Politecnico di Milano

Master's Degree in Computer Science

Software Engineering 2 Project:

Quantum Natural Language Processing for dummies: The first approach to Lambeq's library

Student: Stefano Vighini

ID: 10622788

Academic year: 2021-2022

Contents

List of Acronyms

QC Quantum Computing

NLP Natural Language Processing

QNLP Quantum Natural Language Processing

NISQ Noise Intermidiate Scale Quantum

Introduction

Quantum Computing has raised more and more attention during the past years, due to the development of NISQ quantum computers and their availability in the cloud.

Researchers tried to implement new quantum algorithms for classical problems and compare their performances with respect to classical algorithms. One of these fields is Natural Language Processing (NLP), which will be the focus of this document. This branch of science studies how words and sentences are connected together. Through Machine Learning, we try to solve classification problems (e.g. the membership of a word to a certain context), speech to text, generative algorithms and much more. A new library, Lambeq, has been published on GitHub in October 2021. This package can be used to create high level QNLP algorithms. An example of its usage is Quanthoven, a quantum algorithm which is capable of creating "new" compositions after being trained on some musical tracks.

In order to understand better how QNLP and it's brand new library Lambeq work, we need to review the basics first. I will discuss only what will be strictly necessary to understand the topic, in order to be the simplest i can. For this reason i will not go in depth on QC or other difficult topics that are not the main focus of this project.

Theory

3.1 Quantum Computing

Quantum computers have a new type of architecture that uses qubits instead of bits. The main difference is that they don't have an exact state (that for classical bits is 0 or 1), but they are in a *superposition*, defined by a probability of having one of the two values. When the measure is made, then one exact value will be associated with the bit.
Physically, qubits are superconductors brought at the temperature of almost 0 Kelvin (0.012 in the case of D-Wave quantum computer) that show properties related to quantum particles. The most important one is *entanglement*, that is, roughly speaking, the connection between each other. A change to one qubit may affect also other qubits. This is the core reason why QC is a step ahead of classical computing in the scope of NLP: beacuse the entanglement between qubits is linked to the one of words in natural language. Techically, NLP is **quantum native**.

There are two main approaches to QC: the annealing model and the gate model. In QNLP the gate model is used: it consists of a series of boxes (gates) that take a certain number N of qubits in input, affects their superposition and then gives the same number of qubits as output. Figure 3.1 represents a gate model.

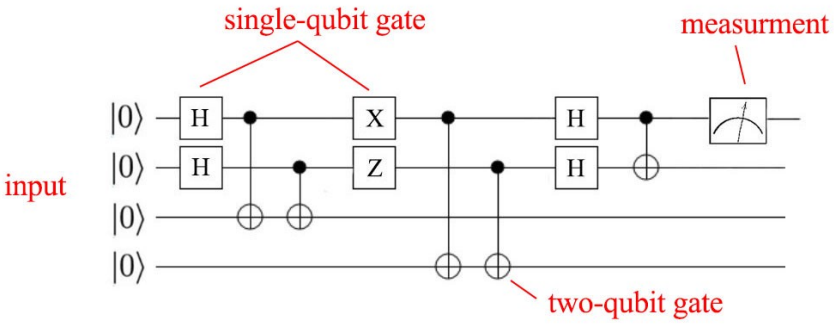


Figure 3.1: Gate model example

3.2 Classical Natural Language Processing

NLP is the branch of science studying how to make a computer interpret human texts (or speeches) and give them meaning in order to be useful for certain tasks. Its applications are numerous: spam detection, machine translations, chat bots, sentiment analysis, text summarization, and so on.

There are mainly 2 approaches to NLP: the *grammatical* approach, who aims at analysing the structure of the sentence; the *bag of words*, which doesn't care about the syntax and grammar, but analyses statistically the relationship between the words (ex. how many times the word "blue" appeared next to "sky"). The first one seems more precise, but may be too complex to implement because of the variety of language. The second one is more scalable, but loses the 'interpretation' of the sentence. The new approaches that are being developed in QNLP are trying to merge these two aspects.

Pipelines are needed in order to preprocess data, and every approach may differ in its implementation. In order to use the bag of words, the text will go through these steps:

- sentence segmentation: splitting a paragraph into sentences
- word tokenization: splitting a sentence, paragraph or text into smaller entities (called *tokens*)
- parts of speech tagging: assigning the roles to tokens (wether it's a noun, verb, adverb and so on)
- text lemmatization: simplyfing a derived word into it's root (*eating* becomes *eat*, *fearful* becomes *fear*)
- identifying stop words: removing words that don't add meaning to the sentence (such as *do, such, this*)

We aren't going more in depth on this, as we will see directly the approach to QNLP in the following chapters and the pipeline is different.

3.3 Pregroup grammar

Pregroup grammar is a grammar formalism used to reduce a text and check wether it's a correct sentence or not.

3.3.1 Definition

Let L be the set of words, T the set of types, and $:$ the relationship that maps words to types. T may contain (for example) the following types: N, S, N0. N defines a noun that doesn't need to be preceded by an article, N0 is a noun preceded by an article and S is a sentence.

Every type can have *left* (N^l) or *right adjoints* (N^r). These can be reduced as follows: $N^l * N \leq 1$ and $N * N^r \leq 1$

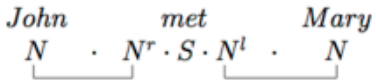
A sentence is considered correct when, after the resolution, there is only an S left.

Words are mapped to a list of types, according to the grammatical structure of the sentence and what other words we expect in the sentence.

3.3.2 An example

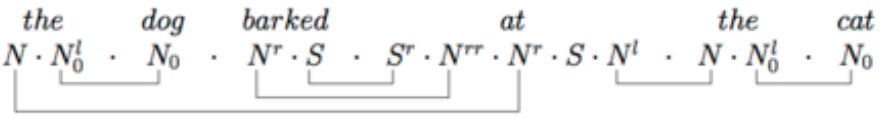
In the sentence: *"John met Mary"*, words are typed in this way.

$$John : N \qquad Mary : N \qquad met : N^r \cdot S \cdot N^l$$



In the sentence: *"The dog barked at the cat"*, words are typed in this way.

$$the : N \cdot N_0^l \qquad dog : N_0 \qquad cat : N_0 \qquad barked : N^r \cdot S \qquad at : S^r \cdot N^{rr} \cdot N^r \cdot S \cdot N^l$$



3.3.3 Considerations

We can easily spot the logic in there:

- the noun sentence *the dog* is resolved as a N, so it's reduced like it was a name
- transitive verbs have both adjoints, while intransitive ones have only the right adjoint, and they also carry the meaning of the sentence S
- prepositions are a bit more difficult to describe, but it's possible to use a backward approach to understand it: adjoints near S simplify to the nouns on both sides, while $S^r \cdot N^{rr}$ simplify to the verb.

3.3.4 Expressiveness and limits

Pregroup grammar is proven to be almost equivalent to context-free grammars in terms of expressiveness. Some sentences (like in Dutch and Swiss-German) give rise to cross-serial dependencies and are beyond context-freeness. In order to go beyond this limit, Combinatory Categorial Grammar (CCG) can be used instead. I won't talk about it in detail, but the principles are the same as the grammar of this article, with some differences in the way types are written and rules that simplify them. I would like to mention the following difference.

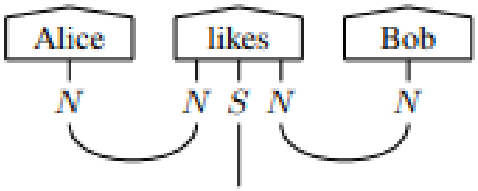
When we talk using informal language, the order of words may not be the proper one, though the sentence is still correct (there is not only one ordering). This can create problems when the types of tokens cannot be simplified. Pregroup grammar is not able to address this problem, while with CCG (thanks to different rules) we can express crossing.

3.4 DisCoCat

Up to now, we just saw how to encode sentences in a way in which they can be parsed, but nothing has been done in relation to their actual use. For this purpose, we introduce Categorical Compositional Distributional semantics (DisCoCat), a mapping from a grammar to its semantics, that is from the sentence to its meaning. The grammar that can be chosen may either be *pregroup grammar* or *CCG*. Sentences in DisCoCat are represented in a very similiar way to the ones of the pregroup grammar, but the interpretation is different.

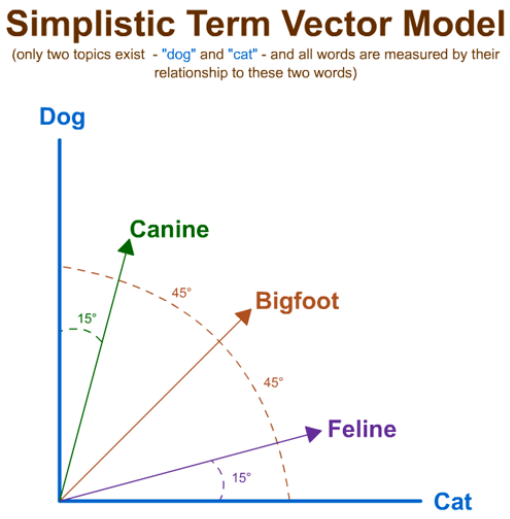
3.4.1 Definition

Let's start from a sentence expressed in pregroup grammar. May we choose "Alice likes Bob", the related DisCoCat *string diagram* is the following:



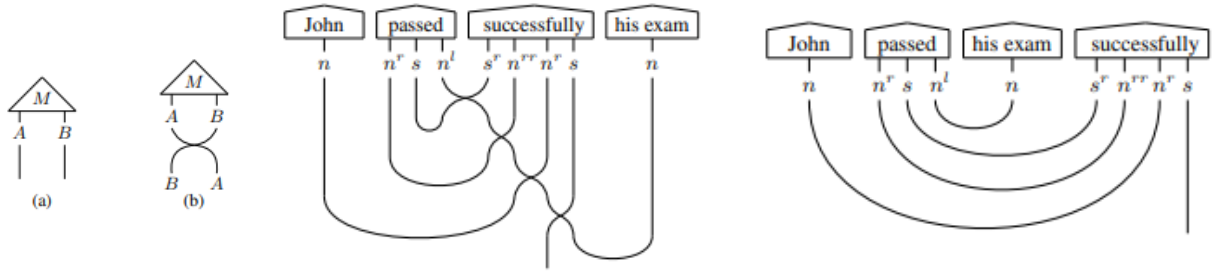
N and S are vector spaces, *Alice* and *Bob* are vectors that belong to N and *likes* is a tensor of order 3, $N \otimes S \otimes N$ (where $\otimes$ is the tensor product). The *cups* in the diagram correspond to *tensor contractions*, so that the vector for the whole sentence lives in S.

The number of dimensions of the vector spaces is chosen arbitrarily, and every dimension represents a word. The direction of the vector in the vector spaces tells us how much the word to examine is related to every dimension.



3.4.2 Swapping

Talking about the order of words in sentences (referring to to chapter 3.3), let's consider the following images:



On the left it's showed how wires swapping is represented. The second and the third image represent how a string diagram with non-canonical ordered words (thus needing swaps in order to simplify types) can be flattened. Flat diagrams are way better also from a computational point of view: in classical computation, swapping means computing a transpose of a matrix, while in quantum computing it implies the use of entangling gates (such as CNOTs).

Lambeq

4.1 Introduction

Lambeq is a QNLP python library implemented by Cambridge Quantum. It puts at disposal high level functions that permits customization of the pipeline used for the natural language processing. You may also choose between a classical trainer or a quantum one.

You can find the documentation here, together with some ipynb examples.
As I tried to replicate the ipynb about quantum trainer, i found some issues. In the next paragraphs I am going to focus on these ones, more than explaining lambeq functioning. Tests have been performed with MM MC dataset, splitting 80 samples for training, 20 for validation and 30 for testing.

4.2 Pipeline

The steps of the pipeline are the following (like pointed in the documentation):

- Sentence input
- Diagram rewriting
- Parameterisation
- Training

4.2.1 Sentence input

In this phase you can create String Diagrams using different parsers. Parsers differ in the way they interpret the string. For example, BobcatParser "implements" Pregroup Grammar and it outputs a diagram mapping words with their type, like we saw in the Theory section.

Unlike BobcatParser, SpiderReader implements the "bag of words" model, not considering the sentence's grammar while parsing.

4.2.2 Parameterisation

In this stage the string diagram is transformed in a lower-level representation, such as tensors or quantum circuit. This is done using ansatzs, that are hypothesis made on the shape of the string

diagrams in order to map wires to quantum states or tensors. The resulting circuit depends by some paramaters specified in the ansatz (such as number of NOUN and SENTENCE) and also by the string diagram itself.

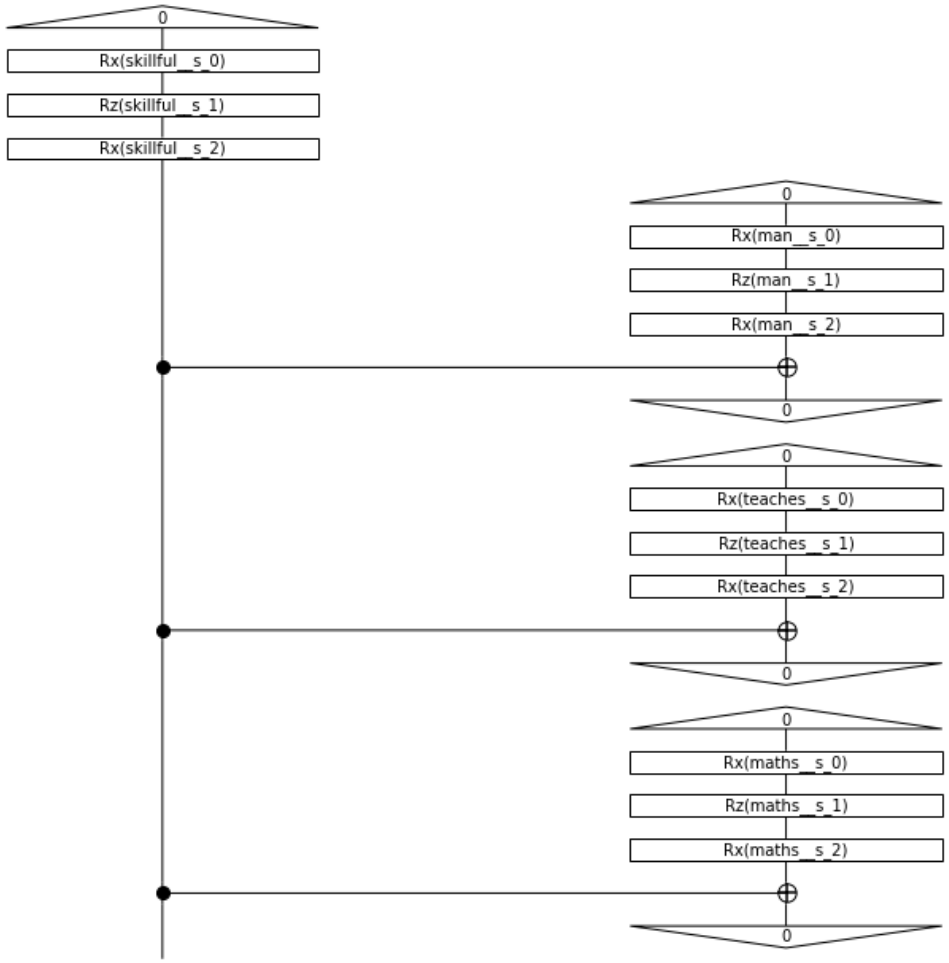


Figure 4.1: Quantum circuit with SpiderReader

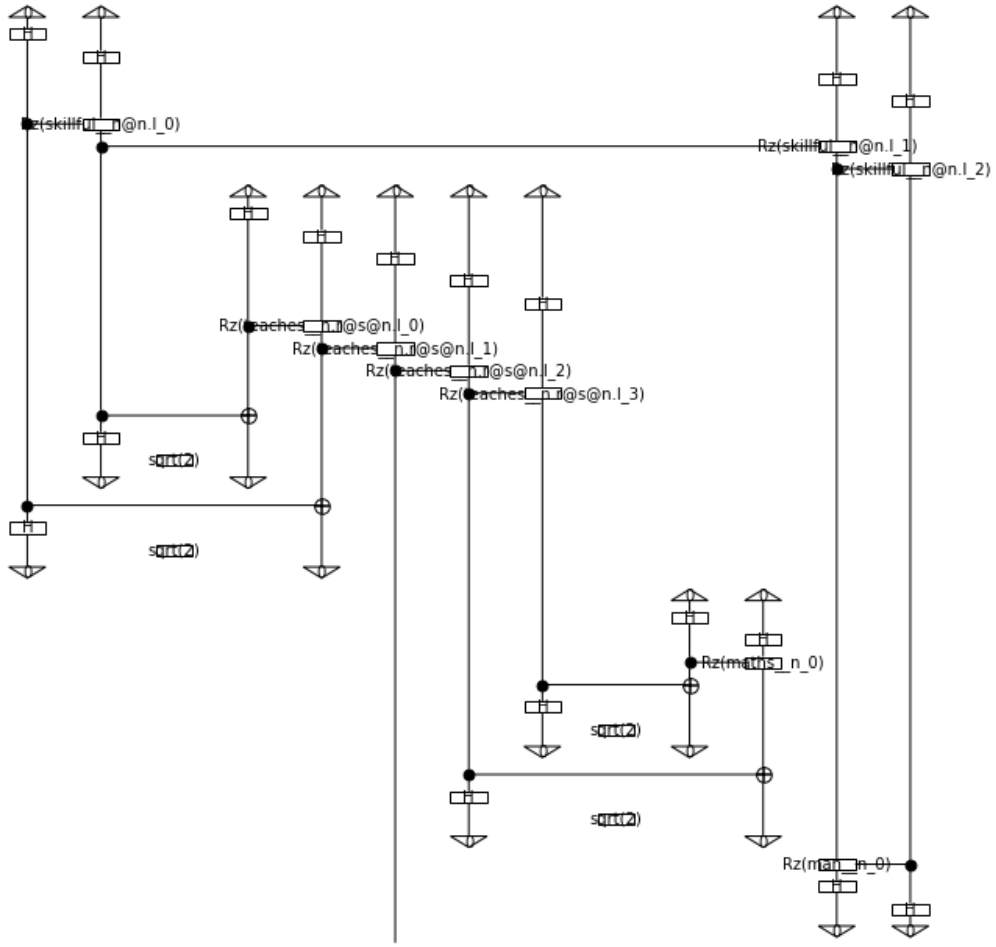


Figure 4.2: Quantum circuit with BobcatParser

4.3 Results

Below there are some runs in the four configuration possible with two chosen parsers (BobcatParser and SpiderReader) and the two types of training (Quantum or Classical).

We can notice some differences in the trainer output that derive from the chosen parser. Using a classical trainer, BobcatParser converges quicklier.

Instead, we can also notice that BobcatParser in QuantumTraining doesn't converge to maximum accurancy, while SpiderReader does. This was also reported by developers.

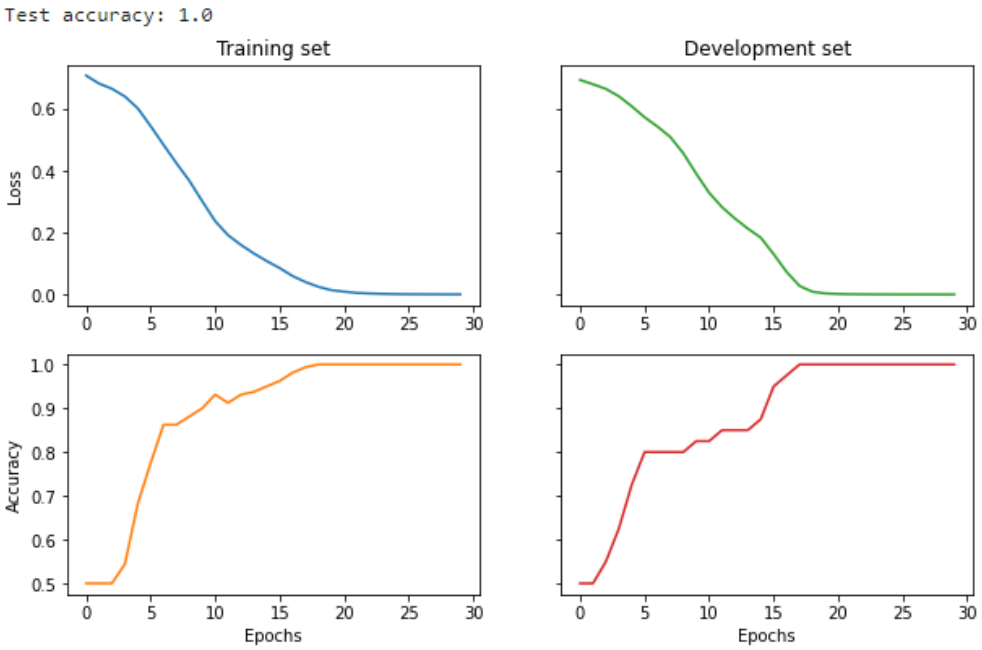


Figure 4.3: BobcatParser with Classical Training

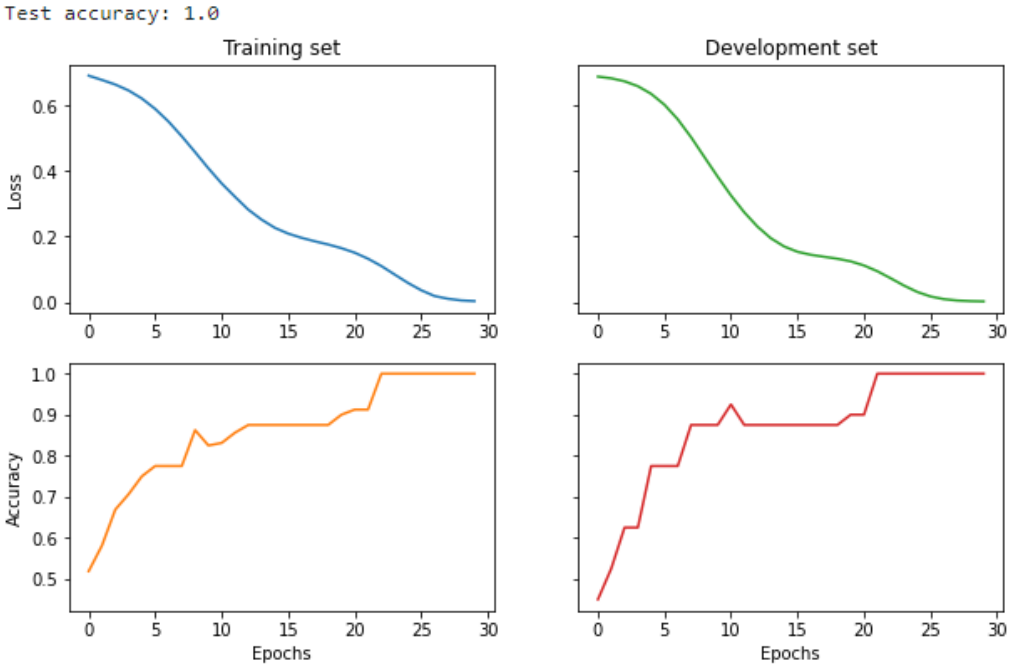


Figure 4.4: SpiderReader with Classical Training

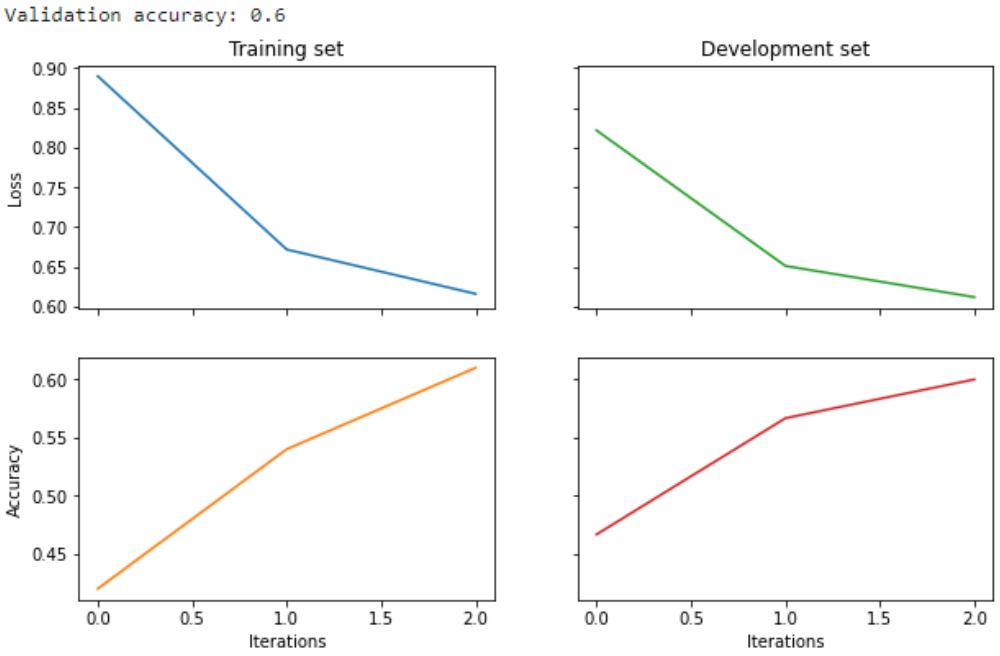


Figure 4.5: BobcatParser with Quantum Training

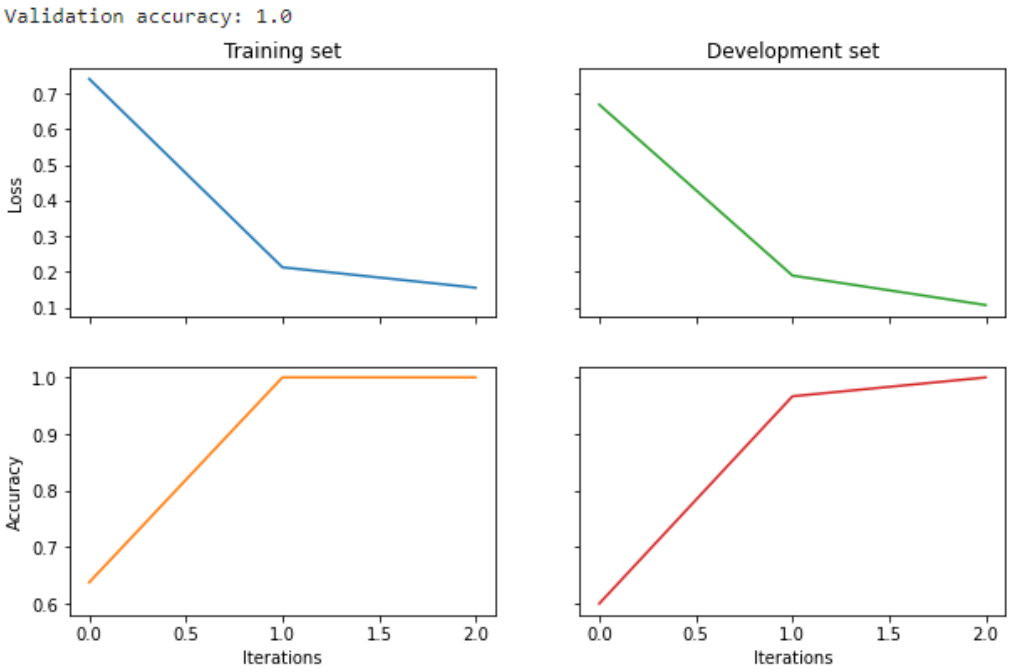


Figure 4.6: SpiderReader with Quantum Training

4.4 Issues in replicating

Below it's a list of some of the difficulties i found replicating the ipynb in the documentation using Lambeq 0.2.2.

- For this version of Lambeq there is a problem downloading the model of BobcatParser. In order to make it work, view Issue #11 on Github. You need to download the model and pass it as argument to BobcatParser.
- Make sure to see Issue #12, you need to modify some lines of code in the library. However, this should be fixed in a next minor release.
- You have to make sure that the output of the Quantum circuit is of the same dimension of the target. In the example, labels are 2 dimensional, while the output of the model was 1 dimensional. I fixed this modifying the Ansatz from

```
ansatz = IQPAnsatz({AtomicType.NOUN: 1, AtomicType.SENTENCE: 0}, n_layers=1,
 n_single_qubit_params=3)
```

  to

```
ansatz = IQPAnsatz({AtomicType.NOUN: 2, AtomicType.SENTENCE: 1}, n_layers=1,
 n_single_qubit_params=3)
```

- Training may be very slow. For me, every epoch was 40 seconds long. This may be because you are using TketModel, that performs a noisy simulation on the QuantumTrainer. To speed it up, you can use NumpyModel, that performs noiseless simulation.

Conclusions

QNLP is a new research area, having its first conference held in 2019. Lambeq's first version was released in October 2021, so it's a brand new library. It surely has some major bugs that need to be fixed, but in my opinion has a lot of potential. It incorporates different functionalities and gives freedom of customization. Actually the documentation only explains functions at a really high level and it's not easy to understand how things work and therefore how to modify the code to optimize it for the use case. Some work could be done in this direction in order to make the library more accessible (however, QNLP is a topic that requires a wide knowledge).