On the Replicability of Knowledge Enhanced Neural Networks in a Graph Neural Network Framework

Luisa Werner, Nabil Layaïda, Pierre Genèves

On the Replicability of Knowledge Enhanced Neural Networks in a Graph Neural Network Framework

Luisa S. Werner[1], Nabil Layaïda[1] and Pierre Genèves[1]

[1]Tyrex team, Univ. Grenoble Alpes, CNRS, Inria, Grenoble INP, LIG, 38000 Grenoble, France

Abstract

In order to extend Knowledge Enhanced Neural Networks, we investigate the replicability of the approach and present a re-implementation of Knowledge Enhanced Neural Networks based on a Graph Neural Network framework (PyTorch Geometric). Knowledge Enhanced Neural Networks integrate prior knowledge in the form of logical formulas into an Artificial Neural Network by adding additional Knowledge Enhancement layers. The obtained results show that the model outperforms pure neural models as well as Neural-Symbolic models. Our long term goal is to be able to address more complex and large-scale knowledge graphs and to benefit from the wide range of functionalities available in PyTorch Geometric. To ensure that our implementation produces the same results, we replicate the original transductive experiments and explain the various challenges and the steps that we went through to reach that goal.

Keywords

Knowledge Enhanced Neural Networks, Neural-Symbolic Integration, Neural Networks, Fuzzy Logic, Relational Learning, Collective Classification, Reproducibility, Replicability,

1. Introduction

Recently, remarkable progress has been made in various research domains thanks to Deep Learning and the application of Artificial Neural Networks (NNs). The major strength of NNs is their ability to extract meaningful features from high-dimensional data without any expert knowledge. Despite their success, Deep Learning models are often criticised for their shortcomings in terms of interpretability, accountability and data-hungriness [1]. While Deep Learning approaches are mostly data-driven, Symbolic AI systems carry out logic-like reasoning steps over language-like representations while being more data efficient and understandable by humans [2]. In order to address the limitations of Deep Learning, the emerging research field of Neural-Symbolic Integration aims to combinate neural approaches with methods from Symbolic AI.

Knowledge Enhanced Neural Network (KENN) [3] is a Neural-Symbolic approach that stacks Knowledge Enhancers (KEs) as additional layers on top of a NN. These layers modify the outputs of the so-called Base Neural Network (Base NN) with respect to some prior knowledge in the form of first-order logic formulas. Both the KEs as well as the Base NN are designed to be fully differentiable and are optimized jointly with backpropagation. The KEs contain learnable *clause weights* that are modified during training and allow the model to be robust to wrong knowledge by setting the respective weights to zero.

While KENN was initially developed for Multi-Label Classification [3], an extension of the model to relational domains was proposed [4]. In order to evaluate the performance of KENN on relational data, the model is applied to a Collective Classification task on the Citeseer data set [5]. The experiments show that the proposed model is not only able to outperform the standalone Base NN but also the Neural-Symbolic models Semantic-Based Regularization (SBR) [6] and Relational Neural Machines (RNM) [7]. Furthermore, when limited training data is available, the use of prior knowledge leads to considerable performance gains.

As new results in Artificial Intelligence (AI) are increasingly derived through experiments on data, the reproducibility of experiments is crucial to obtain more transparent, independent and reliable research results. According to the definitions in [8] and [9], we define a result as *reproducible* if one can take the original code, re-execute it and obtain the reported results. Results are called *replicable* if one can create a new implementation that matches the conceptual and algorithmic descriptions given the article and obtain (or approximately obtain) the communicated results. The replication of results is a way to ensure that the conceptual descriptions and results of the literature are well interpreted. It is also a possibility to guarantee that the results are robust and transparent.

The first experiments with KENN show that it represents a promising Neural-Symbolic approach that has a enormous potential for extension. The current scope of the results is however limited. For example, the Citeseer data set used in KENN [4] is homogeneous and rather small. A *homogeneous* graph contains only one node and

edge type, while a *heterogeneous* graphs contains more than one node or edge type.

In the future, we intend to extensively investigate the applicability of KENN to larger knowledge graphs, the injection of more complex prior knowledge and the interaction with Graph Neural Networks (GNNs). In this context, PyTorch Geometric (PyG) [10] is a graph library based on PyTorch [11]. It integrates numerous benchmark data sets, provides graph-specific data structures, offers scalable algorithms, and implements numerous state-of-the-art GNNs. It therefore provides a good basis and offers many opportunities to extend relational KENN.

To take advantage of PyG, we represent a new interpretation and implementation of relational KENN. We use the conceptual description of relational KENN [4] and the original code written using Tensorflow 2 [12] and Keras [13]. In order to ensure that the conceptual descriptions are well understood, we replicate the transductive experiments on the Citeseer data set.

The main contributions of this work are (1) a new interpretation and implementation of relational KENN based on PyG and (2) the replication of the experiments reported in [4].

2. Knowledge Enhanced Neural Networks

KENN [3] is a Neural-Symbolic learning approach proposed by Alessandro Daniele and Luciano Serafini that was recently extended to relational domains [4]. A KENN consists of two main components: (1) A Base NN that solves a classification task and (2) an additional module of layers called Knowledge Enhancers (KEs) that incorporate prior knowledge. The whole architecture of Base NN and KEs can be trained end-to-end since all components are differentiable. In a classification setting the Base NN takes features $x \in \mathbb{R}^n$ as input and calculates outputs $y \in \mathbb{R}^m$ for m classes. The KEs are stacked on top of the Base NN and modify the predictions with respect to the satisfaction of the prior knowledge. The prior knowledge is specified as a set of clauses in first-order logic that are formulated as disjunction of literals and refer to the output classes. The logical language contains constants, variables and predicates. Unary predicates express properties of constants, while binary predicates describe relations between two constants. To give an example from the Citeseer data set, let the unary predicate $AI(x)$ describe that a scientific paper belongs to the document class AI and let the binary predicate $Cite(x, y)$ indicate a citation between two papers x and y. The clause

$$\forall x, y \,:\, AI(x) \wedge Cite(x, y) \longrightarrow AI(y)$$

describes that two papers belong to the same class AI if they cite each other. A KE revises the predictions y of the Base NN with respect to the satisfaction of the clauses in the prior knowledge and returns updated predictions y'. The KE contains *clause weights* which are modified during training together with the trainable parameters of the Base NN. The clause weight w_c of a clause c quantifies the importance of a specific clause on the input and can be robust to wrong knowledge by being set to zero.

To be incorporated in a neural architecture, the knowledge has to be interpreted in a real-valued domain. Fuzzy Logic [7] is applied to obtain values in a continuous interval of $[0, 1]$. The predictions of the Base NN are used as interpretation of unary predicates in the numeric domain. T-conorm functions map the truth values of grounded atoms to the truth value of a clause and are defined as follows:

$$\perp : [0, 1] \times [0, 1] \longrightarrow [0, 1] \tag{1}$$

To quantify the improvement of the satisfaction of a clause, the concept of a *t-conorm boost function (TBF)* that updates initial predictions, is specified as:

$$\delta \,:\, [0, 1]^n \rightarrow [0, 1]^n \tag{2}$$

In KENN, the following function is chosen as TBF and applied to the preactivation $\mathbf{v}_i$ of the Base NN.

$$\delta_s^{w_c}(\mathbf{v})_i = w_c \cdot \text{softmax}(\mathbf{v})_i \tag{3}$$

A module called *Clause Enhancer (CE)* implements the TBF for each clause. The CE selects the literals concerned by a clause from the preactivations of the Base NN and then applies the TBF of Equation 3 to obtain the changes. The changes introduced by all CEs are aggregated, added to the preactivations and fed into a logistic function. Various groundings of a clause to constants are stored in a matrix format where columns represent the predicates and rows the objects. Once the CE is instantiated, it works on several groundings of a clause.

When relational data is considered, not only unary predicates but also binary predicates are taken into account. In order to be applicable to relational data, some architectural modifications of KENN are required. When a clause contains multiple variables, the same grounded atom may occur in multiple groundings of the clause. Considering the clause $c \,:\, \neg AI(x) \vee \neg Cite(x, y) \vee AI(y)$ of the previous section and the two groundings $c[x/a, y/b]$ and $c[x/b, y/c]$ for example, the same grounded $AI(b)$ has to be enhanced. This leads to complications because changes proposed by the same CE are not aggregated in non-relational KENN. Consequently, adaptions of the data representation are required. In the relational version of KENN [4], the knowledge is split into a set of *unary clauses* that contain only unary predicates and a set of *binary clauses* that contain binary or lower arity predicates. The KE is applied on unary and binary

clauses separately before the changes are added to the preactivations. While the groundings of unary predicates can be represented as a matrix **U** that contains the objects as rows and predicates as columns, the keys of the binary predicates are two-dimensional. Consequently, binary predicates are represented as a matrix **B** that has as many rows as number of edges in a graph and columns as binary predicates. To enhance binary clauses, all predicates of the clause have to be presented together in one matrix on which the CE can be applied. Hence, unary predicates are extended to binary predicates by ignoring one component of the input. Given a unary predicate $P(x)$ for example, it can be extended to two binary predicates $P^X(x, y)$ and $P^Y(x, y)$. Consequently, the relational KENN contains a Join Layer that puts binary predicates and the binarized unary predicates in one matrix **M** on which the CE can operate. After obtaining the changes $\delta\mathbf{M}$, a Group-By Layer extracts the changes that apply to the same grounded atom and aggregates them. By representing unary and binary predicates in a single matrix the Knowledge Enhancement on relational domains is made possible.

3. PyTorch Geometric: A GNN Framework

Since our goal is to extend relational KENN presented in the previous section to large and complex knowledge graphs, we need graph-specific, scalable and flexible solutions. In this context, PyTorch Geometric (PyG) [10] is a graph framework based on PyTorch [11] that enables to process graph-structured data and build state-of-the art as well as customized GNNs. Among other graph libraries, PyG is currently the most popular library and is frequently updated [14]. It provides multiple GPU support, graph-specific mini batching and distributed graph learning, for example. Furthermore, PyG offers easy access to various large-scale benchmark data sets. In particular, the Open Graph Benchmark (OGB) [15] covers different models on versatile, real-world and large-scale graph data. PyTorch Scatter [16] is a small PyTorch library implementing highly optimized scatter and segment operations.

4. Methodology

The replication of experiments is a crucial step to ensure that the new interpretation maps the functionality of relational KENN and that future extensions are built on a solid basis. Our goal here is to re-implement KENN on PyG. Hence, we want to be sure that we are on a safe ground. For that purpose we replicate the transductive results reported in relational KENN [4]. In their work [4], the authors apply relational KENN to a Collective Classification task on the Citeseer data set. The reported results of the transductive experiments are presented in Table 1. The first three columns refer to the experiments in [7]. The last two columns present the experiments conducted with relational KENN. The considered metric is the mean test accuracy over multiple runs of the experiments. Its standard deviation is noted in brackets.

From these results the following conclusions are drawn:

- KENN has a superior performance compared to the standalone Base NN.
- The performance gains thanks to the Knowledge Enhancement are larger when limited training data is available.
- KENN leads to comparable or even superior performance in comparison to the baseline models RNM [7] and SBR [6].

The replication is a more complex task than the reproduction since it requires a comprehensive understanding of every component of the experiments.

We performed the following steps to re-interpret and re-implement relational KENN and replicate the results:

1. Reproduce the reported results in [4]
2. Follow the model description
3. Clarify under specified technical aspects
4. Inspect and solve eventual inconsistencies

In the next sections, we illustrate the process we followed in more details. Table 2 shows the necessary parameters to replicate the results and where they were specified. While some parameters are explicitly mentioned in the KENN paper [4], others could be derived by inspecting the code [17]. Further efforts were needed to identify the values of some parameters that are neither specified in the paper nor in the code.

4.1. Reproduction of Results

As a first step, we reproduced the results reported in [4] with the implementation of KENN [17] based on Tensorflow 2 and Keras. We use the versions of the Python packages as specified in [17]. Thanks to the availability of the original data set along with the code and the clear instructions given by the authors, the original results have been reproduced easily and are consistent with the numbers reported in Table 1. The first three columns contain the mean test accuracies reported in [7]. The following two columns represent the mean test accuracies obtained in [4]. The standard deviation of the test accuracy is noted in brackets. The reproduced results to which we later compare our replicated results, are presented in Table 4.

	Results reported in [7]			Results reported in [4]	
Training Set Dimension	**NN**	**SBR**	**RNM**	**NN**	**KENN**
10%	0.64	0.703 (0.063)	0.708 (0.068)	0.544	0.652 (0.108)
25%	0.667	0.729 (0.062)	0.735 (0.068)	0.629	0.702 (0.073)
50%	0.695	0.747 (0.052)	0.753 (0.058)	0.68	0.744 (0.065)
75%	0.708	0.764 (0.056)	0.766 (0.058)	0.733	0.788 (0.055)
90%	0.726	0.780 (0.054)	0.780 (0.054)	0.759	0.808 (0.049)

Table 1
Results of the transductive experiments reported in [7] (first three columns) and [4] (last two columns) ordered by training set dimension.

Parameter	Value	specified in Paper	specified in the code
Base NN - Number of Hidden Layers	3	☒	☒
Base NN - Number of Hidden Neurons	50	☒	☒
Base NN - Hidden Layer Activation	ReLu	☒	☒
Base NN - Output Layer Activation	Linear	☒	☒
KE - Clause Weight Initialization	Constant, 0.5	☒	☒
KE - Binary Preactivations	500	☒	☒
KE - Number of KE Layers	3		☒
KE - Range Constraint	$[0.0, 500.0]$		☒
Epochs	300		☒
Batch Size	Full-batch		☒
Loss function	Categorical Cross-entropy		☒
Early Stopping - Patience	10		☒
Early Stopping - Min Delta	0.001		☒
Dropout Rate	0.0, no dropout		
Learning Rate	0.001		
Base NN - Weight Initialization	Random, Glorot uniform		
Base NN - Bias Initialization	Constant, Zeroes		
Optimizer	Adam, $\beta_1 = 0.9, \beta_2 = 0.99, \epsilon = 1e-07$		

Table 2
The hyperparameter set used in the experiments with relational KENN [4] on Citeseer.

4.2. Re-interpretation and Re-implementation of KENN

In order to replicate relational KENN in PyTorch, we firstly followed the conceptual descriptions of the layers in KENN and the experimental setting described in the paper [4].

4.2.1. Model Architecture

The structure of a KE layer, its mathematical foundation and its extension to the relational domain is described in the original work [4]. In the experiments, a dense NN with three hidden layers of fifty nodes each and ReLu activation function [18] is applied as Base NN. The output layer of the Base NN has a linear activation. The output of the last layer is processed by a softmax function.

4.2.2. Data Set

The experiments are conducted on the Citeseer data set. Citeseer is a citation graph and a well-known benchmark data set for node classification. It consists of 3312 scientific publications classified into one of six classes and 4732 edges between the nodes. Each publication in the data set is described by a one-hot-encoded word vector of dimension 3703. The data set can be modelled as graph where scientific publications are represented by nodes and citations between them are the edges of the graph. The bag-of-word vectors are used as node features. The version of the Citeseer data set used for the experiments reported in [4] is provided together with the code [17].

4.2.3. Prior Knowledge

The set of logical clauses used in the experiments is explicitly described in [4]. The hypothesis is made that two documents belong to the same class when they cite each other. The following clause is instantiated for each document class (computer science research fields) of the set $\{AG, AI, HCI, ML, DB, IR\}$ from which a set of six clauses is derived as follows:

$$\forall x \forall y : \neg Class(x) \vee \neg Cite(x; y) \vee Class(y)$$

It is assumed that the edges and thus the citations are known a priori. For this reason, the preactivation of the binary predicate *Cite* is set to a high value (500.0). To reduce complexity, only pairs of nodes that are connected by edges are considered. The clause weights are initialized with a constant value (0.5).

4.2.4. Experimental Setup

Since no global agreement on the split of the Citeseer data set into training, validation and test set exists, several sizes of the training set have been tested to evaluate the impact on the model performance. Specifically, the dimensions are [0.1, 0.25, 0.5, 0.75, 0.9] and the respective amount of data is randomly sampled from the main set. The remaining samples are randomly split into a validation set and a test set. Furthermore, to counter the dependence of the results on a particular split, multiple runs are conducted. Per run the data set is split and the model initialised . The mean accuracy over all runs is a more stable indicator for the model performance. In the original experiments, 100 runs were conducted.

In our implementation, the relational layers of KENN are based on the library *PyTorch Scatter* [16]. PyTorch Scatter is a package consisting of a small extension library of highly optimized sparse update (scatter and segment) operations, which are missing in the main PyG package. We use PyG Data objects to store and handle the input data in a format optimized for graph data.

4.3. Missing Parameters

Not all the required specifications and parameter values to re-interpret and implement the approach are mentioned in the relational KENN paper. We needed to examine and test the KENN code published on GitHub at [17] to recover the necessary information.

4.3.1. Model Architecture

Regarding the architecture of the Base NN, three dropout layers [19] are applied to the hidden layers and the dropout rate is set to 0.5. A weight clipping is implemented to enforce the clause weights to stay in a range between $[0.0, 500.0]$ during the training. Three relational KE layers are stacked on top of the Base NN.

4.3.2. Data set

In KENN, only the amount of data chosen for the training set but not for validation and test set is specified. We found that 20% of the remaining samples is selected for the validation set by examining the code. Since the distribution of the classes in Citeseer is by default imbalanced, a re-balancing is conducted. Consequently, the samples are split randomly to achieve an approximately balanced class distribution in the training set.

4.3.3. Prior Knowledge

The way how knowledge can be specified is illustrated in the implementation and the documentation of KENN [20]. Instructions are given on how prior knowledge can be encoded as a text format. This encoding is read by a unary and binary parser in order to create the corresponding KE layers.

4.3.4. Experimental Setup

Not all hyperparameters of the training setting are explicitly mentioned in the paper. We had to examine and run the code to recover them. The training is conducted on the full data batch, so that the batch size corresponds to the size of the training set. Categorical cross-entropy is chosen as loss function and Adam [22] as optimizer. Per run, up to 300 training epochs are carried out. An early stopping mechanism in KENN interrupts the training when the validation accuracy stagnates. The early stopping hyperparameters *patience* and *minimum delta* are set to 10 and 0.001.

4.4. Resolution of Inconsistencies

By following the descriptions of relational KENN [4] and the provided implementation [17], we were able to functionally re-implement the experiments in PyTorch. Nevertheless, the results we initially obtained differed considerably from those reported in [4]. In each component of the system we had to identify the sources of divergence. We investigated the following aspects as potential causes of divergence:

- Wrong interpretation of some concepts described in the paper
- Bugs in the original or replicated implementation
- Deviating initialization of parameters
- Framework-specific settings

	Reproduced KENN Experiments [17]		Replicated Experiments [21]	
Training Dimension	**NN**	**KENN**	**NN**	**KENN**
10%	0.0230	0.2358	0.0090	0.0787
25%	0.0292	0.2367	0.0128	0.0795
50%	0.0476	0.2360	0.01937	0.0810
75%	0.0541	0.2130	0.0250	0.0830
90%	0.1240	0.2375	0.0308	0.0841

Table 3
Average epoch times in the reproduced and replicated experiments with relational KENN.

Inspecting the concrete behavior of NNs is challenging since neural models non-deterministic and essentially data-driven. To get useful insights and identify the sources of divergence, we appied the following strategies:

- Disable random operations temporarily to make the behavior (input and output) of the components deterministic.
- Check the training process step-by-step during a single epoch
- Consider different modules separately
- Implement framework-specific components manually
- Inspect framework-specific default parameters

In detail, we simplified the model, temporarily disabled random operations and inspected a single epoch until both implementations produced the same results. Then, we iteratively put back random operations and complexity while supervising the performance of both models. This allowed us to identify the components that behave differently.

4.4.1. Deterministic Pre-processing

As a first step, we made the pre-processing steps identical for both approaches to guarantee that the same data is fed into the first epoch of the model. The pre-processing has two sources of randomness: Firstly, when the training samples are selected for the training, validation and test set ($\mathrm{Numpy.random.permutation}$) and secondly, in the re-balancing ($\mathrm{list(set(classes))}$). The $\mathrm{set()}$ operator is based on a hash table and the order of the output is not deterministic. We seeded the random permutation and used a sorted list to make the pre-processing deterministic for both approaches.

4.4.2. Model Simplification

We compared the data flow through the KENN model components in the first epoch. At first, the data is fed into the dense layers of the Base NN and subsequently manipulated by the KE layers. The dense layers in the original implementation are modelled as Keras dense layers. Since initialization of weights and biases is not explicitly specified, the default initialization of Keras applies. In this case, kernels are initialized randomly following a Glorot uniform distribution and the bias is initialized with zeroes.

In PyTorch we implemented the dense layers as linear modules. By default the weights and biases are randomly initialized following the uniform distribution $\mathcal{U}[-\frac{1}{\sqrt{k}}, \frac{1}{\sqrt{k}}]$ for a weight matrix with k columns. To better inspect the behavior of the layers, we initialized the kernels in both implementations with constant values. We temporarily deactivated the dropout layers in both approaches since they are random operations. After having completed these steps, the data flow through the Base NN was identical for both models in the first forward pass. Since the KE only introduces clause weights as learnable parameters which are initialized with constant values, the data propagation through the KEs was comparable.

4.4.3. Accuracy, Loss Function and Optimizer

To ensure that both approaches use the same loss function, we replaced the framework-specific categorical cross-entropy functions in Keras and PyTorch by a manual implementation of the loss function according to the following definition of the categorical cross-entropy:

$$\mathrm{L} = -\sum_{i=1}^{\mathrm{n}} y_i \cdot \log \hat{y}_i \tag{4}$$

$\hat{y}_i$ contains the predicted values and y_i the ground truth labels for a sample of size n.

Accordingly, we did not rely on the definition of the accuracy as evaluation metric given by the frameworks but used an implementation of the accuracy from Numpy [23]. In both implementations the Adam optimizer [22] is used. In order to simplify the optimization process, we used Stochastic Gradient Descent instead of Adam.

4.4.4. Activation of Components

To use the same amount of epochs for both approaches, we first disabled the early stopping mechanism temporarily. As explained before, the early stopping interrupts the training when the validation accuracy stagnates over several epochs. At this stage, the forward and backward pass of the first epochs were deterministic and identical for both approaches. Nevertheless, without random operations a neural model is not able to explore the hypothesis space.

In the following, we iteratively put the original elements back for both approaches and observed the results. We firstly activated the random operations in the preprocessing so that different splits were created for each run. Regarding the initialization of weights and biases in the dense layers of the Base NN, we used the default initialization as in KENN. We conducted 300 epochs and 10 runs to check if they were similar. Then, we set the optimizer back to Adam. Adam introduces a set of hyperparameters that is not explicitly specified in the original approach, so the default parameters of Keras are used. When comparing the default initialization of Adam in PyTorch and in Keras in details, we found that the frameworks use different default values. We identified the parameters used in KENN, see Table 2, and set the optimizer in our system to the same values.

When we put the dropout layers back, we noticed that the dropout layers in KENN do not work even though the dropout rate is set to 0.5. By further investigating this aspect, it turned out that dropout layers are only activated if the model is called with the parameter *training* set to *True*. Since this is not the case in the KENN, the dropout layers remain deactivated even though they are added to the model and the dropout rate is set to a value greater than zero. Consequently, to make the models comparable, we also deactivated the dropout layers in our implementation. By enabling the early stopping mechanism with the same parameters as in KENN, we finally obtained results that are consistent with the reproduced and reported results.

4.5. Replicability Summary

By following the ideas and results presented in the KENN paper and by inspecting their code, we were able to finally re-implement and replicate relational KENN in a GNN framework. In order to achieve this goal, the main difficulty that we faced was the under specification of the system and implicit information. The detailed inspection of a simplified and more deterministic setting of the system modules was key to reveal the hidden information and take it in consideration in the replication.

Table 4 summarizes the results obtained by the reproduction and replication of the transductive KENN experiments ordered by training set dimensions. The first two columns contain the results reported in [4]. The following two columns present the reproduced experiments and the last two columns the replicated experiments with our interpretation of KENN. The mean test accuracy over 100 runs is considered as performance metric. The standard deviation is noted in brackets. It can be seen that the reproduction and replication of the results lead to consistent results. We note that the standard deviation of the mean accuracy obtained in the reproduction and replication of the experiments is smaller than reported in the paper. KENN outperforms the standalone Base NN for all experiments and all training set dimensions. Also, in all experiments the improvement gains with KENN in comparison to the standalone Base NN are larger when the training set is small.

Table 3 compares the average epoch times over the 100 runs in seconds for the reproduced and replicated experiments. The timing of an epoch includes one training step and one validation step. It can be shown that our implementation of KENN leads to around twice as fast average epoch times compared to the original KENN implementation. Our experiments were performed on a server running Ubuntu 20.04 LTS equipped with a Intel(R) Xeon(R) Silver 4114 CPU @ 2.20GHz, 192GB RAM, and one NVIDIA Quadro P5000 GPU card of 16 Go.

In conclusion, the replication of results turned out to be a more challenging task than initially thought. In contrast to reproduction, replication requires a full understanding of the proposed concepts together with a detailed inspection of all components of the accompanying experiment. Beyond this, replication can reveal conceptual limitations, bugs, ambiguities. It can also give more insights on potential improvements and extensions. With this work, we demonstrate that the results with relational KENN on Citeseer can be achieved in two different frameworks. The future enhancements of relational KENN with our implementation thus have a solid and reliable foundation.

5. Conclusion and Perspectives

In this work, we introduced our interpretation and implementation of relational Knowledge Enhanced Neural Networks based on the Graph Neural Network Framework PyTorch Geometric and PyTorch Scatter. We verified the results of KENN by replicating the transductive experiments on the Citeseer data set. The replication of results is a necessary step to ensure that the functionality can be safely reused and extended. Furthermore, the various steps we undertook to replicate the experiments and re-implement relational KENN, show that this task is not as straightforward as one may think. They can be taken as a starting point for a more general replicability

	Results reported in [4]		Reproduced KENN Experiments [17]		Replicated Experiments [21]	
Training Dimension	**NN**	**KENN**	**NN**	**KENN**	**NN**	**KENN**
10%	**0.544**	**0.652** (0.108)	**0.5107** (0.092)	**0.6613** (0.0443)	**0.5517** (0.045)	**0.6487** (0.053)
25%	**0.629**	**0.702** (0.073)	**0.6406** (0.060)	**0.7156** (0.043)	**0.6305** (0.022)	**0.7034** (0.035)
50%	**0.68**	**0.744** (0.065)	**0.7063** (0.034)	**0.7573** (0.012)	**0.6763** (0.019)	**0.7422** 0.023)
75%	**0.733**	**0.788** (0.055)	**0.7604** (0.017)	**0.7973** (0.013)	**0.7306** (0.023)	**0.7838** (0.0164)
90%	**0.759**	**0.808** (0.049)	**0.7760** (0.021)	**0.8096** (0.020)	**0.7486** (0.029)	**0.8045** (0.015)

Table 4
The reported [4], reproduced and replicated mean test accuracies of the transductive experiments ordered by the training set dimension. The standard deviations are given in brackets.

method.

This work paves the way for future experiments on real-world data with Knowledge Enhanced Neural Networks and ots application to large-scale knowledge graphs. The experiments on the Citeseer data set can be seen as limited since Citeseer does not represent a large-scale knowledge graph [24]. The application of the concepts of KENN to larger, various and more complex data sets is ongoing research and we could already apply relational KENN to a data set from OGB [15]. In the short term, it would be interesting to test KENN in conjunction with GNNs and extend primary experiments on large-scale graphs.

Acknowledgments

This work has been partially supported by the MIAI Knowledge communication and evolution chair (ANR-19-P3IA-0003). We thank Luciano Serafini and Alessandro Daniele, the authors of KENN, for their support and Jerome Euzenat for his feedback on this work.

References

[1] Z. Susskind, B. Arden, L. K. John, P. Stockton, E. B. John, Neuro-symbolic ai: An emerging class of ai workloads and their characterization, 2021. URL: https://arxiv.org/abs/2109.06133. doi:10.48550/ARXIV.2109.06133.

[2] M. Garnelo, M. Shanahan, Reconciling deep learning with symbolic artificial intelligence: representing objects and relations, Current Opinion in Behavioral Sciences 29 (2019) 17–23. URL: https://www.sciencedirect.com/science/article/pii/S2352154618301943. doi:https://doi.org/10.1016/j.cobeha.2018.12.010, artificial Intelligence.

[3] A. Daniele, L. Serafini, Knowledge Enhanced Neural Networks, 2019, pp. 542–554. doi:10.1007/978-3-030-29908-8_43.

[4] A. Daniele, L. Serafini, Neural networks enhancement with logical knowledge, 2020. URL: https://arxiv.org/abs/2009.06087. doi:10.48550/ARXIV.2009.06087.

[5] Q. Lu, L. Getoor, Link-based classification, in: Proceedings of the Twentieth International Conference on International Conference on Machine Learning, ICML'03, AAAI Press, 2003, p. 496–503.

[6] M. Diligenti, M. Gori, C. Saccà, Semantic-based regularization for learning and inference, Artificial Intelligence 244 (2017) 143–165. URL: https://www.sciencedirect.com/science/article/pii/S0004370215001344. doi:https://doi.org/10.1016/j.artint.2015.08.011, combining Constraint Solving with Mining and Learning.

[7] G. Marra, M. Diligenti, F. Giannini, M. Gori, M. Maggini, Relational neural machines, 2020. URL: https://arxiv.org/abs/2002.02193. doi:10.48550/ARXIV.2002.02193.

[8] H. E. Plesser, Reproducibility vs. replicability: A brief history of a confused terminology, Frontiers in Neuroinformatics 11 (2018). URL: https://www.frontiersin.org/article/10.3389/fninf.2017.00076. doi:10.3389/fninf.2017.00076.

[9] N. P. Rougier, K. Hinsen, F. Alexandre, T. Arildsen, L. A. Barba, F. C. Benureau, C. T. Brown, P. de Buyl, O. Caglayan, A. P. Davison, M.-A. Delsuc, G. Detorakis, A. K. Diem, D. Drix, P. Enel, B. Girard, O. Guest, M. G. Hall, R. N. Henriques,

X. Hinaut, K. S. Jaron, M. Khamassi, A. Klein, T. Manninen, P. Marchesi, D. McGlinn, C. Metzner, O. Petchey, H. E. Plesser, T. Poisot, K. Ram, Y. Ram, E. Roesch, C. Rossant, V. Rostami, A. Shifman, J. Stachelek, M. Stimberg, F. Stollmeier, F. Vaggi, G. Viejo, J. Vitay, A. E. Vostinar, R. Yurchak, T. Zito, Sustainable computational science: the ReScience initiative, PeerJ Computer Science 3 (2017) e142. doi:10.7717/peerj-cs.142.

[10] M. Fey, J. E. Lenssen, Fast graph representation learning with PyTorch Geometric, in: ICLR Workshop on Representation Learning on Graphs and Manifolds, 2019.

[11] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, S. Chintala, Pytorch: An imperative style, high-performance deep learning library, in: Advances in Neural Information Processing Systems 32, Curran Associates, Inc., 2019, pp. 8024–8035. URL: http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf.

[12] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, X. Zheng, TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. URL: https://www.tensorflow.org/, software available from tensorflow.org.

[13] F. Chollet, et al., Keras, 2015. URL: https://github.com/fchollet/keras.

[14] K. Vu, 7 open source libraries for deep learning graphs, 2021. URL: https://dzone.com/articles/open-source-libraries-for-deep-learning-graphs.

[15] W. Hu, M. Fey, M. Zitnik, Y. Dong, H. Ren, B. Liu, M. Catasta, J. Leskovec, Open graph benchmark: Datasets for machine learning on graphs, CoRR abs/2005.00687 (2020). URL: https://arxiv.org/abs/2005.00687. arXiv:2005.00687.

[16] M. Fey, Pytorch scatter, 2018. URL: https://github.com/rusty1s/pytorch_scatter.

[17] A. Daniele, R. Mazzieri, Kenn-citeseer-experiments, 2020. URL: https://github.com/rmazzier/KENN-Citeseer-Experiments.

[18] A. F. Agarap, Deep learning using rectified linear units (relu), 2018. URL: https://arxiv.org/abs/1803.08375. doi:10.48550/ARXIV.1803.08375.

[19] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, R. Salakhutdinov, Dropout: A simple way to prevent neural networks from overfitting, Journal of Machine Learning Research 15 (2014) 1929–1958. URL: http://jmlr.org/papers/v15/srivastava14a.html.

[20] A. Daniele, L. Serafini, Knowledge enhanced neural networks, in: A. C. Nayak, A. Sharma (Eds.), PRICAI 2019: Trends in Artificial Intelligence, Springer International Publishing, Cham, 2019, pp. 542–554. URL: https://github.com/DanieleAlessandro/KENN2.

[21] L. Werner, N. Layaïda, P. Genèves, Replicability of kenn, 2022. URL: https://gitlab.inria.fr/luwerner/replicability_kenn.

[22] D. P. Kingma, J. Ba, Adam: A method for stochastic optimization, 2014. URL: https://arxiv.org/abs/1412.6980. doi:10.48550/ARXIV.1412.6980.

[23] C. R. Harris, K. J. Millman, S. J. van der Walt, R. Gommers, P. Virtanen, D. Cournapeau, E. Wieser, J. Taylor, S. Berg, N. J. Smith, R. Kern, M. Picus, S. Hoyer, M. H. van Kerkwijk, M. Brett, A. Haldane, J. F. del Río, M. Wiebe, P. Peterson, P. Gérard-Marchant, K. Sheppard, T. Reddy, W. Weckesser, H. Abbasi, C. Gohlke, T. E. Oliphant, Array programming with NumPy, Nature 585 (2020) 357–362. URL: https://doi.org/10.1038/s41586-020-2649-2. doi:10.1038/s41586-020-2649-2.

[24] O. Shchur, M. Mumme, A. Bojchevski, S. Günnemann, Pitfalls of graph neural network evaluation, CoRR abs/1811.05868 (2018). URL: http://arxiv.org/abs/1811.05868. arXiv:1811.05868.