Advances in Cognitive Systems Tenth (2022) 1-6 Submitted 9/2022; published 11/2022

Getting Help from the Neighborhood: Local Semantic Averaging for Commonsense Inference in Language Models

Claire Yin CYIN@ALUM.MIT.EDU
MIT Computer Science and Artificial Intelligence Lab, Cambridge, MA 02139 USA

Pedro Colon-Hernandez PE25171@MIT.EDU
MIT Media Lab, Cambridge, MA 02139 USA

Henry Lieberman LIEBER@MEDIA.MIT.EDU
MIT Computer Science and Artificial Intelligence Lab, Cambridge, MA 02139 USA

Abstract

While language models have impressive performance in prediction tasks, they often fail on examples that seem simple to humans. The failures can often be described as a result of models lacking common sense. There have been a number of efforts to encode commonsense knowledge using knowledge graphs, but the challenge remains of how best to perform commonsense inference in a way that has the intuitiveness of the knowledge graphs and the coverage and predictive power of the language models. Relational graph convolutional networks (RGCNs) hold promise for approximate commonsense inference, because they perform a "local semantic averaging" operation. Just as convolutional neural networks (CNNs) aggregate neighboring pixels for computer vision tasks, GCNs aggregate information from neighboring nodes of a graph. RGCNs use relational data to learn how to aggregate this information. We introduce three methods to inject commonsense knowledge into contextual language representations. We show that the representations learned from an RGCN, although trained on considerably less data, still prove useful in a downstream information retrieval task when combined with a transformer-based language model. We find that the earlier the commonsense knowledge injection, the better the performance of the language model on such tasks.

1. Introduction

Understanding language on a human level has two aspects: syntactic and semantic knowledge. Syntactic knowledge captures the way grammar and sentence structure are used to form text, while semantic knowledge captures the meaning of text. Natural Language Processing (NLP) attempts use large, powerful models to learn natural language on a human level. Recently there have been many advances in NLP due to the introduction of powerful pre-trained transformer-based language models. These models implement attention mechanisms to generate useful contextual representations of text that allow for greatly improved performance of various NLP benchmarks [Vaswani et al. (2017)]. Research has analyzed the weights of models to determine what information these models leverage to perform these tasks. There is evidence that they use syntactic knowledge as well as some semantic knowledge [Petroni et al. (2019)]. However, it appears that at times, some of the semantic knowledge learned by these models is inaccurate. This causes the model to hallucinate or produce false information in downstream tasks. This hallucination can be the result of biased or noisy training data that simply induces the storage of incorrect knowledge [Colon-Hernandez et al. (2021)].

This leads to the thought that these models lack some useful knowledge for understanding the meaning of text and for performing well on NLP tasks. If this knowledge can be better captured by these language models, then performance on NLP tasks can be further improved. One resource to help do this is knowledge graphs. Knowledge graphs are collections of knowledge in a structured manner, explicitly encapsulating information in the form of a graph data structure, where nodes represent entities and edges represent relationships between the entities. If we can encode and combine the information in these graphs, with that of the pre-trained transformer-based language models, then we could supplement and correct information that may be missing or is wrong from the pre-trained models. This gives us powerful language models that are rich with commonsense, thus able to read between the lines, tackle more complicated language tasks, and be more interpretable to humans.

In this paper, our goal is to answer the following questions:

1. Can we harness the power of large KGs to improve the performance of large language models on downstream language-related tasks?
2. Is it possible to combine a relational graph convolution network with a transformer-based language model to achieve performance gain in a downstream task?

We first implement a model to learn meaningful embeddings from knowledge graphs. We use a relational graph convolutional network (RGCN), which exploits the multi-relational information encapsulated by edges in graph data. In comparison to other knowledge graph learning models, the RGCN is advantageous in that it allows for information to be effectively shared between neighboring entities, therefore creating more contextually aware embeddings. After testing and validating the RGCN model, we combine the entity embeddings along with the actual model with a transformer-based language model (BERT) to see the effects in a downstream task. Namely, we explore an information retrieval task to rank similar help questions. We test three methods for combining these models, which will be introduced in more depth in chapter 3: input injection, architecture injection, and output injection. We compare the performance of these three methods alongside other baseline methods to determine whether KG can inform and improve learning for language tasks.

This paper will continue with a summary of previous works related to the proposed question. In Section 3, we describe the model implementations, providing a thorough overview of the entire pipeline. In Section 4, the experiments conducted will be described in detail. Section 5 presents and discusses the results of the experiments, including proposed explanations and justification for them. In Section 6, we discuss the future downstream tasks of the model. Section 7 concludes by summarizing results and future work.

2. Prior Work

In the domain of natural language understanding, transformer-based models are large and powerful, and can attain high performance on different tasks. One such model is BERT [Devlin et al. (2019)]. BERT stands for "Bidirectional Encoder Representations from Transformers" trains a transformer encoder with a masked language modeling objective on a large dataset [Vaswani et al. (2017)]. This pre-training in turns allows the BERT model to generate effective contextual representations for input text. Because BERT is pre-trained on language data, the output layers of the model can be fine tuned and adapted to a variety of NLP tasks and achieve good performance.

In comparison to our methodology, the most similar work is KagNet. Lin et al. (2019) explore the use of GCNs, an architecture similar to RGCNs but lacking the capability of using relational information. In order to answer commonsense questions, they use KagNet to score answers with graph representations. The model trains on ConceptNet and achieves impressive performance on CommonsenseQA.

Commonsense Knowledge

In our experiments, we will be using a subset of ConceptNet Liu & Singh (2004). Conceptnet is a collection of facts which humans deem to be "common sense". Commonsense facts are typically simple things that we humans take for granted, and assume that others also know. Because of this, this knowledge is typically not written down. ConceptNet is a collection of these facts. It is represented as a knowledge graph of entities connected by labeled and weighted edges. ConceptNet data is is in the form of (*subject*, *relation*, *object*) triples, with subjects and objects as nodes and relations as the labeled edges of the graph. The data are a compilation of "expert-created resources, crowd-sourcing, and games with a purpose" Liu & Singh (2004); Speer et al. (2018).

Examples of assertions found in ConceptNet alongside their (*subject*, *relation*, *object*) triple representations include:

- A *dog* has a *tail* $\iff$ (*dog*, *HasA*, *tail*)
- A *vacuum* is capable of *cleaning the floor* $\iff$ (*vacuum*, *CapableOf*, *clean floor*)
- A *tool* is an *object* $\iff$ (*tool*, *IsA*, *object*)
- *Going for a jog* causes you to *sweat* $\iff$ (*go for jog*, *Causes*, *sweat*)
- *Bridges* are used for *crossing water* $\iff$ (*bridge*, *UsedFor*, *cross water*)

Across all languages, the entire ConceptNet dataset contains over 21 million edges (relations) between over 8 million nodes (entities). The subset that is in English contains approximately $1,500,000$ nodes. The full data can be downloaded from the ConceptNet 5 GitHub [1]. Due to the large size and high connectivity, we work with subsets of ConceptNet. In our work we use a subset generated from [2] Li et al. (2016).

Apart from ConceptNet, another popular commonsense knowledge graph is ATOMIC Sap et al. (2019). ATOMIC is a collection of if-then facts, which complement ConceptNet's taxonomic

1. GitHub available here `https://github.com/commonsense/conceptnet5/`
2. Available here `https://home.ttic.edu/~kgimpel/commonsense.html`

knowledge. Although we do not use it in our work, we leave it as future work to add this knowledge into our system.

Language models enhanced with commonsense knowledge

Many recent works explore methods for creating more commonsense aware language models, and show how these methods can improve model performance. Zhou et al. (2020) introduce concept-aware language models (CALM) as a method to pre-train language models on commonsense knowledge without the use of knowledge graphs. The transformer-based model jointly uses a generative objective and a discriminative objective to distinguish commonsense-aware sentences in a self-supervised manner. Just pre-training on a small corpus of Wikipedia data, they show significant improvement on natural language understanding (NLU) and natural language generation (NLG) tasks.

Overfitting and bias in fine-tuned models

However, some studies show that such fine-tuned models may be prone to overfitting. Kejriwal & Shen (2020) perform quality and consistency analyses to prove the lack of generalizability of these models. Alongside, Ma et al. (2021) perform model training on different partitions of datasets to measure generalizability. They find that fine-tuning, in comparison to other model adaptation methods, performs well but generalizes poorly to unseen data.

Relational knowledge with language models

Other studies have explored how to combine relational information from knowledge graphs with powerful transformer-based language models, but none have explored the use of RGCNs. One relevant study introduces KnowBERT for architecture injection [Peters et al. (2019)]. They use contextual word representations at the core of their model, augmenting them with explicit, commonsensical embeddings from knowledge bases (KB). They use human-curated knowledge from Wikipedia and WordNet to enhance a BERT model. This results in a knowledge enhanced BERT, hence KnowBERT. Overall, their experiments show that KnowBERT has improved recall on facts in probing tasks and relationship extraction, perplexity, entity typing, and defining ambiguous words. In addition to a significant improvement in task performance, KnowBERT also has a runtime comparable to an ordinary BERT model. Their evaluations suggest that commonsense knowledge embeddings can improve model quality and performance on predictive tasks. Another model is LP-BERT [Li et al. (2022)] for the task of link prediction. They propose a knowledge graph BERT which first undergoes multi-task pre-training on the knowledge graph. Then, the model undergoes knowledge graph fine-tuning that uses negative sampling from positive samples of the KG. This model achieves state-of-art results on link prediction tasks [Li et al. (2022)] for WordNet and Freebase datasets.

There has been additional work on evaluating what commonsense information is present in language models such as BERT Da & Kasai (2019). The authors finde 5 categories of knowledge (namely Visual, Encyclopedic, Functional Perceptual, Taxonomic) that BERT is lacking on. The authors then sample from an additional dataset (RACE dataset Lai et al. (2017)) to find and fine tune on passages in these categories that may be help a BERT model compensate deficiencies in the areas. In addition to this, the authors concatenate the fine tuned BERT embeddings with some knowledge graph embeddings from simple LSTM encoded text assertions that involve the entities that are present in the questions and passages they train their final joint model on (MCScript 2.0 Ostermann et al. (2019)). It is worth noting that the graph embeddings that they concatenate, albeit simple, boost the performance of their system which shows that there is still some information in KGs that is not in BERT.

3. Models

Our method infuses commonsense knowledge into language models and consists of two stages with one model each. In the first stage, an RGCN creates commonsense-informed embeddings representations. In the second stage, we combine the knowledge with contextual language representations to perform an information retrieval task that is described in the next section.

3.1 GCN

We explore GCNs as a method to produce knowledge embeddings from ConceptNet. A GCN functions by performing updates to node features using aggregation functions over neighboring node features. To do so, it encodes a graph structure directly with a stack of convolutional layers. The model uses "an efficient layerwise propagation rule that is based on a first-order approximation of spectral convolutions on graphs" and significantly outperforms preceding methods for graph learning [Kipf & Welling (2016)].

3.2 Single Relation GCN

A downside to GCNs is that they can only handle one kind of relation type. This hinders their performance when applied to graphs that are multi-relational. RGCNs were developed to handle multi-relational data. Compared to GCNs, which extract node representations on graphs with untyped edges, RGCNs can handle different relationships between entities. By encoding different types of relations and connections alongside the structure of the data, RGCNs are capable of learning more meaningful node embeddings. This network is useful for tasks such as link prediction between entities, and entity classification based on individual and relational information.

As an intermediate step, we build a single-relation RGCN, where the set of relation types R has a size of 1. Similar to the GCN, the single-relation RGCN is incapable of distinguishing between different edge types, so we also train each single-relation RGCN on a subset of the data.

3.3 RGCN performs local semantic averaging

Finally, we build an RGCN-based model consisting of relational graph convolution network (RGCN) layers that is extended to multi-relational data. With the knowledge base in graph representation, an RGCN can learn to encode multi-hop relational knowledge from ConceptNet using neural architectures thereby producing embeddings that give even more commonsense/knowledge relationships between words. RGCNs can capture the context of entities by pooling information from related entities. They learn valuable latent features of relational graphs using message passing mechanisms, directly extracting features of neighboring entities and creating contextually rich representations.

While GCNs [Kipf & Welling (2016)] are able to learn from unlabelled graph data by using shared weights for each relation, RGCNs learn separate weights for each relation and thus are adaptable to relational data, making it a very useful model for learning with knowledge graphs. This model is based on the implementation of RGCN for link prediction by the DGL0.8 library [Wang et al. (2019)].

3.4 Methods for injection of information into language model

For the language model, we use BERT. BERT leverages the transformer mechanism to model long-range dependencies between text and learn from the left and right context of text. This modeling encodes syntactic knowledge and, to a certain extent, some semantic knowledge contained in unstructured texts. In our work we utilize the "bert-base-uncased" model from HuggingFace, which consists of 12 encoder layers and contains 110 million parameters.

Using the BERT base model, we attempt three of the methods previously defined and investigated by Colon-Hernandez et al. (2021): input injection, architecture injection, and output injection. We build a separate model for each of the methods. Modifications and implementation details are described below.

3.4.1 Input Model

To perform an *input injection* we modified the base BERT in the following way. After the token embeddings are retrieved, and before the first transformer attention layer, we decode the token ids to produce a sentence. We then run this sentence through a spaCy [Honnibal & Montani (2017)] pattern matcher that has been modified to serve as an entity recognizer with the names of entities (i.e., vocabulary) from the embeddings that we are injecting (i.e., NumberBatch or our RGCN results). We then find the corresponding input ids and their token embeddings, and to the ones that have matching entities, we rescale and sum the entity embedding to the token embedding. We do this rescaling and sum with the same injected embedding for every one of the tokens that encompass the entity (e.g., for New York, for the tokens New and York, we use the embedding that corresponds to New York). For the entities that are not in our vocabulary, we simply utilize a vector of zeros. It is worth noting that we do not pre-train on any additional data for this model; we solely fine-tune it on our downstream task which is the relevant question retrieval task. We will go into details on this task in Chapter 4. The approach can be seen in Figure 1.

3.4.2 Architecture Model

To perform an *architecture injection*, we followed a similar approach to our input injection, however instead of summing our rescaled embeddings at the input, we sum them at the beginning of each of the the attention layers, except the first layer (otherwise it would be the same as the input injection). This approach can be seen in Figure 2. We utilize a different rescaling linear layer in each layer, to let the model inject information in the layers that it finds more effective.

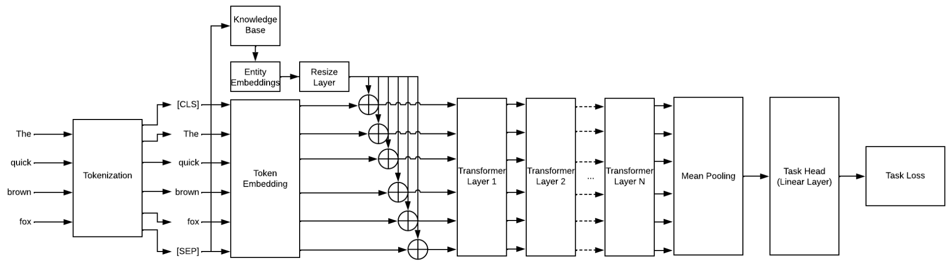

Figure 1. Input injection model.

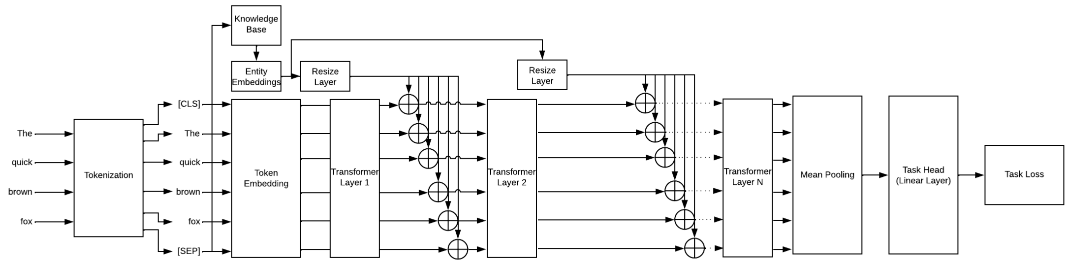

Figure 2. KnowBERT model used for architecture injection.

3.4.3 Output Model

Lastly, to perform an *output injection* we modified the base BERT in the following way. After the final layer, we perform the same process as in *input injection* to find the corresponding injected embeddings for an input sequence and rescale and add them.

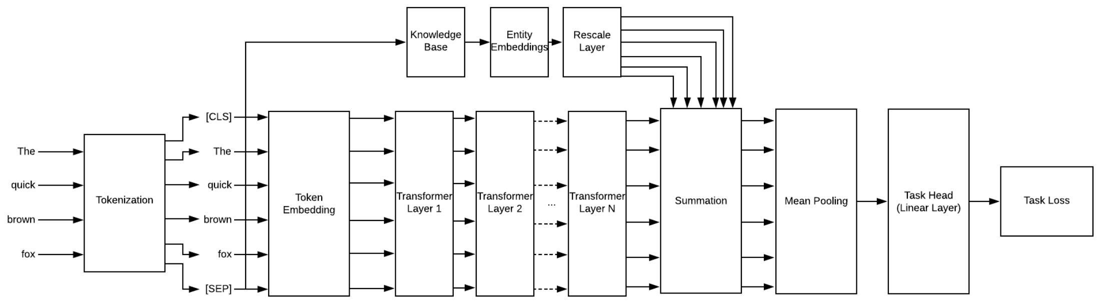

Figure 3. Output injection model.

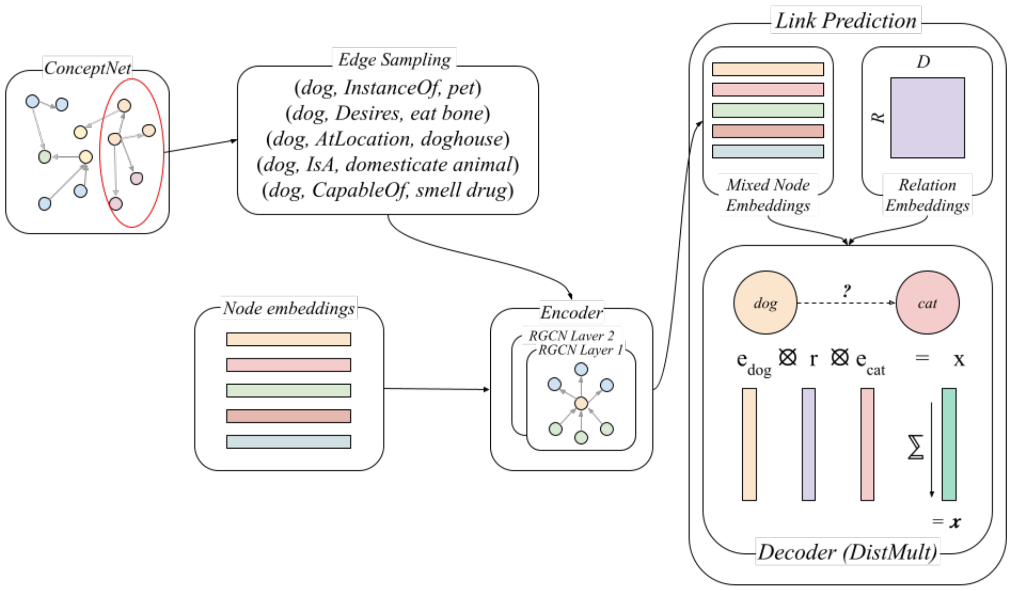

Figure 4. Figure showing RGCN with link prediction on ConceptNet adapted from Thanapalasingam et al. (2021). After edge sampling, the correspoding node embeddings and graph structure are used by the encoder (RGCN layers) to create enriched node embeddings. For triple *(s,r,o)*, corresponding relation and entity embeddings are element-wise multiplied to result in vector x. The elements of this vector are summed to give a scalar x for the probability of *(s,r,o)* being true.

3.5 Pipeline Overview

Overall, the pipeline contains an RGCN Link Prediction model, as well as a knowledge injected BERT model.

First, the RGCN link prediction model takes into account the ConceptNet dataset. It uses RGCN layers to encode node embeddings and relation embeddings, and then predicts relations between two entities. A concrete example is depicted in Figure 3.5. Suppose we take the subset of ConceptNet in the red circle, with *dog* as the central node. We sample the edges connected to the *dog* entity and retrieve node embeddings for each of the connected entities. Then, we feed those through the RGCN layers of the encoder model. This gives us mixed node embeddings that convolute the subgraph to update the embeddings of *dog* as well as its neighbors (*pet*, *eat bone*, etc.). We repeat this for all nodes in the graph. This results in mixed node embeddings as well as relation embeddings that the model updates through the training process. Using these two sets embeddings, we can perform the link prediction task. We run these embeddings through the decoder, in this case the DistMult function, resulting in predictions for each relation.

We can then extract the embeddings learned by the RGCN Link Prediction model in order to an information retrieval-based BERT model. To do this, we follow an approach as described in Section 3.4.1, which utilizes a matcher to find the entities that overlap with the vocabulary of our RGCN. We then rescale and sum these matched embeddings into a corresponding *input, output, or architecture* injection. Finally, we pool the corresponding BERT model's embeddings by averaging them and utilize this as a sentence/input text representation. With this representation, we can then utilize cosine distance to find the most similar embeddings to an input text. We describe this task more formally in the Chapter 4.

4. Evaluation

4.1 RGCN Evaluation

We evaluate our GCN and single RGCN models to serve as baselines for the full RGCN model. For these evaluations, we use the following performance metrics: area under the ROC curve (AUC), accuracy, precision, and recall. We also evaluate the RGCN using two commonly used evaluation metrics: mean reciprocal rank (MRR) and hits@k. As in Schlichtkrull et al. (2017), we investigate the MRR values.

4.2 Evaluation of Injection Methods

Once we achieve an acceptable performance in our RGCN, we proceeded to combine the node graph embedding output from this model into our various injected BERT models. We developed each injection model following the descriptions in Chapter 3.4. With these models, and a spaCy matcher for the vocabulary for our embeddings, we then applied the models to the task of related question retrieval.

Baseline 1: BERT

As a first baseline, we perform link prediction using just BERT. Because this is an unenhanced version of our final model, we expect significantly improved performance from this baseline.

Baseline 2: Numberbatch + BERT

As a second baseline, we combine pre-trained Numberbatch embeddings (pre-trained on the entirety of ConceptNet's 1M+ assertions) with the BERT model. We expect this combination to outperform the single standing BERT model. This is one form of encoding structured information, and we compare the results of our RGCN graph embeddings against this.

Downstream Task: Related Question Retrieval in the AskUbuntu Dataset

For testing our models, we chose the AskUbuntu [dos Santos et al. (2015)] dataset. The objective of this dataset is to match a given help question with related questions. In the dataset, we get a list of queries along with possible alternatives and confounder (unrelated) alternatives. We chose this task, because to determine similar/unrelated questions, the model needs to form a semantic embedding of

Title: How can I boot Ubuntu from a USB?
Body: I bought a Compaq pc with Windows 8 a few months ago and now I want to install Ubuntu but still keep Windows 8. I tried Webi but when my pc restarts it read ERROR 0x000007b. I know that Windows 8 has a thing about not letting you have Ubuntu but I still want to have both OS without actually losing all my data ...

Title: When I want to install Ubuntu on my laptop I'll have to erase all my data. "Alonge side windows" doesnt appear
Body: I want to install Ubuntu from a Usb drive. It says I have to erase all my data but I want to install it along side Windows 8. The "Install alongside windows" option doesn't appear. What appear is, ...

Figure 5. An example taken from Lei et al. (2015) that shows two similar questions in the AskUbuntu dataset. A model trained on this dataset should predict that these questions are similar/highly related.

the actual input. What this means is that if we inject semantic information, this final representation should contain the additional information and help in the task. Specifically, in our task we pool the final layer embeddings by averaging, and use cosine similarity to rank the most similar question to a Query in the given alternatives. An example of similar questions can be seen in Figure 5.

With our BERT models, the output of the task layers is used to calculate the cosine similarity for the query and the possible alternative. A visualization of how this works is shown in Figure 6. Our model tries to predict whether the two things are similar or not: similar, relevant questions are given a label of 1, whereas dissimilar questions are given a label of 0. We use the cosine similarity as a prediction input for the mean squared error loss. In doing this, we are asking the model to find the relevant features that make questions similar/dissimilar.

5. Results and Discussion

In this section we present the results from our testing. We present our significance results in the Appendix in A.

5.1 Results

5.1.1 Baselines: GCN and Single-relation RGCN

First, we perform baseline experiments with a GCN, a network that cannot encode distinct edge types. Because of this, we train one model per relation and compute metric values averaged over every relation type. To perform this training, we filter the training data based on the number of relations. Specifically, we compute the median number of edges across all relation types in `train100k` to be 961, and filter out relations that have less than 961 edges. We use this set of relations across all three datasets. The results are shown in Table 1.

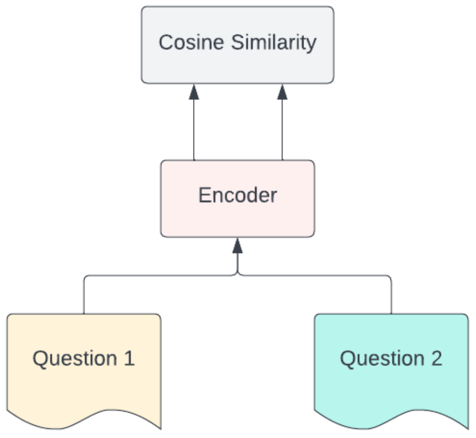

Figure 6. An example based on Lei et al. (2015) that shows the architecture of a cosine similarity model. During training, questions that are similar will achieve higher similarity scores while dissimilar questions will receive lower/negative scores.

Second, we perform baseline experiments on an RGCN that can only encode one relation type. As in the GCN training, we train an individual model on each of the top 16 unique relation types and evaluate the model on accuracy, AUC, precision, and recall. We repeat this for each of the three datasets. The average metrics across the 16 relations for each of the datasets are reported in Table 1.

5.1.2 RGCN

In comparison to the GCNs and single-relation RGCNs, the RGCN model is capable of encoding different relation types. Thus, we train one model on the entire dataset, for each of the datasets (`train100k`, `train300k`, `train600k`). We evaluate the RGCN using accuracy, AUC, precision, recall, and MRR. The results are shown in Table 1.

5.1.3 Injection Models

The results for each of the injection methods on the AskUbuntu dataset are shown below. in Table 2.

5.2 Discussion

5.2.1 GCN and Single-relation RGCN

Overall, the baseline, single-relation GCN achieves high accuracy and AUC ($\sim 70 - 81\%$), and lower precision ($\sim 20 - 46\%$) and recall ($\sim 58 - 65\%$).

For the single-relation RGCN, generally, the model's performance in every metric except recall experiences a slight decrease as the number of samples in the dataset increases. The decrease could

Model	Data	Accuracy	AUC	Precision	Recall	MRR
GCN	`train100k`	$71.35 \pm 0.72\%$	74.41 ± 0.45	45.55 ± 0.62	58.07 ± 0.73	–
	`train300k`	$79.82 \pm 0.73\%$	$\mathbf{80.83 \pm 0.64}$	34.05 ± 3.15	64.17 ± 0.38	–
	`train600k`	$\mathbf{80.17 \pm 0.40\%}$	80.17 ± 0.72	20.61 ± 0.74	61.07 ± 0.71	–
Single-relation RGCN	`train100k`	$72.02 \pm 2.90\%$	77.28 ± 1.83	$\mathbf{75.77 \pm 4.24}$	63.48 ± 3.41	–
	`train300k`	$71.39 \pm 2.65\%$	73.90 ± 4.26	70.41 ± 3.97	$\mathbf{72.15 \pm 2.37}$	–
	`train600k`	$70.44 \pm 1.19\%$	73.39 ± 2.45	67.75 ± 2.37	70.88 ± 1.99	–
RGCN*	`train100k`	$\underline{84.67 \pm 0.81\%}$	$\underline{90.01 \pm 0.63}$	$\underline{80.50 \pm 1.77}$	91.58 ± 1.92	$\underline{93.18 \pm 2.60}$
	`train300k`	$81.52 \pm 1.01\%$	88.42 ± 0.79	74.11 ± 1.15	$\underline{96.98 \pm 0.40}$	79.92 ± 2.79

Table 1. The results for the GCN, single-relation RGCN, and RGCN experiments are shown above, along with the confidence intervals from repeated experimentation. The accuracy, AUC, precision, and recall metrics provide sufficient evidence that the RGCN outperforms the baseline models, with higher results in each metric. Between the baseline models, the GCN model achieves higher accuracy and AUC, while the single-relation RGCN achieves higher precision and recall. The overall top scores are $\underline{\text{underlined}}$ on the table, and the top scores between the baseline models are **bolded**.

	Validation				Testing			
Model	**MAP**	**MRR**	**P@1**	**P@5**	**MAP**	**MRR**	**P@1**	**P@5**
Base Model	61.867	72.762	60.567	$\underline{50.394}$	61.589	73.494	60.573	48.410
Input Injected Model (NumberBatch)	**61.950**	**73.577**	**61.704**	**50.185**	**62.406**	73.235	60.049	**48.337**
Input Injected Model (RGCN)	60.875	71.475	58.658	49.395	62.254	**74.218**	**61.502**	48.202
Architecture Injected Model (NumberBatch)	**61.942**	**72.902**	**61.103**	**50.229**	**62.050**	**73.185**	**59.532**	**48.128**
Architecture Injected Model (RGCN)	53.728	63.607	49.418	43.335	56.069	67.500	52.108	44.137
Output Injected Model (NumberBatch)	61.556	73.134	61.405	**49.823**	$\underline{\mathbf{62.492}}$	$\underline{\mathbf{74.445}}$	$\underline{\mathbf{61.856}}$	$\underline{\mathbf{48.953}}$
Output Injected Model (RGCN)	$\underline{\mathbf{62.098}}$	$\underline{\mathbf{73.829}}$	$\underline{\mathbf{62.336}}$	49.691	61.765	73.857	61.321	48.482
Output Injected Model (RGCN-Learnable)	61.479	73.079	61.411	49.844	62.153	73.813	61.664	48.585

Table 2. Results from injecting a BERT-base model in different settings (Input, Architecture, Output) with NumberBatch and with the graph embeddings from the `train100k` trained RGCN. The overall top scores are $\underline{\text{underlined}}$ and the top scores per injection type (input, architecture, output) are **bolded**.

be a result of the model's hyperparameters being tuned on the `train100k` dataset, then being used for training on each of the following datasets. It is possible that a larger model is needed to encode the additional information found in the larger datasets.

The GCN outperforms the single-relation RGCN on accuracy and AUC, and the single-relation RGCN achieves higher performance on precision and recall. From 3, 4, and 5, we see that the discrepancies between the two models is insignificant in accuracy, but are significant in the other metrics.

5.2.2 RGCN

In comparison to the baseline models, the RGCN shows significant improvement in performance across all metrics except for precision when comparing the RGCN and single-relation RGCN trained on `train100k`, as seen by the p-values in 3. Evidently, the ability to leverage the distinct relation types allows the RGCN to share more features along the additional edges connecting neighboring entities, and produces more effective representations of entities for the link prediction task.

Additionally, we see that the metrics experience a decrease across each of the datasets (`train100k`, `train300k`, `train600k`). Similarly to the single-relation RGCN, this behavior may be due to the fact that the hyperparameters (e.g. learning rate, regularization coefficient, number of hidden dimensions, etc.) were specifically tuned for the RGCN trained on the smallest `train100k` dataset and extended to the RGCNs trained on the larger datasets, while the RGCN may require higher complexity to encode the additional information from the larger datasets.

5.2.3 Injection Models

In the results of the injection models, we can see various notable things. The first of these is that, overall, any of the models that are injected (whether through NumberBatch or the RGCN graph embeddings, and whether they are at input/architecture/output) perform better than the baseline model. This would mean that by just adding in additional information in the naive way that we did, the performance on downstream tasks can be improved.

Secondly, it seems that the input injection models tend to perform better in the validation set, whereas the output injections perform better in the testing set. In both cases they perform better than the baseline model for most metrics. Although unconfirmed, we have a suspicion that the validation set is harder or more out of distribution than the testing set, given that the validation set is a subset of ConceptNet triples that are labelled with lower confidence values than those of the test set triples. With this in mind and considering that we have the same method to infuse the information but at different points of the model, one explanation for this is that since we are adding the signal at the beginning of the model, it will have the most effect and possibly generalization capability. This is because it will affect every layer of the model rather than the internal layers or the output layers. On the other hand, the output models performing better on the test set, if it is more similar in distribution, may mean that these models tend to do better/faster on similar distribution tasks, and would not generalize well to out of distribution tasks.

Thirdly, it appears that the NumberBatch injections tend to improve the base model's performance more than the RGCN injections do. This makes sense given that the NumberBatch embeddings are trained on all of ConceptNet (1M+ assertions) as compared to our RGCN, which is trained on 100k assertions.

Fourthly, it seems that the Architecture injections underperformed in comparison to the Input and Output injections, and even the base model. One possible explanation for this is that we are introducing features at every part of the model, which may be useless or counterproductive at the layer(s) that they are inserted in. In future work, it would be interesting to explore injection at different layers and determine at which layers are the best to inject information.

Lastly, it seems that the Architecture Injected model that utilizes the RGCN embeddings has considerably lower performance. This could be due to a bug in the implementation, as the other ones seem to be on par with each other.

We would also like to note that performance does not decrease too drastically when unfreezing and utilizing the RGCN model to generate embeddings. In future work, it would be useful to explore adding in additional losses to combine both the downstream task model and the RGCN in training to possibly attain greater performance.

6. Future Work: Downstream Tasks

In our work, we used our combined model to perform related question retrieval. We recognize that this may not be the most commonsensical (or related to our knowledge injection) task to test our model in. One other downstream application to further measure our model performance may be testing on the CommonsenseQA dataset, as done in Lin et al. (2019) [Talmor et al. (2019)]. The dataset of question-answer pairs was written by crowd-workers and based on concept relationships found in ConceptNet. In this task, the model is presented with a question along with three distinct answer choices. Two answer choices can be added to increase difficulty of the task. Another downstream task for evaluating our methodology is the SWAG dataset created by Zellers et al. (2018). This dataset consists of multiple-choice commonsense questions created through Adversarial Filtering, which de-biases the dataset through iterative replacement of "easy" samples. Both of these downstream tasks involve intense use of commonsense knowledge such as those found in ConceptNet and in ATOMIC, possibly more so than our question retrieval task.

7. Conclusion

This paper examined a novel method to integrate commonsense knowledge into language models, harnessing both the intuitiveness encoded by knowledge graphs and the power of transformer-based language models. In comparison to existing commonsense inference methods, our methodology uses RGCNs. RGCNs can exploit information stored in local graph neighborhoods and extend to large-scale relational data, such as ConceptNet.

We introduced three injection methods: input injection, architecture injection, and output injection. Through our work, we show that: (1) the earlier the injection of RGCN embeddings, the more effective the injection at improving the BERT model's performance on the MRR and precision @k metrics; (2) there is evidence that the combination of structured graph information and large language models **does** benefit downstream tasks; (3) it is relatively simple to add commonsense knowledge in the form of graph embeddings.

As we continue to seek ways to enrich artificial intelligence with more meaningful, explainable knowledge that mimics human commonsense, methods that harness the power of knowledge graphs will continue to be crucial and impactful. We demonstrate that these methods are very promising ways to enrich model learning with the information in knowledge graphs, and set a strong foundation for future examination of the methods. With more resources, the experiments could be extended to incorporate the entirety of ConceptNet, which could yield even greater improvement in model performance and create a powerful tool in building intelligent, commonsense-aware models.

Acknowledgements

Thanks to MIT CSAIL and the Airforce Office of Scientific Research and DARPA for their grants to the department, and MIT Media Lab and their sponsors. Their contributions and support made this research possible.

References

Colon-Hernandez, P., Havasi, C., Alonso, J., Huggins, M., & Breazeal, C. (2021). Combining pre-trained language models and structured knowledge. *arXiv preprint arXiv:2101.12294*.

Da, J., & Kasai, J. (2019). Cracking the contextual commonsense code: Understanding commonsense reasoning aptitude of deep contextual representations. *Proceedings of the First Workshop on Commonsense Inference in Natural Language Processing* (pp. 1–12). Hong Kong, China: Association for Computational Linguistics. From `https://aclanthology.org/D19-6001`.

Devlin, J., Chang, M.-W., Lee, K., & Toutanova, K. (2019). Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*.

Honnibal, M., & Montani, I. (2017). spaCy 2: Natural language understanding with Bloom embeddings, convolutional neural networks and incremental parsing. To appear.

Kejriwal, M., & Shen, K. (2020). Do fine-tuned commonsense language models really generalize? *CoRR*, *abs/2011.09159*. From `https://arxiv.org/abs/2011.09159`.

Kipf, T. N., & Welling, M. (2016). Semi-supervised classification with graph convolutional networks. *CoRR*, *abs/1609.02907*. From `http://arxiv.org/abs/1609.02907`.

Lai, G., Xie, Q., Liu, H., Yang, Y., & Hovy, E. (2017). Race: Large-scale reading comprehension dataset from examinations. *arXiv preprint arXiv:1704.04683*.

Lei, T., Joshi, H., Barzilay, R., Jaakkola, T., Tymoshenko, K., Moschitti, A., & Marquez, L. (2015). Semi-supervised question retrieval with gated convolutions. *arXiv preprint arXiv:1512.05726*.

Li, D., Yi, M., & He, Y. (2022). LP-BERT: multi-task pre-training knowledge graph BERT for link prediction. *CoRR*, *abs/2201.04843*. From `https://arxiv.org/abs/2201.04843`.

Li, X., Taheri, A., Tu, L., & Gimpel, K. (2016). Commonsense knowledge base completion. *Proc. of ACL*.

Lin, B. Y., Chen, X., Chen, J., & Ren, X. (2019). Kagnet: Knowledge-aware graph networks for commonsense reasoning. *CoRR*, *abs/1909.02151*. From `http://arxiv.org/abs/1909.02151`.

Liu, H., & Singh, P. (2004). Conceptnet—a practical commonsense reasoning tool-kit. *BT technology journal*, *22*, 211–226.

Ma, K., Ilievski, F., Francis, J., Ozaki, S., Nyberg, E., & Oltramari, A. (2021). Exploring strategies for generalizable commonsense reasoning with pre-trained models. *CoRR*, *abs/2109.02837*. From `https://arxiv.org/abs/2109.02837`.

Ostermann, S., Roth, M., & Pinkal, M. (2019). MCScript2.0: A machine comprehension corpus focused on script events and participants. *Proceedings of the Eighth Joint Conference on Lexical and Computational Semantics (*SEM 2019)* (pp. 103–117). Minneapolis, Minnesota: Association for Computational Linguistics. From `https://aclanthology.org/S19-1012`.

Peters, M. E., Neumann, M., IV, R. L. L., Schwartz, R., Joshi, V., Singh, S., & Smith, N. A. (2019). Knowledge enhanced contextual word representations. *CoRR*, *abs/1909.04164*. From `http://arxiv.org/abs/1909.04164`.

Petroni, F., Rocktäschel, T., Lewis, P., Bakhtin, A., Wu, Y., Miller, A. H., & Riedel, S. (2019). Language models as knowledge bases? *arXiv preprint arXiv:1909.01066*.

dos Santos, C., Barbosa, L., Bogdanova, D., & Zadrozny, B. (2015). Learning hybrid representations to retrieve semantically equivalent questions. *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 2: Short Papers)* (pp. 694–699). Beijing, China: Association for Computational Linguistics. From `https://www.aclweb.org/anthology/P15-2114`.

Sap, M., Le Bras, R., Allaway, E., Bhagavatula, C., Lourie, N., Rashkin, H., Roof, B., Smith, N. A., & Choi, Y. (2019). Atomic: An atlas of machine commonsense for if-then reasoning. *Proceedings of the AAAI conference on artificial intelligence* (pp. 3027–3035).

Schlichtkrull, M., Kipf, T. N., Bloem, P., van den Berg, R., Titov, I., & Welling, M. (2017). Modeling relational data with graph convolutional networks. *arXiv preprint arXiv:1703.06103*.

Speer, R., Chin, J., & Havasi, C. (2018). Conceptnet 5.5: An open multilingual graph of general knowledge. *arXiv preprint arXiv: 1612.03975*.

Talmor, A., Herzig, J., Lourie, N., & Berant, J. (2019). Commonsenseqa: A question answering challenge targeting commonsense knowledge. *ArXiv*, *abs/1811.00937*.

Thanapalasingam, T., van Berkel, L., Bloem, P., & Groth, P. (2021). Relational graph convolutional networks: A closer look. *CoRR*, *abs/2107.10015*. From `https://arxiv.org/abs/2107.10015`.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L., & Polosukhin, I. (2017). Attention is all you need. *arXiv preprint arXiv:1706.03762*.

Wang, M., et al. (2019). Deep graph library: A graph-centric, highly-performant package for graph neural networks. *arXiv preprint arXiv:1909.01315*.

Zellers, R., Bisk, Y., Schwartz, R., & Choi, Y. (2018). Swag: A large-scale adversarial dataset for grounded commonsense inference. *EMNLP*.

Zhou, W., Lee, D., Selvam, R. K., Lee, S., Lin, B. Y., & Ren, X. (2020). Pre-training text-to-text transformers for concept-centric common sense. *CoRR*, *abs/2011.07956*. From `https://arxiv.org/abs/2011.07956`.

Appendix A. Significance Testing

In order to quantitatively compare the models and validate the observed results, we perform significance testing. Within each set of models trained on the same dataset (`train100k`, `train300k`, or `train600k`), we test between each pair of models: Single-RGCN vs. GCN, RGCN vs. GCN, and RGCN vs. Single-RGCN. The significance test of model A vs. model B measures the confidence in, or significance of, the difference in performances of the two models.

The results of our significance tests are found in Appendix 3, Appendix 4, and Appendix 5.

	p-value		
Metric	Single-RGCN vs. GCN	RGCN vs. GCN	RGCN vs. Single-RGCN
Accuracy	0.099	**0.000014**	**0.00015**
AUC	**0.0080**	**0.0000078**	**0.00050**
Precision	**0.00024**	**0.000015**	0.14
Recall	**0.027**	**0.000073**	**0.00013**

Table 3. The results of significance testing between models trained on `train100k`. The significant p-values are **bolded**.

	p-value		
Metric	Single-RGCN vs. GCN	RGCN vs. GCN	RGCN vs. Single-RGCN
Accuracy	**0.0013**	**0.035**	**0.0024**
AUC	**0.0091**	**0.00092**	**0.0035**
Precision	**0.000074**	**0.0000032**	**0.041**
Recall	**0.0049**	**0.000000058**	**0.00012**

Table 4. The results of significance testing between models trained on `train300k`. The significant p-values are **bolded**.

	p-value
Metric	Single-RGCN vs. GCN
Accuracy	**0.000025**
AUC	**0.0015**
Precision	**0.000000019**
Recall	**0.00020**

Table 5. The results of significance testing between models trained on `train600k`. The significant p-values are **bolded**.