POKT AI Lab

Report III

Authors:
Nicolas Aguirre (nicolas@poktscan.com)
Ramiro Rodríguez Colmeiro (ramiro@poktscan.com)
POKTscan Data Science Team

May 31, 2024

Disclaimer

This work was partially funded by Pocket Scan Technologies LLC, a company operating on the Pocket Network.

Changelog

v1.0	2024-05-30	First version.

Table of Contents

1 Progress Overview

During the month of May, the socket achieved the following milestones:

- Merged issues on the Test-Bench code:
  - Manager:
    - Add buffers logic [1].
    - Implemented circular buffers logic for metrics [2].
    - Task Triggering [3].
  - Sampler: Generate prompts and save to mongo + Tokenizer [4].
  - Requester: Complete requester MVP [5].
  - General: Fixes and enhancements to create the first version of Morse POC [6].

The expected milestone, finishing an MVP presentation, was not achieved due to some issues during the integration of the first workflows. Specifically we were stuck with some known, but difficult to track, bugs around TemporalIO, Python and asynchronic processes embedded into Language Model Evaluation Harness (LMEH) library [7]. Also we spent some non-negligible time around the tokenizer problem that is discussed in this document, which we believe to be of interest to the POKT Network language model (LM) inference effort.

[1] https://github.com/pokt-scan/pocket-ml-testbench/issues/36
[2] https://github.com/pokt-scan/pocket-ml-testbench/pull/38
[3] https://github.com/pokt-scan/pocket-ml-testbench/issues/40
[4] https://github.com/pokt-scan/pocket-ml-testbench/pull/47
[5] https://github.com/pokt-scan/pocket-ml-testbench/pull/37
[6] https://github.com/pokt-scan/pocket-ml-testbench/pull/51
[7] https://docs.temporal.io/encyclopedia/python-sdk-sync-vs-async

2 Language Models - Tokenizer

A picture is worth a thousand words,
and a word is worth around 0.75 tokens.

Unkown, 2024

Today the LMs, whether large or small, can understand and produce human language (to some extent), but we know that the way that these models process the text differs from how we do it. A LM, before it can start to work with any text, needs a way to break down and translate that text into a form of information that it can process, namely, numbers. This is where the *tokenizer* comes in.

In this sections we briefly introduce the concept of *token*, *tokenization* and *tokenizer* and how they work in a LM. Later we describe an issue that the Machine Learning Test-Bench (MLTB) (and the POKT Network) faces when trying to interact with unknown LMs and propose a requirement for the deployment of models that will solve it.

2.1 From string to token

Formally, the *tokenization* is the process of representing text in smaller meaningful lexical units, called *tokens*. Meaning, breaking it apart in a way that is useful and information is preserved, and assigning each part an unique number. There are three popular ways to do the same, which are:

- **Word Tokenization :** This is the simplest form of tokenization, where each word is considered a token.
- **Sub-Word Tokenization :** Breaking words into subwords and treat each part as tokens.
- **Character Tokenization :** A very naïve tokenization that considers each character as a token.

As always, there is a trade-off between the different approaches, in this case the granularity of the tokens (how many you need to decompose a text) and the number of tokens (how many unique tokens you have). That is, while word tokenization has low granularity and high token count, character tokenization has high granularity and low token count.

Nowadays, the most popular tokenization method is subword tokenization, which is a compromise between word and character tokenization. Tokenizers (advanced ones at least) are standalone machine learning techniques [4, 3, 5] whose output is dictionary that assigns numbers (tokens) to different strings. It is important to note that prior to training a model, tokenizers must be trained on texts that is representative of the training set (ideally the tokenizer texts should not be part of the training set). So, when training a LM for technical english text, the tokenizer should be trained on technical english words, and as a result you should have an effective description of the language to use. For example such a tokenizer might assign a single token to word that repeats to much in technical english, like "therefore", but have to use 2 tokens to represent other common english words such as "puppy" (using "pup" and "py" maybe), because they are rare in technical english.

As we said, the tokenizer is the one in charge of carrying out the tokenization process, which consists (in a nutshell) of a dictionary that contains pairs of tokens and their numerical representation. This numerical representation is the actual data that is analyzed by the LM and for this reason, a direct relationship is established between a LM and its tokenizer.

Tokenizers in Language Models Evaluations

This relationship between a LM and tokenizer is crucial when evaluating the performance of a LM, since one of the main techniques for evaluating a language LM is through the calculation of the probability of occurrence of a sequence of tokens that is conditioned by an initial sequence of tokens. In other words, the probability of following a sentence with a given set of words, when some initial words (context) are provided. In terms of `lm-eval-harness` [1] these two sequences are also called `context` and `continuation`, as shown in Figure 1.

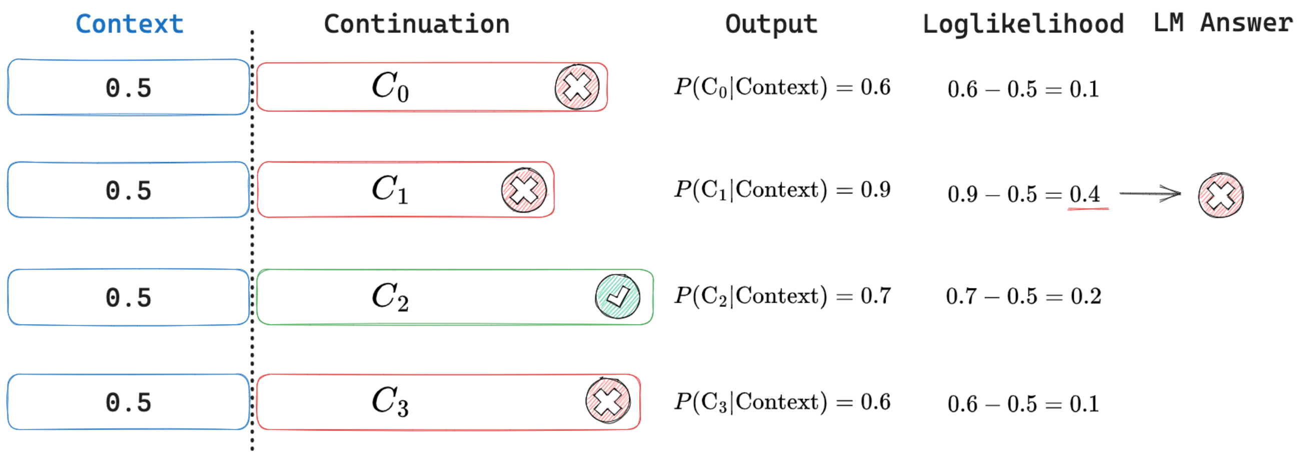
Context
Continuation
Output
Loglikelihood
LM Answer
0.5
C_0
$P(C_0|\text{Context}) = 0.6$
$0.6 - 0.5 = 0.1$
0.5
C_1
$P(C_1|\text{Context}) = 0.9$
$0.9 - 0.5 = \underline{0.4}$
0.5
C_2
$P(C_2|\text{Context}) = 0.7$
$0.7 - 0.5 = 0.2$
0.5
C_3
$P(C_3|\text{Context}) = 0.6$
$0.6 - 0.5 = 0.1$

Figure 1: Example of `context`, `continuation` token sequences and evaluation result. The green segment in `continuation` represents correct segment (ground truth), the *Loglikelihood* column represents the segment of higher probability (assigned by the LM), and finally the *LM Anwer* correspond to the result of the evaluation example. In this case, C_2 should be the higger probability, nevertheless the LM returns C_1 with higher probability.

Without going into the mathematical details, and following the Figure 1, the concatenation of the `context` and `continuation` tokens is sent to the LMs, and from their returns it can be computed the probability of occurrence of the entire phrase, detailed token by token. The LM is considered to respond correctly when the probability of occurrence of the correct continuation is the highest with respect to the other continuations. That is, in terms of the Figure 1, the LM responds incorrectly, since C_2 should have been the continuation with the highest probability, but the LM returns a higher probability for C_1.

2.2 On the lack of knowledge of the tokenizer

Not knowing the tokenizer of a model might be irrelevant when we are only interested in interacting with a model. As long as the Application Programming Interface (API) accepts plain text, we can use the served LM. While this is true, it restricts the versatility of solutions that we can construct around the POKT network. Specifically it poses difficulties in the following aspects:

Decentralized Evaluations

As previously detailed, the LM receives the concatenation of the context and continuation token sequences, and then locally separates the context and continuation probabilities. However, in a decentralized environment, such as that proposed by MLTB, if you do not have access to the tokenizer, it is not possible to separate the context and continuation probabilities. While the plain text of the concatenation of context and continuation could be sent, we could not directly determine which part of the text belongs to the context and which part belongs to the continuation, thus preventing the evaluation of the model. Although one could try to reconstruct the context and continuation phrase from the concatenation of tokens, this is not a trivial task, since the tokenizer could have tokenized differently than the original text was tokenized [1].

Price and Query Optimization

The gateways deployed in the POKT Network will probably want to be able to infer the price of a call prior sending it to a LM-Node. While currently we do not count tokens, it is expected that the price of a LM Remote Procedure Call (RPC) will be modulated by the number of tokens being processed (in some way or to some extent at least). Knowing the tokenizer nature will become a necessity to estimate and optimize the RPC calls.

Model Verification

Given the LM-*Tokenizer* relationship, knowing the tokenizer of a model already let us know something about it. A tokenizer can be used in many models, but a model can use a single tokenizer, this way we reduce the search space of models if we know its tokenizer. Also, if the tokenizer changes at some point, we can be sure that the served model changed.

Another important fact is that we must remain open to model verification techniques. For example, one of these techniques is called *watermarking* [2] and works by detecting the presence of probabilities of certain *tokens* in the produced text.

2.3 Proposed solution

Faced with this situation, there are three other plausible solutions (at least for the MLTB).

The first solution would be to send, in addition to each complete sentence, only the `context` text. This would then allow us to remove those corresponding to the `context` from each of the complete sentences and continue with the evaluation of the model. The disadvantage of this solution is that $N + 1$ calls would be made to the model, where N is the number of complete sentences to evaluate, resulting in a higher cost. Furthermore, in the case of evaluations that contemplate the use of conversational models (such as the case of chat), this solution would not be applicable, since internally these models through its toke add special tokens to indicate the beginning and end of a message in a conversation, just as they usually have a text to condition the model's response, called system prompt.

The second solution would be online tokenization, calling for example an endpoint *tokenize* (in a custom API). This situation would face similar problems, as currently discussed in the community [8].

The third solution would involve sending the complete tokenizer associated with an node address. This was detailed in two PRs [9] [10] in both the LMEH repository and the vLLM repository, a popular LM inference engine. The main advantage of this solution is that the issues of the first and second solutions are avoided. Additionally, as previously detailed, the tokenizer is tied to the model. Therefore we can generate a hash of the tokenizer that sends us an address, save it in a database and then verify that the hash of the tokenizer that is sent at the beginning of a session is equal to the hash stored in the database. If for some reason the hash does not match, it is an unequivocal indication that the LM behind the endpoint is not the same, it has changed, therefore the previous results are not valid, metrics are eliminated from the leaderboard, and that address has to be re-evaluated.

From these three solutions we argue that the third one (having access to the full tokenizer) is the best to implement in the POKT Network. This solution is based on solutions that already being implemented in the LM community and provide us with the largest versatility for future developments.

[8] https://github.com/huggingface/text-generation-inference/issues/1706
[9] https://github.com/EleutherAI/lm-evaluation-harness/pull/1794
[10] https://github.com/vllm-project/vllm/pull/2643

3 Future Work

During the month of May we focused on merging the parts developed until now individually developed. This included the `Manager`, `Sampler` and `Requester`, plus a single `docker-compose` that includes a local-net composed of three `Lean-nodes` with eight validators each and a `Geo-Mesh` for testing that everything works.

During the month of June we hope to complete the MLTB code, including:

1. `Manager`:
    a. Create logic for tokenizer signature task [11].
    b. Clean completed tasks [12].
    c. Add signatures to node [13].
2. `Sampler`:
    a. Create logic for tokenizer signature task [14].
    b. Add asynchronous read operations to the MongoDB collections.
3. `Evaluator`: Create initial code for LMEH processing [15].

Next month we expect to be concluding the POKT AI Lab Socket [16] experiment. We should to have a running demo and being able to show the POKT community some live alpha of the developed product!

[11] https://github.com/pokt-scan/pocket-ml-testbench/issues/42
[12] https://github.com/pokt-scan/pocket-ml-testbench/issues/41
[13] https://github.com/pokt-scan/pocket-ml-testbench/issues/39
[14] https://github.com/pokt-scan/pocket-ml-testbench/issues/43
[15] https://github.com/pokt-scan/pocket-ml-testbench/issues/44
[16] https://forum.pokt.network/t/open-pokt-ai-lab-socket/5056

List of Acronyms

API	Application Programming Interface
LM	language model
LMEH	Language Model Evaluation Harness
MLTB	Machine Learning Test-Bench
RPC	Remote Procedure Call

Reference List

[1] Stella Biderman et al. *Lessons from the Trenches on Reproducible Evaluation of Language Models*. May 23, 2024. DOI: 10.48550/arXiv.2405.14782. arXiv: 2405.14782[cs]. URL: http://arxiv.org/abs/2405.14782 (visited on 05/29/2024).

[2] John Kirchenbauer et al. “A watermark for large language models”. In: *International Conference on Machine Learning*. PMLR. 2023, pp. 17061–17084.

[3] Taku Kudo. *Subword Regularization: Improving Neural Network Translation Models with Multiple Subword Candidates*. Apr. 29, 2018. DOI: 10.48550/arXiv.1804.10959. arXiv: 1804.10959[cs]. URL: http://arxiv.org/abs/1804.10959 (visited on 05/29/2024).

[4] Taku Kudo and John Richardson. *SentencePiece: A simple and language independent subword tokenizer and detokenizer for Neural Text Processing*. Aug. 19, 2018. DOI: 10.48550/arXiv.1808.06226. arXiv: 1808.06226[cs]. URL: http://arxiv.org/abs/1808.06226 (visited on 05/29/2024).

[5] Rico Sennrich, Barry Haddow, and Alexandra Birch. *Neural Machine Translation of Rare Words with Subword Units*. June 10, 2016. DOI: 10.48550/arXiv.1508.07909. arXiv: 1508.07909[cs]. URL: http://arxiv.org/abs/1508.07909 (visited on 05/29/2024).