[1: Meeting (IA, 24/01/20) 2](#_Toc31195993)

[2: Attempting Batching 2](#_Toc31195994)

[Using DataLoader 3](#_Toc31195995)

[Issues 3](#_Toc31195996)

[Speedup: 5](#_Toc31195997)

[3: Experiments and Hyperparameters 6](#_Toc31195998)

[3.1: Full overfit on mini-dataset 6](#_Toc31195999)

[3.1.1. 6](#_Toc31196000)

# 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.

Modifications needed:

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

### 3.1.2

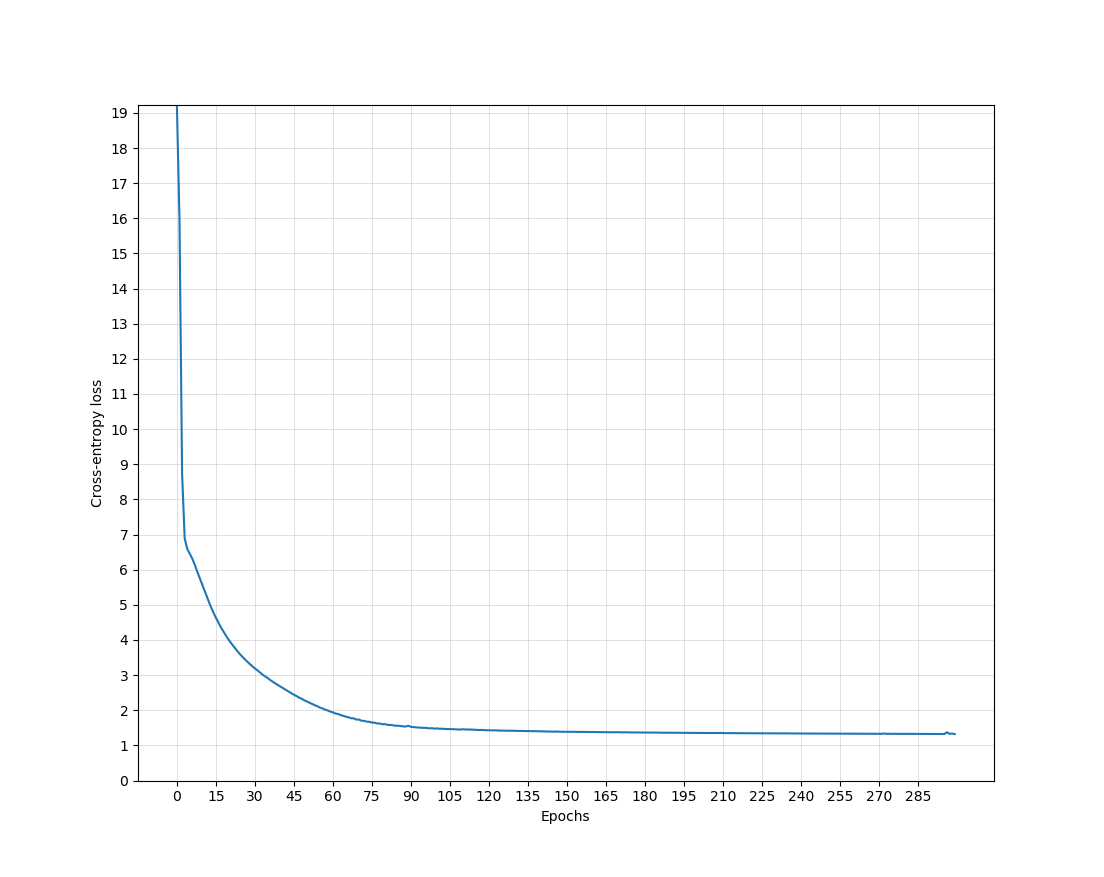
|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| batch size | 8 |  | **training tokens** | **176** |
| graph\_area | 32 |  | **epochs** | **300** |
| learning rate | 0.001 |  | final global step | 6600 |
|  |  |  | token-steps | 52800 |



Training, epoch nll\_loss= 2.21

### 3.1.3

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| batch size | 8 |  | **training tokens** | **88** |
| graph\_area | 32 |  | epochs | 300 |
| **learning rate** | **0.003** |  | final global step | 3300 |
|  |  |  | token-steps | 26400 |



Training, epoch nll\_loss= 1.32218

Raising further 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 ~1.3

I proceed to try 2 more experiments in this series, with a decay schedule for the learning rate, and with a greater graph\_area size (from 32 to 64)

# 4: 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…”