.5/7s. With a sparse Dataframe:

|  |  |  |
| --- | --- | --- |
| *Epoch* | *Training perplexity* | *Validation perplexity* |
| 1 | (1146.51) | (556.16) |
| 2 |  |  |
| 3 |  |  |
| 4 |  |  |
| 5 |  |  |
| 6 |  |  |
| 7 |  |  |
| 8 |  |  |
| 9 |  |  |
| 10 | (359.18) | (300.34) |
| 11 |  |  |
| 12 |  |  |
| 13 |  |  |
| 14 |  |  |
| 15 |  |  |
| 16 |  |  |
| 17 |  |  |
| 18 |  |  |
| 19 |  |  |
| 20 | (268.93) | (249.5) |

# 12: Batch normalization

## Introduction

Originally, when batch normalization was introduced, it was believed that it could mitigate the problem of internal covariate shift:  
During the training stage of networks, as the parameters of the preceding layers change, the distribution of inputs to the current layer changes accordingly, such that the current layer needs to constantly readjust to new distributions.

In 2018, researchers have found that batch normalization does not reduce internal covariate shift, but rather smooths the objective function to improve the performance.

## Method and observations

Batch normalization can be implemented during training by calculating the mean and standard deviation of each input variable to a layer per mini-batch and using these statistics to perform the standardization.

**Caveat:**

For small [mini-batch sizes](https://machinelearningmastery.com/difference-between-a-batch-and-an-epoch/) or mini-batches that do not contain a representative distribution of examples from the training dataset, the differences in the standardized inputs between training and inference (using the model after training) can result in noticeable differences in performance.

This can be addressed with a modification of the method called Batch Renormalization (or BatchRenorm for short) that makes the estimates of the variable mean and standard deviation more stable across mini-batches.

**Tip**:

Batch normalization may be used on the inputs to the layer *before* or *after* the activation function in the previous layer.

It may be appropriate **before** the activation function for activations that may result in non-Gaussian distributions like the rectified linear activation function, the modern default for most network types.

**Consequence**:

The network is more stable during training.

We can use higher learning rates because batch normalization makes sure that there is no activation that has gone really high or really low. (n: it is opportune to increase the decay rate for the learning rate, as well).

**Note**:

Further, it may not be a good idea to use batch normalization and dropout in the same network.

The reason is that the statistics used to normalize the activations of the prior layer may become noisy given the random dropping out of nodes during the dropout procedure.

## Necessity of Batch Renormalization

From a discussion on the PyTorch forum:

“For the past few days, I’ve been training a model that uses batch normalization. While this normalization is crucial to speed up training, performance drops severely once I switch to eval instead of train mode…

The problem seems to be caused by the fact the running estimates are not reliable when using small batch sizes. For a lot of problems (e.g. segmentation), however, increasing the batch size is not feasible due to memory constraints…

The authors of the batch normalization paper acknowledged this issue and wrote a follow-up paper about batch renormalization, a similar technique which should also work with smaller batches.

I was wondering if there were any plans to implement this batch renormalization in PyTorch?”

Answer:” You could always try instance norm (unless you have really few features per channel), which takes away the difference between training and evaluation…”