Advancements in Laotian-Vietnamese Translation: An In-depth exploration of Transformer architectures and improvements

Quang-Minh Hoang
VNU-UET
21021529@vnu.edu.vn

Hoang-Ba Tran
VNU-UET
21020631@vnu.edu.vn

Nghia-Hieu Nguyen
VNU-UET
21021523@vnu.edu.vn

***Abstract*—This report is about the research and experiments on the Translation Model for the Final Project of NLP class INT3406E 20 in Fall 2023. We report on the improvement of the lo-vi translation model that the baseline is from https://github.com/KCDichDaNgu/KC4.0_MultilingualNMT - Neural Machine Translation Toolkit [1]. Our source code is available at: https://github.com/hoangbros03/Translation_Model_10.**

I. Introduction

The transformative Transformer architecture, introduced by Vaswani et al. in their groundbreaking paper "Attention is All You Need" [2] in 2017, marked a paradigm shift in sequence transduction models. Before the advent of Transformers, prevalent approaches relied on intricate recurrent or convolutional neural networks, incorporating both an encoder and a decoder for tasks such as machine translation. The distinguishing feature of the Transformer is its departure from sequential dependencies, opting instead for attention mechanisms that foster global connectivity between input and output. This departure not only facilitates more efficient parallelization during training but also mitigates some of the challenges associated with the sequential nature of recurrent models, allowing the model to consider all positions in the input sequence simultaneously. This self-attention mechanism empowers the Transformer to capture intricate long-range dependencies, distinguishing it as a versatile and powerful architecture. Particularly advantageous in natural language processing tasks, the Transformer's impact extends beyond its initial applications, demonstrating its adaptability and efficiency in handling diverse sequential data. The architectural innovation of the Transformer has left an indelible mark on the landscape of machine learning and artificial intelligence, reshaping the way we approach and model sequential information.

II. Baseline model

A. The base repository

The foundational model, derived from the Neural Machine Translation Toolkit [1], made its debut in 2021 as an expansive open-source toolkit specifically crafted for the field of neural machine translation (NMT). Marking a significant stride in the realm of language processing, the Neural Machine Translation Toolkit was unveiled in 2021, showcasing its prowess as a versatile open-source tool meticulously designed to facilitate both bilingual and multilingual translation tasks. From the initial stages of constructing the model using diverse corpora to the subsequent stages, involving the generation of novel predictions or encapsulating the model in a service-capable Just-In-Time (JIT) format, the toolkit offers a comprehensive suite of functionalities to cater to the diverse needs of NMT applications. Regarding the furnished model, the foundational repository is meticulously constructed, featuring transparent commits that showcase a high level of clarity in the version history. The architectural framework of the project is not only intelligible but also thoughtfully designed, providing a clear roadmap for us to navigate through the intricacies of the codebase. The source code itself is not just well-organized; it is an exemplar of neatness, with insightful comments strewn throughout. These comments serve as valuable guideposts, offering insights into the intricacies of data processing, the construction of the transformer, and the execution steps for running the core model. By delving into the source code, developers gain not only a functional understanding of the program's inner workings but also valuable insights into best practices, enabling them to grasp the nuances of the model's architecture and implementation details. The transparent and informative nature of the source code significantly contributes to a collaborative development environment and fosters a sense of inclusivity, allowing us to engage with and contribute to the evolution of the model.

B. Transformer

The Transformer architecture, when introduced in 2017, changed the field of natural language processing and machine translation dramatically. Unlike traditional sequence-to-sequence models that rely on recurrent or convolutional layers (RNN or LSTM for instances), the Transformer architecture is built upon the mechanism of self-attention. This mechanism allows the model to weigh different parts of the input sequence differently during processing, enabling more effective capture of long-range dependencies in the data.

1) Positional Encoding: Positional Encoding is a crucial component of the Transformer architecture designed to impart information about the order or position of tokens within a

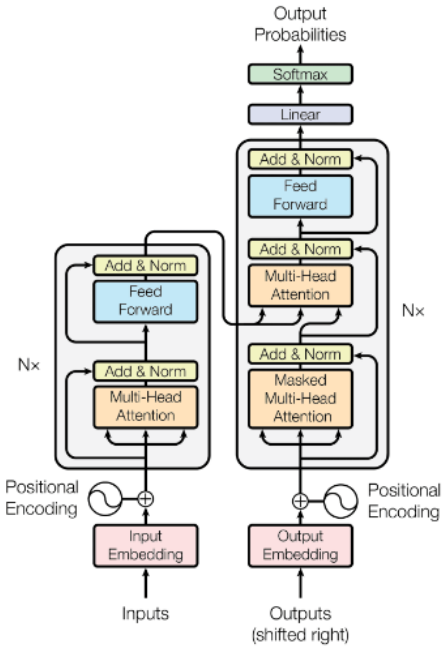

Fig. 1. The architecture of the Transformer in base repository.

sequence. Since the self-attention mechanism in Transformers doesn't inherently account for the sequential nature of input data, positional encoding is added to the input embeddings to inject positional information. Typically, sinusoidal functions of different frequencies are used to create positional encodings, ensuring that the model can distinguish between tokens based on their positions in the sequence, thereby capturing sequential dependencies effectively.

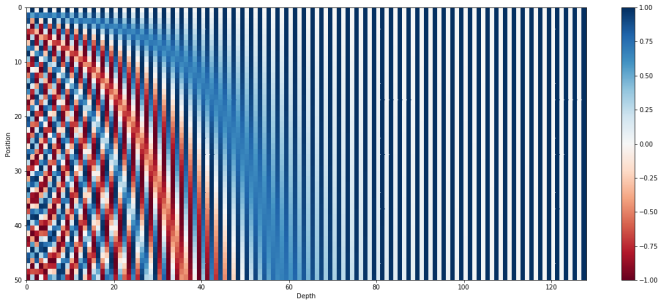

Fig. 2. Positional encoding illustration.

2) Multi-Head Attention: Instead of relying on a single attention head, which can have limitations in capturing complex relationships, Multi-Head Attention uses multiple attention heads in parallel in its architecture. Each end operates independently and learns differently weighted combinations of input sequences, providing the model with a more comprehensive understanding of contextual information. The output from these ends is then linearly combined, allowing the model to attend to different parts of the chain simultaneously. This parallelization not only improves the model's ability to capture long-range dependencies but also facilitates efficient processing, making Multi-Head Attention a key factor contributing to the success of these models. Transformer image in various natural language processing tasks.

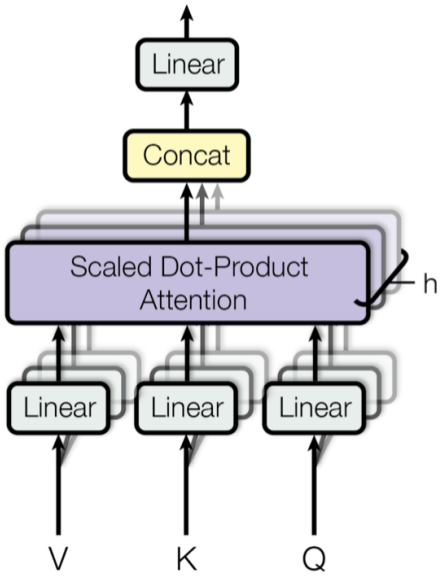

Fig. 3. Multi-Head Attention illustration.

3) Encoder Layer: The Encoder in the Transformer architecture is responsible for processing the input sequence and extracting relevant features. It consists of multiple identical layers, each comprising two main sub-layers: a multi-head self-attention mechanism and a position-wise fully connected feed-forward network. The self-attention mechanism allows the model to weigh different parts of the input sequence differently, capturing long-range dependencies. The feed-forward network introduces non-linearity to the transformations. Additionally, each sub-layer is followed by layer normalization and residual connections, contributing to stable and efficient training.

4) Decoder Layer: The Decoder is tasked with generating the output sequence based on the processed information from the Encoder. Like the Encoder, the Decoder consists of multiple identical layers, each containing three sub-layers: masked self-attention, encoder-decoder attention, and a position-wise fully connected feed-forward network. The masked self-attention mechanism prevents attending to future tokens during decoding. The encoder-decoder attention allows the model to focus on different parts of the input sequence when generating the output. Similar to the Encoder, each sub-layer in the Decoder is followed by layer normalization and residual connections, promoting stability and effective training. Together, the Encoder and Decoder components enable the Transformer to achieve remarkable results in various natural language processing tasks.

C. Translation model

1) Data process: The dataset comprises sentences in both Vietnamese and Lao, segmented by periods and newline characters. The path to the data is specified in the project's configuration file. The data is categorized into monolingual and multilingual forms. Monolingual data is processed in the default loader, while multilingual data is handled in the multilingual loader.

In the default loader, data can take the form of individual sentences or a collection of sentences. Subsequently, the model proceeds to build the vocabulary. In this process, there are three scenarios. If a path exists to a preprocessed vocabulary set, the model loads the vocabulary from that path. If the path is absent but data intended for vocabulary construction is provided, the model builds the vocabulary from the supplied dataset using TorchText's build vocab function. In the event that neither of these data types is available, the model constructs the vocabulary from external files that are preformatted.

In the case of the multilingual loader, data is loaded from multiple training and validation sets. Sentences undergo tokenization, and language-specific tokens are added during data loading. Vocabularies for source and target languages are constructed using TorchText fields, with language-specific tokens being included.

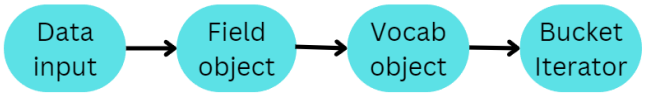

Fig. 4. Database pipeline illustration.

2) Training & evaluating process: The set of languages is maintained during the creation of the MultiDataset instance, ensuring that language-specific tokens are appropriately added to the vocabulary. During the training step in the model, after being processed through the data loader, source and target sequences are transferred to the specified device. The input target sequence is then extracted by removing the last token from the original target sequence. Masks are created to differentiate between padding and actual tokens in both the source and target sequences, playing a crucial role in the attention mechanism of the Transformer. Subsequently, these masked sequences are forwarded to the model. The output of the model for a given sentence is a list of tokens formatted using the loader's detokenize function. This comprehensive approach ensures effective language processing and model training within the specified multilingual context.

After the training process, the inference process plays a pivotal role in determining the efficacy and quality of the final output. Therefore, two prominent strategies, Beam Search and Greedy Search are implemented in the base repository. Beam Search, with its ability to explore multiple possibilities simultaneously, offers a nuanced approach by considering a predetermined number of top candidates at each step. This strategy enhances robustness by mitigating the risk of getting stuck in local optima, ultimately contributing to more accurate and contextually relevant translations. On the other hand, Greedy Search adopts a more straightforward path by selecting the most probable word at each decoding step, leading to a faster inference process.

III. Problems in baseline repository

A. Baseline disadvantages

1) Data provided: The data at hand comprises data files in both Vietnamese and Lao languages. These files, utilized for the training and testing phases of the model, are composed of 100,000 and 2,000 sentences, respectively. Each sentence in the dataset is demarcated by newline characters. Notably, the dataset exhibits a prevalent issue where sentences frequently incorporate redundant or distracting characters, which have yet to be systematically eliminated. Moreover, an additional challenge arises from the presence of sentences that surpass the model's designated length constraints, leading to errors in the training process. Consequently, addressing these issues becomes pivotal for enhancing the overall efficacy and accuracy of the model. Efforts to streamline the dataset by removing superfluous characters and appropriately managing sentence length will contribute significantly to the model's performance and facilitate a more robust training process.

2) Transformer: In fact, the original Transformer was implemented well on base repository. However, in the scope of machine translation task, there are rooms for improvement.

- There is no way to implement pre-trained word embedding. In an ideal scenario, word embedding helps model speed up and decrease the loss value effectively during the training process. To be specific, identical words have the small Euclidean distance in pre-trained word embedding, which can't be achieved using available PyTorch's embedding component.
- Tokenize functions are too simple by just split the sentences based on the space characters. While in English there should be no problem, the performance on Vietnamese and other languages which have many words with spaces (e.g. "đi học", "đường xá", "cầu cống") was affected negatively. Also, there is no instruction in order to improve that from the authors.
- Currently, the number of encoder layers and decoder layers are the same as defined in the config file. However, these number don't need to be the same for training process. With a limited hardware resource, we can reduce the number of decoder layers while keeping the model performance at a reasonable scale.
- Bucket iterator was made to choose the sentences with similar lengths into the training process on each iteration. While we can save time with this briliant strategy, we can't span with into multiple GPUs to parallel training. The reason is justified in the below subsection.

3) Code quality and maintenance: Since the release time of this base repository (2021), python packages including PyTorch, Numpy, and other popular packages have changed

dramatically. These shifts have rendered the existing code base susceptible to numerous vulnerabilities and confused exceptions. For example, we can see the depreciation of certain objects within TorchText, such as Vocab, Field, and BucketIterator, which are no longer featured in contemporary tutorials. On the other hand, we saw some weird errors when training and evaluating due to Consequently, understanding and executing this repository now demands a level of expertise from developers instead of beginner.

B. Ideas to improve

Acknowledge the limitations, we proposed the following ideas to improve the code base and machine translation architecture, including but not limited to:

- Add more data by crawling from popular websites.
- Implement a pipeline to retrieve new data from online websites as well as translate it automatically without human effort.
- Implement a new effective way to tokenize the Laotian and Vietnamese words.
- Change the BucketIterator to another similar object which haven't deprecated yet and support training on multiple GPUs.
- Add some pre-processing and post-processing to enhance the quality of output sentences.
- Change the hyperparameter's number in the configuration file, and add some options for users to fit their interest when training.

IV. Our improvements

A. Data improvements

1) Crawl data: We collect Vietnamese data from the following 3 websites:

- https://vi.wikipedia.org/wiki/Wikipedia
- https://truyenfull.vn/
- https://nhandan.vn/

Utilizing Beautiful Soup, a specialized Python library crafted for the explicit purpose of web scraping, we adeptly extract data from both HTML and XML files. This versatile library furnishes Python developers with expressive idioms for seamless iteration, search, and modification of the parse tree, significantly facilitating the extraction of information from diverse web pages.

Our web scraping methodology commences with an in-depth exploration of the HTML file structure for each target website. This initial phase involves discerning unique characteristics such as the nomenclature conventions for <a> tags and the structural distribution of sentences. Armed with this comprehension, we proceed to craft Python code leveraging Beautiful Soup to systematically gather data from each website.

Our approach involves the extraction of sentences through the discernment of <p> tags, indicative of paragraph structures in HTML. Upon initiating the web scraping process, we dynamically navigate through each web page, systematically adding further links based on specific <a> tags encountered during the parsing. This iterative process ensures a comprehensive exploration of the web ecosystem surrounding the initial pages.

Subsequently, we extend our data collection efforts by following the added links, perpetuating the data retrieval process until a predefined number of sentences is acquired or until the available links are exhausted. This iterative and adaptive approach allows us to curate a diverse and comprehensive dataset, fostering the robust training of models and the extraction of meaningful insights from the dynamic content of varied web domains.

After crawling the data, the sentences will be written into files with the extension ".vi" and names corresponding to each website.

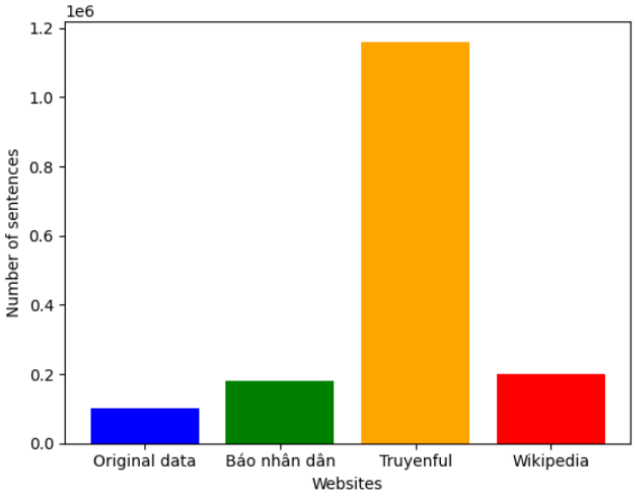

Fig. 5. Number of data crawled.

2) Translation: Upon obtaining a dataset in Vietnamese, our workflow seamlessly expanded to include the translation of these collected sentences into the Lao language.

Our initial strategy involved leveraging the googletranspy library, a powerful and freely available Python library that integrates with the Google Translate API. This library ingeniously taps into the Google Translate Ajax API, enabling seamless calls to essential methods such as detect and translate. The intent was to harness the versatility and accessibility of this library for our Vietnamese-to-Lao translation needs.

However, practical testing revealed a significant limitation: the translation process proved to be notably sluggish when dealing with a substantial number of sentences. Specifically, the effectiveness of the code diminished when the sentence count exceeded 10,000, leading to undesirable delays and compromising overall efficiency during translation tasks. Recognizing the importance of optimizing our workflow, we decided to reevaluate our approach and seek a more efficient solution to address this bottleneck.

In light of these insights, we deemed it imperative to explore alternative options that would circumvent the performance

issues identified with the googletrans-py library. Our commitment to enhancing efficiency and mitigating time loss prompted us to pivot towards a more robust and scalable solution, ensuring that our translation process aligns seamlessly with the demands of a larger dataset.

Our process began by segmenting files with the ".vi" extension into smaller, more manageable versions, each housing a maximum of 20,000 sentences. This segmentation strategy was devised to enhance efficiency and streamline subsequent translation tasks.

To facilitate the translation pipeline, we executed Python code to transition files from the ".vi" extension to their corresponding ".txt" counterparts. Subsequently, we employed online tools to further convert the ".txt" files into the ".docx" format, a step crucial for the subsequent interaction with translation services.

Google Translate emerged as our tool of choice for the translation phase, seamlessly converting Vietnamese files into their Lao equivalents. Notably, we adhered to a constraint of 20,000 sentences per file, a deliberate choice aimed at preventing overload issues with Google Translate that could arise from excessively large file sizes, ultimately mitigating potential errors during the translation process.

The translation of ".docx" files in Lao was seamlessly handled using the python-docx library, an adept Python tool tailored for processing and manipulating "docx" files. This step facilitated the conversion of Lao files from the ".docx" extension to the ".lo" extension, aligning with our project's conventions.

After obtaining the divided Vietnamese files and how to convert them to Lao files, we uploaded them to Google Drive and assigned the task of translating specific files to each member. The output of this process is Lao language files with the ".lo" extension that have been processed.

In the final phase, these processed and translated files were methodically merged back together. This comprehensive dataset, now comprising translated Lao sentences, augments our training and testing datasets, empowering the model with linguistic diversity and contributing to its robustness in handling multilingual data sources. Our systematic approach ensures the seamless integration of translation processes, resulting in an enriched dataset poised for the advancement of our model's proficiency.

3) Preprocess: After get the large files containing new data, there some steps needed to make the data cleaner. Using regular expressions, we meticulously eliminate superfluous and meaningless symbols in the translation process. Only characters from Laotian and Vietnamese languages, digits, and punctuation symbols within sentences are retained, as revealed in our pre-processing Python file, readily available in our public repository.

After that, sentences with inadequate size also be removed. In fact, we already know that spliting these sentences into smaller sentences is a better choice, but since we are using a pair of source and target files, removing is safer. This method guarantees that sentences at index i in both source and target

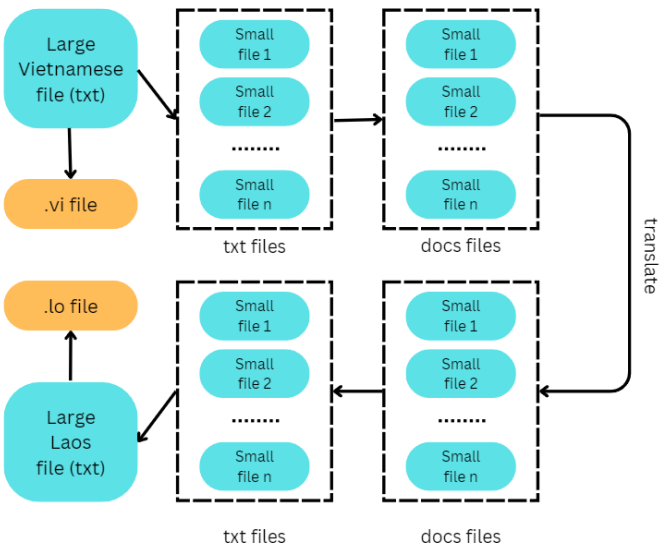

Fig. 6. Translate process illustration.

files unequivocally share identical meanings, fortifying the integrity of the dataset.

4) Tokenizer: In the pursuit of optimal tokenization customization, we have digged into the official documentation, uncovering a range of options that could be used to our specific needs. We figured out that Field object allows to pass a customize tokenize function. When no function is set (as default in base repository), it just split each word in a sentence to a list by using the space character as the indicator.

To achieve effective tokenization for both Laotian and Vietnamese sentences, we installed and used the Laonlp and Underthesea packages. These tools proved to be highly efficient with a complex process to correctly tokenize the words, involving mapping substring of sentence into its own dictionary containing tokenized words that is hard and out of scope of this project. By using these available tools, we ensure the accurate segmentation of language-specific nuances in the sentences.

Despite the efficacy of our chosen tokenization tools, a significant challenge emerged during the tokenization process, especially when dealing with large datasets that we added. The tokenization functions, however, kept all tokenized sentences and functions in the RAM until the entire dataset was processed. This approach led to failures, even on systems equipped with 16GB of RAM on our hardware. As a consequence, we need to find a solution to keep the sentences well-tokenized while not comsume too much RAM usage.

To overcome this memory limitation, we devised a strategic adaptation by creating a separate script for pre-tokenization before importing the data into the Field object. We used the symbol which has the unicode code of U+1019B from Lycian language. Nowadays, this is an acient language that rarely used in any document (we can't even see it correctly when writing this report on Overleaf). During pre-tokenization, this

symbol was inserted between each tokenized word, creating a format that could be efficiently handled by the Field object like the default process. This innovative approach significantly optimized the tokenization process, enabling smooth and efficient processing even with large datasets, while maintaining the integrity of the tokenized information. Users can toggle this option by add the key ‘pre_tokenize‘ in the configuration file.

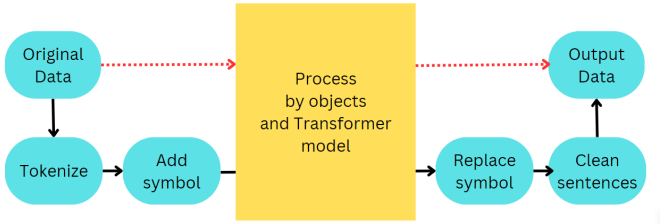

Fig. 7. Data process illustration. Red arrows are the original direction, while black arrows are the new direction of this pipeline.

B. Model improvements

When investigating the insights behind model performance, we recognize the pivotal role played by multi-head attention layers. Because of that reason, we tried to improve the performance more by scaled the number encoder and decoder layers. Despite the evident advantages, the training process was slowdown dramatically which is a notable challenges for us. In fact, training each epoch on default setting just need around one hour to finish, but what happen if we will train a much larger amount of data? On the paper [3] of Hi Liu that we found on the Internet, he add his co-worker introduce a new architecture named "Re-transformer" by changing the scale between Multi-Head Attention layers and Feed Forward layers. Consequently, to strike a balance between computational efficiency and model effectiveness, we changed the Transformer’s architecture that each encoder will has three multi-head attention layers, a substantial increase from the initial single layer.

Beside, the increase the number of d_model and head of multi head attention helped us to fully utilized the power of GPU and captured the features of language better. The more heads of Multi-Head Attention has, the deeper and wider in understanding the aspects in every sentence and therefore the better performance of our model is possible.

Another noteworthy feature we have incorporated into our model is the flexibility to independently determine the number of encoder and decoder layers. Notably, this flexibility allows for a strategic imbalance, where the number of encoder layers can be increased independently of the decoder layers. This deliberate design choice enables us to choose the number of layers to fit in specific task requirements. This configuration proves particularly advantageous in scenarios such as machine translation tasks, where a higher number of encoder layers can contribute to improved performance and efficiency in handling input data.

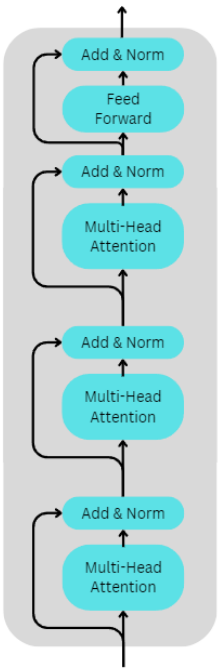

Fig. 8. New encoder layer illustration.

C. Code improvements

In order to update the code fixing some exceptions may rising with new version of Python, we have done the following change:

- Passed the Loader option when reading YAML configuration file to eliminate the warning and ensure the security of our code.
- Fixed the bug when length of tokenized sentence is more than allowed by stripping and reshaping the input shape correctly in both training and evaluating process.
- Add an option to log the training & evaluating process into Wandb. It helps users to visualize the performance of model better by providing real-time chart. There is no need to look carefully at a dark terminal.
- Created a trycatch to catch Exception when calculating the BLEU score. Before that, we have meet an exception that is difficult to reproduce the bug. In fact, it neither raise with every dataset that we passed nor every epoch that we trained. Moreover, model only be saved *after the BLEU score was processed*. Consequently, a mechanism to let the model continue training was implemented.
- Fixed output shape in inference phase. It happened due to a mismatch in initial shape and target shape in the last iteration. As a result, we changed it a little bit to prevent that error.

V. EXPERIMENT

A. Our train strategy

Our model development strategy unfolds in a sequential manner, starting with the training of a baseline model to establish a fundamental performance benchmark and define our initial expectations. After that, we introduce incremental improvements to the model, then testing and evaluating its performance at each stage. Throughout this iterative process, we encounter some technical challenges discussed on above sections that we can't use BLEU score as our primary evaluation metric. To navigate these challenges effectively, we lean the reduction in validation loss value (not the validation loss value itself due to different of components in each improvement iteration) as a reliable and informative guide throughout the model refinement process. This approach allows us to always have a deliverable product.

B. Our environment and accelerate hardware

Embarking on this challenging project, the complexity and resource demands dictated that we shouldn't rely on personal laptops. Instead, we conducted thorough experimentation across various platforms, such as Colab, Kaggle, and even AWS EC2. Each platforms has it own pros and cons:

- Colab is a platform that provide free access to use the NVIDIA T4 for any personal use, as well as option to mount the Google Drive into the working session which make handing file tasks much easier. However, it doesn't provide much details about our usage, and it's easy to be disconnected from the runtime.
- Kaggle provides similar resources with Colab, but with much more clear information. Users can get 30 hours of using GPU each month on each account, which seems to be more generous than Colab. Also, there is a way to "commit" your notebook, and Kaggle will run all cells in this notebook, even when we closed its tab, and keep the output available to view and download. However, we can't mount Google Drive and make any edit on other files, except re-write these files totally. In fact, we usually need to modify minor part in file (e.g. add some line to print the variables' values), and it's impossible on Kaggle.
- AWS EC2 (Elastic Compute Cloud) is a part of Amazon.com's cloud-computing platform, Amazon Web Services (AWS), that allows users to rent virtual computers on which to run their own computer applications. It's a paid service, but we can run the GPU base on our demand, and there is no limit of editing or mounting. In other words, it's like hiring a remote computer which has superior power than our local laptop has.

In overall, each platform offered distinct advantages, enabling us to explore and test the model's capabilities in diverse computing environments. Also, by using AWS EC2, we gained some important knowledge in the domain of clouding computation, which is important domain that many technology companies is involving.

TABLE I
COMPARISON OF DIFFERENT PLATFORMS

Platform	Price	GPU	Mountable	Edit Files
Colab	Free	T4	Yes	Yes
Kaggle	Free	2x T4	No	No
g4dn (AWS EC2)	$0.7/hour	T4	Yes	Yes

C. Training experiments

1) Train the baseline model: In our approach to model development, we initiate the process by training a baseline model to gauge its initial performance and establish clear objectives. For this purpose, we employ a configuration comprising six encoder and decoder layers, utilizing provided data and default tokenization techniques. The training phase is conducted on Kaggle, leveraging the computational power of the NVIDIA P1000 GPU. The entire training process spans approximately nine hours, during which we meticulously log and monitor the model's progress using Wandb, a platform that facilitates efficient experiment tracking and visualization of key metrics. This comprehensive strategy ensures a systematic and data-driven approach to refining and optimizing our model for the desired task. The details of configuration is in below table:

TABLE II
BASELINE MODEL CONFIGURATION

Parameter	Value
Epochs	40
Batch Size	64
D_model	512
Learning Rate (Lr)	0.2
Encoder Layers	6
Decoder Layers	6

2) Train with enhanced model: We extend our model architecture by incorporating additional multi-head attention layers, as previously discussed in the literature. In a parallel effort to isolate and evaluate the specific impact of the multi-head attention mechanism on the model's performance, we reduce the number of encoder and decoder layers to 3. This strategic adjustment allows us to systematically analyze the contribution of multi-head attention to the overall model performance. By maintaining control over other parameters and selectively modifying the attention layers, our experimentation aims to provide insights into the effectiveness and influence of multi-head attention within the broader context of our model. Colab is used to experiment and ensure error-free on the entire process, while Kaggle is used to train this version of model.

3) Train with full improvement features: After achieving initial success with our model, we proceeded to enhance its capabilities by incorporating tokenization and integrating our extensive dataset into the training process. Given the computationally intensive nature of this augmented training regimen, with each epoch requiring approximately 2.5 hours to complete, we opted for the robust computational resources offered by the g4dn instance on AWS EC2. This decision was

TABLE III
ENHANCED MODEL CONFIGURATION

Parameter	Value
Epochs	40
Batch Size	64
D_model	1024
Learning Rate (Lr)	0.2
Encoder Layers	3
Decoder Layers	3

motivated by a desire to mitigate the risk of errors stemming from unstable runtimes and to ensure a smooth and reliable training process. Additional encoder layer also be added to focusing on source sentence understanding, while input and output size of Multi-Head Attention is reduced to fit into GPU provided.

TABLE IV
FINAL MODEL CONFIGURATION

Parameter	Value
Epochs	20
Batch Size	64
D_model	512
Learning Rate (Lr)	0.2
Encoder Layers	4
Decoder Layers	3

VI. EVALUATION AND METRIC

A. Metric

1) LabelSmoothingLoss: Label Smoothing Loss is a regularization technique used in machine learning, particularly in the context of classification tasks. Its purpose is to prevent a model from becoming too confident or overfitting to the training data. Label smoothing works by introducing a small amount of uncertainty or "smoothing" into the true labels during the training process.

In a standard classification setup, when training a model, the target labels are usually one-hot encoded. For example, in a classification task with three classes, a true label for a particular example might look like [0, 1, 0]. The model is trained to predict this one-hot encoded label.

Label smoothing, on the other hand, modifies the true labels to be a combination of the original one-hot encoded label and a uniform distribution over all classes. This is done to introduce some level of uncertainty into the training process. Instead of having a hard 1 for the true class and 0 for other classes, label smoothing might assign, for example, a value of 0.9 to the true class and distribute the remaining 0.1 equally among other classes.

The idea is that by providing a less deterministic target for the model during training, it prevents the model from fitting the training data too closely and encourages it to learn more robust and generalizable features.

In the model, it utilize the PyTorch implementation of Label Smoothing Loss. It takes the model's predictions and the true labels, applies log softmax to the predictions, creates a smoothed target distribution, and calculates the loss based on the negative log likelihood between the predicted and smoothed distributions. This helps in training models that are more robust and less prone to overfitting.

$$\text{BLEU} = \text{BP} \cdot \exp\left(\sum_{n=1}^{N} w_n \cdot \log p_n\right)$$

Fig. 9. BLEU Score Formula

2) BLEU score: The BLEU (Bilingual Evaluation Understudy) score is a widely employed metric in the field of natural language processing and machine translation to quantitatively assess the quality of generated text. This metric, introduced by Kishore Papineni and his colleagues, offers a numerical evaluation of the similarity between a machine-generated translation and one or more human reference translations. BLEU operates by comparing n-grams (contiguous sequences of n items, usually words) between the candidate and reference translations.

In our evaluation process, we leverage the 1-gram to 4-gram BLEU scores, calculating their averages to offer a nuanced and holistic appraisal of our model's translation performance. The utilization of multiple n-grams, such as 1-gram, 2-gram, 3-gram, and 4-gram, enables BLEU to capture varying degrees of linguistic complexity and context. The score for each n-gram precision is computed, and an overall BLEU score is obtained by combining these precision scores through a weighted geometric mean. This multi-gram approach enhances the robustness of BLEU, allowing it to account for both local and global coherence in the generated text. The resulting average BLEU score serves as a valuable and interpretable measure of the model's linguistic proficiency, with higher scores indicating a closer alignment with human reference translations. Its popularity in the evaluation of machine translation models stems from its simplicity, efficiency, and ability to provide a comprehensive assessment across different levels of linguistic granularity.

B. Results

After numerous training attempts, we achieved some results with each version of model. Every record was publicly saved by Wandb.

1) Baseline model: In the baseline model, the BLEU score exhibits a gradual increase, accompanied by some fluctuations during the initial epochs. However, as the training progresses, the model's performance stabilizes, demonstrating a more consistent trend up to the 40th epoch. Ultimately, the baseline model achieves a BLEU score of 21.14, indicating a notable improvement in translation quality over the course of training. The observed fluctuations in the early epochs are a common characteristic of training dynamics, often reflecting the model's adjustment to the complexities of the task at hand.

Due to this result, we set 21.14 as our baseline model score, and our target is to surpass this score as much as we can.

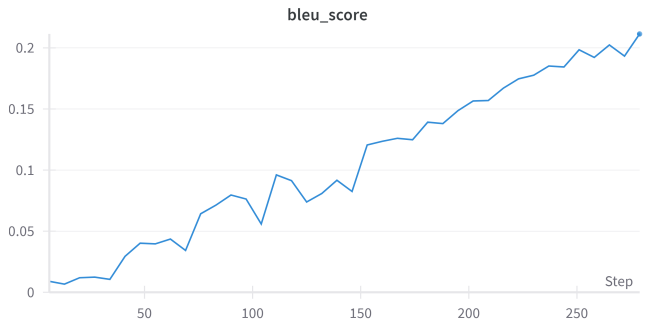

Fig. 10. Chart showing the BLEU score on baseline model.

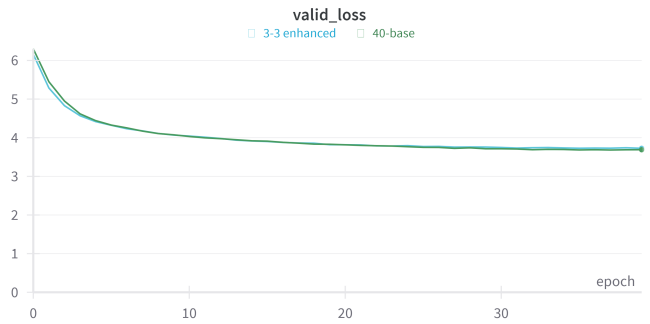

Fig. 12. Compare the valid score between two models.

2) Enhanced model: Upon enhancing the encoder layers, the enhanced model showed a slight enhancement compared to the baseline model while maintaining a comparable training time per epoch. Notably, the BLEU score of this refined model occasionally aligns with that of the baseline model during certain epochs in the middle of the training process. This intriguing phenomenon suggests that the model, despite its enhancements, undergoes similar challenges and adjustments during specific phases of training. As the training process unfolds, the refined model ultimately attains a BLEU score of 22.43, signifying a noteworthy increase of 6.1% over the baseline. This enhancement demonstrated the effectiveness of the modifications made to the encoder layers in bolstering the model's translation capabilities and overall performance.

In terms of validate loss value, there is no difference between two model at all. It implies that the Label Smoothing Loss may not be helpful in term of determining the how much of the improvement of our model. We will take steps to further investigating my claim in the final model.

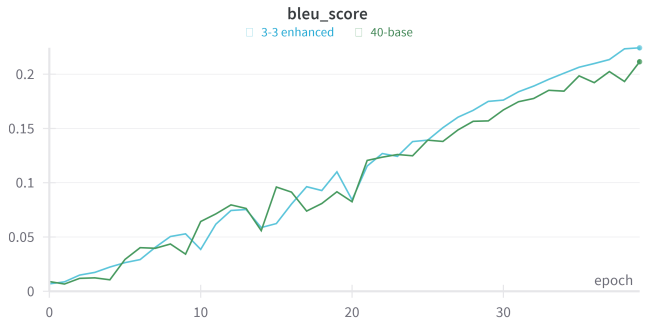

Fig. 11. Compare the BLEU score between two models.

3) Final model: In the final model, several interesting phenomena come to light. One of them is the utilization of tokenization significantly affect positively into the reduction of the loss value compared to our baseline model. This swift decrease in training loss indicates the model's enhanced ability to grasp and adapt to the underlying patterns within the data. However, after the 10th epoch, it seems that the model has seemingly converged, with the training loss plateauing and exhibiting minimal variations thereafter.

An error has been identified in the BLEU score calculation process during the evaluation of our model. Despite attempts to isolate and rectify the issue using a subset of the data, the bug has proven to be elusive and challenging to reproduce consistently. Our investigation revealed that the error surfaced as early as the 4th epoch, and in light of time constraints, we implemented a temporary solution by incorporating a try-catch mechanism. This approach allows the model evaluation to proceed, albeit without the accurate computation of the BLEU score.

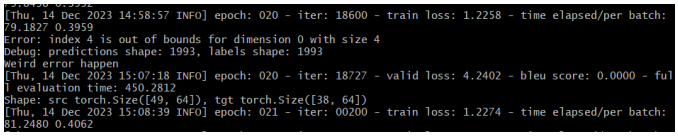

Fig. 13. Envidence of the error when calculating BLEU score.

Also, the validation loss in the final model, which started at a low value, was much higher than other trained models in the 8-10th epoch. Normally, it was an indicator of a failure of the model. To fully evaluate the situation, we decided to calculate the BLEU score in the epoch 10th and see the unexpected result as table VI. Obviously, the different between validation loss values of models is not a reliable indicator as predicted.

Since it took around 2.5 hours for each epoch to looping through 1.6 million sentences, we only can train it within 20 epochs, then do the necessary steps to calculate the BLEU score on that epoch. Notably, we just received the result of 20.7, which is disappointing. However, looking at the test dataset, we figured out that 40% sentences inside that file is poor quality. These poor quality sentences contain many unecessary symbols and have very short length on the average. In fact, we trained our model using data from articles and novels, which have a distribution much differ than this test dataset.

4) Private test evaluation: As required from lecturer, we downloaded the private test, conducted inference, and computed the BLEU score through a meticulous and honest process, adhering to ethical standards. After that, the resulting

TABLE V
BLEU SCORE AT EPOCH 10

Type	BLEU score
Baseline model	6.42
Enhanced model	3.86
Final model	**17.93**

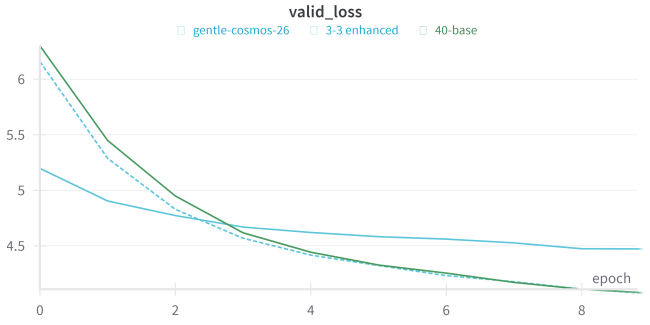

Fig. 14. Compare the valid score in the first 10 epochs.

score of 28.88 reflects the model's performance on this specific test set. In fact, one possible hypothesis was our private test set and the additional data set we incorporated share a commonality that they both comprise Vietnamese articles, exhibiting a similar distribution of data. This alignment in data characteristics enhances the model's adaptability and performance, as the diversity within the training data aligns closely with the patterns present in the private test set. It also demonstrate the effectiveness of our efforts in creating a model capable of successfully handling diverse Vietnamese text, as reflected in the notable BLEU score achieved on the private test set. The result can publicly viewed at here as our proof.

TABLE VI
BLEU SCORE AT PRIVATE TEST

Type	BLEU score
Baseline model	21.14
Final model	**28.88**

VII. CONCLUSION

Our exploration into the intricate components of the baseline model has delved into the nuanced intricacies of data processing, dissecting the placement of attention classes within both the Encoder and Decoder components of the transformer. Having unraveled the inner workings of the model and comprehended its data-handling mechanisms, we strategically devised a comprehensive set of enhancements. These encompassed the implementation of data augmentation techniques, the incorporation of a multi-head attention layer, adjustments to the number of Encoder/Decoder layers, and an overarching refinement of the underlying codebase.

In the pursuit of model improvement, we encountered hurdles, notably during the training phase with the augmented dataset. Despite these challenges, our perseverance paid dividends, leading to a noteworthy and statistically significant enhancement. This transformative progress saw a substantial surge in Bleu scores discerned during the model's inference mode on the test sets. The demonstrable success underscores the efficacy of our proposed improvement strategies, validating their correctness and affirming their positive impact on model performance.

As we celebrate this achievement, we envision a promising future for the model, anticipating its continued evolution to contribute meaningfully to the field of Natural Language Processing (NLP).

VIII. MEMBER CONTRIBUTIONS

- Hoang-Ba Tran: 37.5%
- Quang-Minh Hoang: 37.5%
- Nghia-Hieu Nguyen: 25%

REFERENCES

[1] N. H. M. C. N. V. V. N. T. V. N. P. T. T. H. V. Nguyen Hoang Quan, Nguyen Thanh Dat, "Vinmt: Neural machine translation toolkit," in *https://arxiv.org/abs/2112.15272*, 2022.

[2] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, "Attention is all you need," 2023.

[3] W.-L. C. Huey-Ing Liu, "Re-transformer: A self-attention based model for machine translation," *Procedia Computer Science*, 2023.