BERTHOVEN - A MELODY GENERATOR PLUGIN BASED ON GOOGLE'S BERT MODEL

Riccardo Rossi

Dipartimento di Elettronica, Informazione e Bioingegneria (DEIB), Politecnico di Milano
Piazza Leonardo Da Vinci 32, 20122 Milano, Italy
riccardo18.rossi@mail.polimi.it

ABSTRACT

The rapid advancements in artificial intelligence and natural language processing [1] have opened up new possibilities for creative applications in the field of music generation. This paper presents *BERThoven*, a melody generator plugin that leverages Google's BERT (Bidirectional Encoder Representations from Transformers) model [2]. *BERThoven* aims to assist composers and musicians in generating melodic ideas by harnessing the power of deep learning techniques.

The project's primary objectives are both to develop a user-friendly and intuitive plugin that can seamlessly integrate with popular digital audio workstations (DAWs) and to provide a meaningful study of the underlying melody generation deep neural model based on transformers [3]. *BERThoven* incorporates the BERT model [2], a transformer-based neural network architecture known for its ability to capture contextual relationships in text data. By adapting BERT to the domain of music, the plugin can generate musically coherent and stylistically diverse melodies based on user input.

***Index Terms*—** melody generator, BERT model, deep learning, plugin development, music composition, artificial intelligence, transformers

1. INTRODUCTION

This article presents a comprehensive exploration of *BERThoven*, a melody generation system that encompasses both its underlying deep neural network model and its implementation as a complete JUCE [4] plugin. *BERThoven* is designed to generate musically coherent and stylistically diverse melodies, leveraging the power of artificial intelligence. The focus of this article is to provide a detailed understanding of the model's technical aspects and its practical application within the context of a fully functional JUCE plugin.

The deep neural network model at the core of *BERThoven* is thoroughly examined, including its architectural design, training methodology, and evaluation. Complementing the discussion on the deep neural network model, the implementation of *BERThoven* as a JUCE plugin is also discussed. JUCE, a widely used C++ framework for audio application development, serves as the basis for building *BERThoven*'s user interface, MIDI file [5] generation functionality, and integration with digital audio workstations. This practical implementation enables users to interact with *BERThoven* directly, input their music, and obtain AI-generated melodies in the form of MIDI files.

This comprehensive examination of *BERThoven* as a complete system for melody generation showcases its potential to empower musicians and composers with AI-driven creative capabilities. By merging theoretical concepts with practical implementation, this article highlights the possibilities of AI-driven melody generation as a valuable resource in the music composition process.

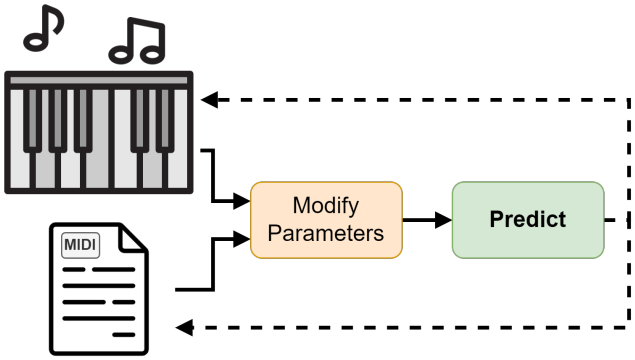


Figure 1: Image depicting the general workflow of the *BERThoven* Plugin. The input is to be intended as either playing notes on the included keyboard or adding a MIDI file from an external source.

2. THE BERTHOVEN PLUGIN

BERThoven is a complete application that can generate melodies based on input given by a user. This application was developed as an audio plugin using JUCE, a C++ framework for audio applications. The underlying model used to generate melodies was trained in Python using PyTorch [6], saved as a file and exported to C++ using the C++ equivalent of PyTorch, called libtorch [7]. Being a fine-tuned BERT, the model needed functions in order to tokenize the given input strings into sequences of tokens which were then converted into tensors that could actually be fed to the network. These functions were simply imported from the *transformers* module [8] on the Python's side of the application and were implemented by hand in C++ [9] to match the ones used in Python.

In general, the plugin system allows users to generate melodies in two ways (as shown in Fig. 1): by playing notes directly on the plugin's keyboard [10] or by loading a MIDI file onto the plugin interface. After composing or selecting the MIDI input, the user can press a button to predict the next n notes of the sequence. The plugin offers some parameters to personalize the user experience and the interaction with the generator, such as the number of notes to predict and the length of the predicted notes. Once the notes are predicted, the user can drag and drop the output as a MIDI file onto his DAW of choice or somewhere in his system where will be stored.

2.1. Inference in C++

As previously stated, string tokenization functions had to be implemented by hand on the C++ side of the system. To understand the inference process on this side of the application, those two functions will be explained and discussed, with the help of some *pseudo-code*.

The first function is called `get_vocab`. It reads all the available tokens for the BERT model from a text file and stores them inside two standard maps (where one maps tokens to token ids and the other maps token ids to tokens). Inside the file, tokens are organized in lines, where the first element of the line is the string representing the token and the second one is the id of that specific token.

```
function get_vocab (path_to_file) {
    map token2id
    map id2token

    file = open_file (path_to_file)

    for each line of the file {
        token = line[0]
        token_id = int(line[1])
        token2id[token] = token_id
        id2token[token_id] = token
    }

    return token2id, id2token
}
```

The second function, `preprocess`, takes a given input string and tokenizes it using BERT's standard representation via token ids and attention masks (that specify the importance of each token in the sequence) [11]. This tokenized string is then padded to a maximum length (given as a parameter to the function) and converted to two distinct tensors: one for the token ids and one for the attention masks.

```
function preprocess (text, max_length) {
    vector input_ids (max_length)
    vector masks (max_length)

    for each word w of the string {
        input_ids.put(token id corresponding to w)
        masks.put(1)
    }

    pad input_ids with pad tokens -> [PAD]
    pad masks with zeros
    tensors = convert input_ids, masks to tensors
    return tensors
}
```

By implementing these functions, the C++ code can prepare the input tensors to be fed to the fine-tuned BERT model for prediction. Overall, the inference process follows a simple scheme (also depicted in Fig. 2):

1. Store MIDI notes (either played or from file).
2. Create a vector `in` of strings containing the last 100 input notes, represented as their corresponding pitch number in an octave (from 0 to 11).
3. For each note to predict:
   - Join the elements of vector `in` to form a single string `str` with notes separated by a blank space.
   - Preprocess string `str` to get the input tensors.
   - Predict the next note through inference on the model.
   - Store result in a separate vector `out`.
   - Store result at the end of vector `in` and shift it.

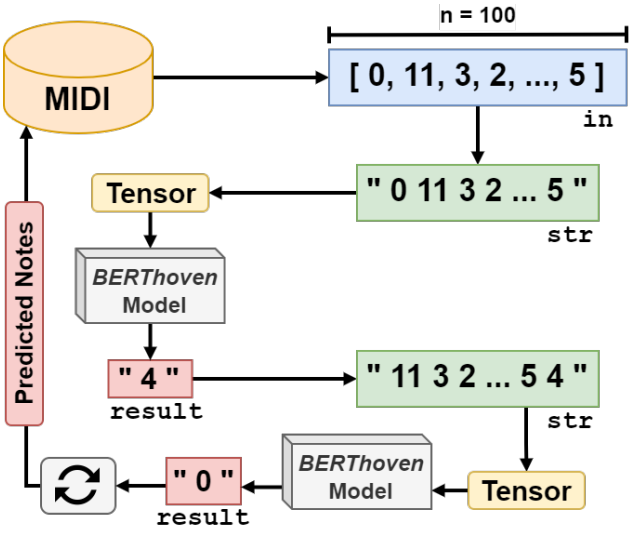


Figure 2: An example of the inference process used to predict notes.

4. Convert the predicted sequence to MIDI messages.
5. Store predicted MIDI messages in the same vector where user input MIDI notes are stores (i.e., they become the input for the next prediction).
6. Return the sequence of predicted MIDI messages.

The plugin then proceeds to create a MIDI file containing the predicted notes, where each note has the duration specified by the user prior to the predicting process.

3. GUI AND MIDI INTEGRATION

The *BERThoven* plugin was designed to be straightforward and easy to use. Its user-friendly interface (Fig. 3) enables quick and easy interaction, allowing musicians and composers to generate melodies effortlessly.

There is a total of 7 UI components, including the virtual MIDI keyboard. Those are, in the order in which they appear on the GUI:

- *Num. of Notes Slider*: Controls the number of notes to predict for each press of the *Predict* button. It ranges from 1 to 50 notes.
- *Dur. of Notes Slider*: Controls the duration of predicted notes as they are written on the output MIDI file. It ranges from 0.5 to 4 quarters (step of 0.5).
- *MIDI Drop Target Area*: Here is where external MIDI files can be dragged for them to be read as input by the plugin.
- *MIDI Drag Source Area*: Here is where the output-generated MIDI file can be dragged from. The MIDI file is never moved, always copied.
- *Reset Button*: Used to reset the input sequence of MIDI notes to initial conditions.
- *Predict Button*: Used to generate the next n notes of the sequence.
- *MIDI Keyboard*: Here is where the user can play a melody that will be stored as input on the plugin.

Figure 3: A screenshot of *BERThoven*'s GUI.

The user can also play melodies on an external MIDI device and the played notes will be visible on the MIDI keyboard component present in the GUI, indicating that they are being registered as an input sequence for prediction.

In order to further facilitate the user experience, the predicted sequence of notes is provided as a MIDI file with a single track that contains the notes in their MIDI message form. This was achieved by using JUCE's internal functions to create a MIDI sequence from the predicted MIDI messages and writing it to an actual file on disk. This generated MIDI file can then be dragged out from the plugin to be copied into an external location chosen by the user (inside the DAW or on the file system). The generated output MIDI file will be available until the user predicts a new sequence of notes or until he closes the plugin.

4. A MELODY GENERATOR BASED ON TRANSFORMERS

As previously stated, *BERThoven* builds upon Google's BERT model [2], a transformer-based neural network architecture originally developed for language modeling tasks.

4.1. What is BERT?

BERT, developed by Google Research, represents a groundbreaking advancement in the field of language modeling and understanding. It is a transformer-based neural network architecture that excels at capturing contextual relationships in textual data. Unlike traditional language models that process text in a unidirectional manner, BERT incorporates bidirectional training, enabling it to consider the complete context surrounding each word in a sentence. This bidirectional approach equips BERT with a profound understanding of the interdependencies and nuances within the text.

The adaptability and versatility of BERT make it an ideal candidate for extending its capabilities beyond traditional language processing tasks. By applying BERT to the realm of music composition, *BERThoven* aims to leverage the model's language understanding abilities to generate musically coherent and diverse melodies. By aligning BERT's contextual comprehension with the domain-specific requirements of melody generation, *BERThoven* provides a unique and powerful tool for composers and musicians seeking to explore novel creative avenues.

4.2. Adapting BERT For Music Classification

One of the remarkable aspects of BERT is its adaptability to diverse domains beyond natural language processing. In the context of music, BERT can be leveraged for classifying sequences of notes into distinct classes that represent the twelve notes of an octave. This adaptation allows *BERThoven* to generate musically coherent and stylistically diverse melodies based on the learned representations.

In this adaptation, the input to BERT consists of strings of length n, where each element represents a note. The objective is to classify these strings into one of twelve classes corresponding to the twelve notes in an octave (e.g., C, C#, D, D#, E, F, F#, G, G#, A, A#, B).

To prepare the data for BERT, the note strings are tokenized, similar to how words are tokenized in natural language processing. Each note is represented as a token, and special tokens are added to indicate the beginning and end of the sequence. Additionally, token embeddings and position embeddings are incorporated to capture the relative positions and relationships between the notes within the sequence.

By fine-tuning BERT on a dataset of annotated note sequences, the model becomes proficient at classifying unseen note strings into their respective note classes. The learned representations capture the nuances and contextual dependencies of musical sequences, enabling *BERThoven* to generate music based on the user's input and preferences. Ultimately, *BERThoven*'s model architecture is composed of a fine-tuned BERT model with two additional dropout and linear layers that convey the output into the predefined pitch class categories.

4.3. The Dataset

In order for it to fully comprehend the intrinsic patterns of musical composition, the model was trained with a collection of MIDI files containing piano music, mostly consisting of pieces from the *Final Fantasy* official soundtrack. To prepare the data for the training and validation processes, the following procedure was applied:

1. Read each MIDI file and store its notes, representing them as integer numbers from 0 to 11 (the twelve notes of an octave).
   - If the notes at time t are stacked to form a chord, skip them.
2. Divide the parsed notes into input sequences and output labels (Fig. 4) where:
   - The input sequence is composed of 100 sequential notes.
   - The output label is the note that comes after the input sequence.
3. Do this starting from **each** note of the sequence.

Sequences of length 100 were picked specifically to enhance the ability of *BERThoven* to generate more coherent melodies based on a "long" succession of notes, further improving the overall user experience.

Figure 4: Example of how training data is prepared, using input sequences of length 6. The green block is the input sequence, while the red one is the output label.

Oversampling was then applied to the fetched data to mitigate the effects of class imbalance. Oversampling involves replicating or synthesizing instances from the minority class (less frequently occurring notes) to balance the distribution of class labels in the dataset. By increasing the representation of the minority classes, the model can better learn the patterns and characteristics associated with those notes, improving overall classification performance. Oversampling was achieved by simply duplicating data instances of the minority classes to match in number those of the most represented class.

4.4. Training and Validation

To train the *BERThoven* model for note classification, the dataset was divided into training, validation and test sets to assess the model's performance on unseen data, which is essential when dealing with music generation. Overall, training and validation were performed on a limited set of sequences due to the large computational cost of the procedure (even when using accessible Cloud services).

The training loop followed a supervised approach where each note string was associated with a corresponding class label representing one of the twelve distinct notes of an octave. The loss function used was a cross-entropy loss.

The validation dataset was used to prevent overfitting and to generalize the capabilities of the model. After each epoch, the model's performance on both the training and validation set was evaluated in terms of total loss and total accuracy per epoch.

5. TECHNICAL EVALUATION

In the previous sections, an overview of the model's architecture, dataset preparation and training process was presented. Building upon that foundation, the technical evaluation section offers an in-depth exploration of the neural network model, assessing some technical questions that ultimately lead to the final iteration of *BERThoven*'s deep neural network model. The following lines will provide answers to these technical questions, addressing them one by one.

5.1. Q1: How Do Training Parameters Affect The Overall Model's Performance?

In searching for the best possible version of the model, a number of tests were conducted on the model's training and validation phases to study how the model's performance changed as a function of different hyperparameters.

All tests and evaluations conducted throughout this technical analysis were executed on a consistent machine configuration with the following characteristics:

- Processor: AMD Ryzen 3600X.
- Graphics Card: NVIDIA GeForce RTX 3060.
- Operating System: Windows 11 Professional.
- Software Frameworks and Libraries: Python, PyTorch.

It is also important to note that the seed for all the different RNGs used during testing was manually set to further reduce the randomness of the generated numbers and of the order of input sequences when shuffling.

To have a clearer understanding of the registered values during training and evaluation analysis, plots (shown in Fig. 8) have been created to show how the two metrics used for evaluation - accuracy and loss - change over the course of the number of training epochs. Here are the results of the conducted tests, divided by test type:

1. **Number of input sequences (for both training and validation).** It can be seen from Fig. 8a that the number of input sequences fed to the model during training highly affects its performance in terms of accuracy and loss.
   - For $n = 100$, it can clearly be seen that the model fails to converge, as accuracy remains in a low range (lower than 30%). This happens because there are too few samples for training.
   - For $n = 1000$, the model begins to show better results, yet it remains limited in its ability to accurately predict notes. This, again, is probably due to a scarcity of samples to learn from.
   - For $n = 5000$, the model shows the best results out of the four tested scenarios. This happens probably because the model has enough sequences to conduct efficient learning on the training data, while at the same time, there is little evidence of noise in the limited number of fed sequences, a factor that further limits the model's "confusion" during prediction, implying a higher accuracy and lower loss.
   - For $n = 10000$, the model still holds significant results, as both accuracy and loss are only slightly worse than those of the model trained with 5000 sequences.
2. **Learning rate.** Learning rate has a huge impact on the speed at which the model trains. It can be seen from the graph in Fig. 8b that the higher the learning rate, the faster the model is able to achieve convergence during training. All values chosen for the learning rate to test the model are noticeably smaller when compared to the ones usually picked for other types of models (like LSTMs, for instance). The choice of keeping these lower values comes from Google's official paper for BERT [2], where they explain that a learning rate greater than 5×10^{-5} could cause a catastrophic forgetting of the pre-trained model's parameters, drastically decreasing the model's performance.
3. **Batch size.** Batch size also has a significant impact on the model's performance (Fig. 8c).
   - For $bs = 2$, the model shows a significantly more rounded curve and an overall higher accuracy after 20

epochs of training. At the same time, while the train loss decreases intensively to reach that of the other models, the validation loss is much higher than that of the other considered scenarios. This is probably because using "tiny" batches introduces more noise in the gradient estimates during training, which in return leads to a more erratic learning process, making it difficult for the model to generalize well to unseen data.

- For $bs = 8$, $bs = 16$ and $bs = 32$, there is a common trend in how the models behave during training. Having a greater batch size seems to improve the total loss of the model, while at the same time, it seems to decrease the overall accuracy. This happens because larger batch sizes provide a smoother and more averaged estimation of gradients during training.

4. **Different combinations of shuffling and oversampling over the input sequences.** To show the effects of oversampling and shuffling of the input sequences, four tests have been conducted using all four logic combinations of the two pre-processing techniques (Fig. 8d).

   - When both shuffling and oversampling are turned off, the model seems to have the worst results out of the four studied cases, with both a lower accuracy and a higher overall loss.
   - When only oversampling is turned on, on the other hand, the model seems to have the best results in terms of validation accuracy and loss, while it is beaten by the model with both oversampling and shuffling turned on in terms of training accuracy and loss.
   - When only shuffling is turned on, the model behaves better than the not-shuffled, not-oversampled one, but results show a significant decrease in performance due to the lack of uniformity in the representation of the label classes.
   - When both shuffling and oversampling are turned on, the model shows the highest results during training and the second to highest results during validation.

5.2. Q2: How Well Does The Model's Measured Performance Cope With its Actual Usage Inside The Plugin?

Having thoroughly examined and discussed the various test cases and their corresponding results, it is essential to consider the practical implications and real-life application of the model. While the evaluation provides valuable insights into the model's performance under controlled conditions, understanding its effectiveness in real-world scenarios is crucial.

In real-life usage, the model will encounter a range of inputs, musical styles, user preferences, and other variables that may differ from the controlled test cases. These real-world factors can significantly impact the model's performance, generalization capabilities, and ability to generate musically coherent melodies.

While the variation of batch size and learning rate have little to no effect on an actual usage of the model, changing the number of input sequences and applying different shuffling and oversampling techniques have a significant impact on the user's perceived quality of the overall system. More specifically:

1. **Varying Number of Input Sequences.** The higher the number of training sequences considered, the higher the ability of the model to create more diverse melodies, due to the simple fact that it had more data to improve upon during training. At the same time, a lower number of sequences can lead to repetitions and "stagnation" in the model's ability to generate new, interesting melodies. This means that higher performances from a technical point of view don't necessarily translate to a higher perceived quality from the user's perspective, as other subjective factors come into play when dealing with music and the perceived quality of a melody.
2. **Shuffling.** Because of how the model was trained, its ability to generate coherent melodies comes from how the input sequences are created, where there is a certain level of coherence from a melodic point of view, as the model learns to create pieces of music note by note, following a highly sequential flow of data. Introducing shuffling means that this level of coherence may be lost entirely, rendering the generated melodies seemingly more random and perceived as making "less sense" in the overall scope of melody composition.
3. **Oversampling.** On the other hand, not applying oversampling to the input sequences means that the less represented class labels are less frequently present during training, which translates into a complete avoidance of those specific notes during inference of the model. For instance, if the model has a low percentage of a certain note in its expected prediction output, the model is less likely to produce said note as a prediction output during inference, thus making the predicted sequence of notes rather "monotone" or, again, "stagnating". Introducing oversampling, on the other hand, improves the ability of the model to create melodies with a more evenly distributed prediction of notes.

5.3. Q3: Why Was One Particular Instance Of The Trained Model Chosen Over The Others?

Taking into account both the technical evaluation and the user perception, the chosen model represents the best compromise. It strikes a balance between technical measured performance and the subjective assessment of the generated melodies by the user. The final proposed model presents the following hyperparameters:

1. Number of Input Sequences: 10000.
2. Learning Rate: 5e-6.
3. Batch Size: 16.
4. Shuffling: off.
5. Oversampling: on.

The model was trained for a total of 20 epochs as a longer training would have had little to no effect on the model's performances (as the model had already converged to stable results).

6. RESULTS

It can be seen from Tab. 1 that the final proposed model for melody generation based on BERT achieved notable results both in terms of accuracy and loss.

	Accuracy	Loss
Train	0.9873	0.0036
Validation	0.8318	0.0494
Test	0.8260	—

Table 1: Results of *BERThoven*'s model.

These results, while already commendable, could be further enhanced by employing a more specific and coherent dataset. Utilizing such a dataset would lead to a marked improvement in the coherence of the generated melodies, resulting in more musically consistent outcomes and - in theory - higher performances in terms of accuracy and loss, especially during validation and testing.

6.1. Generated Melodies

As a creative application, *BERThoven* requires evaluation not only in technical terms but also in terms of the perceived quality of the generated melodies. Presented below are some examples of generated melodies, facilitating a discussion on the more artistic aspects of the application.

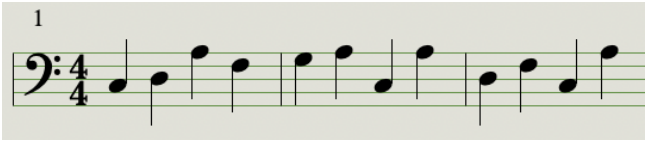


Figure 5: Generated melody example 1.

The first example (in Fig. 5) presents a very simple melody that was obtained directly after starting the application and predicting the next 12 notes. Even from this short excerpt of music, it can be observed that the plugin is able to keep a certain degree of simplicity and coherence during generation.

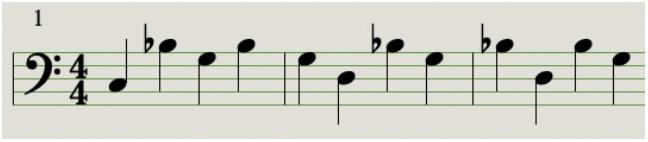


Figure 6: Generated melody example 2.

The second example (Fig. 6) was obtained by predicting another eight-bar piece starting from the previous example and keeping only the last three bars of the generated melody. Again, *BERThoven* is able to keep a high level of coherence by using a subset of note tonalities and changing the order in which notes appear.

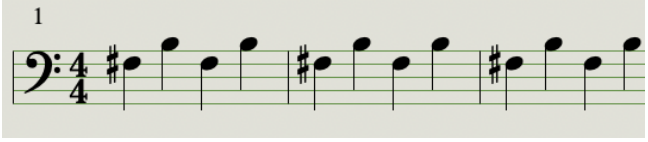


Figure 7: Generated melody example 3.

The third and last example (Fig. 7) shows instead one of the weaknesses of the application. Indeed, the analysis reveals a notable tendency towards high repetitiveness in the generated melodies. Multiple occurrences of the same bar can be observed, and in some instances, a state may arise where a particular bar is perpetually repeated. However, this cycle can be interrupted by introducing new and distinct notes from an external source, injecting variation and freshness into the melody. This phenomenon usually only occurs when the input notes or pre-made melodies are highly repetitive on their own (as in the case of this particular example).

7. CONCLUSION

In conclusion, *BERThoven* represents a successful endeavor in harnessing the power of artificial intelligence and deep learning to create a sophisticated melody generation system based on transformers. Through the combination of a deep neural network model and user-friendly plugin interface, the system stands as a powerful tool for musicians and composers for generating musically coherent melodies.

7.1. Future Improvements

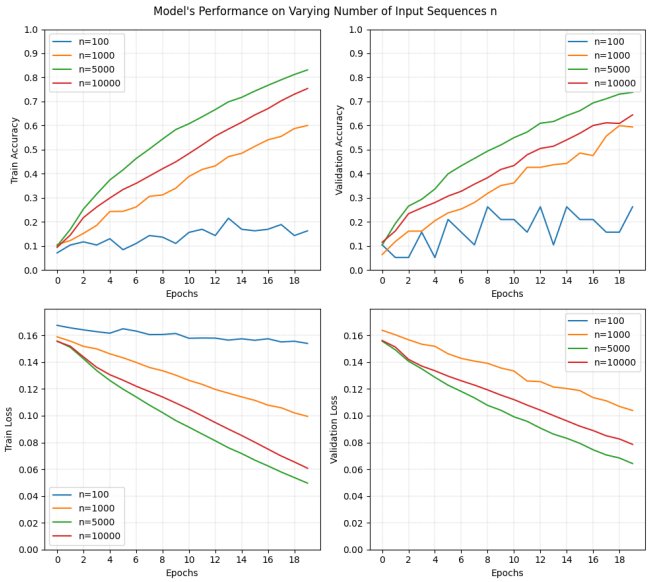
The future of *BERThoven* holds promising opportunities for enhancements and refinements. As with any AI-driven system, continuous improvement is essential to adapt to evolving user needs and technological advancements. To that end, the following bulleted list outlines potential areas for future improvements:

- Enhanced Dataset: Expanding and diversifying the dataset used for training could further improve the model's ability to handle different musical genres and styles. Incorporating a more extensive range of musical compositions would contribute to better generalization and performance.
- Fine-tuning: Fine-tuning the model on specific musical styles or scales could enable *BERThoven* to generate melodies that adhere more closely to particular user preferences and requirements.
- Real-time Adaptation: Implementing real-time adaptation capabilities would allow the model to adapt and respond dynamically to user inputs during melody generation, providing a more interactive and personalized experience.
- Contextual Understanding: Enhancing the model's understanding of musical contexts, such as chord progressions and harmonies, could lead to the generation of more musically expressive and harmonically rich melodies.

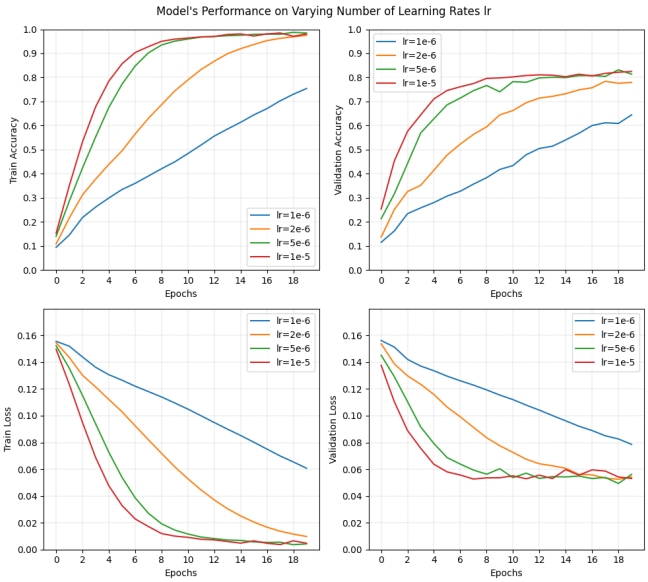
8. ACKNOWLEDGMENTS

I would like to express my sincere gratitude to Politecnico di Milano and the Goldsmiths University of London for providing me with the opportunity to embark on this inspiring journey of exploring deep generative models for music generation. Their support and resources have been invaluable in shaping this project.

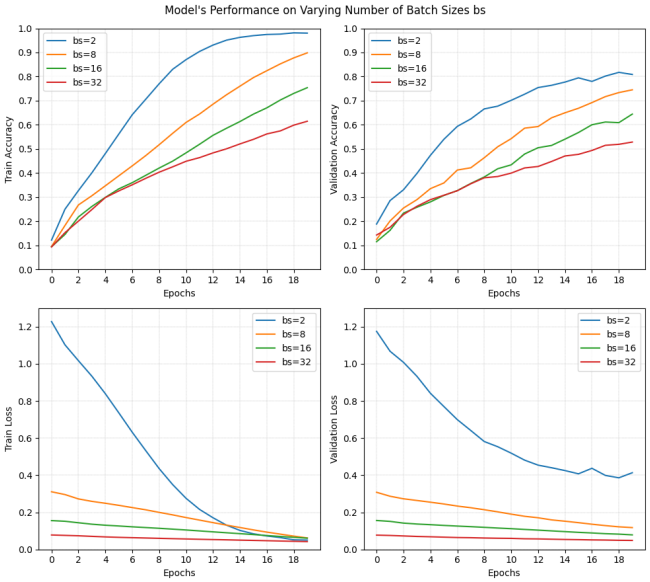
I am especially grateful to my friend Matteo, whose insightful suggestion to utilize BERT as a starting point for the proposed model was a pivotal moment in the development of *BERThoven*. His innovative idea set the foundation for this project and sparked a creative vision that has been a driving force throughout.



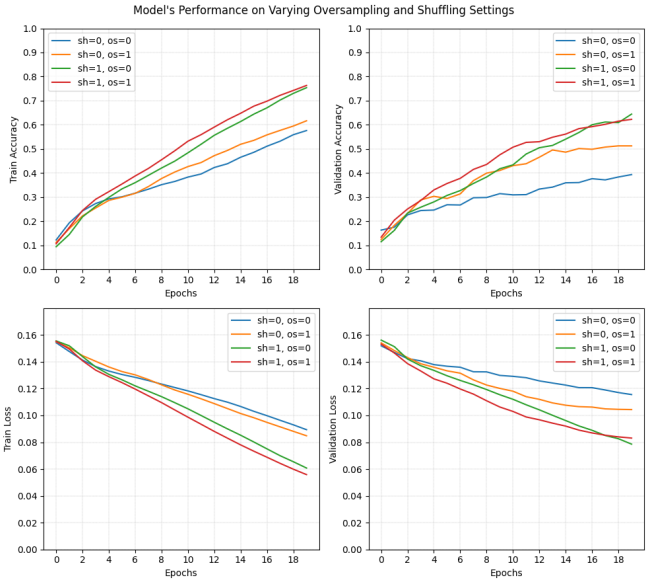
(a) Varying Number of Input Sequences.



(b) Varying Learning Rates.



(c) Varying Batch Sizes.



(d) Different Combinations of Shuffling and Oversampling.

Figure 8: Plots containing results for each variation of hyperparameters.

9. REFERENCES

[1] Google Cloud, "What Is Natural Language Processing?", https://cloud.google.com/learn/what-is-natural-language-processing?hl=it#:~:text=Natural%20language%20processing%20(NLP)%20uses,media%20sentiment%20and%20customer%20conversations.

[2] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," 2019.

[3] Rick Merritt, "What Is a Transformer Model?", NVidia Blogs, 2022, https://blogs.nvidia.com/blog/2022/03/25/what-is-a-transformer-model/.

[4] Juce Official Website, https://juce.com/.

[5] MIDI Association, "Official MIDI Specifications", https://www.midi.org/specifications.

[6] PyTorch Official Website, https://pytorch.org/.

[7] PyTorch Official Website, "Installing C++ Distributions of PyTorch", https://pytorch.org/cppdocs/installing.html.

[8] HuggingFace, "transformers Module's Official Documentation", https://huggingface.co/transformers/v3.0.2/index.html.

[9] Yunusemre Özköse, "Pytorch and C++ 6: Bert Text Classification in C++" https://medium.com/mlearning-ai/pytorch-c-6-bert-text-classification-in-c-b5d94350f564.

[10] JUCE, MidiKeyboardComponent Documentation, https://docs.juce.com/master/classMidiKeyboardComponent.html.

[11] Analytics Vidhya, "An Explanatory Guide to BERT Tokenizer" https://www.analyticsvidhya.com/blog/2021/09/an-explanatory-guide-to-bert-tokenizer/.