Pytorch Implementation of Transformer

Yangmei Deng*
Department of Computer Science
Old Dominion University
Norfolk, VA 23529
ydeng003@odu.edu
Yuming He
Department of Computer Science
Old Dominion University
Norfolk, VA 23529
yhe004@odu.edu
Mahsa Sharifi
Department of Computer Science
Old Dominion University
Norfolk, VA 23529
mshar004@odu.edu

December 9, 2019

Abstract

This is a PyTorch implementation of the Transformer model in "Attention is All You Need"[1]. Transformer is a sequence to sequence framework utilizes the self-attention mechanism, instead of CNN or RNN, and achieve the state-of-the-art performance on WMT 2014 English-to-German translation task.

1 Background

RNN and CNN are previously used, state of the art methods to solve sequence to sequence problems. Although these methods were amazingly good, they have some limitations, that leads us towards using transformer.

First, they are both very slow. As we know, RNN cannot be parallelized. The input of the next step is the output of the previous step. Second, RNN and CNN both have the long term memory problem. It means that if we have a

*Use footnote for providing further information about author (webpage, alternative address)—*not* for acknowledging funding agencies.

word that refers to another word in some previous sentence, they are not able to understand that.

In transformer, fortunately, because of using Multi Head Attention, all the previous problems will be solved.

2 Transformer

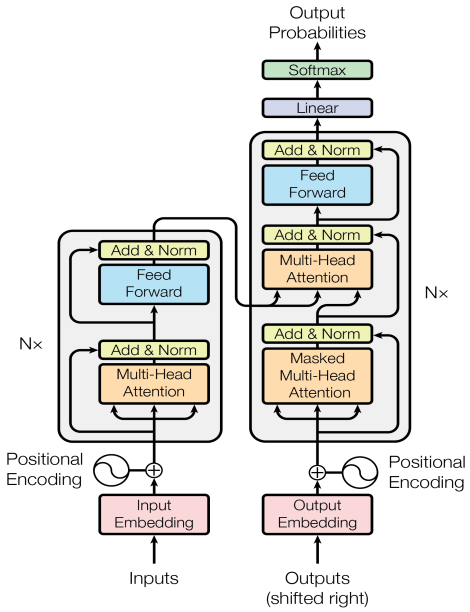

Figure 1: Transformer architecture.[1]

The diagram above shows the overview of Transformer. Similar to other sequence to sequence problems, Transformer has a encoder and decoder. What makes Transformer unique is that Attention mechanism plays an important role in it.

In the encoder, it has one Multi-Head Attention sublayer. In the decoder, it has a Masked Multi-Head Attention sublayer and a Multi-Head Attention layer.

In the Attention mechanism, it will generate a query, key, value for each element of every input sample. And then find the value of all the elements in this input sample that matches each query best. While Multi-Head Attention sublayer enables the model to be more powerful. Each head can find a piece of information of the input and then combine together as the output.

When we implement Transformer, all the calculation of the Multi-Head Attention sublayer would be matrix multiplication. This means we can parallel the calculation. That's why Transformer is faster than previous methods.

2.1 Encoder

The input to the encoder will be the preprocessed English sentence which has transformed each word of an input sample to a number and padded the input sentence to the max_seq_len.

The first thing the Encoder does to the input vector is embed each element of the input sample to a d_model = 512 dimensional vector. And then inject the position information in the output of Embedder. Next it will go through the Encoder sublayers N=6 times.

The output of the Encoder would be the key and value of the Multi-Head Attention layer of Decoder.

2.2 Decoder

The "Outputs" entering the Decoder will be the preprocessed French sentence which did the same process as the input of Encoder.

Similar to the Encoder, it will go through the Embedding and Positional Encoding modules. And then go into the Decoder sublayers.

Before going into the Decoder sublayers, the model will mask the output of Positional Encoding module first. Because when we translate a sentence with Transformer, it will generate output sentence word by word. The first time it goes through the Decoder cycle, it will predict the first word as the output. And then use the predicted words from previous cycles as the input to the Decoder and combine the output of Encoder to predict the next word. The model will keep doing that until it predict the last word. But when we train the model, each time we go through the Decoder cycle, it will assume the first certain words of the target sentence is the prediction from previous cycles. Therefore we need to mask the output of the Positional Encoding module before send it to the Decoder sublayers. The number of elements we need to mask depends on the number of times the model has ran through Decoder cycle.

As in Encoder, the Decoder also goes through the Decoder sublayers N=6 times.

Lastly, the model would do a linear layer on the output of the Decoder and then goes through a softmax layer to get the probability vector of the output word.

3 Implementation

We implemented Transformer using Python programming language and Pytorch framework. Four .py files are used to construct the whole architecture. They are sublayer.py, encoder.py, decoder.py and transformer.py.

3.1 sublayer.py

5 modules, WordEmbedder, PositionalEncoder, Attention, MultiHeadAttention, FeedForward, are included in sublayer.py.

WordEmbedder takes a padded sentence as input, and transfer each word of the sentence to a d_model = 512 dimensional vector. The output of WordEmbedder would be a d_model * max_seq_len 2D vector. PositionalEncoder injects the position information in the output of WordEmbedder.

Module Attention calculates attention using q, k, v generated from the input vector. If Attention is called from Decoder, a mask would be passed to Attention as a parameter.

MultiHeadAttention would split the query, key, and value to 8 heads, and then call Attention to get the output of 8 heads Attention.

FeedForward takes the output of MultiHeadAttention as input. The dimensionality of input and output is the same as the dimension of model.The inner-layer's dimension is 2048 as indicated in the paper

3.2 encoder.py

Encoder calls the WordEmbedder first, and then calls PositionalEncoder. After that Encoder will run through MultiHeadAttention and FeedForward N=6 times.

As indicated in section 2, the output of Encoder would be the key and value of the Multi-Head Attention layer of Decoder.

3.3 decoder.py

There are two classes, DecoderCycle and Decoder, in decoder.py. Each time the model goes through DecoderCycle, it will predict a word. DecoderCycle takes masked output sentence as input. First call WordEmbedder, PositionalEncoder, and then run through masked MultiHeadAttention, MultiHeadAttention and FeedForward N=6 times.

The Decoder class will call DecoderCycle until all words (max_seq_len words) in the sentence are predicted.

3.4 transformer.py

After Encoder and Decoder are constructed, Transformer will call Encoder, Decoder, and then generate two outputs. One is the predicted sentence, the other is a dictionary_size * max_seq_len 2D vector. Each element of this 2D vector shows the probability of each word in the target dictionary.

4 Data

Our models are evaluated on the English-German dataset of the Ninth Workshop on Statistical Machine Translation (WMT 2014). You can find the source file from `https://nlp.stanford.edu/projects/nmt/`. There are totally 4.5 Million sentence pairs in the dataset.

First, we use NLTK package to split, tokenize and normalize the original sentence to lowercase. Then, we use the respective top 50 thousand frequent

words in both languages to label encoding the tokens. All words out of the range 50 thousand are encoded by number 0 and we keep number 1 and number 2 for two special tokens: start of sentence and end of sentence.

As the sentences have different length and our models ask for fixed input dimensions, we use a padding token, 50000, to preprocess all of the encoded sentences to a same fixed length(max_seq_len). Specially, for the target sentences, we add start of sentence token at first and end of sentence token in the end before padding the sentences. Finally, all of the words in the encoded sentences would be represented by a trainable word embedding. An illustration of the process is in Table 1.

Table 1: WMT 2014 English-to-German dataset preprocess (Totally:4.5M sentence pairs)

	English	German
Original pairs	For more information, read our privacy statement.	Mehr Informationen hierzu entnehmen Sie bitte unserer Datenschutzerklärung.
Tokenize (lowercase)	['for', 'more', 'information', ',', 'read', 'our', 'privacy', 'statement', '.']	['mehr', 'informationen', 'hierzu','entnehmen','sie', 'bitte', 'unserer', 'datenschutzerklärung', '.']
Dictionary (TOP:50K)	[13, 53, 126, 4,889, 39, 2483, 1172, 5]	[92, 43467, 3467, 6742, 47, 391, 137, 0, 4]
Padding (Max:80)	[13, 53, 126, 4,889,39, 2483, 1172, 5, X,X,...,X]	[st,92, 43467, 3467, 6742,47, 391, 137, 0, 4,et,X,X,...,X]
Word Embedding length=512	13–> [0.15,0.71,...] 53 –> [0.2...] . . X –> [0.7...]	st –> [0.13,0.92...] 43467–> [0.2...] . . X –> [0.4...]

5 Models

We use Adam Optimizer $\beta_1 = 0.9$, $\beta_2 = 0.98$, $\epsilon = 10^{-9}$ and learning rate updating rule(1) provided in the paper "Attention Is All You Need" to train out model.

$$lrate = d_{model}^{-0.5} * min(step\_num^{-0.5}, step\_num * warmup\_steps^{-1.5}) \quad (1)$$

$warmup\_steps = 4000$ is indicated in the paper. In the first $warmup\_steps$, the steps to update weight would be large, after that the steps will decrease.

The base model trained in the paper used 8 GPUs, had a total of 100K steps. It took 12 hours for the authors of the paper to finish training their base model.

We were unable to train our model with multiple GPUs. So we trained with single GPU. This means it will took us much longer time than it did in the paper. Another thing is we could not train the model with a large batch size. We started with batch_size = 64. But every time we train our model, we got a "CUDA out of memory" error(figure 2) immediately. The HPC department informed us that we had to decrease the batch size to fix the problem. So we end up with batch_size = 4 which makes our training process even slower.

```
RuntimeError: CUDA out of memory. Tried to allocate 14.00 MiB (GPU 0; 15.75 GiB total capacity; 14.61 GiB already allocated; 8.12 MiB free; 364.46 M
iB cached)
```

Figure 2: "CUDA out of memory" error message.

Therefore we could not finish the whole training process within a limited time. We can only train some small epochs, like 3, 5, with a subset of the whole dataset to show that our model works.

5.1 Model 1

We use 100K pairs of sentences, and split it into training, validation, testing dataset to train our first model with 3 epochs. The warmup_steps we used is 4000 which is the same as it is indicated in the paper. 1 day and 20 hours were used to train the model. The BLEU score, which is used in the paper to evaluate model, we get for this model is 3.64. Figure 3 is a sample prediction of Model 1. We can see that we can predict the length of the sentence but failed to predict any word in the sentence.

```
tensor([    1,     0,     0,     0,     0,     0,     0,     0,     0,     0,
            0,     0,     0,     0,     0,     0,     0,     0,     0,     0,
            0,     0,     0,     0,     0, 50000, 50000, 50000, 50000, 50000,
        50000, 50000, 50000, 50000, 50000, 50000, 50000, 50000, 50000, 50000,
        50000, 50000, 50000, 50000, 50000, 50000, 50000, 50000, 50000, 50000,
        50000, 50000, 50000, 50000, 50000, 50000, 50000, 50000, 50000, 50000,
        50000, 50000, 50000, 50000, 50000, 50000, 50000, 50000, 50000, 50000,
        50000, 50000, 50000, 50000, 50000, 50000, 50000, 50000, 50000, 50000])
```

Figure 3: Sample prediction of Model 1

Figure 4 shows the loss of train and validation dataset. There is a loss decrease for training dataset, but not an obvious decrease for validation set.

Figure 5 shows the loss of the first 200 batches of the first epoch of training dataset. The loss is stable for the first 50 batches, and then it drops down dramatically, kept fluctuating from batch 75.

Four factors might affect this model are: too many padding in the target sentences; comparing to vocabulary dictionary(size 50K), sample(100K) is limit; batch size is too small which will create a lot of noise; step size is too big which might skip the minimum point. Four models were trained to validate these factors.

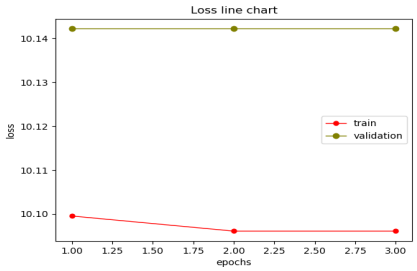

Figure 4: Loss line chart of train and validation set for Model 1

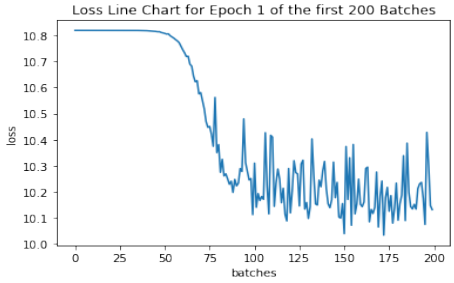

Figure 5: Loss line chart of first 200 batches of the first epoch for Model 1

5.2 Model 2

Model 2 used the same dataset as Model 1 but updated the weight after every 10 mini batches(size 4). Prediction of Model 2 is similar to Model 1, only predicts noise and padding. BLEU for Model 2 is 3.84 which is higher than Model 1. This means Model 2 predict the length of the sentence more accurate than Model 1.

Figure 6 shows the loss line chart of the first 200 batches. It is clear that updating weight after several mini batches can avoid noise for training.

5.3 Model 3

Model 3 increases the training dataset to 300K samples. We couldn't finish 1 epoch after 5 days training, and got the "CUDA out of memory" error again. But from Figure 7 we can see that the lowest loss of the first 300 batches is smaller than previous models

5.4 Model 4

In the paper, the author used $warmup\_steps = 4000$ to train their base model with total training steps = 100K. This means $warmup\_steps$ is only 0.04% of the whole training steps. While our model only train 3 or 5 epochs with 100K data, therefore the total training steps are 7500 or 12500. So 4000 takes a big

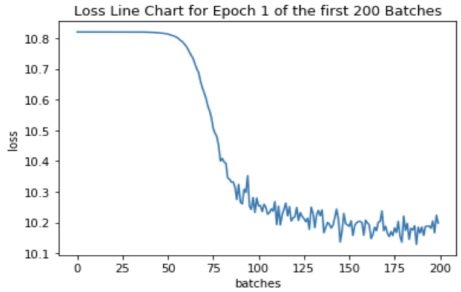

Figure 6: Loss line chart of first 200 batches of the first epoch for Model 2

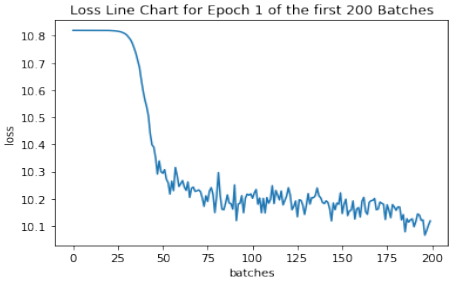

Figure 7: Loss line chart of first 200 batches of the first epoch for Model 3

proportion of our training process. So we decrease the *warmup_steps* to 0.04% of the total training step to train a model with 5 epochs use the same data as Model 1.

BLEU score for Model 4 is 3.65. Figure 8 is a sample prediction of Model 4. Now we can redict the length of the sentence and provide a end of sentence sign.

```
tensor([[    1,     0,     0,     0,     0,     0,     0,     0,     0,     0,
             0,     0,     0,     0,     0,     0,     0,     0,     0,     0,
             0,     0,     0,     0,     0,     2, 50000, 50000, 50000, 50000,
         50000, 50000, 50000, 50000, 50000, 50000, 50000, 50000, 50000, 50000,
         50000, 50000, 50000, 50000, 50000, 50000, 50000, 50000, 50000, 50000,
         50000, 50000, 50000, 50000, 50000, 50000, 50000, 50000, 50000, 50000,
         50000, 50000, 50000, 50000, 50000, 50000, 50000, 50000, 50000, 50000,
         50000, 50000, 50000, 50000, 50000, 50000, 50000, 50000, 50000, 50000]])
```

Figure 8: Sample prediction of Model 4, 2 in the red square is end of sentence sign.

Figure 9 shows the loss line chart. The loss decreases start from about 20 batches which means it can learn the pattern of the sentence much faster.

5.5 Model 5

Until now we are still not able to predict any word of the sentence. Therefore, too many padding might be the most important factor that affect the model. To solve this problem, we divide the dataset to 6 subsets based on the length

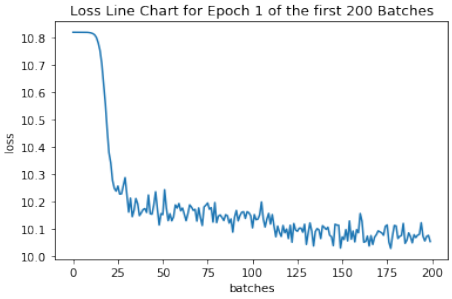

Figure 9: Loss line chart of first 200 batches of the first epoch for Model 4

of the sentence. The length for each subset is <10, 10-20, 20-30, 30-40, 40-60, 60-80, respectively. We train a model for each case with 5 epochs using 100K data. We name models for the six cases as Model 5.1, 5.2, etc.

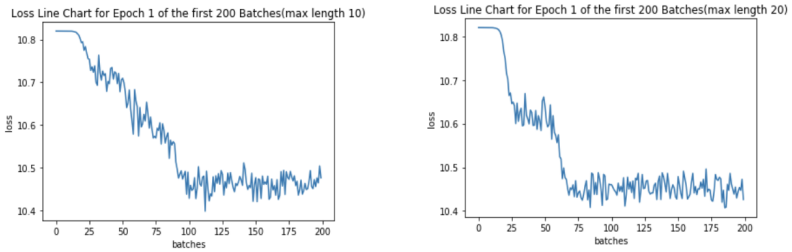

Figure 10: Loss line chart of Model 5.1 and Model 5.2

Figure 10 shows the loss line chart for Model 5.1 and Model 5.2. The pattern is different from Model 1 to Model 4. This is because we don't have that much padding now. So the model will learn the padding pattern of the sentence gradually.

We finished training for Model 5.1 to Model 5.4, the BLEU we get are 6.09, 8.38, 5.41, 6.09. The BLEU for Model 5 is much better than previous models. In Model 5.1 and Model 5.2, we can predict 0, 1, 2, 4, 50000 which is better than previous models, too. But for Model 5.3 and Model 5.4, we still can only predict 0, 1, 50000. We believe if we increase the size of the training dataset and the training epochs, we can get a better prediction.

6 Conclusion

Following are our conclusion from this project. Accumulate gradients help avoid noise. Training dataset size matters to the loss. Correct warmup steps help model decrease loss dramatically. Divide dataset based on the length of sentence solve the too much padding problem.

References

[1] Vaswani A., Shazeer N. & Parmar N., et al. Attention Is All You Need. In: *NIPS.* ; 2017.doi:10.1017/S0952523813000308