ENSIAS

Defense of the graduation project
Software Engineering Department

ORACLE Labs

Implementation of GraphSage in PGX

Presented by:

Ben Ayad, Mohamed Ayoub

Supervised by:

- Mr **Rhicheek,** Patra
- Mrs **Chiadmi**, Dalila
- Mr **Guermah,** Hatim

Jury:

- Mr **Baina,** Karim
- Mrs **Berrada,** Bouchra

University Year 2018-2019

Outline

- Project Context
- Graph Learning
- Model Design and Implementation
- Improving the model
- Evaluation of the model
- Limitations and difficulties
- Conclusion and future works

Project Context

Oracle Labs:

- Oracle Labs is the sole organization at Oracle that is devoted exclusively to **research**.

- Missions: **Identify**, **explore**, and transfer **new technologies** that have the potential to substantially **improve Oracle's** business.

- There are four major approaches:
  - **Exploratory research**
  - **Directed research**
  - **Consulting**
  - **Product incubation**

- Machine learning is widely used.
- No input => No learning

- DATA is the **core trigger** of every Machine learning process

- Classic (acceptable) inputs:
  - Tabular data
  - Vector representations of data
  - Images

- Problematic :
  - Text data
  - Graph data

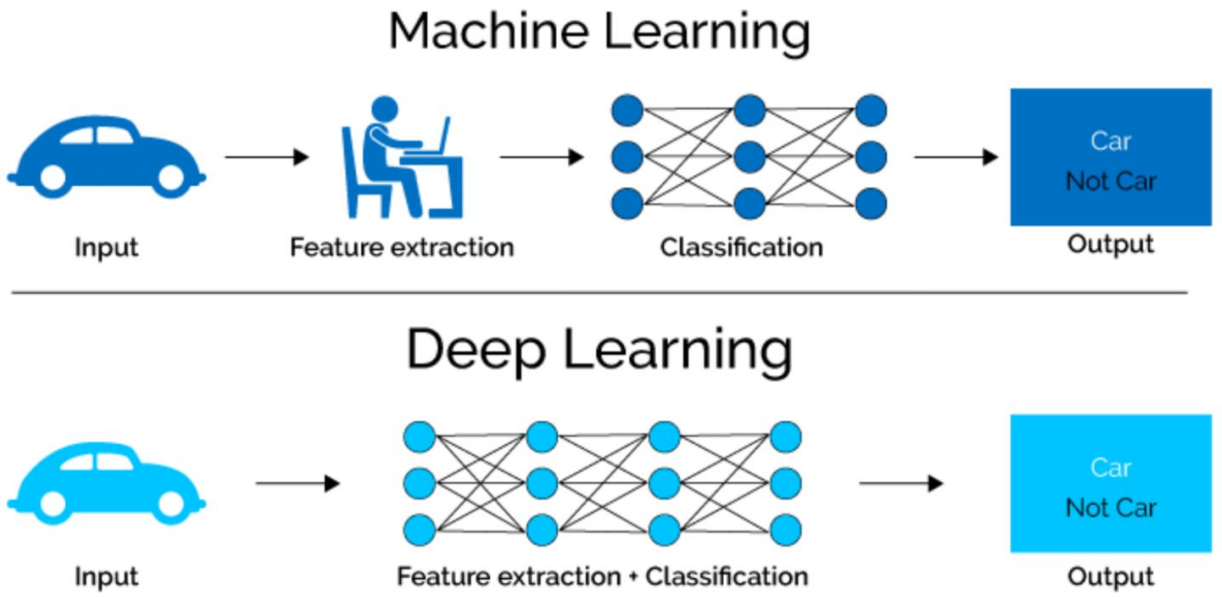


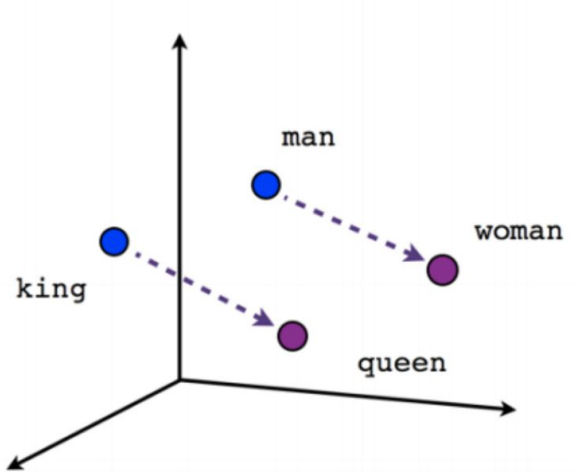
Figure 1: Machine Learning VS Deep Learning

How do we address the problem?

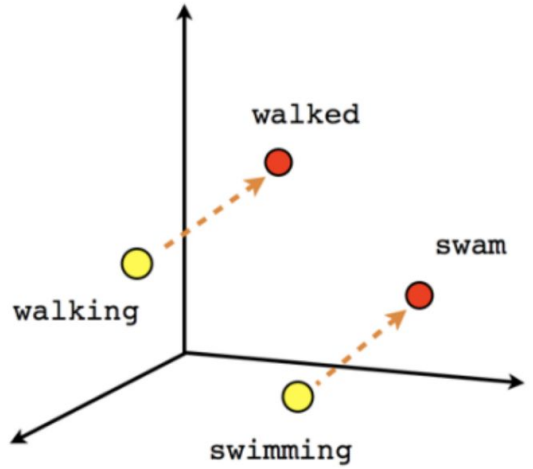
- Word embeddings
- Graph embeddings

What we are trying to do?

- Learn **low dimensional vector representation** of each word (similarly nodes)
- Mapping



Male-Female



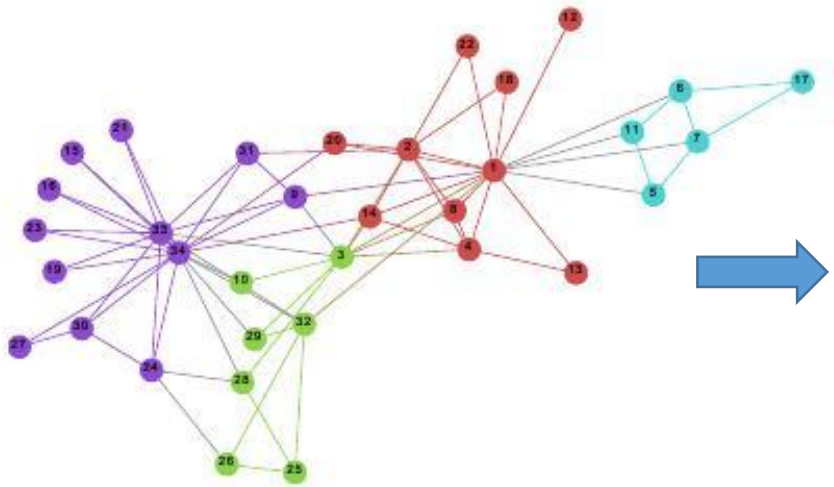
Verb tense

Difference between the two approaches:

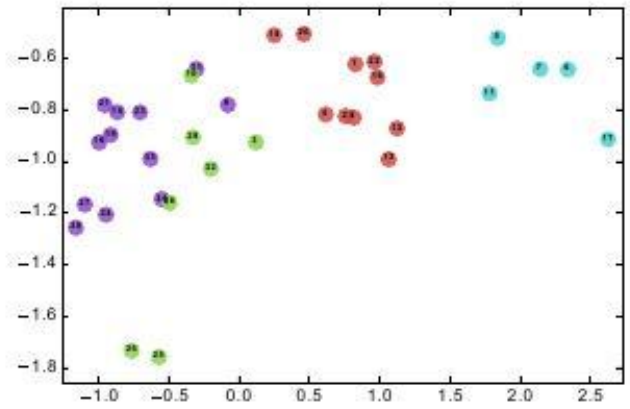
- Classic approach >> Manually features engineered
- Embedding approach >> **ML use**

- **Embedding = vector representation**
- **Classic structure = tabular (vector) data**

Generate embeddings of a graph = Graph Learning



Graph representation



Vector representation

Acceptable Input for machine learning models

From an **abstract representation** to an **Euclidean representation**

Why we favor Euclidean representations:

- We can feed vector representation we can feed to a machine learning model: **Classification**, **Prediction**, **Clustering** …

Project was held in within the PGX team

- **PGX** [**Parallel Graph AnalytiX]:** PGX is a toolkit for graph analysis - both *running algorithms* such as PageRank against graphs, and performing SQL-like *pattern-matching* against graphs, using the results of algorithmic analysis. Algorithms are parallelized for extreme performance.

- **What can you do with PGX?**
  - **Loading graphs from a variety of sources.**
  - **Applying graph pattern matching.**
  - **Running parallel, high-performance graph algorithms**: PGX provides built-in implementations of many popular graph algorithms.

- **General Idea of the project: Add a new Graph Learning ALGORITHM to the library**

New Graph Learning Model = GraphSage

- **Paper:** Inductive Representation Learning on Large Graphs, 2018

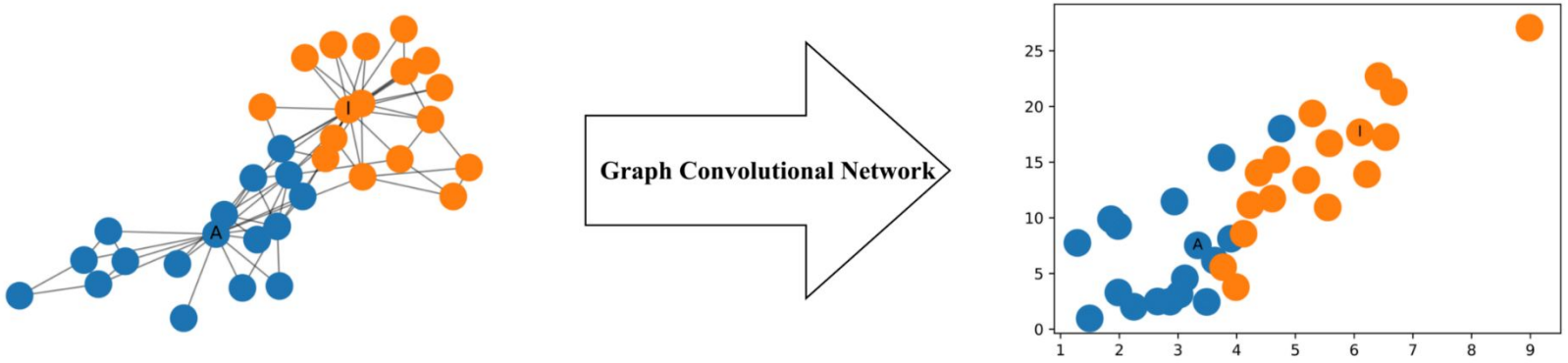
- To implement GraphSage in PGX:
  - To have GraphSage as **one the tools of PGX**

Objectives details:

- Only use **PGX** and **DL4J** for the implementation:
  - **PGX** for all graph computations
  - **DL4J** for all machine learning backend
- Implementing an **efficient** solution, accuracy wise
- Implementing a fast solution for **large-scale** datasets
- **Test**, **evaluate** and **improve** the implemented model

Graph Learning

- Node embeddings: Map nodes to low-dimensional embeddings.
- The majority of these methods are **transductive: The whole graph should be inputted.**
- Machine learning systems nowadays, operate on **evolving graphs** and **constantly encounter unseen nodes:**
  - ❑ **FACEBOOK Dataset: NEW USERS**



There are a lot of methods concerning Graph Learning:

- **Transductive approaches**, mostly based on matrix factorization, Like **DeepWalk**.
  - Model should be trained again each time we encounter a new node.

- **Inductive approaches**, like **GraphSage**:
  - Could be generalized to unseen nodes.

There is already an existing approach in PGX: **DeepWalk**

DeepWalk: Word embedding analogy

- **Generate sentences from graphs (Random walks)**
- **Use word embeddings models**

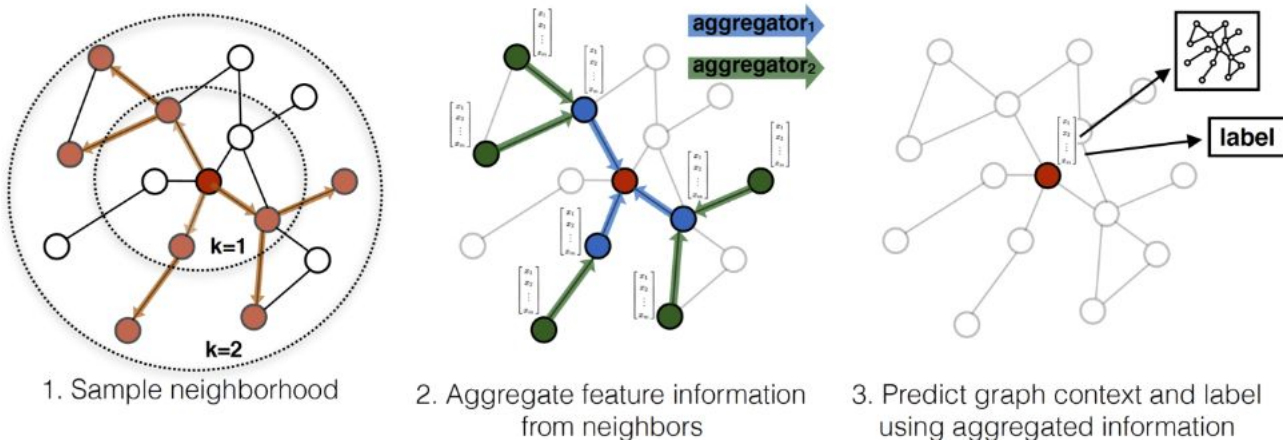
Limitations of DeepWalk:

- Generalization on unseen nodes
- Lacks support for attributed graphs >> INITIAL FEATURES (NODE ATTRIBUTES)

- It's not space-efficient, as a feature vector is learned for each node.
- No parameter sharing.
- Only trained in an unsupervised manner.

GraphSage: Sample And Aggregate

Intuition: Instead of training individual embeddings for each node, we learn a function that generates embeddings by sampling and aggregating features from a node's local neighborhood.

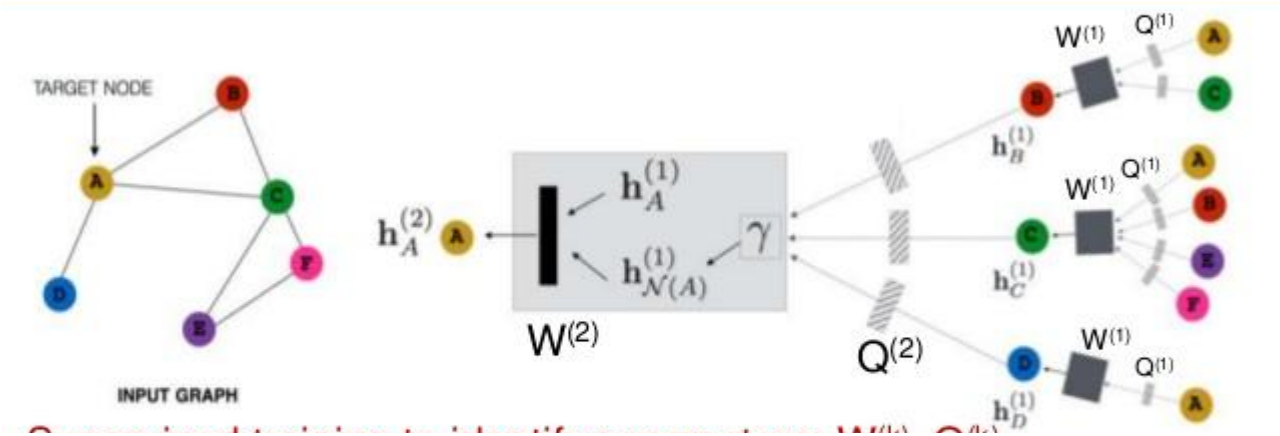


1. Sample neighborhood

2. Aggregate feature information from neighbors

3. Predict graph context and label using aggregated information

GraphSage: Sample And Aggregate



GraphSage: Forward Pass of the model

Algorithm 1: GraphSAGE embedding generation (i.e., forward propagation) algorithm

Input : Graph $\mathcal{G}(\mathcal{V}, \mathcal{E})$; input features $\{\mathbf{x}_v, \forall v \in \mathcal{V}\}$; depth K; weight matrices $\mathbf{W}^k, \forall k \in \{1, ..., K\}$; non-linearity σ; differentiable aggregator functions $\text{AGGREGATE}_k, \forall k \in \{1, ..., K\}$; neighborhood function $\mathcal{N} : v \rightarrow 2^{\mathcal{V}}$

Output: Vector representations $\mathbf{z}_v$ for all $v \in \mathcal{V}$

$\mathbf{h}_v^0 \leftarrow \mathbf{x}_v, \forall v \in \mathcal{V}$;
for $k = 1...K$ **do**
  for $v \in \mathcal{V}$ **do**
    $\mathbf{h}_{\mathcal{N}(v)}^k \leftarrow \text{AGGREGATE}_k(\{\mathbf{h}_u^{k-1}, \forall u \in \mathcal{N}(v)\})$;
    $\mathbf{h}_v^k \leftarrow \sigma\left(\mathbf{W}^k \cdot \text{CONCAT}(\mathbf{h}_v^{k-1}, \mathbf{h}_{\mathcal{N}(v)}^k)\right)$
  end
  $\mathbf{h}_v^k \leftarrow \mathbf{h}_v^k / \|\mathbf{h}_v^k\|_2, \forall v \in \mathcal{V}$
end
$\mathbf{z}_v \leftarrow \mathbf{h}_v^K, \forall v \in \mathcal{V}$

Why GraphSage is Better

- **Inductive** approach, it can make predictions on unseen nodes or even sub-graphs.
- It takes into consideration the nodes **initial features**.
- Uses **parameter sharing**.
- It **scales very** well.

>> This is why we chose to GraphSage add to PGX algorithms stack.

What we tried to do here is to reproduce the Paper's benchmark shown below.

Table 1: Prediction results for the three datasets (micro-averaged F1 scores). Results for unsupervised and fully supervised GraphSAGE are shown. Analogous trends hold for macro-averaged scores.

	Citation		Reddit		PPI	
Name	Unsup. F1	Sup. F1	Unsup. F1	Sup. F1	Unsup. F1	Sup. F1
Random	0.206	0.206	0.043	0.042	0.396	0.396
Raw features	0.575	0.575	0.585	0.585	0.422	0.422
DeepWalk	0.565	0.565	0.324	0.324	—	—
DeepWalk + features	0.701	0.701	0.691	0.691	—	—
GraphSAGE-GCN	0.742	0.772	**0.908**	0.930	0.465	0.500
GraphSAGE-mean	0.778	0.820	0.897	0.950	0.486	0.598
GraphSAGE-LSTM	0.788	0.832	**0.907**	**0.954**	0.482	**0.612**
GraphSAGE-pool	**0.798**	**0.839**	0.892	0.948	**0.502**	0.600
% gain over feat.	39%	46%	55%	63%	19%	45%

- Using Pytorch's implementation, and PGX to generated DeepWalk embeddings, we got the same results:

- Using Cora dataset:
  - A citation dataset (2807 vertices, 5632 edges) and 7 classes

Approach	F1-Score
Raw features	0.769
GraphSage (PyTorch+NetworkX)	0.853
GraphSage (DI4J+PGX)	--
DeepWalk + Raw Features	0.782
DeepWalk (PGX)	0.596

Implementation

PGX:

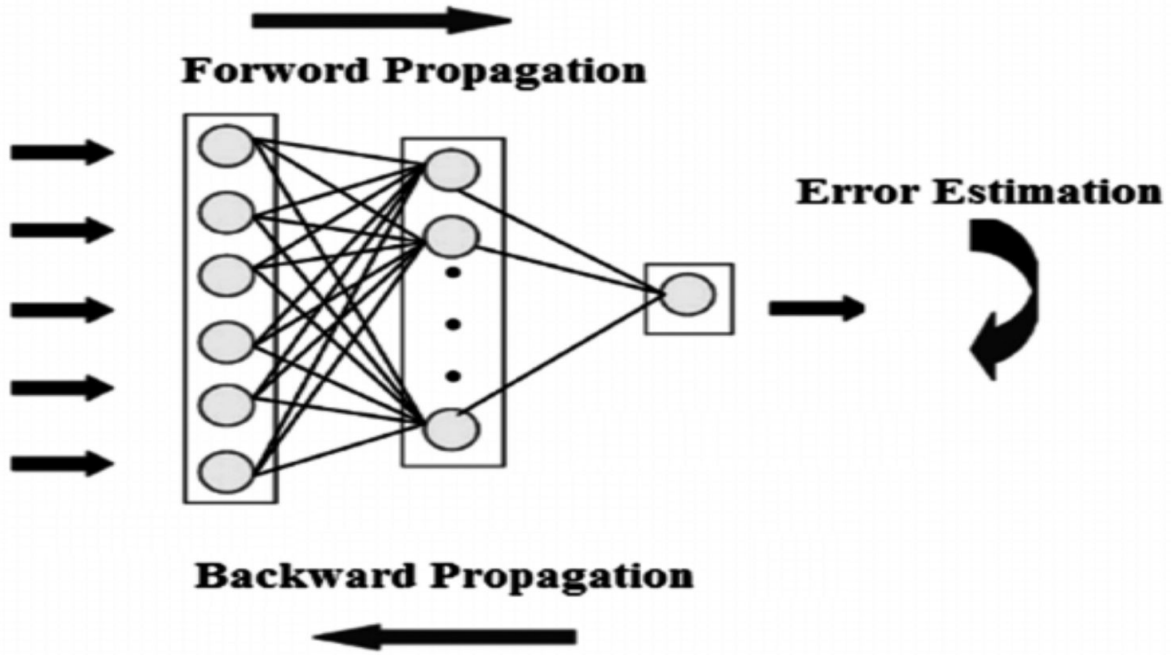
- All graph computations, extracting neighbors, extracting features.

DL4J: For the machine learning backend.

- DL4J is open source.

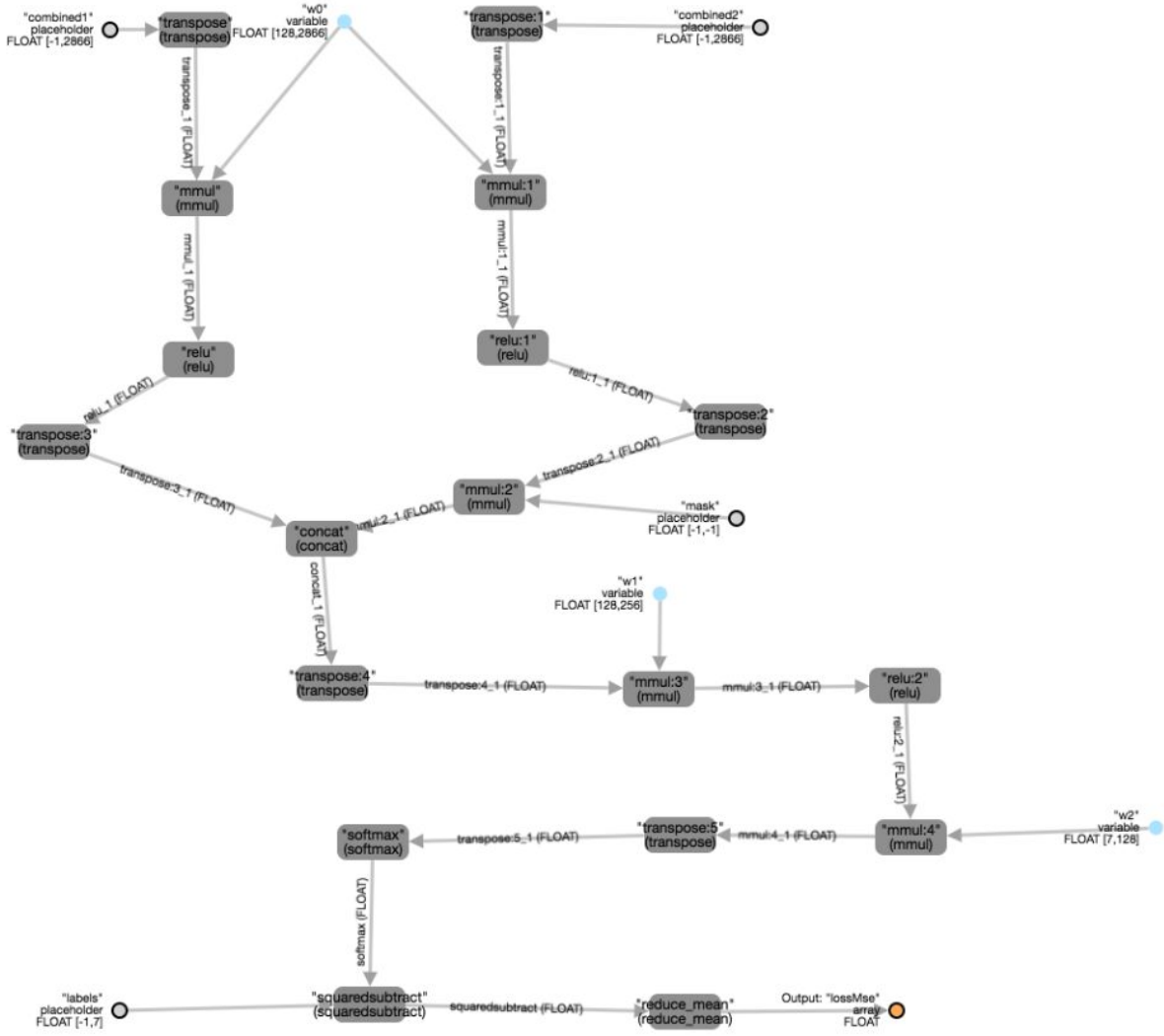
Like every Deep Learning Project, there are 3 major parts:

1. Forward propagation.
2. Backward Propagation.
3. Updating the weights.



DL4J Computation Graph

(page 36)



Improving the performances

The effect of each technique is recorded in the report:

- **Loss functions:** Functions used to evaluate our model
  - **Softmax Cross Entropy**
  - **Sigmoid Cross Entropy**

- **Activation functions:** They decide where a neuron or not should be activated or not
  - **ReLu**
  - **LeakyRelu**

- **Optimizers:** They are used to update the weights
  - **ADAM**
  - **ADAGRAD**

- **Batch normalization:** We normalize the output of each layer

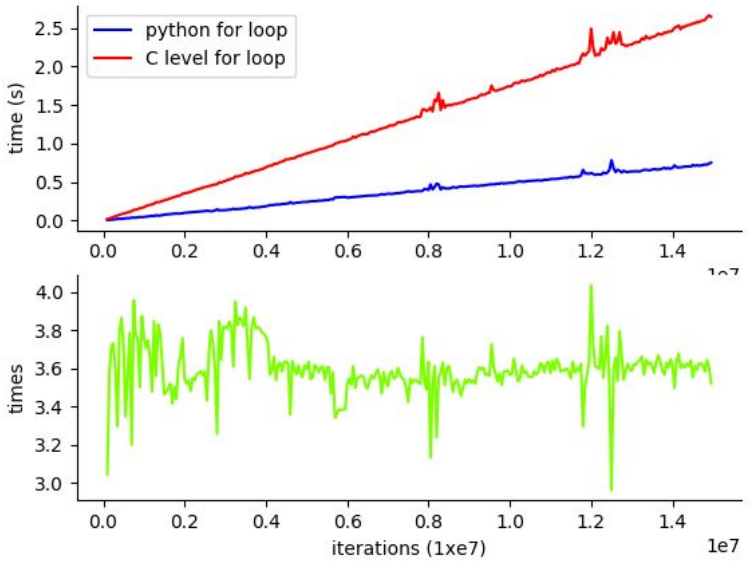
- **Weight initialization :** XAVIER initialization is used in the model

Parallelization:

- The application is fully parallelized for maximum performances
- The more CPU power, the more it would be faster

Vectorization:

- Using C levels loops instead of the using actual loops



- **Batching for inference**: Process data in chunks and then concatenate the results

Batch Size	Training + Inference Time (1 epoch)
128	34m 5s
1024	42m 2s

- **Sampling the neighbors**: sub-sample the set of neighbors of each used nodes

Using Numpy and Pandas for preprocessing:

- PANDAS DataFrames are very
- Highly suitable for big datasets
- >> We used PANDAS for all preprocessing steps <<
- **USE CASE:**
  - Split a 200k & 50 M graph dataset in 7 minutes,
  - The normal approach would take approximately 11 H

Model Evaluation

Reddit dataset: (large scale dataset)

Nodes: ~ 200K | Edges: ~ 50 M

Number of initial features: 602

Number of classes: 41

Split ratio : 70 / 30

- Train: Nodes: 163 075 | Edges: 23 230 993
- Test: Nodes: 69 890 | Edges: 47 396 905

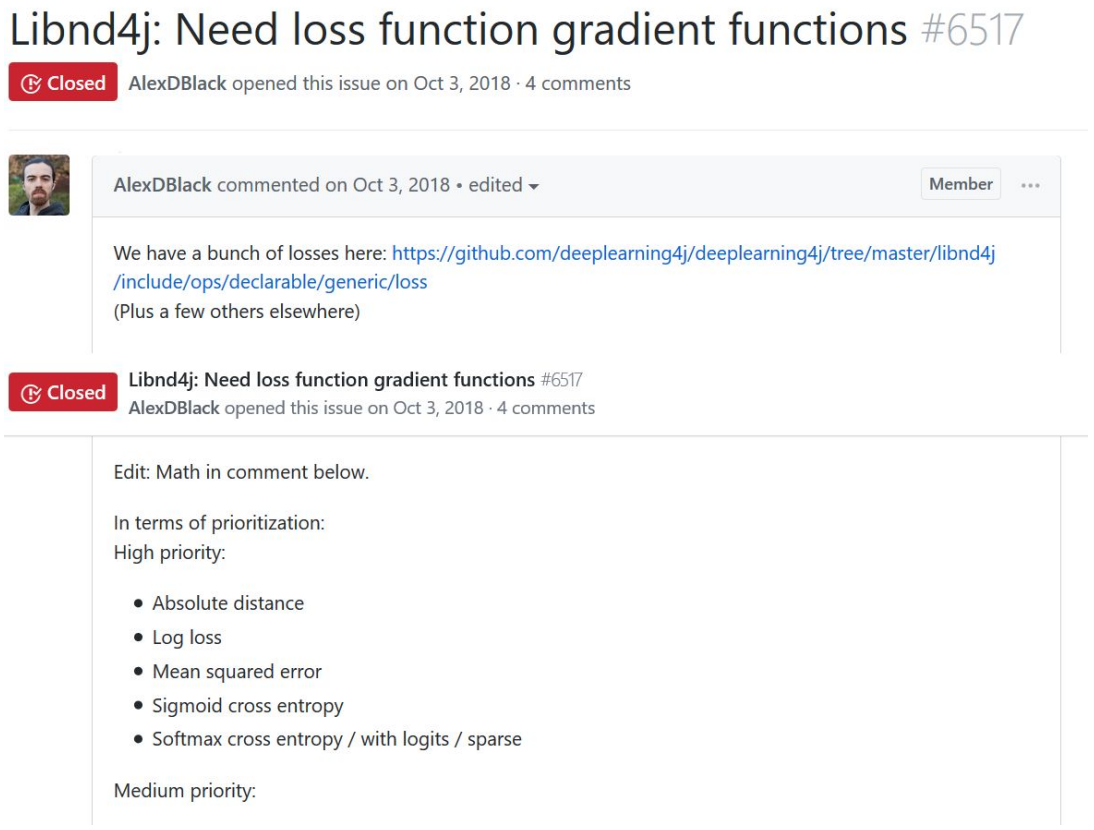
Comparisons results:

F1 score of the classification task

DeepWalk (PGX)	0.35
DeepWalk + Raw features	0.7
GraphSage (PGX)	0.8

Limitations and difficulties

- DL4J is not stable, it still in beta versions.
- Quick anecdote: There was a new release during this period.
- Still evolving.



Libnd4j: Need loss function gradient functions #6517

Closed AlexDBlack opened this issue on Oct 3, 2018 · 4 comments

AlexDBlack commented on Oct 3, 2018 • edited — Member

We have a bunch of losses here: https://github.com/deeplearning4j/deeplearning4j/tree/master/libnd4j/include/ops/declarable/generic/loss
(Plus a few others elsewhere)

Closed **Libnd4j: Need loss function gradient functions** #6517
AlexDBlack opened this issue on Oct 3, 2018 · 4 comments

Edit: Math in comment below.

In terms of prioritization:
High priority:

- Absolute distance
- Log loss
- Mean squared error
- Sigmoid cross entropy
- Softmax cross entropy / with logits / sparse

Medium priority:

- SameDiff is the entry point to ND4J's **autodiff**.
- The backward pass is fully done through Samediff.
- **Beta3 Vs Beta4**

Beta3 Vs Beta4 SameDiff improvements

- Removed reliance on periodic garbage collection calls for handling memory management of out-of-workspace (detached) INDArrays (Link)
- Added INDArray.close() method to allow users to manually release off-heap memory immediately (Link)
- SameDiff: Added TensorFlowImportValidator tool to determine if a TensorFlow graph can likely be imported into SameDiff. Reports the operations used and whether they are supported in SameDiff (Link)
- Added Nd4j.createFromNpzFile method to load Numpy npz files (Link)
- Added support for importing BERT models into SameDiff (Link, Link)
- Added SameDiff GraphTransformUtil for performing transfer learning and other graph modifications (Link, Link, Link)
- Evaluation, RegressionEvaluation etc now support 4d (CNN segmentation) data formats; also added Evaluation.setAxis(int) method to support other data formats such as channels-last/NHWC for CNNs and NWC for CNN1D/RNNs. Defaults to axis 1 (which matches DL4J CNN and RNN data formats) (Link, Link)
- Added basic ("technology preview") of SameDiff UI. Should be considered early WIP with breaking API changes expected in future releases. Supports plotting of SameDiff graphs as well as various metrics (line charts, histograms, etc)

GraphSage: **Neural Networks** Vs **Computation Graphs**

Computation Graphs:

- Multiple inputs at each level.
- Different shapes for each batch pass.
- Debugging more difficult.

>> Make you dive deep and code more things from scratch

Conclusion

- **Prototype is done:**
  - Efficient
  - Fast
  - Tested
  - Improved

- Add complex aggregators
- Add more modularity
- Implementation of the unsupervised version
- Use GraphSage instead of DeepWalk

Thank you for your time!