Exploring the Impacts of Architecture and Scale on GNN Performance on Relational Data

By: Joseph Guman, Atindra Jha, and Christopher Pondoc

Milestone Overview

We set up the infrastructure for our milestone to help users experiment with different tasks, embedding models, and architectures for Relbench and Relational Deep Learning (RDL). The general skeleton of the tutorial is there, and we'd love any feedback on how to make our explanations more transparent and enable users to explore even more dimensions. The code is open-source here.

Contribution: Addressing Tutorial Errors

The first step of our tutorial was learning more about Relbench and utilizing the tutorials the team had open-sourced. While we were able to get the tutorial about loading data to work, the tutorial for training a modal on Colab end-to-end unfortunately did not work, as we kept getting errors around PyG (in particular, an error about how `'NeighborSampler' requires either 'pyg-lib' or 'torch-sparse'`). We resolved this error locally by following some of the debugging threads located across GitHub, and we also shared a working Colab. We raised an issue and are working on a PR within the Relbench repository to encourage further discussion.

Decomposing Ablations

Once we had the tutorial up and running, we abstracted away the code to helper files and focused on setting up the code for running different ablations. In particular, our opinionated view on our project is to **have the tutorial focus on the core tradeoffs in Relational Deep Learning while abstracting away all the nitty-gritty details as helper code.** We deep dive into our experiments and initial results below.

Note about the Dataset

A note about the dataset: we decided to stick with `rel-f1` for the tutorial, as opposed to using another dataset such as `rel-amazon`. This is primarily because this dataset is small enough to load and run experiments on in Colab, making it more accessible.

Transfer Learning and Fine-Tuning

Zero-Shot Learning on Graph Tasks

The first question we wanted our tutorial to answer was whether or not training a model on a singular task would allow it to generalize to another similar but different task. We also explored how finetuning might affect downstream model performance on different tasks. As mentioned above, we focused on the `rel-f1` dataset and entity classification tasks.

To start, we trained an RDL model on the `driver-dnf` task. This entity classification task focuses on predicting whether a driver will not finish a race in the next 1 month. We set up the code to load the data, train the model, and evaluate it on the test data split. The primary change we emphasized was the switch in the loss function and metric to look for: in particular, given that the original tutorial focused on link prediction, we switched the loss function to binary cross-entropy loss and the target metric to AUROC. We describe both within our tutorial.

We could roughly match the results on the Relbench leaderboard by training a model from scratch. However, after evaluating the model that we trained on the `driver-dnf` task, zero-shot, on another entity classification task – `driver-top3`, which predicts if a driver will qualify in the top 3 of a race in the next month – the results were nowhere near what was reported by the Relbench team for this task, dropping accuracy by over 40%, from a peak test accuracy of 0.71 on the `driver-dnf` task to 0.29 on the `driver-top3` task. This makes sense, as the tasks are somewhat opposite. If `driver-dnf` is `true` for a driver, it becomes significantly less likely that they will finish in the top 3. As such, this suggests the model is learning a representation that generalizes from just “DNF” results to representing a driver’s general skill level. Since the model was trained to predict the former, a dramatic fall in accuracy on the `driver-top3` task aligned with what we expected.

Finetuning the Pre-Trained GNN for a Different Downstream Task

From there, we took the model trained on the `driver-dnf` task and finetuned it on the `driver-top3` task. Even after only training for around five epochs – half the number of epochs of the pre-trained model – we saw performance metrics improving drastically and roughly matching the results reported on the RelBench leaderboard.

When compared to training a new model from scratch on the `driver-top3` task directly, we saw that the results were comparable (best test accuracy of 0.82 for the finetuned model vs. 0.792 for the one trained from scratch). **Figure 1** below illustrates the best test accuracy we observed for our various experiments.

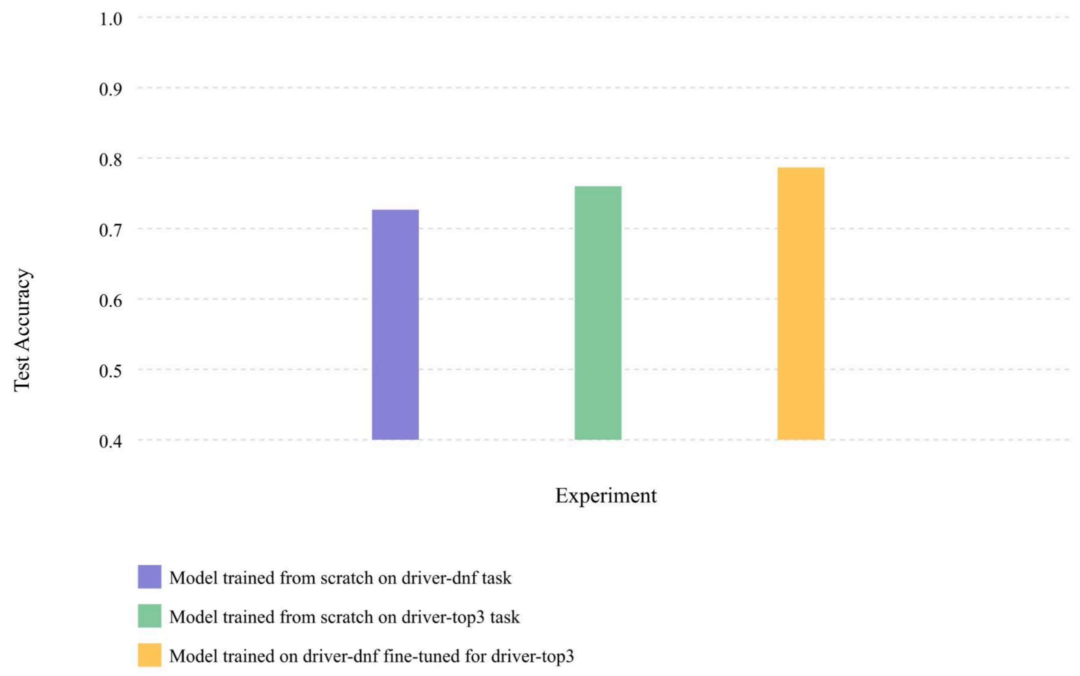

Figure 1: Best Test Accuracy for various tasks

Of course, this suggests that the initialization of weights is not the most critical in these tasks and that transfer learning may be employed in a more compute-constrained environment. However, we present the challenge to the user of trying it out on more tasks and datasets to see if this hypothesis generalizes empirically.

Adjusting the Embedding Model for Node Features

As part of the RDL pipeline, the authors propose using a text embedding model to encode row-level data into initial node embeddings. One of the ablations performed in the original Relbench paper is experimenting with GloVe embeddings and more state-of-the-art BERT-style embeddings.

In traditional NLP, BERT embeddings are much more popular given that they are contextual -- the vector representation depends on the surrounding words, compared to static embeddings used by GloVe -- and can handle words outside of their vocabulary. In addition, their embedding size is 768 compared to GloVe's 300, which introduces an opportunity for more expressiveness.

Our team implemented this initial experiment and found comparable results when training on the `driver-dnf` task with both GloVe and BERT-style embeddings. Of course, this could be attributed to the fact that the models are close in size and perform similarly on general embedding benchmarks like MTEB. Thus, we also implement an interface to help users experiment with other embedding models from HuggingFace. We provide the interface below:

```
Unset
from src.embeddings.custom import CustomTextEmbedding
text_embedder_cfg = TextEmbedderConfig(

text_embedder=CustomTextEmbedding(model_name=<INSERT_HUGGINGFACE_MODEL_HERE>,
device=device), batch_size=128
)
```

Varying Architectures within RDL

Recall that the core RDL pipeline is as follows:

- **Heterogeneous temporal graph**: Each table represents a node type, each row represents a node, and primary-foreign-key relationships represent an edge.
- **Deep learning model**: Deep tabular models encode row-level data into initial node embeddings. We then use a GNN to update node embeddings.
- **Temporal-aware subgraph sampling**: Sample a subgraph around each entity node at a given seed time (to ensure no information leakage). The subgraph is then inputted to GNN to predict the target label.
- **Task-specific prediction head + loss:** Apply an MLP on the entity embedding to make a binary cross-entropy loss prediction.

For the last part of our tutorial, we also led the user in implementing variants of the GNN architecture within RDL. We specifically focused on adding and subtracting GNN layers from our pipeline. First, we doubled the number of GNN layers in our RDL pipeline, moving from 2 to 4. We hypothesized that adding more layers would create a more expressive network that could understand more complex relationships. However, we found that training using double the number of layers made the model perform worse over time. Given the task's simplicity and the dataset's size, it was likely we were overfitting.

Given that doubling the number of layers led to less optimal results, we also tried the opposite strategy and halved the number of layers in the network. Interestingly, we saw that even with halving the number of layers to 1, we do the same as we did with two layers. Once again, this might be more task-specific than a general conclusion about GNNs and the RDL pipeline.

Future Work

Our progress has allowed us to have a fully functional tutorial that learners can use to understand Relbench and RDL's nuances better. Looking forward, we have many areas to extend our work – both coming from our original proposal and new ideas – but we want to ensure that we keep the core insights/flow of the tutorial intact. Some ideas include:

- **Implementing more architectures** – as an extension to adding and removing layers, we will also finish our GCN and GAT implementations to replace GraphSage.
- **Including more baselines** – in our original proposal, we loft the idea of using tabular models or a LightGBM classifier to compare against RDL. While these examples would undoubtedly motivate the utilization of RDL better, we also aren't sure if it's better to focus on improving RDL versus acknowledging that it is better than older methods. We also believe that focusing on the content we have right now will lead to better adoption of Relbench by showcasing its flexibility and power as a package and benchmark.
- **Larger + Bigger Datasets** – we also want to support exploring these trends across other datasets, assuming learners have enough access to compute. We've had some recent issues with getting manageable compute, but looking to see whether these trends generalize to other datasets and tasks would be pretty relevant.