MINISTRY OF EDUCATION AND TRAINING
CẦN THƠ UNIVERSITY
COLLEGE OF INFORMATION AND COMMUNICATION TECHNOLOGY
DEPARTMENT OF INFORMATION TECHNOLOGY

PROJECT
INFORMATION TECHNOLOGY

Topic

Using Transformer model to translate from English to Vietnamese

Student: Huỳnh Phi Hồng
Code: B2005839
Term: K46

Cần Thơ, XX/2023

MINISTRY OF EDUCATION AND TRAINING
CẦN THƠ UNIVERSITY
COLLEGE OF INFORMATION AND COMMUNICATION TECHNOLOGY
DEPARTMENT OF INFORMATION TECHNOLOGY

PROJECT
INFORMATION TECHNOLOGY

TOPIC

Using Transformer model to translate from English to Vietnamese

Instructor Student Ph.D/M.Si. Lâm Nhựt Khang Huỳnh Phi Hồng Code: B2005738
Term: K46

Cần Thơ, XX/2023

Contents

I. Introduction to Transformer model

Transformer[1] is a deep learning model which was introduced by Google researchers in 2017. It does not compute the input and output representations by using Recurrent Neural Network (RNN) or Concurrent Neural Network (CNN) mechanic but instead using encoder-decoder structure and self-attention mechanism which weights differently for each part of the input data.

The model is mainly used in field of Natural Language Processing (NLP) like for text classification, summarization, translation, text generation, with many models such as Bert[2] ,GPT-3[3] , RoBERTa[4] achieve state-of-the-art results. Another common field for Transformer model is Computer Vision (CV) for tasks like object detection, image classification, video classification.

I.1. General Architecture

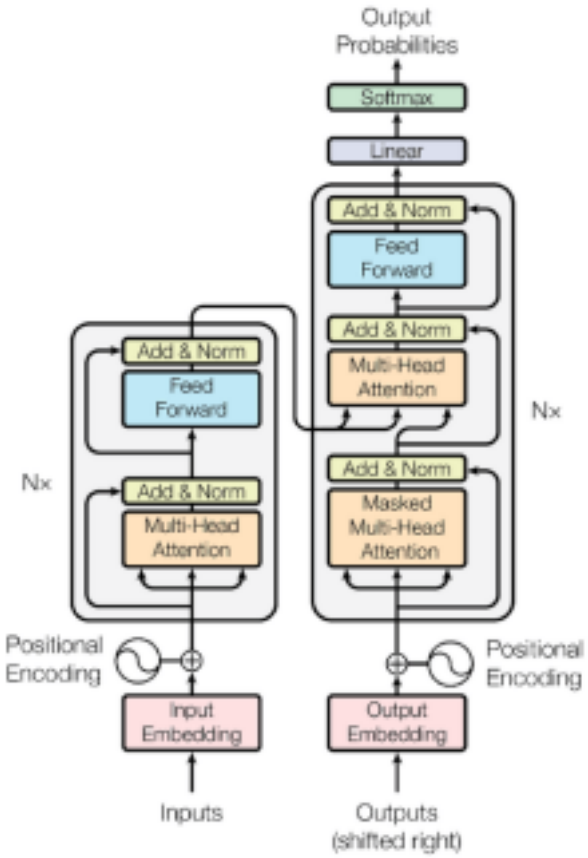

Figure 1 Transformer architecture[5]

The model architecture is generally composed of two components: Encoder and Decoder

- Encoder: It accepts input that represents text and then process them into numerical representation (features) that contains information about which part of the inputs are relevant to each other. The output is then passed into decoder.
- Decoder: It uses the Encoder's representation which incorporated contextual information alongside with its usual sequence input to generate a sequence.

I.2. Input Embedding layer

A word embedding layer can be thought of as a lookup table to grab a learned vector representation of each word. Neural networks learn through numbers so each word maps to a vector with continuous values to represent that word. However, Transformer processes

words in parallel, so with just word embedding the model can't tell where the words are. Therefore, we need a mechanism to inject information about where the words are in the input vector.

I.3. Positional Encoding

The creators of the transformer came up with a solution using sin and cosine functions to obtain information about positions into the input. At even dimension indices the sine formula is applied and at odd dimension indices the cosine formula is applied

$$\text{[illegible]}(\text{[illegible]},2\text{[illegible]})=\text{[illegible]}(\text{[illegible]}/\text{[illegible]}^{2\text{[illegible]}/\text{[illegible]}})$$

$$\text{[illegible]}(\text{[illegible]},2\text{[illegible]}+1)=\text{[illegible]}(\text{[illegible]}/\text{[illegible]}^{2\text{[illegible]}/\text{[illegible]}})$$

Where:

pos: Position of an object in the input sequence

d_{model}: Dimension of the output embedding space

$PE_{(pos,\,j)}$: Position function for mapping a position *pos* in the input sequence to index (k,j) of the positional matrix

n: User-defined scalar, set to 10,000 by the authors.

i: Used for mapping to column indices $0 \leq i < d/2$, with a single value of *i* maps to both sine and cosine functions

I.4. Encoder

Encoder can consist of many similar encoder layers. Each encoder layer consists of two main components: multi head attention and feedforward network. There are also residual connections and layer normalization

- **Multi-Headed Attention**

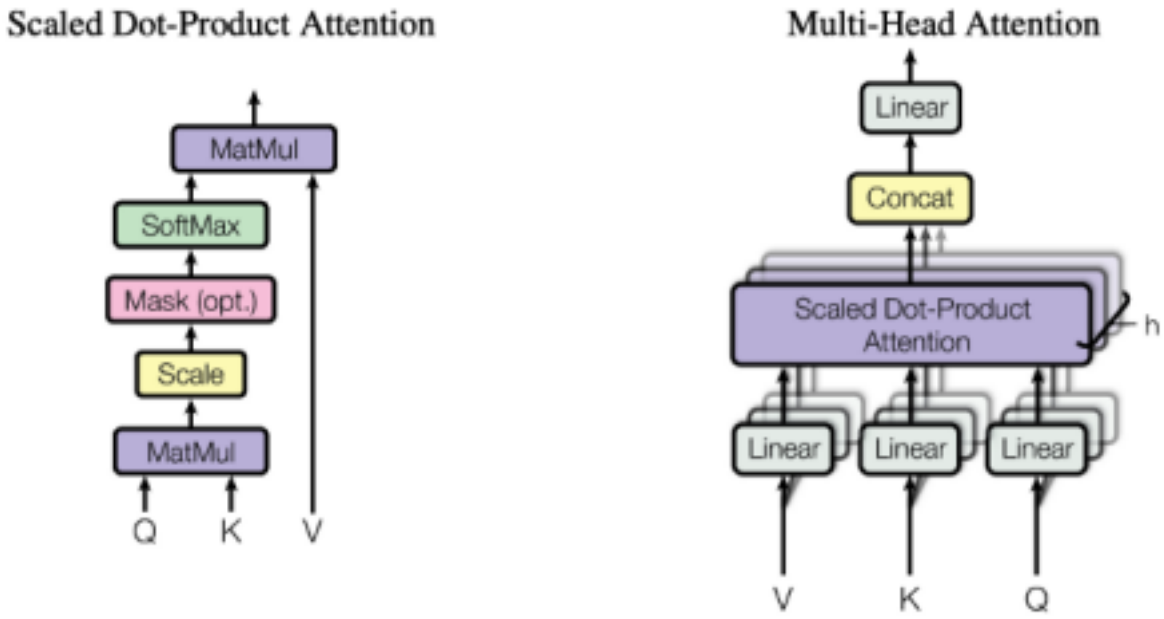

Figure 2 Multi-headed Attention[6]

Multi-headed attention in the encoder applies a specific attention mechanism called self attention. Self-attention allows the models to associate each word in the input, to other words. To achieve self-attention, we feed each word of the input sequence into 3 distinct fully connected layers to create the query, key, and value vectors.

- Query vector: vector used to contain the information of the searched word. It likes the query of google search.
- Key vector: a vector used to represent the information of the words which are compared to the searched word. For example, the website that google will compare with the Keyword that you search.
- Value vector: The vector represents the content, the meaning of the words. It's like the web page content is displayed to the user after the search.

After feeding the query, key, and value vector through a linear layer, the queries and keys go through matrix multiplication to produce a matrix. the scores of the matrix then get scaled down by getting divided by the square root of the dimension of query and key. It then uses the softmax function to normalize the scores within matrix in range 0-1. Finally, the matrix is multiplied by the value matrix generated beforehand to obtain the attention based matrix.

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

= ▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯ (▯▯▯▯^▯▯ √▯▯_▯▯) ▯▯

We want the model to detect various patterns from each word of the input. From each self attention we learned one pattern; Therefore, to learn more pattern, we want to apply more self attention. To do that, we need to split query, key, value into N vectors before applying self attention. Each split is then applied self-attention individually.

- **Residuals Connection and Normalization Layer**

In the architecture of transformer models, residuals connection and normalization layer are used everywhere. those techniques that help the model train faster and prevent the loss of information during the process.

I.5. Decoder

Decoder receive from encoder 2 vectors key and value to decode source sequence to generate another sequence. The architecture of the decoder is very similar to the encoder, except that there is an additional multi head attention in the middle used to learn the relationship between the word being generated and the words in the source sequence.

The process of the masked multi-head attention is similar to that of the regular multi-head attention. However, we need to mask the words from the future that have not been generated by the model. To do this, we simply multiply a vector that contains 0 and 1 values after the scaling of multiplication of matrices Q and K. As a result, the model will only pay attention to all the words before the position of the next generating word.

II. Training model

II.1. Install all necessary libraries

```
pip install transformers datasets evaluate sacrebleu accelerate
sentencepiece pip install sentencepiece
```

II.2. Load dataset

I use the KDE4 dataset[7] with the language pair of en-vi.

```
from datasets import load_dataset

raw_datasets = load_dataset("kde4", lang1="en", lang2="vi",
cache_dir="D:/HuggingFaceCache/")
```

The dataset used has 42782 pairs of sentences.

```
DatasetDict({
    train: Dataset({
        features: ['id', 'translation'],
        num_rows: 42782
    })
})
```

Using the train_test_split method to create the training set and validation set:

```
split_datasets = raw_datasets["train"].train_test_split(train_size=0.9,
seed=20) split_datasets["validation"] = split_datasets.pop("test")
split_datasets
```

```
DatasetDict({
    train: Dataset({
        features: ['id', 'translation'],
        num_rows: 38503
    })
    validation: Dataset({
        features: ['id', 'translation'],
        num_rows: 4279
    })
})
```

II.3. Prepare before begin training model

In this training session I fine-tune the model from en-vi model of Helsinki-NLP[8].
Loading tokenizer:

```
from transformers import AutoTokenizer

model_checkpoint = "Helsinki-NLP/opus-mt-en-vi"
tokenizer = AutoTokenizer.from_pretrained(model_checkpoint, return_tensors="pt",
cache_dir="D:/HuggingFaceCache/")
```

Loading model:

```
from transformers import AutoModelForSeq2SeqLM

model =

AutoModelForSeq2SeqLM.from_pretrained(model_checkpoint)
```

Loading data collator

```
from transformers import DataCollatorForSeq2Seq
```

```
data_collator = DataCollatorForSeq2Seq(tokenizer, model=model)
```

Define preprocess function

```
max_length = 128

def preprocess_function(examples):
    inputs = [ex["en"] for ex in examples["translation"]]
    targets = [ex["vi"] for ex in examples["translation"]]
    model_inputs = tokenizer(
        inputs, text_target=targets, max_length=max_length,
    truncation=True )
    return model_inputs
```

Applying on datasets

```
tokenized_datasets = split_datasets.map(
    preprocess_function,
    batched=True,
    remove_columns=split_datasets["train"].column_names,
)
```

Using SacreBLEU as metric for evaluation:

```
import evaluate

metric = evaluate.load("sacrebleu")
```

Building DataLoader:

```
from torch.utils.data import DataLoader

tokenized_datasets.set_format("torch")
train_dataloader = DataLoader(
    tokenized_datasets["train"],
    shuffle=True,
    collate_fn=data_collator,
    batch_size=8,
)
eval_dataloader = DataLoader(
    tokenized_datasets["validation"], collate_fn=data_collator,
batch_size=8 )
```

Instantiating AdamW optimizer:

```
from transformers import AdamW

optimizer = AdamW(model.parameters(), lr=2e-5)
```

Send all objects to accelerator.prepare() method:

```
from accelerate import Accelerator
accelerator = Accelerator()
model, optimizer, train_dataloader, eval_dataloader =
    accelerator.prepare( model, optimizer, train_dataloader,
    eval_dataloader
)
```

Prepare scheduler and declare number of training epochs

```
from transformers import get_scheduler

num_train_epochs = 10
num_update_steps_per_epoch = len(train_dataloader)
num_training_steps = num_train_epochs * num_update_steps_per_epoch

lr_scheduler = get_scheduler(
    "linear",
    optimizer=optimizer,
    num_warmup_steps=0,
    num_training_steps=num_training_steps,
)
```

Create repository object to upload model to hub

```
from huggingface_hub import Repository, get_full_repo_name

model_name = "kde4-en-vi-test"
repo_name = get_full_repo_name(model_name)
```

Cloning repository in a local folder

```
output_dir = "kde4-en-vi-test"
repo = Repository(output_dir, clone_from=repo_name)
```

II.4. Training

```
def postprocess(predictions, labels):
    predictions = predictions.cpu().numpy()
    labels = labels.cpu().numpy()

    decoded_preds = tokenizer.batch_decode(predictions, skip_special_tokens=True)

    # Replace -100 in the labels as we can't decode them.
    labels = np.where(labels != -100, labels, tokenizer.pad_token_id)
    decoded_labels = tokenizer.batch_decode(labels,
    skip_special_tokens=True)

    # Some simple post-processing
    decoded_preds = [pred.strip() for pred in decoded_preds]
```

```
    decoded_labels = [[label.strip()] for label in
    decoded_labels] return decoded_preds, decoded_labels

from tqdm.auto import tqdm
import torch
progress_bar = tqdm(range(num_training_steps))

for epoch in range(num_train_epochs):
    # Training
    model.train()
    for batch in train_dataloader:
        outputs = model(**batch)
        loss = outputs.loss
        accelerator.backward(loss)

        optimizer.step()
        lr_scheduler.step()
        optimizer.zero_grad()
        progress_bar.update(1)

    # Evaluation
    model.eval()
    for batch in tqdm(eval_dataloader):
        with torch.no_grad():
            generated_tokens = accelerator.unwrap_model(model).generate(
                batch["input_ids"],
                attention_mask=batch["attention_mask"],
                max_length=128,
            )
        labels = batch["labels"]

        # Necessary to pad predictions and labels for being gathered
        generated_tokens = accelerator.pad_across_processes(
            generated_tokens, dim=1, pad_index=tokenizer.pad_token_id
        )
        labels = accelerator.pad_across_processes(labels, dim=1, pad_index=-100)

        predictions_gathered = accelerator.gather(generated_tokens)
        labels_gathered = accelerator.gather(labels)

        decoded_preds, decoded_labels = postprocess(predictions_gathered,
labels_gathered)
        metric.add_batch(predictions=decoded_preds, references=decoded_labels)

    results = metric.compute()
    print(f"epoch {epoch}, BLEU score: {results['score']:.2f}")

    # Save and upload
    accelerator.wait_for_everyone()
    unwrapped_model = accelerator.unwrap_model(model)
    unwrapped_model.save_pretrained(output_dir,
```

```
        save_function=accelerator.save) if accelerator.is_main_process:
            tokenizer.save_pretrained(output_dir)
            repo.push_to_hub(
                commit_message=f"Training in progress epoch {epoch}",
                blocking=False )
```

Results of training:

```
100% 48130/48130 [1:38:14<00:00, 14.01it/s]
100% 535/535 [03:11<00:00, 2.19it/s]
epoch 0, BLEU score: 50.17
100% 535/535 [03:24<00:00, 1.49it/s]
epoch 1, BLEU score: 50.66
100% 535/535 [03:24<00:00, 2.03it/s]
epoch 2, BLEU score: 50.76
100% 535/535 [03:40<00:00, 1.15it/s]
epoch 3, BLEU score: 49.33
100% 535/535 [03:29<00:00, 2.25it/s]
epoch 4, BLEU score: 51.19
100% 535/535 [03:43<00:00, 1.40it/s]
epoch 5, BLEU score: 49.75
100% 535/535 [03:45<00:00, 2.23it/s]
epoch 6, BLEU score: 47.44
100% 535/535 [03:37<00:00, 1.61it/s]
epoch 7, BLEU score: 49.14
100% 535/535 [03:38<00:00, 2.10it/s]
epoch 8, BLEU score: 48.21
100% 535/535 [03:40<00:00, 1.77it/s]
epoch 9, BLEU score: 48.21
```

Using the fine-tuned model:

```
from transformers import pipeline

translator = pipeline("translation",
model="choidf/kde4-en-vi-test") translator("Here is your bill,
please look it over.")
```

Result:

```
[{'translation_text': 'Đây là hóa đơn của bạn, xin hãy xem lại.'}]
```

III. Conclusion

In conclusion, the model for translating English to Vietnamese has been successfully

trained. The model is fine-tuned from Helsinki-NLP model using MarianMT framework. From the results of evaluation of 10 epochs, we could see that the BLEU metric scores were fluctuating and shown a downward trend (from score of 50.17 in epoch 0 to 48.21 in epoch 9), which could be inferred that it was overfitted to the trained data. This is likely caused by the small size of corpus and too high number of epochs.
Therefore, in the future, we could improve the model by preparing larger dataset, increasing dropout and adding an early stopping.

References:

[1],[5],[6] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. *Attention Is All You Need*. arXiv:1706.03762v5

[2] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. *BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding.* arXiv:1810.04805

[3] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. *Language Models are Few-Shot Learners.* arXiv:2005.14165v4

[4] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. *RoBERTa: A Robustly Optimized BERT Pretraining Approach.* arXiv:1907.11692v1

[7] kde4 · Datasets at Hugging Face: https://huggingface.co/Helsinki-NLP

[8] Helsinki-NLP (Language Technology Research Group at the University of Helsinki) (huggingface.co): https://huggingface.co/Helsinki-NLP