QNLP: Quantum Natural Language Processing

Blake A. Wilson

February 16, 2024

Abstract

Quantum computers have recently emerged as experimental platforms for natural language processing tasks due to the mapping between the high-dimensional tensor representations of Categorical Compositional Distributional (DisCoCat) models and quantum states. Packages like *lambeq* make implementing DisCoCat for natural language tasks on quantum hardware approachable. In this work, we provide preliminary results for three string diagram generators for DisCoCat and perform an ablation study to test the performance of models trained on static tree readers and random tree readers implemented in lambeq.

1 Introduction

Natural language processing is composed of several subfields, including text classification. For example, let's consider the following sentence about cooking from the meaning classification (MC) dataset:

"skillful man prepares sauce ."

For this sentence, we want to generate a model that correctly classifies it as a cooking sentence. All the sentences $\{\mathbf{x}_i\}^n$ in the MC dataset are either about programming $\mathbf{y}_i = 0$ or cooking $\mathbf{y}_i = 1$, indicated by the class variable $\mathbf{y}_i$. To classify sentences, we use the Categorical Compositional Distributional (DisCoCat) framework. DisCoCat uses category theory as a

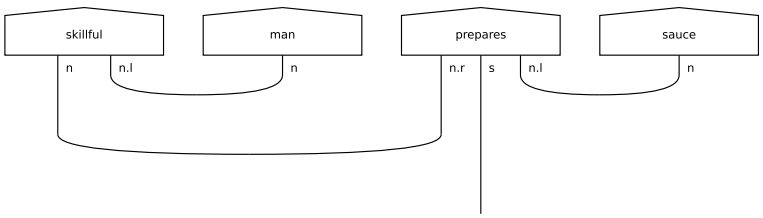


Figure 1: **A string diagram generated by the *Bobcat* class for the sentence "skillful man prepares sauce ."**

basis for defining semantics in sentences. Once the semantics have been determined by a *string diagram* using a pregroup grammar, linear maps are applied to tensor embeddings to produce vector representations of the meaning of phrases. The framework begins as abstract mappings but quickly becomes practical through these linear mappings. In this work, we use *lambeq* for generating string diagrams given pregroup grammars and training the linear maps for classification. To expand the capabilities of *lambeq*, we consider its *TreeReader* class which constructs deterministic trees of strings diagrams. We were unsuccessful in using *TreeReader* for generating *trainable* string diagrams, so we implement two custom string diagram generators known as *RandomTreeParser* and *DeterministicTreeReader* which generates random and deterministic tree-based string diagrams, respectively. We then perform an ablation study using both of these string diagram generators.

1.1 Background

We begin by constructing an abstract representation of sentences using string diagrams. Each string diagram is a tree of the relationships between words in a sentence for performing tensor operations that will perform classification. To create a string diagram, we first tokenize each word in the sentence "skillful man prepares sauce ." into a set

$$\{\text{“skillful”}, \text{“man”}, \text{“prepares”}, \text{“sauce”}, \text{“.”}\}$$

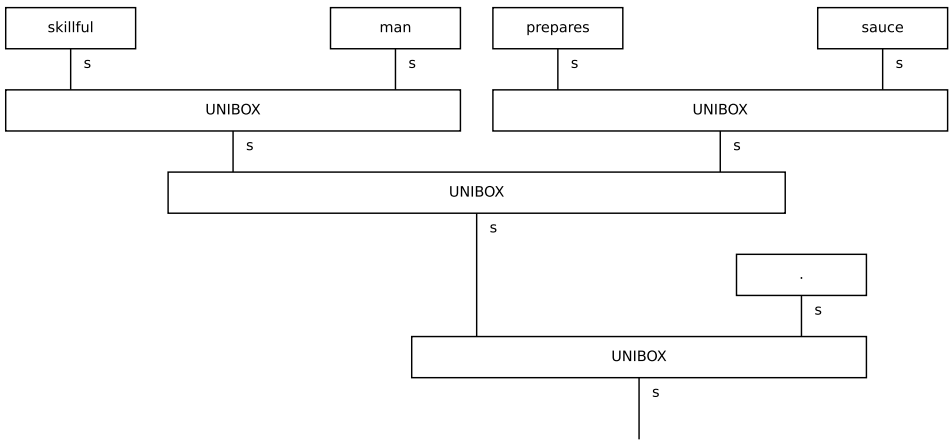


Figure 2: **TreeReader example diagram**

by using the whitespace as a delimiter. Then, we use a *pregroup grammar* for each word to give the sentence semantic meaning and construct the nodes and edges. A pregroup grammar is a set of symbolic rules that define a string diagram for a sentence. We begin by considering an atomic type $\mathbf{n}$ that has a left $\mathbf{n}^l$ and a right $\mathbf{n}^r$ adjoint such that the following rules apply

$$\mathbf{n}^l \cdot \mathbf{n} \to 1 \to \mathbf{n} \cdot \mathbf{n}^l$$

$$\mathbf{n} \cdot \mathbf{n}^r \to 1 \to \mathbf{n}^r \cdot \mathbf{n}$$

where $\cdot$ is a tensor contraction. For example, the verb "prepares" takes in a noun phrase "skillful man" on the left and an object noun "sauce" on the right to generate a sentence. The verb "prepares" will have a tensor embedding representation defined by the pregroup grammar $\mathbf{n}^l \otimes \mathbf{n} \otimes \mathbf{n}^r$. When the verb acts on a noun phrase given by the tensor product $\mathbf{n}^l \otimes \mathbf{n}^r$, the verb performs a tensor contraction with a weighted tensor embedding to produce an output noun phrase of type $\mathbf{n}$.

We use parsers to generate string diagrams, such as the *Bobcat* parser in the *lambeq* package which produced the string diagram in Figure 1. Alternatively, we can use trees for representing string diagrams.

1.2 *TreeReader*

The *TreeReader* class in the *lambeq* package is an alternative way of generating string diagrams using box representations. For example, the tree generated by the *TreeReader* in Figure 2 is not the same as the tree in Figure 1 generated by *Bobcat* because *Bobcat* uses a more informed pregroup grammar for specific words. However, one could construct an equivalent tree using the TreeReader. While the documentation for generating trees using *TreeReader* to generate string diagrams was plentiful, I had a difficult time finding how to use a diagram generated by *TreeReader* for training a model.

1.2.1 Problems with *TreeReader* and *stairs*

Using the string diagrams generated by the *TreeReader* and *stairs* classes generate the following error:

```
The arrow is ill-defined.
Applying the functor to a box returns dom = Ty(),
cod = s expected dom = s @ s, cod = s
```

To debug this error, I first reverted to a previous version of *lambeq* and used the old backend for training the model. However, I ran into a similar implementation error. It's possible I missed a simple way of implementing *TreeReader*, but I digged through documentation for a while to no avail. The second thing I tried was to test implementations of the *stairs_reader*, *spiders_reader*, and *cups_reader* classes. I found that the *cups_reader* works as expected despite using boxes, but the *stairs_reader* breaks similarly to the *TreeReader*. Digging into when the dom changes, as indicated by the error, I found that the *then()* call in the __rshift__ operator is overwriting the dom but due to time constraints I decided to pursue the more familiar direction of generating custom string diagrams using cups. I will now demonstrate my contributions and present more hurdles to overcome for future work.

2 Methods

In this section, I will describe all three readers I've implemented and the training process for the models.

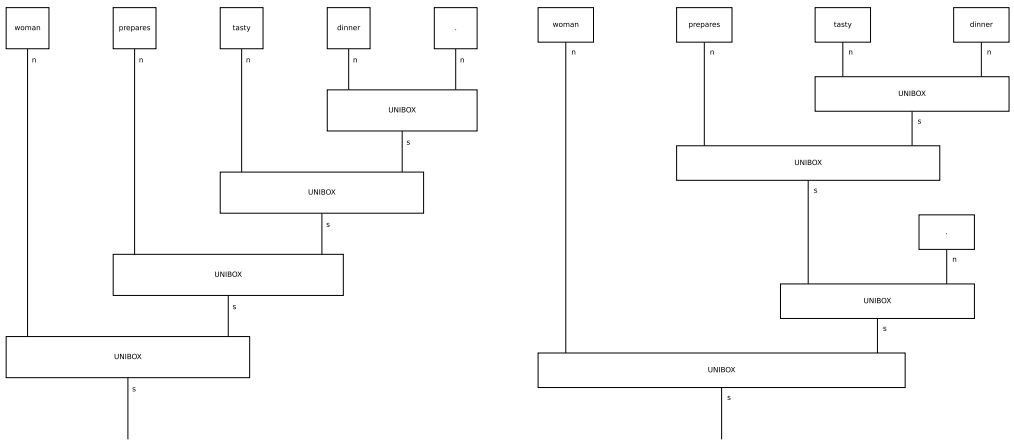


Figure 3: *RandomTreeReader* generates different trees for the same sentence "mary prepares tasty dinner ."

2.1 RandomTreeReader

I first begin by implementing a random tree reader similar to the *TreeReader* in the *lambeq* package. The *RandomTreeReader* first tokenizes a sentence using the whitespace delimiter. Then, it builds the tree nodes by using a post-order traversal with a random midpoint in the list of tokens to separate the tokens into a left and a right subtree. I demonstrate the class in the following figure; both trees use the *RandomTreeReader* class on the same sentence and generate different trees.

2.2 DeterministicTreeReader

I implemented a *DeterministicTreeReader* to replace *TreeReader*. What makes it different is that the string diagrams generated are more akin to the string diagrams generated by *Bobcat* than *TreeReader* while still maintaining the tree structure of *TreeReader*. This implementation uses custom *TreeNode* classes for building a tree before the string diagram. After generating a list of words, the tree is constructed by passing in cup indeces to the *Diagram.create_pregroup_diagram* function. The *DeterministicTreeReader* class is similar to the *RandomTreeReader* but instead of using a random midpoint,

it uses the deterministic middle point for separating the tree. The midpoint is the middle of the list of tokens. I demonstrate the class in Figure **??**; both trees use the *DeterministicTreeReader* class on the same sentence and generate the same trees.

2.2.1 Problems with *DeterministicTreeReader*

The main problem with *DeterministicTreeReader* is that it doesn't use the *lambeq* backend for generating the trees. Due to time constraints, I could not familiarize myself with the backend to build a true extension of *lambeq*. However, I am confident that given more familiarity, I could add similar classes and improve *TreeReader* to work as expected.

2.3 *RandomTreeParser*

Lastly, I implemented a *RandomTreeParser* as a random reader that generates random trees like *RandomTreeReader* using string diagrams from words and cups similar to *DeterministicTreeReader* and *Bobcat*.

2.3.1 Problems with *RandomTreeParser*

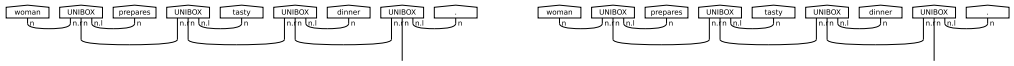
The main problem with *RandomTreeParser* is that the cups passed to *Diagram.create_pregroup_diagram* are reordered. Due to time constraints I could not fix this error. We can see in Figure 4 that the cups are randomly configured, though valid, but generate the same string diagram.

2.4 Training the Models

Despite the issues discussed with the readers, it is possible to train models using the readers. The training loss function is the Binary Cross Entropy loss function to train the model,

$$\mathcal{L}(\hat{\mathbf{y}}, \mathbf{y}) = -\frac{1}{N}\sum_{i=1}^{N}\left[\mathbf{y}_i \log(\hat{\mathbf{y}}_i) + (1-\mathbf{y}_i)\log(1-\hat{\mathbf{y}}_i)\right], \tag{1}$$

which is minimized using the Adam optimizer. The model is trained on the MC dataset and validated on a validation subset. The model is trained for 50 epochs using a batch size of 30, learning rate of $3 \times 10^{-}2$, and a random integer seed between 0 and 100.



(a) Cup indeces given as: [[0, 1], [3, 10], [4, 5], [7, 8], [6, 9], [11, 12], [2, 13], [15, 16]]

(b) Cup indeces given as: [[0, 1], [3, 4], [2, 5], [7, 10], [8, 9], [11, 12], [6, 13], [15, 16]]

Figure 4: *RandomTreeParser* generates the same string diagrams for difference cup indeces on the sentence "mary prepares tasty dinner ."

Figure 5: Training results for the models trained on the *DeterministicTreeReader* and *RandomTreeParser* readers.

3 Results

The training results are provided for two models trained on both the *DeterministicTreeReader* and *RandomTreeParser* readers.

Figure 5 shows the training results for the models. The models trained on the *DeterministicTreeReader* and *RandomTreeParser* readers perform similarly. The models have very similar performance, likely due to the reodering of the string diagrams mentioned in the issues section for the *RandomTreeParser* reader. Both models quickly converge to 100% accuracy within 20 epochs on the training sets. Both also achieve around 84% accuracy on the final validation set.

4 Discussion

The original goal of this work was to demonstrate deterministic and random *TreeReader* classes for generating string diagrams for sentence classification. However, due to issues outlined in the background section, generate the string diagram using the *TreeReader* leads to errors during training. Therefore, three alternative readers are demonstrated, two of which are compatible with model training. Given more time, I would like to collaborate with the engineers who developed *lambeq* to learn how to faithfully implement *TreeReader* for training, properly define random cup index assignments.

Overall, I spent approximately 20 hours on this project. I spent the most time on debugging my implementations of the string diagrams, learning about DisCoCat, and writing this report.

5 Running the Code

The code for this project is available in the *qnlp/* package directory. To run the code, first install the package in a virtual environment using the following commands:

```
python -m venv .env
source .env/bin/activate
python -m pip install -e .
```

Then, the following script can be run to train the models:

```
python test/testTorch.py
```

To plot the training results:

```
python test/testPlot.py
```

To reproduce the errors mentioned in this work:

```
python test/testErrors.py
```

To generate/plot the string diagrams:

```
python test/testReaders.py
```