Producing novel music with adjustable difficulty using AI

Oliver Morgan
om270@kent.ac.uk

School of Computing
University of Kent

Word Count: 6,100

May 21, 2024

Abstract

This should provide a concise summary of the paper. Typically, it will be between 100 and 200 words in length.

This project explores the use of AI to generate piano music with various difficulty, a new idea within the computer generated music domain. Using recurrent neural networks and a custom dataset, we developed a generative music model and a difficulty classifier. We use these two models in tandem to be able to generate our music. Through extensive testing and experimentation, we were able to evaluate the effectiveness of our system along suggesting potential improvements that could be made to make the system more effective. Overall, this report helps to show that there is room for this kind of system within the AI-driven music composition field, and it offers information about the challenges and opportunities of generating music with varying levels of difficulty.

1 Introduction

Throughout the past few years, we have seen significant advancements in Artificial Intelligence (AI) generated music with projects such as Google's Magenta Transformer (Huang et al., 2018) and OpenAI's Musenet (Payne, 2019). These developments have led to a field where AI system are capable of composing music in various styles and genres.

However, despite these advancements, there is a gap in the space of generated music specialised for practice. While current AI-generated music offers long-form compositions that sound cohesive and perform well, there exists no way to generate pieces that are personalised for the musician. Our project seeks to address this by using AI to generate symbolic piano music with adjustable difficulty levels. By using a custom dataset of graded piano pieces from syllabuses such as ABRSM and Trinity, we are able to generate and classify roughly according to their difficulty levels. This allows users to tailor their generated music to suit their individual skill level and enhance their own practice sessions.

2 Background

2.1 LSTM

Recurrent Neural Networks (RNNs) are a type of neural network that is able to process sequential data by using a feedback mechanism. RNNs have recurrent connections that allow it to capture information from previous time steps. Each RNN unit can process its input and updates the weights and biases based on the current input and previous hidden state (Sherstinsky, 2020). This ability to remember context from previous values in the hidden state allows the RNN to process sequential data with memory. However, RNNs can suffer from the vanishing/exploding gradient problem which can mean the RNN cannot effectively capture long-term dependencies which is often prevalent in music. This happens because as the gradients are back

propagated, they tend to exponentially shrink or grow, which can cause the network to struggle to learn from past data in a sequence. Gers, Schmidhuber and Cummins (2000) say that standard RNNs can fail to learn from data 5-10 time steps in the past due to this vanishing/exploding gradient problem.

LSTMs are implemented by using 3 distinct gates to control the flow of information alongside a cell state, which refers to the long term memory, and a hidden state, which represents the short term memory. The first gate used is called the forget gate, (Staudemeyer and Morris, 2019) which determines how much of the cell state should be remembered. A sigmoid activation function given by this formula:

$$f(x) = \frac{e^x}{e^x + 1}$$

is used here to pass the output of our values with their weights and biases to restrict our value to between 0 and 1, which allows it to be represented as a percentage. This is how much of our cell state will be remembered from previous time steps. Next is the input gate, this gate determines how much new information should be added to the cell state. One block is used to calculate a 'potential memory', a Tanh activation function given by this formula:

$$tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

This activation function is used to turn the processed input to a value between -1 and 1 which works well determining a new memory to add to the cell state. A second block is used with a Sigmoid activation function to determine how much of this new potential memory should be added to the cell state. The final step is the output gate which updates the hidden state by passing the new cell state value into a Tanh activation function multiplied by the output of a Sigmoid activation function similarly used previously to determine how much of this potential short term memory to be passed as hidden state. This output can then be taken as an output at any stage for our LSTM. This improvement on the RNN allows the LSTM to retain

information for data sequences above lengths of 1000. As LSTMs capture patterns in sequential data, they are well-suited for use in tasks such as music generation and classification.

Transformers are another type of recurrent architecture that has become popular in various applications such as natural language processing, music generation and more. We chose not to use transformers for this project as we had personal interests in learning how LSTMs worked along with the simplicity of an LSTM seemed more suitable for this project where the sound and quality of the generated music wasn't the highest priority.

2.2 Music Theory

There are some fundamental music concepts that are important to understand for generating music with AI.

- Tempo: The speed at which the music is played. This can either be indicated by a beats per minute (BPM) marker which can show how long a certain note should be played (e.g. 120 BPM). Tempo can also be indicated by the corresponding Italian word for the tempo range (e.g. *Moderato* for 108 — 120BPM). There can also be markings that correspond to a change in tempo over time, such as *Ritardando* which refers to gradually slowing down.

- Dynamics: The volume at which the performance is being delivered, this can be indicated by dynamic markings such as ***p*** or *piano* which means soft or quiet. Gradual changes can also be indicated by the italian words (e.g. *Crescendo* meaning increasing in volume) or hairpins, which can be placed covering a passage of music to show a change in dynamics. For example, a *Crescendo* hairpin opens towards the left $>$.

- Articulation: This determines how a note is sounded, they can affect dynamics, pitch and more. An example of an articulation could be *Legato* which indicates the music notes are to be played smoothly and

connected, or *Staccato* signifies that the notes should be played detached.

- Pitch: The frequency at which the note is played, low frequencies refer to low notes and high frequencies refer to high notes. A **note** is just a symbol that signifies a specific frequency, some examples of notes could be A3, C1, G5. A key is a group of specific notes that form a basis for the music composition. An example of a key could be A major, which includes the notes A, B, C#, D, E, F#, G#. A chord is when more than 1 note is played simultaneously, these can be formed and named in various ways. An accidental is when a note is played as sharp (+1 semitone) or flat (-1 semitone), these accidentals are usually the black keys on the piano.

- Duration: The length of time that a music note or rest is held for. Symbolic music can include rests, which is a period of non-playing in a piece. In symbolic music can include ties, these ties can tie multiple notes together that extends its duration beyond its original length. When notes are tied together, it is played as a single continuous note.

- Rhythm: This serves as the pattern of when the notes are played and how long the notes are played for. A time signature that can specify the number of beats per measure, which is a core element of rhythm. It consists of two numbers displayed as a fraction, the top number indicates the number of beats per measure and the bottom number indicates what type of note a beat corresponds with. For example, in a time signature of 4/4 (most common time signature), each beat corresponds with a quarter notes and every 4 beats will form a measure.

2.3 Music Difficulty

Many varying factors are included when determining the complexity of performing a piece of music. These factors include technical ability but also

more nuanced ideas such as interpretation or maturity of a piece. Some important factors will be listed below, but there are many factors not listed that can have an effect on the difficulty.

- Duration: The length of a musical piece can be a significant factor when determining the difficulty of a piece. Longer compositions tend to require more focus and endurance to play accurately, thus increasing the difficulty. However, a piece can be short and still difficult if it includes constant complex passages.

- Speed: The speed or tempo that a piece is played at will influence its difficulty as, in general, a faster tempo would require more coordination and precision to play whilst a slower tempo could be more simple. Additionally, tempo can change throughout a piece, which can increase difficulty if the musician must constantly alter their playing speed.

- Tonality: The key signature and tonal structure of a composition can affect the difficulty of performance. In general, some key signatures are generally considered more difficult than others, but this can be entirely up to the musician. For example, C Major with no accidentals is usually the key signature that musicians are initially taught, but it is considered by some as one of the most difficult keys to play in due to it requiring an unnatural hand shape to play. Additionally, the key signature changing throughout a piece can increase difficulty as the performer is required to constantly adjust the notes they are playing.

- Style: Different genres of music such as jazz, classical or more can have different stylistic characteristics that require various skills depending on the genre. For example, a jazz piece may require an understanding of jazz harmony and rhythm that a classical pianist may not be practised in. Because of factors like this, ABRSM offers different assessments for classical and jazz pianists.

- Expression: The expression required whilst playing a piece is something

that is impossible to quantify properly when it comes to difficulty. Musical expression includes the emotion and interpretative aspects of a performance that can affect dynamics, tempo, articulation that won't always be explicitly noted and is subjective to the performer.

In general, a lot of the factors going into what makes a piano piece difficult can be subjective, some musicians may be adept at one aspect but weak at another which makes them more suited to playing specific pieces.

2.4 Tokenisation

Tokenisation is an important pre-processing step in generating symbolic music, as it is necessary to ensure our training data in the form of Musical Instrument Digital Interface (MIDI) files can be used as input for our neural network. Tokenisation works by breaking up data into small chunks called tokens, where these tokens will refer to the data stored in each musical element. Some of the attributes that we store in our tokens include pitch and duration. It is also important to ensure our tokens can be represented in a form that is readable by neural networks, such as storing them as integers.

2.5 Previous Works

In the past 5 years, we've seen significant improvements to generated music through ground-breaking research and innovative projects. There are many ways to generated muse using computers, some methods include using Genetic Algorithms (de León et al., 2018), Generative Adversarial Networks (Yang, Chou and Yang, 2017). In work done by Siphocly, El-Horbaty and Salem (2021), they evaluate the top 10 methods used for computer generated music.

Alongside different computation approaches, there are two main forms that the music is in: symbolic and audio-based representation.

Audio-based representation uses audio signals to train the models. An advantage of audio-based representation is that it can be easier to find sig-

nificant data for, and that the output does not need any further processing to view. A disadvantage of this representation is that it can be difficult to control certain performance attributes like dynamics or articulation. One of the most popular examples of a system generating raw audio is Deepmind's WaveNet (van den Oord et al., 2016). WaveNet is a deep neural network system that uses a component called 'casual convolutions' which allows the model to respect the ordering the data is in without having recurrent connections (like in RNNs). This WaveNet model can be applied to music generation, as seen in Luo (2022)'s work in generating music in the style of Bach.

Symbolic representation is when the music is represented through general sheet music, which is one of the most popular approaches for displaying music, especially within musicians. Musical elements are represented as specific symbols that convey information such as pitch, duration, dynamics and other markings. There can be other symbols such as key signatures, that represent the key of the music, or time signatures, which indicates the rhythm of the music. A disadvantage of symbolic representation is that it can be hard to truly represent how a music piece should sound with only symbols, which may be more apparent in a live performance. One of the most significant developments in computer generated music is the Google Magenta Transformer (Huang et al., 2018), which uses transformers (an expansion upon LSTMs and RNNs) to generate symbolic music sequences up to lengths of a minute-long. Another successful system using symbolic representation is OpenAI's MuseNet (Payne, 2019) once again using transformers to generate compositions up to 4 minutes long with 10 different instruments.

There has been research comparing different computer music generation methods, Siphocly, El-Horbaty and Salem (2021)'s work compares the top 10 AI algorithms for music generation including methods such as rule-based algorithms, genetic algorithms, deep neural networks and more. This paper helps to understand the strengths and weaknesses of these different approaches. Additionally, (Hsu and Greer, 2023)'s work in comparing Markov

models and recurrent neural networks (LSTMs) in jazz music generation. These two papers assisted me in deciding which method to use for our music generation. This paper and Payne (2019)'s implementation of recurrent neural networks helped me in our own implementation.

An important factor in AI music generation is the choice of dataset used for training and evaluating the AI models. One popular dataset used for training music generation models is the JSB Chorales (Boulanger-Lewandowski, Bengio and Vincent, 2012) dataset, which consists of 500 MIDI files from Johann Sebastian Bach's (JSB) chorales. Another commonly used dataset is the MAESTRO dataset (Hawthorne et al., 2019), this contains both MIDI and audio representations and has been used in projects such as MuseNet (Payne, 2019), PerformanceRNN and more.

Different approaches tend to use different tokenisation methods, this aims to encode the musical information in a way that is understandable by machine learning models. MidiTok (Fradet et al., 2023) is a python library that can tokenise MIDI files through various MIDI tokenisations (REMI, TSD, MuMIDI and more). Another example is in (Hsu and Greer, 2023)'s work where they made a custom simple tokeniser. In our work, we decided to improve this tokeniser to help it suit our application further.

While the methods presented have shown significant progress in the field of computer music generation and many ways to customise this generated music, there currently exists no way to alter the difficulty of the music generate for learners. This limitation shows a weakness in these systems for music education.

3 Aims

The primary aim of our project is to research and develop an AI-driven system for generating piano music adjustable difficulty, specifically tailored for effective piano practice. This project will involve the development of a custom music generation and classification model using LSTMs, which will

be the foundation of our system. A custom dataset will also be produced with MIDI files split into different classes (difficulties). This dataset will be used for training our system. Our project will be evaluated to test its effectiveness in generating compositions with adjustable difficulties.

By achieving these aims, our project will contribute to the area of AI generated music and music education with the ability to personalise music for practice.

4 Implementation

Our AI music generation system was implemented in Python, a commonly used programming language for machine learning. We used PyTorch for machine learning tasks, Music21 for music processing and analysis alongside libraries such as NumPy, Matplotlib and more.

4.1 Dataset Creation

As there existed no suitable dataset with graded piano MIDI files, we had to collect and organise the data ourselves, this process involved several stages to ensure the data was correctly formatted and relevant to the research objectives.

Our first step was compiling a list of graded piano pieces from grades 0-8 from both the Trinity and ABRSM syllabuses. These syllabuses offer piano pieces for each grade that are used in examinations for students. This provided me with a clear structure for categorising the difficulty levels of piano pieces. Our next step was collecting the MIDI files of the appropriate pieces. we were able to use Classical Archives[1] to help locate a lot of the publicly available MIDI files. We were also given special access to a collection of 15,000 MIDI files from Classical Archives, which made it easier for me to find the necessary files.

[1] Available at `https://www.classicalarchives.com/midi.html`

Additionally, we used the website Piano Syllabus[2] which is a database of community graded piano pieces, we used this to help find further data examples for our dataset in combination with the Classical Archives files.

Once we collected these MIDI files, we had to ensure the quality and usability of each MIDI file. This was to guarantee that they would be suitable for use with our models. The next step was to meticulously organise each file according to the corresponding grade, where they were placed into the appropriate folder and labelled with the title and composer. This helped to be able to make sure it was easy to retrieve specific pieces for evaluation purposes.

Throughout the data collection process, we ensured that we complied with all ethical and legal standards in regard to the MIDI files. This involved checking that all files collected were publicly available and able to be used without acquiring specific licences. Through the meticulous labelling of the files in the dataset, we can easily locate any file in the dataset if necessary.

4.2 Tokenisation and Parsing

In order to process the MIDI files in our dataset, we used music21[3] which is a Python library useful for various kinds of musical analysis and processing. This allowed us to parse our MIDI files into a format that allows me to extract relevant information such as pitch, duration, etc.

For each MIDI file, we performed several methods inbuilt into the music21 library to simplify the extraction of the musical elements. One of the issues we encountered during the coding of the parser was that music21 streams (essentially a list, used for storing the music) can have multiple parts and voices. This caused issues initially within our parser and generated music when notes that were supposed to be played together were processed separately, causing the represented music to be erratic and inaccurate. This was solved by using the `chordify()` function from Music21, which simpli-

[2]Available at `https://pianosyllabus.com/x-default.php`

[3]Documentation available at: `https://web.mit.edu/music21/`

fies complex scores with multiple parts and combines them into one part that represents the whole score. This allowed me to capture more accurate information about each MIDI file during parsing.

An issue we encountered as a result of the `chordify()` method was that tied notes were difficult to accurately represent due to how the music21 package handles them. A method called `stripTies()` can locate tied notes, strip the ties and add the duration of the previously tied notes. This should result is no audible change. This method was unable to be used on chords due to limitations within the music21 package and how it manages tied notes. To solve this issue, we created a loop that would do something similar to the `stripTies()` method but applied to chords. This piece of code solves most issues and is able to strip most notes, but can sometimes not strip everything. This causes some slight differences in the represented music, but is not significant to the result of the project. Given more time, we would work on solving this issue.

Another issue was that the individual notes in a chord from the music21 stream all have the same duration.

Our tokenisation process is an expansion of the simple tokeniser in Hsu and Greer (2023) where we include more data during parsing, this includes chords, tempo marks, keys and more. This expansion was important as it allowed me to more accurately represent the MIDI files in our dataset, as these details can be crucial for determining the difficulty of a piece.

A problem with the tokenisation process implemented by Hsu and Greer (2023) is that for each chord, only a single duration is stored for the whole chord. This was an issue for our updated parser as when we use the `chordify()` method and the altered `stripTies()` loop, we will represent our full piece as chords, so it's very likely the notes in these chords should have different durations. During testing, we noticed that the represented music was inaccurate to the original. As a simple remedy to this, we store an individual duration for each note in the chord, along with the corresponding pitches.

When parsing and tokenising, a dictionary of unique tokens is created

alongside lists of the tokens in order that represent each MIDI file. After this, the dictionary and list of sequences are outputs for the function.

4.3 Token Processing

It was necessary to process the generated tokens to allow them to be used in the neural network. First, each unique token is mapped to an integer that references its index in the dictionary of tokens. Then the tokens in the sequences were converted to the corresponding integer, a necessary step. Once this was complete, the sequences were split into sequences of 8 long. Splitting our music into sequences of the same length allows our training data to be suited for processing in batches, and also will lower memory requirements. A longer sequence length would've allowed the model to capture more context which could improve performance, however, this does increase the time taken per iteration. **Figure 1** shows the relationship between these performances and runtime. A sequence length of 8 was chosen in our system as it was a solid balance between performance and training speed. To obtain the output data for training, we used a similar method described in Hsu and Greer (2023) and Payne (2019)'s work, where we use a sequence of tokens to predict the next token. In our system, we used a sequence of 8 notes to predict the 9th note. **Figure 2** is used to help display this.

After splitting our music into sequences of length 8 and obtaining the input-output pairs, we reshape our input to make it suitable for batch processing within our neural network, alongside converting the arrays to tensors (a necessary step for our PyTorch model). Next is to normalise our input values to keep them between 0 and 1. After these imperative steps, we can organise our data by putting them into a PyTorch DataLoader object, which helps to make batch processing easy during training.

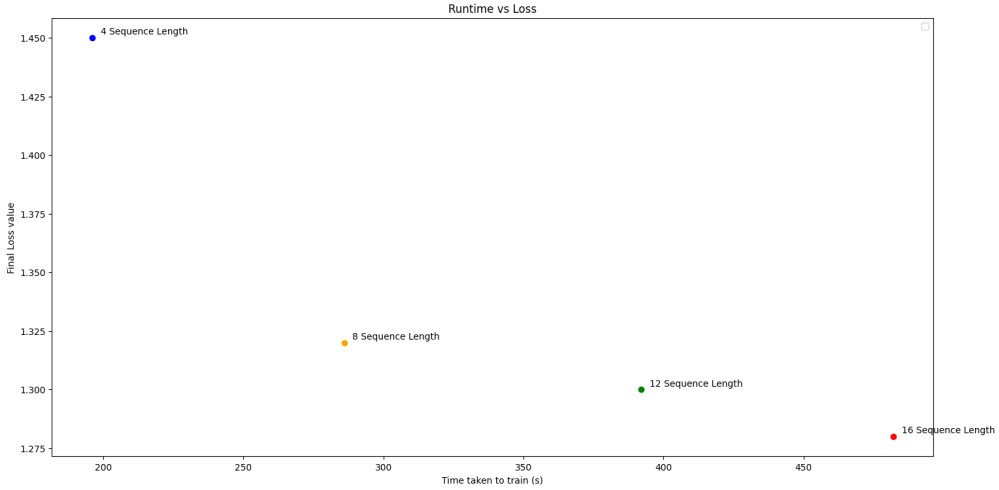

Figure 1: Graph displaying the relationship between Runtime and Loss with different sequence lengths

4.4 Model Implementation

This section will include the implementation of our two models, the generative and classifier model.

4.4.1 Music Generation Model

The MusicRNN model is designed for generating music events based on input sequences of music events. It uses LSTM cells to allow the model to capture long-term dependencies in the music. The model architecture consists of bidirectional LSTM layers followed by fully connected linear layers. Bidirectional LSTM's were used as we believe capturing dependencies in forward and backwards directions would help the model to learn more relationships, as there can be musical patterns in both directions in piano pieces. During training, we give 8 long sequences and during generation, we give the model a seed to start with. Dropout is introduced between the LSTMs layers to help reduce over-fitting. During generation, we applied softmax to the output to receive probabilities, then applied a probability distribution to obtain

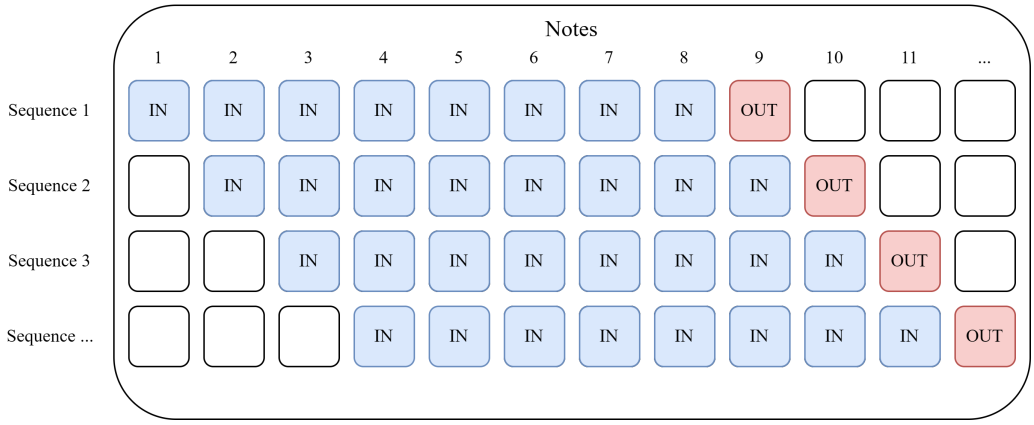

Figure 2: Diagram explaining how the sequences represent the full music.

a generated note.

4.4.2 Difficulty Classifier Model

The difficulty classifier model is created similarly to the generation model but instead of taking a specific sequence length, the input is a sequence corresponding to a music piece. As the training data could have varying lengths, we use pad the sequence to ensure they are all the same length.

Another significant different from the music generation model was that the classifier model has more fully connected layers, with ReLU activation between each layer. We believe that trying to categorise music difficulty with our limited data would be challenging for a single LSTM layer, so adding more linear layers helped to improve the performance of the model in classifying difficulty.

4.5 Hyperparameter Tuning

In our system, there are many hyperparameters that affect the performance of both of the models. In the generation model, the general loss and accuracy is important to ensure the generated music is cohesive, despite the music quality not being the focus of this project. An over-fitting generative model

may lack diversity, whilst an over-fitting classifier model may be unable to generalise and adapt to unseen data.

4.5.1 Music Generation Model

- Hidden Size: Hidden size is an important parameter, as it determines how many features can be represented in the hidden state. A higher hidden size can help to capture and learn complex patterns, (Merity, Keskar and Socher, 2017) however it does increase computational complexity and runtime. we chose a hidden size of 384 for our model, as our testing proved that 384 was a good balance between runtime and performance. Figure 3 and 4 help to show this.

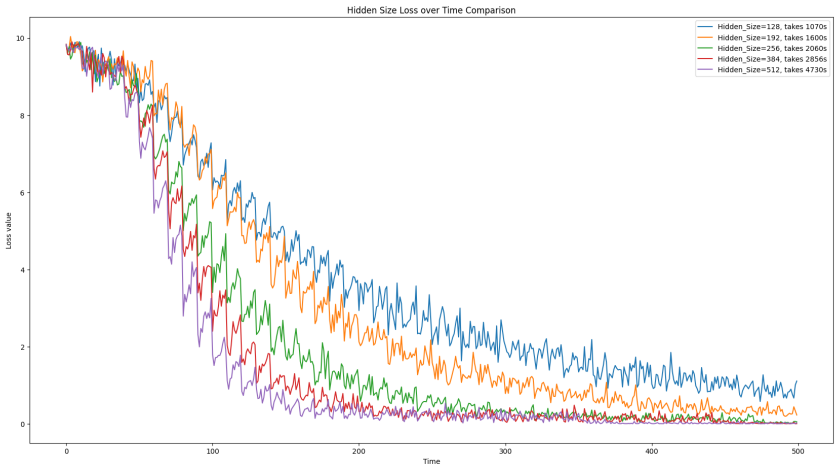

Figure 3: Graph displaying the relationship between loss and runtime when changing hidden size

- Number of LSTM layers: The number of stacked LSTM layers can improve performance and allow the model to capture more complex relationships, but once again can increase training time. We found that using 3 LSTM layers was a perfect value, as increasing this didn't see any significant improvements in performance.

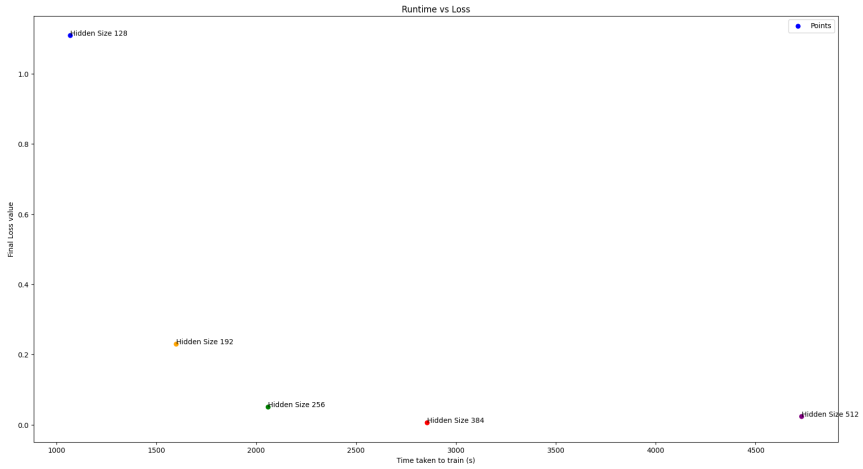

Figure 4: Graph displaying the relationship between final loss value and runtime for different hidden sizes

- Dropout: Dropout is a regularisation technique that is used in preventing over-fitting by randomly setting some units to zero during training, the dropout is implemented between each stacked LSTM layer. Preventing over-fitting helped the generated music to be less predictable. Cheng et al. (2017)'s work found that a dropout value of 0.3 worked well for their experiments, and we found that it also worked well for our system.

- Epochs: The number of epochs refers to the number of times the entire dataset is given to the neural network. Increasing the number of epochs. In general, training on more epochs can improve performance, but too many epochs could result in over-fitting. We trained our model on 100 epochs as we had limited data and more epochs would've meant clear overfitting of our model.

- Optimiser and Loss Function: When training our generative model, we used the Adam optimiser (Kingma and Ba, 2014), a commonly used optimiser in deep learning and was used in Hsu and Greer (2023)'s work

when implementing RNNs music generation. When trying to apply a similar optimiser such as SGD (Stochastic Gradient Descent), it failed to match the performance that Adam had. Additionally, the CrossEntropyLoss function was used, as the model is technically a multi-class classification system.

4.5.2 Difficulty Classifier Model

- Hidden Size: When trying to optimise our hidden size, we opted to keep our hidden size the same as our generative model at 384 as we noticed it was performing well without having to test different values.
- Number of LSTM layers: When increasing the number of LSTM layers in our difficulty classifier (values above 2), we noticed behaviour where even if the test accuracy was high, the model didn't perform well in practice. When setting the number of layers to 1, this behaviour was resolved.
- Dropout: As our model only used a single LSTM layer, we could not implement dropout in the same way as the generative model. Therefore, to help prevent overfitting, we added dropout between the linear layers.
- Epochs: During testing, we noticed that the performance started to plateau after 20 epochs and started overfitting. As a result of this, we set the number of epochs to only 20. Implementing early-stopping would allow this to be done automatically.
- Optimiser and Loss Function: Once again, the Adam optimiser and CrossEntropyLoss function was used, as it's a multi-class classification problem.

4.6 Training and Generation

The models were trained on a GPU using PyTorch's CUDA support, this allowed me to speed up training time compared to just using a CPU.

4.6.1 Music Generation Model

When undertaking the final training on our model, we chose not to use a test or validation set as the result of this music generation was not the focus of the project, therefore no validation was necessary. Additionally, with our limited data, it was a priority to maximise the amount of training data. This model was trained for 100 epochs at a batch size of 64 which took $\tilde{4}$ hours.

Using the trained generative model, we then move onto generating. A random seed is taken which has 16 notes from a given piece. This seed serves as a starting point for the generation process, as we have initial notes for the prediction process. We then iterate through a number of steps equal to the number of notes we want to generate, in our case we chose 200 notes. Similar to the training process, we used 8 note sequences to generate the 9th note. This 9th note was generated by applying a Softmax function to the output of the model to obtain probabilities, then applying a probability distribution over these probabilities. We chose to do this compared to other methods of just selecting the highest probability as it would add more variety to the output even if the model was slightly overfitting. Once we choose a note, it is used in the next sequence of 8 notes for further generation, alongside being decoded and added to a list of generated notes. This process is repeated until we have generated all notes. The output of this process is a decoded sequence of notes alongside an encoded sequence of notes for use with the classifier later.

4.6.2 Difficulty Classifier Model

The classifier model is trained on the same data as the generative model, where instead of using sequences of length 8, we used the full individual pieces. This model was trained for 20 epochs at a batch size of 16 which took $\tilde{5}$ minutes.

Once the model was trained, we were then able to apply the model to the generated pieces. The user will input a value from 0-8, corresponding to the difficulty of the generated piece. The system then generates a piece

where this model will classify its difficulty, if the classified difficulty is not the same as the requested difficulty, the system will generate a new piece to be classified until the difficulties match.

4.7 Post Processing/MIDI creation

Once we have our decoded sequence of generated notes, we give this sequence to a function to create the midi elements from the tokens. This is done using the Music21 library once again where we translate each token sequentially into the relevant music21 MIDI element, and we insert the element at the correct time. For example, to create a chord object, each note was created with the corresponding durations and pitch. These notes were then placed into a chord object which was added to the stream. During development, our midi creation function had to adapt to the developments we made to the parser, the chord object creation explained above is an example of an adaption that was made as a result of the improved parser. Once the full stream has been created, the stream is converted to a midi file through a simple line of code: `output_stream.write('midi', fp='output_file.mid')`. This midi file can then be displayed in an external program such as MuseScore [4]

We were able to test the performance of our parser and MIDI converter by directly connecting them and comparing the original input to the parsed and converted output. This helped us to tweak our system to represent the input data more completely.

5 Results

The results generated from our system has shown the ability to have varied difficulties when the user requests it. The generative model can produce diverse music within the difficulties, so not all difficulty 0 pieces sound the same, for example.

[4]Available at `https://musescore.org/en/download`

There are sometimes often clear differences between generated music of vastly different difficulty (such as a grade 0 vs grade 8). However, it can be difficult to notice the different between two consecutive grades of difficulty (such as grade 1 vs grade 2). This weakness highlights the complexity of modelling musical difficulty and the need for improvement in this area.

Additionally, the raw generated music itself might not be suitable to practice with due to certain artefacts, introduced by the parser, may inflate the perceived difficulty of the piece.

6 Evaluation

Quantitatively evaluating difficulty can be difficult, as there are many factors that can go into determining the difficulty of a piano piece. However, one way that could be used is analysing the number of different pitches within the music using pitch class entropy.

Pitch class profiling involves counting the number of occurrences of each pitch within a music piece. This is useful to gain insight into the harmonic characteristics of the music. A pitch class histogram can be used to display the number of pitches graphically. A musical piece with more varied pitches may indicate greater difficulty, compared to a less varied piece that may be easier as it uses the same notes more often. A 12-dimensional pitch class histogram $\overrightarrow{h}$ is often used to display this, which contains the number of each note stored.

Pitch class entropy, used in Wu and Yang (2020)'s of evaluating AI-composed music, is a measure of uncertainty or randomness of the pitch distribution. We believe that pitch class entropy could be used as a method to analyse to measure the tonal difficulty of a piano piece. A higher entropy indicates a higher variability within the notes of the piece, which could suggest greater difficulty compared to a piece with low entropy, which would consistently use the same notes. A piece that consistently uses multiple notes would indicate more complex musical techniques, such as changing keys dur-

ing the piece or using notes from outside the current key. To calculate the pitch class entropy, the pitch class histogram $\overrightarrow{h}$ is normalised so that the resulting histogram has a sum equal to 1 as a probability distribution. The entropy $H(\overrightarrow{h})$ is then calculated with this equation:

$$H(\overrightarrow{h}) = -\sum_{i=0}^{11} h_i \log_2(h_i)$$

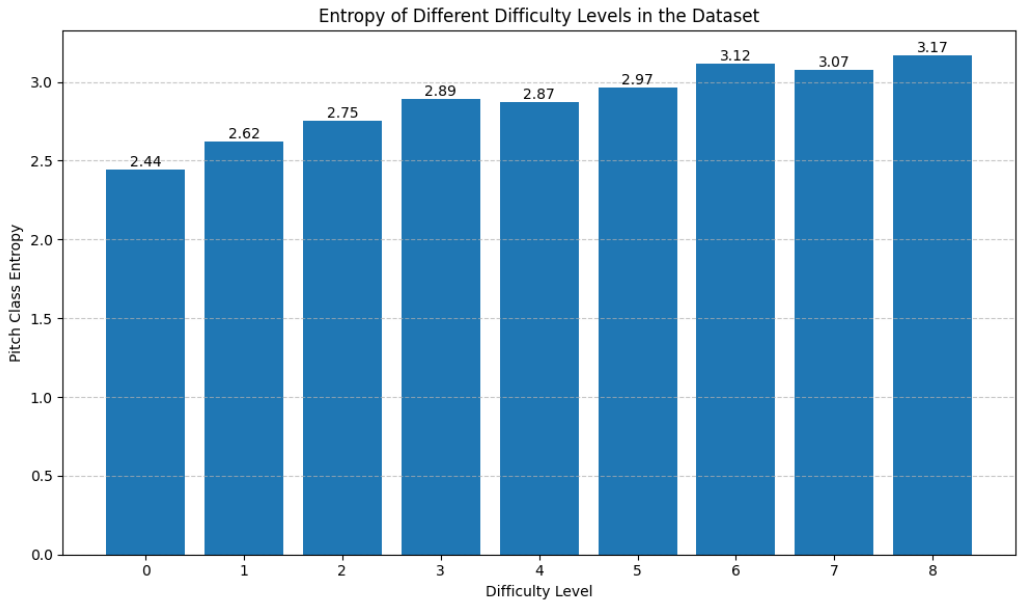

Figure 5: Graph displaying the relationship between pitch class entropy and difficulty in real examples.

In **Figure 5**, we can see that as difficulty increases, the entropy tends to increase. This trend helps to highlight that entropy could be used as a valuable metric for quantifying the complexity of piano pieces and evaluating AI-composed music.

We chose to compare the entropy of real piano pieces versus the piano pieces generated by our system for each grade. **Figure 6** shows that, on average, the pitch class entropy from the generated pieces in higher than the entropy of the real pieces taken from the dataset. This is expected as our model was trained on minimal data and did not focus on the quality

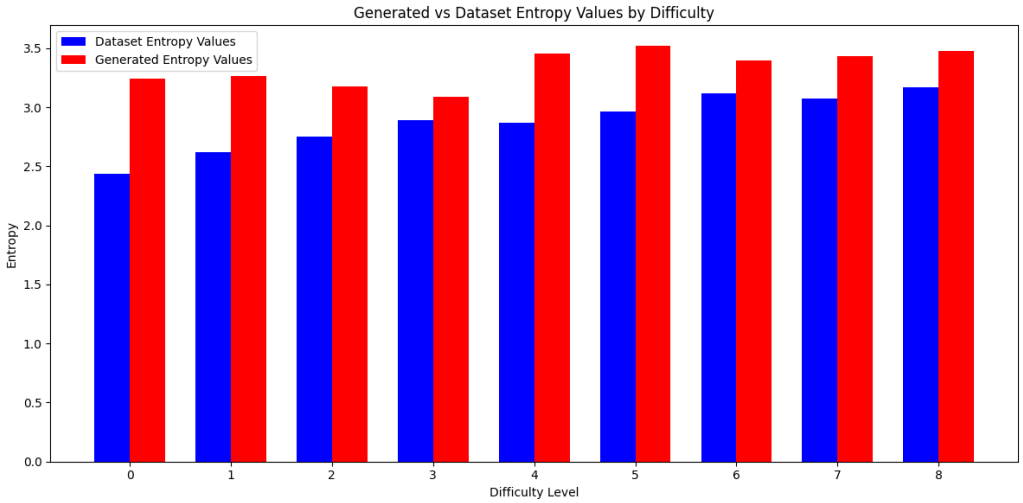

Figure 6: Graph displaying the difference between entropy in real data compared to computer generated music.

of the music. We can also see that the entropy values for the AI-generated music does not follow the linear increase in entropy as difficulty, instead we can observe two groups of similar entropy values on the graph. From the difficulty levels 0-3 we see an average entropy value of 3.19 whilst from 4-8 has an average entropy value of 3.45. This may show that instead of being able to classify into 9 separate difficulty levels, our current model is more suited to classifying into two distinct groups of difficulty: low difficulty (0-3) and high difficulty (4-8). This shows that there may be limitations within the model, and its ability to capture the fine differences between each difficulty level.

Whilst pitch class entropy is a useful metric to assess the tonal complexity of different piano compositions across various grades, there are many other factors that go into determining the difficulty level of a composition.

One metric that could be used to evaluate the variation in rhythm would be the average number of tempo changes per piece. These tempo changes can increase the perceived difficulty of a musical piece. A piece with lots of tempo changes may require the performer to constantly adapt and change

their playing style to suit the new tempo. We can see in **Figure 7**[5], as the difficulty increases, the average number of tempo changes per piece increases. This metric was measured by counting the average number of tempo changes per piece across the 9 difficulty levels.

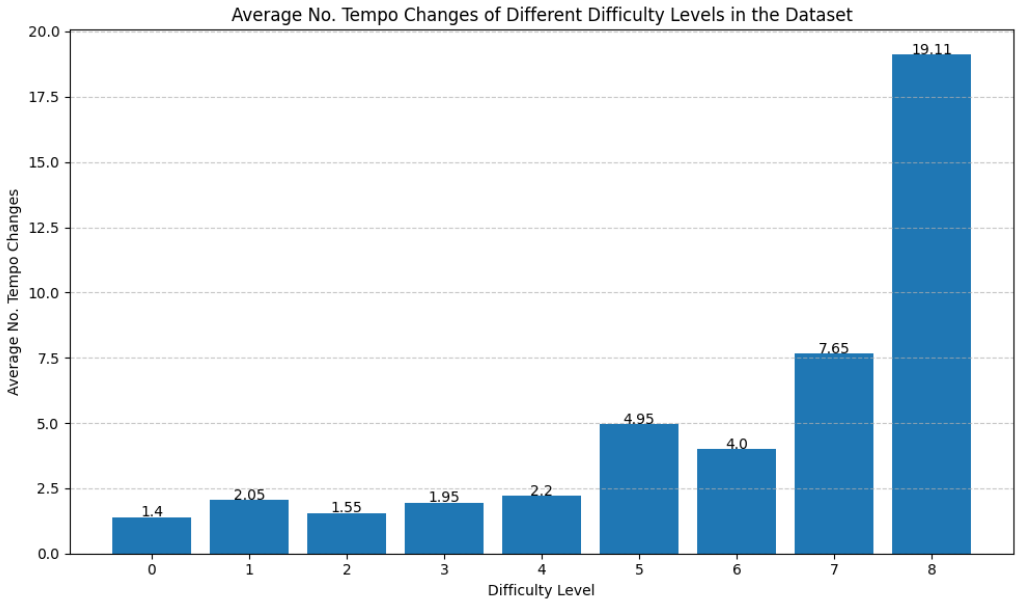

Figure 7: Graph displaying the relationship between the average no. of tempo changes and difficulty.

If the computer generated music followed a similar trend to the trend shown in **Figure 7**, then it would be an indicator that our system can represent the average tempo changes for each difficulty.

Figure 8 shows that our computer generated music does not follow the steady increase in tempo changes as difficulty increases. Instead, we once again see two groups of difficulty with a higher average number of tempo changes in the more difficult group (4-8) compared to the easier group (0-5).

[5]Grade 8 was adjusted to account for an outlier with 1000 tempo changes, this piece happened to use individual BPM markers for each note when slowing down or speeding up. This is a very uncommon way of notating this tempo change. This outlier was removed when calculating the average.

This shows that once again our model is capable of distinguishing difficulty but is unable to consistently differentiate between two consecutive difficulty levels. This behaviour could be a result of how our model processes the tokens representing tempo changes. These tempo changes are represented just like notes or chords and do not represent their significance to the model. With more data, it is likely the significance of these tempo changes would be realised however with our limited data, this is not the case.

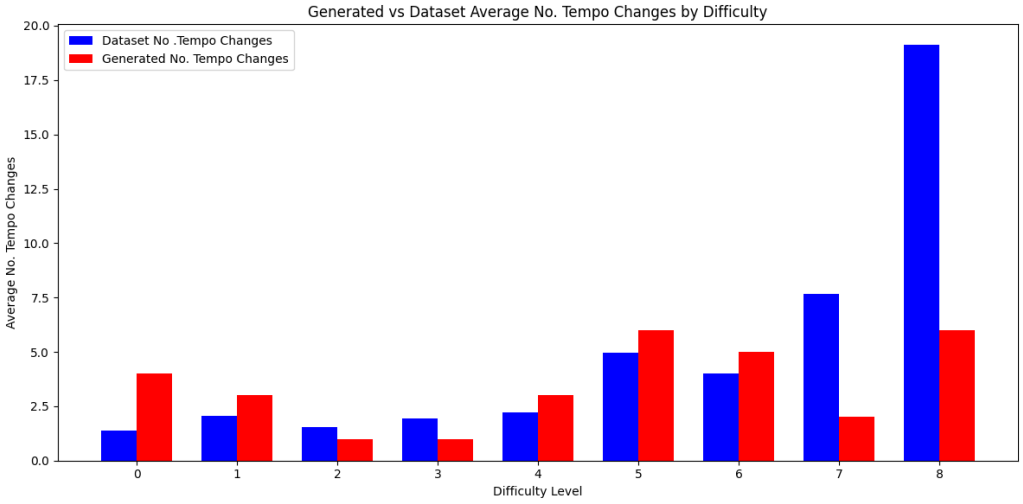

Figure 8: Graph displaying the difference between the average no. of tempo changes in real data compared to computer generated music.

The final metric we used for evaluation was the altered note rate, which was introduced in the work by Sébastien, Sébastien and Conruyt (2011). This metric is the proportion of altered notes in a piece, altered notes refers to when a note has an accidental (sharp or flat) that is outside the key. The altered note rate provides information on the tonality and harmonic complexity of a piece. In general, a higher altered note rate indicates greater tonal complexity, as more notes may extend outside the key. This can be complex to play as the majority of notes that will be played would be within the key. Analysing this metric will help to determine the pattern between altered note rate and difficulty, alongside comparing our results to real world

data using this.

Figure 9 shows the linear relationship between the average altered note rate and difficulty of real piano pieces. As shown in the figure, there is a clear trend indicating that as the difficulty level raises, the average altered note rate also increases. This trend reinforces the idea that at higher difficulty levels, there is higher harmonic complexity.

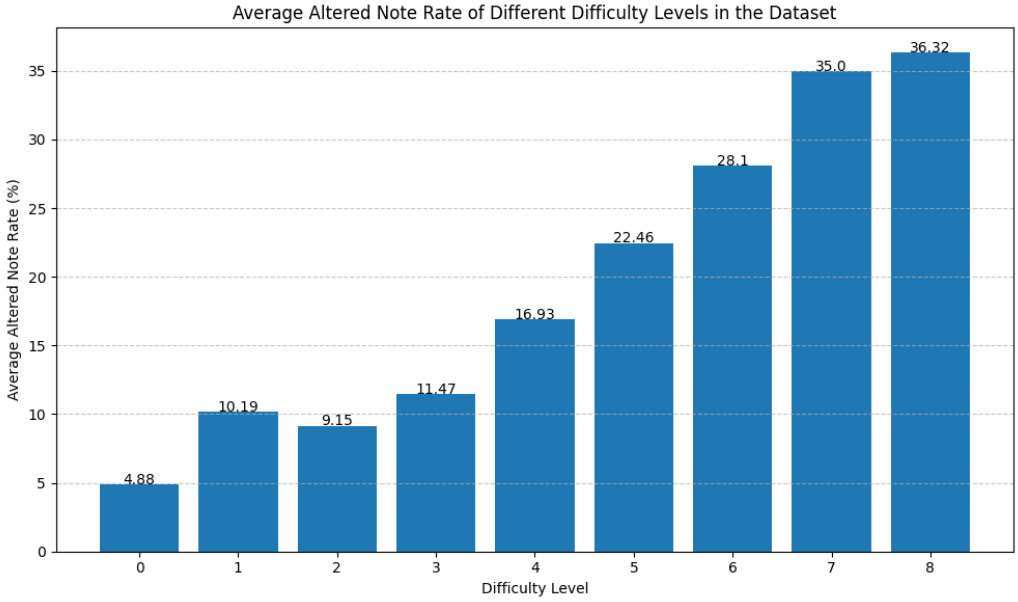

Figure 9: Graph displaying the relationship between the average altered rate and difficulty.

When comparing our computer generated music to real world data taken from our dataset, we can see that the computer generated music once again shows two groups of difficulties with different mean altered note rates. **Figure 10** shows this comparison clearly. We can also observe a linear decrease in altered note rate from difficulty grades 0-3. This anomaly is likely due to small sample size. This analysis shows that our system is once again capable of distinguishing two difficulty groups, but is incapable of classifying each grade 0-8.

Alongside evaluating different metrics, there are different things that can

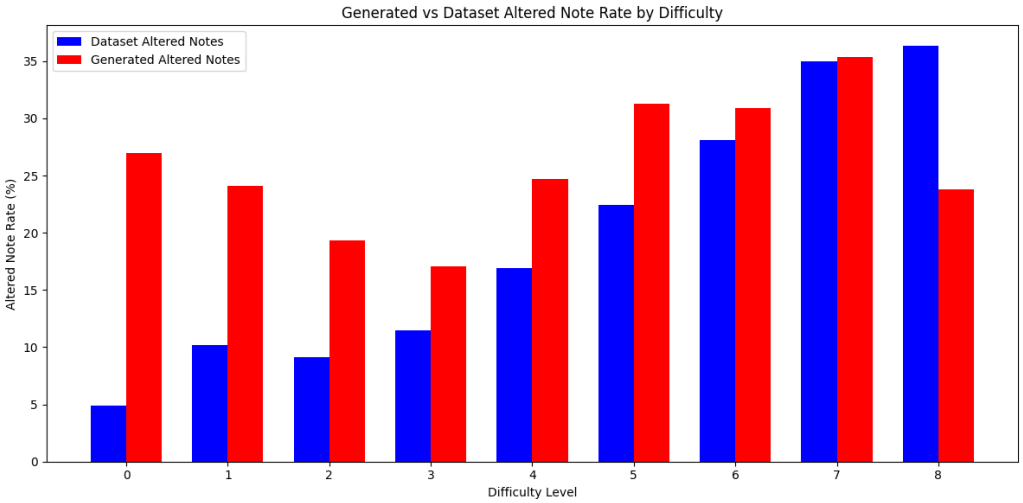

Figure 10: Graph displaying the difference between the average altered note rate in real data compared to computer generated music.

be noticed when evaluating the success of our generative music. Using MuseScore 4 to visualise the output MIDI files from our system as sheet music, we noticed that there can be excessive, very short rests scattered throughout the sheet music. These rests, although minor, can contribute to the noise of the sheet music, causing confusion and potentially needed rectifying or ignoring by the musician to play the piece. There are also, occasionally, disproportionally short and erratic notes across all difficulties. These notes can present a challenge for a beginner musician to try to play when trying to practice using this AI-generated music. It is possible that using a program to visualise the MIDI files, such as Synthesia[6] these rests and short notes may seem more playable as programs like these allow for less refined playing of pieces. However, sheet music is the standard way to practice and learn piano music and would be expected during performance exams.

However, we can also note a clear difference in playability between a grade 0 and grade 8, but a significant difference was not clear between adjacent difficulty levels. This reinforces the idea that our model is more able to

[6]Available at `https://www.synthesiagame.com/`

generate compositions at 2 distinct levels compared to 9.

7 Conclusions

In conclusion, our work has given a comprehensive explanation on generating piano music with varying levels of difficulty using AI. Through the implementation of a generative music model along with a difficulty classifier model trained on a custom dataset, this paper demonstrates the system's capability to produce and classify piano pieces that vary in difficulty.

The custom dataset serves as a solid foundation to build similar AI-powered music generation for piano music. Additionally, this may have uses outside of machine learning.

The aim of the project was to generate piano music for effective practice. We believe with small adjustments with the parser, to eliminate issues involving short rests and erratic notes, the current model will generate music that is playable and can be used for practice. Additionally, the aim was to generate piano pieces from each difficulty 0-8 however the current model may be more suited to generate a low and high difficulty piece.

There are aspects which are not taken into consideration when tokenising and processing the data, such as dynamics or articulations. Not having this data may affect the ability to accurately model difficulty further, as well as limit the capability of the ability to generate varied music.

Additionally, this project has helped to improve our own understanding around concepts such as recurrent neural networks and music processing within programming. Given more time, I would be interested in learning about implementing transformers and other types of systems for generating music.

Whilst our system shows promise, there remain areas for improvements and refinement.

7.1 Further Work

Despite the achievements made by this project, there is still room to improve and refine on this system.

Firstly, increasing the size of our dataset would help to improve the model's ability to learn by introducing more training data alongside improving evaluation. Having only 20 files per difficulty limited the ability of the model, especially the classifier, as trying to accurately model difficulty may be exceptionally difficult without significant data. Improving the parser's issues would be another way to enhance the performance of the model, especially with the playability of all difficulties due to the erratic notes. Additionally, enhancing the tokenisation process to include more information such as dynamics or articulation could help to improve the models' ability to generate varied music, alongside improving the difficulty classification. Continued optimisation of the hyperparameters would help to maximise the performance and efficiency of our models. Additionally, exploring different architectures such as transformers may help to improve the overall performance of the system whilst adding attention mechanisms to help the models' learning of long term dependencies.

Overall, by addressing these key issues, we can improve the practicality of our AI-driven music generation system.

8 Acknowledgement

Special thanks to Classical Archives and Piano Syllabus for providing access to their extensive databases, which significantly contributed to the success of this project.

Thanks to Dominique Chu for providing continuous support and supervision during this project.

References

Boulanger-Lewandowski, N., Bengio, Y. and Vincent, P. (2012). Modeling temporal dependencies in high-dimensional sequences: Application to polyphonic music generation and transcription. *Proceedings of the 29th International Conference on Machine Learning, ICML 2012*, 2, pp. 1159–1166.

Cheng, G., Peddinti, V., Povey, D., Manohar, V., Khudanpur, S. and Yan, Y. (2017). An exploration of dropout with lstms.

de León, P. J. P., Iñesta, J. M., Calvo-Zaragoza, J. and Rizo, D. (2018). Data-based melody generation through multi-objective evolutionary computation. *Machine Learning and Music Generation*, pp. 87–106.

Fradet, N., Briot, J.-P., Chhel, F., El, A., Seghrouchni, F. and Gutowski, N. (2023). miditok: A python package for midi file tokenization.

Gers, F. A., Schmidhuber, J. and Cummins, F. (2000). Learning to forget: continual prediction with lstm. *Neural computation*, 12, pp. 2451–2471.

Hawthorne, C., Stasyuk, A., Roberts, A., Simon, I., Huang, C.-Z. A., Dieleman, S., Elsen, E., Engel, J. and Eck, D. (2019). Enabling factorized piano music modeling and generation with the MAESTRO dataset. In *International Conference on Learning Representations*.

Hsu, C. and Greer, R. (2023). Comparative assessment of markov models and recurrent neural networks for jazz music generation.

Huang, C.-Z. A., Vaswani, A., Uszkoreit, J., Shazeer, N., Simon, I., Hawthorne, C., Dai, A. M., Hoffman, M. D., Dinculescu, M. and Eck, D. (2018). Music transformer.

Kingma, D. P. and Ba, J. L. (2014). Adam: A method for stochastic optimization. *3rd International Conference on Learning Representations, ICLR 2015 - Conference Track Proceedings*.

Luo, S. (2022). Bach genre music generation with wavenet-a steerable cnn-based method with different temperature parameters. *ACM International Conference Proceeding Series*, pp. 40–46.

Merity, S., Keskar, N. S. and Socher, R. (2017). Regularizing and optimizing lstm language models. *6th International Conference on Learning Representations, ICLR 2018 - Conference Track Proceedings*.

Payne, C. (2019). Musenet.

Sébastien, V., Sébastien, D. and Conruyt, N. (2011). Dynamic music lessons on a collaborative score annotation platform. In *International Conference on Internet and Web Applications and Services*.

Sherstinsky, A. (2020). Fundamentals of recurrent neural network (rnn) and long short-term memory (lstm) network. *Physica D: Nonlinear Phenomena*, 404, p. 132306.

Siphocly, N. N. J., El-Horbaty, E. S. M. and Salem, A. B. M. (2021). Top 10 artificial intelligence algorithms in computer music composition. *International Journal of Computing and Digital Systems*, 10, pp. 373–394.

Staudemeyer, R. C. and Morris, E. R. (2019). Understanding lstm-a tutorial into long short-term memory recurrent neural networks.

van den Oord, A., Dieleman, S., Zen, H., Simonyan, K., Vinyals, O., Graves, A., Kalchbrenner, N., Senior, A. and Kavukcuoglu, K. (2016). Wavenet: A generative model for raw audio.

Wu, S. L. and Yang, Y. H. (2020). The jazz transformer on the front line: Exploring the shortcomings of ai-composed music through quantitative measures. *Proceedings of the 21st International Society for Music Information Retrieval Conference, ISMIR 2020*, pp. 279–286.

Yang, L. C., Chou, S. Y. and Yang, Y. H. (2017). Midinet: A convolutional generative adversarial network for symbolic-domain music generation. *Pro-*

ceedings of the 18th International Society for Music Information Retrieval Conference, ISMIR 2017, pp. 324–331.