Contents lists available at ScienceDirect

SoftwareX

journal homepage: www.elsevier.com/locate/softx

Original software publication

JGNN: Graph Neural Networks on native Java

Emmanouil Krasanakis [*], Symeon Papadopoulos, Ioannis Kompatsiaris

Centre for Research and Technology Hellas, 57001 Thermi, Thessaloniki, Greece

ARTICLE INFO

Article history:
Received 31 October 2022
Received in revised form 29 March 2023
Accepted 23 June 2023

Keywords:
Graph neural networks
Machine learning
Edge computing

ABSTRACT

We introduce JGNN, an open source Java library to define, train, and run Graph Neural Networks (GNNs) under limited resources. The library is cross-platform and implements memory-efficient machine learning components without external dependencies. Model definition is simplified by parsing Python-like expressions, including interoperable dense and sparse matrix operations and inline parameter definitions. GNN models can be deployed on smart devices and trained on local data.

Code metadata

Current code version	*1.1.0*
Permanent link to code/repository used for this code version	https://github.com/ElsevierSoftwareX/SOFTX-D-22-00351
Permanent link to reproducible capsule	
Legal code license	*Apache License, 2.0*
Code versioning system used	*git*
Software code language	*Java*
Compilation requirements, operating environments and dependencies	*none*
If available, link to developer documentation/manual	https://github.com/MKLab-ITI/JGNN/blob/main/README.md
Support email for questions	maniospas@iti.gr

1. Motivation and significance

Graph Neural Networks (GNNs) have become a widely acclaimed machine learning paradigm that combines node features with relational information, and many frameworks have been created to integrate them in GPU computing [1–4]. However, contrary to typical neural networks, GNNs can also learn well from small scale data, such as tens of training labels spread throughout graphs of a few hundred nodes. At the same time, graph shift operators, which propagate node representations to neighbors, do not necessitate GPUs to be efficiently computed. Thus, GNN computational demands are easier to meet with low-end computing compared to other machine learning paradigms.

Consequently, and given the trend of converting other types of data (e.g., images) to graphs so that data mining turns into graph mining [5–7], GNNs could supply artificial intelligence to a variety of less computationally savvy environments, such as power-efficient smart devices. There, they could not only perform inference, but also learn from local data, such as private annotations of device users. In addition to supporting other types of data mining, GNNs on the edge could also analyze raw relational data, for instance to perform fake news detection personalized to the relation contexts of social media application users [8]. This is not supported by existing technologies, whose design spaces cover either resource-intensive computing or non-GNN edge computing (Section 2).

To fill this technological gap, we developed a native Java library, called *JGNN*, to define, train, and run neural networks and GNNs under limited resources. Machine learning components are implemented from scratch to be memory efficient and avoid external dependencies, for example that would require hardware- and platform-specific binaries. Thus, the library is lightweight to include in software packages and is supported by any device hosting the Java Virtual Machine (JVM), such as Android smartphones. It also trains architectures under limited memory and can define them with a few lines of Java code by parsing Python-like expressions.

* Corresponding author.
E-mail addresses: maniospas@iti.gr (Emmanouil Krasanakis), papadop@iti.gr (Symeon Papadopoulos), ikom@iti.gr (Ioannis Kompatsiaris).

https://doi.org/10.1016/j.softx.2023.101459

2. Scientific and technological context

2.1. Graph neural networks

Graph neural networks (GNNs) account for relational information between data samples by organizing the latter as nodes of graphs and exchanging representations between neighbors; they start from representation matrices $H^{(0)}$ whose rows are either node features or end-to-end trained embeddings, define operations that pool (e.g., average) neighbor representations, and pass these through dense transformations. Operations are typically written in matrix form and engage symmetric normalizations $A = D^{-1/2}A_{bin}D^{-1/2}$ of binary graph adjacency matrices A_{bin}, where D are diagonal matrices of node degrees. Often, the renormalization trick [9] adds numerical stability with self-loops to all nodes, and feature or edge dropout improve training robustness.

To understand prospective usage of the library, we present example GNNs. GCN [9] comprises two 64-dimensional layers:

$$H^{(\ell+1)} = \sigma(AH^{(\ell)}W^{(\ell)})$$

where $W^{(\ell)}$ are trainable layer weights and $\sigma(x)$ an activation applied element-by-element on matrix elements — typically chosen as $relu(x) = \max\{x, 0\}$. To not oversmooth across edges, APPNP [10] introduces bias towards $H^{(0)}$ via 10 layers:

$$H^{(\ell+1)} = aAH^{(\ell)} + (1-a)H^{(0)}$$

for hyperparameter $a \in (0, 1)$ and 64-dimensional embeddings $H^{(0)}$ extracted from node features with a two-layer perceptron. Other approaches make architectures deeper [11], account for heterophily (dissimilar nodes linking to each other) [12], apply neighbor attention [13,14], or implement message passing [15,16]. JGNN supports such principles with operations that express them in matrix form.

2.2. Learning at the edge

Edge computing processes data in the devices generating them. There are three popular edge learning directions, all of which acknowledge limitations in processing power and available memory: *(a) Collaborative learning*, such as federated [17,18] and gossip [19,20] learning, creates local model fragments that collectively learn to make predictions similar to would-be centralized infrastructure. However, it tends to rely on high-spec machines (e.g., servers) for critical computations, such as sample gradient aggregations, is otherwise limited to naive (e.g., linear) models, and in most cases requires high device connectivity. *(b) Model compression* [21–23] deploys models that sacrifice some predictive power to fit device resources. Training and simplifying large models is resource intensive and only pre-trained solutions can be deployed. *(c) In-device training* of models is less popular, partially due to non-GNN learning focusing on high accuracy instead of resource management [24]. Existing tools are limited to either simple models (e.g., SVMs) or wrapping compiled C++ code that is difficult to re-purpose for GNNs. To our knowledge, as of writing, no other machine learning framework for Java –the base language of Android devices– fully supports backpropagation through sparse matrix operations needed to train GNNs.

Here, we position JGNN relative to libraries of similar capabilities. We do not address collaborative learning ones, as they are non-autonomous. The most important comparison to make is with *deeplearning4j* [25], which is the premier Java machine learning framework [26]. This is designed for distributed learning and dense matrix data. Therefore, and despite its maturity in other fields, such as in deploying foundational image processing models or training deep convolutional architectures, it does not support GNNs, which can only be made computationally tractable with sparse representations of adjacency matrices. Additionally, although not necessarily impractical, it requires platform-specific binary files that significantly increase application size.

Other lightweight Java libraries that integrate parameter learning, such as *Java-ML* [27] and *JSAT* [28], provide specific algorithms but not interfaces to build custom architectures. Furthermore, they eschew deep learning in favor of simpler paradigms that account for Java's computational speed limitations. Hence, GNN integration needs to start from scratch. All edge learning frameworks, including deeplearning4j, are also designed for training on independent data batches and not for the recurring GNN principle of passing both training and inference samples through architectures to let them exchange information via edges.

Finally, compression systems like Tensorflow-lite [29] can perform inference at the edge by building models in Python and porting them to JVM devices. However, only recent approaches address model customization on local data [30] and, even then, accommodating computationally intensive dense matrix operations limits support for end-to-end-training.

Overall, there is a lack of Java libraries supporting GNNs, despite (and perhaps due to) the excellence of existing solutions in other types of machine learning. Therefore, and given that Java is a popular language at the edge, we argue that JGNN covers an important solution space.

3. Software description

3.1. Software architecture

JGNN's code base comprises source code, examples, and documentation. The source code is organized into three major modules under the `mklab.JGNN` namespace: (a) `core` defines tensor and matrix primitives and related operations, (b) `nn` defines neural network components, and (c) `adhoc` provides high-level functionality, such as fast data loading and symbolic parsing to define neural architectures from textual descriptions. These packages are further split into the 16 sub-packages displayed in Fig. 1, which span 86 files and 8.7k lines of code.

Top-level packages define abstract base classes to be polymorphically extended in sub-packages. For example, `Tensor` and `Matrix` abstract classes reside in namesake files under `core` and are inherited by classes of the `core.tensor` and `core.matrix` sub-packages respectively. Source code is extensible (e.g., its implementation of neighbor attention requires less than 50 lines of code). All components are documented with Javadoc and verified with integration tests.

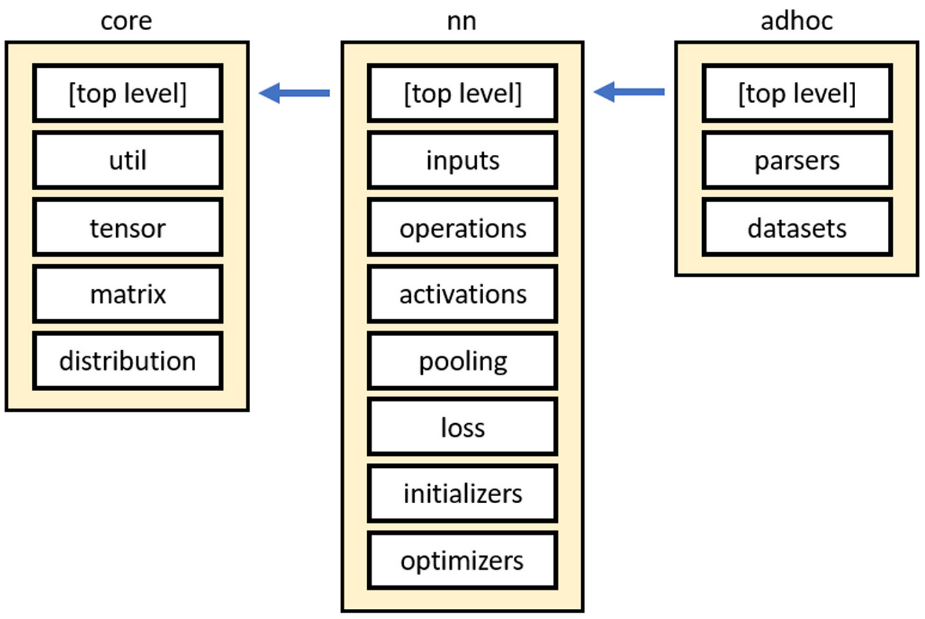

Fig. 1. JGNN packages, dependencies between them, and sub-packages.

3.2. Interoperable primitives

Core primitives define common arithmetic operations, such as element-by-element addition and multiplication, and in-place variations that save on memory allocation. Tensors and all derivative data types –including matrices– are handled with the same programming interfaces. To achieve interoperability between sparse and dense data, the library internally determines the type of arithmetic operation results based on ad-hoc criteria. For example, addition involving dense tensors always yields dense outputs, whereas the type of matrix multiplication outcome is determined via stochastic estimation of its expected density. When possible, JGNN reframes index traversal loops into iterating only over the minimum number of allocated memory elements.

Primitive dimensions can be named (`null` names are wildcards). Names are checked together with dimension sizes for logical integrity during operations and transferred to respective result dimensions. Finally, transparent data views eschew memory allocation. For example, `M.asTransposed()` returns a transposed view of a matrix `M`, whereas `M.accessRow(0).asColumn()` accesses the first row of a matrix (to use as a base `Tensor`) and reshapes it into an one-row column matrix without copying any data. Everything happens under the hood so that developers only need to think about the data types and dimension names of inputs.

3.3. Symbolic architecture definition

The `nn` package provides a large number of machine learning components that can be combined via Java code to define complex architectures, such as GNNs. To avoid hand-wiring data transfers, and therefore reduce errors and simplify usage patterns, the `adhoc` package provides model building mechanisms; these declare neural model parameters and hyperparameters, and construct models by parsing Python-like expressions. For instance, parsing the text

```
y=sigmoid(dropout(x, 0.5)@ matrix(features, classes, reg)+vector(classes))
```

creates all necessary components to apply dropout with rate 0.5 on an input matrix `x` with a number of columns equal to a declared hyperparameter `features`, perform a learnable dense transformation into `classes` dimensions by multiplying it with a parameter matrix, add a learnable vector bias term on each row, and pass the output through a sigmoid activation. Of shown operations, `@` performs matrix multiplication, the `matrix` method defines usage of an anonymous dense matrix variable with the provided dimensions and weight decay `reg`, and the `vector` method defines an anonymous learnable vector variable with the provided length. Both variables are constructed once and their elements are learned during neural training. Architecture hyperparameters, constant tensors, named parameters, input variables, and output variables are declared with appropriate method calls. Including the above, the library implements 35 operations to use in symbolic definitions.

A base `ModelBuilder` class parses GNN expressions, and two extensions add simplifications to architecture definitions. First, `LayeredBuilder` replaces *{l}* and *{l+1}* predicates with automatically increasing layer counters. Layers can be named (Section 4). This extension retains full control of input and output declarations (e.g., to consider adjacency matrices as inputs instead of constants). Second, `FastBuilder` supplies methods that further simplify popular component usage.

3.4. Additional features

Training machine learning architectures, such as those constructed with model builders, involves parsing (batch) inputs, desired outputs, loss functions, and optimizers, running over many epochs, and eventually selecting parameters based on validation data. JGNN provides a helper class called `ModelTraining` to define and run such procedures. This supports different losses between training and validation, as well as setting up early stopping based on stagnant validation loss.

To aid debugging, syntax errors are checked during parsing (e.g., to catch unknown variable names). Architecture diagnostics can print the routing between components, export execution graphs into graphviz format for visualization [31], and check for unused computational branches. Exceptions thrown for incompatible primitive dimension sizes or names are enriched by pointing to the neural components where errors occur and their inputs.

4. Illustrative example

We provide an example of defining, training, and testing the APPNP architecture [10] on node classification. Other architectures are obtained by modifying symbolic layer definitions, as detailed in the documentation. For experimentation, we load one of the datasets the library automatically downloads and obtain node features, labels, and an adjacency matrix. These could be manually constructed for new data. Using the library's matrix manipulation methods, diagonal elements are set to 1 to facilitate the renormalization trick and symmetric normalization is applied. Dimension names aid potential debugging.

```
Dataset dataset = new Datasets.Citeseer();
Dataset dataset = new Citeseer();
Matrix features = dataset
 .features()
 .setDimensionName("nodes", "features");
Matrix labels = dataset
 .labels()
 .setDimensionName("nodes", "labels");
Matrix adjacency = dataset.graph()
 .setMainDiagonal(1)
 .setToSymmetricNormalization()
 .setDimensionName("nodes", "nodes");
long numClasses = dataset.labels().getCols();
```

We use the FastBuilder to define the architecture for loaded data with a functional programming style that chains method calling. We first configure variable initialization hyperparameters, namely dimension sizes `features,hidden,classes` and the weight decay `reg`. FastBuilder starts from the input feature matrix, and we declare two dense layers. Then, the top layer is remembered as `0`, a diffusion constant `a` is defined, and the constant `A` holding the adjacency matrix –this is automatically declared by the FastBuilder– is used to diffuse `h{0}`. New layers are added when parsing `.layer` definitions, and `.layerRepeat` iterates the diffusion's definition 10 times to avoid repetition. The final layer is submitted for node classification with a namesake FastBuilder method; this applies the softmax transformation, sets a list node identifiers as the input, and keeps corresponding predictions.

```
ModelBuilder modelBuilder = new FastBuilder(adjacency, features)
 .config("reg", 0.005)
 .config("features", features.getDimensionSize("features"))
 .config("hidden", 64)
 .config("classes", numClasses)
 .layer("h{l+1}=relu(h{l}@matrix(features, hidden, reg)+vector(hidden))")
 .layer("h{l+1}=h{l}@matrix(hidden, classes)+vector(classes)")
 .rememberAs("0")
 .constant("a", 0.9)
 .layerRepeat("h{l+1} = a*(dropout(A, 0.5)@h{l})+(1-a)*h{0}", 10)
 .classify();
```

We now employ the `ModelTraining` class to automate model variable training with specific optimizers, loss functions, and number of epochs. More complex tasks, such as graph classification, can be implemented with the more generic `LayeredBuilder` and trained with manual calls to built model backpropagation.

```
ModelTraining trainer = new ModelTraining()
  .setOptimizer(new Adam(0.01))
  .setEpochs(300)
  .setPatience(100)
  .setLoss(new CategoricalCrossEntropy())
  .setValidationLoss(new CategoricalCrossEntropy());
```

To train the declared model, we retrieve it from its builder, initialize its variables, and provide to it the training strategy alongside a feature matrix input (for node classification, this is a column of all node identifiers), a matrix of corresponding labels, and slices of training and validation node indices. Slices collect row indices to gather from inputs and outputs, and can be constructed from fixed numerical ranges or collected from ranges within other shuffled slices.

Table 1
GNN training resources on the Cora dataset.

GNN	Tensorflow			JGNN			JGNN reduced		
	Acc	Time	Memory	Acc	Time	Memory	Acc	Time	Memory
GCN	76.8%	26 s	2,560 MB	87.5%	249 s	40 MB	87.5%	85 s	14 MB
APPNP	89.1%	43 s	2,304 MB	88.2%	306 s	40 MB	87.8%	117 s	28 MB

```
Slice nodes = dataset.samples().getSlice().shuffle(100); // shuffle seed 100
Model model = modelBuilder.getModel()
 .init(new XavierNormal())
 .train(trainer,
  nodes.sampleIdsAsFeatures(), // inputs = all node ids
  labels,
  nodes.range(0, 0.6),   // 60% of inputs for training
  nodes.range(0.6, 0.8)); // 20% of inputs for validation
```

We make predictions by running the model for the feature matrix input and casting the first and only output to the `Matrix` class (this is equivalent to Java's typecasting). We iterate over the last 20% of shuffled nodes left for testing, access each row of the predictions and corresponding labels, and check whether their maximum is found at the same position. Labels are sparse matrices, but knowing this detail is not necessary to write the testing code.

```
Matrix output = model
 .predict(nodes.samplesAsFeatures())
 .get(0)
 .cast(Matrix.class);
double acc = 0;
for(Long node : nodes.range(0.8, 1)) {
 Matrix nodeLabels = labels
  .accessRow(node)
  .asRow();
 Tensor nodeOutput = output
  .accessRow(node)
  .asRow();
 acc += nodeOutput.argmax()==nodeLabels.argmax()?1:0;
}
System.out.println("Acc\t "+acc/nodes.range(0.8, 1).size());
```

The implementation is concise, especially by Java standards. Not counting necessary `import` statements, the pipeline spans only 55 lines of code. Code writing is simplified thanks to symbolically parsing large parts of the architecture definition.

5. Impact

To our knowledge, JGNN is the first Java machine learning library that supports training of GNNs on edge devices. As such, estimating its impact on shaping future applications or research is difficult. For now, we acknowledge the emerging trend of moving data processing to the edge and point to the popularity of GNNs in centralized machine learning; the library adapts this paradigm to an emerging setting. We used early versions in decentralized social media applications [32], where we empirically corroborated the usefulness of graph learning at the edge.

As an indication of how few resources are consumed, we replicate the two GNN architectures of Section 2.1 for node classification and capture the time and minimum memory needed to train them within edge devices on the well-known Cora dataset [33], which comprises 2,708 nodes and 57,884 edges. We also explore lightweight GNN variations that reduce layer dimensions to the number of output classes. Due to lack of direct competitors, we compare JGNN with equivalent Tensorflow GPU implementations [34] written in Python.

Table 1 shows test set accuracy on a 60-20-20 train-validation-test split and resource consumption to train across 300 epochs. All implementations run on a 2.6 GHz CPU and DDR3 RAM. Tensorflow also leverages a 1.2 GHz GPU with 1,920 cores and DDR6 memory. Implementations exhibit similar accuracy and we attribute differences to our components running on 64-bit (instead of 32-bit) arithmetics. Overall, JGNN runs within 14-40 MB of memory, which is over 70 times less than Tensorflow. In edge applications, we expect usage with similar or a fraction of these data volumes, but JGNN is scalable; memory and computation demands are proportional to the number of edges plus the data volume of feature matrices. Given enough memory, graphs can comprise up to the JVM's maximum long number (typically $2^{64}-1$) of edges, with dense operations being restricted up to the maximum integer number (typically $2^{31}-1$) of nodes. GNN versions with reduced parameters are trained in time comparable to GPUs with minimal accuracy drop. Training with more layer dimensions is more time consuming due to lack of parallelization for dense data transformations; we do not leverage edge GPUs in favor of memory efficiency. Nonetheless, the library can take advantage of multiple processor cores for parallel batching on tasks that support it, like graph classification.

Concerning the library's portability, a JAR file holding cross-platform JVM instructions consumes 168 kB of disk space. *deeplearning4j*'s latest release stripped down to its basic neural network functionality and without external dependencies needs at least 5 MB of disk space for hardware-specific code, which doubles for deployment with cross-platform binaries.

6. Conclusions

In this work we presented JGNN, a lightweight native Java library to define, train, and run GNNs on the edge. This provides interoperable implementations of common sparse and dense matrix operations needed to define graph learning, and boasts concise programming interfaces. Well-performing yet lightweight architectures run in only a fraction of the memory needed by performance-centric frameworks. Programming interfaces are easy to use with a few lines of code. The library is deployed via Jitpack for integration in Maven and Gradle projects and its source code and extensive documentation are publicly available under the Apache 2.0 license at the GitHub repository https://github.com/MKLab-ITI/JGNN. Developers are encouraged to contribute to the code base.

Declaration of competing interest

The authors declare the following financial interests/personal relationships which may be considered as potential competing interests: Emmanouil Krasanakis reports that financial support was provided by the European Commission. Emmanouil Krasanakis reports a relationship with Centre for Research and Technology-Hellas that includes: employment.

Data availability

Data are appropriately cited and automatically downloaded upon running the example code snippets.

Acknowledgment

Development of the package was partially funded by the European Commission under contract number H2020-951911 AI4Media.

References

[1] Scarselli F, Gori M, Tsoi AC, Hagenbuchner M, Monfardini G. The graph neural network model. IEEE Trans Neural Netw 2008;20(1):61–80.
[2] Wang MY. Deep graph library: Towards efficient and scalable deep learning on graphs. In: ICLR workshop on representation learning on graphs and manifolds. 2019.
[3] Fey M, Lenssen JE. Fast graph representation learning with PyTorch Geometric. 2019, arXiv preprint arXiv:1903.02428.
[4] Ferludin O, Eigenwillig A, Blais M, Zelle D, Pfeifer J, Sanchez-Gonzalez A, et al. TF-GNN: graph neural networks in TensorFlow. 2022, arXiv preprint arXiv:2207.03522.
[5] Shekkizhar S, Ortega A. Graph construction from data by non-negative kernel regression. In: ICASSP 2020-2020 IEEE international conference on acoustics, speech and signal processing. ICASSP, IEEE; 2020, p. 3892–6.
[6] Hu H, Yao M, He F, Zhang F. Graph neural network via edge convolution for hyperspectral image classification. IEEE Geosci Remote Sens Lett 2021;19:1–5.
[7] Pradhyumna P, Shreya G, et al. Graph neural network (GNN) in image and video understanding using deep learning for computer vision applications. In: 2021 second international conference on electronics and sustainable communication systems. ICESC, IEEE; 2021, p. 1183–9.
[8] Dou Y, Shu K, Xia C, Yu PS, Sun L. User preference-aware fake news detection. In: Proceedings of the 44th International ACM SIGIR conference on research and development in information retrieval. 2021, p. 2051–5.
[9] Kipf TN, Welling M. Semi-supervised classification with graph convolutional networks. 2016, arXiv preprint arXiv:1609.02907.
[10] Klicpera J, Bojchevski A, Günnemann S. Predict then propagate: Graph neural networks meet personalized pagerank. 2018, arXiv preprint arXiv:1810.05997.
[11] Chen M, Wei Z, Huang Z, Ding B, Li Y. Simple and deep graph convolutional networks. In: International conference on machine learning. PMLR; 2020, p. 1725–35.
[12] Zhu J, Yan Y, Zhao L, Heimann M, Akoglu L, Koutra D. Beyond homophily in graph neural networks: Current limitations and effective designs. Adv Neural Inf Process Syst 2020;33:7793–804.
[13] Veličković P, Cucurull G, Casanova A, Romero A, Lio P, Bengio Y. Graph attention networks. 2017, arXiv preprint arXiv:1710.10903.
[14] Zhang J, Shi X, Xie J, Ma H, King I, Yeung D-Y. Gaan: Gated attention networks for learning on large and spatiotemporal graphs. 2018, arXiv preprint arXiv:1803.07294.
[15] Gilmer J, Schoenholz SS, Riley PF, Vinyals O, Dahl GE. Neural message passing for quantum chemistry. In: International conference on machine learning. PMLR; 2017, p. 1263–72.
[16] Feng J, Chen Y, Li F, Sarkar A, Zhang M. How powerful are k-hop message passing graph neural networks. 2022, arXiv preprint arXiv:2205.13328.
[17] Li X, Huang K, Yang W, Wang S, Zhang Z. On the convergence of fedavg on non-iid data. 2019, arXiv preprint arXiv:1907.02189.
[18] Li T, Sahu AK, Talwalkar A, Smith V. Federated learning: Challenges, methods, and future directions. IEEE Signal Process Mag 2020;37(3):50–60.
[19] Blot M, Picard D, Cord M, Thome N. Gossip training for deep learning. 2016, arXiv preprint arXiv:1611.09726.
[20] Hegedűs I, Danner G, Jelasity M. Gossip learning as a decentralized alternative to federated learning. In: IFIP International conference on distributed applications and interoperable systems. Springer; 2019, p. 74–90.
[21] Buciluǎ C, Caruana R, Niculescu-Mizil A. Model compression. In: Proceedings of the 12th ACM SIGKDD international conference on knowledge discovery and data mining. 2006, p. 535–41.
[22] Cheng Y, Wang D, Zhou P, Zhang T. A survey of model compression and acceleration for deep neural networks. 2017, arXiv preprint arXiv:1710.09282.
[23] He Y, Lin J, Liu Z, Wang H, Li L-J, Han S. Amc: Automl for model compression and acceleration on mobile devices. In: Proceedings of the European conference on computer vision. ECCV, 2018, p. 784–800.
[24] Véstias MP, Duarte RP, de Sousa JT, Neto HC. Moving deep learning to the edge. Algorithms 2020;13(5):125.
[25] Eclipse Deeplearning4j Development Team. Deeplearning4j: Open-source distributed deep learning for the JVM. 2016.
[26] Parvat A, Chavan J, Kadam S, Dev S, Pathak V. A survey of deep-learning frameworks. In: 2017 International conference on inventive systems and control. ICISC, IEEE; 2017, p. 1–7.
[27] Abeel T, Van de Peer Y, Saeys Y. Java-ml: A machine learning library. J Mach Learn Res 2009;10:931–4.
[28] Raff E. JSAT: Java statistical analysis tool, a library for machine learning. J Mach Learn Res 2017;18(1):792–6.
[29] Li S. Tensorflow lite: On-device machine learning framework. J Comput Res Dev 2020;57:1839.
[30] Demosthenous G, Vassiliades V. Continual learning on the edge with tensorflow lite. 2021, arXiv preprint arXiv:2105.01946.
[31] Ellson J, Gansner E, Koutsofios L, North SC, Woodhull G. Graphviz—open source graph drawing tools. In: International symposium on graph drawing. Springer; 2001, p. 483–4.
[32] Sarridis I, Gkatziaki V, Krasanakis E, Giatsoglou N, Sarris N, Papadopoulos S, et al. Helios. TALK: A decentralised messaging framework that preserves the privacy of users. Open Research Europe 2022;2(29):29.
[33] Sen P, Namata G, Bilgic M, Getoor L, Galligher B, Eliassi-Rad T. Collective classification in network data. AI Mag 2008;29(3):93.
[34] Abadi M, Agarwal A, Barham P, Brevdo E, Chen Z, Citro C, et al. TensorFlow: Large-scale machine learning on heterogeneous systems. 2015, Software available from tensorflow.org.