Deep Learning for Ontology Reasoning

Patrick Hohenecker, Thomas Lukasiewicz
Department of Computer Science
University of Oxford
Oxford, United Kingdom
{patrick.hohenecker, thomas.lukasiewicz}@cs.ox.ac.uk

Abstract

In this work, we present a novel approach to ontology reasoning that is based on deep learning rather than logic-based formal reasoning. To this end, we introduce a new model for statistical relational learning that is built upon deep recursive neural networks, and give experimental evidence that it can easily compete with, or even outperform, existing logic-based reasoners on the task of ontology reasoning. More precisely, we compared our implemented system with one of the best logic-based ontology reasoners at present, RDFox, on a number of large standard benchmark datasets, and found that our system attained high reasoning quality, while being up to two orders of magnitude faster.

1 Introduction

In the last few years, there has been an increasing interest in the application of machine learning (ML) to the field of knowledge representation and reasoning (KRR), or, more generally, in learning to reason over symbolic data—cf., e.g., Gabrilovoch *et al.* (2015). The main motivation behind this is that most KRR formalisms used today are rooted in symbolic logic, which allows for answering queries accurately by employing formal reasoning, but also comes with a number of issues, like difficulties with handling incomplete, conflicting, or uncertain information and scalability problems.

However, many of these issues can be dealt with effectively by using methods of ML, which are in this context often subsumed under the notion of statistical relational learning (SRL; Getoor and Taskar, 2007)—cf. Nickel *et al.* (2016) for a recent survey. Notice, though, that the use of ML for reasoning is a tradeoff. On the one hand, ML models are often highly scalable, more resistant to disturbances in the data, and can provide predictions even if formal reasoning fails. On the other hand, however, their predictions are correct with a certain probability only. In contrast to this, formal reasoners are often obstructed by the above problems, but if they can provide inferences, then these are correct with certainty.

We believe that the combination of both fields, i.e., ML and KRR, is an important step towards human-level artificial intelligence. However, while there exist elaborate reasoning systems already, SRL is a rather young field that has, we believe, not hit its boundaries yet. Therefore, in this work, we introduce a new approach to SRL based on deep learning, and apply it to the task of reasoning over ontological knowledge bases (OKBs). These are knowledge bases (KBs) that consist of a set of facts together with a formal description of the domain of interest—the so-called ontology. The reason why we chose this very task is its practical significance as well as the fact that it commonly comprises extensive formal reasoning.

The motivation for employing deep learning, however, which refers to the use of neural networks (NNs) that perform many sequential steps of computation, should be fairly obvious. In the last ten years, deep learning has been applied to a wide variety of problems with tremendous success, and constitutes the state-of-the-art in fields like computer vision and natural language processing (NLP) today. Interestingly, there are also a few published attempts to realize formal reasoning by means

of deep NNs. However, these focus on rather restricted logics, like natural logic (Bowman, 2013) or real logic (Serafini and d'Avila Garcez, 2016), and do not consider reasoning in its full generality. Besides this, »*reasoning*« appears in connection with deep learning mostly in the context of NLP—e.g., Socher *et al.* (2013).

The main contributions of this paper are briefly as follows:

- We present a novel method for SRL that is based on deep learning with recursive NNs, and apply it to ontology reasoning.
- Furthermore, we provide an experimental comparison of the suggested approach with one of the best logic-based ontology reasoners at present, RDFox (Nenov *et al.*, 2015), on several large standard benchmarks. Thereby, our model achieves a high reasoning quality while being up to two orders of magnitude faster.
- To the best of our knowledge, we are the first to investigate ontology reasoning based on deep learning on such large and expressive OKBs.

The rest of this paper is organized as follows. In the next section, we review a few concepts that our approach is built upon. Section 3 introduces the suggested model in full detail, and Section 4 discusses how to apply it to ontology reasoning. In Section 5, we evaluate our model on four datasets, and compare its performance with RDFox. We conclude with a summary of the main results, and give an outlook on future research.

2 Background

As mentioned in the introduction already, our work lies at the intersection of two, traditionally quite separated, fields, namely ML and KRR. Therefore, in this section, we review the most important concepts, from both areas, that are required to follow the subsequent elaborations.

2.1 Ontological Knowledge Bases (OKBs)

A central idea in the field of KRR is the use of so-called ontologies. In this context, an ontology is a formal description of a concept or a domain, e.g., a part of the real world, and the word »*formal*« emphasizes that such a description needs to be specified by means of some knowledge representation language with clearly defined semantics. This, in turn, allows us to employ formal reasoning in order to draw conclusions based on such an ontology.

An important aspect to note is that an ontology is situated on the meta-level, which means that it might specify general concepts or relations, but does not contain any facts. However, in the sequel we only talk about a number of facts together with an ontology that describes the domain of interest, and we refer to such a setting as an ontological knowledge base (OKB).

In practice, and in the context of description logics (Baader *et al.*, 2007), ontologies are usually defined in terms of unary and binary predicates. Thereby, unary predicates are usually referred to as concepts or classes, and define certain categories, e.g., of individuals that possess a particular characteristic. In contrast to this, binary predicates define relationships that might exist between a pair of individuals, and are usually referred to as relations or roles.

What is really appealing about ontologies is that they usually not just define those predicates, but also rules that allow us to draw conclusions based on them. This could encompass simple inferences like every individual of class `women` belongs to class `human` as well, but also much more elaborate reasoning that takes several classes and relations into account. Notice further that we can view almost any relational dataset as an OKB with an ontology that does not specify anything except the classes and relations that exist in the data.

Based on the fact that we hardly ever encounter ontologies with predicates of arity greater than two in practice, we confine ourselves to this particular case in the subsequent treatment—the approach introduced in this work can be easily extended to the general case, though. Any OKB that is defined in terms of unary and binary predicates only has a natural representation as labeled directed multigraph[1] if individuals are interpreted as vertices and every occurrence of a binary predicate as a

[1] If we really need to account for predicates of arity greater than two, then we can view any such dataset as a hypergraph, and extend the RTN model introduced in the next section with convolutional layers as appropriate.

directed edge. Thereby, edges are labeled with the name of the according relation, and vertices with an incidence vector that indicates which classes they belong to. Notice, however, that, depending on the used formalism, OKBs may adhere to the so-called open-world assumption (OWA). In this case, a fact can be **true**, **false**, or **unknown**, which is, e.g., different from classical first-order logic. The presence of the OWA is reflected by according three-valued incidence vectors, whose elements may be any of 1, -1, or 0, respectively, and indicate that an individual belongs to a class, is not a member of the same, or that this is unknown.

2.2 Recursive Neural Tensor Networks (RNTNs)

Recursive NNs (Pollack, 1990) are a special kind of network architecture that was introduced in order to deal with training instances that are given as trees rather than, as more commonly, feature vectors. In general, they can deal with any directed acyclic graph (DAG), since any such graph can be unrolled as a tree, and the only requirement is that the leaf nodes have vector representations attached to them. An example from the field of NLP is the parse tree of a sentence, where each node represents one word and is given as either a one-hot-vector or a previously learned word embedding.

Unlike feed-forward networks, recursive NNs do not have a fixed network structure, but only define a single recursive layer, which accepts two vectors as input and maps them to a common embedding. This layer is used to reduce a provided tree step by step in a bottom-up fashion until only one single vector is left. The resulting vector can be regarded as an embedding of the entire graph, and may be used, e.g., as input for a subsequent prediction task.

In this work, we make use of the following recursive layer, which defines what is referred to as recursive neural tensor network (RNTN; Socher *et al.*, 2013):

$$g(\mathbf{x}, R, \mathbf{y}) = \mathbf{U}_R f\left(\mathbf{x}^T \mathbf{W}_R^{[1:k]} \mathbf{y} + \mathbf{V}_R \begin{bmatrix}\mathbf{x} \\ \mathbf{y}\end{bmatrix} + \mathbf{b}_R\right), \tag{1}$$

where $\mathbf{x}, \mathbf{y} \in \mathbb{R}^d$, $\mathbf{U}_R \in \mathbb{R}^{d \times k}$, $\mathbf{V}_R \in \mathbb{R}^{k \times 2d}$, $\mathbf{W}_R \in \mathbb{R}^{d \times d \times k}$, $\mathbf{b}_R \in \mathbb{R}^k$, and f is a nonlinearity that is applied element-wise, commonly $\tanh$. Thereby, the term $\mathbf{x}^T \mathbf{W}_R^{[1:k]} \mathbf{y}$ denotes a bilinear tensor product, and is computed by multiplying $\mathbf{x}$ and $\mathbf{y}$ with every slice of $\mathbf{W}_R$ separately. So, if $\mathbf{z}$ is the computed tensor product, then $\mathbf{z}_i = \mathbf{x}^T \mathbf{W}_R^{[i]} \mathbf{y}$. In addition to the actual input vectors, $\mathbf{x}$ and $\mathbf{y}$, the tensor layer accepts another parameter R, which may be used to specify a certain relation between the provided vectors. This makes the model more powerful, since we use a separate set of weights for each kind of relation.

In general, recursive NNs are trained by means of stochastic gradient descent (SGD) together with a straightforward extension of standard backpropagation, called backpropagation through structure (BPTS; Goller and Küchler, 1996).

3 Relational Tensor Networks (RTNs)

In this section, we present a new model for SRL, which we—due to lack of a better name—refer to as relational tensor network (RTN). An RTN is basically an RNTN that makes use of a modified bilinear tensor layer. The underlying intuition, however, is quite different, and the term »*relational*« emphasizes the focus on relational datasets.

3.1 The Basic Model

As described in the previous section, recursive NNs allow for computing embeddings of training instances that are given as DAGs. If we face a relational dataset, though, then the training samples are actually vertices of a graph, namely the one that is induced by the entire relational dataset, rather than a graph itself. However, while this does not fit the original framework of recursive networks, we can still make use of a recursive layer in order to update the representations of individuals based on the structure of dataset. In an RTN, this deliberation is reflected by the following modified tensor layer:

$$\tilde{g}(\mathbf{x}, R, \mathbf{y}) = \mathbf{x} + \mathbf{U}_R f\left(\mathbf{x}^T \mathbf{W}_R^{[1:m]} \mathbf{y} + \mathbf{V}_R \mathbf{y}\right), \tag{2}$$

where the notation is the same as in Equation 1 except that $\mathbf{V}_R \in \mathbb{R}^{k \times d}$.

The intuition here is quite straightforward. While individuals in a relational dataset are initially represented by their respective feature vectors, big parts of the total information that we have are actually hidden in the relations among them. However, we can use a recursive network, composed of tensor layers like the one denoted in Equation 2, to incorporate these data into an individual's embedding. Intuitively, this means that we basically apply a recursive NN to an update tree of an individual, and thus compute an according vector representation based on the relations that it is involved in. For the RTN, we adopted the convention that a tensor layer $\tilde{g}$ updates the individual represented by $\mathbf{x}$ based on an instance $(\mathbf{x}, R, \mathbf{y})$ of relation R that is present in the data. Furthermore, if the relations in the considered dataset are not symmetric, then we have to distinguish whether an individual is the source or the target of an instance of a relation. Accordingly, the model has to contain two sets of parameters for such a relation, one for updating the source and one for the target, and we denote these as $R^{\triangleright}$ and $R^{\triangleleft}$, respectively. This means, e.g., that $\tilde{g}(\mathbf{x}, R^{\triangleleft}, \mathbf{y})$ denotes that the embedding of $\mathbf{x}$ is updated based on $(\mathbf{y}, R, \mathbf{x})$.

The foregoing considerations also explain the differences between Equation 2 and the original tensor layer given in Equation 1 (Socher *et al.*, 2013). First and foremost, we see that in our model $\mathbf{x}$ is added to what basically used to be the tensor layer before, which is predicated on the fact that we want to update this very vector. Furthermore, $\mathbf{x}$ does not affect the argument of the nonlinearity f independently of $\mathbf{y}$, since $\mathbf{x}$ by itself should not determine the way that it is updated. Lastly, there is no bias term on the right-hand side of Equation 2 to prevent that there is some kind of default update irrespective of the individuals involved.

We also considered to add another application of the hyperbolic tangent on top of the calculations given in Equation 2 in order to keep the elements of the created embeddings in $[-1, 1]$. This would ensure that there cannot be any embeddings with an oddly large norm due to individuals being involved in a large number of relations. However, since we did not encounter any problems like this in our experiments, we decided against the use of this option, as it could introduce additional problems like vanishing gradients.

3.2 Training

As already suggested before, we usually employ RTNs in order to compute embeddings for individuals that are used as input for some specific prediction task. Therefore, it makes sense to train an RTN together with the model that is used for computing these predictions, and whenever we talk about an RTN in the sequel, we shall assume that it is used together with some predictor on top of it. If we only care about individual embeddings irrespective of any particular subsequent task, then we can simply add a feed-forward layer—or some other differentiable learning model—on top of the RTN, and train the model to reconstruct the provided feature vectors. This way, an RTN can be used as a kind of relational autoencoder.

Training such a model is straightforward, and switches back and forth between computing embeddings and making predictions based on them. In each training iteration, we start from the feature vectors of the individuals as they are provided in the dataset. Then, as a first step, we sample mini-batches of triples from the dataset, and randomly update the current embedding of one of the individuals in each triple by means of our RTN. The total number of mini-batches that are considered in this step is a hyperparameter, and we found during our experiments that it is in general not necessary to consider the entire dataset.

Next, we sample mini-batches of individuals from the dataset, and compute predictions for them based on the embeddings that we created in the previous step. In doing so, it makes sense to consider both individuals that have been updated as well as some that still have their initial feature vectors as embeddings. This is important for the model to learn how to deal with individuals that are involved in very few relations or maybe no one at all, which is not a rare case in practice. Therefore, in our experiments, we used mini-batches that were balanced with respect to this, and switched back to step number one as soon as each of the previously updated individuals has been sampled once.

The loss function as well as the optimization strategy employed depends, as usual, on the concrete task, and is chosen case by case.

3.3 Related Models

In the field of SRL, there exist a few other approaches that model the effects of relations on individual embeddings in terms of (higher-order) tensor products—cf., e.g., Nickel *et al.* (2011, 2012). However, these methods, which belong to the category of latent variable models, are based on the idea of factorizing a tensor that describes the structure of a relational dataset into a product of an embedding matrix as well as another tensor that represents the relations present in the data. The actual learning procedure is then cast as a regularized minimization problem based on this formulation. In contrast to this, an RTN computes embeddings, both during training and application, by means of a random process, and is thus fundamentally different from this idea.

4 Reasoning with RTNs

4.1 Applying RTNs to OKBs

As discussed in Section 2.1, OKBs can be viewed as DAGs, and thus the application of an RTN to this kind of data is straightforward. Therefore, we are only left with specifying the prediction model that we want to use on top of the RTN. In the context of an OKB, there are two kinds of predictions that we are interested in, namely the membership of individuals to classes, on the one hand, and the existence of relations, on the other hand. From a ML perspective, these are really two different targets, and we can describe them more formally as follows: let $\mathcal{K}$ be an OKB that contains (exactly) the unary predicates $P_1, \ldots, P_k$ and (exactly) the binary predicates $Q_1, \ldots, Q_\ell$, and $\mathcal{T} \subseteq \mathcal{K}$ the part of the OKB that we have as training set. Then $t^{(1)}$ and $t^{(2)}$ are two target functions defined as

$$t^{(1)} : \begin{cases} individuals(\mathcal{K}) \to \{-1, 0, 1\}^k \\ i \mapsto \mathbf{x}^{(i)} \end{cases}$$

and

$$t^{(2)} : \begin{cases} individuals(\mathcal{K})^2 \to \{-1, 0, 1\}^\ell \\ (i, j) \mapsto \mathbf{y}^{(i,j)} \end{cases}$$

such that $\mathbf{x}_m^{(i)}$ equals 1, if $\mathcal{K} \models P_m(i)$, -1, if $\mathcal{K} \models \neg P_m(i)$, and 0, otherwise, and $\mathbf{y}_m^{(i,j)}$ is defined accordingly with respect to $Q_m(i, j)$.

Notice that all of the arguments of the functions $t^{(1)}$ and $t^{(2)}$ are individuals, and can thus be represented as embeddings produced by an RTN. For computing actual predictions from these embeddings, we can basically employ an ML model of our choice. In this work, however, we confine ourselves to multinomial logistic regression for $t^{(1)}$, i.e., we simply add a single feed-forward layer as well as a softmax on top it to the RTN. For $t^{(2)}$, we first add an additional original tensor layer as given in Equation 1, like it was used by Socher *et al.* (2013), and use multinomial logistic regression on top of it as well.

4.2 Predicting Classes and Relations Simultaneously

While the targets $t^{(1)}$ and $t^{(2)}$ may be regarded as independent with respect to prediction, this is clearly not the case for computing individual embeddings. We require an embedding to reflect all of the information that we have about a single individual as specified by the semantics of the considered OKB. Therefore, the tensor layers of an RTN need to learn how to adjust individual vectors in view of both unary and binary predicates, i.e., classes and relations. To account for this, we train RTNs—facing the particular use case of ontology reasoning—on mini-batches that consist of training samples for both of the prediction targets.

5 Evaluation

To evaluate the suggested approach in a realistic scenario, we implemented a novel triple store, called **NeTS** (**Ne**ural **T**riple **S**tore), that achieves ontology reasoning solely by means of an RTN. NeTS provides a simple, SPARQL-like, query interface that allows for submitting atomic queries as well as conjunctions of such (see Figure 1).

```
NeTS> dbpedia:Person(?X),dbpedia:placeOfBirth(?X,?Y)

?X                        ?Y
=======================   ============================
dbpedia:Aristotle         dbpedia:Stagira_(ancient_city)
dbpedia:Albert_Einstein   dbpedia:Ulm
⋮                         ⋮
```

Figure 1: Example of a simple query in NeTS.

When the system is started, then the first step it performs is to load a set of learned weights from the disk—the actual learning process is not part of NeTS right now, and may be incorporated in future versions. Next, it observes whether there are previously generated embeddings of the individuals stored on disk already, and loads them as well, if any. If this is not the case, however, then NeTS creates such embeddings as described above. This step is comparable with what is usually referred to as materialization in the context of database systems. Traditionally, a database would compute all valid inferences that one may draw based on the provided data, and store them somehow in memory or on disk. In contrast to this, NeTS accounts for these inferences simply by adjusting the individuals' embeddings by means of a trained RTN, which obviously has great advantages regarding its memory requirements. Note further that we do not store any actual inferences at this time, but rather compute them on demand later on if this happens to become necessary.

Subsequent processing of queries is entirely based on these embeddings, and does not employ any kind of formal reasoning at all. This, in turn, allows for speeding up the necessary computations significantly, since we can dispatch most of the the »*heavy-lifting*« to a GPU.

Our system is implemented in Python 3.4, and performs, as mentioned above, almost all numeric computations on a GPU using PyCUDA 2016.1.2 (Klöckner *et al.*, 2012). For learning the weights of our RTNs, we again used Python 3.4, along with TensorFlow 0.11.0 (Abadi *et al.*, 2015).

5.1 Test Data

To maintain comparability, we evaluated our approach on the same datasets that Motik *et al.* (2014) used for their experiments with RDFox (Nenov *et al.*, 2015).[2] As mentioned earlier, RDFox is indeed a great benchmark, since it has been shown to be the most efficient triple store at present. For a comparison with other systems, however, we refer the interested reader to Motik *et al.* (2014).

The test data consists of four Semantic Web KBs of different sizes and characteristics. Among these are two real-world datasets, a fraction of DBpedia (Bizer *et al.*, 2009) and the Claros KB[3], as well as two synthetic ones, LUBM (Guo *et al.*, 2005) and UOBM (Ma *et al.*, 2006). Their characteristics are summarized in Table 1.

While all these data are available in multiple formats, we made use of the ontologies specified in OWL and the facts provided as n-triples for our experiments. Furthermore, we considered only those predicates that appear for at least 5% of the individuals in a database. This is a necessary restriction to ensure that there is enough data for an RTN to learn properly.

5.2 Experimental Setup

All our experiments were conducted on a server with 24 CPUs of type Intel Xeon E5-2620 (6×2.40GHz), 64GB of RAM, and an Nvidia GeForce GTX Titan X. The test system hosted Ubuntu Server 14.04 LTS (64 Bit) with CUDA 8.0 and cuDNN 5.1 for GPGPU. Notice, however, that NeTS does not make any use of multiprocessing or -threading besides GPGPU, which means that the only kind of parallelization takes place on the GPU. Therefore, in terms of CPU and RAM, NeTS had about half of the resources at its disposal that RDFox utilized in the experiments conducted by Motik *et al.* (2014).

[2] All of these datasets are available at `http://www.cs.ox.ac.uk/isg/tools/RDFox/2014/AAAI/`.

[3] `http://www.clarosnet.org`

	Claros	DBpedia	LUBM	UOBM
KRR formalism	OWL	OWL 2	OWL	OWL
# of Individuals	6.5 M	18.7 M	32.9 M	0.4 M
# of Facts	18.8 M	112.7 M	133.6M	2.2 M
# of Classes	40 (13)	349 (12)	14 (4)	39 (5)
# of Relations	64 (20)	13616 (16)	13 (6)	22 (11)

Table 1: Characteristics of the test datasets. All quantities refer to explicitly specified rather than inferred data, and the values in parentheses describe the classes and relations, respectively, that appear with at least 5% of the individuals.

	Classes		Relations	
	Avg. Accuracy	Avg. F1	Avg. Accuracy	Avg. F1
Claros	0.969	0.954	0.955	0.942
DBpedia	0.978	0.959	0.961	0.940
LUBM	0.961	0.948	0.959	0.947
OUBM	0.972	0.953	0.973	0.951

Table 2: The accuracies and F1 scores, averaged over all unary and binary predicates, respectively, for each dataset.

Predicated on the use of the RTN model, the datasets, including all of their inferences, were converted into directed graphs using Apache Jena 2.13.0[4] and the OWL reasoner Pellet 2.4.0[5]—all of the import times reported in Table 3 refer to these graphs. This reduced the size of the data, as stored on disk, to approximately on third of the original dataset. Furthermore, we removed a total of 50,000 individuals during training, together with all of the predicates that these were involved in, as test set from each of the datasets, and similarly another 50,000 for validation—the results described in Table 2 were retrieved for these test sets.

5.3 Results

In order to assess the quality of NeTS, we have to evaluate it on two accounts. First, we need to consider its predictive performance based on the embeddings computed by the underlying RTN model, and second, we must ascertain the efficiency of the system with respect to time consumption.

We start with the former. To that end, consider Table 2, which reports the accuracies as well as F1 scores that NeTS achieved on the held-out test sets, averaged over all classes and relations, respectively. We see that the model consistently achieves great scores with respect to both measures. Notice, however, that the F1 score is the more critical criterion, since all the predicates are strongly imbalanced. Nevertheless, the RTN effectively learns embeddings that allow for discriminating positive from negative instances.

Table 3, in contrast, lists the times for NeTS to import and materialize each of the datasets along with the respective measurements for RDFox (Motik *et al.*, 2014). As mentioned before, materialization refers to the actual computation of inferences, and usually depends on the expressivity of the ontology as well as the number of facts available. We see that NeTS is significantly faster at the materialization step, while RDFox is faster at importing the data. This is explained as follows. First, NeTS realizes reasoning by means of vector manipulations on a GPU, which is of course much faster than the symbolic computations performed by RDFox. As for the second point, RDFox makes use of extensive parallelization, also for importing data, while NeTS runs as a single process with a single thread on a CPU.

[4] `https://jena.apache.org`

[5] `https://github.com/Complexible/pellet`

	NeTS		RDFox			
	Import	Materialization	Import	Materialization		
Claros	242	28	48	2062	/	—
DBpedia	436	69	274	143	/	—
LUBM	521	52	332	71	/	113
OUBM	9	11	5	467	/	2501

Table 3: The times for import and materialization (in seconds). For RDFox, these are the numbers reported by Motik *et al.* (2014) for computing a lower (left) and upper bound (right), respectively, on the possible inferences.

However, from a practical point of view, materialization is usually more critical than import. This is because an average database is updated with new facts quite frequently, while it is imported only once in a while.

Notice, however, that neither of the measures reported for NeTS contains the time for training the model. The reason for this is that we train an RTN, as mentioned earlier, with respect to an ontology rather than an entire OKB. Therefore, one can actually consider the training step as part of the setup of the database system. For the datasets used in our experiments, training took between three and four days each.

6 Summary and Outlook

We have presented a novel method for SRL based on deep learning, and used it to develop a highly efficient, learning-based system for ontology reasoning. Furthermore, we have provided an experimental comparison with one of the best logic-based ontology reasoners at present, RDFox, on several large standard benchmarks, and showed that our approach attains a high reasoning quality while being up to two orders of magnitude faster.

An interesting topic for future research is to explore ways to further improve our accuracy on ontology reasoning. This could be achieved, e.g., by incorporating additional synthetic data and/or slight refinements of the RTN architecture.

Acknowledgments

This work was supported by the Engineering and Physical Sciences Research Council (EPSRC), under the grants EP/J008346/1, EP/L012138/1, and EP/M025268/1, as well as the Alan Turing Institute, under the EPSRC grant EP/N510129/1. Furthermore, Patrick is supported by the EPSRC, under grant OUCL/2016/PH, and the Oxford-DeepMind Graduate Scholarship, under grant GAF1617_OGSMF-DMCS_1036172.

References

Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

Franz Baader, Diego Calvanese, Deborah L. McGuinness, Daniele Nardi, and Peter F. Patel-Schneider. *The Description Logic Handbook: Theory, Implementation, and Applications*. Cambridge University Press, 2nd edition, 2007.

Christian Bizer, Jens Lehmann, Georgi Kobilarov, Sören Auer, Christian Becker, Richard Cyganiak, and Sebastian Hellmann. DBpedia—A crystallization point for the Web of Data. *Web Semantics: Science, Services and Agents on the World Wide Web*, 7(3):154–165, 2009.

Samuel R. Bowman. Can recursive neural tensor networks learn logical reasoning? http://arxiv.org/abs/1312.6192v4, 2013.

Evgeniy Gabrilovoch, Ramanathan Guha, Andrew McCallum, and Kevin Murphy, editors. *Knowledge Representation and Reasoning: Integrating Symbolic and Neural Approaches*, Palo Alto, California, 2015. AAAI Press.

Lise Getoor and Ben Taskar. *Introduction to Statistical Relational Learning*. Adaptive Computation and Machine Learning. MIT Press, 2007.

Christoph Goller and Andreas Küchler. Learning Task-Dependent Distributed Representations by Backpropagation Through Structure. In *IEEE International Conference on Neural Networks*, volume 1, pages 347–352, 1996.

Yuanbo Guo, Zhengxiang Pan, and Jeff Heflin. LUBM: A benchmark for OWL knowledge base systems. *Web Semantics: Science, Services and Agents on the World Wide Web*, 3(2–3):158–182, 2005.

Andreas Klöckner, Nicolas Pinto, Yunsup Lee, B. Catanzaro, Paul Ivanov, and Ahmed Fasih. PyCUDA and PyOpenCL: A Scripting-Based Approach to GPU Run-Time Code Generation. *Parallel Computing*, 38(3):157–174, 2012.

Li Ma, Yang Yang, Zhaoming Qiu, Guotong Xie, Yue Pan, and Shengping Liu. Towards a Complete OWL Ontology Benchmark. In *Proceedings of the 3rd European Semantic Web Conference (ESWC 2006)*, pages 125–139, 2006.

Boris Motik, Yavor Nenov, Robert Piro, Ian Horrocks, and Dan Olteanu. Parallel Materialisation of Datalog Programs in Centralised, Main-Memory RDF Systems. In *Proceedings of the 28th AAAI Conference on Artificial Intelligence (AAAI 2014)*, pages 129–137, 2014.

Yavor Nenov, Robert Piro, Boris Motik, Ian Horrocks, Zhe Wu, and Jay Banerjee. RDFox: A Highly-Scalable RDF Store. In *Proceedings of the 14th International Semantic Web Conference (ISWC 2015), Part II*, pages 3–20, 2015.

Maximilian Nickel, Volker Tresp, and Hans-Peter Kriegel. A Three-Way Model for Collective Learning on Multi-Relational Data. In *Proceedings of the 28th International Conference on Machine Learning*, pages 809–816, 2011.

Maximilian Nickel, Volker Tresp, and Hans-Peter Kriegel. Factorizing YAGO. In *Proceedings of the 21st International Conference on World Wide Web*, pages 271–280, 2012.

Maximilian Nickel, Kevin Murphy, Volker Tresp, and Evgeniy Gabrilovich. A Review of Relational Machine Learning for Knowledge Graphs. *Proceedings of the IEEE*, 104(1):11–33, 2016.

Jordan B. Pollack. Recursive distributed representations. *Artificial Intelligence*, 46(1):77–105, 1990.

Luciano Serafini and Artur d'Avila Garcez. Logic tensor networks: Deep learning and logical reasoning from data and knowledge. arXiv:1606.04422v2, 2016.

Richard Socher, Danqi Chen, Christopher D. Manning, and Andrew Y. Ng. Reasoning with neural tensor networks for knowledge base completion. In *Advances in Neural Information Processing Systems 26*, pages 926–934. 2013.