REASONING OVER KNOWLEDGE BASES

Thesis submitted by

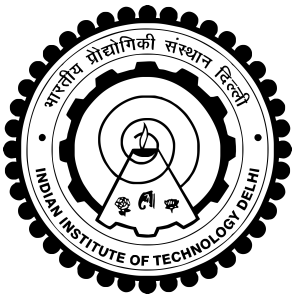
Kaladi Lalith Satya Srinivas
2017CS10340
Nagendra Kokku
2017CS10343

under the guidance of

Prof. Maya Ramanath

in partial fulfilment of the requirements
for the award of the degree of

Bachelor of Technology



Department Of Computer Science and Engineering
INDIAN INSTITUTE OF TECHNOLOGY DELHI

January 2021

ABSTRACT

KEYWORDS: Reasoning over Knowledge Base; Sub-symbolic Reasoning; Neural Theorem Provers; Greedy Neural Theorem Provers; Conditional Theorem Provers.

We study Neural Theorem Provers (NTPs), which are neural networks used to end-to-end differentiable proving of queries to the knowledge base by operating on dense vector representations of symbols. These neural networks are constructed recursively by taking inspiration from the backward chaining algorithm as used in Prolog. Specifically, we replace symbolic unification with a differentiable computation on vector representations of symbols using a radial basis function kernel, thereby combining symbolic reasoning with learning subsymbolic vector representations. There are currently three iterations of Neural Theorem Provers. We want to study these and identify the bottlenecks and limitations and how well they perform on standard datasets larger than those used in the papers.

Contents

Chapter 1

THEORY

1.1 Introduction

The prominent methods used for completing Knowledge Base (KB) are neural link prediction models that learn the distributed vector representations of symbols, allowing the model to encode similarities. For example, If the vector of the predicate symbol grandfatherOf is similar to the vector of the symbol grandpaOf, then the model can conclude they express a similar relation. While this practice is great at encoding similarities, it often fails to capture more complex reasoning patterns that involve several inference steps. For example, if ABE is the father of HOMER and HOMER is a parent of BART, we would like to infer that ABE is the grandfather of BART. Such transitive reasoning is inherently hard for neural link prediction models as they only learn to score facts locally. In contrast, Symbolic theorem provers like Prolog are capable of this type of multi-hop reasoning but cannot learn representations of symbols and identify similarities between them.

Neural Theorem Provers (NTPs) are introduced as end-to-end differentiable provers for basic theorems formulated as queries. They use a backward chaining algorithm as a base to recursively build the neural network and learn the distributed vector representations of the symbols. The success score of the proof paths becomes differentiable with respect to vector representations of symbols and enables us to learn representations of entities, predicates, and other parameters involved in rule inducing. The later iterations of the model tried to make the model more efficient by limiting the choices of rules that need to be considered at any point in the proof path.

1.2 Neural Theorem Prover

In the following, we describe the recursive construction of NTPs – neural networks for end-to-end differentiable, proving that allow us to calculate the gradient of proof successes with respect to vector representations of symbols. NTPs can be divided into three main modules that are called upon recursively to prove a query. Each module takes an input of atoms and proof state and outputs a list of new proof states. A proof state S = (ψ, ρ) is a tuple, where ψ is the substitution set for the variable at that point of the proof, while ρ is the success score of the current proof.

1.2.1 Unification module

Unification of two atoms, i.e., a goal and a rule head, is the basis of backward chaining. In typical backward chaining used in Prolog, we try to match the symbols in the atoms, and if they don't, unification fails, and the proof can be aborted. In NTP, however, we look for similarity between the atoms and update the substitution set along with the success score, and output a new proof state.

The signature of the unification module is L x L x S $\rightarrow$ S, where L is the list of terms or atoms (triples, usually) and S is a proof state. Unify iterates through both lists of terms, compares them, and updates them. If one of the symbols is variable, it is added to the substitution set. Otherwise, the vector representations of the symbols are used to calculate the similarity using Radial Base Function (RBF) kernel with a hyperparameter μ.

1. unify(H, G, S):
2. match H, G with:
3. $[], [] \rightarrow$ S
4. $\_, [] \rightarrow$ FAIL
5. $[], \_ \rightarrow$ FAIL
6. $h : H^1, g : G^1 \rightarrow unify(H^1, G^1, S^1)$ where $S^1 = (S^1_\psi, S^1_\rho)$ and $S^1_{psi} =$
   (a) $S_\psi \cup \{h/g\}$ if $h \in V$
   (b) $S_\psi \cup \{g/h\}$ if $g \in V$ and $h \notin V$
   (c) S_ψ otherwise
7. and $S^1_\rho =$
   (a) min(S_ρ, $\exp(-\|\theta_h - \theta_g\|/2\mu^2)$) if h, g $\notin$ V
   (b) min(S_ρ, 1) otherwise

1.2.2 OR module

This model tries to apply rules to our goals and outputs the possible proof states. The signature of this module is L x $\mathbb{N}$ x S $\rightarrow S^N$, where L is the set of atoms, $\mathbb{N}$ is the set of natural numbers which specify maximum depth of our proof and S is the set of proof states. We implement OR as:

1. $or^{\mathbb{K}}_\theta(G, d, S) = [S^1 | S^1 \in and^{\mathbb{K}}_\theta(\mathbb{B}, d, unify_\theta(H, G, S)) for H :- \mathbb{B} \in \mathbb{K}]$

where H :- $\mathbb{B}$ denotes every rule in KB $\mathbb{K}$, where H is the head atom of the rule and $\mathbb{B}$ denotes the list of atoms in the body. The or module recursively calls and module with the proof state obtained from the unification module.

1.2.3 AND module

The and module first uses a function substitute that substitutes all the variables in atoms according to the substitution set available. The substitue function is as follows:

1. $substitute([], \_) = []$
2. $substitute(g : G, \psi) = \{\ x \text{ if } g/x \in \psi, g \text{ otherwise}\} : substitute(G, \psi)$

The signature of and module is L x $\mathbb{N}$ x S $\rightarrow S^N$, where L is the domain of list of atoms, $\mathbb{N}$ is set of natural numbers which dictates the maximum depth of the proof and S is the set of proof states. The and module is as follows:

1. $and_\theta^{\mathbb{K}}(\_, \_, FAIL) = FAIL$
2. $and_\theta^{\mathbb{K}}(\_, 0, \_) = FAIL$
3. $and_\theta^{\mathbb{K}}([], \_, S) = S$
4. $and_\theta^{\mathbb{K}}(G : \mathbb{G}, d, S) = [S^2 | S^2 \in and_\theta^{\mathbb{K}}(\mathbb{G}, d, S^1) \text{ for } S^1 \in or_\theta^{\mathbb{K}}(substitute(G, S_\psi), d - 1, S)]$

1.2.4 Bottleneck

We can observe that in the or module, we iterate through every rule and fact present in our KB $\mathbb{K}$ at every stage of our proof. This is a highly inefficient way of expanding our proofs since we only use the path which gives the maximum score for updating our model parameters. This is especially infeasible on large knowledge bases, not only in terms of time but also memory, as we have to store all these proof states in memory and pick the maximum score. A serious optimization on reducing the scope of rules to be expanded upon is required.

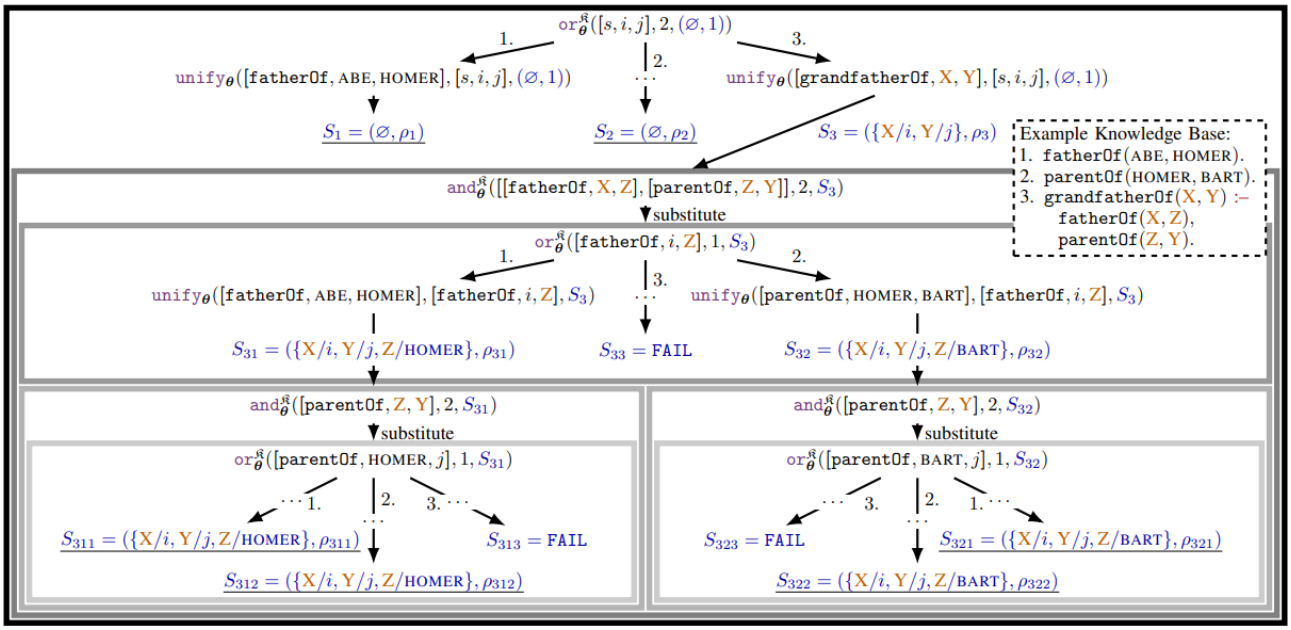


Figure 1.1: Exemplary construction of an NTP computation graph for a toy knowledge base. Indices on arrows correspond to application of the respective KB rule. Proof states (blue) are subscripted with the sequence of indices of the rules that were applied. Underlined proof states are aggregated to obtain the final proof success. Boxes visualize instantiations of modules (omitted for unify). The proofs S33, S313 and S323 fail due to cycle-detection (the same rule cannot be applied twice).

1.3 Greedy Neural Theorem Prover

Greedy NTPs (GNTPs) tries to address the complexity and scalability of NTPs. As mentioned in the previous section, the bottleneck of NTP is that it iterates through the rules and facts present in the Knowledge Base (KB). We also mentioned that all the proof states that are not the maximum would be discarded at the end; thus, selecting the rules that produce the best scores is important by some heuristic. A couple of heuristics are introduced for selecting facts (at the terminal stage, depth=1) and for selecting rules based on the goal.

1.3.1 Fact selection

Unifying subgoal with all facts in our KB is not feasible for larger KBs. Considering the expression,

$$ntp_\theta^{\mathbb{K}}(G, 1) = max_{F:-[]\in\mathbb{K}} S_\rho^F$$

where S_ρ^F is the unification score of G with the fact F. NTP implementation currently

iterates through all the facts and take the maximum score as the output. The order of complexity of this approach is $O(|\mathbb{K}|n)$, where n is the number of subgoals. This means that, at inference time, we only need the largest proof score for returning the correct output. Similarly, during training, the gradient of the proof score with respect to the parameters θ can also be calculated exactly by using the single largest proof score:

$$\frac{\partial ntp_\theta^{\mathbb{K}}(G,1)_\rho}{\partial \theta} = \frac{\partial max_{F:-[]\in\mathbb{K}} S_\rho^F}{\partial \theta} = \frac{\partial S_\rho^*}{\partial \theta}$$

GNTP uses the Nearest Neighbour Search (NNS) to efficiently compute the maximum score S^*. It uses the exact and approximate NNS framework for efficiently searching $\mathbb{K}$ for the best supporting facts for a given sub-goal. Specifically, it uses the exact L2-nearest neighbor search, and, for the sake of efficiency, we update the search index every ten batches, assuming that the small updates made by stochastic gradient descent do not necessarily invalidate previous search indexes.

1.3.2 Rule selection

A similar idea is used to select rules for proving a given goal G. We expand the proofs with rules whose rule heads are closest to the goals. We can partition the rules and facts that are present in our KB $\mathbb{K}$ that has the same template. We can select the best k rules that are of certain template which may produce the best scores. The or module is adapted as follows:

$$or_\theta^{\mathbb{K}}(G,d,S) = [S^1 | H :- \mathbb{B} \in N_P(G), P \in \beta, S^1 \in and_\theta^{\mathbb{K}}(\mathbb{B},d,unify_\theta(H,G,S))]$$

where P is the template of the rules to pick from, and H :- $\mathbb{B}$ is the best rules in the neighborhood of goal G under the template P (N_P). So instead of unifying the goal with all rule heads, it constrains the unification to only the rule heads in the neighborhood of G.

1.3.3 Bottleneck

The bottleneck in the case of GNTP is updating the search indexes used for the NNS framework every few batches. This can even be observed in the substantial growth of run time for the batch in which an update of the search index is done.

1.4 Conditional Theorem Prover

Similar to GNTP, CTP tries to efficiently select rules upon which to expand the proof. CTP does it by introducing a new module select, which is essentially a neural network that takes the vector representation of the goal head as an input and outputs the rule with a defined template. The parameters of the select module are trainable, and the module will learn rules that agree with the data present in our Knowledge Base (KB).

So the or module will be modified to accommodate the select module instead of iterating through all rules present in our KB. The or module is as follows:

$$or(G, d, S) = [S^1 | \text{ for } H :- \mathbb{B} \in select_\theta(G) and S^1 \in and(\mathbb{B}, d, unify(H, G, S))]$$

Several variations of the select module are also introduced, which include linear, attentive, and memory-based goal reformulators.

1.4.1 Goal Reformulators

Linear Reformulator

Here, select module is defined as a linear function of the goal predicate. For example, for a template of 1 head, 2 body terms rule, it will be as follows:

$$select_\theta(G) = [F_H(G) :- F_{B_1}(G), F_{B_2}(G)]$$

where

1. $F_H(G) = [f_H(\theta_G), X, Y]$
2. $F_{B_1} = [f_{B_1}(\theta_G), X, Z]$
3. $F_{B_2} = [f_{B_2}(\theta_G, Z, Y]$

where each $f_i : \mathbb{R}^k \to \mathbb{R}^k$ is a linear function of the form $f_i(x) = W_i x + b_i$, where $W_i \in \mathbb{R}^{k*k}$ and $b_i \in \mathbb{R}^k$. Thus similar to GNTP, instead of iterating through all the rules in KB, we can apply this linear transformation on our goal predicate to get the rule we need to expand upon.

Attentive Reformulator

This reformulator includes a useful prior to the select module, namely the predicate symbols that are present in our KB. The method used to incorporate this prior is to generate

distributions of goal G over all the predicates present in our KB. Let the set of predicates/relations be R. The select module is as follows:

$$f_i(x) = \alpha E_R$$
$$\alpha = softmax(W_i x) \in \Delta^{(|R|-1)}$$

where $E_R \in \mathbb{R}^{|R|*k}$ is the predicate embedding matrix of R, $W_i \in \mathbb{R}^{k*|R|}$ and $\alpha \in \Delta^{|R|-1}$ is the attention distribution over the predicates R. This method is especially useful if the number of predicates is less than embedding size and also provides meaningful predicates in rules faster.

Memory-Based Reformulator

Memory-based reformulators help us to inspect what kind of rules are being formed by the select module. We store the n rules in our KB in memory matrices $[M_1, M_2, ...M_m]$, where $M_i \in \mathbb{R}^{n*k}$ contains the i-th predicate of all the n rules. Now similar to attentive reformulator, we compute an attentive distribution over these n rules using our goal predicate embedding. The rule is obtained by taking the weighted average of the rules in the memory matrices. We can also inspect the rules better by observing the memory matrices and attention distribution for the predicates.

$$f_i(x) = \alpha M_i$$
$$\alpha = softmax(Wx) \in \Delta^{n-1}$$

where $f_i : \mathbb{R}^k \to \mathbb{R}^k$ is a differentiable function that gives the goal produces an attention distribution $\alpha \in \Delta^{n-1}$ over the rules present in our memory.

1.4.2 Bottleneck

The bottleneck with respect to time and memory in this model comes from the step where unification scores are calculated with every fact present in our Knowledge Base (KB) at the terminal node step. This effect is particularly visible in larger datasets. A similar approach of GNTP can be used for selecting facts while still adopting the select structure to generate rules can be used to overcome this bottleneck, though we still need to pursue this idea.

1.5 TransE

Multi-relational data refers to directed graphs whose nodes correspond to entities and edges of the form (head, label, tail) (denoted (h, l, t)), each of which indicates that there exists a relationship of name label between the entities head and tail. Models of multi-relational data play a pivotal role in many areas.

TransE is an energy-based model for learning low-dimensional embeddings of entities. In TransE, relationships are represented as translations in the embedding space: if (h, l, t) holds, then the embedding of the tail entity t should be close to the embedding of the head entity h plus some vector that depends on the relationship l. This approach relies on a reduced set of parameters as it learns only one low-dimensional vector for each entity and each relationship.

Training: Given a training set S of triplets (h, l, t) composed of two entities $h, t \in E$ (the set of entities) and a relationship $l \in L$ (the set of relationships), our model learns vector embeddings of the entities and the relationships. The embeddings take values in R^k (k is a model hyperparameter) and are denoted with the same letters, in boldface characters. The basic idea behind our model is that the functional relation induced by the l-labeled edges corresponds to a translation of the embeddings, i.e. we want that $h + l \approx t$ when (h, l, t) holds (t should be a nearest neighbor of $h + l$), while $h + l$ should be far away from t otherwise. Following an energy-based framework, the energy of a triplet is equal to $d(h+l, t)$ for some dissimilarity measure d, which we take to be either the $L1$ or the $L2$ -norm.

Chapter 2

EXPERIMENTS

2.1 Introduction

2.1.1 Setup

We have conducted all our experiments on Baadal (IITD's cloud computing platform) Virtual Machine with 32GB Memory. We have used Tensorflow version 1.12.0 for executing GNTP code and PyTorch for executing CTP code. We have made changes to both the models so that they use python pandas to read triples from data files. This makes the file reading part faster.

Following are the steps followed to run the models on the machine.

1. Install conda and set up a virtual environment with TensorFlow version 1.12 and python version as 3.6.
2. Run the command `pip install -r requirements.txt` in folders containing these modules, separately. This will install all dependency libraries.
3. After installing all the dependencies, run the command `python setup.py install` in both the folders to install the models.
4. Finally, enter the command to run the models. Example commands for CTP and GNTP are mentioned below.
5. For executing CTP:

```
python3 ./bin/gntp-moe-cli.py
    --train data/wn18rr/train.tsv
    --dev data/wn18rr/dev.tsv
    --test data/wn18rr/test.tsv
    -c data/wn18rr/clauses.v1.pl -E ranking --max-depth 1 -b 1000 --corrupted-pai
    -l 0.005 --l2 0.001 --k-max 10 --all -F 5 -R 1 -I 100 --seed 0 --model-type r
    --initializer uniform --rule-type standard --decode --kernel rbf --unificatio
    -i faiss -e 100 --auxiliary-epochs 0 --test-batch-size 10000
    --only-rules-epochs 95 --only-rules-entities-epochs 0 --input-type standard
    --train-slope --check-path data/wn18rr/dev.256.tsv --check-interval 1000
```

6. For executing GNTP:

```
python3 ./bin/clutrr-cli.py
--train data/clutrr-emnlp/data_db9b8f04/1.2,1.3,1.4_train.csv
--test data/clutrr-emnlp/data_db9b8f04/1.10_test.csv
-S 1 -s concat -V 128 -b 32 -d 2 --test-max-depth 4 --hops 2 2 2 2 -e 5
-o adagrad -l 0.1 --init random --ref-init random
-m 5 -t min -k 20 -i 1.0 -r memory -R 256 --seed 1
```

2.1.2 Datasets

The Authors have included a few standard datasets like WN18RR, WN18, FB15K-237, Nations, Kinship, UMLS, etc. they have also used these datasets to compare GNTP and CTP with other link-prediction models like ComplEx, TransE, etc. The datasets and the number of facts in each dataset are mentioned below.

Name of Dataset	**Number of Facts**
Countries	1,158
Nations	2,565
Kinship	10,686
UMLS	6,529
Freebase (fb15k-273)	270,000

Table 2.1: Table showing Datasets and Number of Facts.

The Authors also used a dataset called CLUTTR, which is significantly larger than the ones mentioned above, to showcase that CTP can handle larger data better compared to GNTP. But when we looked at CLUTTR dataset, we observed that this is a special dataset in which the bottleneck of CTP goes away. CLUTTR contains rows in which there is a story, which is the collection of local facts for that row, and it also has some queries, which are the goal atoms we will work with. So for every goal at a terminal node, we only need to unify it with the facts from the same row, which is a significantly smaller set of facts than the complete facts in KB, like in other cases. Thus we think that CLUTTR is not reliable dataset to make the conclusion that CTP handles bigger data better than GNTP, which we also reflected later in the experiments we conducted, where GNTP outperforms CTP in terms of memory usage and time taken.

It can be observed from the table above that these datasets are far smaller when compared to real-world knowledge bases. Hence, we have collected larger datasets that are within the models limits. One of the main datasets that we have considered is YAGO. The primary YAGO dataset contains around 2 billion facts. As both CTP and GNTP store all the facts

Number of Entities	Number of Facts
Top 10	679,326
Random 10,000	33,793
Random 10,000	22,774
Random 100,000	243,830
Random 100,000	329,850
Random 270,000	612,996
Random 270,000	634,349
Bottom 700,000	595,601

Table 2.2: Table showing Number of Entities and Number of Facts in each Datasets sampled.

in the memory, it will not be feasible to run these models on this dataset given a machine with 32GB memory. So we have considered a liter version of YAGO, which has 5.4 Million facts. We tried running both the model with the same parameters used by the Authors. The memory got exhausted. Hence we tried fine-tuning the hyper-parameters. Though we were able to execute GNTP, after changing hyper-parameters like batch size, which has consumed 29GB of memory, we could not run CTP. Hence, we created datasets by sampling entities from the lighter-YAGO dataset to run experiments to observe Memory utilization and execution time limitations. We have created datasets by sampling top-k frequent, least-k frequent, random-k entities and extracted all the facts from the dataset related to these entities to conduct our experiments.

2.1.3 Versioning, Dependency Issues

Firstly, we tried replicating the results that the authors have claimed. We ran into a few versioning issues (for GNTP) like the version of TensorFlow, dependency libraries, python version, etc. We have tried the versions mentioned in requirements.txt and setup.py, but we were not able to run the module. So, we manually checked for the version that satisfies all conditions and changed requirements.txt and setup.py accordingly.

2.1.4 Sampling Dataset

As mentioned before, we tried using the lighter version of YAGO. We sampled the data for the following reasons.

1. To observe how memory utilization, execution time depends on the Number of Entities and the Number of Facts.
2. Compare GNTP and CTP in terms of Memory utilization and execution time.
3. Maximum size of KB, each model, can handle, given a 32 GB machine.

4. Accuracy improvement per epoch for each model.

Hence we have made a script to sample the data. The script first goes through all the triples and store the frequency of an entry as a subject or object, predicate. After this, we have sampled top-10 frequent entities (an entity is one that was present as a subject or object). Analyzing top frequent entities provides us a situation with less number of entities and more number of facts. Similarly, analyzing the least frequent entities helps us compare them with a similar number of facts and a far more number of entities. We have randomly sampled 10K, 100K, 270K entities and extracted all the facts related to these entities. We have used these datasets as our base datasets for analyzing and observing limitations in terms of memory utilization and execution time. We have also used standard datasets by FreeBase and WordNet for analyzing loss, accuracies of the models, which are still work in progress at the time.

2.2 Replicating the Results

After resolving all versioning and dependency issues, we tried executing the modules on the standard datasets to replicate the author's results. By default, the GNTP model evaluates/validates the model for every 1000 batches. The estimated evaluation time was around 26 hours. There were 8684 batches, so it took 10-11 days for the GNTP model to execute completely. The results after complete execution were in accordance with the results mentioned in the paper with a slight margin of error.

Similarly, we have executed CTP on a CLUTTR dataset. The size of the dataset file is larger than that of GNTP's, but the execution time is many folds less. We have gone through the CLUTTR dataset and CTP code to reason why CTP is much faster and accurate though the dataset is larger than that of GNTP.

Metrics: Mean Reciprocal Rank (MRR), Hits@m for m = 1, 3, 5, 10.

2.2.1 Analysis

The CTP essentially have to go through all the facts at the terminal node to compute scores and select the facts with the best scores. As mentioned before, each CLUTTR data entry has a story associated with it. Because of this property, we don't have to go through all the facts, but just the ones associated with the story. Hence, it makes the bottleneck of CTP disappear as the number of facts to iterate at the terminal node decreases drastically. This is not true in the case of general datasets involving triples. The authors have used

CLUTTR dataset to show that CTP can handle larger datasets when compared to GNTP. But this claim is only true under certain conditions, like the dataset should be of CLUTTR format. Hence we have started to analyze the performance of CTP and GNTP on more general datasets.

2.3 Memory Utilization Analysis

As the author claims that CTP can handle larger datasets better than GNTP, we tried comparing memory utilized by GNTP and CTP. We also wanted to observe how memory utilized by these models is related to a number of Entities and a number of Facts.

Datasets: We have used the datasets mentioned in the Table 2.2 for this experiment.

Hyper-parametres: The following are the hyper-parameters used while running both models.

Hyper-parameter Name	**Value**
Embedding Size	20
Learning Rate	0.1
Batch Size	32
Reformulator (for CTP)	Linear

Table 2.3: Table Showing hyper-parameters and their values

Method: We have executed the command to run each module (separately) on a dataset and observed how the net memory varies for every batch. We have repeated this process on all the datasets we have mentioned.

Metrics: On executing the command `free -mh`, we can view the free and used memory in the machine. The 'used' memory is the metric we have chosen to estimate the memory used by each model.

2.3.1 Results

Entities	**Facts**	**Memory (by GNTP)**	**Memory (by CTP)**
Top 10	679326	1.9 GB	4.9 GB
Random 10,000	33,793	542 MB	786 MB
Random 10,000	22,774	498 MB	699 MB
Random 100,000	243,830	1.6 GB	2.2 GB
Random 100,000	329,850	2.0 GB	2.8 GB
Random 270,000	612,996	3.3 GB	4.8 GB
Random 270,000	634,349	3 GB	4.7 GB
Bottom 700,000	595,601	4 GB	4.8 GB

Table 2.4: Table Showing memory utilized by each model when run on the corresponding dataset.

2.3.2 Analysis

It can be observed from Figure 2.1 that, as both numbers of facts and number of entities increases, till 700K entities, the memory utilized by both the models increases. But after that, as mentioned before, the number of entities decreases, and the number of facts increases. It can be observed that the memory utilized by GNTP decreases, whereas memory utilized by CTP increased slightly.

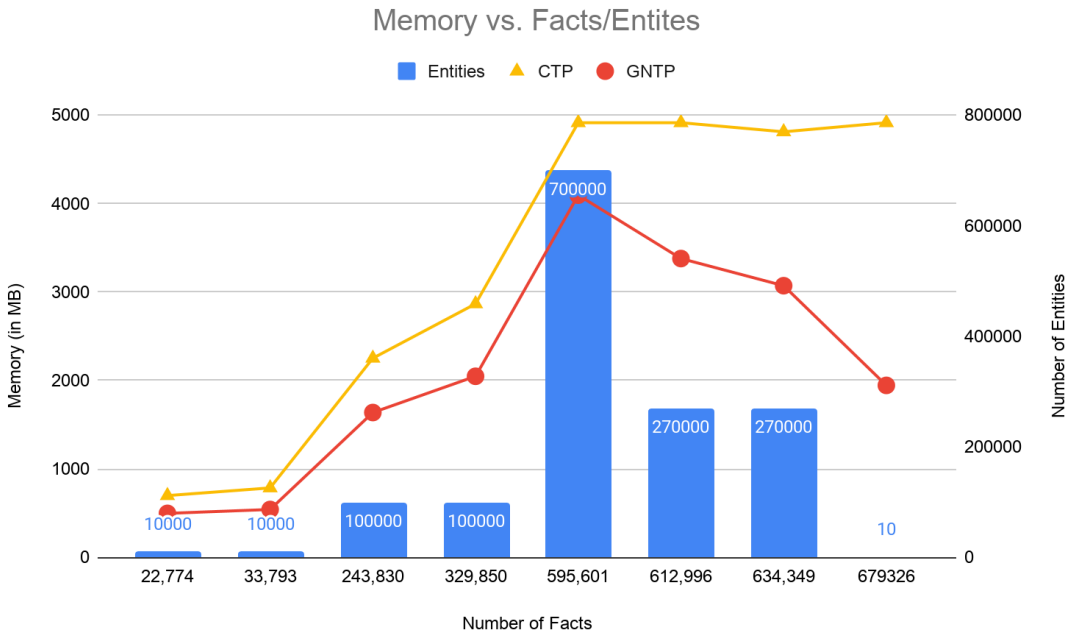


Figure 2.1: Plot showing Number of Facts on X-axis, Memory (in MBs) on left vertical axis and number of Entities on right vertical axis

The memory utilized by CTP is always greater than that of GNTP. But, considering the almost similar number of facts, the difference between GNTP and CTP increases as the number of Entities decreases.

We can say that the memory utilized by GNTP is more sensitive to a number of entities that to a number of facts. In the case of CTP, memory utilized by this model is more sensitive to a number of facts than to a number of entities.

2.4 Execution Time Analysis

As mentioned before, GNTP took a lot of time to execute a small dataset, but CTP took far less time when run on the CLUTTR dataset. Our goal is to analyze how these models perform on larger datasets, with triples, in terms of time. This gives us a better idea of which model performs better while handling larger datasets in terms of execution time.

Datasets: We have used the datasets mentioned in the Table 2.2 for this experiment.

Hyper-parametres: The hyper-parameters used for this experiment are the same as the parameters mentioned in Table 2.3.

Method: To observe the time it takes for each model to train a batch, we have made few changes to the source code of both models. After re-installing the models, we executed them independently, one at a time, and observed the average time it takes to train one batch for both the models. We have extrapolated this average batch times to obtain epoch time too.

Metrics: The average time it takes for a model to train a batch, time it takes for a model to complete one epoch.

2.4.1 Results

Correlation	**CTP**	GNTP
Entities Vs Memory	0.5891616523	0.8617175819
Facts Vs Memory	0.9959181818	0.8900775666
Entities Vs Time	0.07485990126	0.160307427
Facts Vs Time	0.9923655232	0.9848214856

Table 2.5: Table Showing correlation between Number of Entities, Facts and Memory, Time

Entities	**Facts**	Time per batch (by GNTP)	Time per batch (by CTP)
Top 10	679326	3-5 sec	480 sec
Random 10,000	33,793	0.2 sec	13 sec
Random 10,000	22,774	0.15 sec	8.3 sec
Random 100,000	243,830	1.5 sec	130 sec
Random 100,000	329,850	1.7 sec	160 sec
Random 270,000	612,996	3.3 sec	420 sec
Random 270,000	634,349	3.4 sec	450 sec
Bottom 700,000	595,601	3-3.5 sec	420 sec

Table 2.6: Table Showing average time (in seconds) it takes to train a batch by each model when run on the corresponding dataset.

2.4.2 Analysis

The relation between Time per batch/epoch and number of Facts/Entities is not so clear from the plots 2.2, 2.3. Hence we have observed the correlation between time and number of facts, entities. From the table 2.5 it can be observed that there is almost zero correlation

between time and number of entities. Whereas the correlation between time and number of facts is almost 1.

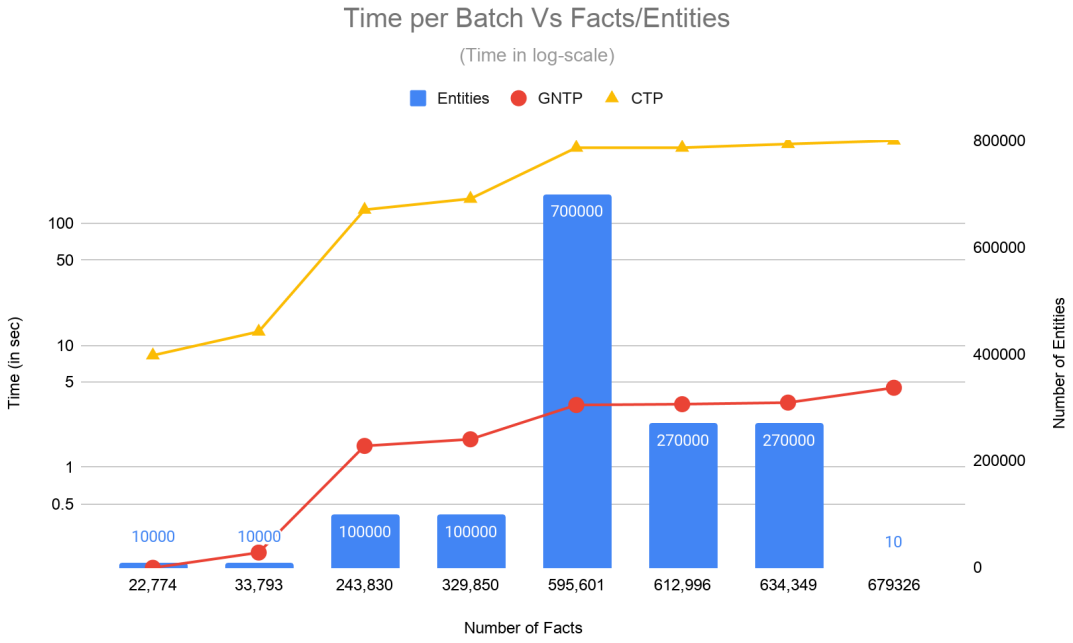


Figure 2.2: Plot with Number of Facts on X-axis, Time per Batch (in sec, log-scale) on left vertical axis and number of Entities on right vertical axis

Hence, we can say that the execution time is almost independent of number of entities whereas fully dependent and proportional to a number of facts. It can also be observed from the table 2.6 and plot 2.2 that execution time of GNTP is less than CTP in many folds.

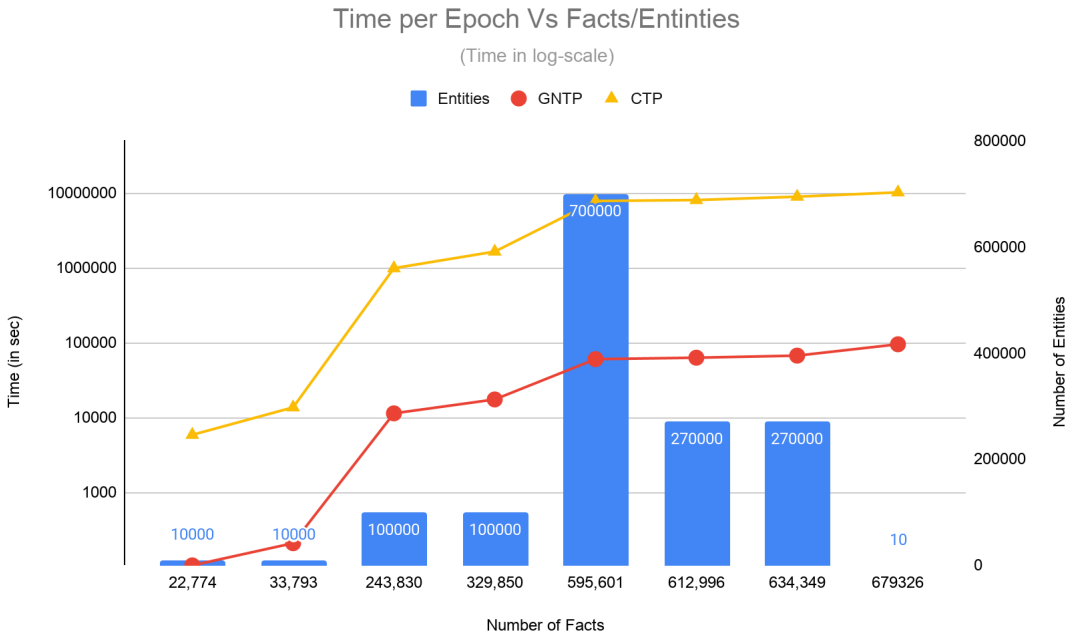


Figure 2.3: Plot with Number of Facts on X-axis, Time per Epoch (in sec, log-scale) on left vertical axis and number of Entities on right vertical axis

2.5 Performance Analysis

GNTP came out to be superior in terms of execution time and memory utilization over CTP. To analyze performance efficiency, we have to compare the final scores and accuracies of these models. But, it takes so much time to run entirely, even on smaller datasets, so we cannot compare the accuracy as of yet. Hence we have compared the Loss per epoch of both models.

Datasets: To analyze performance, the datasets we have created (Table 2.2) cannot be used as these datasets are not complete and standard. Hence, we have used a standard dataset of FreeBase.

Hyper-parametres: The hyper-parameters used for this experiment are the same as the parameters mentioned in Table 2.3, except the batch size, which we've changed to 1000.

Method: CTP and GNTP both use Binary Cross Entropy as their scoring function, but GNTP sums up individual scores, but CTP uses the mean of all individual scores. Hence, we have changed the source code of GNTP to make 'mean' the default option. Then, we

have parsed the log files of both models to tabulate loss per epoch. In the case of GNTP, by default, only loss per batch is displayed. Hence we used the mean and standard deviation of these losses for every epoch (i.e., total triples/ triples per batch = number of batches per epoch) as Loss and error-in-loss of GNTP.

Metrics: Loss, error-in-loss after every epoch.

2.5.1 Results

Following are the results that we have observed.

Epoch	Loss (CTP)	Error (CTP)	Loss (GNTP)	Error (GNTP)
1	1.54	0.04	40.81	5.14
2	1.4	0.04	25.98	3.47
3	1.28	0.03	16.22	2.23
4	1.18	0.03	10.01	1.4
5	1.09	0.02	6.18	0.85
6	1.01	0.02	3.91	0.49
7	0.94	0.02	2.58	0.29
8	0.89	0.02	1.95	0.1
9	0.84	0.01	1.62	0.07
10	0.79	0.01	1.37	0.08
11	0.76	0.01	1.2	0.19
12	0.73	0.01	1.33	0.13
13	0.7	0.01	0.98	0.09
14	0.67	0.01	0.75	0.04
15	0.65	0.01	1.22	0.49

Table 2.7: Table Showing Loss, Error-in-loss per epoch of CTP and GNTP.

2.5.2 Analysis

We can observe from the plot 2.4 loss after the first epoch for CTP is far less than GNTP. Though the loss of GNTP reduces rapidly compared to CTP, CTP will produce better results after very few epochs while GNTP takes more epochs to achieve similar results. CTP can reach a sub-optimal position after a very few epochs compared to GNTP. This observation can be used to make a model that is space, time-efficient as GNTP, and performance efficient as CTP. A thorough analysis is pending in this particular aspect.

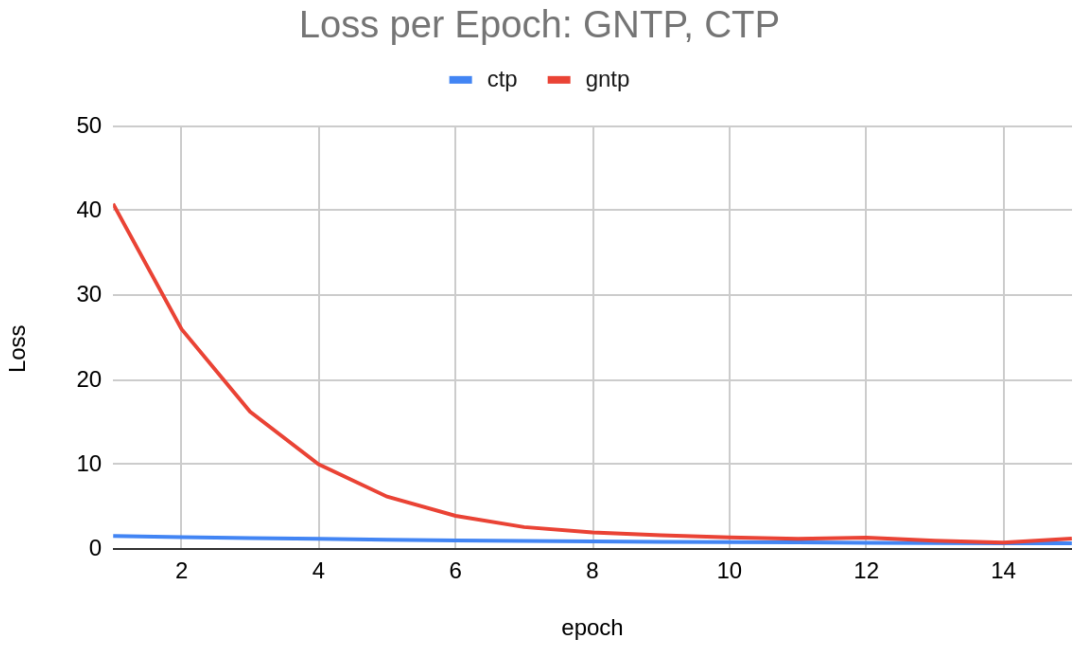


Figure 2.4: Plot with Number of Epochs on X-axis, loss per epoch of both the models on Y-axis.

Chapter 3

Evaluation: Experimenting with FB15k-237 (freebase) dataset

3.1 Evaluation metrics

MRR: Now that we figured out the limitations of the theorem prover models, we deduced that datasets like Freebase and WordNet can be trained within our computational resources. Our next goal is to get the actual metrics of the model on these datasets and compare it to other standard link prediction models. With the same set-up as the above experiments, we trained our model on Freebase and then ran the evaluation script the authors provided on the trained parameters. The metrics used for these models is Mean Reciprocal Rank (MRR).

Analysis on evaluation time: The evaluation metrics are calculated by dividing the triples into batches and then find ranks of the triples by batch and aggregate the MRR over all the batches. But the time taken per batch is observed to be too high for us to be able to run the evaluation to completion. According to our estimation, it will take about $\tilde{7}0$ days for all the batches to be evaluated. Note that this experiment is ran on CTP.

3.1.1 Bottlenecks

Mean Reciprocal Rank: First, to understand the bottleneck of the evaluation script, let us look at the evaluation metric MRR and how it is calculated. For each triple (s, p, o), we have to rank all the entities in our KB over the scores of (x, p, o) in our model and find the rank of our subject 's' and similarly rank our object 'o' over the scores of (s, p, x). And the we add the reciprocal ranks and average it out to find the MRR.

$$MRR = 1/|Q| * \Sigma_{i=1}^{|Q|} 1/rank_i$$

where $rank_i$ refers to the rank position of the first relevant entity for the i-th triple.

We use a slightly modified version since there are two ranks, that of subject and that of object, to calculate the MRR.

$$MRR = 1/(2 * |Q|) * \Sigma_{i=1}^{|Q|}(1/rank_{s_i} + 1/rank_{o_i})$$

Bottleneck: We can see that this method needs us to call the 'score' function of CTP number of entities times for each triple. This is implemented as the 'forward' function in the models. We identified this as the bottleneck of the evaluation script and also confirmed it using profilers and checking the runtimes of all the functions that are called.

3.1.2 Workaround

We can see that it is really inefficient to calculate scores over all the entities and ranking them. So we need to find a way to reduce the scope to most probable 'k' entities instead of all the entities. That is, given a subject, predicate pair (s, p), we need to come up with a way to extract k entities that are true as the object for this particular subject, predicate pair and vice versa for predicate, object pair.

Other Link prediction models: Our idea is that if we can estimate the object embedding from subject, predicate pair, we can perform a KNN search on our entity embedding space and extract the nearest k entities to our estimate. But there is no way to estimate the third element in the triple given a pair. We decided to look for other standard link prediction models in which such an estimation possible. Then we planned to incorporate those parameters of the model into our evaluation script and reduce the scope of the entities for our evaluation.

TransE

TransE is a standard link prediction model. It models relationships by interpreting them as translations operating on the low-dimensional embeddings of the entities. Relationships are represented as translations in the embedding space: if (s, p, o) holds, the embedding of the tail entity o should be close to the embedding of the head entity s plus some vector that depends on the relationship p.

$$e_s + e_p = e_o$$

Type	Metrics of our implementation	Metrics of standard TransE
MRR	0.209815	-
Hits@1	0.154256	-
Hits@3	0.224470	-
Hits@5	0.260750	-
Hits@10	0.314302	0.349

Table 3.1: Our TransE implementation metrics over FreeBase.

We implemented the TransE model in the same code base as the CTP and GNTP models and tested it on freebase dataset. We observed that we got similar results as the standard implementation of TransE. So we decided that our implementation is good enough to use for the evaluation of CTP.
The following changes are to be made to include TransE in the evaluation:

1. We need to load the pre-trained TransE parameters and pass it to evaluate function.
2. For every triple, we can extract top 'k' entities from TransE embedding space that are closest to TransE estimate for both subject and object canditates.
3. instead of using the 'forward' function, we use 'score function' to calculate scores of all the 'k' entities.
4. Use these ranks to calculate MRR.

3.1.3 Results

As we expected, reducing the scope of entities to rank over heavily improved the time efficiency of the evaluation. The time taken for evaluation is about 12 hours for the test dataset of Freebase.

Type	k=10
MRR	0.563997
Hits@1	0.460471
Hits@3	0.565230
Hits@5	0.641405
Hits@10	0.884442

Table 3.2: Metrics of CTP using TransE parameters over FreeBase.

3.1.4 Analysis

One important thing to note is, the rank we get by extracting top k entities and the rank we get by the scores of all the entities can be vastly different. It also depends on how well the TransE and CTP performs. The better performances of both the models, the closer the actual rank is to the rank we calculate by extracting the top k entities.

For lower k, the metrics will always be higher. Because as k decreases, the worst rank, that is $k+1$, our entity can have decreases. The same can be inferred through our results. So it is preferable for k to be higher as that will generate better metrics, that is closer to the actual metrics without the TransE extraction heuristic. But higher k requires more computational resources both in terms of memory and time. So there should be a trade off between computational capabilities and accuracy of the metrics.

Chapter 4

Code changes by author

4.1 Changes

After being able to generate the metrics for the data, we decided to move on to one of the ideas we had previously to reduce the CTP bottleneck. That is, to introduce a KNN like search at the terminal step of CTP, so that it needs to calculate the scores of top k closest facts instead of all the facts in KB. We decided that this step will make the model much more efficient. As we were looking into this, we discovered that the authors made a new commit a month ago in which they made a few changes.

The changes include the KNN type fact selection, just like GNTP, for the CTP model. The new CTP builds an index on the facts and selects the best facts at the terminal node of CTP. There is also a new evaluation function in which similar indexes are used to get k best entities to rank over. This change should increase the efficiency by a lot. Other changes include a lot of streamlining the code and removing the unnecessary files.

4.2 Results of updated model

4.2.1 command

The following command is used to run the following experiments:

```
nohup python3 ./bin/hoppy-sample-cli.py
--train /home/baadalvm/data/nations/fb15k-273/train.csv
  --test /home/baadalvm/data/nations/fb15k-273/test.tsv
--dev /home/baadalvm/data/nations/fb15k-273/dev.tsv
-e 20 -b 100 --seed 1 --eval-batch-size 100
--save nations_params.txt -k 100 --refresh 100 > log.txt 2>&1
```

4.2.2 Time and Memory usage

Training: The run time for training the model reduced drastically after the KNN implementation of extracting best facts. For 20 epochs, the old implementation took about 2

days to train. While the new model takes about 3 hours to train the same. Memory usage is similar to the old model and well within our computational capabilities.

Evaluation: However, for evaluating the test dataset, the new implementation still takes around 5 days to complete, which we ran once for comparision. While it is much better than the time taken for evaluation of the old model, it is still not desirably fast enough. So we decided to use our trained TransE parameters to extract the top k candidate entities to rank over. Since the 'score' function is much faster in the new model, evaluating TransE is much faster and the evaluation only takes about 15 minutes to complete.

Type	**ctp eval**	**k=10**	**k=25**	**k=50**	**k=100**
MRR	0.068452	0.592663	0.532390	0.495425	0.458804
Hits@1	0.044654	0.487311	0.453236	0.428400	0.401027
Hits@3	0.072769	0.604790	0.541745	0.506587	0.467636
Hits@5	0.088081	0.679840	0.590362	0.545994	0.500599
Hits@10	0.110294	0.931280	0.673396	0.613630	0.556886

Table 4.1: Metrics of updated CTP using TransE parameters over FreeBase dev set.

Type	**k=10**	**k=25**	**k=50**	**k=100**
MRR	0.590076	0.528809	0.502258	0.449609
Hits@1	0.483460	0.447718	0.436822	0.391625
Hits@3	0.602365	0.542290	0.511189	0.458028
Hits@5	0.679859	0.587462	0.551549	0.490301
Hits@10	0.924216	0.673556	0.614629	0.545832

Table 4.2: Metrics of updated CTP using TransE parameters over FreeBase test set.

k	**time taken per batch(sec)**
10	5.5
25	8.52
50	12.58
100	20.5

Table 4.3: Time taken to evaluate a batch.

k	memory usage (GB)
10	2.8
25	2.9
50	3.1
100	3.4

Table 4.4: Time taken to evaluate a batch.

4.3 Analysis

We can see that the results of the evaluation using TransE are as expected. As k increases, all the metrics (MRR and hits) are decreased, since more entities are being considered for ranking, the rank our desired entity can remain same as before or get worse. The time taken and memory usage increases with increase in k as expected.

CTP evaluation vs TransE evaluation: Despite the large runtime of CTP evaluation, we ran it to see how the results compare to our implementation. As we can see, the CTP evaluation produces results that are really bad compared to ours. Part of the reason can be the heuristic used to select the entity candidates. Their heuristic is to build an index on the facts using (s, p) as the key and o as the values and vice versa, while TransE trains its parameters so that the missing entity can be estimated. Their heuristic isn't designed to give the entities that are not in the training facts and the similarity in the third entity is not being considered. Thus TransE might be a better heuristic in selecting entities for ranking.

It is also to be noted that we say that TransE evaluation is a better heuristic in terms of the results we obtained. But our original goal is to approximate/replicate the results of CTP, i.e, when all entities are being ranked for all triples, which is computationally infeasible. So, in approximating just how CTP performs, for the same k, whichever has the lowest metrics, is the better heuristic. This is because, ultimately, the CTP model gives the scores of the selected entities and if the rank of an entity as per one heuristic is lower than that of the other heuristic, that means the heuristic is not selecting other entities that might have a better CTP score. So in that way their heuristic is the better one in estimating the CTP performance.

Our implementation can be looked at as a new model rather than approximating performance of CTP. It can be looked at as combination of CTP and TransE, where we train both the models and when evaluating we extract the best entities using TransE entities and then

rank them using CTP scores. The metrics we obtain are really good and close to standard link prediction models on freebase dataset.

Our implementation can be looked at as a new model rather than approximating performance of CTP. It can be looked at as combination of CTP and TransE, where we train both the models and when evaluating we extract the best entities using TransE entities and then rank them using CTP scores. The metrics we obtain are really good and close to standard link prediction models on freebase dataset. But for smaller values of k, any two link prediction models combined can produce really good results as the worst case rank is drastically reduced and the scoring model takes advantage from the selecting model. Below is the result obtained on freebase dataset, by training a ComplEx model and a TransE model. We used TransE for KNN selection of entities and ComplEx for scoring.

Type	**dev set**
MRR	0.687651
Hits@1	0.593790
Hits@3	0.708272
Hits@5	0.789553
Hits@10	0.967385

Table 4.5: ComplEx evaluation with TransE selection with k=10 on Freebase dev set.

Chapter 5

Code study

5.1 Files Discussed

1. Main script: bin/hoppy-cli.py
2. Data class: ctp/training/data.py
3. Knowledge Base class: ctp/smart/kb.py
4. CTP model: ctp/smart/simple.py
5. evaluation script: ctp/evaluation/slow.py
6. evaluation with TransE: ctp/evaluation/transe.py

5.1.1 Main Script

The script does the following:

1. Loads the training, dev, test data into a Data class object.
2. ```
   data = Data(train_path=train_path, dev_path=dev_path,
    test_path=test_path, test_i_path=test_i_path,
    test_ii_path=test_ii_path, input_type=input_type)
   ```
3. Creates a NeuralKB model with the Data object as an input.
4. ```
   base_model = NeuralKB(entity_embeddings=entity_embeddings,
   predicate_embeddings=predicate_embeddings, k=k_max,
   facts=facts, kernel=kernel, device=device, index_type=index_type,
   refresh_interval=refresh_interval).to(device)
   ```
5. Initializes the goal reformulators (select module) based on number of hops given.
6. ```
   def make_hop(s: str) -> Tuple[BaseReformulator, bool]:
           nonlocal memory
           if s.isdigit():
               nb_hops, is_reversed = int(s), False
           else:
               nb_hops, is_reversed = int(s[:-1]), True
           res = None
           if reformulator_type in {'static'}:
   ```
   ```

```
            res = StaticReformulator(nb_hops, embedding_size,
                init_name=ref_init_type, lower_bound=lower_bound,
                upper_bound=upper_bound)
        elif reformulator_type in {'linear'}:
            res = LinearReformulator(nb_hops, embedding_size,
                init_name=ref_init_type, lower_bound=lower_bound,
                upper_bound=upper_bound)
        elif reformulator_type in {'attentive'}:
            res = AttentiveReformulator(nb_hops, predicate_embeddings,
                init_name=ref_init_type, lower_bound=lower_bound,
                upper_bound=upper_bound)
        elif reformulator_type in {'memory'}:
            if nb_hops not in memory:
                memory[nb_hops] = MemoryReformulator.Memory(nb_hops,
                                  nb_rules, embedding_size, init_name=ref

            res = MemoryReformulator(memory[nb_hops])
        elif reformulator_type in {'ntp'}:
            res = NTPReformulator(nb_hops=nb_hops,
                embedding_size=embedding_size, kernel=kernel,
                init_name=ref_init_type, lower_bound=lower_bound,
                upper_bound=upper_bound)
        assert res is not None
        return res, is_reversed
```

7. Initializes the CTP model (called the hoppy/SimpleHoppy model in the code) with NeuralKB object as an input.

8. `model = SimpleHoppy(model=base_model, entity_embeddings= entity_embeddings, hops_lst=hops_lst).to(device)`

9. Proceeds to train the model and then evaluate it on test and dev datasets.

5.1.2 Data Class

Loads the triples from all the files mentioned.

```
def read_triples(path: str) -> List[Tuple[str, str, str]]:
    triples = []
    with open(path, 'rt') as f:
        for line in f.readlines():
            s, p, o = line.split()
            triples += [(s.strip(), p.strip(), o.strip())]
    return triples
```

5.1.3 NeuralKB

The main components of this class are:

1. Entity embeddings
2. Predicate embeddings
3. Facts
4. Indices for $(s, p, o), (s, p), (p, o)$ search.
5.
```
index_factory = {
                'faiss': lambda: FAISSSearchIndex(),
                'np': lambda: NPSearchIndex(),
                'nms': lambda: NMSSearchIndex()
            }

    assert self.index_type in index_factory

    self.index_sp = index_factory[self.index_type]()
    self.index_po = index_factory[self.index_type]()
    self.index_spo = index_factory[self.index_type]()
```

Methods

score: The score function takes a triple (a batch of them) as an input and selects the closest facts in our Knowledge Base using index_spo and then calculates the similarity between the goal triple and the facts selected. The scores are max pooled and then returned.

```
_, _, neigh_spo = self.neighbors(rel, arg1, arg2, f_rel_emb,
        f_arg1_emb, f_arg2_emb, is_spo=True)

neigh_spo_tensor = self.to_tnsr(neigh_spo)
...
scores = self.kernel(b_emb_2d, f_emb_2d).view(batch_size, -1)

res, _ = torch.max(scores, dim=1)
return res
```

forward: The forward function is used for evaluating a triple, i.e, for every triple (s, p, o), rank s w.r.t. (p, o) and rank p w.r.t. (s, p). There are multiple implementations of forward function.

1. forward: finding k best entities using index_sp and index_po for the batch of triples as a whole. That is, we get k entities to rank over for the whole batch of triples.

2. ```
   # for all the triples in the batch at a time.
   neigh_sp, neigh_po, _ = self.neighbors(rel, arg1, arg2,
           f_rel_emb, f_arg1_emb, f_arg2_emb)
   ```

3. forward_: Brute force implementation, ranks over all the entities in Knowledge Base.

4. forward__: (Author's preferred method) finding k best entity for each of the triple instead of for the whole batch.

5. ```
   # for each the triple in the batch at a time.
   for i in range(batch_size):
           if arg1 is not None:
                   ...
                   _, _, neigh_spo = self.neighbors(i_emb_rel,
                   i_emb_arg1, emb, f_rel_emb, f_arg1_emb,
                   f_arg2_emb, is_spo=True)
                   neigh_spo_tensor = self.to_tnsr(neigh_spo)
   ```

5.1.4 CTP/SimpleHoppy model

The main components of this class are:

1. Entity embeddings
2. Predicate embeddings
3. NeuralKB model
4. Reformulators (hops_lst in code)

Reformulators: Reformulators are the select module that learns to generate rules from the data that we discussed in the theory section. We use multiple select modules for practical purposes, like generating rules with different templates.

$$select_\theta(G) = [F_H(G) : -F_{B_1}(G), F_{B_2}(G)]$$

where

1. $F_H(G) = [f_H(\theta_G), X, Y]$
2. $F_{B_1} = [f_{B_1}(\theta_G), X, Z]$
3. $F_{B_2} = [f_{B_2}(\theta_G, Z, Y]$

Methods: This class also has functions score, forward (all variations) like NeuralKB. These methods take the goal triples and expand them on reformulators/select module to generate rules and thus generating new atoms. At the terminal step, SimpleHoppy model simply calls the respective score or forward functions of NeuralKB model of the final ground atoms. So the structure of all these functions are very similar and the method called on the NeuralKB model is about the only thing that differs.

5.1.5 Evaluation script

Forward method of the CTP model is called on batch wise data, and we will get the scores of all the entities w.r.t. subject, predicate pair and vice versa for predicate, object pair. These scores are processed to remove the entities that we already have in our database. We simply make the scores of those entities to be negative infinity. We then find the scores of our subject and our predicate and proceed to find MRR and hit values.

```
if model.model.facts[0].shape[0] < 90000:
                res_sp, res_po = model.forward_(tensor_xp_emb,
                                                tensor_xs_emb, tensor_xo_emb)
            else:
                res_sp, res_po = model.forward__(tensor_xp_emb,
                                                 tensor_xs_emb, tensor_xo_emb)

            _scores_sp, _ = res_sp
            _scores_po, _ = res_po

            scores_sp, scores_po = _scores_sp.cpu().numpy(),
                                   _scores_po.cpu().numpy()

...

for elem_idx in range(batch_size):
        for tmp_o_idx in o_to_remove:
                if tmp_o_idx != o_idx:
                    scores_sp[elem_idx, tmp_o_idx] = -np.infty

        for tmp_s_idx in s_to_remove:
                if tmp_s_idx != s_idx:
                    scores_po[elem_idx, tmp_s_idx] = -np.infty

        rank_l = 1 + np.argsort(np.argsort(
```

```
                    - scores_po[elem_idx, :]))[s_idx]
        rank_r = 1 + np.argsort(np.argsort(
                    - scores_sp[elem_idx, :]))[o_idx]
        mrr += 1.0 / rank_l
        mrr += 1.0 / rank_r

        for n in hits_at:
            hits_at_n(n, rank_l)

        for n in hits_at:
            hits_at_n(n, rank_r)
```

5.1.6 Evaluation with TransE

Since the model's evaluation script is very slow, we used TransE to try to approximate the evaluation metrics. We load the TransE parameters in the main script before starting the evaluation and pass it as an input to the evaluate function. Now in the evaluate function, for each triple (s, p, o), using TransE embeddings, we estimate o_{est} and then do a KNN search on TransE embeddings to select best k entities as candidate objects. We then calculate the scores for all these entities and find the rank of the object o and vice versa for subject s.

```
# index used to do KNN search on TransE embeddings
transe_index = NMSSearchIndex()
transe_index.build(transe_entity_embeddings.cpu().detach().numpy())
...
#getting the k best entities
sp_emb = [o] + transe_index.query(diff_sp,
        k=num_entities_select)[0].tolist()
po_emb = [s] + transe_index.query(diff_po,
        k=num_entities_select)[0].tolist()
...
```

Chapter 6

References

1. Minervini, Pasquale, et al. "Learning reasoning strategies in end-to-end differentiable proving." International Conference on Machine Learning. PMLR, 2020.

2. Rocktäschel, Tim, and Sebastian Riedel. "End-to-end differentiable proving." Advances in Neural Information Processing Systems. 2017.

3. Minervini, Pasquale, et al. "Differentiable Reasoning on Large Knowledge Bases and Natural Language." (2020): 125-142.

4. Trouillon, Théo, et al. "Complex embeddings for simple link prediction." International Conference on Machine Learning (ICML), 2016.

5. Bordes, Antoine, et al. "Translating embeddings for modeling multi-relational data." Advances in neural information processing systems 26 (2013): 2787-2795.

6. Bordes, Antoine and Usunier, Nicolas and Garcia-Duran, Alberto and Weston, Jason and Yakhnenko, Oksana, et al. "Translating embeddings for modeling multi-relational data"

7. Source code of CTP: `https://github.com/uclnlp/ctp`

8. Source code of GNTP: `https://github.com/uclnlp/gntp`