SAPIENZA
UNIVERSITÀ DI ROMA

Engineering in Computer Science, Neural Networks course

Stochastic Training of Graph Convolutional Networks with Variance Reduction

December, ay 2019/20

Authors:

[Umberto di Canito – 1851685, Daniele Buonadonna –1855047]

Contents

1

Introduction

In this project [6], we analyzed the performance and behavior of a **GCN (Graph Convolutional Network)**. All the project is based on the paper ' `Stochastic Training of Graph Convolutional Networks with Variance Reduction`' [2] (*Jianfei Chen, Jun Zhu, Le Song)*

In particular, in the original paper was been develop control variate based algorithms which allow sampling an arbitrarily small neighbor size; this because previous attempts on reducing the receptive field size by subsampling neighbors do not have a convergence guarantee, and their receptive field size per node is still in the order of hundreds.

About the dataset, the original paper is based on six different ones but we tested only on *Cora*, *PubMed* and *Citeseer*. Moreover, the original paper proved new theoretical guarantee for our algorithms to converge to a local optimum of GCN.

Graph convolution networks (GCNs) generalize convolutional neural networks (CNNs) to graph structured data. The "graph convolution" operation applies same linear transformation to all the neighbors of a node, followed by mean pooling and nonlinearity.

By stacking multiple graph convolution layers, GCNs can learn node representations by utilizing information from distant neighbors. GCNs and their variants have been applied to semi-supervised node classification, inductive node embedding, link prediction and knowledge graphs, outperforming multi-layer perceptron (MLP) models that do not use the graph structure, and graph embedding approaches that do not use node features.

2

Design aspects

2.1 Neural Networks architecture

In this section we will explain the design aspects we followed in order to carry on the same experiment the authors did. To work and build every type of neural network, from LTSM Neural Network to RNN and to implement the Convolutional Neural Network structure.

To work on the project we used *GoogleColab* that is an useful tool from Jupiter Project [9], and the main libraries used are:

– `PyTorch` [7] to manage tensors, networks and the activation functions

– `DGL` [3] to manage and manipulate the dataset and for retrieving of informations

– `MatplotLib` [5] to build graphical representation of the results

– `Numpy`, `GoogleColab`, `Warnings`, `Random` and others to do other useful things as tools.

2.2 Activation functions

We implemented one architecture on which we based our test. It is a GCN structure provided from DGL website [4]. In order to achieve the results on experiments, we tested the architecture by training them on `Cora`, `PubMed`, `Citeseer` and by varying the linear function.

At the end, we found that the architecture selected from DGL performed significantly better in all the experiments, independently from the linear function. Furthermore, the architecture suggested from DGL is made up of 2 layers, as proposed in the original paper.

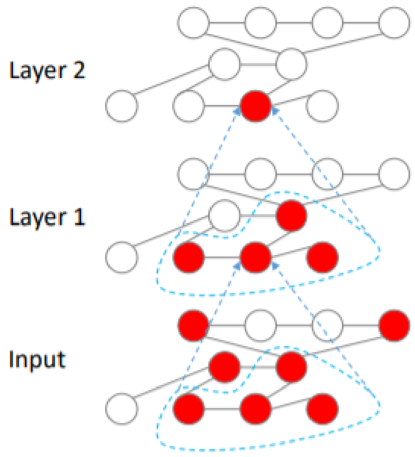

Figure 2.1: Layers represented in the original paper

With the control variate algorithm the layers representation with the activation/historical nodes is the following:

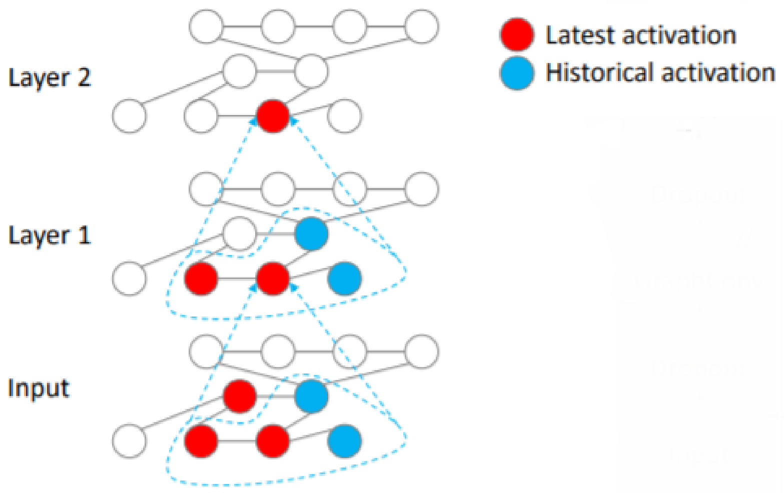

Figure 2.2: Layers represented in the original paper with control variate algorithm

2.2.1 Linear activation function

Linear functions are used in order to retrieve the linear transformation of the layers. In particular for a generic classification problem, the linear function is computed like:

$$y = xW^T + b \tag{2.1}$$

where W and b are the vectors initially randomized that represents **weights** and **biases**, respectively.

When we talk about the datas on graphs, we need to consider the structure of graph itself. This is done by considering the *propagation* of the nodes. The linear function became like following:

$$Z^{l+1} = PH^{(l)}W^{(l)} \tag{2.2}$$

where:

- P is the **propagation matrix** and it is computed as follows:

$$P = \hat{D}^{-\frac{1}{2}}\hat{A}\hat{D}^{-\frac{1}{2}} \tag{2.3}$$

  better explaining:

  – $\hat{A}$ is the *adjacency matrix* with I: $\hat{A} = A + I$
  – $\hat{D}$ is the *diagonal matrix* of degrees: $\hat{D}_{uu} = \sum_{v=1}^{n} \hat{A}_{uv}$

- H is the *non-linear function of* Z

- W is the matrix of *learning parameters*

In the original paper there is a proposal of new kind of linear function, that make use of variations on the parameters of original formulation. This is done in order to improve efficiency in time consumption terms. So the original formulation became as following:

$$Z^{(l+1)} = \left(\hat{P}^{(l)}(H^{(l)} - \hat{H}^{(l)}) + P\hat{H}^{(l)}\right) W^{(l)} \tag{2.4}$$

Again, better explaining:

- $\hat{P}^{(l)}$ is the propagation matrix at layer l, computed on a minibatch of nodes

- $\hat{H}^{(l)}$ is the resultant matrix from stacking of historical activation $\hat{h}_v$

2.2.2 ReLU: Non-linear function

ReLU (rectified linear unit) [8] is an activation function defined as the positive part of its argument:

$$f(x) = x^+ = max(0, x), \tag{2.5}$$

where x is the input to a neuron. This is also known as a ramp function and is analogous to half-wave rectification in electrical engineering.

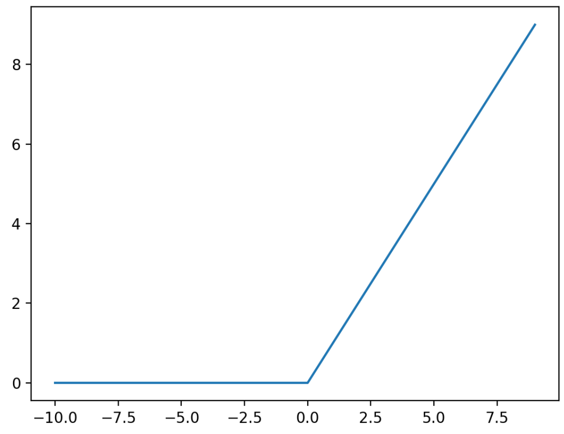

Figure 2.3: ReLU plot

In the original paper, non-linear function is used to computed H^{l+1} as following:

$$H^{l+1} = \sigma(Z^{l+1}) \tag{2.6}$$

In the code implementation that we developed the non-linear function is the ReLU function.

3

Dataset

In the original paper all the algorithms have been tested on six graph datasets with these characteristics:

Dataset	V	E	Degree	Degree 2
Citeseer	3327	12431	4	15
Cora	2708	13264	5	37
PubMed	19717	108365	6	60
NELL	65755	318135	5	1597
PPI	14755	458973	31	970
Reddit	232965	23446803	101	10858

Table 3.1: Original datasets used. From left to right: number of vertexes, edges, average number of 1-hop and 2-hop neighbors per node for each dataset. Undirected edges are counted twice and self-loops are counted once

Moreover, originally have been used for all six the same train/validation/test splits. In our case we decided to use just `Citeseer`, `PubMed`, `Cora` since hardware constraints.

In particular, to retrieve the informations of datasets we used the `DGL` library. Following a snippet example of code in which we take the informations about *Cora* dataset:

```python
from dgl.data import citation_graph as citegrh
import networkx as nx
def load_cora_data():
    data = citegrh.load_cora()
    features = th.FloatTensor(data.features)
    labels = th.LongTensor(data.labels)
    mask = th.ByteTensor(data.train_mask)
    g = data.graph
    print(g)
    # add self loop
    g.remove_edges_from(nx.selfloop_edges(g))
    g = DGLGraph(g)
    g.add_edges(g.nodes(), g.nodes())
    return g, features, labels, mask
#get data
g, features, labels, mask = load_cora_data()
```

4

Implementation and experimental results

In the original paper, was developed novel control variate-based stochastic approximation algorithms for GCN. Moreover, were utilized the historical activations of nodes as a control variate. What was shown is that while the variance of the NS estimator depends on the magnitude of the activation, the variance of their algorithms only depends on the difference between the activation and its historical value.

Furthermore, the algorithms bring new theoretical guarantees. At testing time, they give exact and zero-variance predictions, and at training time, their algorithms converge to a local optimum of GCN regardless of the neighbor sampling size D(l). The theoretical results allow they to significantly reduce the time complexity by sampling only two neighbors per node, yet still retain the quality of the model.

4.1 Control Variate Based Algorithm

This algorithm utilizes *historical activations* to reduce the estimator variance. Their main idea was to maintain the history $\overline{h}_v^{(l)}$ for each $h_v^{(l)}$ as an affordable approximation. Each time when $h_v^{(l)}$ is computed, we update $\overline{h}_v^{(l)}$ with $h_v^{(l)}$. What is expected is that $\overline{h}_v^{(l)}$ and $h_v^{(l)}$ to be similar if the model weights do not change too fast during the training.

So this is an estimator, called CV and if we consider $\overline{H}^{(l)}$ be the matrix formed by stacking $\overline{h}_v^{(l)}$, then CV can be written as:

$$\hat{P}^{(l)}(H^{(l)} - \overline{H}^{(l)}) + P\overline{H}^{(l)} \tag{4.1}$$

Going into the details of implementation, it is composed by 5 steps:

1. Randomly select a minibatch $V_B \in V_L$ of nodes;
2. Build a computation graph that only contains the activations $h_v^{(l)}$ and $\overline{h}_v^{(l)}$ needed for the current minibatch;
3. Get the predictions by forward propagation as 4.1

4. Get the gradients by backward propagation, and update the parameters by SGD;
5. Update the historical activations.

Observations
The computational graph at Step 2 is defined by the receptive field $r^{(l)}$ and the propagation matrices $\hat{P}^{(l)}$ at each layer. The receptive field $r^{(l)}$ specifies the activations $h_v^{(l)}$ of which nodes should be computed for the current minibatch, according to 4.1.

In the original paper it was assumed that $h_v^{(l)}$ always needed to compute $h_v^{(l+1)}$ i.e., v is always selected as a neighbor of itself. The receptive fields are illustrated in the figure 2.2, where red nodes are in receptive fields, whose activations $h_v^{(l)}$ are needed, and the histories $\overline{h}_v^{(l)}$ of blue nodes are also needed.

So the pseudo code is the following:

```
for each minibatch VB in V do
    Compute the receptive fields and stochastic propagation
   matrices
as Alg. 1.
    (Forward propagation)
    for each layer $l \leftarrow 0$ to $L-1$ do
        $Z^{(l+1)} \leftarrow \left(\hat{P}^{(l)}(H^{(l)} - \overline{H}^{(l)}) + P\overline{H}^{(l)}\right) W^{(l)}$
        $H^{(l+1)} \leftarrow \sigma(Z^{(l+1)})$
    end for
    compute the loss $L = \frac{1}{|V_B|} \prod_{v \in V_B} f(y_v, Z_v^{(L)})$
    (Backward propagation)
    $W \leftarrow W - \gamma_i \nabla_W L$
    (Update historical activations)
    for each layer $l \leftarrow 0$ to $L-1$ do
        for each node $v \in r^{(l)}$ do
            $\hat{h}_v^{(l)} \leftarrow h_v^{(l)}$
        end for
    end for
end for
```

Listing 4.1: Training with the CV algorithm

The `Alg.1` is implemented as follow:

```
$r^{(l)} \leftarrow V_B$
for layer $l \leftarrow L-1$ to 0 do
    $r^{(l)} \leftarrow \emptyset$
    $\hat{P}^{(l)} \leftarrow 0$
    for each node $u \in r^{(l+1)}$ do
        $r^{(l)} \leftarrow r^{(l)} \cup \{u\}$
        $\hat{P}_{uu}^{(l)} \leftarrow \hat{P}_{uu}^{(l)} + P_{uu}^{(l)} n(u)/D^{(l)}$
        for $D^{(l)} - 1$ random neighbors $v \in n(u)$ do
            $r^{(l)} \leftarrow r^{(l)} \cup \{v\}$
            $\hat{P}_{uv}^{(l)} \leftarrow \hat{P}_{uv}^{(l)} + P_{uv}^{(l)} n(u)/D^{(l)}$
        end for
    end for
end for
```

Listing 4.2: Constructing the receptive fields and random propagation matrices.

4.2 Learning rate choice

During training, an adaptive optimizer [1] is being used to *adapt* the learning rate (η) during the training phase. Although, the choice of the learning rate took a long time for us, since it affect greatly the performances. For each data set, we run some short learning cycles with four different learning rate ($10^{-1}, 10^{-2}, 10^{-3}, 10^{-4}$) in order to test the performances. Since the stochastic nature of the GCN, multiples tries were done. The graph below shows the results of one try, regarding the CiteSeer data set, showing the accuracy score referred to the order of magnitude of the learning rate.

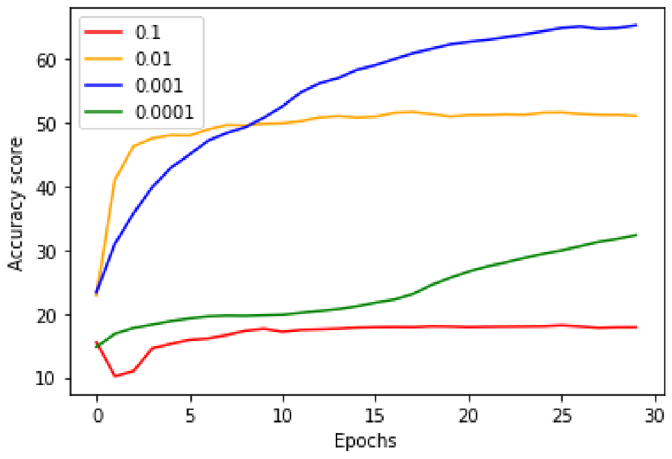

Figure 4.1: Accuracy score at maximum 30 epochs for CiteSeer data set

The same studies were done referred to the loss values, but they were unsatisfactory due to the fact that, often, the loss trends of 10^{-2} and 10^{-3} were overlying, not showing a defined pattern.
At the end, we made the following decision for the three different data set:

- Cora $\eta = 2 \cdot 10^{-2}$
- CiteSeer $\eta = 2 \cdot 10^{-3}$
- PubMed $\eta = 2 \cdot 10^{-3}$

4.3 Results and discussion

What we had done is to use the approximation proposed by [2] and compare the scores of accuracy and loss computed by a standard GCN and one GCN using the control variate estimator (CV).

4.3.1 Cora data set

For what concerns the Cora data set, we used a GCN with a depth of 2, the first one with a width of 16 and the last (the output layer) with a width of 7, starting from an input layer of 1433 nodes. Multiples learning attempts were done and then the average was made in order to retrieve a more satisfactory trend. As shown in figures 4.2 and 4.3, both accuracy and loss trends were similar, even if they differ by a few points.

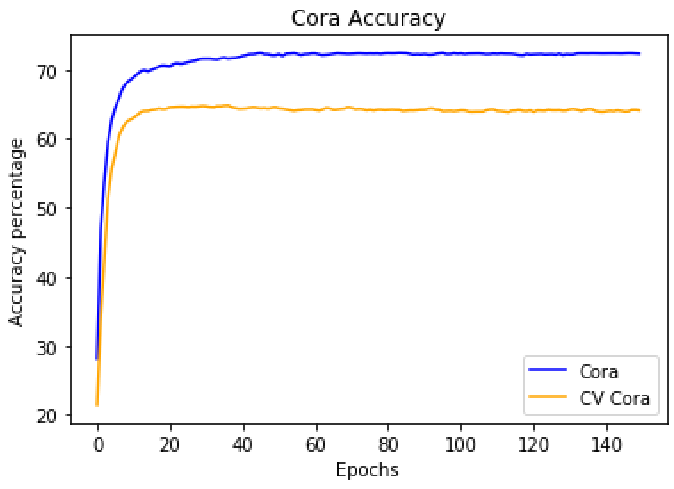

Figure 4.2: Accuracy score for the Cora dataset

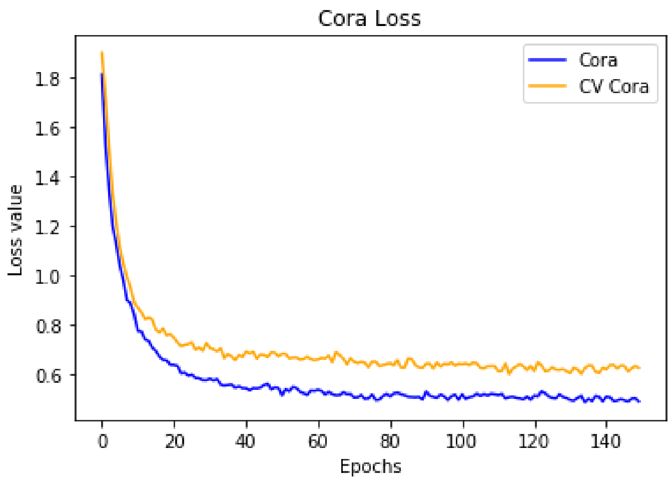

Figure 4.3: Loss score for the Cora dataset

4.3.2 CiteSeer data set

Regarding the CiteSeer data set, we used again a GCN with a depth of 2, the first one with a width of 40 and the last (the output layer) with a width of 6, starting from an input layer of 3703 nodes. In figure 4.4 and figure 4.5, it's possible to notice that the *CV* line (the orange one) is more noisy than the pattern shown by the standard GCN (the blue one). This is due to the fact that, for hardware and time limitations, we were not able to perform the same number of tries for both models.

For the *CV* variant, in fact, were made less attempts, leaving noisy data affects the plot.

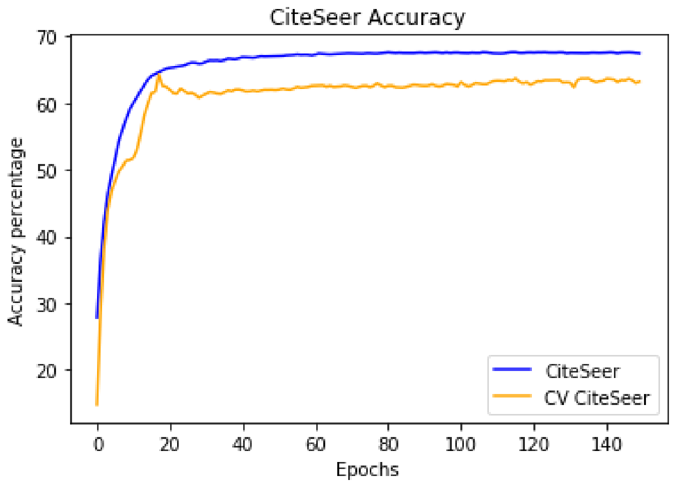

Figure 4.4: Accuracy score for the CiteSeer dataset

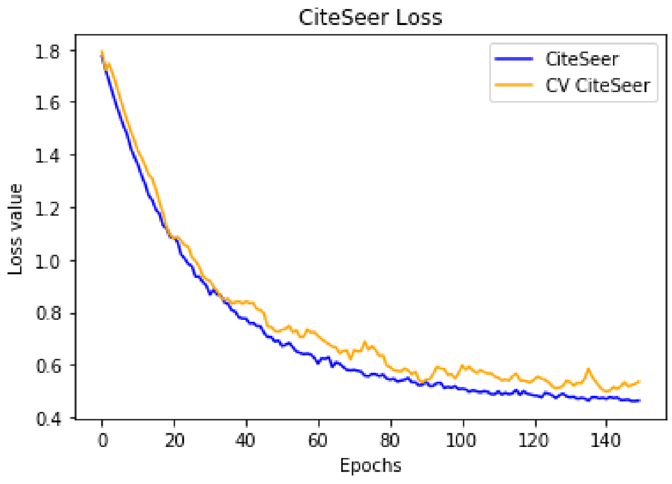

Figure 4.5: Loss score for the CiteSeer dataset

4.3.3 PubMed data set

Again, we used a GCN with 2 hidden layers, the first one with a width equal to 16 and last equal to 3. Each node of the PubMed citation graph has a feature vector of 500 entries. As shown in both figure below, the *CV* values of accuracy and loss were really near the standard GCN values. It is also possible to see that these scores were obtained in a quite small number of epochs. This time, in fact, we needed to reduce the number of epochs due to hardware limitations since the initial graph was containing 19717 nodes, a lot more respect the 2708 of Cora and the 3327 of CiteSeer.

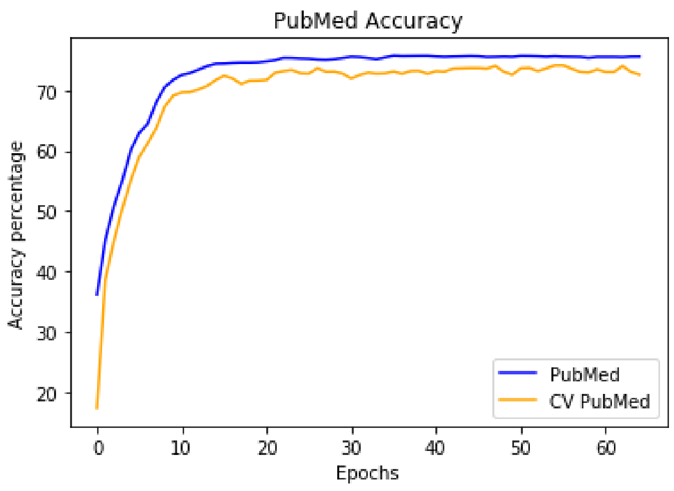

Figure 4.6: Accuracy score for the PubMed dataset

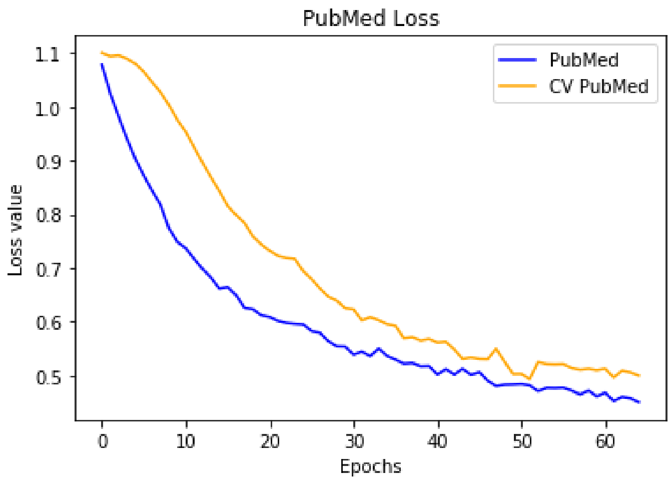

Figure 4.7: Loss score for the PubMed dataset

5

Comparison and further improvements

Looking at the results retrieved from all the three data sets, it is possible to see that the algorithm performs better on PubMed than on Cora or CiteSeer data. This is probably due to the dimension of PubMed, about six times greater than the other two respect to the number of nodes. Since what we have done is, indeed, an approximation, we are not surprised at all from this.

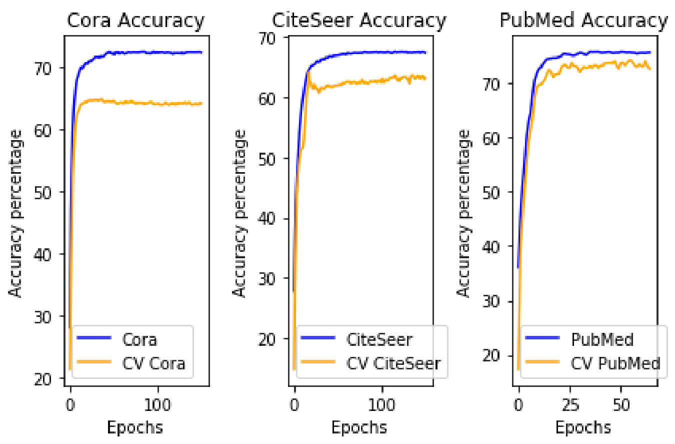

Figure 5.1: Comparison of accuracy between the data sets

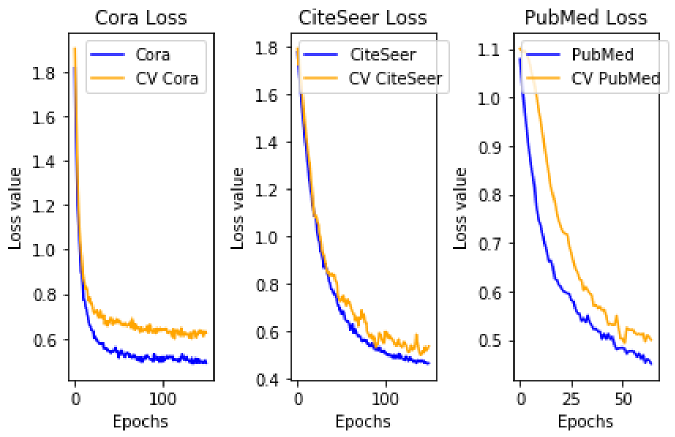

Figure 5.2: Comparison of loss values between the data sets

What could have been done is to try our implementation of the algorithm with greater datasets like *NELL*, *PPI* or *Reddit*. Although, this was not possible for us, because of hardware limitation.
The previous results were computed using GoogleColab environment, that gives us an average of 12 GB of RAM memory and 358 GB of disk memory available. Everything was coded in Python language.

Bibliography

[1] *Adam Optimizer*. URL: https://pytorch.org/docs/stable/_modules/torch/optim/adam.html#Adam.

[2] Jianfei Chen, Jun Zhu, and Le Song. *Stochastic Training of Graph Convolutional Networks with Variance Reduction.* 2017. arXiv: 1710.10568 [stat.ML].

[3] *DGL*. URL: https://docs.dgl.ai/en/latest/index.html.

[4] *DGL (example)*. URL: https://docs.dgl.ai/en/latest/tutorials/models/1_gnn/1_gcn.html.

[5] *MatplotLib*. URL: https://matplotlib.org.

[6] *Project on GitHub repositary*. URL: https://github.com/umbertodicanito/Stochastic-Training-of-Graph-Convolutional-Networks-with-Variance-Reduction.

[7] *PyTorch*. URL: https://pytorch.org.

[8] *ReLU*. URL: https://en.wikipedia.org/wiki/Rectifier/_(neural_networks).

[9] Wikipedia. *Project Jupyter*. [Online; checked in 27.11.2019]. URL: https://en.wikipedia.org/wiki/Project/%5C_Jupyter.