“Quantum Natural Language Processing for Binary Sentiment Detection”

-A work using Lambeq Package (By Amirali Malekani Nezhad)

Table of contents

Abstract

This paper is dedicated to a novel introduction to QNLP as a translation of Neural net models onto the Quantum Mechanics paradigm by representing the core model using quantum circuits and qubit mapping whilst not assuming any quantum mechanical nature for the nature of language and/or NLP. Through this paper we will go through the motivation for this approach, the theoretical approach taken, the detail of the pipeline and lastly the comparison between the two paradigms in a manner to compare accuracy and computational speed. In this paper, we present results on the QNLP model conducted on a Noisy Intermediate-Scale Quantum (NISQ) computer and Ideal Simulator for datasets of size ≥ 300 sentences. By utilizing the formal similarity(representation similarity) of the "compositional model of meaning" by Coecke et al. (2010) with quantum theory, we will provide representations for sentences which possess a natural mapping to quantum circuits. We will use these representations to implement and successfully train a QNLP model that will solve a binary sentence classification task for depression detection, ran on a QPU.

Keywords : QNLP, CNN, Quantum Mechanics, NISQ, binary classification, depression detection, compositional model of meaning

Motivation : *The methodology is that we want to translate the classical NLP model(s) using Quantum Mechanical representations to allow for migrating to QC paradigm for computational supremacy and provide a foundation for QNLP given it is a recently founded research, in order to exploit the quantum supremacy of Quantum hardware.*

Introduction

Bringing the premise of computational speeds exponentially higher than the current standard, quantum computing has been rapidly evolving to become one of the most practical cutting-edge areas of modelling and computation paradigms in computer science, solving two categories of challenges :

1) Speed : Solving problems which take too long on a classical supercomputer centuries to run, can be run on a QPU in a matter of minutes.
2) Representation : Solving problems which classical computers cannot even represent using classical frameworks.

Until recently, most of the work in quantum computing was purely theoretical or concerned with simulations on classical hardware (simulated runs of QPU, using CPUs and/or GPUs to run Quantum Models, not truly quantum computing), the advent of the first quantum computers available to public, referred to as Noisy Intermediate-Scale Quantum (NISQ) devices (quantum computers use qubits as their analog of bits for computation, which are physical entities which are prone to interference from the outside environment, hence are kept in an isolated, subzero container, but still prone to outside noise such as thermal waves, hence noisy, and since there is a limit to the depth of a circuit, similar to a neural net, they are intermediate-scaled), has already led to some promising practical results and applications spanning a wide range of topics such as cryptography (Pirandola et al., 2020), chemistry (Cao et al., 2019)(InQuanto), and biomedicine (Cao et al., 2018) as well as optimization in a variety of fields using Quantum Annealing (D-Wave systems).

One of the fields which has widely utilized the framework of QC is Machine Learning, using NISQ systems for a variety of ML tasks, especially used in classification and simulation of evolutionary operators, hence perhaps an immediate question is whether this new paradigm of computation can also be used for all subsets of ML, namely NLP. Such applicability may be to the end of leveraging the computational speed-ups for language and semantic-related problems, as well as for investigating how quantum systems, their mathematical formulation and representation and the way information is encoded “quantumly” may lead to conceptual and practical advances in representing and processing language meaning beyond computational speed-ups. Inspired by these features, quantum natural language processing, a field of research still in its infancy, aims at the development of NLP models explicitly designed to be executed on quantum hardware.

In this paper we present one medium-scale model consisting of linguistically-motivated NLP tasks running on quantum hardware. The motivation behind this experiments is not to demonstrate some form of "quantum advantage" over classical implementations in NLP tasks; it is believed this is not yet possible due to the limited capabilities of currently available quantum hardware, given decoherence of circuits and scale of current models.

In the following passages we will illustrate how the classical modelling and coding paradigm can be translated to a quantum-friendly form. From an NLP perspective, the task involves some form of sentence classification: for each sentence in the dataset, we use the compositional model of Coecke et al. (2010) – often dubbed as DISCOCAT (DIStributional COmpositional CATegorical) – to compute a state vector, which is then converted to a binary label, indicating whether a sentence is depressive or not. The model is trained on a standard binary cross entropy objective, using an optimization technique known as Simultaneous Perturbation Stochastic Approximation (SPSA), which differs from the traditional approach of gradient-based optimization.

The choice of DISCOCAT is motivated by the fact that the derivations it produces essentially form a tensor network, which means they are already very similar to how quantum computers process data using tensor of multiple qubits, producing many-body states configurations. Furthermore, the model comes with a rigorous treatment of the interplay between syntax and semantics and with a convenient diagrammatic language, allowing for a diagrammatic formulation of semantic and grammatic relation between the entities, following the Lambek formulation.

In Approach, we will observe the manner the produced diagrams find a natural mapping to quantum circuits (the basic units of computation on a quantum computer) and how sentences of different grammatical structure are mapped to different quantum circuits, which the model will train on and we will also speculate how tensor contraction (the composition function) has its expression in terms of quantum gates(operators which are applied to qubits to alter their state, modeled by performing rotations on the Bloch sphere representation.[1]

**For our model we use a Depression Detection dataset(323 sentences) with approximately half of the sentences being depressive and half non-depressive, therefore a binary classification use-case. We*

[1] *The circuit is composed of Hadamard, RX and RZ and CZ gates, which we can observe using .to_tk() function which allows us to view a render of TKET circuit for each sentence.*

demonstrate that the model converges smoothly, and that it produces good results in both quantum and simulated runs.

Approach

In the following passage(s) we will go in detail through the model pipeline and explain the process at each step, in parallel to the CNN pipeline and compare results regarding speed of training and prediction accuracy.

3.1) Problem Statement and Hypothesis

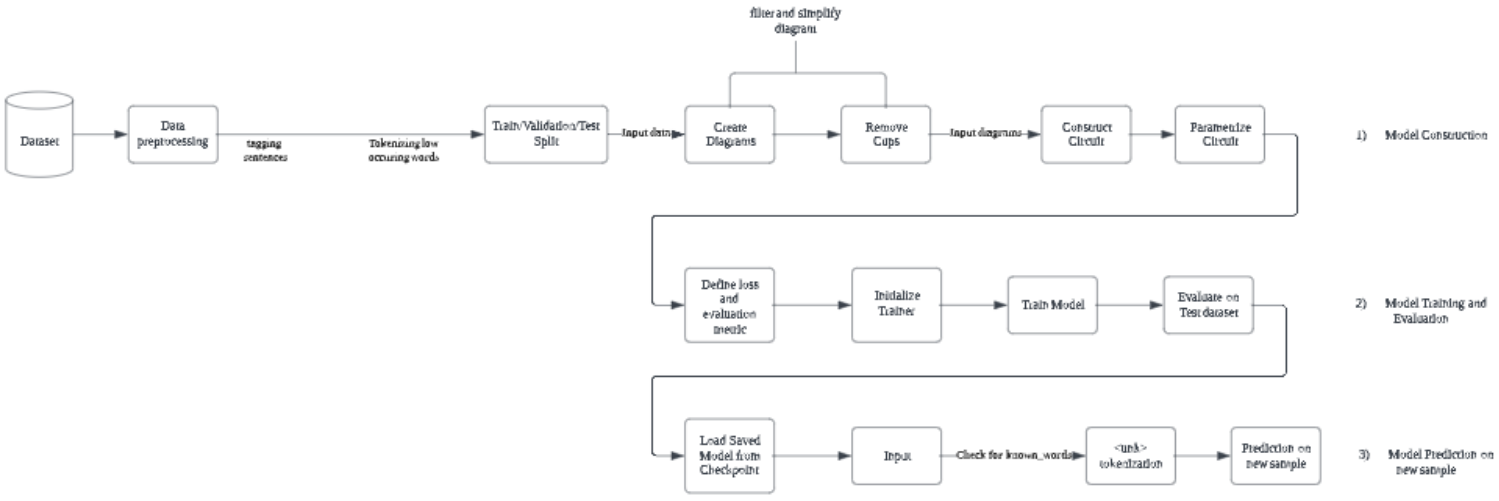
Problem Statement : Design a model to perform binary classification based on analyzing the semantics of an ensemble of sentences and classify whether they are **depressive** or **non-depressive**.

Hypothesis : It is hypothesized that the QNLP model will perform binary classification for text sentiment analysis on depression indication use-case with %80 accuracy and less than 0.1 loss.

3.2) Model Summary

- **Model** : Binary QNLP Classifier (Lambeq)
    - **Objective** : Perform Binary Classification
    - **Cost function** : SPSA
    - **Epoch** : 300
    - **Batch size** : 30
    - **Epsilon** : 2.22 e^{-16}
- **Use-case :** Depression Detection

3.3) Pipeline



Pipeline Description

3.3.1) Model construction and training

1. Prepare dataset
2. Create train/test/validation split
3. Preprocess sentences at sentence level
4. Construct diagrams from sentences
5. tokenize dataset/corpus
6. Filter and simplify diagrams
7. Construct circuits
8. Parametrize circuits
9. Define loss and evaluation metric
10. Initialize trainer
11. Train circuits
12. Evaluate on test dataset

3.3.2) Model prediction on new sample(s)

1. Load trained model from checkpoint
2. Load dataset/corpus and form a list of all known words
3. Create diagram of the input
4. Tokenize input sample for <unk> token
5. Create circuit
6. Run model and output prediction label

**In the following passage we will go into detail as what each step of the pipeline does and which libraries are used for each step when necessary.*

Prepare data : This step illustrates the process of providing the dataset, hence we will cover the dataset methodology. For this paper we used a Depression Detection Tweets dataset which consists of 1 feature and a binary label. The feature consists of sentences and the label is whether the sentence is depressive or non-depressive. The data has been scrapped from Tweeter and labeled based on the semantic being depressive or not, consisting of over 300 rows.

Create train/test/validation split : This step is only applicable given your dataset is not already split into train and test. For our model we will also use a validation dataset as well. The ratio for all is equal, being approximately %33. (For our case we will have a %50, %30, %20 split for training, validation and test set respectively)

Preprocess sentences at sentence level : At this step we will perform sentence rewriting using our own rewrite rules defined for sentence level rewriting, as opposed to Rewriter object instance which works at diagram level. Given the choice of parser, we can use this instead of the Rewriter AND/OR together as a complementary feature. We provide rewrite rules for auxiliary, connector, determiner, punctuation, spelling check, emoji removal and suffix tokenization which can be used in SpaCy tokenization instead.

Construct diagrams : We will input the datasets into the Spiders_reader from Lambeq library to construct the semantic diagrams based on Lambek's grammatical mathematical representation, where for instance a noun is by form "n", and a verb is by form "n.l@s@n.r" where n.l and n.r are nouns adjoint from left and right. Lambek provides a thorough description of how to formulate the grammatical types in this manner.

**You can also use WebParser which is a more formidable parser, but inconvenient given the necessity of a WSL and/or Linux OS to run the parser. For our model we have also provided the pkl processed form of the diagrams from the Webparser.*

Tokenize dataset/corpus : We will use a replace Functor using Discopy.rigid library to interact with the sentence diagrams. Each sentence diagram will be scanned for words which have a low occurrence (frequency of less than 3 in the entire dataset) and tokenize this as <unk> token. What this allows is to train the model on the <unk> token as a means of words which are so rare they can be assumed as unknown, which will allow us to calculate the prediction in prediction on new samples segment. This step also includes the reformation of words like "he's" and "they're" to become "he" + " 's " and "they" + " 're" for better interpretation of suffix.

Filter and simplify diagrams : In this step we will remove the cups from the diagrams to simplify. the basic idea is to first store a list of the words of the DisCoCat diagram, then scan the diagram from top to bottom. When we meet a cup that "fully contracts" a word to another sub-diagram, it performs the diagrammatic transpose and removes the cup. "fully contracted" means the word /sub-diagram has no more open wires after adding that cup. The full algorithm does two passes: one where we "compress" the cups into a "nested cup box", and the same procedure happens, and one where we try to cup removal with the cups compressed. The combination of two strategies seems to yield optimal cup removal for most of the diagrams output by the parser.

Construct circuits : We will first initialize the ansatz which will act as the diagram-to-circuit converter. There are multiple parameters for defining the ansatz, namely :

- AtomicType.NOUN : The number of qubits used to represent a noun, often set either 1 or 2 given the NISQ limitations, but ideally and naively speaking, the more the better.
- AtomicType.SENTENCE : The number of qubits used to represent the label, since we are performing binary classification, we will use 1 qubit, where $|0\rangle$ would mean depressive and $|1\rangle$ would indicate non-depressive.
- AtomicType.PREPOSITIONAL_PHRASE : The number of qubits used to represent a preposition.
- n_layers : This indicated the depth of the circuit, once again the deeper the better but for our case 1 suffices, default set to 1 as Euler decomposition.
- n_single_qubit_params : This essentially indicates the number of single-qubit rotations used by ansatz.

Then we input the diagrams into the ansatz and construct the circuits.

Parametrize circuits : In this step we will define the backend which the model will use to run, traditionally we use the Aer backend widely used by TKET and QISKIT models. We will further initialize the TKET model which will be fed the circuits of all datasets and the backend config.

**This step is optional given whether you want to use TKET model or not. For our instance we will use NumpyModel for a more accurate representation of QPU simulations and speed.*

Define loss and evaluation metric : In this step we will initialize the trainer, and specify the params as follows :

- model : model which we wish to train

- loss_function : loss function is default loss (we will use binary cross entropy given we are performing binary classification)
- epochs : number of epochs, for our model 300
- optimizer : SPSA optimizer
- optim_hyperparameters : optimizer hyperparameters, which is fed to SPSA
- evaluate_functions : Basic y-hat error function
- evaluate_on_train : whether the model should evaluate the train, which is set to TRUE
- verbose : text, this is discussed in more detail in verbosity levels in Lambeq docs
- seed : a random seed

Train circuits : We will call the trainer to fit the model on training and validation datasets and set the logging step to 12 (update of the model for every 12 epochs) and let the model train. Time taken for each epoch is approximately 50 seconds (49.9999998), rendering each 12 epochs approximately 10 minutes.

Evaluate test dataset : We will finally evaluate the model's performance on the test dataset and output the resulting accuracy. For this step we will call the acc function and pass prediction and the actual labels for the test dataset instances.

Load trained model from checkpoint : After the model is trained, it is saved under a folder named runs, which can be accessed in the future for accessing that specific trained model, hence we only train models once and can use them for as many times as we wish across different computers. For this part we will load the model from the checkpoint using TketModel.from_checkpoint to access the model, where we pass the model path address and backend config.

Load dataset/corpus and form a list of known words : Given the scenario of facing inputs where there exists one or more words which the model has never seen, it is crucial to tokenize the model based on the list of known words from the corpus. For this we first will form the list of all known words from the entirety of dataset and save them in a list without repetition.

Create diagram of input : In this step we will construct the diagram of the input sentence using the Spiders_reader Parser.

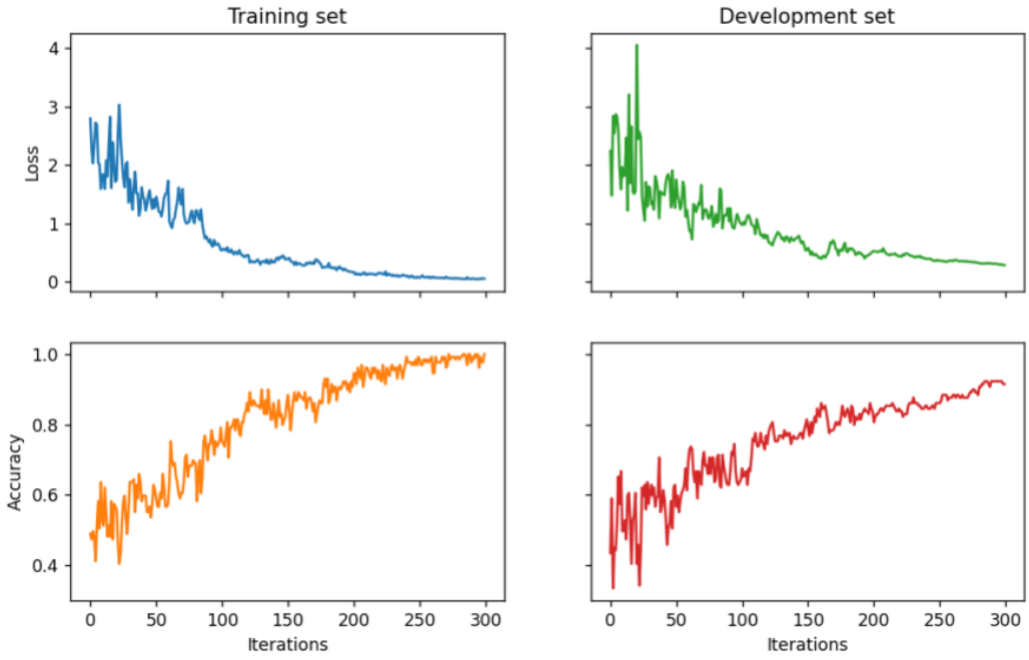
Tokenize input sample for <unk> token : as mentioned, we need to tokenize unknown words as <unk>, and we will do so by performing a check and then calling the replace function seen in the previous tokenizer step and replace the word box given it is not in the list of known words.

Create circuit : We will pass the tokenized diagram to the ansatz after filtering it and construct the circuit.

Run model and output prediction label : Lastly we will run the model and output the prediction label which is in the form of an np.array of two elements, indicating the likelihood of the sample being each label respectively where we call an activation and given which element is more likely, we choose that as the predicted label.

Model Performance

Given the dynamics of the hyperparameters and choice of number of qubits, we have experimented with the best combination for the best model :



Model 1

Conclusion Summary

Given the field of QNLP (as well as Lambeq itself) being a quite new field of research, there are still a great amount of work to be done in order to allow it to reach its fullest potential and be comparable to classical deep neural nets. Given the use of Qubits there is still much needed hardware advance to be made in order to provide a more powerful model with more qubits and deeper circuits.

Currently the model yields an accuracy of %100 on training set, %91 on validation set and %81 on test dataset for a binary sentiment analysis depression detection model. To reach a more optimal model requires further experimentation with different models (given the initial use of TKET and it being incredibly slow currently to use, we migrated to NumpyModel, hence there is still a need to reform the models and test again based on TKET), based on different alpha values, different number of layers as well as different number of qubit representations.

Future works

Given lambeq still being in its infancy, it portrays great potential as the first ever QNLP based framework, using ZX calculus and Lambek's modelling of NLP which is something which has never been done before and has been showing great results.

Regarding the future works for this particular model, we will try to enable a GPU-based environment to try the model with TKET at a considerably faster speed and try the results of model being trained with TKET. Furthermore we can enable unique tokenizers for special letters, acronyms, shorthand words (for instance idk, tbh, rn etc.) as well as better parsers, which as discussed would be a WSL-enabled interpreter for locally runnable Web Parser, allowing for an improved semantic diagram-conversion for a variety of grammatical sequences. Lastly we will provide a better range for the hyperparameters and the batch size given the dataset in order to increase the model accuracy and decrease the loss.

References

QNLP in Practice : Running Compositional Models of Meaning on a Quantum Computer

https://arxiv.org/abs/2102.12846

Appendix

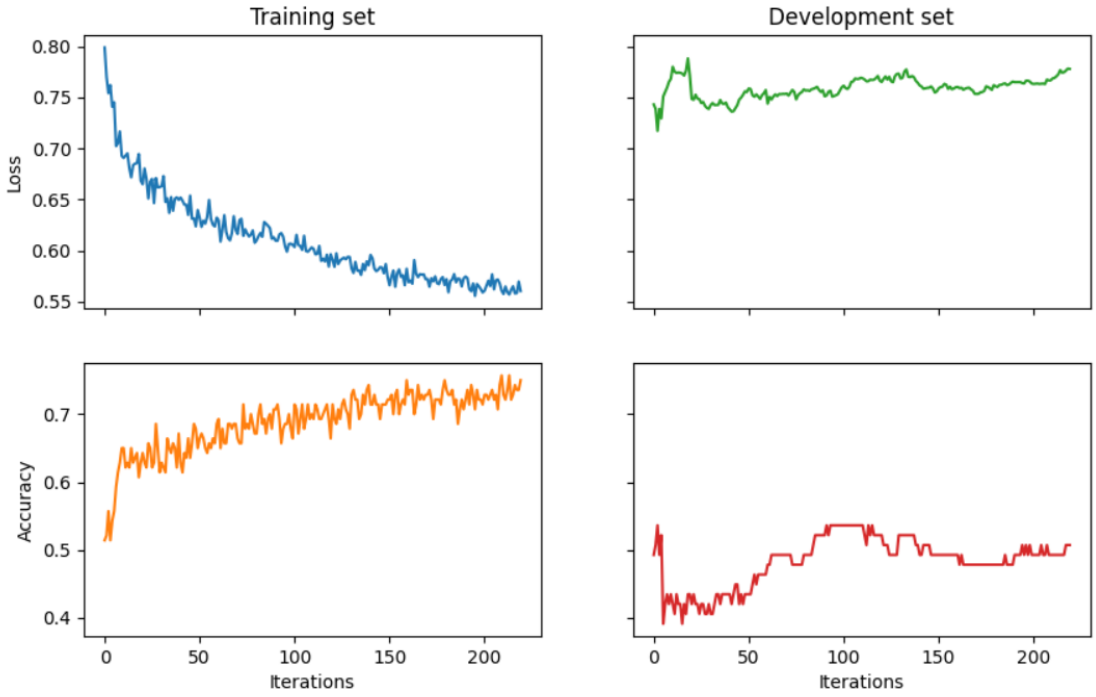


Figure 1 Bobcat parser

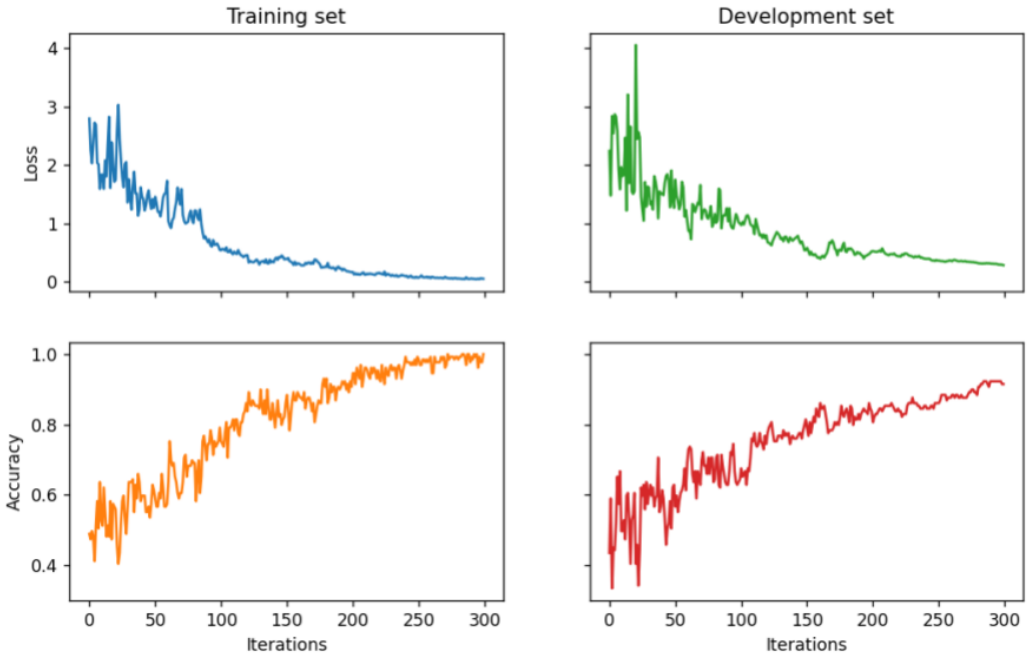


Figure 2 Spiders_reader

Resources (Github repo and Experiment Summary)

https://github.com/ACE07-Sev/QNLP-Quantinuum-Challenge

https://docs.google.com/document/d/1A_Wk0rcOOAFr3w6K7tsWd8YB4VOL9dXteOaea0Bgyy4/edit