A Deep Dive into Music Generation with Modern Deep Learning Architectures

Zhe Rao
MDA, Faculty of Sicence
Western University
London, Canada
zrao6@uwo.ca

Chengfei Xie
MDA, Faculty of Sicence
Western University
London, Canada
cxie73@uwo.ca

***Abstract*—With the numerous breakthroughs in the field of deep learning in the past decade, we were able to study the sophistication of supervised learning to its extreme. Machines or models could outperform humans in many specific tasks, but how was their "creativity"? In this project, we explored two powerful neural network architectures for generating music: Long-Short Term Memory (LSTM) and the revolutionary Transformer architecture. In addition, we experimented with several models with different hyperparameters for each architecture. Moreover, we experimented with two approaches to tokenize the MIDI files: notes played at time steps and in terms of musical events.**

Keywords—music generation, deep learning, LSTM, transformer

I. Introduction

Musical composition is the art of expressing the beauty of music, representing human ingenuity and creativity. Especially for classical compositions, the patterns within the music are dynamic and complex, and only the musical wizards are able to capture it and produce great compositions. However, as a powerful technique to detect hidden patterns and produce new combinations based on existing ideas, deep learning could be used to capture the musical structures.

In this work, we strive to explore the possibilities of advanced neural architectures' creativity in music generation. We primarily focus on developing a deep learning network to generate piano composition. We customize our dataset with MIDI files from classical compositions and popular music. As the music is abstract and has many variations compared to text or images, we experiment with different encoding methodologies, including "one-hot-encoding" and REMI tokenization. Initially, we apply Long-Short Term Memory (LSTM) models to overcome the shortcomings of recurrent neural networks. Then we implement the Transformer architecture with several experimental models that can be properly trained given our limited computing power.

In the rest of the paper, we discuss the related work of music generation in section II. Data processing methods will be explained in section III. In section IV, we will illustrate the architecture of the neural net models, followed by the experimentation and results in section V. Lastly, additional functionalities will be discussed in section VI and potential issues of overfitting in section VII.

II. Related Work

In this section, we present our investigation of predecessors' work on music tokenization mechanisms, MIDI-like, REMI and Compound words.

Previously, Huang et al. proposed a sophisticated tokenization approach called MIDI-like [1]. It creates an event-based representation of the music, including three types of events: Note-On, Note-Off, and Time-Shift. The Note-On event indicates the "action of hitting a specific key of the piano keyboard" [1], while the Note-Off event indicates the release of that key. The Time-Shift event captures the intermediate gap between events. This method provided more flexibility than the strict "one-hot-encode" approach.

Revamped MIDI-derived events (REMI) [1] is an improvement in the work of MIDI-like representation proposed by Huang & Yang. Compared to languages and images, musical compositions have several high-level semantic information about the music included in the MIDI file but are discarded by the MIDI-like representation, such as downbeat, tempo and chord. Huang and Yang revised their previous work with the refined events, including Note-Duration, Bar, Position, Tempo, and Chord. With this representation, the model could compose a more stable rhythm and capture more details in the music, thus enhancing the model's performance on music generation. We adapted this architecture in our work, and a detailed explanation is included in section III A.

Compound word representation (CP) [2] has a similar concept to REMI that encodes the events of the music. However, CP has advantages that the model could compose full-song music instead of minute-long in the case of REMI or any previous methods. As REMI represents the music with two sequences of events, each sequence contains all the events for a single track (i.e., left hand or right hand). The researchers suggested that "different types of tokens may possess different properties" [2], and they should be treated differently. They proposed to first group the events into a super token (e.g., group Pitch, Duration, and Velocity together as one token). Secondly, fill the missing event types with "[ignore]" token. This is something that we look forward to exploring in our near future.

III. Preprocessing

We utilized two tokenization techniques to pre-process the MIDI files.

A. *"One-Hot-Encoding"*

Firstly, we took the "one-hot-encoding" approach, breaking the MIDI files into discrete time steps where each time step is represented by a length-88 tensor (corresponding to the 88 keys on the keyboard). Each element in the tensor will have a value of 1 if that key is being played at the moment and 0 otherwise. This is the inferior tokenization mechanism because it encodes nothing but the time steps into the representation. Information such as tempo, velocity, and structure of the music is all lost. As Motroc once mentioned, "a machine learning model is only as good as the data it is fed" [3]. We suspected that the model would not be able to structure a proper melody. While most music generated is unstable, some melodies sounded unexpectedly good to our surprise.

On the plus side, with this straightforward encoding mechanism, we were able to understand how RNN works, and this method presents us with straightly visualizable music pieces. An example of classical tokenization is as follows.

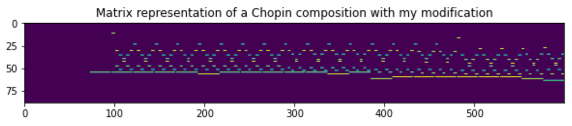

Fig. 1. Visualization of Classical Composition.

The tokenization approach in the following implementation [4] distorted the music after tokenization. The tokenization for the same composition is as follows.

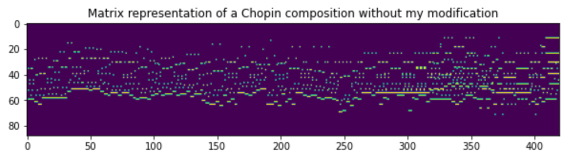

Fig. 2. Visualization of Distorted Classical Composition

Therefore, to ensure that the model is training on the original masterpieces, we used the Pypianoroll package to perform the tokenization without distorting the original data. The code can be found with "Creating Dataset – One Hot.ipynb".

This encoding mechanism, however, requires us to post-process the output sequence. Traditionally, each output would have the size of a number of distinct time steps or events; each element in the vector would be activated with the softmax function, which produces the probability that the note or event is in the next step. However, that process is only suitable for predicting the next note or event, which is not the same as predicting the next time step. To illustrate, while there are only 88 distinct values for notes, there are far more possible distinct values for time steps $\left(i.e., \binom{88}{2}\right)$. For example, there are 3828 possible combinations of playing two keys in a time step. Therefore, we need a different mechanism to calculate the loss so that the model can learn through backpropagation. Our output vector for next-note prediction is length-88. We need to allow multiple elements in the vector to be activated as 1 (representing the keys being played in the next time step). Therefore, we flatten the vector ($i.e.$ $[4,79,-2,\ldots]$), then stack another vector on top of that and compute the second vector as one minus the output vector $\left(i.e.\ \begin{bmatrix}-3 & -78 & \cdots\\ 4 & 79 & \cdots\end{bmatrix}\right)$. Then we obtained a matrix with shape (2, 88). We lastly applied the sigmoid activation to each column of the matrix (88 times) to predict whether the key is being played at the next time step, for each of the 88 keys. Of course, this also requires us to flatten the target sequence.

B. *REMI*

As mentioned above, the "one-hot" approach to encoding the music files cannot capture the structure of the music. Hence resulted in unstructured predicted music (with LSTM), such as:

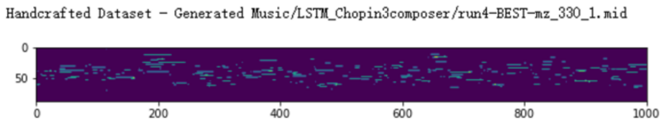

Fig. 3. Unstructured Classical LSTM Composition

Therefore, we wanted to capture the information about the musical structure explicitly. After reading the work of Taiwan AI Lab on Pop Music Transformer (Huang & Yang, 2020), we implemented their tokenization strategy, namely, Revamped MIDI-derived events (REMI). In this approach, the MIDI file is tokenized as events instead of time steps. Events could include: bar event (signals beginning of a music section), position event (indicates which position are the keys being played at), pitch event (indicates what keys to be played), and velocity event (indicates the strength of the keys played). In their paper, they incorporated additional events such as chord events, which indicate the combination of keys that can be played in a specific music section. Although they have open-sourced their code, including the chord extraction algorithm, we realized events were missing until almost the end of our project (i.e. when we tried to combine the tracks for classical compositions). We then decided to leave this to future exploration.

After the musical structure had been explicated and encoded into the input tokens, we observed a more structured composition from the LSTM model:

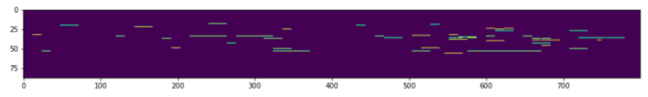

Fig. 4. More Structured Classical LSTM Composition

The difference might seem trivial in terms of visualization. It is much more distinguishable by listening to and comparing the two pieces of music. They have been attached to a zip file called "Generated Music." The first composition is called "LSTM_3composers - run4-BEST-mz_330_1.mid" and the second composition is "PopComplexLSTM - waldstein_1-run1-LAST.midi".

We only trained transformer models with REMI (with pop music instead of classical music) due to the limited computational power available to us. The compositions from the transformer were just incomparably beautiful. (The music is stored as "PopTransformer-run-2-001-BEST_Best")

0
50
0
200
400
600
800

Fig. 5. Pop Transformer Composition

However, the composition of the transformer is not exactly comparable to that of LSTM because of the difference in how we used the model to predict events. This will be discussed in detail in the Additional Functionality section of the report.

It is worth mentioning that although the REMI encoding mechanism explicitly provided the model with structural information and hence the produced music has better structure, and it weakened the model's ability to generate "stable" music with proportional length. For example, 800 events might be converted into roughly 30 seconds of music. However, 800 events predicted by the transformer model might be converted into the music of 15 seconds or less. Our investigations revealed that some events predicted by the model did not follow the rules of REMI (e.g. produced many **bar** events without **pitch** events), which resulted in a later portion of the events being unable to be converted into "proper" music

IV. Neural Architecture

Although we consider composing music as a "creative" task, it is not the same process as we were trying to ask a machine to perform the task. The models are incapable of "coming up" with a melody and composing a piece of music with that melody. Under the hood, the model could predict the likelihood of musical time steps or events given some starting point (e.g. some prior music time steps or events) after we fed music examples to the model.

A. LSTM

Our initial attempt at music generation utilized recurrent network architecture because a music piece is naturally divided into a sequence of time steps or events. The special aspect of the recurrent neural network (RNN) is the hidden state it maintains, which could capture the "flow" or the "memory" of the music. Consider an example, if the music were at the song's climax, the prediction for the next time step or event would be different from when the music was at the end of the song. That information was captured by the hidden state.

The general flow of a recurrent neural network is as follows:

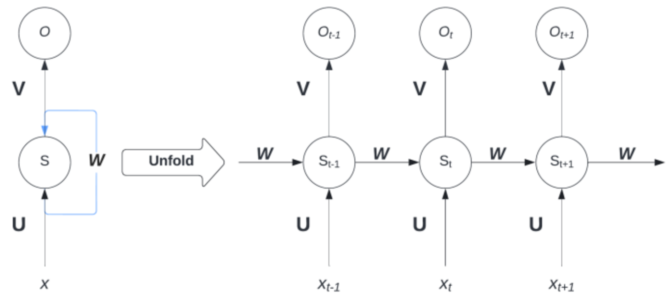

Fig. 6. Recurrent Neural Network: Under the Hood

As illustrated above, there are three sets of weight matrices, namely U, W, V. Each of them represents a different stage in the network architecture. When the input is multiplied by weight matrix U (plus a bias), this represents the embedding of the input at time step t (x_t). This stage projects each element in the input tensor to a vector in a pre-defined dimension, which is a representation of the original element. When the hidden state tensor from the last time step (s_{t-1}) is multiplied by another weight matrix (W), combined with the embedded input and a hyperbolic tangent activation, the current hidden state tensor is calculated. One copy of the hidden tensor would continue to be updated as more inputs from future time steps are fed in. One copy of the hidden tensor would be fed into a final linear layer to produce a prediction for the next time step (i.e. prediction for x_{t+1}). That took place when the copy of the hidden state was multiplied by the third weight matrix V and added to a bias.

The model shares the weights in all time steps. The model will try to find a universal approach to 1) correctly capture the "flow" of the music given some combination of inputs prior to the current step and 2) find meaningful representations of the inputs.

One significant problem associated with RNN is that when the model performs the backpropagation through time to calculate gradients for updating weights, the number of gradients that would be multiplied is proportional to the number of time steps that the model needs to process, which is equivalent to the size of the input sequence. If there are an enormous amount of time steps, a large number of gradients will end up multiplying together, resulting in an exploding and vanishing gradient problem. This was the reason that we decided to implement a variation of RNN, which is called LSTM. With this architecture, instead of updating every time step in the same way, there were "gates" that control how much information, if any, of the current input should be used to update the hidden state. This resulted in two hidden states instead of one. The cell state controls the overall "flow of the music," whereas the second hidden state serves as the "gates." One natural question that follows is, how do we decide what information is important enough to be used to update the hidden state? The answer is that we do not. The model would "learn" about this when it performs backpropagation.

How does LSTM work exactly? Here is a snapshot inside LSTM taken from a blog:

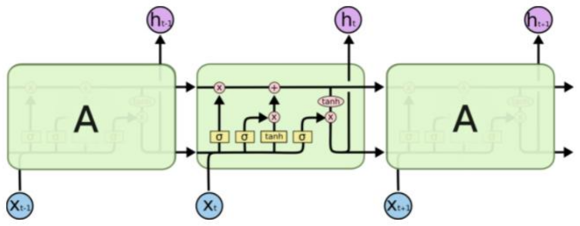

Fig. 7. LSTM: Under the Hood [5]

There are three sigmoid activations representing the three stages in LSTM. The first sigmoid activation serves as a "forget" gate, and it decides if the current information should be written to the cell state or not. The second sigmoid activation combined with hyperbolic tangent activation individually applied to the "local hidden state" (which is the hidden state apart from the cell state) decides what information from the current hidden state should be used to update the cell state (which is the overall "flow" of the music). The last sigmoid activation combined with a hyperbolic tangent activation applied to the updated cell state forms the tensor that would be fed into the final linear layer for prediction of the next time step. It is worth noting that one copy of this hidden tensor would flow toward the future steps and would be updated in the next time step.

B. *Tranformer*

The Transformer architecture is much superior to the LSTM architecture in multiple ways.

In terms of input representation and model complexity, LSTM only uses a single embedding layer, whereas Transformer utilizes the self-attention mechanism on top of input embedding. For example, if the embedding size is set to be 512. Then the embedding layer would project each input into the 512th dimension – resulting in a length-512 tensor. LSTM stops here, whereas Transformer calculates attention scores with the length-512 tensor to find a much more meaningful representation of the tensors in the context of the input sequence. In our implementation, we utilized a multi-head attention layer with 8 heads to capture multiple meaning variations of an input tensor in the sequence context. Our hidden dimension (or model dimension) was set to 1024; after each input token was projected into length-1024 tensors through the embedding layer, it broke the length-1024 tensor into 8 length-128 tensors to calculate 8 attention scores. Then the 8 tensors were concatenated into 1 tensor (with residual and layer normalization added) and fed into the feedforward network with 2048 neurons to complete one encoder stage. In our implementation, we utilized 12 encoders. The output from the previous encoder is fed into another encoder layer (multi-head attention and feedforward layers with residual and layer normalization). The process is repeated 12 times. Afterwards, the output of the last encoder layer is fed into a linear layer and softmax to produce the predicted probabilities. Notice that we do not need the decoder layer because this task is not sequence-to-sequence in nature (e.g., machine translation from English to French) where we encode one information and decode it in another form. In the case of music generation, we do not encode-decode. Therefore, we did not implement the mechanisms on the decoder side of the Transformer. A complete workflow of Transformer from the revolutionary paper "Attention is All You Need" is indicated in Fig. 8.

In terms of sequential representation, LSTM processes everything sequentially, which significantly reduces the ability to perform parallel computing and lengthens the training time. However, the transformer uses an optional encoding that encodes the positional (sequential) information into the embedded input tensor (before feeding it into the first encoder layer). This allowed the magnificent self-attention mechanism to be applied without restricting processing order.

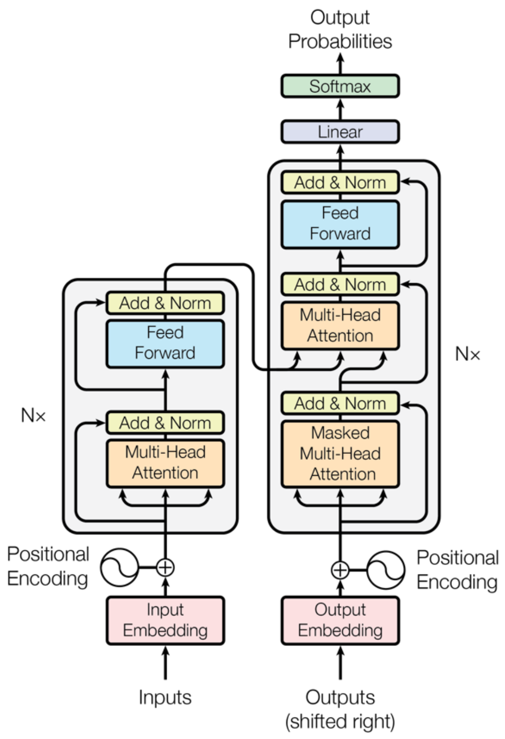

Fig. 8. Transformer Architecture [6]

V. MODEL COMPLEXITY & RESULTS

Due to the complexity and elegance of classical music, we wished to experiment with real complex models and see how much beauty of classical composition can be recreated. However, since we have limited computational power, we could not experiment with enormous complex models.

A. *LSTM*

For our LSTM models, we mainly experimented with two configurations: the complex model has 512 as embedding size, 1024 as hidden size, and 2 LSTM layers. The 2 LSTM layers mean that the first copy of the hidden state tensor was fed into another LSTM layer instead of the final output layer. This increases the model's flexibility significantly. In addition, we implemented two final fully connected layers with 512 neurons and 88 (or 218: the size of events vocabulary) neurons, respectively. In order to make the model robust, we utilized 30% dropout before and after LSTM layers and 50% in the first feedforward layer. We also utilized layer normalization to smooth out the training.

The simple version of the LSTM models has embedding size 128, hidden size 512 and 1 LSTM layer. It only has one final fully connected layer. We found no significance in music composed from the two versions of LSTM models. The composed music pieces were similar to those described in the tokenization section.

B. *Transformer*

For the Transformer architecture, we first experimented with 2 encoder layers with 4 heads (multi-head attention); the embedding size and hidden size were 512 (hidden size is the length of attention layer output). The training and validation loss stopped improving at around 2.3 (which is terrible). The composed music consisted of a few repeated keys throughout, as shown in Figure 9. The model failed to capture any dynamics and variations from the training music pieces. Our training data was padded full songs, which we believe made the matter worse.

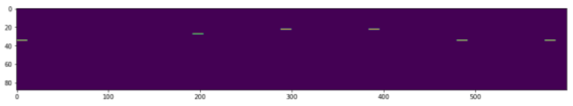

Fig. 9. Super Simple Transformer composition

We later decided to segment the musical pieces into sequences of length-512. We define the more complex model as follows: the embedding size and hidden size were 1024 with 8 heads. We set 12 encoder layers with 2048 as the number of neurons in the feedforward layer as the final portion of our encoder layer. After 9 hours of training, we obtained the exciting result:

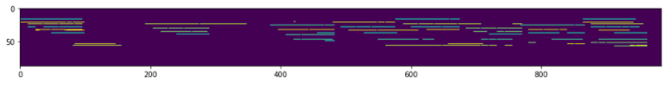

Fig. 10. Pop Transformer Composition

We are currently training another transformer with a training sequence of length-2048, and we look forward to the result! (It would take 36 hours to train 1 epoch)

VI. Additional Functionalities

A. Generating music

The music generation methods that we coded to compose music with our model are different because of the difference in their architectures.

For our LSTM models, the process is more straightforward. Since each prediction of the next time step or event from the LSTM model is a computation between the current input and the hidden states where the "flow of the music" is captured. We could extract the hidden states of a portion of some input music, then iteratively predict the next time steps one at a time. This approach is equivalent to having the model predict the continuation of some music. After extracting the hidden states, we take the last time step from the input sequence and use it as an input sequence (i.e. length-1 sequence). With the hidden states extracted, the model would produce a prediction for the next time step, along with the updated hidden states. We then append the new prediction to the prediction list (containing generated music) and use that prediction as the new input sequence. Then the next prediction and updated hidden states are produced, and so on. We believe that this approach qualifies more as "music generation." However, the generated music was often dissatisfactory.

For our Transformer model, the process is less intuitive. Firstly, we take a sequence of music as input. For example, the input sequence could be the first 500 events of a music piece. The output sequence from the model is the prediction for event 2 to event 501. We then append the predicted event 501 to the input sequence and remove the first event from the sequence. The input sequence is now back to length-500. Lastly, we append the predicted event 2 to the prediction list (containing generated music). The pre-specified output sequence length determines the number of iterations the process would be repeated.

B. Combining track

In terms of REMI, we were perplexed at the beginning that pop music is tokenized as one array of tokens representing the song, whereas classical music is tokenized as multiple arrays of tokens. We initially thought that we needed to find a way to process multiple tracks at once to learn the classical composition. It crossed our mind that we could achieve the objective by modeling each track with a separate model or creating a new dictionary to capture the instances in different tracks at one time step. However, tracks are associated with each other as part of the music, and if we model them separately, we will lose the ability to capture that information. In addition, the events in each track are not aligned in terms of time steps (e.g. one track has 1100 events, while the other has 800 events), and combinations of them could create an enormous-sized dictionary.

Through our analysis of REMI encodings, we discovered the pattern that, in order to play a key, three events must be present in this order: **position**, **pitch**, and **velocity**. The **bar** event separates each section of the music. We then came up with the idea to group the events in different tracks by their position. For example, we could take all the events among all tracks between two **bar** events. Place them based on their **position** event. The keys with a smaller position should be played first. We then designed a program with two pointers always pointing at positions of two tracks and arranged the events according to their relative position to each other. If one pointer reaches a **bar** event, we then take the entire remaining events of the second track until the next **bar** event to synchronize the sections of the music. After two tracks are combined, we could combine the combined track with another track until there is only one track left. We could easily combine any REMI tokenization that creates multiple tracks into a single track in this approach.

C. Customized Dataset

The direct approach for training the models is with the entire songs in our dataset. We can set up a predetermined sequence length, then pad or truncate each sequence into the pre-specified length. In our classical dataset, we gathered 1358 compositions. Here is the distribution of the tokenized sequence lengths:

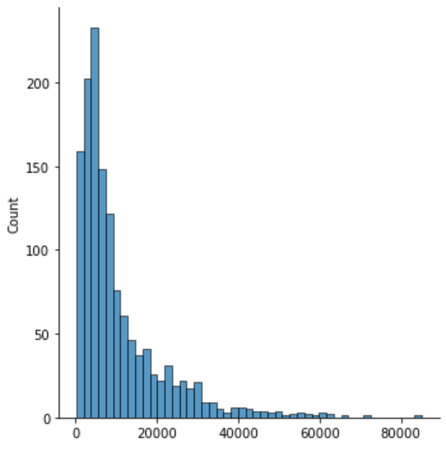

Fig. 11. Distribution of music length

We could specify the padding through the embedding layer in the torch library and create the mask for the encoder in Transformer architecture. However, in our opinion, this would either add unnecessary information (with more than half of the music being 0 paddings) or ignore some information (with more than half of the music being truncated). In addition, we believe it would be easier for the models to learn the local patterns of the music instead of the global patterns (we have an NVIDIA RTX 2060 GPU with 6G of RAM, the models that we could train are limited). In that spirit, we created our own datasets based on those music pieces for both "one-hot" and REMI tokenization.

For the "one-hot" approach, since each time step is represented by a length-88 vector, we stored one training sample in a CSV file with a size 2000 * 88 (if we define a sequence as 2000 time steps). (It was not later that we realized NumPy files take up much less space and are much faster). We segment the length-2000 sequence every 100 time steps. For example, if the first sample is from time step 1 to time step 2000, the following sample would be from time step 101 to step 2010. We could create much more samples with smaller time steps; however, the training would take too long, and we believe the outcome would not be improved in a significant way.

For the REMI encoding, unlike the previous approach where each element in the sequence is a length-88 vector, REMI tokens are numbers. Therefore, we could store the entire training samples or validation samples in one NumPy file. This is much more convenient and faster.

Due to the difference in defining the samples, we modified the customized dataset class for each DataLoader to account for the difference.

VII. Discussion

A. *Overfit?*

It baffled us trying to understand the difference between LSTM and Transformer in terms of predicting events. The LSTM could predict events one by one. We use the newest prediction and the input for the next prediction, and we could write as many as possible. However, Transformers takes a sequence as input and produce a sequence of the same length as output. It cannot perform predictions one by one. That makes sense because, with the self-attention mechanism, the encoding of the tokens needs the whole "context" (i.e. the whole sequence). With LSTM, the hidden state controls the overall "flow" of the music, with that information, we could predict a single event, or a sequence of events, every input is independent of each other in terms of how the LSTM layer processes them. However, the Transformer encodes information of the whole sequence (in terms of similarities) into each element in the sequence. Therefore, LSTM does not care about the next input or how many inputs follow, while Transformer requires that information. Since for each sequence, we took the sequence from the first element until the second last element as input and the sequence from the second element until the last element as output, the output only differs by one event compared to the input. We are terrified that what if the model is "smart enough" to see the trivial solution: shifting everything by 1 event and using all the information to predict the next event? The illustration is as follows:

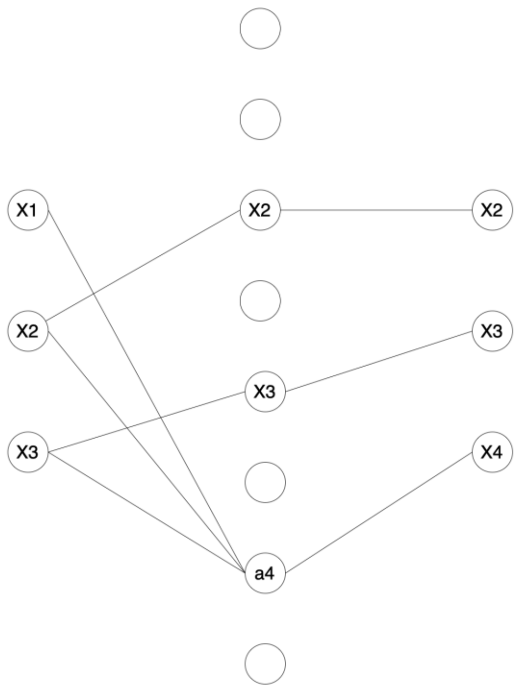

Fig. 12. The Trivia Solution

What we can experiment with is adjusting loss calculation. We could limit the loss calculation to predicting the last 10 events. In other words, we truncate the model output to the last 10 predicted events and compare those events with the last 10 events of the target sequence. Through backpropagation, the model is adjusting itself to produce more accurate predictions for the last 10 events. Before, when the loss is calculated, even if the model got the last prediction (which is essential) completely wrong while correctly "predicted" the other events, the loss would be extremely low. The model could potentially get the idea of "I am doing a great job," but it is not! We believe this is something that we could explore in the near future. Afterwards, when performing the generation part, we take the last prediction from each iteration (which should be highly accurate) and append it to the prediction list. Then we would have a piece of music entirely predicted by the model with the same difficulty as LSTM.

References

[1] C.-Z. A. Huang, A. Vaswani, J. Uszkoreit, N. Sahzeer, I. Simon, C. Hawthorne, A. M. Dai, M. D. Hoffman, M. Dindulescu and D. Eck, "Music Transformer," 12 Dec 2018. [Online]. Available: https://arxiv.org/abs/1809.04281. [Accessed 12 April 2022].

[2] W.-Y. Hsiao, J.-Y. Liu, Y.-C. Yeh and Y.-H. Yang, "Compound Word Transformer: Learning to Compose Full-Song Music over Dynamic Directed Hypergraphs," 7 Jan 2021. [Online]. Available: https://doi.org/10.48550/arXiv.2101.02402. [Accessed 12 April 2022].

[3] G. Motroc, "A machine learning model is only as good as the data it is fed," 6 April 2018. [Online]. Available: https://jaxenter.com/apache-spark-machine-learning-interview-143122.html. [Accessed 10 April 2022].

[4] A. R. Jha, "Mastering PyTorch," [Online]. Available: https://github.com/PacktPublishing/Mastering-PyTorch/blob/master/Chapter06/music_generation.ipynb. [Accessed 12 Apirl 2022].

[5] C. Olah, "Understanding LSTM Networks -- colah's blog," 27 August 2015. [Online]. Available: https://colah.github.io/posts/2015-08-Understanding-LSTMs/. [Accessed 12 April 2020].

[6] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser and I. Polosukhin, "Attention Is All You Need," 12 June 2017. [Online]. Available: https://arxiv.org/abs/1706.03762. [Accessed 3 May 2022].