RE-IMPLEMENTATION OF MUSIC TRANSFORMER: GENERATING MUSIC WITH LONG TERM STRUCTURE

Zheng Cong Koh, Jin Hong Yong & Jin Yi Yong
School of Electronics and Computer Science
University of Southampton, UK
{zck2g15,jhy3g15,jyy1g15}@ecs.soton.ac.uk

ABSTRACT

A memory efficient music transformer was never before seen until Huang et al. (2019)'s implementation. By re-implementing the music transformer, we will verify the results obtained (NLL loss) for the J.S. Bach Chorales dataset experiment. In this paper, we first outline our model implementation details, and then address missing details of model implementation, choice of hyperparameters, and method of music generation. We assumed a dropout probability of 0.1 and variable learning rate similar to that in Vaswani et al. (2017). 2 random pitches from the vocabulary was used as the first two notes in the generated sequence, and beam search was used for subsequent generations. Generated sequence does not show any long term structure, deviating far from the original implementation.

1 INTRODUCTION

The aim of this paper is to assess the implementation of music-generating transformer model by Huang et al. (2019) through re-implementation[1]. It is intended to reproduce the results achieved under the J.S. Bach Chorales dataset experiment. In this paper we did not reproduce the results for the MAESTRO dataset. The above aim has led to the following core objectives:

1. Implement a Baseline Transformer. Calculate NLL with added positional sinusoids, and with concatenated positional sinusoids as relevant timing information respectively.

2. Implement Transformer with efficient relative-attention. Calculate NLL without both relative pitch and time information and with added positional sinusoids.

1.1 BACKGROUND

The field in music-generation using a transformer model still lacks critical investigation, and Huang et al. (2019) is the first to succeed in its implementation, where complex expressive piano performances were modelled, generating thousands of tokens with exceptional internal consistency using relative self-attention. Besides, Huang et al. (2019) has improved the algorithm from Shaw et al. (2018), dramatically reducing memory requirements from $O(L^2D)$ to $O(LD)$, where L is the sequence length and D is the model's hidden state dimension, making the model memory efficient. In order to produce a coherent musical piece with recurring elements at different levels, a model often needs to reference elements in the past and improvise for variations. For the task, self-attention is a promising method as it allows a model to access once generated outputs during every step of its generation. However, as music consist of elements that are tied to relative differences, such as the extremely important pitch and timing, relative attention is more refined for the task to process the pairwise relations.

[1]Repository for our re-implementation: https://github.com/skynet-comp6248/music-transformer-comp6248.

2 IMPLEMENTATION DETAILS

The original implementation of the paper was done in the Tensor2Tensor framework (Vaswani et al., 2018). However, our re-implementation of the attention mechanism and the transformer model was done using PyTorch 1.0.1. The reason we chose to use PyTorch was to ensure reproducibility as the Tensor2Tensor framework and the music transformer model are from the same organisation (Google Brain). Majority of the code was implemented according to the description in the paper, with some parts of the original transformer model reused from re-implementations available online.

2.1 PROCESSING DATASET AND MODEL OUTPUTS

The J.S. Bach Chorales dataset[2] used for training has been pre-separated into training, validation and testing splits. Additionally, the four-part scores have already been discretized onto a 16th-note grid, therefore each score from the dataset is essentially a numpy array of shape (N,4), where N is number of time steps in the score, and 4 is the number of columns to represent the four voices soprano (S), alto (A), tenor (T) and bass (B). Unfortunately none have been mentioned in the paper regarding the vocabulary of this dataset. However from the dataset, we observe that pitches of voices vary between values $36 - 81$, indicating 46 available pitches, with addition of NaN as a pitch for silent voices. For our model to learn when the voice is silent, pitch value 0 is used instead of NaN.

Using the description provided in the paper, we serialise the grid into a sequence in the form of $S_1A_1T_1B_1S_2A_2T_2B_2$..., where the subscript gives the time step. As the most common sequence length in the dataset is 1024 and since there is no mention on the length of token sequences used for training, we thus assume that all sequences fed into the transformer model have a sequence length of 1024. This is done by cutting the scores up to length of 1024 for longer scores and pad shorter scores using pad token (pitch value of 1). Therefore, the total size of the vocabulary becomes 48.

As there isn't any mention on what the target outputs of the model should be, we then assumed the target output to be the score starting from the second pitch, i.e. $A_1T_1B_1S_2A_2T_2B_2$.... This means that given the first note it should predict what the second note should be and so on.

Before the sequences can be used for training, the values from the original vocabulary of $[0, 1, 36, 37, ..., 81]$ needs to be converted to a different vocabulary of $[0, 1, 2, 3, ..., 46, 47]$ because the embedding layers (`torch.nn.Embedding`) in the transformer model requires input tensors containing the indices to extract, which can only vary from 0 to the size of the vocabulary (48).

2.2 BASELINE TRANSFORMER

We build up the baseline transformer model which acts as a wrapper module containing other modules, which includes the encoder, decoder and the final dense layer. The following two subsections discusses the implementation assumptions that we've made due to lack of details from the paper.

2.2.1 CONCATENATING POSITIONAL SINUSOID

In the published paper the authors explored that by concatenating instead of adding the positional sinusoids to the input embedding, they are able to improve the performance as the model can more directly learn its absolute positional mapping. We re-implemented this by using `torch.cat()` on the embedding layer output and the positional encodings, with both having the same dimension D to produce an output tensor with the dimension $2D$.

2.2.2 LAYER NORMALISATION IN ENCODER AND DECODER LAYER

The published paper states that implementation of transformer models is based on Vaswani et al. (2017)'s implementation. However, Vaswani et al. (2017) did not explicitly mention both the normalisation of input embeddings to the encoder layer and output embeddings to the decoder layer. A normalisation layer before both the input and output embeddings was found in the implementation of the Transformer model in Tensor2Tensor framework[3], possibly because Vaswani et al. (2017) found

[2]JSB Chorales dataset: `https://github.com/czhuang/JSB-Chorales-dataset`

[3]Tensor2Tensor's implementation of Transformer model: `https://github.com/tensorflow/tensor2tensor/blob/master/tensor2tensor/models/transformer.py`

improvement to the model in doing so. Therefore, our work has assumed the use of normalisation layer in the Transformer models and included it in our implementation.

2.3 TF WITH MEMORY EFFICIENT IMPLEMENTATION OF RELATIVE ATTENTION

It was mentioned in Huang et al. (2018) that the relative attention functionality is already available in the Tensor2Tensor framework. We re-implemented the relative attention functionality by porting Huang et al. (2019)'s implementation of relative attention in Tensor2Tensor to PyTorch.

Steps for retrieving relative position embeddings $E^{r\top}$ are implemented according to `get_relative_embeddings_left`[4] function in Tensor2Tensor framework. The "skewing" procedure to convert the original absolute-by-relative indexed matrix of $QE^{r\top}$ into an absolute-by-absolute indexed matrix is implemented the same way it was done in the Tensor2Tensor framework's `_absolute_position_to_relative_position_masked` function. However, to replicate the padding steps we adapted `torch.nn.Functional.pad()` to replicate TensorFlow's `tf.pad()` as both functions are slightly different. As for slicing, we looked at how tensors were sliced in `_absolute_position_to_relative_position_masked` while referring to Huang et al. (2019)'s paper, and sliced the tensors with `ndarray` slicing.

2.4 CHOICE OF HYPERPARAMETERS

Query and key channel size (att), hidden size (hs), feed-forward hidden size (ff), hidden layers (L), and number of heads (h) were set according to Table 2 in Huang et al. (2019)'s paper. However, the learning rate for the model was not specified for this dataset and has to be determined from Vaswani et al. (2017)'s Attention Is All You Need paper according to the published paper. We implemented variable learning rate mentioned in Vaswani et al. (2017)'s paper, with the same 4000 warm up steps. In addition to this, the drop-out probability is not mentioned, and we assumed a value of 0.1, same as that used for Vaswani et al. (2017)'s Transformer model.

2.5 MUSIC GENERATION

With the trained model weights, we can then use the model to generate music to verify the implementation. As the paper did not provide any details on the method of generating music for unconditioned samples (without priming sequences), we made multiple assumptions to accomplish it.

First, we used 2 random pitches from the vocabulary as the first two notes in the generated sequence. Next, beam search with a beam width of $k = 3$ is performed and the generated output with largest probability is chosen. At each time-step, beams kept from previous time-step are extended by having the model predict the next output based on previous beams. Beams with the best scores are kept and the algorithm is repeated until combined generated output reaches a length of 1024. This length is chosen as it is the same length for the sequences used to train the model. The scores for each output are calculated using the log probability of the sequence generated thus far.

At each time step during generation, the generated sequence up to that time step is fed back into both the encoder and decoder of the model as input and target respectively. The encoder's output is passed on to the decoder of the model, then finally moving on to the dense layer.

After the generated sequence is obtained, we then revert the index values back to the original pitch before re-shaping the array back to the same shape as the dataset. The array is then converted to a `NoteSequence()` using Magenta to be plotted and listened[5].

3 RESULTS & ANALYSIS

From Figure 1 and 2, training loss for baselineTF decreases as number of epochs increases, whereas validation loss decreases initially, but keeps increasing indefinitely afterwards, implying overfitting.

[4]Relative attention functionality (dot_product_self_attention_relative_v2): `https://github.com/tensorflow/tensor2tensor/blob/master/tensor2tensor/layers/common_attention.py`

[5]Check the `gen_sequence_audio.ipynb` and `gen_train_audio.ipynb` files in the repository.

Both of baselineTF + concat and RelativeTF models exhibit similar patterns for both behaviour of training loss and validation loss. Training loss decreases as number of epochs increases, but leverages off at a relatively high ($\approx$ 3.0) NLL loss. On the other hand, validation loss fluctuates over the course of training, with RelativeTF model having higher variance in fluctuation. Further investigation on why such fluctuations occur is needed, as there are various reasons such as a biased validation set, insufficient regularisation, unshuffled dataset and more. In general, all three models are not able to reproduce the respective NLL loss in Table 2 of Huang et al. (2019)'s paper. The lack of information on implementation details, choice of hyperparameters such as the learning and drop-out rate, and preparation of data might have been the possible causes for the inability to reproduce the desired results.

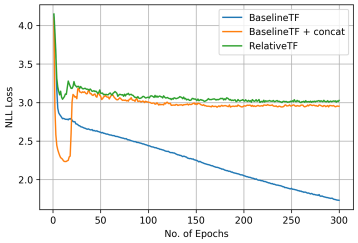

Figure 1: Training loss, 300 epochs

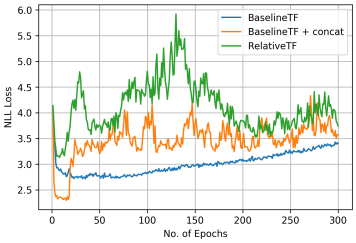

Figure 2: Validation loss, 300 epochs

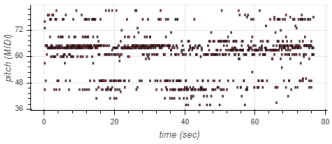

Figure 3: Generated sequence from BaselineTF + concat model

Using the BaselineTF with concatenating positional sinusoid after training for 300 epochs, a generated sequence with a length of 1024 notes is obtained. The generated sequence is plotted and shown in Figure 3.

Looking at Figure 3, it can be observed that the generated output does not contain any pleasant melody nor shows any long term structure, and for the majority of the sequence there are only a few pitches that are played repeatedly. This result is far from what is presented on their generated samples. This suggests that the model did not learn correctly during the training phase.

4 Conclusion

In this re-implementation, we have built our models in PyTorch. We setup our baseline transformer with the assumptions: Performing normalisation before both the input and output embeddings, and concatenating positional sinusoids instead of adding them to the input embedding. Memory efficient relative attention transformer was implemented by referring to the Tensor2Tensor implementation by Huang et al. (2019). Both models were trained with dropout probability of 0.1 and learning rates from Vaswani et al. (2017), and musical sequence is generated using beam search, with 2 random pitches as input to generate the first two notes. However, we failed to replicate the results from Huang et al. (2019) for the J.S. Bach Chorales dataset, with relatively high NLL loss for the models and the sequences do not exhibit long term structure, suggesting that the models have failed to learn correctly during training.

References

Cheng-Zhi Anna Huang, Monica Dinculescu, and Ian Simon. Music transformer: Generating music with long-term structure. Website, 12 2018. URL `https://magenta.tensorflow.org/music-transformer`. last checked: 15.05.2019.

Cheng-Zhi Anna Huang, Ashish Vaswani, et al. Music transformer. In *International Conference on Learning Representations*, 2019.

Peter Shaw, Jakob Uszkoreit, and Ashish Vaswani. Self-attention with relative position representations. *arXiv preprint arXiv:1803.02155*, 2018.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need, 2017.

Ashish Vaswani, Samy Bengio, et al. Tensor2tensor for neural machine translation. *CoRR*, abs/1803.07416, 2018.