CS779 Competition: Machine Translation System for India

Harsh Bihany
210406
{harshb21}@iitk.ac.in
Indian Institute of Technology Kanpur (IIT Kanpur)

Abstract

In this report, I present my approach to the Machine Translation Competition Phase 2. I explored the **transformer** based *Sequence-2-Sequence* model, and made modification in model size, and implementation approach. My common-decoder model outperformed the vanilla transformer models. Our experiments were primarily centered around translating Indic (Bengali, Gujarati, Hindi, Kannada, Malayalam, Tamil, and Telgu) sentences to their corresponding English counterparts. My best model achieved a **BLEU score** of 0.169, placing me 3^{rd} on the final leaderboard.

1 Competition Result

Codalab Username: H_210406
Final leaderboard rank on the test set: 3
charF++ Score wrt to the final rank: 0.396
ROGUE Score wrt to the final rank: 0.434
BLEU Score wrt to the final rank: 0.169

2 Problem Description

Machine Translation System for India (Phase 2), is the projection of a popular *Natural Language Processing* sub-domain of Machine Translation, i.e., translating text from one language to another using computational algorithms, in the Indian context.

The objective of the machine translation task was to develop model(s) of capable of translating *Indic* sentences which includes languages like, Bengali, Gujarati, Hindi, Kannada, Malayalam, Tamil and Telgu to their corresponding *English* counterparts.

Machine translation is an inherently tough task, because of *morphological complexity* present in languages, *variation of scripts* in languages and *semantic preservation* of text amongst other hurdles.

3 Data Analysis

Let's begin by some elementary introduction to the training dataset: `train_data2.json`.

3.1 Dataset size

We have three *Indo-Aryan* languages (Bengali, Gujarati, Hindi), and four *Dravidian* languages (Kannada, Malayalam, Tamil, Telgu).

Language-pair	Number of sentence pairs
English-Bengali	68848
English-Gujarati	47482
English-Hindi	80797
English-Kannada	46794
English-Malayalam	54057
English-Tamil	58361
English-Telgu	44904

Table 1: Size of train dataset

We have the most number of English-Hindi pairs, which is also somewhat expected, because of the vast prevalence of Hindi.

Compared to the previous dataset size in phase-1 of the competition where the task was to translate *English* sentences to their *Indic* counterparts, the dataset size is almost identical.

3.2 Length distribution

Plots shown are made after removing the absolute outliers. Another note is the usage of *Byte-pair encoding Sennrich et al. [2016]*, a popular subword-tokenization scheme to tokenize sentences and words into fixed sized vocabularies. The hyperparameter chosen for the vocabulary size was $32,000$.

3.2.1 Hindi

Consider the English-Hindi length of sentence distribution (*the length of sentence is the number of tokens it is divided into*; previous tokenizations [BPE] hold)

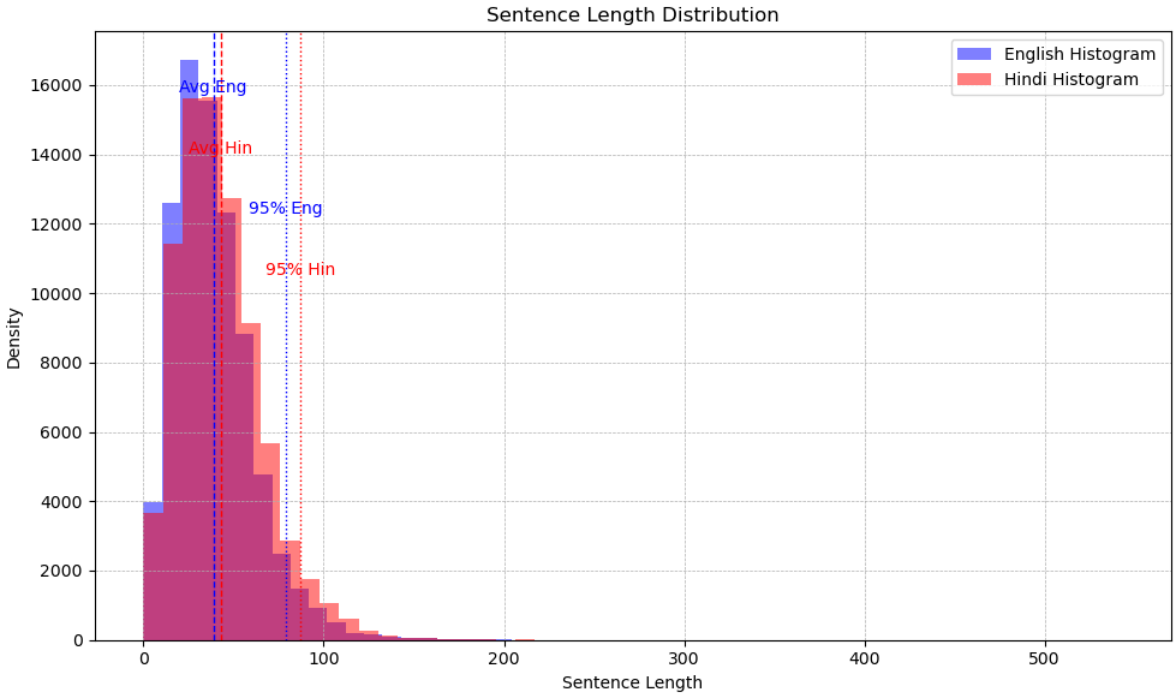

Figure 1: English-Hindi length of sentences

Cannot be compared to previous phase-1's length distribution data, as the tokenization scheme remains drastically different.

	English	Hindi
Average	38.78	42.79
95-percentile	79	87
Maximum	2025	613

Table 2: Statistics regarding length of English-Hindi pairs of sentences

	English	Tamil
Average	25.13	63.07
95-percentile	51	126
Maximum	183	398

Table 3: Statistics regarding length of English-Tamil pairs of sentences

3.2.2 Tamil

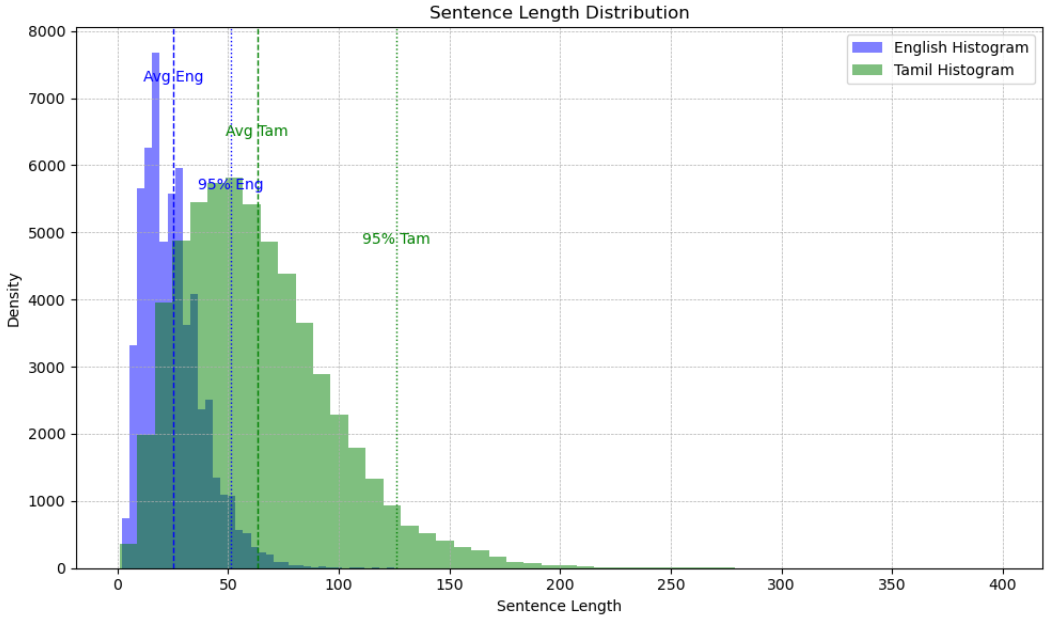

Figure 2: English-Tamil length of sentences

The following observation which can be made is that in terms of bytes of space a sentence takes, English is much more concise than Tamil. Also there are not any major outlier in the dataset, which was the case in phase-1 dataset.

3.3 Word Cloud

Word clouds Heimerl et al. [2014] are visual representations of text data where the size of each word indicates its frequency or importance, with frequently occurring terms displayed prominently and larger than less common terms. They provide a quick way to discern the most prominent terms in a body of text.

Figure 3 is the Word Cloud for the entire English corpus. The word-cloud has similar word-distribution as the previous phase-1 English-corpus.

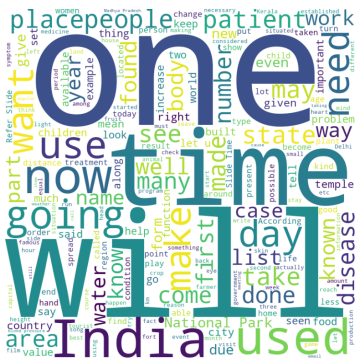

Figure 3: WordCloud for English

4 Model Description

4.1 Implementation detail

The dataset becomes unmanageable, with generic tokenization and processing, because of the massive size of the vocabulary we generate.

To overcome this I have used *Byte-pair encoding* with fixed vocab size which is 32,000, for all the languages, *English* and other *Indic* languages. To implement it I have used **Google**'s support for **sentence-piece** models (Google [2021]).

To create ready hand support for this I had pre-trained and saved the language specific models in `.model` and `.vocab` files that the Google's sentencepiece API generates.

To further use these files to `tokenize` and `detokenize` sentences, I had created a `Language` class that takes in as attribute the location to the `.model` file. The following instances of the class act as *wrapper-class* for the pre-trained sentence-piece models.

Having gained some insights into what machine-translation demands and what are the possible pitfalls and limitations of *auto-regressive recurrent models*, I decided to shift gears and get on with **Transformers Vaswani et al. [2023]** as a sequence-2-sequence model for the task, for the second phase of the competition.

For the model itself, I had implemented the Vaswani et al. [2023] paper from scratch, because things seemed more manageable in it, also to retrieve attention weights to generate attention maps, I get easier access to it. The implementation took inspiration from: hkproj [2023]

For inference I have tried both **greedy decoding** as well as **beam search**, with better results paid off by beam search. Details follow below.

4.2 Model trial 1: Transformer implementation

The first transformer I used was again as previously mentioned and implementation of **Vaswani et al. [2023]** paper.

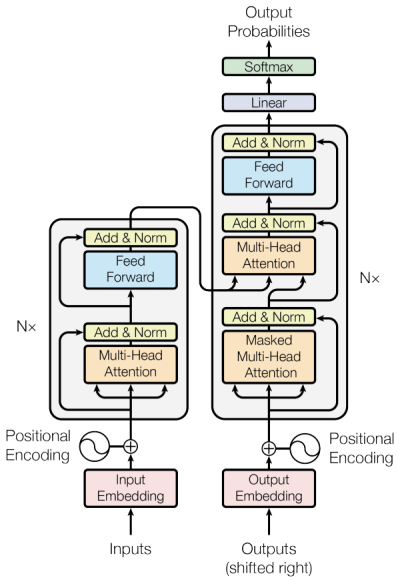

Figure 4: The transformer model from the **Attention is all you need** paper

The model parameters I used were as follows: `D_MODEL = 512`, were this represents the size of the embedding vector. The `NUM_HEADS = 8` and the `NUM_LAYERS = 6`. These parameters are borrowed from the paper's original model size. The feedforward size of the hidden layer in **positionwise feedforward block** was 2048.

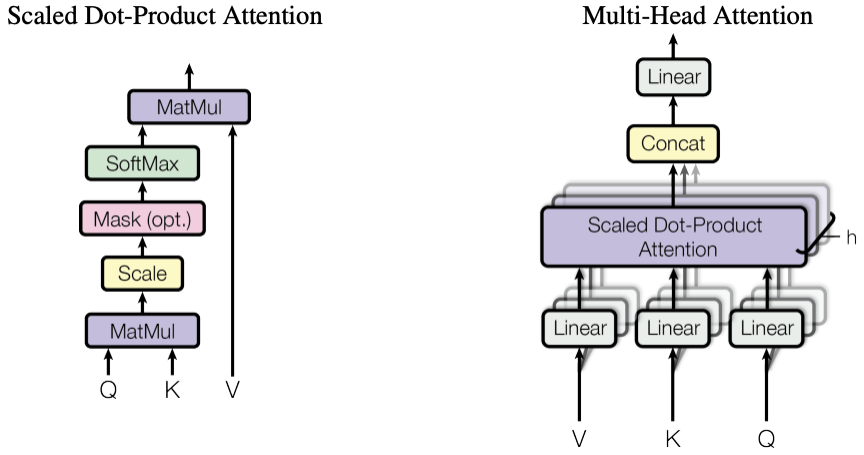

Figure 5: Multi-head attention as presented in the **Attention is all you need** paper

The parameter `NUM_HEADS` represents Figure 5 'h', and the parameter `NUM_LAYERS` represents Figure 4 'N'. In terms of scores, compared to the scores that the recurrent models gave, this was a massive step-up. The sentences generated by the model started to make sense lexicographically and semantically.

The results on greedy and beam search options of inference for the same model are presented later. Also we had separate transformers for each translation pair between the *Indic* languages and the *English* language.

4.3 Model trial 2: Scaling up

The model architecture was exactly the same as before. However I just scaled up the model parameters of `D_MODEL` to 1024 and `D_FF` which is the size of the hidden layer of the positionwise feedforward block to 4096.

To my surprise the scores took a dip (as presented in the table). I stuck with beam search from here on taking the improvement it provided into account.

4.4 Model trial 3: Multi-encoder-single-decoder

The last modification to the model, was not a modification in the model but was a implementation heuristic. The model parameters I used were as follows: `D_MODEL = 256`. The `NUM_HEADS = 8` and the `NUM_LAYERS = 6`. The feedforward size of the hidden layer in **positionwise feedforward block** was 1024. I had a common decoder across all the language pairs, as the common to-translate language was *English*. The model had a language-specific encoders, and language specific output prediction-layer, however the decoder block remained constant.

To be consistent with the task at hand, I decided to pretrain the complete model on the *Hindi to English* dataset, and then froze the decoder block, which was further used for the other language pairs.

5 Experiments

Having shifted to **subword-tokenization** I did not language-specific support from say `spacy` Honnibal and Montani [2017] for *English* or **Indic-NLP** Kunchukuttan [2020] for *Indic* languages.

This is because the *Byte-pair encoding* used is an *unsupervised approach* towards tokenization and lemmatization and do not need the language specific consideration but is based out of the frequency of occurrence of character sets.

The optimizer was the `Adam` optimizer with the following parameters presented:

```
torch.optim.Adam(model.parameters(), lr=0.0001, betas=(0.9, 0.98), eps=1e-9)
```

The models *mask* prevented the model from focusing on the `PAD_TOKEN`. The *sequence length* all the sentences were padded to was 75 (a hyperparameter chosen out off both managing training expectations and choosing a length that encompassed most of the sentences).

The major experimentations were with playing with the **transformer model**, and finally the *common-decoder* model.

6 Results

Model	charF++ score	ROGUE score	BLEU score	Total score
4.2 (greedy)	0.352	0.404	0.143	0.899
4.2 (beam)	0.365	0.409	0.154	0.928
4.3	0.383	0.338	0.137	0.858
4.4	**0.395**	**0.434**	**0.168**	**0.997**

Table 4: Results for various models on the dev set

The transformer architecture is a thing of beauty, and it depicted rightly so here. Compared to recurrent models, these are massive improvements.

My final rank in the **training phase** was **2**.

7 Error Analysis

Attention allows a model to focus on specific parts of the input sequence when producing an output. This is what manifests in a transformer.

Attention map

Attention maps, often visualized as heatmaps, are graphical representations of the attention weights in models that use the attention mechanism, such as in many sequence-to-sequence tasks. In transformers, we have attention three ways, first is the encoder's self-attention, second is the decoder's self-attention and lastly we have encoder-decoder's cross-attention. Following are some attention heat maps.

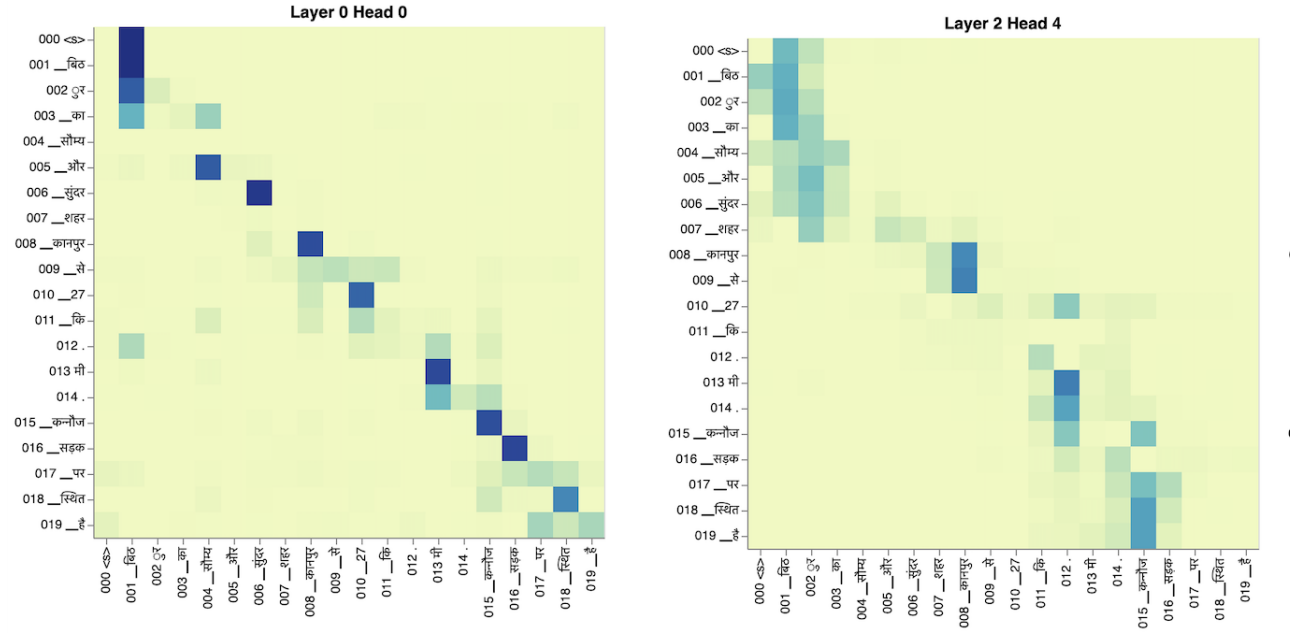

Figure 6: Encoder-self-attention heat maps

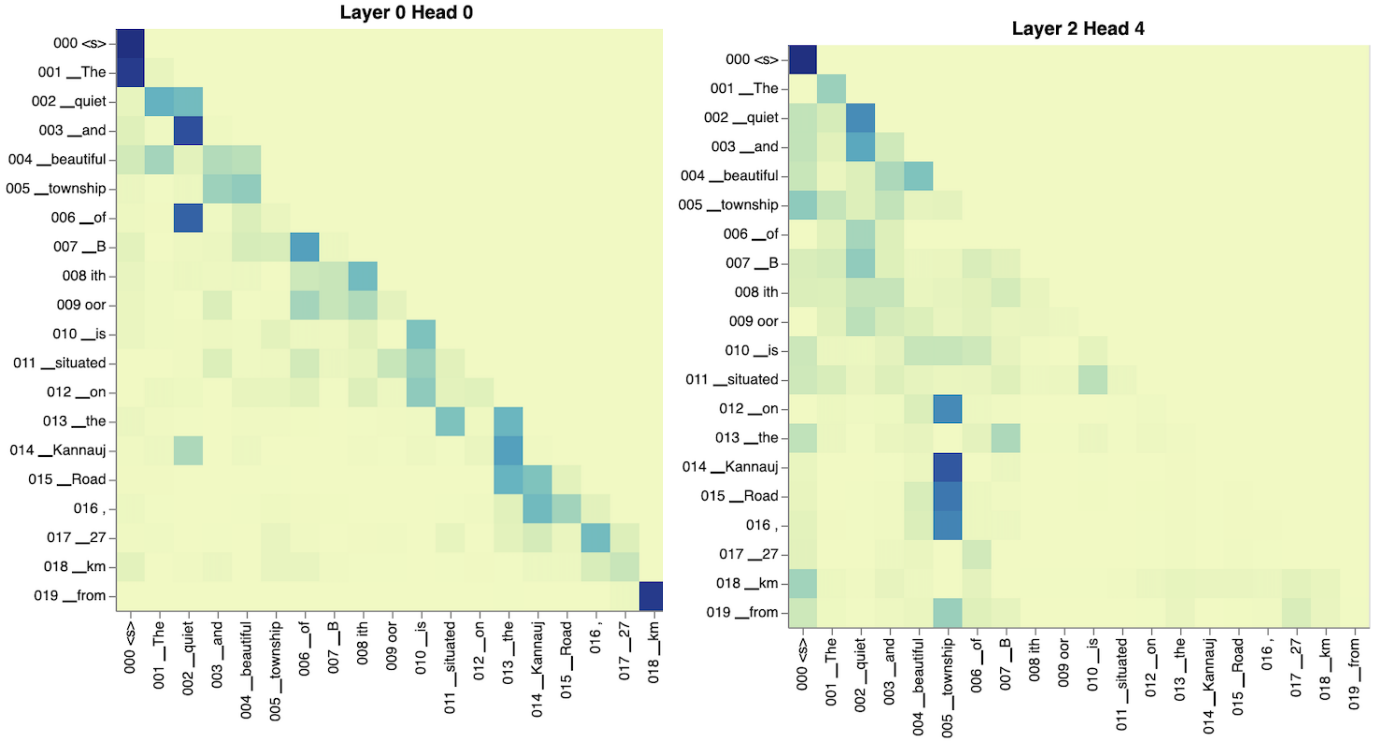

Figure 7: Decoder-self-attention heat maps

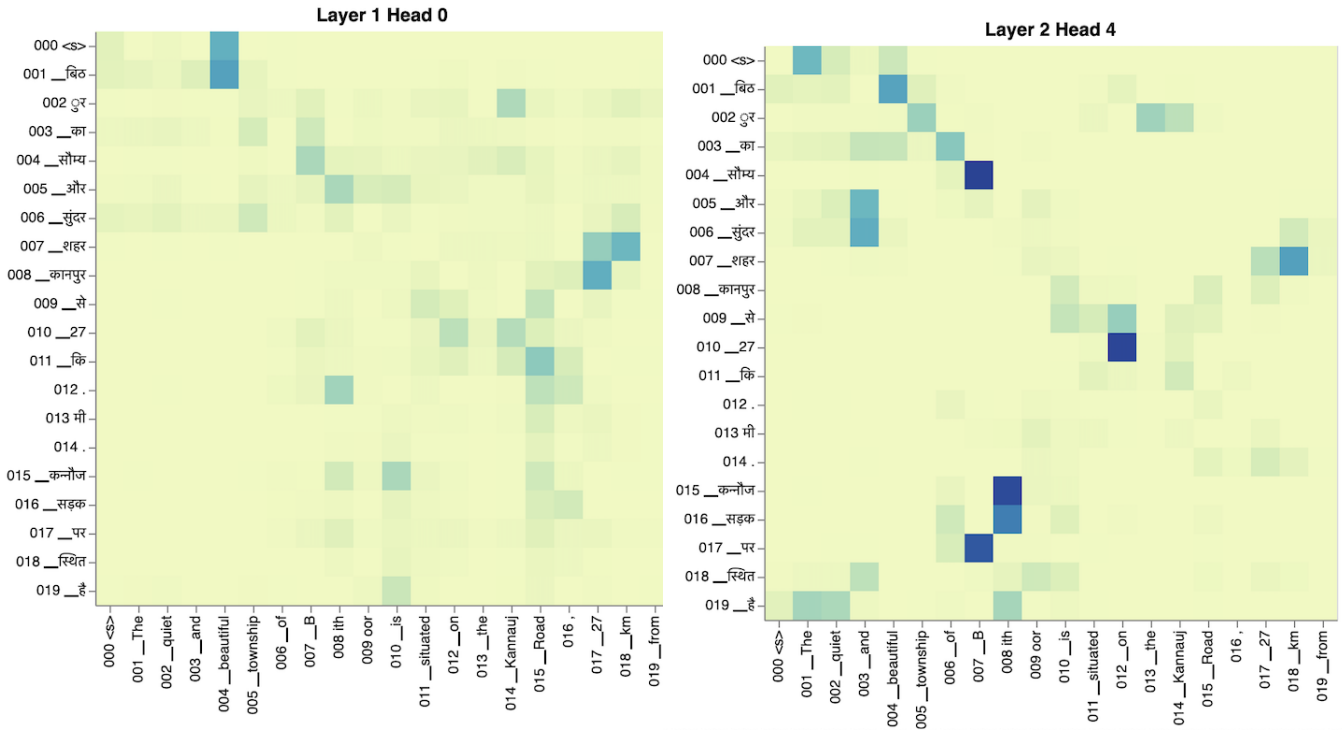

Figure 8: Cross-attention heat maps

We can observe, that the encoder and decoder focuses on the same words more than others, however since there are multiple heads and each learns a slightly different version of attention energies, we have overall a good representation provided by both the encoder and decoder.

On the other hand, the cross attention shows how different words interact. Again multiple heads across multiple layers, provides an overall context that is desirable.

8 Conclusion

Attention mechanism which is a massive improvement over its baseline recurrent networks, and the **transformer architecture** fully manifests the essence of attention. Hence it is highly recommended that one begins by internalizing the attention mechanism and how it works, after the basics of the *encoder-decoder* task is understood, followed by the implementations within the transformer architecture. One important note to take from the assessment apart from the obvious learning into such model architectures and how they work, was to maintain the readability of the code for future reference and access. It proved important to redo and enhance existing code base time and again.

One thing, which I would have done had I some more time, was *hyperparameter tuning* over the model parameters, and size of the model.

References

Google. "sentencepiece", 2021. URL `https://github.com/google/sentencepiece`.

Florian Heimerl, Steffen Lohmann, Simon Lange, and Thomas Ertl. Word cloud explorer: Text analytics based on word clouds. In *2014 47th Hawaii International Conference on System Sciences*, pages 1833–1842, 2014. doi: 10.1109/HICSS.2014.231.

hkproj. pytorch-transformer, 2023. URL `https://github.com/hkproj/pytorch-transformer`.

Matthew Honnibal and Ines Montani. spaCy 2: Natural language understanding with Bloom embeddings, convolutional neural networks and incremental parsing. To appear, 2017.

Anoop Kunchukuttan. The IndicNLP Library. `https://github.com/anoopkunchukuttan/indic_nlp_library/blob/master/docs/indicnlp.pdf`, 2020.

Rico Sennrich, Barry Haddow, and Alexandra Birch. Neural machine translation of rare words with subword units, 2016.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need, 2023.