Reimplementing Neural Tensor Networks for Knowledge Base Completion in the TensorFlow framework

Dustin Doss, Alex LeNail, Clare Liu

December 2015

Abstract

Reasoning with Neural Tensor Networks for Knowledge Base Completion has become something of a seminal paper in the short span of two years, cited by nearly every knowledge base completion (KBC) paper since its publication in 2013. It was one of the first major successful forays into the field of Deep Learning in approaching knowledge base completion, and was unique for using deep learning "end to end".

TensorFlow is a tensor-oriented numerical computation library recently released by Google. It represents algorithms as directed acyclic graphs (DAGs), nodes as operations and edges as schemas for tensors. It has a robust python API and bindings to GPUs.

We reimplemented Socher's algorithm in the TensorFlow framework with an elegant implementation in a modern language, achieving accuracy results significantly better than random guessing. Our code is freely available on MIT's Github. [1]

1 Introduction

We were initially intrigued by a Kaggle competition called the Allen AI Science Challenge (posed by the Allen Institute in Seattle), the goal of which is to answer 8th grade multiple-choice science questions with minimal training data, but access to knowledge bases (3). Two months after the competition commenced, (with an obvious baseline of 25 percent) top scores hardly exceeded 50 percent. We initially thought we might attempt to submit an entry to the challenge, but have since backed off to what we think is likely the key missing component in others' approaches: complete knowledge bases. We suspect that given a complete knowledge base, a knowledge base augmented by some inference algorithm to complete missing edges, we might be able to supersede the 50 percent threshold.

A knowledge base is a representation of factual knowledge, traditionally in a graph-like structure. In a knowledge base, entities are represented as nodes and

[1] https://github.mit.edu/clareliu/6.806_knowledge_base_completion

relations as edges. There are a discrete number of relation types, usually a quite small set of them. Knowledge bases characteristically suffer from incompleteness, in the form of missing edges. If A is related to B, and B is related to C, oftentimes A is related to C, but knowledge bases often don't have these relations explicitly listed, because they're simply common sense, and can easily be inferred by a human. This is the "common sense" often missing in Artificially Intelligent systems, especially question answering systems, which rely on knowledge bases heavily.

There are a variety of open knowledge bases available today, including but not limited to YAGO, Wordnet, and Freebase. These are often developed by hand. Freebase, for example, was put together by contractors working for Google, seeking to improve search results with richer understandings of entities in search queries.

Humans overlook facts we consider "obvious," for example the knowledge base may specify that "MIT" is "located in" "Cambridge" and "Cambridge" is "located in" "Massachusetts" but may neglect to draw another "located in" edge from "MIT" to "Massachusetts". This is the very simplest kind of missing edge we might encounter in a knowledge base, and we'd like to develop a method to predict the likely truth of new facts with more complicated structure, in effect, reasoning over known facts and inferring new ones.

One approach to knowledge base completion involves traversing edges of the knowledge graph and composing their scoring functions to predict new edges (4). However, we believe that more interesting relationships between entities can be inferred using a neural network model. Socher et al's paper *Reasoning with Neural Tensor Networks for Knowledge Base Completion* presents an interesting deep learning approach to knowledge base completion. We reimplemented the algorithm described in Socher's paper using TensorFlow, an open-source machine learning library that represents algorithms as graphs and includes powerful features such as auto-differentiation, which can be used to simplify the backpropagation step in deep learning.

2 Background

This section summarizes Socher's neural tensor network model for knowledge base completion, which we reimplemented using TensorFlow.

2.1 Overview

Entities are represented as nodes in a knowledge graph while relations are represented as labeled, directed edges. Therefore, a specific relationship can be defined as the triple (e_1, R, e_2), where e_1 and e_2 are the two entities and R is the relation between them. Socher's model learns vector representations for entities in a knowledge base to predict entity-relationship triplets using a neural tensor network (NTN).

Unlike previous knowledge base completion approaches, where an entity is represented as a single vector, Socher's method represents an entity as the average of its word vectors. This allows for the sharing of statistical strength between entities containing common words (6). For example, even though "tiger" and "Bengal tiger" are different entities in a knowledge base, there are many similarities between a generic tiger and a Bengal tiger, so they should share many of the same relationships. Figure 1 shows a visual depiction of Socher's method.

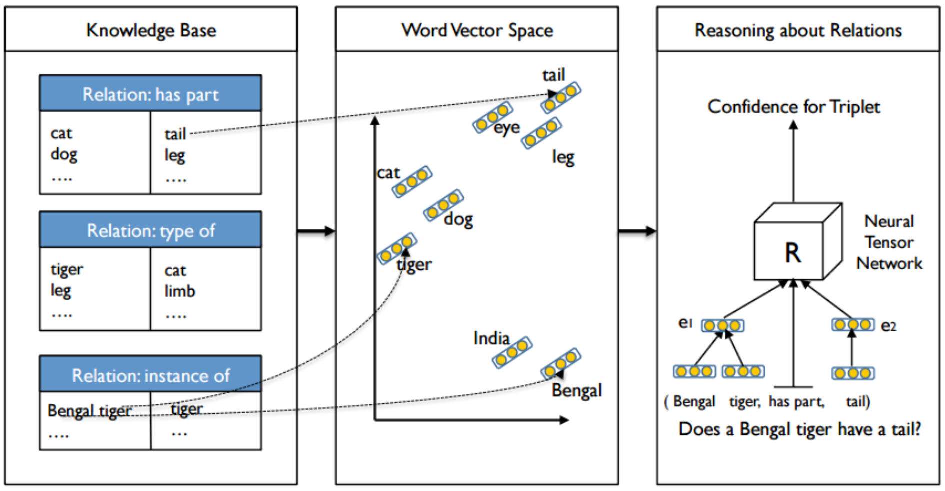

Figure 1: This diagram shows how words in a knowledge base are mapped to vectors and averaged to construct entity vectors. Entity relation triples are passed to a neural tensor network, which calculates the confidence that the two entities are in a relationship

2.2 Neural Tensor Network Model

Socher's Neural Tensor Network (defined below) uses a bilinear tensor layer to relate the two entity vectors across multiple dimensions. The function g represents the confidence that the entities e_1 and e_2 are in the relationship R. (6).

$$g(e_1, R, e_2) = u_R^T f(e_1^T W_R^{[1:k]} e_2 + V_R \begin{pmatrix} e_1 \\ e_2 \end{pmatrix} + b_R) \tag{1}$$

f represents the tanh nonlinearity function applied elementwise. u and B are vectors in $\mathbb{R}^k$, e_1 and e_2 are d-dimensional embeddings of entities, and W is a $\mathbb{R}^{d \times d \times k}$ tensor.

$e_1^T W_R^{[1:k]} e_2$, the bilinear tensor product, computes a vector h in R^k, where k is the depth of the tensor (number of slices). Each entry in h is equal to $e_1^T W_R^{[i]} e_2$, a product which results in a scalar. (6).

V is a matrix in $\mathbb{R}^{k \times 2d}$, and the product of V and $\begin{pmatrix} e_1 \\ e_2 \end{pmatrix}$ adds a linear neural-network layer to the neural tensor network's output. A visualization of this model is shown in Figure 2.

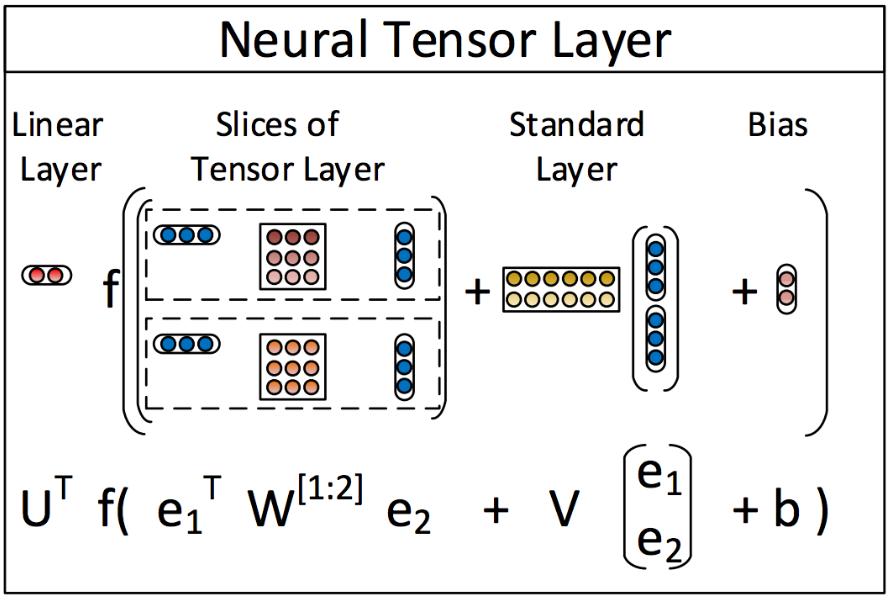
Neural Tensor Layer
Linear Layer
Slices of Tensor Layer
Standard Layer
Bias
f
+
+
U^T f(e_1^T $W^{[1:2]}$ e_2 + V $\begin{pmatrix} e_1 \\ e_2 \end{pmatrix}$ + b)

Figure 2: A visualization of the parameters of Socher's Neural Tensor Network Model with $k = 2$ slices.

2.3 Objective Function

To evaluate the model, triples are corrupted by swapping random entities between them. For example, the triples *(Pablo Picasso, nationality, Spain)* and *(Barack Obama, nationality, United States)* can be corrupted by swapping "Spain" and "United States", resulting in the incorrect triples *(Pablo Picasso, nationality, United States)* and *(Barack Obama, nationality, Spain)*. Socher's model optimizes the NTN parameters $\Omega = (u, W, V, b, E)$ by minimizing the following objective function (6):

$$J(\Omega) = \Sigma_{i=1}^{N} \Sigma_{c=1}^{C} max(0, 1 - g(T^{(i)}) + g(T_c^{(i)})) + \lambda ||\Omega||_2^2 \tag{2}$$

N is the number of true training triples, C is the number of corrupt false examples generated for each true triple, and λ is a regularization parameter which modulates the L2 norm penalty for the size of the parameters.

$g(T^{(i)})$ represents the confidence for the correct triplet and $g(T_c^{(i)})$ represents the confidence for the corrupted triplet. Intuitively, this objective function maximizes the margin between the confidence in the true triple and the false triple.

2.4 Training Function

Socher's model is trained by taking derivatives with respect to the NTN's parameters, like in backpropagation for traditional neural networks. Equation 3 represents the derivative for the jth slice of the tensor layer and Equation 4 represents the jth element of the hidden tensor layer (6).

$$\frac{\partial g(e_1, R, e_2)}{\partial W^{|j|}} = u_j f'(z_j) e_1 e_2^T \tag{3}$$

$$z_j = e_1^T W^{[j]} e_2 + V_j \begin{pmatrix} e_1 \\ e_2 \end{pmatrix} + b_j \tag{4}$$

Socher's model uses the L-BFGS algorithm for parameter estimation and optimization. When the model is trained, a confidence threshold T_R is set for each relation type. If $g(e_1, R, e_2)$ exceeds T_R, then e_1 and e_2 are predicted as being in the relation R. Otherwise, the e_1 and e_2 are predicted as not being in the relation R. The accuracy of the model is then determined by calculating the percentage of triples that are classified correctly (6).

3 Datasets

Socher evaluated his neural tensor network model on two knowledge bases: Wordnet and Freebase. Wordnet is a knowledge base for the English language which groups words into sets of synonyms and contains relations between these sets. Wordnet contains 38,696 entities and 11 relations (5). Freebase is a collaborative knowledge base which connects entities as a graph. Freebase contains 75,043 entities and 13 relations (2). We used pretrained word vectors from word2vec rather than using random initialization and we evaluated our implementation on the Wordnet dataset (see Section 6).

4 Implementation

We reimplemented Socher's Neural Tensor Network using TensorFlow, a numerical computing library similar to Theano and Torch, but slightly different: Although it is optimized for neural networks and has plenty of additional helper functions for computations in that domain, it was not only built for Machine Learning. Secondly, it has visual debugging tools (called TensorBoard) built in to visualize learning and diagnose potential bottlenecks. (1).

4.1 Data Flow

TensorFlow programs can be divided into three stages: initialization, graph building, and evaluation. In the initialization stage, we need to provide placeholders to define the shapes of the inputs to our model. (1)

The graph building stage is further divided into three steps that form a training loop: inference, loss, and training. The inference step first builds the graph as far as needed to return a tensor containing output predictions. Next, the loss step adds the loss operations to the graph. The training step adds the operations needed to minimize loss to the graph. (1)

Finally, an evaluation graph is generated in order to evaluate the accuracy of the model's predictions. Figure 3 depicts the data flow through a typical TensorFlow program. (1)

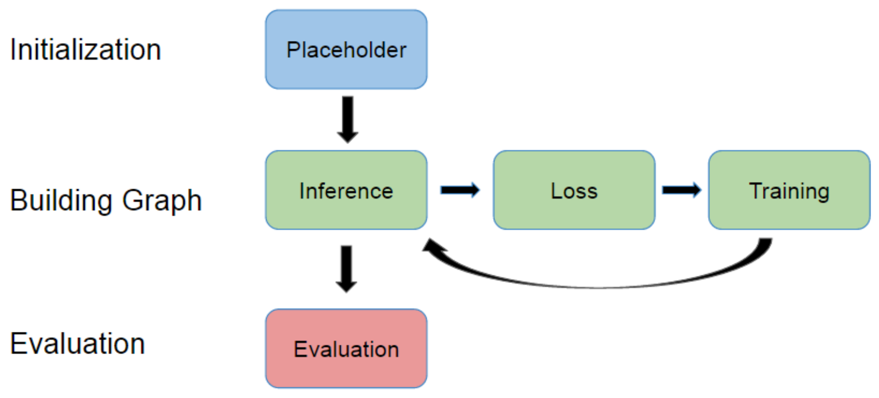

Figure 3: This figure shows the four stages of a TensorFlow program: Inference, loss, training, and evaluation

4.2 Hyperparameters

To better compare our model to Socher's model, we used the same hyperparameters as Socher. We set $k = 2$ (number of slices), $C = 10$ (corrupted triples per correct triple), the dimensionality of the hidden layer to 100, and the number of training iterations to 500 (6). We also used the AdaGrad optimizer in the training function because TensorFlow does not yet provide support for the L-BFGS optimizer.

4.3 Implementation Details

Following the paradigm of TensorFlow laid out above, we implemented Socher's Neural Tensor Network across five files:

- In *params.py*, we defined both the model hyperparameters (listed above) as well as input/output file parameters.
- In *ntn_input.py*, we defined a set of functions to load and preprocess the entity and relation data; training, test, and development sets; and preinitialized word embeddings.
- In *ntn.py*, the bulk of the model was defined. We used four functions corresponding to the three data stages shown in figure 3, each defining the relevant portion of the graph. Inference took in true/corrupted example triples and outputted a list of corresponding probabilities; loss calculated the loss function as defined in Socher 2013; training ran the AdaGrad optimizer on the batch to minimize loss. Evaluation, running on a separate pipeline, took in the results of inference and compared them to known data labels (true or false edges).
- Finally, *ntn_train.py* and *ntn_eval.py* defined the data pipelines and built their respective graphs to train and evaluate the model respectively.

Below, we shall discuss some of the challenges and limitations inherit in TensorFlow and how they affected our implementation of Socher's algorithm.

4.4 Challenges

TensorFlow, being a relatively new machine learning framework, is predictably lacking in accurate and comprehensive documentation. In addition, the mental shift to TensorFlow's "build-evaluate" model, as opposed to a more familiar functional model, required significant work.

These general difficulties aside, there were a few specific problems faced within the framework which affected our implementation. As discussed above, TensorFlow currently has no implementation of the L-BFGS optimizer. We instead used the AdaGrad optimizer with a learning rate of 0.01; however, this was reported as being suboptimal compared to L-BFGS.

More troublesome, however, was an undocumented limitation in one of the key TensorFlow operations utilized in our implementation. tf.DynamicPartition was used in our initial implementation to split the given examples in inference based on their relation type. However, this function is currently incompatible with TensorFlow's automatic differentiation. This led us to preprocess and split each batch before performing the inference. It is unclear how this affected the accuracy found in our implementation.

5 Baselines

Socher's neural tensor network model achieved a 86.2 percent accuracy using the Wordnet dataset and a 90.0 percent accuracy on the Freebase dataset. Socher's paper also compared their NTN model's results to many other models for knowledge base completion, such as the Distance Model, Single Layer Model, and Bilinear Model (6). Results are reported below.

Model	**Wordnet**	**Freebase**
Random Guessing	50.0	50.0
Distance Model	68.3	61.0
Single Layer Model	76.0	85.3
Bilinear Model	84.1	87.7
Socher's NTN Model	86.2	90.0

6 Results

The following table shows our model's precision after completing n iterations on the Wordnet dataset, where n varies from 0 to 500.

Iterations	**Wordnet Accuracy**
0	50.260
70	68.911
100	69.513
500	68.152

After 500 iterations, our model achieved a final accuracy of 68.15 percent on the Wordnet dataset, with an optimal test accuracy of 69.513 after iteration 100. Our model significantly outperforms a baseline of random guessing and achieves a similar accuracy as the distance model. However, our model underperforms compared to other neural network models, including Socher's neural tensor network model.

Our implementation had several deviations from Socher's model, which may explain some of the inconsistencies in our results. For example, our implementation did not calculate thresholds for each relation and instead used a threshold value of 0.0 for all relations when determining final predictions.

The NTN model is trained to maximize the true scores and minimize corrupted scores for each relation. However, this does not specify any information about the cutoff, or threshold, that is generated to optimally seperate true and false examples. Therefore, a more successful implementation might use a development set to pick these cutoffs deterministically and for each relation, to further improve the algorithm's ability to label any given example.

In addition, as previously mentioned, our implementation used the AdaGrad optimizer instead of the L-BFGS optimizer. Socher also experimented with the AdaGrad optimizer, but "found that it performed slightly worse" (6).

Finally, our model achieved a higher accuracy for 100 iterations than for 500 iterations, which suggests that TensorFlow may be overfitting our model's paramters. Therefore, additional tuning of our model's hyperparameters may be necessary to achieve optimal accuracy results.

7 Conclusions and Future Work

This project was in part an exploration and evaluation of TensorFlow and in part an evaluation of Neural Tensor Networks for Knowledge Base Completion. As a result, we have conclusions about each of these components.

TensorFlow is surprisingly immature as an open-source project. Even though it was released by Google, embarrassingly simple errors abound in the documentation, and painfully missing functions which are cornerstones of comparable frameworks are unadressed. TensorFlow seems to have been developed to effortlessly support classic CNN architectures and programs, but our relatively arcane Neural Tensor Network model had to jump through a great many hoops to exist inside the TensorFlow paradigm.

Robust Multi-GPU support is still missing in TensorFlow (though it's being worked on at Google right now). It may be the case that Google prematurely released the library or simply that they did not design it for our particular use case. Worth noting is that none of us have much experience with numpy/scipy, Torch, or Theano, which means we may have overlooked some much easier way of accomplishing our goals which was not obvious to us. When we first set out to built a Neural Tensor Network in TensorFlow, it wasn't clear to us how to define the Bilinear Tensor Product. It turns out the canonical way of defining most operations is by reshaping the data such that the original operation becomes a matrix multiply, or several, interspersed with reshapes. This provides evidence for our naiveté which might extend into other parts of our implementation.

This project also provided us an opportunity to peer into the domain of Knowledge Base Completion and new approaches to that problem. Socher and Chen's Neural Tensor Network approach seems quite insightful and performs quite well. We weren't able to reach feature parity with them (they do a fair bit of pre-processing and post-processing, as well as using a different optimizer) which substantially harmed our performance.

Since their work, a number of other approaches that built off of Socher and Chen's work have improved slightly on their performances, but no breakthrough has been accomplished since. We wonder what might result from adjoining the tensors (each indexed by R) into one larger tensor in order to share statistical strength across relation types. We also wonder whether there might be a better way to model relations, which can often mean different things in different contexts, rather than rigid on/off settings. Which brought us to the idea that knowledge bases might be a sub-optimal way of modeling knowledge. Many systems deployed across a variety of domains use knowledge bases quite successfully, but knowledge bases represent a set of factual truths in a very strict manner, which inevitably leads to their incompleteness – many truths are subtle and complex, perhaps even most of them.

We also wonder whether there might be a way to constructively use RNNs to build entity representations. Socher et al. abandon the idea because of the fact that entities with names like Giuseppe Bandolino don't have much information in their ordering, meaning that performing anything more than averaging the word vectors is overkill. But many entities could benefit from an RNN-based entity construction, such as "Bank of America" which is not simply the average of America and Bank. It stands to reason there may be a way to productively use RNNs for entity representation.

We look forward to seeing the maturing of TensorFlow, which is undoubtedly promising, and advances in the field of Knowledge Base Completion, which we think has much potential for growth.

References

[1] Abadi, M., Agarwal, A., Barham, P., et al. (2015). TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems.

[2] Freebase. (n.d.). Freebase Documentation. Retrieved December 10, 2015, from http://wiki.freebase.com/wiki/Main_Page

[3] Kaggle. (2015). The Allen AI Science Challenge. Retrieved December 10, 2015, from https://www.kaggle.com/c/the-allen-ai-science-challenge/

[4] Guu, K., Miller, J., & Liang, P. (2015). Traversing Knowledge Graphs in Vector Space. EMNLP.

[5] Princeton University. (2010). About Wordnet. Retrieved December 10, 2015, from http://wordnet.princeton.edu

[6] Socher, R., Chen, D., Manning, C., & Ng, A. (2013). Reasoning With Neural Tensor Networks for Knowledge Base Completion. Advances in Neural Information Processing Systems (NIPS).