Deep Learning of RDFS rules

Bassem Makni, James Hendler
Rensselaer Polytechnic Institute
maknib@rpi.edu, hendler@cs.rpi.edu

Abstract

In this paper we present a novel approach of RDFS reasoning using sequence to sequence learning. Recurrent neural networks are being used successfully for Natural Language Processing tasks. We extend their use for rules learning, namely RDFS rules. We reach a materialization level of LUBM similar to OWLIM.

1 Introduction

After a few decades of low popularity relative to other machine learning methods, neural networks are trending again with the Deep Learning movement. The advances are mainly due to hardware improvements, especially in GPUs, and to the algorithmic solutions to the vanishing gradients problem [Pascanu et al., 2012]. Convolutional nets are being used successfully for image classification [Krizhevsky et al., 2012] and object categorization [Huang and LeCun, 2006]. While recurrent networks are used for sequence learning, such as videos, texts and speech recognition. The most used recurrent networks architectures are Long Short Term Memory (LSTM) [Gers et al., 2000] and Gated Recurrent Unit (GRU) [Chung et al., 2014]. Recurrent networks are used for various natural processing tasks, from sentiment analysis to language modeling and even question answering. When the input and output of the data is modeled as a sequence such in the translation task, Sequence to Sequence [Sutskever et al., 2014] learning is used. Though RDF graphs are not sequences per se, we are using in this paper sequences of triples input in order to generate sequences of inferred triples.

2 Approach

Our goal is to design a recurrent network able to learn the entailments of the different RDFS rules and then use the network to generate the materialization of an RDF graph. Our evaluation measure will be:

- Precision recall: where we focus on how much entailments can we generate compared to other semantic inference engines such as OWLIM [Kiryakov et al., 2005] and OpenRDF [Broekstra et al., 2002] and check if we generated any noise triples.
- Speed performance: where we compare our materialization speeds to other parallel materialization algorithms such as "Parallel materialization of the finite RDFS closure for hundreds of millions of triples" [Weaver and Hendler, 2009]

To achieve these goals, our approach consists of three phases:

Preparing the RDFS ground-truth In this phase, we collect RDF triples that match the different RDFS rules patterns. The W3C recommendation "RDF 1.1 Semantics" [Hayes and Patel-Schneider, 2014] lists 13 RDFS rules (Table 1).

As the rules RDFS4a and RDFS4b, RDFS8 and RDFS10 have the same input pattern but different entailments, we transform them to one input pattern with two entailments. We also neglect some rules such as RDFS:ContainerMembershipProperty which does not have any instances in the SPARQL endpoints we used to collect the ground-truth nor in the LUBM [Guo et al., 2005] ontology. The subset of modified rules we used is listed in Table 2.

We used a combination of triples collected from DBpedia [Lehmann et al., 2015] and from BBC Backstage SPARQL endpoint for programmes and music [Smethurst, 2012]. The reason we needed this combination is that the DBpedia ontology is an upper level ontology, and we did not find enough samples for RDFS:subClassOf and RDFS:subPropertyOf properties, while the BBC ontology is a domain ontology, and its SPARQL endpoint contains more samples of these properties.

For each pattern in Table 2, we collected 10 thousands samples.

Recurrent network design To design our recurrent network, we used Keras [Chollet, 2015], a deep learning framework that facilitates the design of neural network models and can run the model using Theano [Bergstra et al., 2011] or TensorFlow [Abadi et al., ] backends.

Our model illustrated in Figure 1 is a sequential model.
If any IRI aaa in D
Then aaa RDF:type RDFS:Datatype.

If aaa RDFS:domain xxx.
yyy aaa zzz.
Then zzz RDF:type xxx.

If aaa RDFS:range xxx.
yyy aaa zzz.
Then yyy RDF:type xxx.

If aaa RDFS:domain xxx.
yyy aaa zzz.
Then yyy RDF:type xxx.

If xxx aaa yyy.
Then xxx RDF:type RDFS:Resource.

If xxx RDF:type RDFS:Resource.
Then xxx RDF:type RDFS:Property.

If xxx RDFS:subPropertyOf yyy.
Then xxx RDF:type RDFS:Property.

If xxx RDF:type RDF:Property.
Then xxx RDF:subPropertyOf RDFS:Resource.

If xxx RDF:subPropertyOf RDFS:Resource.
Then xxx RDF:subPropertyOf RDFS:Resource.

If xxx RDF:subPropertyOf RDFS:Resource.
Then xxx RDF:subPropertyOf RDFS:Resource.

If xxx RDF:subPropertyOf RDFS:Resource.
Then xxx RDF:subPropertyOf RDFS:Resource.

If xxx RDF:subPropertyOf RDFS:Resource.
Then xxx RDF:subPropertyOf RDFS:Resource.

If xxx RDF:subPropertyOf RDFS:Resource.
Then xxx RDF:subPropertyOf RDFS:Resource.

If xxx RDF:subPropertyOf RDFS:Resource.
Then xxx RDF:subPropertyOf RDFS:Resource.

If xxx RDF:subPropertyOf RDFS:Resource.
Then xxx RDF:subPropertyOf RDFS:Resource.

If xxx RDF:subPropertyOf RDFS:Resource.
Then xxx RDF:subPropertyOf RDFS:Resource.

If xxx RDF:subPropertyOf RDFS:Resource.
Then xxx RDF:subPropertyOf RDFS:Resource.

If xxx RDF:subPropertyOf RDFS:Resource.
Then xxx RDF:subPropertyOf RDFS:Resource.

If xxx RDF:subPropertyOf RDFS:Resource.
Then xxx RDF:subPropertyOf RDFS:Resource.

If xxx RDF:subPropertyOf RDFS:Resource.
Then xxx RDF:subPropertyOf RDFS:Resource.

If xxx RDF:subPropertyOf RDFS:Resource.
Then xxx RDF:subPropertyOf RDFS:Resource.

If xxx RDF:subPropertyOf RDFS:Resource.
Then xxx RDF:subPropertyOf RDFS:Resource.

If xxx RDF:subPropertyOf RDFS:Resource.
Then xxx RDF:subPropertyOf RDFS:Resource.

If xxx RDF:subPropertyOf RDFS:Resource.
Then xxx RDF:subPropertyOf RDFS:Resource.

If xxx RDF:subPropertyOf RDFS:Resource.
Then xxx RDF:subPropertyOf RDFS:Resource.

If xxx RDF:subPropertyOf RDFS:Resource.
Then xxx RDF:subPropertyOf RDFS:Resource.

If xxx RDF:subPropertyOf RDFS:Resource.
Then xxx RDF:subPropertyOf RDFS:Resource.

If xxx RDF:subPropertyOf RDFS:Resource.
Then xxx RDF:subPropertyOf RDFS:Resource.

If xxx RDF:subPropertyOf RDFS:Resource.
Then xxx RDF:subPropertyOf RDFS:Resource.

If xxx RDF:subPropertyOf RDFS:Resource.
Then xxx RDF:subPropertyOf RDFS:Resource.

If xxx RDF:subPropertyOf RDFS:Resource.
Then xxx RDF:subPropertyOf RDFS:Resource.

If xxx RDF:subPropertyOf RDFS:Resource.
Then xxx RDF:subPropertyOf RDFS:Resource.

If xxx RDF:subPropertyOf RDFS:Resource.
Then xxx RDF:subPropertyOf RDFS:Resource.

If xxx RDF:subPropertyOf RDFS:Resource.
Then xxx RDF:subPropertyOf RDFS:Resource.

If xxx RDF:subPropertyOf RDFS:Resource.
Then xxx RDF:subPropertyOf RDFS:Resource.

Table 1: From [Hayes and Patel-Schneider, 2014]

starting with a masking layer. The masking layer allows
the input of the recurrent layer to be of variable length.
As some of the rules in Table 2 have two triples (6 re-
sources) and some have only one triple (3 resources),
we pad the 3 entities with 3 zeros, and we use a mask
value of zero. The third parameter of the masking layer
"13" is the size of the one hot vector, that we will detail
in the encoding/decoding section.
The second layer is the input LSTM layer with 256 hid-
den units. We varied the hidden units hyper-parameter
between 32 and 1024. Lower values harm the accuracy
while higher values give better accuracy and are slower
in training. 256 was a good trade-off for accuracy and
training speed.

With the repeat vector layer we repeat the encoded input
for each output time-step. The second LSTM layer is the
output recurrent layer. The TimeDistributedDense layer
chooses for each time-step which resource encoding to
output. And lastly, the softmax activation layer selects
the most probable class from the 13 possible classes.

Table 2: Ground Truth collection patterns

Encoding/Decoding To encode the ground truth data col-
lected in the first phase for the network input, we tried
many approaches:

- One hot encoding (global): We assign an id for each
  RDF resource in the full ground truth, and we use
  these ids to generate one hot encoded vectors for
  each input and output. This leads to very big vec-
  tors and learning become very slow. To minimize
  the size of these vectors we tried:
- Word2Vec: we encode the full ground truth using
  Word2Vec [Mikolov et al., 2013] algorithm where
  we treat each triple as one sentence. With this en-
  coding we achieved good performance, accuracy
  and validation accuracy on the ground truth, but
  when testing with LUBM, we got poor test accu-
  racy. The reason is that the word vectors generated
  for LUBM were very different from those of DB-
  pedia+BBC.
- One hot encoding (local): A simpler approach
  that performs much better with good accuracy
  and learning speed is to encode each input sep-
  arately. We keep a hash of the RDF resources
  used in the entailments namely: RDF:type,
  RDFS:domain, RDFS:range, RDFS:Resource,
  RDFS:subPropertyOf, RDFS:subClassOf,
  RDF:Property and RDFS:Class. To these 8
  resources, a maximum of 4 new resources can
  be added when encoding the rule RDFS2 input,
  plus the padding value, hence the vector size of 13

| RDFS1 | If any IRI aaa in D
Then aaa RDF:type RDFS:Datatype. |
|-------|---------------------------------|
| RDFS2 | If aaa RDFS:domain xxx.
yyy aaa zzz.
Then zzz RDF:type xxx. |
| RDFS3 | If aaa RDFS:range xxx.
yyy aaa zzz.
Then yyy RDF:type xxx. |
| RDFS4 | If xxx aaa yyy.
Then xxx RDF:type RDFS:Resource. |
| RDFS5 | If xxx RDF:subPropertyOf yyy.
yyy RDF:subPropertyOf zzz.
Then xxx RDF:subPropertyOf zzz. |
| RDFS6 | If xxx RDF:subPropertyOf RDFS:Resource.
Then xxx RDF:subPropertyOf RDFS:Resource. |
| RDFS7 | If xxx RDF:subPropertyOf RDFS:Resource.
Then xxx RDF:subPropertyOf RDFS:Resource. |
| RDFS8 | If xxx RDF:subPropertyOf RDFS:Resource.
Then xxx RDF:subPropertyOf RDFS:Resource. |
| RDFS9 | If xxx RDF:subPropertyOf RDFS:Resource.
Then xxx RDF:subPropertyOf RDFS:Resource. |
| RDFS10 | If xxx RDF:subPropertyOf RDFS:Resource.
Then xxx RDF:subPropertyOf RDFS:Resource. |
| RDFS11 | If xxx RDF:subPropertyOf RDFS:Resource.
Then xxx RDF:subPropertyOf RDFS:Resource. |
| RDFS12 | If xxx RDF:subPropertyOf RDFS:ContainerMembershipProperty.
Then xxx RDF:subPropertyOf RDFS:member. |
| RDFS13 | If xxx RDF:type RDFS:Datatype. 
Then xxx RDF:subClassOf RDFS:Literal. |
for the one hot vector in the model. It is crucial that we keep the encoding of the RDF/RDFS resources consistent throughout the training. For each encoded input we maintain a reverse hash that we use to decode the model output.

**Incremental materialization** After the training of our network on the DBpedia+BBC ground truth, we use the network to materialize the LUBM graph. Our incremental materialization (Figure 2) consists of:

1. Running a SPARQL query (Listing 1) to collect all rules patterns against an RDF store hosting the LUBM graph.
2. Encoding the collected triples.
3. Using the trained network to generate new triples encodings.
4. Decoding the output to obtain RDF triples.
5. Inserting the generated triples back in the RDF store.

We repeat these steps till step 1 does not retrieve new triples anymore.

```python
PREFIX RDFS: <http://www.w3.org/2000/01/RDF-schema#>
PREFIX RDF: <http://www.w3.org/1999/02/22-RDF-syntax-ns#>

select distinct ?s1 ?p1 ?o1 ?s2 ?p2 ?o2 where {
  { ?s1 RDFS:domain ?o1 .
    ?s2 ?s1 ?o2
    bind(RDFS:domain as ?p1 )
    bind(?s1 as ?p2 ) } 
union {
  ?s1 RDFS:range ?o1 .
  ?s2 ?s1 ?o2
  bind(RDFS:range as ?p1)
  bind(?s1 as ?p2 ) }
union {
  ?s1 RDF:type RDF:Property
  .
  filter(isURI(?o1))}
union {
  ?o1 RDF:subPropertyOf ?o2 .
  bind(RDFS:subPropertyOf as ?p1)
  bind(RDFS:subPropertyOf as ?p2)
  bind(?o1 as ?s2) }
union {
  ?s1 RDF:subPropertyOf ?o1 .
  ?o1 RDF:subPropertyOf ?o2 .
  bind(RDFS:subPropertyOf as ?p1)
  bind(RDFS:subPropertyOf as ?p2)
  bind(?o1 as ?s2) }
union {
  ?s1 RDF:subPropertyOf ?o1 .
  ?o1 RDF:subPropertyOf ?o2 .
  bind(RDFS:subPropertyOf as ?p1)
  bind(RDFS:subPropertyOf as ?p2)
  bind(?o1 as ?s2) }
union {
  ?s1 RDF:subPropertyOf ?o1 .
  ?o1 RDF:subPropertyOf ?o2 .
  bind(RDFS:subPropertyOf as ?p1)
  bind(RDFS:subPropertyOf as ?p2)
  bind(?o1 as ?s2) }
union {
  ?s1 RDF:subPropertyOf ?o1 .
  ?o1 RDF:subPropertyOf ?o2 .
  bind(RDFS:subPropertyOf as ?p1)
  bind(RDFS:subPropertyOf as ?p2)
  bind(?o1 as ?s2) }
```

Figure 1: Sequence to Sequence recurrent model

Figure 2: Incremental materialization flowchart
3 Results

We run most of our experiments using a desktop GPU "GeForce 840M", which has 2Gb of memory and 3 multiprocessors with 128 CUDA cores each. We used also a server GPU "Titan X Black" for some experiments. The learning and evaluation runs twice fast on the Titan compared to the Geforce 840M.

As of the software stack we used, it contains:

- keras 1.0.1 [Chollet, 2015]
- Theano 0.8.1 [Bergstra et al., 2011]
- TensorFlow 0.8.0 [Abadi et al., 1]
- CUDA 7.5 [Nvidia, 2007]

3.1 RDFS Learning

From the data collected in the DBpedia+BBC ground truth, we use 20% for validation test. The training process takes less than 10 minutes with 10 iterations over the data, and we reach 0.998 validation accuracy (Figure 3)

3.2 Incremental RDFS materialization of LUBM1

Lehigh University Benchmark (LUBM) is a benchmark for Semantic Web repositories, it can be used to generate RDF graphs using an ontology of the academic domain. LUBM1 is generated using one university, and contains 100 thousands triples. OWLIM generates 41 113 thousands inferred triples. When running our incremental materialization, we needed 3 iterations (Figure 4) to achieve the stagnation of the materialization. The total number of generated inferences is 48 972 triples. And all of them are valid. Some of them are already explicit triples, this is why the total number of inferred triples is higher than the number of implicit triples from OWLIM. Our algorithm missed 369 triples that were inferred by OWLIM. Most of the missed triples are RDFS axioms. Which gives us a 0.992 level of materialization compared to OWLIM.
3.3 Inference examples
When running our incremental materialization on these RDF triples, we obtain the corresponding entailments.

RDFS2 rule
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix lubm: <http://www.Department3.University0.edu> .
@prefix dep3: <http://swat.cse.lehigh.edu/onto/univ-bench.owl#> .
@prefix fullProf0: <http://www.Department3.University0.edu/FullProfessor> .

lubm:publicationAuthor rdfs:domain lubm:Publication .
fullProf0:Publication0 lubm:publicationAuthor dep3:FullProfessor0 .
⇒

@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix lubm: <http://www.Department3.University0.edu> .
@prefix dep3: <http://swat.cse.lehigh.edu/onto/univ-bench.owl#> .
@prefix fullProf0: <http://www.Department3.University0.edu/FullProfessor> .

dlubm:undergraduateDegreeFrom rdfs:range lubm:University .
⇒

@prefix lubm: <http://www.Department3.University0.edu> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .

4 Conclusions and Future work
Our prototype proves that sequence to sequence neural networks can learn RDFS rules and be used for RDF graph materialization. We are currently experimenting with larger LUBM graphs with 5 thousands universities and comparing our materialization speeds to the state of the art in parallel materialization.

5 Acknowledgment
This work was supported by the Air Force Research Laboratory (AFRL), under award number FA8750-15-2-0078.

References

