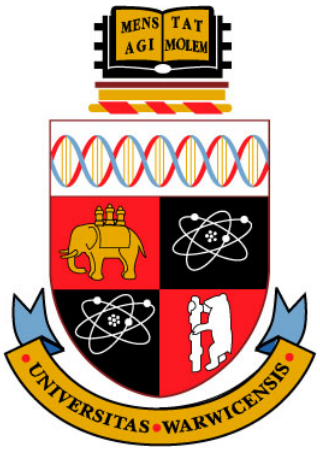


Automatic Generation of Graded Sight-Reading Music Using Neural Networks

CS310 Computer Science Project

Harry Fallows

Supervisor: Dr. Jane Sinclair

Department of Computer Science

The University of Warwick

2019–20

Abstract

Sight-reading is a skill practised by musicians to improve their ability to play new pieces, without the need for rehearsal. A student's sight-reading ability is tested in the majority of music examinations; the examination boards set specific requirements for what a student should be able to play at a given grade. This project aims to develop a system for generating sight-reading music using artificial neural networks. Existing projects for generating music using neural networks struggle to produce pleasant sounding and complex melodies. These issues should not affect the generation of sight-reading music significantly, since sight-reading pieces tend to be relatively uncomplex and monophonic. Long short-term memory networks were utilised for this project due to their ability to handle sequential data with long-term dependencies. The results show that when required to produce a single melodic line of notes, an LSTM network can produce suitable sight-reading music. Results from human evaluation show that the optimal generation method and network (trained with notes only) produces suitable sight-reading music 50% of the time for the given instrument grade. With the addition of slurring, the best reported suitability increases to 67%. The addition of more musical features causes deterioration in the network's ability to generalise over the training data.

Keywords: Neural network, RNN, LSTM, Sight-reading, Sheet music, Music generation

Acknowledgements

I would like to thank Dr. Jane Sinclair for her support with this project, and Dr. Victor Sanchez for his technical advice. I would also like to thank members of The University of Warwick Symphony Orchestra for their help with evaluating music produced throughout the development process. The authors of the Music21 Python library deserve recognition, since without it, this project could never have extended as far as it did. Finally, I would like to thank my mum, my girlfriend, my friends, my family, and Jane for helping me through the stressful and frustrating moments experienced during this project.

Musical Definitions

Definitions made with reference to the Cambridge Dictionary [1]

Stave: The five lines and four spaces on which notes and other musical notations are written.

Sheet Music: Music in its printed form consisting of one or more musical staves.

Bar: One of several equally divided sections of music, consisting of a number of beats specified in the time signature.

Time Signature: A symbol at the start of a piece of music consisting of two vertically stacked numbers. The top number represents the number of beats in a bar; the bottom number represents the length of each beat as a fraction of a semi-breve (whole note).

Key Signature: A number of sharps (♯) or flats (♭) at the start of a piece indicating the altered pitches of notes for a piece of music.

Key: A group of notes chosen based on their relationship with one particular note, indicated, but not defined, by the key signature.

Polyphonic: Music consisting of many different voices or parts.

Monophonic: Music containing a melodic line played by a single instrument or voice.

Octave: The range between two pitches, eight steps (or 12 semitones) apart. The higher of the two pitches has a frequency of double the lower one.

Contents

Introduction

Sight-reading is a part of graded musical instrument examination where the student is asked to perform a piece of music they have never seen before. The examinee is given a short piece of music with a maximum preparation time of half a minute, before being asked to play it [2]. Sight-reading is used as a testing parameter because it is an essential skill for a musician and can improve a player's ability to learn new pieces and perform in ensembles [3]. Out of the two main boards of music examination, sight-reading testing is mandatory from grades 1 to 8 for ABRSM, and mandatory from grades 6 to 8 for Trinity Guildhall [2, 4]. According to the ABRSM website, over 600,000 people take music examinations every year [5], so there is a large market for a product which can help with practising this skill. When preparing for a music examination, students often purchase sight-reading practice books containing short pieces of music similar to those that will feature in their exam. This project attempts to remove the necessity of purchasing these books by artificially composing new practice pieces.

1.1 Motivations

As a musician who has been through the process of music examinations, I understand the difficulty of finding appropriate music to practise sight-reading. This is in part due to the specific requirements set by the examination boards, and the lack of practice pieces published by them. An ABRSM-published sight-reading book contains roughly 24 pieces per instrument per grade [6], which are often rationed over the many months of preparing for a music examination. A common result of this is the re-use of old sight-reading pieces from old examination specifications or other examination boards. The problem with this is that there is no guarantee that they will fit the same requirements and provide an authentic replica of what a student may be given in an examination. Provided with a system for artificially producing sight-reading music, a student would be able to practise their sight-reading ability without worrying about running out of practice pieces.

1.2 Project Scope

From an early stage in the project, it was understood that developing a system capable of producing sight-reading music for all instruments and all grades would not be possible. This is due to the large number of combinations existing, which would require an extensive data collection stage spanning a

lengthy period of time. Because of this, the scope was defined as an implementation of a system capable of generating music from a dataset of pieces and a quantitative set of requirements defined by the examination board. This would then be evaluated using the dataset and requirements from one grade and one instrument. The chosen instrument was the violin because it is one of the most popular instruments for examination. There were 35,587 violin examination entries in 2009 alone [7]; the only instrument to surpass this was the piano whose sight-reading music contains two staves and is therefore not be considered in this project. Another reason for choosing the violin is that it is very similar to other stringed instruments and the results from producing violin music would indicate that similar results could be achieved with the viola, cello and double bass. The ABRSM stringed instrument sight-reading specification includes all stringed instruments, each with very similar requirements [3]. Grade 5 was chosen as the grade instance to evaluate the system on as it would give a good middle-ground of how the system will perform on grades ranging from 1 to 8.

The specification stated that the objective of the project was to create a neural network that could be trained on graded sight-reading music, have the network then produce music of a similar standard, and output the music in sheet-music format such that it could be played by a musician [8]. The document further describes the necessity to minimise the visible difference between computer-generated music and the human-composed music from the original dataset.

The progress report for this project describes the initial focus on generating melodies that accurately depict those that might feature in a piece of sight-reading music [9]. This can be separated into two features that the network would need to take into account, the pitch of the notes and the length of the notes. The report further highlights the necessity for this to be expanded to other musical features, such as dynamics, in order to meet the specification set by the examination board. This is not strictly true since these features could be added on later by a human; a grade-specific melody is still a viable output since the majority of the composition process has been done. However, the addition of musical features has been attempted within this project to varying degrees and with varying results. This was achieved through the conceptualisation of a hierarchy of features, describing each musical feature by its importance in a piece of sight-reading music and the degree to which its inclusion optimises the overall composition process. This is discussed further in section 6.6.

Potential extensions were also suggested in the specification [8]. These include producing a user interface (UI) for displaying generated pieces of sight-reading music, integrating a scanning feature allowing a user to expand the training dataset, and a classification network capable of classifying existing pieces as a specific grade of difficulty. These extensions were deemed out of scope due to the time constraints of the project. The UI and the piece-classification network are discussed in section 9.1 of this document as areas of interest for future development. The scanning feature was seen to not be necessary due to the existence of a similar, suitable piece of software called PhotoScore [10]. This software is an extension of the music composition program Sibelius [11] and enables the user to scan and import printed and handwritten music.

Background

2.1 Existing Sight-Reading Practice Tools

There are a couple of sight-reading tools available online for students to use to practise their sight-reading skills. One of these tools is a gamified app created by the ABRSM examining board called "Sight-reading Trainer" [12]. This app is exclusively designed for piano players and covers the content for grades 1 to 5. It contains games for learning short rhythms and allows students to track their progress. The concept of the app is very straight-forward and targets the specific areas of sight-reading that many students find challenging. Despite its benefits, the fact that it only supports the piano means that there is no examination-board-specific tool for students taking sight-reading examinations in other instruments.

Another tool available is a web-app for picking sight-reading pieces based on their difficulty and contents called "Sight Reading Factory" [13]. This tool allows the user to choose an instrument, an arbitrary difficulty level, and a number of other options before displaying a piece of music that fits those requirements. This is a very useful tool for students, as any set of requirements can be specified for the piece of sight-reading the student wants to be given. The only downside of this app is that there is no indication of how difficult a piece is with relation to an examination board or grade; it is useful for general sight-reading practice, but not for preparation for a graded examination.

2.2 Existing Music Generation Projects

Generating music is a relatively popular area of research within the artificial neural network community; there are many examples of researchers developing neural networks for mimicking their favourite artist or composer [14, 15, 16].

One example of this looked into the generation of video game piano music from the game "Final Fantasy" [14]. This project selected the note pitches and chords from each piece producing one very large array of note pitches, before splitting them into fixed-length sequences with a single output for training. Generating music was done by inputting randomly selected sequences from the training data, predicting a single output note pitch, appending the prediction to the input sequences, and then predicting the next note. This was the foundation of most music generation neural networks researched and may be referred to as "sliding window generation" in further sections of this document.

The problem with generating from an input selected within the training data is that in an ideal scenario where the network is able to learn the patterns of the compositions, the result of generation produces identical sequences to those that feature within the training data. The only scenario in which new music is generated is when the network is given the last input sequence from the training dataset and is essentially given the task of continuing the piece of music. Another issue with this project is that the only musical feature it takes into account is note pitch. There is no effort made to integrate rhythm into the training or generation; the author simply assumes all notes are of the same length and then applies an arbitrary fixed note length to any note produced. This may produce suitable results for the background music used when playing a video game, but would certainly be unsuitable for the task of generating interesting and pleasant sounding music. It would particularly be unsuitable for generating sight-reading music since rhythm is one of, if not the single most important aspect of sight-reading testing. Despite the overlooked aspects of this project, it is still able to produce variant music and demonstrates the capabilities of LSTMs for generating music.

Another example attempted to recreate music in the style of Led Zeppelin, from a dataset of MIDI files [15]. This project used a similar sliding window generation technique where a short context is provided to the network to stimulate generation. The input of the network is sequences of 128-dimension vectors, representing 128 possible note pitches, distributed over a number of time-steps (MIDI ticks). The network also takes into account the volume of each note played by utilising the "velocity" variable within the MIDI file format. Although this is relatively simple to implement due to the quantifiable aspect of volume within the file format, it would be more difficult when writing sheet music where dynamic changes are generally more sparse and playing volumes are somewhat subjective. There has been some experimentation with sequence lengths and the removal of silence within music; however, the results are generally either disorganised and unpleasant music, or consistent tones or periods of silence. When attempting to reproduce the iconic "Stairway to Heaven", the results "[do] not resemble the original song". Although a few variations of training and prediction were tested, there was only one variation of the neural network structure mentioned. Training multiple models in a grid search manner and providing some quantitative metric, such as the overall difference between a generated and original song, along with the subjective opinion of a listener could have potentially produced better results. A similar method was used to produce music from a dataset of classical guitar music [17]. This paper describes two different network designs, a standard multi-layer LSTM network and an autoencoder LSTM network. The best results obtained from this experiment came from a 2-layer LSTM network structure, where 2 seconds of polyphonic music was generated. These limited results were caused by the failure of the network to generalise over the variably structured and polyphonic dataset. This is not an issue that can be related to the task of sight-reading generation since the data is very uniformly structured and consists of purely monophonic melodies.

Another approach used a bi-axial LSTM configuration trained on Bach fugues along with some other classical piano pieces [16]. The network contained one axis for note pitches and another for time-step lengths, each with influence over the other. The time axis implementation was taken from another polyphonic LSTM music generation paper [18] and consists of a binary value indicating whether a note is articulated or sustained. This approach is interesting because it enables the network to produce notes of different lengths, unlike previous examples. The style of generation is similar to the "sliding

window" technique used in other examples; the LSTM note axis takes as input, a concatenation of activations from the previous note window and the activations of the last time axis layer of the note. The results from this experiment show that the bi-axial configuration managed to produce small extracts of comprehensible melody with some harmony and polyphony with a "local sense of coherence". The harmony and polyphony aspect is not important for this project due to the fact that, excluding piano, most solo instruments are only required to play a single note melody in a sight-reading examination. An issue brought up in this paper, which is relevant for this project, is the "sparse" and "significantly less consistent and coherent" music which is generated when there is a lack of training time and training data. This was a significant concern because, as described in the motivations section, a dataset of suitable sight-reading music does not exist. This concern, as well as how it was addressed, is further discussed in the sections 6.1 and 6.2.1 of this document.

Other methods attempted for generating music using neural networks include the use of adversarial training, both with LSTM [19], and also convolutional neural networks [20]. Generative adversarial networks are a class of neural networks that are designed to produce new content [21]. The adversarial approach consists of training two models, a generator for producing the new content and a discriminator for distinguishing between real and generated data [19]. With the convolutional approach, the results were reported to outperform their recurrent neural network counterparts in some aspects, although outputs were also reported to have the "occasional violation of music theory". With the LSTM approach, the adversarial training was attributed with helping the network "learn patterns with more variability" and enabling the network to produce music with better polyphony. Although these results do show benefits for utilising a discriminator network, this approach was seen to be out of scope. This is due to the additional complexity of implementation, and the limited perceived benefits in the realm of complex polyphony - useful for writing a symphony, but not for a short piece of monophonic sight-reading music.

2.3 Artificial Neural Networks

Artificial neural networks are computational models designed to loosely replicate the structure and functionality of the brain. Different types of neural networks can be used to tackle a variety of tasks in the same way a human or animal would, by learning from experience [22]. Basic feed-forward artificial neural networks consist of an input layer, a number of hidden layers, and an output layer. The layers of a neural network are made up of nodes; each layer is connected to the next with parameters called weights, which are adjusted during the training. The other adjustable parameters in a basic feed-forward neural network are the biases, which have the effect of increasing or decreasing the net input of the activation function of a node [22]. The activation function of a node dictates the relationship between its input and its output. During training, the network inputs are fed through the network using forward propagation; the output of a node is defined by the weighted summation of the outputs and the bias from the previous layer with an activation function applied to it. This is illustrated in Figure 2.1. Popular activation functions include sigmoid, hyperbolic tangent and rectified linear unit, each mapping the input of a node to the domains of ($x \in \mathbb{R}$) $0 < x < 1$, $-1 < x < 1$ and $0 < x < \infty$ respectively. Figures 2.3, 2.4 and 2.5 display these functions graphically with their corresponding

function definitions.

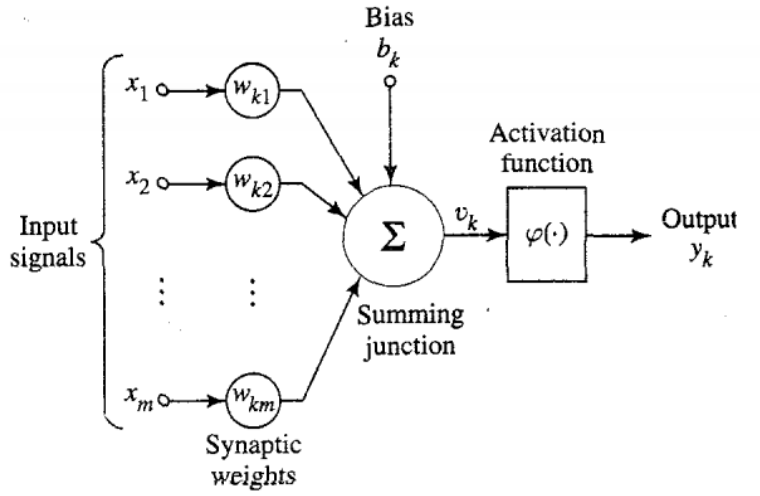


Figure 2.1: Nonlinear model of a neuron [22]

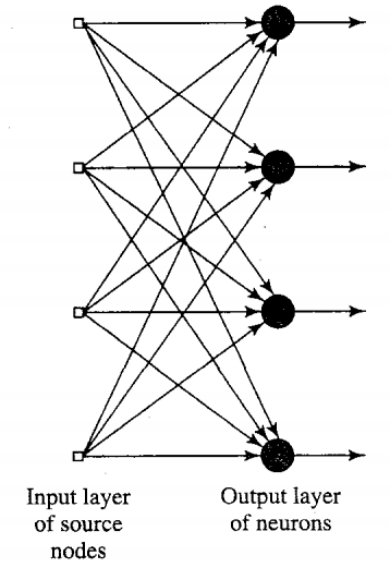


Figure 2.2: Single layer of a full-connected feedforward network [22]

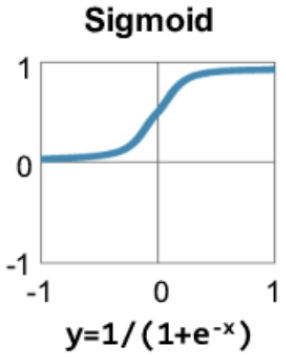


Figure 2.3: Sigmoid Function [23]

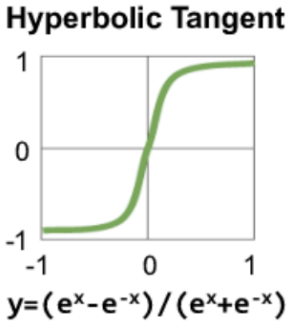


Figure 2.4: Hyperbolic Tangent (tanh) Function [23]

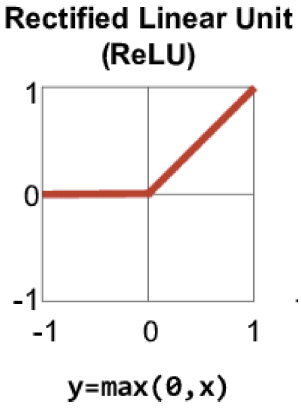


Figure 2.5: Rectified Linear Unit Function [23]

The performance of the network during training is measured using a loss function, which measures the difference between the output produced by the network and the target output from the training data. In order to adjust the network parameters effectively, a process called backpropagation is used. Backpropagation seeks to find a local minimum of the function describing the relationship between the loss function and each individual network parameter, by calculating its gradient. This is done by moving backwards in the network and repeatedly applying the chain rule to calculate the partial derivative of the loss function with respect to each network parameter. The gradient of each function

indicates the direction the parameter needs to be adjusted in order to minimise the loss function [23]. Figure 2.6 is a graphical representation of the relationship between the loss of the network, $f(x)$, and the value of a single network parameter, x.

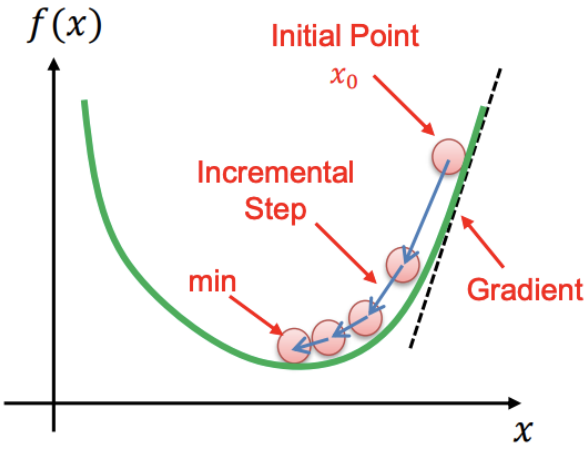


Figure 2.6: The gradient descent process [23]

2.4 Recurrent Neural Networks and Long Short Term Memory

Recurrent neural networks are a type of neural network containing feedback connections which allow previous outputs to influence future decisions [23]. As opposed to vanilla neural networks, which take fixed size vectors as inputs, recurrent neural networks are able to operate on sequences of vectors [24]. This is important in the modelling of sequential data as they learn based on the temporal dependencies of the input, in order to produce an output. There exist many problems which require the capacity to operate on temporal influence, some of these include: natural language processing, stock price prediction and music generation. Recurrent neural networks can be visualised unwrapped as in Figure 2.7, where h_0 to h_t represent the outputs of the network at each time-step in the input sequence.

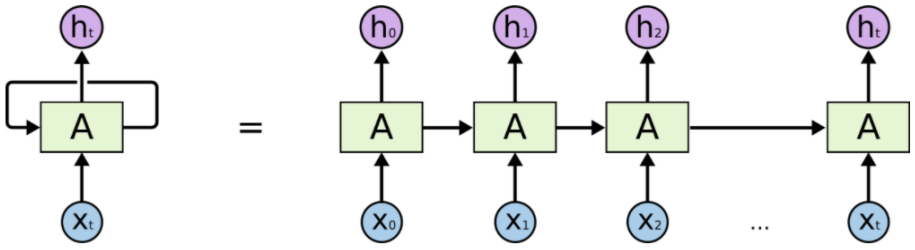


Figure 2.7: Unravelled neural network [25]

Whilst initially recurrent neural networks appear to solve the issue of learning from sequential data, learning longer-term dependencies proves to be very difficult [26]. This is accredited the vanishing

gradient problem. The vanishing gradient problem occurs when many small gradients, calculated for gradient descent, are multiplied together, resulting in very small gradients. This causes the network weights and biases to stop updating effectively [27]. In 1997 a solution to this problem was proposed called "long short term memory" [28]. The key difference between a standard recurrent neural network and LSTM is the presence of a cell state [25]; the cell state allows the network to retain information from several time-steps back.

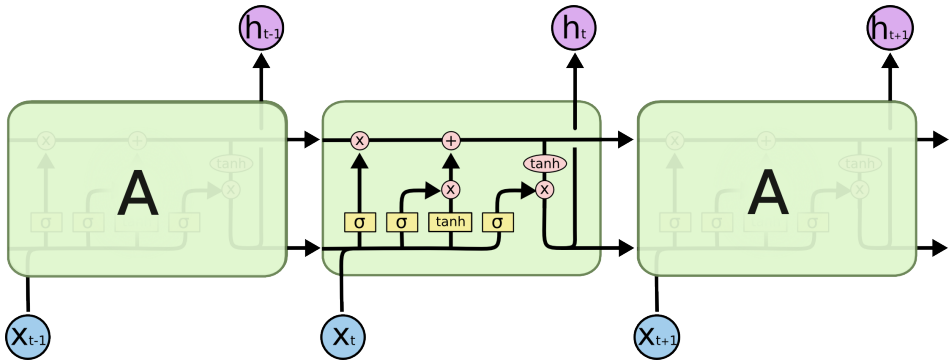


Figure 2.8: Unravelled LSTM network [25]

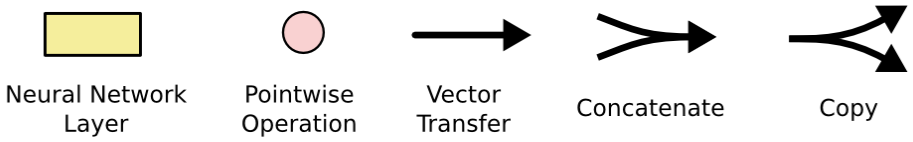


Figure 2.9: LSTM cell component key [25]

An LSTM cell can be deconstructed into a number of layers with unique functions. Each function has an effect on either the cell's state or the cell's output or both.

Forget Layer

The first layer of the network is the forget layer, which decides the information that should be forgotten from the cell state of the previous time-step. This is decided based on the vector concatenation of the new input and the hidden state from the previous time-step [29]. The result of this is then multiplied by the cell state from the previous LSTM cell instance. Figure 2.10 highlights the location of the forget gate within an LSTM cell. Figure 2.11 shows the equation used to compute its output; σ represents the sigmoid activation function, W_f represents the trained weight relating to the forget gate and b_f is the relative bias.

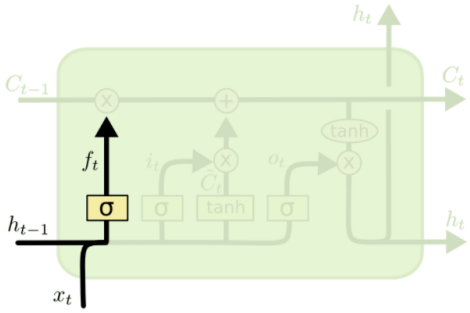


Figure 2.10: LSTM forget layer [25]

$$f_t = \sigma(W_f \cdot [h_{t-1}, x_t]) + b_f$$

Figure 2.11: Forget layer equation [25]

Input Layer

The second layer, the input layer, is made up of two parts: a sigmoid layer deciding which values need to be updated, and a hyperbolic tangent layer for creating new candidate values which could be added to the cell state [25]. The inputs to both of these functions are calculated from the vector concatenation of the new input and the hidden state from the previous output. These outputs are then multiplied and added to the cell state as an update. Figure 2.12 highlights the input layer within the LSTM cell. Figures 2.13, 2.14 and 2.16 show the calculations performed at each part of layer.

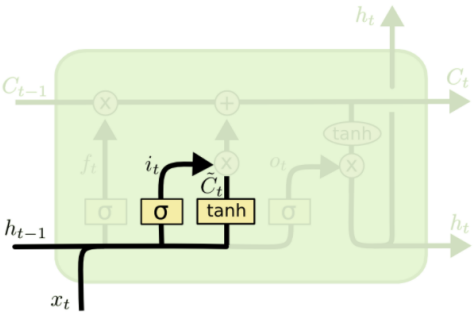


Figure 2.12: LSTM input layer [25]

$$i_t = \sigma(W_i \cdot [h_{t-1}, x_t]) + b_i$$

Figure 2.13: Input layer update calculation [25]

$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t]) + b_C$$

Figure 2.14: Input layer candidate value calculation [25]

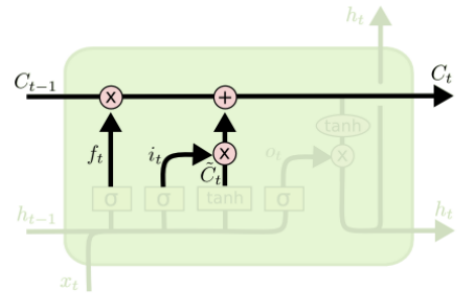


Figure 2.15: LSTM cell state adjustments [25]

$$C_t = f_t \cdot C_{t-1} + i_t \cdot \tilde{C}_t$$

Figure 2.16: New cell state calculations [25]

Output Layer

The final layer is the output layer, which decides the value of the "hidden state" of the cell. This is calculated as the product of a sigmoid-activated vector concatenation of the new input and previous hidden state, and the hyperbolic tangent-activated cell state [29]. This output is then used as an input for the next time-step, representing the h_{t-1} input value in the following iteration.

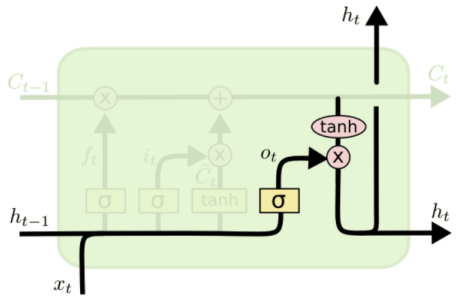


Figure 2.17: LSTM output layer [25]

$$o_t = \sigma(W_o \cdot [h_{t-1}, x_t]) + b_o$$

Figure 2.18: Calculation on the input and previous hidden state to form part of the output[25]

$$h_t = o_t \cdot \tanh(C_t)$$

Figure 2.19: Output (hidden state) equation [25]

LSTMs are trained using "Back Propagation Through Time", which applies the regular backpropagation algorithm to each time-step within the network as if it were unravelled and each time-step represented a single layer within the network [23]. Figures 2.10, 2.12 and 2.17 show the different layers in an LSTM cell.

LSTM neural networks were decided to be the most suitable for this project due to the widespread use of them in the field of algorithmic music generation. LSTMs are specifically designed to operate on sequential data with long-term dependencies [25]. Music can be modelled as a sequence of notes played over time-steps, with some long-term repeating motifs, themes or ideas.

2.5 Keras

Keras is a high-level API used for building a variety of neural networks [30]. Keras contains implementations of LSTM neural network layers, which are the main neural-network-related focus for this project. Keras was used in the majority of existing projects discussed in section 2.2.

2.6 Music21

Another key area of background research was in music manipulation in Python; one of the key challenges of this project is converting sheet music into something a neural network can interpret. Music21 is a toolkit for handling music within Python, containing functionality for parsing music scores, extracting notes and other musical features, manipulating notes and exporting scores into popular music file formats [31]. This library proved to be invaluable in this project and hugely expedites pre and post-processing within the system.

Project Management

For a project of this scale, spanning several months and consisting of multiple components, adequate project management is paramount. The management of the project, for the large part, adhered to the guidelines of the specification [8], with some changes along the way.

3.1 Time Management

The exact scope of the project was not defined at the start of the project due to the variety of unknowns in the complexity and performance of certain approaches for generating sight-reading music. For this reason, the concept of an iterative development cycle was concluded to be the best strategy for tackling the problem. It was concluded that a variant of the agile development methodology would be the most appropriate for this project. This is due to the short development cycles and the ability to handle unstable, changing requirements [32]. Having an iterative development cycle meant that a minimum viable product could be designed before implementing any additions, such as the inclusion of other musical features. A sprint Gantt chart was devised to outline the key stages in each sprint cycle. Figure 3.1 shows the planned sprint timetable; Figure 3.2 shows a more realistic representation of the sprints from a post-project perspective. The key difference between the planned sprint timetable and the resultant sprint timetable is the allocation of time for hyperparameter tuning. In the project initialisation stage, the time taken for hyperparameter tuning was not accounted for; however, during the development cycles, it became apparent that a large portion of the sprint needed to be reserved for optimising the network. Another aspect of the sprint cycle that changed was the integration of the testing stage into the development stage; it became apparent that unit testing each piece of code after it was written, was more intuitive than testing all of the code at the end of the development stage.

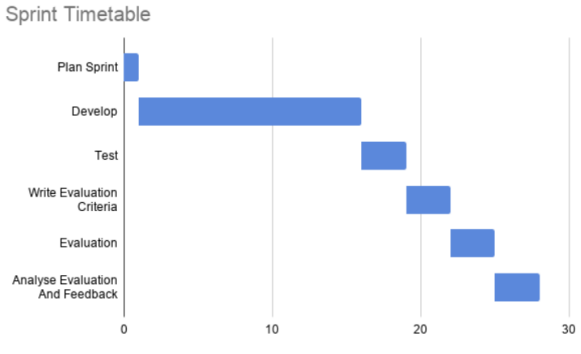


Figure 3.1: Initial 4-week sprint cycle Gantt chart designed for use in the project specification [8].

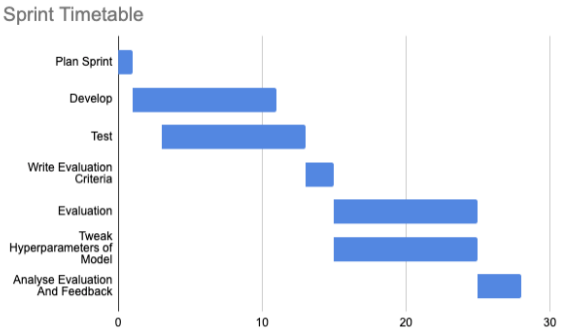


Figure 3.2: Adapted sprint cycle Gantt chart.

Each stage of the sprint timetable can be summarised as follows:

1. **Plan Sprint:** Considering the progress made from the previous sprint, provide a feasible scope for the progress that could be made in the upcoming sprint.
2. **Develop:** Implement the code required to fulfil the requirements set in the planning stage.
3. **Test:** Test code in small parts to ensure it functions as required. If required, and possible, perform exhaustive checks on all possible inputs and outputs of functions.
4. **Write Evaluation Criteria:** Improve or alter previous evaluation criteria if necessary.
5. **Evaluation:** Perform evaluation on models and system as a whole. The final stage of the evaluation is human evaluation of the music, which is done once the optimal classification model has been found. This is described further in the "Evaluation" section of this document (section 5).
6. **Tweak Hyperparameters of Model:** Change the hyperparameters of the model and re-evaluate the model(s). Perform iteratively until satisfactory/required results have been achieved.
7. **Analyse Evaluation and Feedback:** This is mainly in regard to the human evaluation. Are there any violations in musical theory, or a specific aspect of music that the model does not perform well with? These results are passed to the initial stage of the following sprint to perform alterations on the system as a whole.

Alongside the sprint timetable, a full timetable of the project was created; this is displayed in the appendix of this document (Figure 1). The full timetable outlined three sprints over the duration of the project, with a preliminary stage for background research. In reality, the project consisted of four sprints, since the results from the initial sprint were not considered sufficient due to a flaw in the design of the system - this is discussed further in section 4.3.3. The achievements of each sprint are summarised below:

Sprint 1: The framework for pre and post-processing of music, utilising functionality from the Music21 Python library [31].

Sprint 2: Initial progress in music generation.

Sprint 3: Experimentation with data augmentation resulting in better quality music, without significant overfitting of the network.

Sprint 4: Experimentation with the addition of other musical features, improving on the results of the previous sprint.

Sprints were managed using Trello boards; a Trello board was used per sprint with additional boards for some larger tasks within each sprint. The Trello board was kept relatively simple and consisted of 4 columns:

Backlog: Contains the tasks required to be completed in the sprint. This was not used exhaustively due to its similarity to the "To Do" column.

To Do: Contains the tasks that are about to be undertaken.

Doing: Contains the tasks currently being worked on. The only instance of multiple tasks existing in the "Doing" column is when a task cannot be completed from start to finish at once. An example of this would be waiting for a model to train.

Done: Contains completed tasks.

An example of this is shown in Figure 3.3

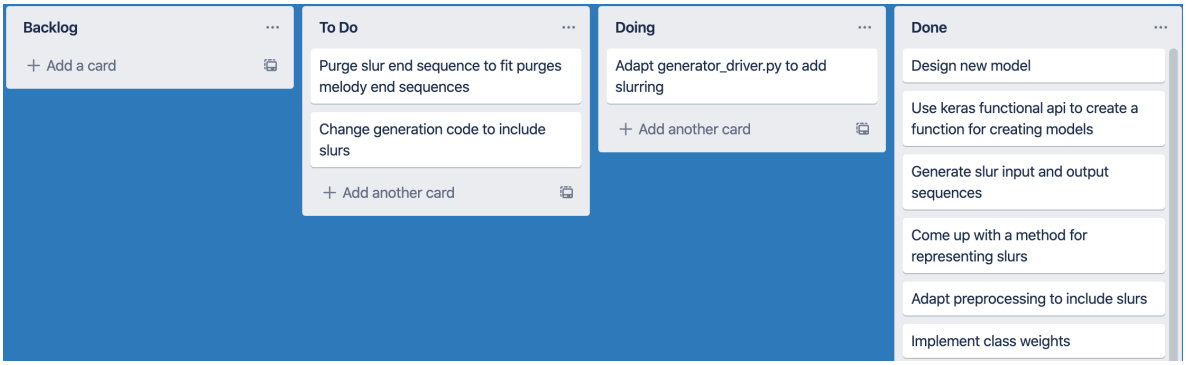
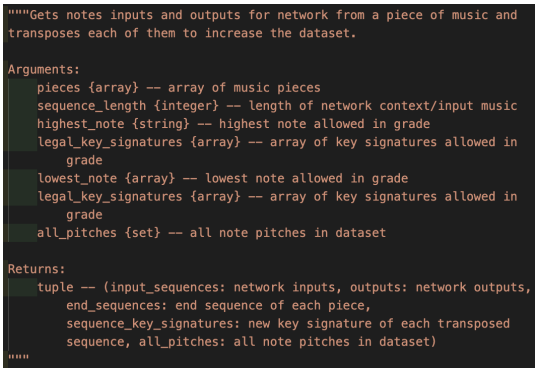


Figure 3.3: Screenshot of Trello board used for adding slurs in the final sprint of the project.

3.2 Code Management

Version control was performed using GitHub. A master branch was used to maintain the current fully-tested working solution, with subsequent branches for the items currently under development. GitHub enabled the fast transfer of code between the development machine and the servers used to train models. Code is separated into files, each containing a different portion of functionality. In order to maintain code readability, the PEP 8 Python style guidelines have been adhered to [33], and each function is adequately described with document strings; this includes a description and type definition

for each parameter as well as the function's return value. Figure 3.4 shows an example of a document string used to describe a function.

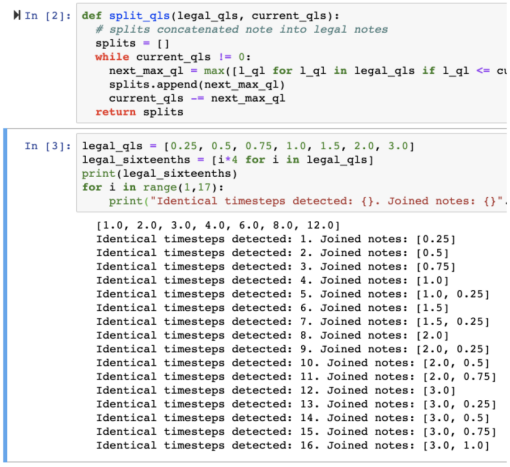


```
"""Gets notes inputs and outputs for network from a piece of music and
transposes each of them to increase the dataset.

Arguments:
    pieces {array} -- array of music pieces
    sequence_length {integer} -- length of network context/input music
    highest_note {string} -- highest note allowed in grade
    legal_key_signatures {array} -- array of key signatures allowed in
        grade
    lowest_note {array} -- lowest note allowed in grade
    legal_key_signatures {array} -- array of key signatures allowed in
        grade
    all_pitches {set} -- all note pitches in dataset

Returns:
    tuple -- (input_sequences: network inputs, outputs: network outputs,
        end_sequences: end sequence of each piece,
        sequence_key_signatures: new key signature of each transposed
        sequence, all_pitches: all note pitches in dataset)
"""
```

Figure 3.4: A screenshot of an example docstring used to describe a function.



```
In [2]: def split_qls(legal_qls, current_qls):
            # splits concatenated note into legal notes
            splits = []
            while current_qls != 0:
                next_max_ql = max([l_ql for l_ql in legal_qls if l_ql <= cu
                splits.append(next_max_ql)
                current_qls -= next_max_ql
            return splits

In [3]: legal_qls = [0.25, 0.5, 0.75, 1.0, 1.5, 2.0, 3.0]
        legal_sixteenths = [i*4 for i in legal_qls]
        print(legal_sixteenths)
        for i in range(1,17):
            print("Identical timesteps detected: {}. Joined notes: {}".

        [1.0, 2.0, 3.0, 4.0, 6.0, 8.0, 12.0]
        Identical timesteps detected: 1. Joined notes: [0.25]
        Identical timesteps detected: 2. Joined notes: [0.5]
        Identical timesteps detected: 3. Joined notes: [0.75]
        Identical timesteps detected: 4. Joined notes: [1.0]
        Identical timesteps detected: 5. Joined notes: [1.0, 0.25]
        Identical timesteps detected: 6. Joined notes: [1.5]
        Identical timesteps detected: 7. Joined notes: [1.5, 0.25]
        Identical timesteps detected: 8. Joined notes: [2.0]
        Identical timesteps detected: 9. Joined notes: [2.0, 0.25]
        Identical timesteps detected: 10. Joined notes: [2.0, 0.5]
        Identical timesteps detected: 11. Joined notes: [2.0, 0.75]
        Identical timesteps detected: 12. Joined notes: [3.0]
        Identical timesteps detected: 13. Joined notes: [3.0, 0.25]
        Identical timesteps detected: 14. Joined notes: [3.0, 0.5]
        Identical timesteps detected: 15. Joined notes: [3.0, 0.75]
        Identical timesteps detected: 16. Joined notes: [3.0, 1.0]
```

Figure 3.5: A screenshot of a Jupyter Notebook used to unit test a function by checking all input and output combinations exhaustively.

Unit testing was mainly performed in Jupyter Notebook [34], where small extracts of code were tested for consistency and correctness. An example of this is shown in Figure 3.5, where a function "split_qls" is exhaustively tested with all input and output combinations.

3.3 Legal, Social, Ethical and Professional Issues

The only ethics-related issue brought up in this project relates to the human evaluation performed. In this method, evaluators were asked to give feedback, in the form of a survey, on the system's performance; this is further described in section 5.2. The "Ethical Consent for Undergraduate Projects" resource states that a project "recruit[ing] a small number of participants within the students' social circle, family, peers, or colleagues do[es] not require formal consent" [35]; since the human evaluation never extended beyond the author's social circle, no formal consent was required.

Design

4.1 System and Project Requirements

Section 1.2, entitled "Project Scope", lays out the general scope of the project; this section seeks to formalise the specific requirements set out at the start of the project to produce a functioning system. These requirements have been separated into functional and non-functional requirements, and are each labelled as a "Must", "Should", "Could" or "Won't" in accordance with the MoSCoW requirements guide [36, 37].

4.1.1 Functional Requirements

M The system must preprocess MusicXML files into the format required by the neural network.

M The system must be able to take into account note pitches and note lengths.

M The system must be able to output music in a sheet music format.

M The system must be able to receive a JSON file containing quantitative requirements set out by an examination board's graded sight-reading specification.

M The system must produce music that adheres to the quantitative requirements given.

M The system must be tested on an instance of an instrument and a grade.

S The system should take into account musical features besides just notes.

C The system could have a feature capable of scanning music to add to the dataset.

C The system could have a user interface for displaying generated pieces.

C The system could include a feature for classifying pieces as a specific instrument grade.

W The system won't be tested on all instrument and grade combinations.

W The system won't produce polyphonic or multi-stave music.

4.1.2 Non-functional Requirements

M The system must produce coherent melodies, suitable for a piece of sight-reading music.

S The network should be able to produce music in a timely manner (faster than a human composer).

4.2 Sliding Window Generation

As mentioned previously, a many-to-one recurrent neural network featured in the majority of systems and projects researched. This may seem counterintuitive in regard to generating music because the higher-level goal is to produce sequences of music, without having to supply any unique input (music written by a human composer). However, multiple calls to the network with almost identical inputs can produce sequences of unique notes. This is done using a method referred to in this document as "sliding window generation". The theory behind this method is that in order to produce unique music using something inherently uncreative, an artificial neural network, there needs to be some initialisation of a state. This comes in the form of an input sequence of notes representing the preceding section of music to the single note that the network will produce. When generating a piece of music, the network is provided with a "context" of the music produced previously and is given the task of producing the next note. This can be iteratively called to produce large sequences of notes, by appending the output of the network to the input of the following call to the network. This enables the system to produce sequences of music, one note at a time. The "context" provided to the network remains the same size and shifts along the theoretical stave in discrete time-steps. This is better illustrated in Figure 4.1 and described algorithmically in Figure 4.2.

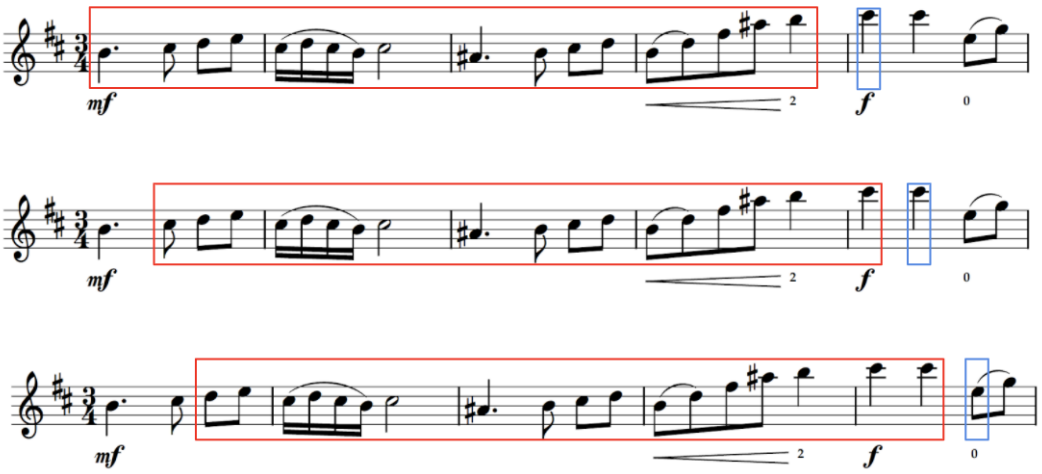


Figure 4.1: Sliding window illustration

Algorithm 1: Sliding window generation

```
Data: seed, length
// seed (array):  initial context for the network
// length (int):  length of desired piece
Result: piece
// piece (array):  generated piece of music
piece ← [] ;                                        // initialise empty array
for i ← 0 to length do
    // Each iteration produces one note
    note ← predict(seed) ;          // predict next note based on input seed
    piece.append(note);                    // append predicted note to piece
    seed ← seed[1:];                 // remove the first value from the seed
    seed.append(note); // append the predicted note to the end of the seed
end
```

Figure 4.2: Sliding window generation algorithm

In order to implement the sliding window technique, the original dataset needs to be preprocessed in such a way that the network can be provided with fixed-length contexts of music along with corresponding single note outputs. The network must then be trained on each of these pairs of inputs and outputs until it is able to generalise over the patterns in the music. During the generation stage, as mentioned previously, the network must be provided with an initial context to stimulate generation. This input, whilst theoretically representing the preceding notes, can be thought of as an initialisation of state, where the network is given an idea of the style of the music it is required to generate. These indications could include the key of the piece, the tonality of the piece and the types of rhythms that appear within the piece. Once the network has been initialised with the seed context, the notes can be generated iteratively to produce a new piece of music. Iterative note generation is performed until

a piece of the required length is produced; the original input seed is not retained in the final piece of music, only the notes generated by the network.

4.3 System Design

4.3.1 Seeding Methods for Generating New Music

In order to generate a new piece of music, the sliding window generation method requires an initial seed of music. Two sustainable seeding methods were conceptualised. The first of these methods involved retaining the last window of context from each piece of music within the dataset. These contexts are not used in the training of the network because they have no corresponding output since they're at the end of the piece. Generating a piece in this manner has the effect of continuing the piece of music past its composed ending. This method was seen as sustainable because once one end seed has been used to generate a piece of music, the end seed of the generated piece could replace the one used to generate it. The other method proposed was to artificially generate seeds capable of generating sensible music. This involves:

1. Finding all unique whole-bar rhythms that occur within the dataset.
2. Counting their occurrences within the dataset.
3. Choosing one from a frequency-weighted distribution.
4. Selecting a key from the specification.
5. Randomly selecting notes that fit within that key and within the range allowed in the specification.
6. Applying the notes to the chosen rhythm.

Both of these methods were tested on each model, the results of which are discussed in the section 7.

4.3.2 Ensuring Generated Music Fits The Specification

One of the key challenges of this project is producing sight-reading music that is ensured to fit the specification set by the music examination board. In order to better restrict the music generated by the system, a set of quantitative requirements was devised based on those found in the specifications. These quantitative requirements are aspects of the music that can be strictly imposed, and would not have to be learned by the network. These requirements include, but are not limited to, the allowed pitch range, the allowed key signatures and the length the pieces of music should be. The implementation of these quantitative restrictions is addressed in section 6.11.1. Along with adhering to the quantitative requirements set out by the examination board, the generated music must also adhere to some unquantifiable qualities. These include, but are not limited to: the sounds and flow of the rhythms and melodies, the changes in key and tonality throughout the pieces and the relative positioning of articulations and other musical features; these qualities need to be learned by the

network. Some aspects of the music fall in both categories, for example, the use of semi-quaver and dotted quaver patterns. The ABRSM violin sight-reading specification states that dotted quavers can feature in sight-reading music for grades 3 onwards [3], however, it also states that they should occur in "simple patterns" like the pattern shown in Figure 4.3. The quantifiable aspect of this requirement is that the network is allowed to produce dotted-quavers; the restriction that a dotted-quaver must be succeeded by a semi-quaver is a rhythmic dependency that the network must learn.

Figure 4.3: Dotted quaver followed by a semi-quaver.

4.3.3 Accounting For Note Lengths

Designing the system to consider pitches when generating music was quite straightforward. The pitches are the main aspect the network is making decisions on, and all projects researched take note pitches into account. The other aspect that needs to be taken into account, highlighted as a "Must" in the requirements section (4.1), is the length of notes. The relative note lengths of a piece form its rhythms; the complexity of these rhythms can be the difference between a piece that is very easy to play and a piece that is very difficult to play. Three main methods for producing note lengths were considered.

Encoding Note Objects as Pitches and Note Lengths

This involved having each possible network input and output describing both a pitch and a length, for example, an "A5 quaver". Although this method may seem sensible, it increases the number of classes significantly since the number of possible notes is the total number of combinations of note pitches and note lengths. This creates very sparse data, where each note only appears a few times and some note combinations may not even be represented. This was the method used in the first sprint of development before it was quickly realised that it would be unsuitable.

Additional Network Input for Note Lengths

This method requires the network to have two inputs and two outputs, one signifying the pitch of the note and one signifying the length of the note. This could be implemented in two ways, a secondary binary input indicating the time that a note finishes being played or a secondary multi-class input indicating the length of the relevant note. The first of these methods requires splitting the notes into equal time steps within the network - this is similar to Weel [15]. In post-processing, these time-steps would have to be joined up based on the note end indicator output of the network - this is similar to Kotecha et al. [16]. Figure 4.6 demonstrates this concept. This method could work in theory; however, issues could arise where a different pitch is selected by the network, but the network regards the note as sustained.

The other method relies on the note length input indicating the length of the note. This method seems simpler; however, it adds complexity to the network as it would have to choose both a pitch and

a note length. Additionally, the length of the context given to the network is not of fixed size, since each time-step can represent a different amount of time. This is problematic since each note would be evaluated by the network equally, regardless of how long it is played for. Musically speaking this is not how a listener or performer would view each note, as sustained notes would dominate the sound of melody more than shorter notes.

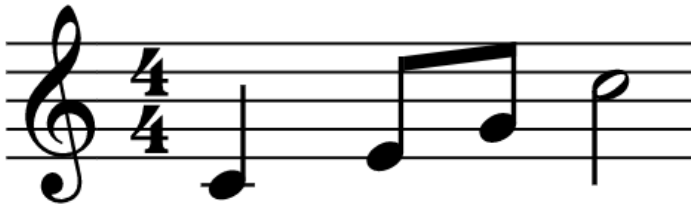


Figure 4.4: C major arpeggio, reference for Figures 4.5 and 4.6

pitches: [C3, E3, G3, C4]
note lengths: [1, 0.5, 0.5, 2]

Figure 4.5: Notes from Figure 4.4 represented as pitches and note lengths (measured in quarter lengths/number of crotches).

pitches: [C3, C3, E3, G3, C4, C4, C4, C4]
start new note: [1, 0, 1, 1, 1, 0, 0, 0]

Figure 4.6: Notes from Figure 4.4 represented as pitches over quaver time-steps (or half-crotchets/0.5 quarter lengths) and note starts (1 represents start new note, 0 represents sustain previous note).

Splitting Notes Into Time-steps

The final (and simplest) method considered requires the notes to be split into discrete time-steps, such that all note lengths can be reconstructed using an integer number of time-steps. This is the same as the method described in Figure 4.6, but without the "start new note" binary input. In this method, the only input/output required by the network is the note pitch, as there is no indication of where the note starts and ends; the boundaries between separate notes in a piece are regarded as the positions where the pitch changes. This relies on the assumption that there are no instances of adjacent notes of the same pitch since adjacent identically-pitched notes would be (wrongfully) concatenated. The issue of wrongful concatenation is mitigated with the following measures:

- Notes are only concatenated into note lengths allowed by the specification. This removes instances of the wrongful concatenation of notes resulting in a violation of the sight-reading specification.
- Identically-pitched notes across bar lines are separated.
- The addition of other musical features, such as slurs, decreases the likelihood of two adjacent notes being regarded as identical. This relies on the premise that the more features that are taken into account, the more likely there is to be some distinguishing factor between two seemingly-identical notes. An example of this would be a tied note, where two notes of the same pitch are

joined together with a curved line indicating they are to be played as one. The consideration of ties could enable the system to see them, rightly, as separate notes. This is because one note would be marked as the start note of a tie, rendering them non-identical.

This method was chosen as it was simple, added no extra complexity to the network, and retained the same length of time for each musical context. Although the method of dividing notes into discrete time-steps seems to be the most sensible solution, irrational note lengths cannot be represented in this manner. The only irrational note lengths in the ABRSM stringed instrument specification [3] are triplet notes, which can feature in the grades beyond grade 6. Whilst they don't necessarily feature in every piece of sight-reading music beyond this level, the lack of support for them within this system is certainly a limitation.

4.3.4 Technologies Used

The "Background" section (2) of this document highlights the areas of interest in the major technical aspects of the project. These include the implementations of neural networks and the processing of music. The system is coded entirely in Python [38]; Python is the most widely accepted programming language for modern neural computing. Because of this, Python has a plethora of libraries designed to aid in data manipulation and the building of neural networks. The data manipulation packages used in this project include: pandas [39], sklearn [40], and NumPy [41]. The neural networks for this project were implemented using the Keras [30] Python library running on top of TensorFlow [42]. The concern was raised in the progress report that the high-level nature of Keras could cause hindrance and inflexibility as the network became more complex [9]; however, this did not materialise. The music processing for the system was done using the Music21 Python library [31]. This included, but was not limited to: the importing of music files, the extraction of musical features, and the generation of scores from the music output by the network.

4.4 Network Design

A key benefit of the sliding window method is the simplicity of the neural network. Due to the single output nature of the generation, the network used could be a single-label multi-class classification model. This means that the neural network is given an input sequence, in the form of a musical context, and is tasked with predicting the single next note that should succeed the given sequence. The single-label classification definition of the network dictated several structural and design decisions.

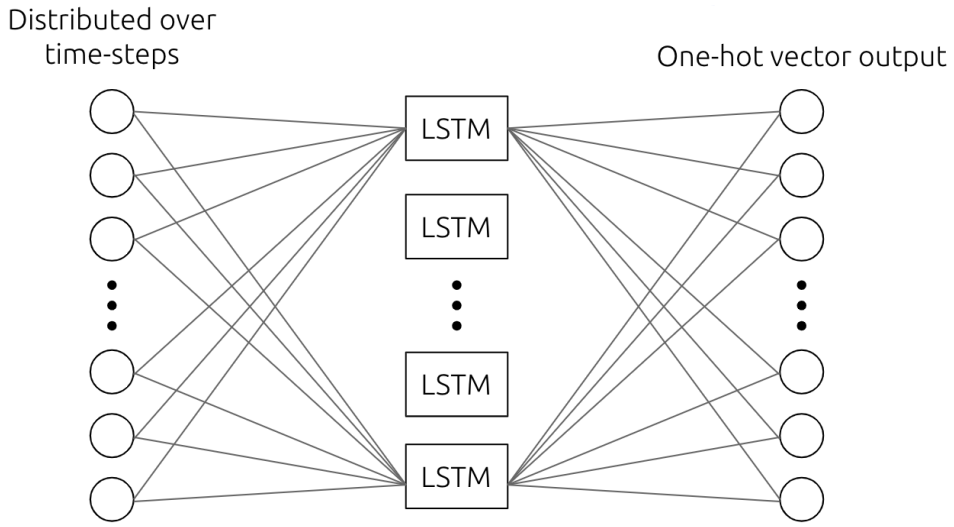


Figure 4.7: Network structure

The structure of the inputs is a time-distributed sequence of one-hot encoded vectors; each vector represents a single note being played and is indicated by a 1 at the index of the encoded note and 0s at the indices of all the other encoded notes. This requires the notes to be encoded, which is done by enumerating over all possible notes and assigning integer values to each note within the range of 0 and the total number of possible notes. Other input methods were considered, namely ordinal encoding and learned embedding [43]. Ordinal encoding assumes there is an ordering of inputs and encodes each possible input as a single value within a range. This may seem suitable initially, since notes can be ordered by their pitch. However, in written sheet music, more than one note can represent the same pitch (enharmonic equivalence), which means they would be encoded the same. It was initially considered that rather than encoding the name of the note, the absolute pitch of the note could be encoded, however, this would cause complications since the original representation of the note pitch would be required in the post-processing to display it on a stave. Learned embedding is another method of categorical encoding where the dimensions of the input vector are reduced through the network learning some ordinal dependencies during training. Learned embedding is particularly useful when operating on textual data since the number of unique words can be in the tens of thousands with many words being similar [44]. The result is a reduced encoded vector usually in the order of tens or hundreds [43]. Since the number of unique pitches is already in the order of tens, the learned embeddings method was deemed unnecessary for this project.

The hidden layers of the network, as discussed previously, are LSTM units; they are responsible for learning the relationship between each time-step. The number of units and the number of hidden layers is a hyperparameter which needs to be tested and altered (tuned), in order to find an optimal layout. When the LSTM layers are stacked, it is important to return the hidden state of the LSTM unit after each time-step, so that it can be input into the next LSTM layer. Adding additional layers allows the network to learn more features about the music [45]. The only hidden state required from the final layer LSTM is the last time-step, since no further processing of the whole sequence is done. The final layer of the network is the output layer, which is a dense layer representing a one-hot encoded

vector identical to those found in the network input. The activation function of the output layer is the softmax function, defined by the equation in Figure 4.8. The softmax function activates each node in the layer with a number between 0 and 1, representing the probability that the class is chosen [46]. The softmax function ensures all nodes within the layer sum to 1 with the most activated node representing the chosen class - in this context, the note that the network has chosen.

$$\sigma(z_i) = \frac{\exp(z_{id})}{\sum_{j=1}^{d} \exp(z_{ij})} \forall d$$

Figure 4.8: The softmax function

The loss function of the network was chosen to be categorical cross-entropy loss, also known as categorical log loss; this is due to its extensive use in classification problems. The loss function is used to optimise the neural network weights and biases by measuring the closeness between the predicted value and the target value from the dataset during training [23]. Categorical cross-entropy loss is a variation of entropy loss, defined mathematically in Figure 4.9. As mentioned previously, the loss function is used to optimise the network by partially differentiating with respect to the network's parameters (weights and biases). This gives the direction the value of the parameter needs to move, increase or decrease, in order to minimise the loss function. The size of the step made to minimise the loss function is defined by the network's optimiser; the type of optimiser chosen for this network is an adaptive learning rate optimiser. This is because, in a fixed step-size optimiser such as stochastic gradient descent, the step size (or learning rate) is a hyperparameter which requires tuning. Using an adaptive learning rate optimiser reduces the necessity to tune the learning rate through exhaustive and time-consuming parameter searches. There are many examples of adaptive learning rate optimisers, each with their own benefits and drawbacks. It was decided that RMSprop and Adam would be two potential candidates for the best optimiser for this network based on their popularity in the field of music generation using neural networks. RMSprop normalises the gradient of each network parameter using an exponential weighted average of the squared gradients accumulated from previous steps and is used in the video game music [14] and Led Zeppelin replication [15] projects evaluated in section 2.2. Adam is a more recently developed optimiser which performs normalisation like RMSprop but also utilises the exponential weighted average of the gradients from previous steps, acting as momentum. Adam is the optimiser used in the polyphonic guitar music generation project evaluated in section 2.2 [17]. Both optimisers were tested for their performance during the hyperparameter tuning stages of development.

$$L(y, \hat{y}) = -\frac{1}{n} \sum_{i=1}^{n} \sum_{j=1}^{m} (y_{ij} \cdot log(\hat{y}_{ij}))$$

Figure 4.9: Categorical cross-entropy loss equation

The network is trained using mini-batch gradient descent, where subsets of the dataset are passed through the network before the network's parameters are updated. Traditional gradient descent per-

forms updates on the network's parameters only once the entire dataset has been traversed, and whilst this produces smoother updates to the network, it is very inefficient. Stochastic gradient descent is an alternative to this, where only one training example is used before parameter adjustments are made. Stochastic gradient descent achieves a significant speed boost, however, network updates are often very noisy; mini-batch gradient descent is a middle ground between these two methods, achieving benefits in both performance and efficiency [23]. Once gradient descent has been performed based on all mini-batches of data in the dataset, the data is shuffled and split into different mini-batches to repeat the process. Each of these traversals through the entire dataset is referred to as an "epoch"; the number of epochs the network is trained for depends on the performance increase provided by each epoch. Once the network's performance stops increasing, the network has finished training.

Additional design features of the network include the use of dropout layers, which are a regularisation feature used to prevent overfitting. Overfitting occurs when a network learns the training data too precisely and fails to generalise - this results in poor performance when tested on unknown data. Overfitting can be detected through the use of validation data, which is essentially test data used during training; when the network stops improving when tested on the validation data, the network will have reached its optimal state. Dropout layers set a specific fraction of the outputs from the previous layer to zero during training; the fraction chosen is another hyperparameter that requires tuning. Dropout layers are present after each of the LSTM layers to prevent overfitting during training; additionally, recurrent dropout was added to the LSTM units to achieve the same effect between each time-step.

Evaluation

The evaluation of this system comes in two parts: evaluating the performance of the networks as a single-label multi-class classifier, and evaluating the quality of the music produced by the system as a whole.

5.1 Evaluating The Network

Evaluating the network involves abstracting it from the context of music generation and looking at it purely as a class predictor. The optimal performance of the network occurs when it is able to predict the next note in a sequence with 100% accuracy. This is not strictly feasible, due to the imperfectness of neural networks and the degree of subjectivity of music. Fortunately, unlike complex and pleasant sounding music, sight-reading music tends to lack creativity, suggesting that it should be easier to predict.

During the training of the network, the loss function is minimised; this represents the decrease in the network's inaccuracy. Alongside this, an accuracy function is used to represent the number of notes correctly predicted as a percentage of the entire dataset. Although these metrics seem very similar, there is a key difference. The loss function is calculated as a combination of all output nodes and represents the difference between the predicted output vector and the target vector. This includes comparing the predicted values, which are represented by $x \in \mathbb{R}$, $0 < x < 1$ and the target values, which are represented by $x \in \{0, 1\}$. This is useful for improving the network, since it regards the outputs of the network as continuous values and allows for gradual refinement. However, it does not give a definite metric of how the network will predict single notes. The chosen loss function can be seen in Figure 4.9. The accuracy function ignores all classes except for one, the chosen class. The accuracy is calculated by taking the maximum value from the predicted output vector; this relies on the outputs being on a probability distribution which is ensured by the use of the softmax function (see Figure 4.8). The accuracy is calculated as the number of instances where the predicted and target class are equal, divided by the total number of training samples.

In addition to the training loss and accuracy, at the end of each training epoch, the network is evaluated using validation data. The validation data is not used for any training purposes, only as a testing metric throughout the training to detect overfitting and indicate when the network stops improving. After the network has finished training, the training and validation loss and accuracies over the training period can be plotted as in Figures 5.1 and 5.2.

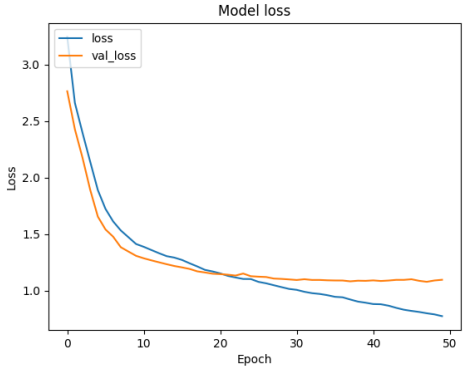


Figure 5.1: A graph showing the relationship between the losses of a model and the number of epochs it is trained for.

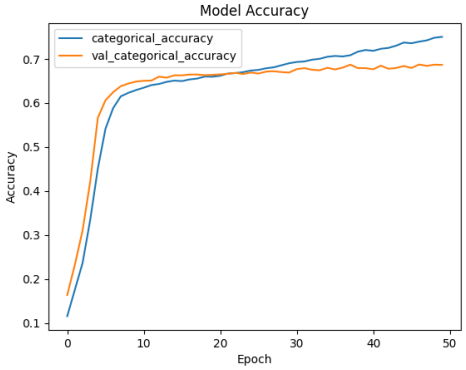


Figure 5.2: A graph showing the relationship between the accuracies of a model and the number of epochs it is trained for.

Once the network has been trained, it can be tested on test data retained from the original dataset. This provides an understanding of how the network will perform on new data since the test data has had no influence whatsoever on the training of the model. During the evaluation stage of the development sprints, multiple networks with different configurations were trained and compared using the results from the test data. This type of evaluation resulted in the discovery of the best hyperparameters for the network, through exhaustive searches of all chosen hyperparameter combinations. Figure 5.3 shows some potential hyperparameter choices and their corresponding values; training on all combinations of these parameters would result in 16 (2^4) different models. The values chosen for testing were based on those chosen by similar projects, and those that performed well in previous grid searches and sprints.

LSTM layers	LSTM units	Dropout	Optimiser
2	128	0.2	RMSprop
3	256	0.3	Adam

Figure 5.3: Example hyperparameter choices

Confusion matrices are another tool used to evaluate the performance of a classification network. All combinations of predicted values and target values are represented in a two-dimensional matrix; this shows the combinations of classes often confused with others [47]. In the context of music, it would be expected that some notes could be confused with other notes of a similar pitch or within the same key. Figure 5.4 shows an example of a confusion matrix where the number of classes is reduced by the removal of the notes' respective octaves. The dark line down the diagonal of the matrix represents correct predictions, and the lighter-in-colour squares represent small fractions of incorrectly classified data.

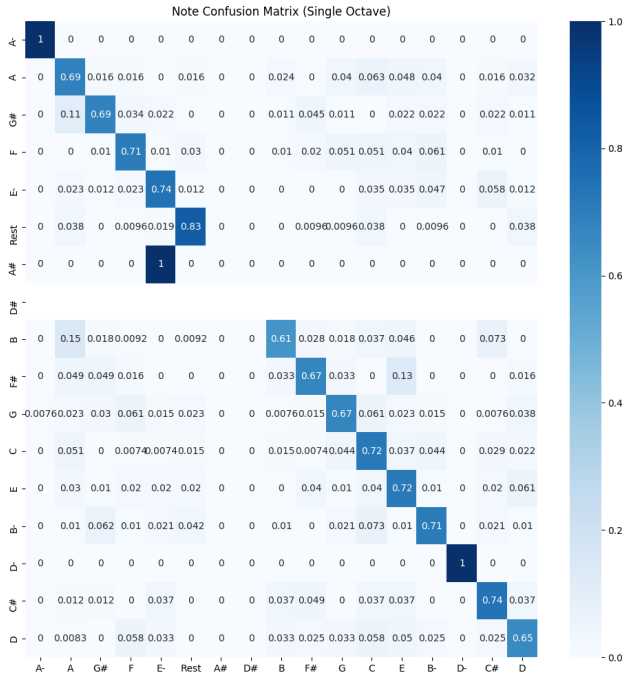


Figure 5.4: A confusion matrix of true notes (y-axis) against their respective predicted notes (x-axis), reduced to a single octave.

5.2 Evaluating The Quality of Music Generated

The other type of evaluation done was through a broader look at the ability of the system to produce appropriate sight-reading music. Analysing the quality of the music proved to be difficult due to the, previously mentioned, subjectivity of music. That being said, useful quantitative analysis could be done by comparing some features of the music produced by the network, to the original dataset. These features include the frequency of certain note pitches or note lengths over the entire dataset and the average number of out-of-key-signature notes per piece of music. The latter aided in the decision to assign a new key signature to a piece after it is generated, rather than use the same key signature of the input seed that was used to generate it. Figures 5.5 and 5.6 are examples of plots generated using this type of analysis. Figure 5.5 shows the distribution of the number of unique pitches per piece in datasets generated using different seed methods. Figure 5.6 shows the relative frequency of each note length in datasets generated using different seed methods.

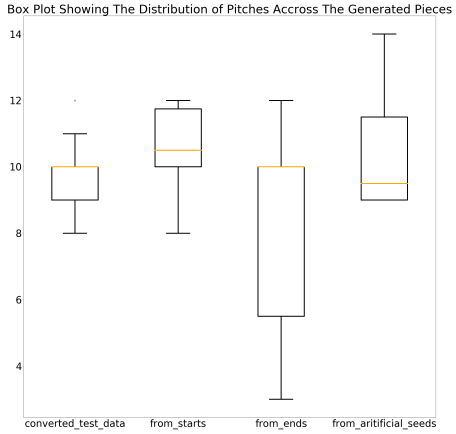


Figure 5.5: A box plot showing the distribution of the number of pitches per piece of music using different seeds.

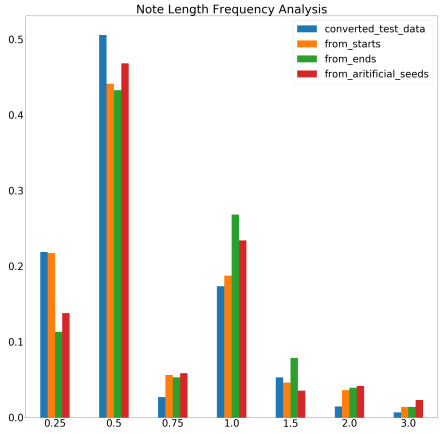


Figure 5.6: A graph showing the frequency of note lengths within datasets of pieces generated using different seeds.

Alongside the graphical musical analysis, some human evaluation was performed on generated pieces of music. This took two forms: a visual analysis performed by the author to look for violations in music theory and unusual rhythms, and opinion-based reviews obtained in a blind trial with musicians. The first of these proved to be quite a time-consuming task as it involved looking through several pieces of music per generation method, per model. This type of evaluation was limited to the models which performed best in other, more efficient aspects of the evaluation. The qualitative features mentioned in section 4.3.2 were a base for this evaluation, since the indication of rhythms mentioned in the specification suggest that the network has learned to produce appropriate rhythms. Conversely, the appearance of poorly grouped notes within bars and odd rhythms suggest the opposite.

The second form of human evaluation involved getting feedback from musicians about the quality of music produced, and its suitability for the given instrument and grade. This evaluation was performed at the end of each sprint in order to measure the progress made in the development process and to highlight areas that required improvement. Each candidate was given two pieces of music produced using each generation method from the best performing model in the respective sprint, along with two control pieces. The candidate was asked to play each piece of music at least once, and answer questions relating to them in a Google Forms document [48]. Figure 5.7 shows the questions asked to each candidate, and Figure 5.8 shows an example screenshot of the Google Forms document provided to the evaluators.

1. How good does the melody of this piece sound?
2. How well does this piece fit within the time signature?
3. How well does this piece fit within the key signature?
4. How difficult is the rhythm to play?
5. How difficult are the pitch intervals to play?
6. How difficult is this piece to play in general?
7. Is this piece of music suitable for a sight-reading piece at this grade?

Figure 5.7: Questions asked to evaluators in reference to pieces of music generated by the system. Questions 1 to 6 are answered on a scale of 1 to 5, question 7 is answered as either yes or no.

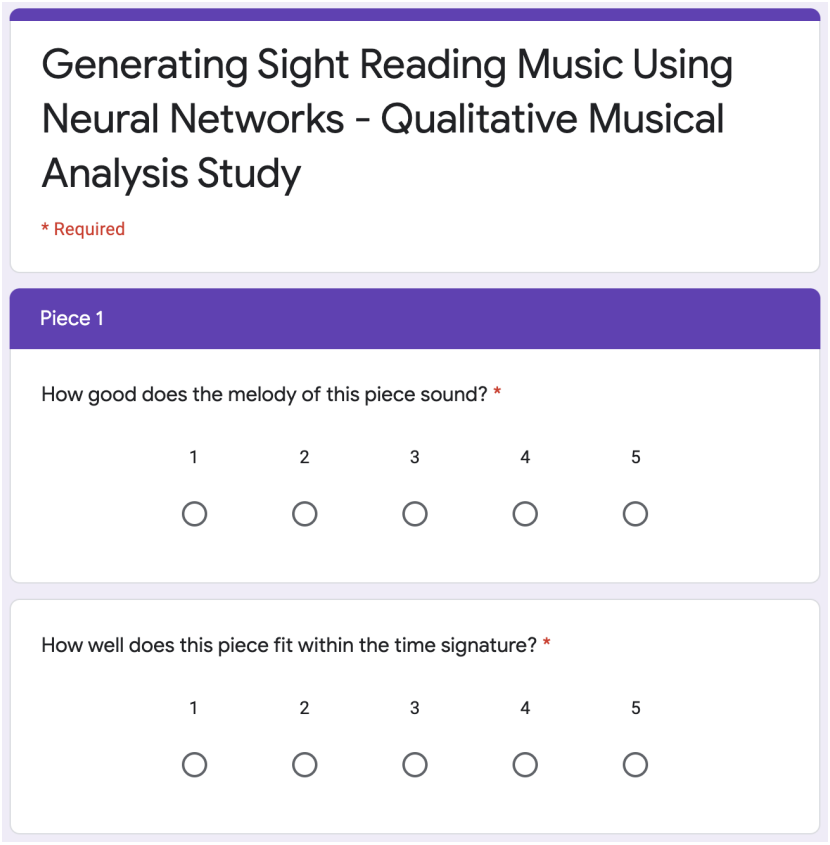


Figure 5.8: Screenshot of Google Forms questionnaire used in evaluating the quality of music produced by the system.

Methodology and Implementation

6.1 Data Collection

The data used to train the network came from graded-sight reading practice books. The data collection process involved searching for practice books suitable for ABRSM violin grade 5 sight-reading examinations, of which three suitable books were found [6, 49, 50]. The next stage involved scanning the pieces within each book using PhotoScore [10]. Although this software assisted the digital conversion of sight-reading pieces, it did not include some musical features such as slurs and also produced some errors in the positioning and pitches of notes. Sibelius was used to correct errors in the pieces and add any missing features [11], resulting in digital copies that were almost identical to their physical counterparts. A notable change made to each piece was the removal of any anacruses[1], which was done to ensure all bars within a single piece are of the same length. The file format chosen for the pieces was MusicXML [51], a file format specifically designed to store sheet music. This was chosen over the MIDI file format, due to MIDIs unsuitability at displaying sheet music caused by the distortion of some musical features and the lack of support for slurs. MIDI handles visually continuous changes to dynamics and tempo, such as crescendos (gradual increase in volume) and ritardandos (gradual decrease in speed), as discrete steps, which means they cannot be retrieved and displayed in their original format. MusicXML, however, stores these features as elements spanning over a period of time.

From the pieces obtained, a number of them contained features not allowed in the specification. Most notably and problematic were the instances of triplet notes in some pieces and the use of unusual time signatures. These pieces were removed from the dataset due to the negative impact they could have on the network's behaviour. The total number of sight-reading pieces obtained in the data collection process, after removing unsuitable pieces, was 48.

6.2 Preprocessing

The music data requires preprocessing such that it can be used to train the network. The initial stage of the preprocessing involves importing the music files into a Music21 stream and converting the stream into an array of note tuples which contain the two pieces of information required by the system: the pitch and the note length. The pitch of a note is represented as a string containing the note name, "A"

[1] An incomplete bar added to the start of a piece of music; another incomplete bar is usually added to the end of the piece whose length is the difference between the length of the first bar, and the bar length defined in the time signature.

for example, and the note octave, "3" for example. The length of the note is represented as a fraction of a crotchet (quarter note); for example, 0.25 represents a $\frac{1}{16}$ note. Once the pieces have been converted to array form, they need to be split into equal time-steps as discussed in section 4.3.3. The highest common factor of each note length is used as the time-step; this is because all note lengths can be reconstructed from an integer number of time-steps. From grade 3 stringed instrument sight-reading onwards, according to the specification [3], the smallest note allowed is a semi-quaver (sixteenth note). All other note lengths within the entire specification are divisible by semi-quavers, so it was deemed an appropriate time-step length. The shortest note before grade 3 is a quaver, so an eighth note could be used as the time-step for generating sight-reading music for grades 1 and 2. Although, even with sixteenth-note time-steps, the network should learn not to produce sixteenth notes. The result of this process is an array of note names each representing the note played at a given time-step.

The "Design" section (4) describes the necessity for the music to be separated into sequences of fixed size, with the next corresponding note to be played as the output. Figure 6.1 shows the algorithm for generating these input sequences and output notes. Following this, the network inputs and outputs are one-hot encoded. This is done by encoding each note pitch with a number from zero to the number of pitches, and then replacing each pitch in the inputs and outputs with a vector of 0s and a single 1 at the index of the encoded note. This results in the inputs being matrices stored as two-dimensional NumPy arrays [41], with time on one axis and the note played on the other, and the outputs being single vectors stored as one-dimensional NumPy arrays representing one note.

Algorithm 2: Obtaining the network inputs and outputs

```
Data: pieces, length
// pieces (array):  all of the pieces to train on
// length (int):  length of sequence (or context) required
Result: inputs, output
// inputs (array):  array of network inputs
// outputs (array):  array of network outputs
inputs ← [] ;                                          // initialise empty array
outputs ← [] ;                                         // initialise empty array
for p ← pieces do
    // Iterate over each piece
    for i ← 0 to length(piece) − length do
        // Iterate over each note in piece
        inputs.append(p[i : i + length]) // Append sequence of notes to inputs
        outputs.append(p[i + length]) // Append next note to outputs
    end
end
```

Figure 6.1: The algorithm used to produce the network inputs and outputs for training.

The final stage of preprocessing consists of shuffling the data and separating it into train, validation and test data, and then serialising it for later use. Initially, the serialisation step was not taken, and

the preprocessing stage was run before the training of each model; however, this was deemed very inefficient, so serialisation was introduced. The splitting of the data is done using the sklearn function "train_test_split" [40], which splits its inputs into specified fractions. A split of 80:10:10 was chosen. This is relatively conservative with test data; however, the initial dataset was not deemed large enough to allocate more. Object serialisation was performed using the pickle Python package [52].

6.2.1 Data Augmentation

During the evaluation stage of the second development sprint, it became apparent that the size of the dataset was not sufficient for training the network. Two problems with the results from the second sprint were attributed to this: the system was only producing good quality music with a very overfit network, and the system was unable to appropriately fit music into bars due to varying bar lengths. The second of these problems was attributed to a lacking dataset because the proposed solution involved training networks for each time signature, thus ensuring equal bar lengths for each piece. This separation of networks involved dividing the dataset further, creating even smaller datasets to train on. It was decided that methods for expanding the dataset must be explored in order to improve the performance of the system.

The Music21 library contains a corpus of pieces obtained from a wide variety of sources [53]; this dataset was viewed as a potential source of expansion. The theory behind expanding the dataset using non-sight-reading pieces consisted of the following steps:

1. Extract violin parts from a dataset.
2. Filter the parts by imposing the quantitative restrictions defined in the grade specification, for example, the note range.
3. Remove the parts that also fit the specification for the grade below. This step is used to remove any pieces that would be regarded as too easy to play.

This method was seen as a way of expanding the dataset with sufficiently difficult music; however, even imposing the light restrictions on the note-lengths and pitches decreased the original dataset exceedingly. At this point, the attempt at expanding the original dataset with new music was conceded. This is because it was obvious that imposing all restrictions would result in only a couple of extra pieces from a corpus of over 3000 pieces, and the pieces obtained would not necessarily represent the types of melodies that feature in sight-reading examination pieces.

Data augmentation was the other method attempted, which involves altering the current data in some way to expand the dataset [23]. Data augmentation with image data can include: rotating, flipping and cropping images, and in many different combinations. In order to augment the pieces of music in the dataset, it was decided that transposing each network input/output combination up and down the stave would be the best option. Transposing an extract of music entails changing its pitch and key, whilst retaining its tonal structure [54]; this is displayed on the staves in Figures 6.3, 6.4, 6.5 and 6.6. The two aspects of the piece which change during transposition are the pitches of the notes and the key signature. For this reason, a transposition is only regarded valid for the augmented dataset if its key signature is allowed in the specification and all of the note pitches lie within the pitch range

defined in the specification. The pitch checks are done by tracking the maximum and minimum pitches of an extract of music before transposing it up and down until the range constraint is violated. Data augmentation saw the grade 5 violin sight-reading dataset increase from ∼4000 sequences to ∼60000 sequences; this increases the dataset by a factor of 15.

Using the data augmentation technique described facilitated the splitting up of pieces based on the time signature. The significance of this is that the network is always trained on music with equal bar lengths and thus can fit notes within bars easily. For the case of grade 5 violin sight-reading, this means training four networks, one for each time signature allowed in the grade.

Figure 6.2: Data augmentation techniques performed on images [23]

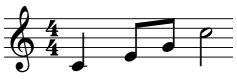


Figure 6.3: Untransposed notes.

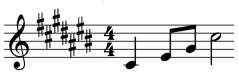


Figure 6.4: Notes from Figure 6.3 transposed up by one semitone.

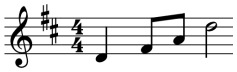


Figure 6.5: Notes from Figure 6.3 transposed up by two semitones.

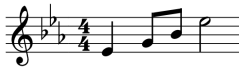


Figure 6.6: Notes from Figure 6.3 transposed up by three semitones.

6.3 Network Implementation

The network is implemented using the Keras library [30] which is run on a TensorFlow backend [42]. The simplicity of the network design, a classification network with a single input and output, meant that it could be implemented with the Keras "Sequential" model. Sequential describes the linear stack of layers used, with each layer connecting to the next. Some features of the network are not hardcoded, such as the number of LSTM units per layer, the dropout after each layer and the optimiser. These features are taken as parameters when the "create_model" function is called to allow for multiple different structures of the network to be tested. Figure 6.7 shows a diagram of an LSTM network implemented using the Keras sequential model.

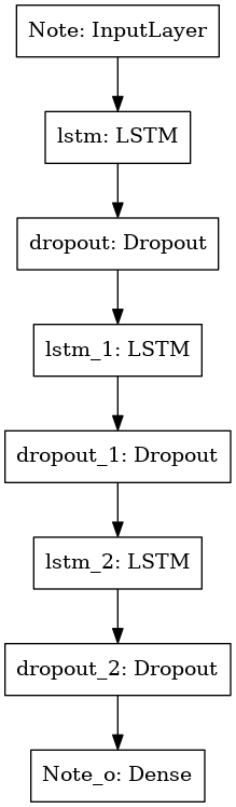


Figure 6.7: Diagram showing the structure of a simple Keras-generated model.

6.4 Training The Network

The training of networks over the duration of the project was performed on the University of Warwick Department of Computer Science compute nodes [55]; this is because they are designed to run heavy computation processes like training neural networks. The compute nodes can be accessed via secure shell, code was transferred by pulling code from GitHub, and trained models were retrieved using secure copy.

Two main training hyperparameters are to be taken into account when training a neural network using mini-batch gradient descent: the number of epochs to train for and the batch size. The number of training epochs remained relatively fluid throughout the project, with the majority of networks being trained for between 50 and 100 epochs. As mentioned previously, the loss of the network on the training and validation data is recorded throughout the training of a network to gauge the model's improvements in performance. Model checkpoints are used to save the state of the network parameters at epochs that produce the minimum loss and validation loss. When applied to the saved structure of the model, the saved weights can be used to restore the conditions of the network at its "optimal" state during training. The batch size chosen for training the network was fixed at 128 data samples; this (quite large) mini-batch size was selected due to its promise of smooth updates to the network.

The directory structure of the model proved to be very important during training to keep track

of all aspects of the system and network. Figure 6.8 shows the directory structure of a model folder, the folder created upon training a new model. More files are added to the model folder after training, such as test results and generated pieces.

```
model_folder ..... folder containing all information about the model
├── model_choices.pickle ..... dictionary containing model choices, e.g. time signature
├── model_summary.txt ..... summary of model structure and hyperparameters
├── model.h5 ..... saved model structure
├── model.png ..... diagram of model structure, as in Figure 6.7
├── output_weights ..... folder containing saved model weights
│   ├── loss ..... folder containing weights of network at minimum training loss
│   │   └── best_loss_weights.hdf5 ..... model weights at minimum loss
│   ├── val_loss ..... folder containing weights of network at minimum validation loss
│   │   └── best_val_loss_weights.hdf5 ..... model weights at minimum validation loss
│   ├── accuracy_history.png ..... plot of model accuracy against epochs
│   ├── loss_history.png ..... plot of model loss against epochs
│   └── training_results.pickle ..... metrics from training, e.g. loss history
└── resource_info.json ..... model resources info, e.g. location of data
```

Figure 6.8: Diagram showing and describing the files kept when training a model.

6.5 Generating Music and Post Processing

Generated music using a trained model consists of a number of steps:

1. Load the model structure and weights of the trained network.
2. Select a seeding method to initialise the network; this gives the network a sense of the key and style of music it is being asked to generate.
3. Generate single notes repeatedly using the sliding-window generation method illustrated in Figure 4.1 and described algorithmically in Figure 4.2.
4. Decode the output notes from one-hot vectors of note pitches.
5. Join the notes together - each note currently represents a sixteenth note time-step.
6. Select a key signature for the piece of music.
7. Export the music as a MusicXML file.

The initial step of generating music consists of loading the saved, trained model and assigning the saved weights to it. Next, an input seed needs to be selected in order to inspire the network to generate unique music. As discussed in section 4.3.1 “Seeding Methods for Generating New Music”, two seeding methods were considered: using the ends of pieces from the dataset or an artificially generated seed. Once a seed has been produced, it needs to be encoded in the same way the data was encoded for training the model. This involves encoding the seed using the same encodings as used for the training data, and then converting each note into a one-hot encoded vector. The seed is then fed into the

trained network to begin the sliding-window generation. Once a piece of the desired length has been generated, the neural network is no longer required, and the post-processing can begin.

The output of the sliding-window algorithm is an array of one-hot vectors, or a one-hot matrix with the notes on one axis and the time-steps on the other. The first step in post-processing is to split the notes into bars; this is done because the notes need to be re-joined, and notes cannot be joined across bar lines. The number of time-steps in a bar can be calculated from the time signature. The notes are then joined within each bar if they are identical and form note lengths allowed within the specification; this process reforms the notes into variable lengths, applying a rhythm to the generated piece.

The key applied to the piece is calculated using the Krumhansl-Schmuckler key-finding algorithm [56]; Music21 contains an implementation of this algorithm for deciding the key of a piece by analysing the frequency of note pitches. Initially it was undecided as to whether the original key of the seed should be used as the key of the generated piece, however using the Krumhansl-Schmuckler algorithm produced pieces containing fewer out-of-key notes. Once the key signature has been applied, the piece can be exported into the MusicXML file format.

6.6 Extending The Network to Include Extra Musical Features

As discussed in the project specification [8] and the scope (1.2) and requirements (4.1) sections of this document, the network should be able to take into account musical features other than just notes. Initially, when designing the new system, the violin grade 5 sight-reading dataset was analysed for the types of musical features it contains. These musical features are grouped by their common parent objects in the Music21 library, and the proposed way of processing and predicting them. Features are split into the following categories, ordered by the proposed importance of their inclusion within the network:

1. **Notes**: Notes have already been accounted for.
2. **Slurs**: Slurs are particularly important in stringed instrument sight-reading grades; slurs indicate that two or more notes should be played without separation. In the case of stringed instruments, the notes must be played without changing the direction of the bow.
3. **Discrete Dynamics** (Dynamics): Dynamic markings such as “forte” or “piano” indicate the volume at which a section of music should be played.
4. **Spanners**: Spanners are named after the parent object of a group of musical features in Music21; spanner objects are features that span over multiple notes. Crescendos and diminuendos are examples of this.
5. **Note Expressions and Articulations** (Articulations): Note expressions and articulations that are specific to a note, such as bow markings and “staccato” notes.

6. **Text Expressions**: Text expressions are written words and phrases indicating the style of play, for example, “pizzicato” or “a tenuto”.

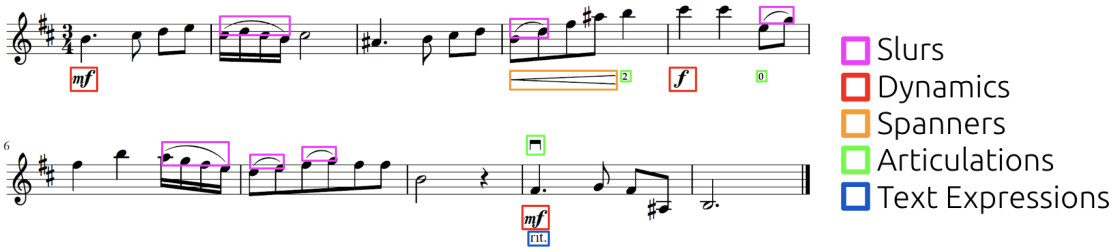


Figure 6.9: Stave highlighting the different musical features taken into account by the network.

The importance of their inclusion was decided by a combination of their frequency within the dataset and the extent of the consideration by the player. The theory behind this was that the less important a musical feature is deemed to be, the easier it is to add post-generation due to its sparsity in pieces of sight-reading music - adding an up-bow marking is easier than adding all of the slurring to a piece.

The addition of musical features prompted the contemplation of how they would be handled within the network. These features can be generated sequentially along with the melody, since there exists dependencies between them all. Features are split into two categories: single-label classification features and multi-label classification features. The single-label classification features are ones which require a single instance to be selected at every time-step - notes are an example of this (one pitch per time-step). The multi-label classification features are ones where more than one instance can appear in a time-step and, more importantly, none of them can be chosen to appear in a given time-step - text expressions are an example of this. The only single-label classification feature, other than notes, is the discrete dynamics, where one dynamic marking must be chosen for each subsection of music; this can be implemented in the same way as notes are, as a single-label classification network.

For the multi-label classification features, the way the network calculates loss and the activation function of the output layer needs to be changed. The loss function for the single-label classifier was categorical cross-entropy loss (see Figure 4.9); this loss function is specifically designed for single-label classification. The binary equivalent, binary cross-entropy loss, performs the same calculations as categorical cross-entropy loss but over two categories. Rather than choosing a class from a number of output nodes, each node is evaluated as its own binary classifier. This is ideal for the multi-class problem as multiple output nodes can be chosen, or even none at all. The activation function for the output layer for the multi-class features is the sigmoid function (see Figure 2.3). The sigmoid function was used over the softmax function because each output node can be evaluated independently, not as a probability of selection in comparison with other nodes. Additionally, a node is classed as “chosen” if its activation is greater than 0.5; in the softmax function, only one node can have an activation of greater than 0.5.

Adding these features to the network involved switching to the Keras functional API [30]; the Keras functional API allows the user to specify the inputs and outputs of each layer within the network. The reason this switch was required was due to the support of multiple inputs and outputs to the network;

this is essential for using different activation and loss functions for the outputs of the network. For modularity in adding and removing musical features, separate inputs and outputs are used for each feature. Figure 6.10 shows the network structure of a multi-input, multi-output model.

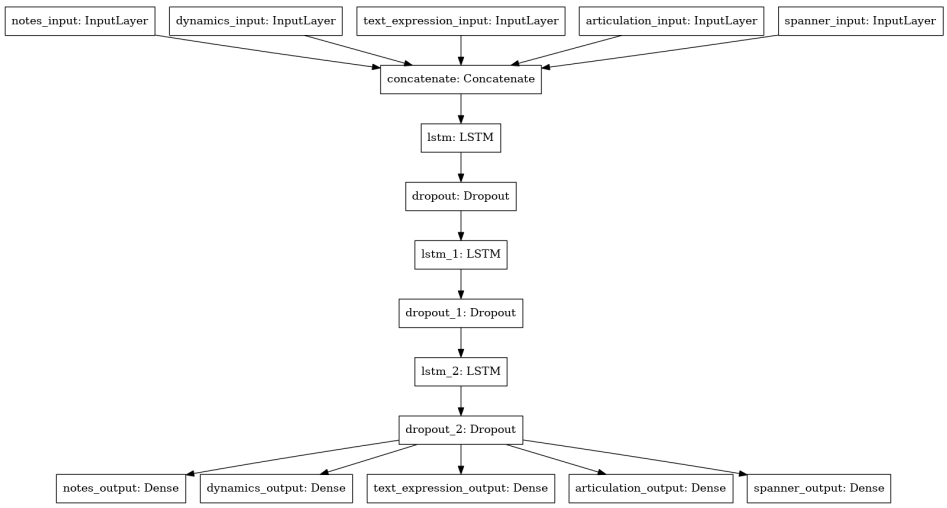


Figure 6.10: Multi-input multi-output neural network example.

6.7 Evaluating The Network With Extra Musical Features

The loss of a multi-output network is a concatenation of each individual loss. Weights can be applied to each loss; however, for simplicity, no loss weights have been applied in this implementation. Separate accuracy metrics are calculated for each output during training and testing. During testing, each of these accuracy results need to be taken into account; however, as a rule of thumb, the accuracy between each feature tends to correlate. The "Note" output accuracy should still be used as the principal accuracy function, since it is the dominant feature of the network.

Confusion matrices, although mainly for single-label classification problems, can still be used to visualise the true positive, true negative, false positive and false negative rates of a trained multi-class model on test data. This is done by separating the feature into each of the possible labels and creating individual matrices for each; each matrix contains a "feature on" and "feature off" section on each axis. Figure 6.11 shows a confusion matrix for slurs.

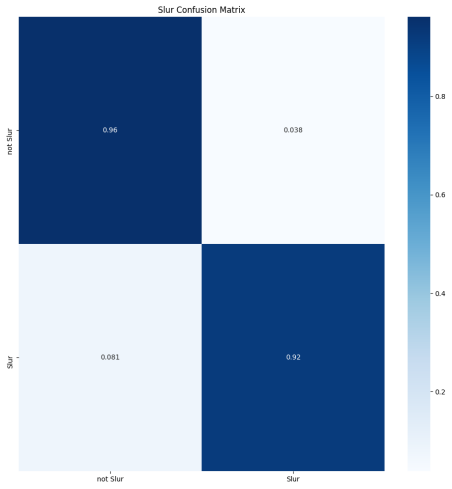


Figure 6.11: A confusion matrix of true slur status (y-axis) against their respective predicted slur status (x-axis). The squares represent true negatives, false positives, true positives and false negatives in the top-left, top-right, bottom-right and bottom-left respectively.

6.8 Preprocessing With Extra Musical Features

Adaptations to the preprocessing were required in order to account for extra musical features. The original preprocessing of notes remains the same; however, each musical feature contains small variations on how they are preprocessed. Separate functions are required to extract the musical features from the scores - these methods are described below:

Slurs: Slur objects are extracted from the piece, and the first and final note of each slur is identified. A note is considered "slurred" if it is slurred onto the next note in the sequence, leaving the final note of the slur being considered as unslurred. This is so that the network can recognise when one slur ends, and another begins, providing no negative impact on the recognition of the final note of the slur, since all slurs are extended by one note in the post-processing.

Dynamics: All discrete dynamic markings and their relative offsets are extracted from the piece. The dynamic at any time-step is regarded as the most recent dynamic marking, and only changes when a new dynamic marking is written.

Spanners: Spanners are preprocessed in the same way as slurs since slurs are actually a type of spanner object. However, they are separated for the purpose of including slurs without having to include all other spanner objects.

Articulations: The types of note expressions and articulations are encoded and set to a 1 in a vector if they feature on the note that is represented in the given time-step.

Text Expressions: The types of text expressions are encoded and set to a 1 in a vector if they feature at a given time-step.

All musical features are referenced by their type in the encodings and decodings. This means they can be added dynamically and instantiated in post-processing without the hard-coded knowledge of their exact type, only their parent class which defines how they are processed by the system. To fit with the augmented (transposed) note sequences, all feature sequences need to be copied according to the number of suitable transpositions found for each note sequence. Helpfully, Keras models can take named inputs and outputs so data can be stored in dictionaries, and no further consideration needs to be taken to correctly index them. When a model is trained, the required features are selected. For example, if notes and slurs are selected, only the note and slur inputs and output are used.

6.9 Training With Extra Musical Features

The addition of extra musical features requires no major changes to the training of the network. One observed difference between the training of just notes and the training of additional features was that the loss and accuracy of feature predictions converged much faster and in general, were far more accurate. This could be due to a number of factors, including fewer classes to predict, more repetitive patterns, and sparser data. The latter of these factors is an issue since the network might deliberately choose to never activate a node if it is only activated in 1 in 100 time-steps - still achieving 99% accuracy. For this reason, experimentation was done with class weightings proportional to the inverse of each label's frequency and the use of a loss function which places more focus on the hard-to-classify classes. The first of these solutions implements a higher loss weighting for less frequent classes, for example, the loss for incorrectly predicting a diminuendo (a false negative) is lower than the loss for incorrectly predicting the absence of a diminuendo (a false positive). The latter of these solutions is called focal loss, which defines an exponential decrease in the resultant loss with reference to the classification probability. This increases the importance of correcting wrongly classified samples [57]. An implementation of this loss function for use with Keras was found on GitHub [58].

$$f(p_t) = -\alpha(1-p_t)^{\gamma} log(p_t)$$

Figure 6.12: Focal loss function on a single example p_t;γ and α are hyperparameters indicating the level of focus [57].

6.10 Generating Music and Post Processing With Extra Musical Features

Applying seeds with more features became a challenge, since it was initially believed that the seed would need to cover all musical features. This is simple for the end sequence seeds since they already have their own features. However, this is a challenge for artificial seeds because there is no simple way to generate musical features for a seed whilst maintaining musical integrity. After further contemplation,

it was realised that due to the sparsity of most features, namely text expressions, articulations and spanners, it would be feasible that in one context none would appear. This leans on the premise that the outputs of the network for generating these features are perfectly capable of deciding that none should be present in a given time-step. Since these features are also connected to the other inputs of the network, such as notes, it is feasible that many of the dependencies for generating them rely on changes predominantly to the other inputs. An example of this might be that a diminuendo is often present when the note lengths get longer and the melody slows down, rather than a crescendo happening three bars ago. The only other features taken into account when generating artificial seeds are as follows:

Slurs: Slur patterns for a seed are generated in the same way rhythms are generated; rhythms are chosen from the frequencies of whole-bar rhythms appearing in the dataset - this is described further in section 4.3.1. The addition of also taking into account the slur patterns over these rhythms and storing them as a tuple of (rhythm pattern, slur pattern), enables the generation of slur seeds that match the seed rhythm.

Dynamics: The single-label nature of the discrete dynamics does not necessarily mean that an input is required. However, in all training sequences, a dynamic is present at every time-step. For this reason, the network should learn that the majority of dependency from the dynamics output comes from the dynamics input, since in most cases, the dynamic remains the same from time-step to time-step. This sparsity of changes to the dynamics led to the decision to randomly choose one dynamic marking for the input and sustain it throughout the entire seed.

All other feature inputs to the network for the initial step are zero matrices, signifying none of the features were present in the last window.

The same sliding window technique is used to generate music with extra musical features. The format of the input to the trained network differs slightly in the fact that it requires a number of vector sequences, each specifying the inputs of a different musical feature. As mentioned previously, Keras supports named inputs to the network, so the input is formed as a dictionary of feature names and their respective seeds. Once a single note has been chosen by the network, the same process of selecting the highest probability class and setting it to a 1 is used for notes and dynamics. For the multi-label features instead of selecting the maximum node activation, any output node with an activation of greater than 0.5 is set to 1 with all others set to 0. Once a whole piece has been generated, all features are decoded.

Identifying identical adjacent notes, and more importantly, distinguishing similar but non-identical notes, should only be improved through the addition of extra musical features. Time-steps that have some distinguishing factor from the preceding or succeeding time-steps should not be joined together to form one longer note. Adding slurs, dynamics, spanners and articulations provides another dimension used for comparison. Text expressions are the only feature not used in the joining of notes due to their placement at a specific time-step rather than at the position of a note - including these would incorrectly separate notes. The Music21 library provides functionality for adding other musical features to the stave, so no further post-processing is required past the note joining stage.

6.11 Running The System

As mentioned in section 3.2, different files are used to separate the functionality of the system. In addition to these files, there are files containing the "driver code" used to run each stage on some given data. All driver code scripts contain command line parameters which can be used to change how the code is run - most parameters are optional and have default values. Argument parsing is handled by the argparse package [59]; all optional and required arguments for a file, along with their descriptions, can be found by running the following code in the command line: `python <filename>.py --help`.

6.11.1 Inputs Required

The inputs required in order to generate sight-reading music are: a dataset of test pieces and a JSON file containing the requirements of the instrument and grade. Figure 6.13 shows the requirements JSON file used to test the system; the format of each key-value pair is explained below:

quarter_lengths : These represent the allowed note lengths, each defined as a fraction of a quarter note (crotchet).

time_signatures : The possible time signatures that could appear in the grade.

key_signatures : The possible key signatures that could appear in the grade. Positive numbers indicate the number of sharps; negative numbers indicate the number of flats.

rest_lengths : The allowed lengths of rests, each defined as a fraction of a quarter note (quaver).

length : The range of piece lengths that could feature in the grade, measured in number of bars.

pitch_range : The range of pitches allowed in the grade.

keys : The possible keys that could appear in the grade. Capitalised letters represent major keys, lowercase letters represent minor keys, and the "-" character represents "flat".

```
{
  "quarter_lengths": [1, 0.5, 2, 3, 1.5, 0.75, 0.25],
  "time_signatures": ["4/4", "2/4", "3/4", "6/8"],
  "key_signatures": [1, 2, 3, 0, -1, -2, -3, 4, -4],
  "rest_lengths": [1, 2, 0.5],
  "length": [8, 16],
  "pitch_range": ["G3", "E6"],
  "keys": [
    "D", "A", "G", "e", "C", "F", "B-", "a", "d", "g", "E-", "E",
       "A-", "b", "c"
  ]
}
```

Figure 6.13: Contents of requirements.json for ABRSM grade 5 violin.

6.11.2 Preparing Data

All data preprocessing can be run using the `data_preparation.py` script. This script takes in a folder containing the data and requirements for a specific instrument grade, as well as a number of optional arguments, and performs all data processing required. This includes, but is not limited to: all preprocessing, analysing the data for seed rhythms, and splitting the data into train, test and validation data. Figure 6.14 shows all of the parameters accepted by `data_preparation.py`.

data_preparation.py parameters				
Char Flag	Flag	Description	Default Value	Required
-f	--folder	Folder containing requirements (requirements.json) and data (data/).	n/a	Yes
None	--time_signatures	Time signatures to create data for.	All available time signatures	No
-a	--dont_augment_data	Don't augment data flag. -a for don't augment, nothing for augment.	False	No
-b	--bars_context	Length of musical context given to the neural network.	4	No

Figure 6.14: All parameters accepted by `data_preparation.py`.

6.11.3 Training a Model

A model can be trained by running the `driver.py` script. The only required parameter is, again, the folder containing the data (and now preprocessed data too), and the JSON file containing the grade requirements. Optional parameters can be used to modify the default layout of the model or to change other aspects of training. Figure 6.15 shows all of the parameters accepted by `driver.py`.

driver.py parameters				
Char Flag	Flag	Description	Default Value	Required
-f	--folder	Folder containing requirements (requirements.json) and data (data/).	n/a	Yes
None	--features	Which extra musical features to have. Slurs: s, Dynamics: d, Spanners: p, Articulations: a, Text expressions: t. Example format: --features sdpat.	s (although notes are input by default, to specify a model containing only notes "--features n" can be used)	No
-a	--dont_augment_data	Don't augment data flag. -a for don't augment, nothing for augment.	False	No
None	--early_stopping	Stop the model once the validation loss has reached a minimum.	False	No
-u	--lstm_units	Number of LSTM units in each layer.	512	No
-d	--dropout	Dropout after each LSTM layer.	0.3	No
-l	--lstm_layers	Number of LSTM layers in network.	3	No
-s	--single_loss_function	Loss function for single-label classification features (e.g. notes).	categorical-_crossentropy	No
-m	--multi_loss_function	Loss function for multi-label classification features (e.g. expressions).	binary-_crossentropy	No
-o	--optimiser	Optimiser for network.	Adam	No
-i	--last_layer_inputs	Input multi-label features before final layer of network.	False	No
None	--time_signatures	Time signatures to create model for.	All available time signatures	No
None	--epochs	Number of epochs to train for.	60	No

Figure 6.15: All parameters accepted by `driver.py`.

6.11.4 Generating Music

Music can be generated from a trained model by running the `generator_driver.py` script with the location of the model folder as a parameter. Figure 6.16 shows the parameters accepted by `generator_driver.py`. If neither the loss_flag or val_loss_flag are chosen, both are set to true. If neither the end_sequences_flag or artificial_seeds_flag are chosen, both are set to true.

generator_driver.py parameters				
Char Flag	Flag	Description	Default Value	Required
-m	--model_folder	Folder containing model.	n/a	Yes
-p	--no_generated_pieces	Number of generated pieces.	50	No
-l	--loss_flag	Generate music from the weights saved at the minimum loss of the network during training.	False	No
-v	--val_loss_flag	Generate music from the weights saved at the minimum validation loss of the network during training.	False	No
-e	--end_sequences_flag	Generate music from end sequence seeds.	False	No
-a	--artificial_seeds_flag	Generate music from artificial seeds.	False	No

Figure 6.16: All parameters accepted by `generator_driver.py`.

Results

This section describes the results of the evaluation of the system. For each section, only the most relevant evaluation metrics are mentioned.

7.1 Without Data Augmentation

Data augmentation was introduced at the third sprint of the development process. Prior to this, sequences of notes were extracted from the dataset without any alterations. Understandably, this produced data with significantly imbalanced classes due to the far larger representation of notes in the middle of the note range compared to the extremes. A size of context needs to be defined in order to specify the input size of the network; this was chosen to be 48 time-steps. 48 was chosen because it is the smallest number which is divisible by the number of time-steps in each time signature allowed in ABRSM violin grade 5 sight-reading examinations. Figure 7.2 shows the optimal parameters found under these conditions; Figure 7.1 shows the loss history of this model during training.

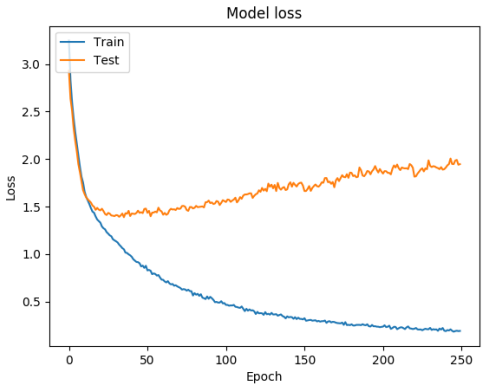


Figure 7.1: The training and validation (labelled as "Test") losses plotted against the number of training epochs.

Parameter	**Value**
LSTM layers	2
LSTM units	128
Dropout	0.3
Loss function	Categorical cross-entropy
Optimiser	RMSprop
Sequence (context) length	48
Training epochs	250
Features	Notes only
Training dataset size	∼4000 sequences

Figure 7.2: Parameter choices for optimal model found.

The reason the model needs to be trained for so long is because the music generated from networks trained for fewer epochs is monotonous and completely unsuitable. Allowing the model to substantially

overfit proved to be the only way of producing variant music. As discussed previously, overfitting is the process of a network learning aspects of the training data too specifically and under-performing when tested using validation or test data. Figure 7.1 shows the model overfitting, which is evident by the gentle increase in the validation loss (orange) from around the 40th epoch. This gives rise to the concern that any generated music is too similar to the music in the dataset. The theory being that the network regurgitates extracts from the training data rather than learning the fundamental features of good melodic and rhythmic patterns, and generating unique dataset-inspired music. Figures 7.3 and 7.4 show examples of music generated by the neural network described above. Figure 7.3 shows music generated by the network, with the weights taken from the 39th epoch of training. The constant tone played over the entire piece indicates the network has not learned the relationship between differently pitched notes and is unable to create a variant melody. Figure 7.4 shows music generated by the network, with the weights taken from the 230th epoch of training. This piece is significantly different to the piece generated from the 39th epoch, as it contains many different pitches and appears to have some concept of a melody.

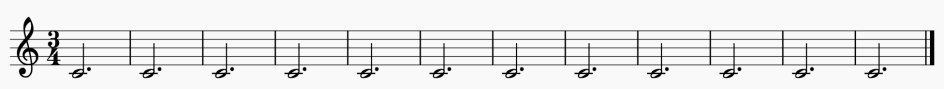


Figure 7.3: Music generated with an artificial seed by taking the weights at the minimum validation loss of the network. This occurred at the 39th epoch of the model trained in Figure 7.1.

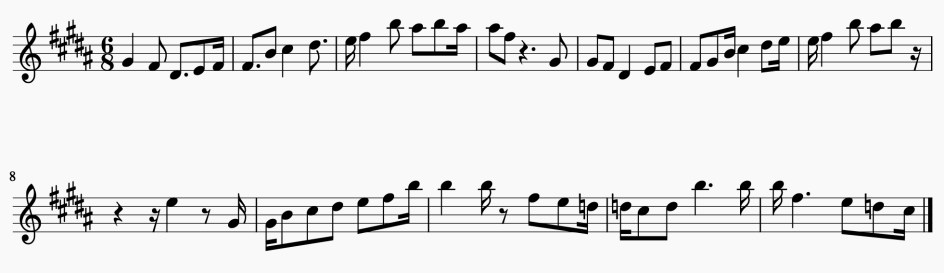


Figure 7.4: Music generated with an end seed by taking the weights at the minimum loss of the network. This occurred at the 230th epoch of the model trained in Figure 7.1.

Looking at the datasets of pieces generated using different seeding methods and network weights, the relative quality of the music can be inferred from its similarity to the original dataset. Figure 7.5 shows the spread of the number of notes featuring in each generated dataset. The left boxplot, showing analysis from the network generated with the minimum validation loss weights applied, shows meagre results in generating music containing multiple pitches. The number of notes generated per piece does not extend past four, which is very insufficient. The right boxplot shows the analysis from the minimum loss-weighted network. This plot shows a remarkable difference in the network's ability to produce music containing multiple pitches consistently. The seeding method showing the worst performance is the end-of-piece seeding method. This could be due to the frequency of rests within the last few bars of the music in the original dataset and the general slowing down of the melody, providing less variant end seeds. This is shown in Figure 7.6, where the end-seed-generated music

dataset contains almost three times as many rests as the original dataset.

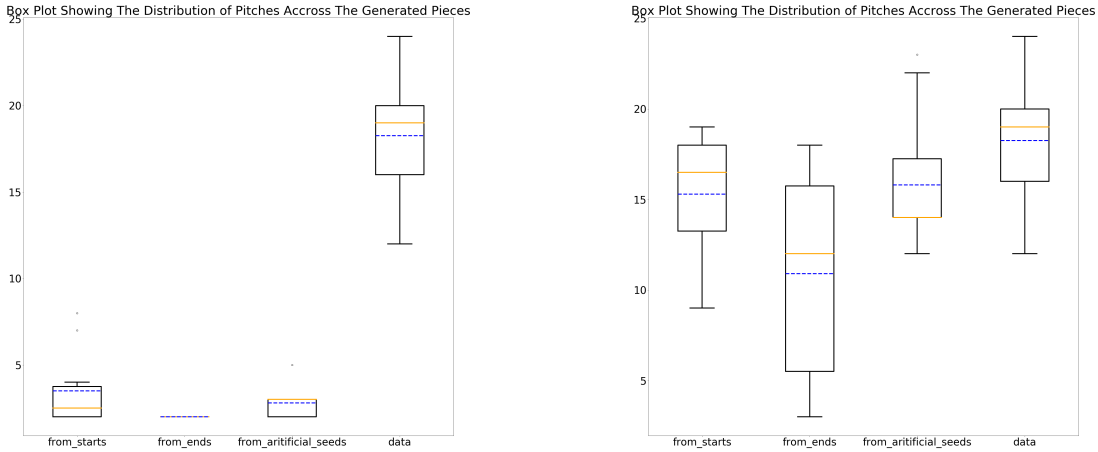


Figure 7.5: Boxplots showing the range in the number of pitches per piece generated with different seeds, using the minimum validation loss and loss weights respectively. In these plots the box represents the interquartile range, with the whiskers reaching to a maximum of 1.5x the interquartile range. The orange line indicates the median value and the dashed blue line indicates the mean. The key on the x-axis indicates the seeding method used. The "data" label represents the pieces from the original dataset. The "from_starts" label references a seeding method discarded from consideration past the second sprint, due to it being regarded as unsustainable and therefore not useful.

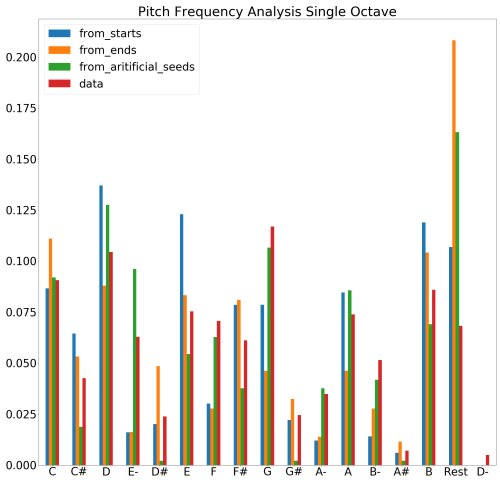


Figure 7.6: Graph showing the frequency of pitches within each dataset of generated pieces as a fraction of the total number of pitches.

The datasets generated by the network with the weights saved from the 230th epoch, where the minimum loss was recorded, were used for the human evaluation. At this stage, four test subjects submitted results. The results from the human evaluation showed the majority of generated pieces to be unsuitable for violin grade 5 sight-reading (see Figure 7.8). From the other questions asked to the test subjects, the rhythmic difficulties of pieces showed interesting results. The generated pieces were reported to, on average, contain more difficult rhythms (as seen in Figure 7.7). This was attributed to the inability of the network to gauge where each bar ends, which is due to the variety of bar lengths the network was trained on. Examples of this type of confusion are visible in Figure 7.4 where notes are often pitched the same across bar lines. This indicated that the network intended for them to be joined in post-processing, however, their appearance in different bars prevented this. A result of this is the occasional occurrence of stray semi-quavers (sixteenth) notes, which *should* only really occur in pairs or when succeeding a dotted quaver.

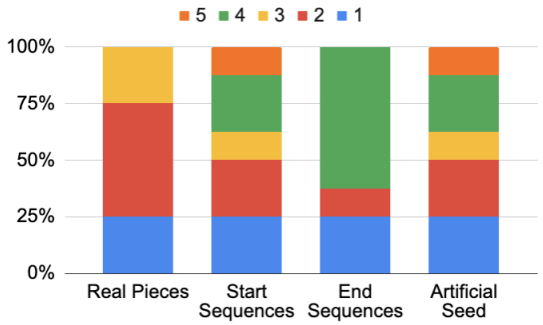


Figure 7.7: Bar graph showing the levels of difficulty (from 1 to 5) of the rhythms in pieces reported by the test subjects. The y-axis represents the proportion of responses for each possible response type.

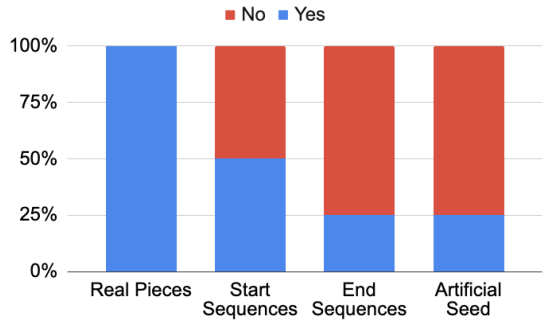


Figure 7.8: Bar graph showing the appropriateness (yes/no) of pieces for grade 5 sight-reading reported by the test subjects.

7.2 With Data Augmentation

The use of data augmentation allowed for the splitting up of data based on the time signature. This means that the context length can be defined in the number of bars rather than time-steps; four bars was chosen to be suitable. The results in this section come from the training of networks on music in the time signature 6/8 (four bars of 6/8 music is 48 time-steps).

The optimal classification network was selected through a number of grid searches over a selection of hyperparameters. The largest of these grid searches involved creating 48 different models based on all combinations of the hyperparameters listed in Figure 7.9; the entire results table of this grid search is available in the appendix (see Table 1). The results of this grid search concluded that a model containing 256 units, 2 LSTM layers, 0.2 dropout, trained with the Adam optimiser achieved the highest accuracy when tested on the weights saved at the minimum validation loss. Although the results were very close, this model was still chosen as the optimal network at this stage. The full evaluation results for this model, with the weights saved at the minimum loss, are available in the

appendix. Grid searches of this size were not feasible in the later stages of the project due to the increase in model size, resulting in greater computation cost. Larger models containing 512 LSTM units were tested after this gird search; however, the results showed significant overfitting at an early stage in the training process with similarly shaped loss history plots to that in Figure 7.1.

LSTM units	LSTM layer	Dropout	Loss function	Optimiser
64	**2**	**0.2**	**categorical cross-entropy**	**Adam**
128	3	0.3	categorical focal loss	RMSprop
256				

Figure 7.9: Table containing model hyperparameters. The hyperparameters for the chosen model are in bold.

This model showed good results in the confusion matrices (shown in Figure 7.10), with the majority of notes being classified correctly with an accuracy in the high sixties or low seventies. However, there are a couple of instances of misclassification. The note D# is classified incorrectly in all instances, however, it only features in one key signature allowed in ABRSM grade 5 violin sight-reading, so rarely features in the original dataset. Figures 2 and 3 in the appendix show the confusion matrices over the entire note pitch domain for both the optimal augmented and non-augmented models. The augmented model matrix shows an improvement from the non-augmented matrix especially in the highest and lowest pitches, where class imbalance is prevalent in the original dataset; transposing the sequences increases the representation of the notes at the extremities of the pitch range within the dataset.

Figure 7.10: Confusion matrix created using the optimal grid search model with the weights stored at the minimum validation loss. The y-axis represents true labels; the x-axis represents the predicted labels.

The analysis of generated music shows far better results than obtained previously; the network is able to produce variant music with the weights taken at the lowest validation loss. However, in comparison to the original dataset, there is a significant difference between the number of unique notes per piece (see Figure 7.11). Taking the weights at the minimum recorded loss, around the 60th epoch of training, produces datasets containing more variance. Figure 7.12 shows this spread, with over 50% of pieces for each generation method falling in the same range as the original dataset. These results do not resemble the original dataset, in terms of the number of unique pitches per piece, as well as with the unaugmented data. However, this network is only minimally overfitting in comparison to the considerably overfit network with augmented data, which needed to be trained for 250 epochs to provide any suitable results. This is visible in Figures 7.13 and 7.14 which show the changes in the losses and accuracies of the network against the number of epochs it is trained for. The performance of end-seed-generated music shows an improvement from the unaugmented results. This is mainly due to the reduction in the number of end seeds approved for use. To decrease the number of rests produced using this seeding method, all end seeds ending in a rest are removed from the dataset. These rests could be the result of padding applied to the beginnings and ends of pieces in the data preparation stage to remove the necessity of the system to account for anacruses.

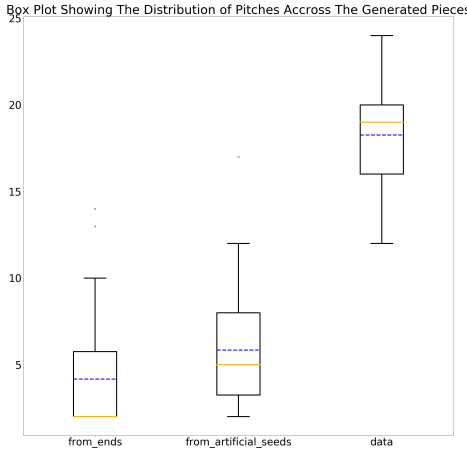


Figure 7.11: Boxplot showing the range in the number of pitches per piece generated with different seeds, using the weights saved at the minimum recorded validation loss during training.

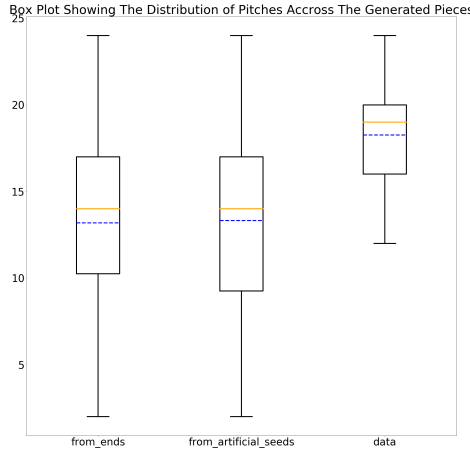


Figure 7.12: Boxplot showing the range in the number of pitches per piece generated with different seeds, using the weights saved at the minimum recorded loss during training.

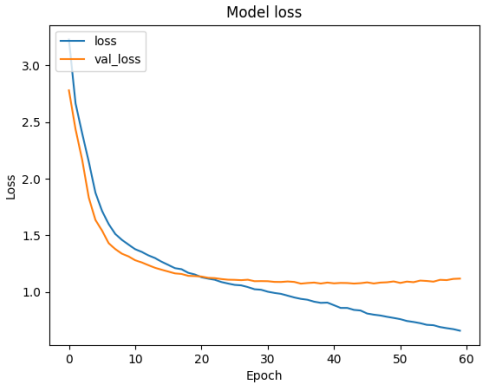


Figure 7.13: Graph showing the change in loss and validation loss during training.

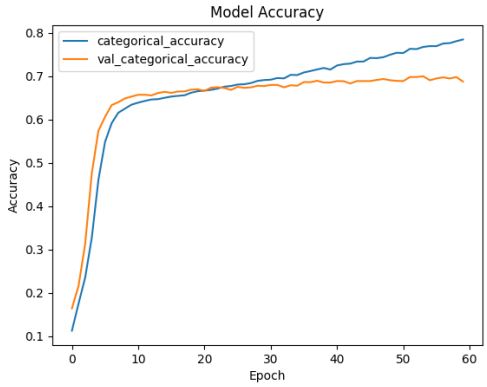


Figure 7.14: Graph showing the change in accuracy and validation accuracy during training.

Figures 7.15 and 7.16 show music generated by the network. The music generated by the weights saved at the minimum loss is far more appropriate for grade 5 violin sight-reading and resembles a coherent melody. There is also a vast improvement in the ability of the network to produce music that fits within bars. This can be seen in the lack of semi-quavers present on opposing sides of bar lines, something which is present in the unaugmented results (see Figure 7.4).

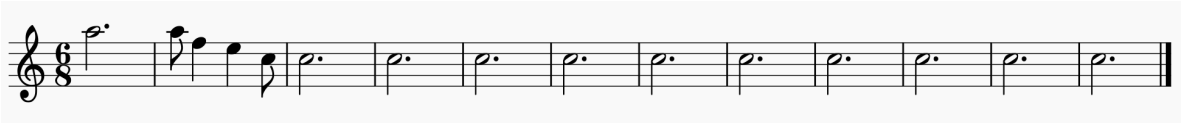


Figure 7.15: Music generated with an end seed by taking the weights at the minimum validation loss of the network.

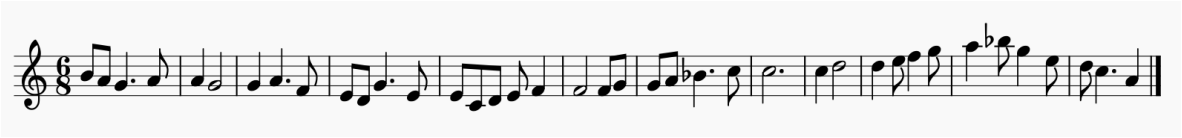


Figure 7.16: Music generated with an artificial seed by taking the weights at the minimum loss of the network.

The datasets generated using the minimum loss weights were put to five test candidates under the same conditions as the previous test; these results are available in the appendix (see Tables 6, 7 and 8). The results show a marked improvement in the difficulty of rhythms for music generated with end seeds (see Figure 7.18), with the results almost replicating those of the original dataset (real pieces). Despite the reported increase in the absolute rhythmic difficulties of artificial-seed-generated music, relative to the reported difficulties of real pieces, the results are quite similar. This observation highlights the necessity for a control dataset. The reported overall difficulty of pieces generated from end seeds seems to also replicate the difficulty of real pieces almost identically (see Figure 7.17). Those generated by artificial seeds, however, appear to be more difficult to play on average.

Figure 7.19 shows perhaps the most important metric of the evaluation, the evaluators' opinions on whether the pieces are suitable for ABRSM grade 5 violin sight-reading. 50% of the pieces generated by end seeds are said to have been suitable, compared to 20% of those generated with artificial seeds, and 90% of the real pieces.

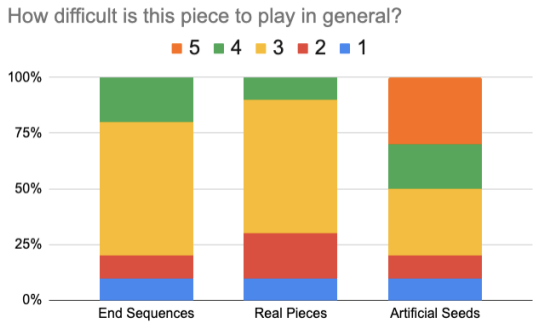


Figure 7.17: Bar chart showing the reported difficulties of pieces by test candidates, separated by seeding methods.

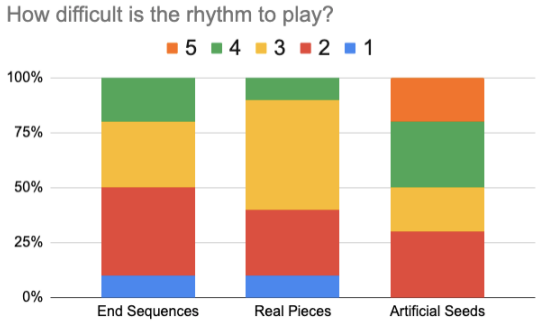


Figure 7.18: Bar chart showing the reported difficulties of rhythms by test candidates, separated by seeding methods.

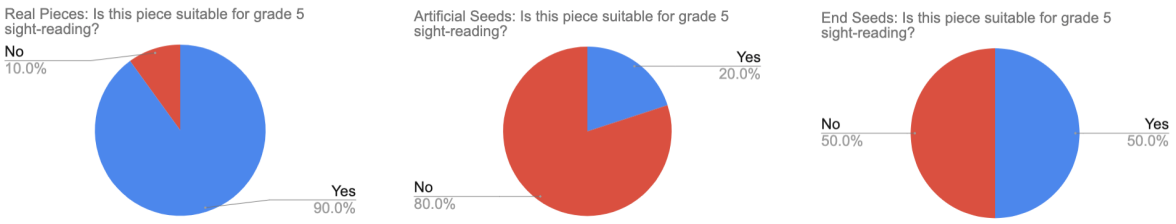


Figure 7.19: Pie charts showing the percentages of pieces suitable for ABRSM grade 5 violin sight-reading reported by test subjects, separated by seeding method. Blue represents suitable. Red represents unsuitable.

7.3 With Extra Musical Features

The addition of musical features added a surplus of required evaluation in order to cover all levels of automation. For this reason, more focus has been put on the evaluation and optimisation of models containing only the most important musical features.

7.3.1 With Slurring

The addition of slurring to the network proved to have an overall regularising effect during training, even with larger networks which have a tendency to overfit easily. Figures 7.20 and 7.21 show that even with a network trained over fewer epochs, the network containing only notes shows evidence of significant overfitting whereas the one containing notes and slurs does not.

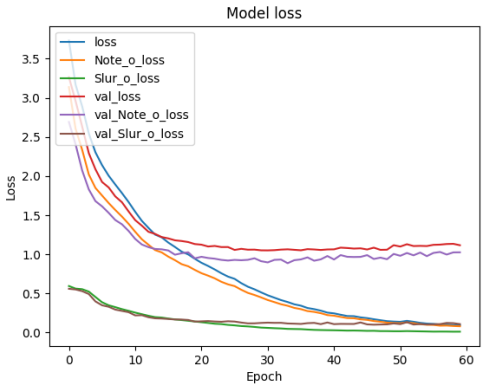


Figure 7.20: Loss history of a model trained with notes and slurs for 60 epochs with 3 layers, 512 LSTM units per layer and a dropout of 0.3.

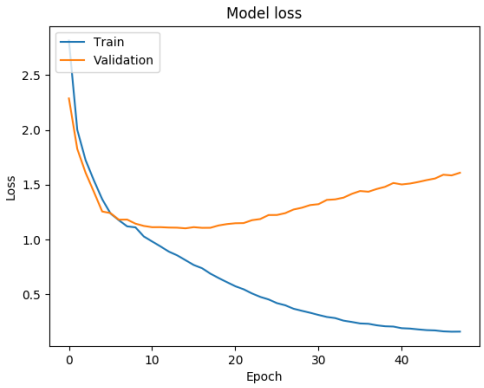


Figure 7.21: Loss history of a model trained with notes for 50 epochs with 3 layers, 512 LSTM units per layer and a dropout of 0.3.

A grid search containing 16 models over the parameters listed in Figure 7.24 was performed, each model being trained for 60 epochs. The optimal model based on the highest note and slur accuracy achieved with test data with the weights saved at the minimum validation loss proved to be: 3 layers of 256 LSTM units with a dropout of 0.3, trained with the Adam optimiser. These results are taken from Table 2, available in the appendix. Despite this, the equivalent model with 512 LSTM units was

able to obtain higher test accuracies with its minimum loss weights. For this reason, along with the results of an analysis of the generated datasets, this model was chosen as the optimal one. The full evaluation results for this model, with the weights saved at the minimum validation loss, are available in the appendix.

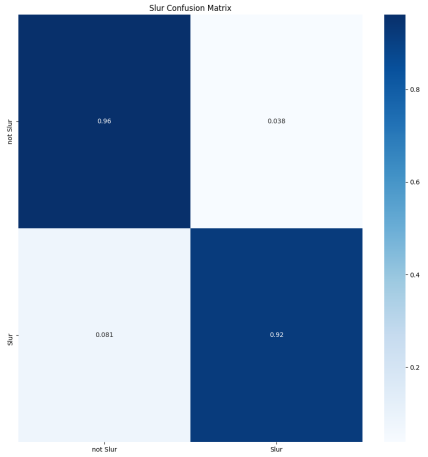


Figure 7.22: Confusion matrix showing the accuracy of slur predictions on the test data with the chosen model. The y-axis represents true labels; the x-axis represents the predicted labels.

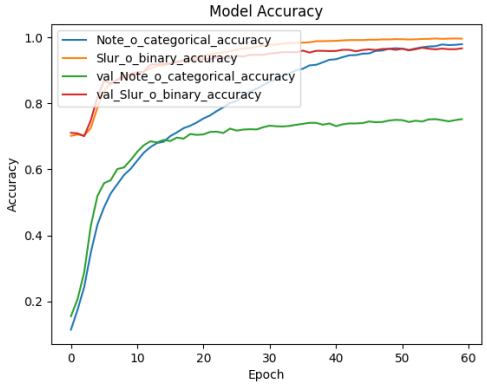


Figure 7.23: Graph showing the history of note and slur accuracies during the training of the chosen model.

An interesting observation from the test results is that the note accuracies tend to be higher in the note-slur models than in the models just containing notes (see Table 1 in the appendix). Further models were trained containing class weightings on just the slurs, and on both the slurs and notes. The results from adding class weights to the slurs were indistinguishable from the identical models trained without weights. Training with weightings on the notes as well, resulted in irregular updates to the network and no improvement on the overall accuracy. For this reason, class weightings were not considered any further.

LSTM layers	LSTM units	Dropout	Optimiser	Single-label loss function (notes)	Multi-label loss function (slurs)
2	256	**0.3**	RMSprop	**categorical cross-entropy**	**binary cross-entropy**
3	**512**		**Adam**		

Figure 7.24: Table containing model hyperparameters. The hyperparameters for the chosen model are in bold.

From a musical perspective, the music generated by this model seems to be consistently better than with just notes. There is evidence of the network learning the rhythmic dependencies, such as the relationship between dotted-quavers and semi-quavers (as seen in bars 2, 3, 10, 11 and 12 of Figure 7.26). The style of music also seems to remain relatively consistent throughout the generated pieces, with repeating rhythmic patterns over differently pitched notes. The phrasing of notes within bars

also shows a marked improvement, a sentiment echoed by the results of the human evaluation (see Figure 7.27). One noticeable hiccup of the system is the occasional separation of identical notes as the penultimate and last notes of a slur. This is seen in bar 10 of Figure 7.25 and bar 9 of Figure 7.26. The separation of these notes provides no difference to how the notes sound when played on a violin with no additional articulation marking; however, the notation does seem less intuitive than if they were joined.

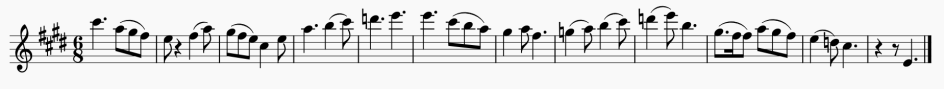


Figure 7.25: Music generated by the chosen model with an end seed.

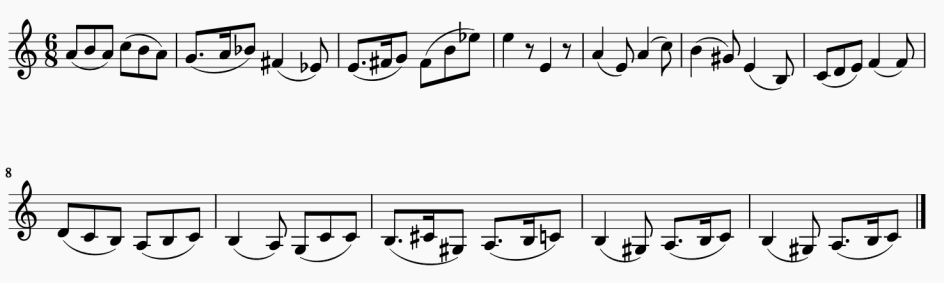


Figure 7.26: Music generated by the chosen model with an artificial seed.

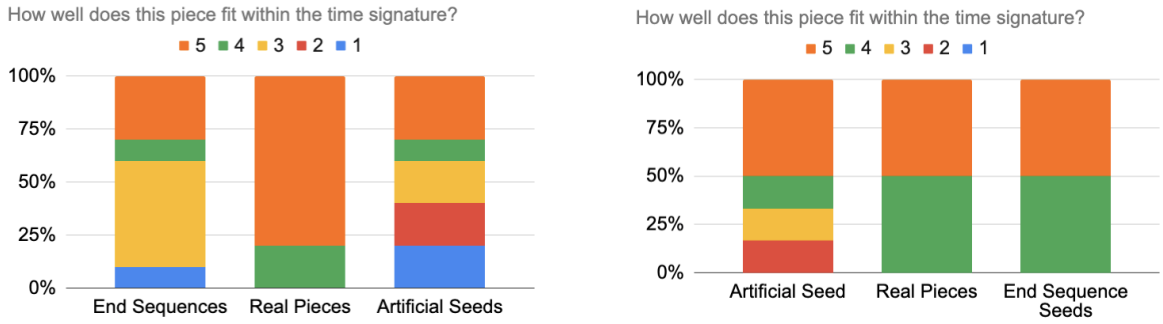


Figure 7.27: Bar charts showing the results to the question: "How well does this piece fit within the time signature?". The left chart shows the results from the chosen model generating only notes; the right chart shows the results for the chosen model generating notes and slurs.

The human evaluation for this model was done with three evaluators; the full results of this evaluation are available in the appendix (see Tables 9, 10 and 11). The results show a large increase in the suitability of artificial-seed-generated music for ABRSM grade 5 violin sight-reading (see Figure 7.28). This could be attributed to the addition of slurring to the input seed, allowing the network to have a better idea of the rhythms and phrasing of the piece.

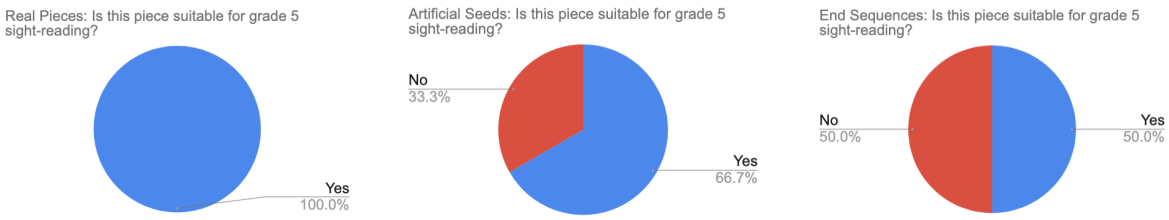


Figure 7.28: Pie charts showing the percentages of pieces suitable for ABRSM grade 5 violin sight-reading reported by test subjects, separated by seeding method. Blue represents suitable. Red represents unsuitable.

7.3.2 With More Features

The hyperparameter tuning of models containing features beyond slurs was streamlined through the use of a single set of changing parameters applied to all cases. The mandate for this came from the necessity to train as few models as possible, due to the high computation cost and time constraints of the project. All models were chosen to contain three layers of 512 LSTM units with a dropout of 0.3 and were trained with the Adam optimiser. The changing parameters were the multi-label loss function (used for slurs, spanner objects, text expressions and articulations) and a boolean variable referred to as "last layer inputs". The use of focal loss was considered (as opposed to cross-entropy loss) for multi-label features due to the increased sparsity of data, especially in text expressions and note articulations. The last layer inputs variable references the input of all extra musical features only before the final LSTM layer of the network, to give the network time to process the notes by themselves. Figure 7.29 illustrates this. The full grid search results for each added feature are displayed in Tables 3, 4 and 5 in the appendix.

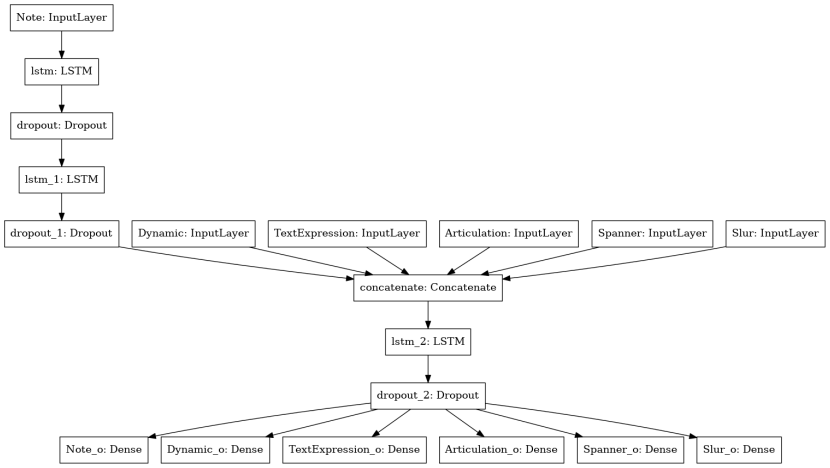


Figure 7.29: Model diagram showing extra musical features being input to the network before the final layer.

The best performing network for each instance of selected features used the binary cross-entropy loss function for multi-label features and had the last layer inputs flag set to false. The binary focal

loss function, whilst still converging well for most features, caused inconsistent updates to the network resulting in a lower performance for note prediction. The graphs in Figure 7.30 compare the relative accuracies of a model trained with cross-entropy loss and a model trained with focal loss. These graphs show the extent of the effect the focal loss function has on the training of the notes.

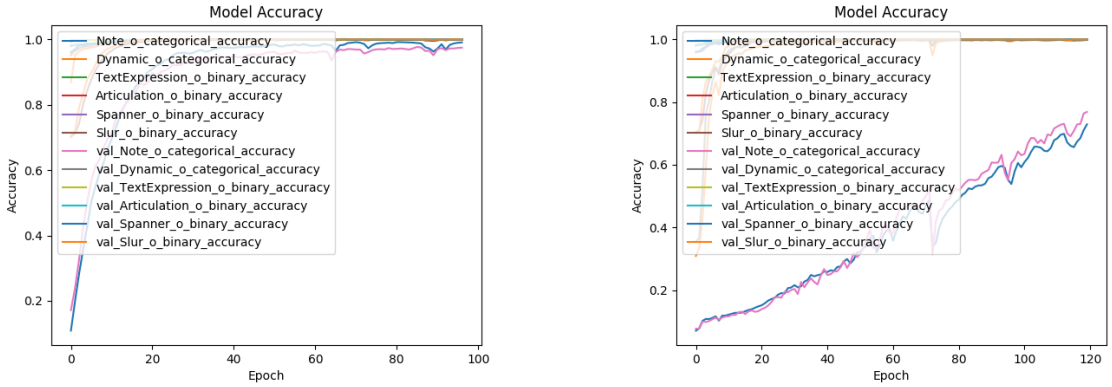


Figure 7.30: Graphs showing the relationship between the accuracy of each musical feature against the number of epochs the network is trained for. The left graph contains the accuracy history of the chosen model trained with all musical features. The right graph contains the accuracy history of an identical model trained with binary focal loss as the multi-label feature loss function.

After the grid search containing notes, dynamics and slurs was complete, it was decided that only the weights saved at the minimum validation loss state of the network were required to be retained. This is due to the regularisation effect that adding more features seemed to have on the network, allowing it to train for longer without showing performance degradation when tested on validation data. All models were trained for 120 epochs with early stopping, where the minimum validation loss was detected. The results for each of the additional features shows resounding accuracy when predicting certain features. Figures 7.31 and 7.32 show 100% accuracy on every class. This is a clear red flag, since the network should not be able to predict something relatively subjective with 100% accuracy. It is clear that the network is overfitting in some way; however, this overfitting is not being detected through the use of validation data. The cause of this overfitting can be traced back to actions taken during the data augmentation step in the pre-processing. Section 6.2.1 describes the necessity to copy the feature inputs and outputs to match the various transpositions of note inputs and outputs, since the features cannot be transposed or augmented in the same way. The reason for this overfitting is conjectured to be because the network is memorising the relative positions of features from those in the training dataset and is then tested on the exact same input/output combinations during testing and validation. The fundamental principle of using a validation and test dataset is that the network should not be exposed to any data in either, for training purposes. This is violated with the combination of the data augmentation method used and the inclusion of extra musical features. This overfitting does not affect the network training on solely notes, since there are no identical note input/output combinations. However, it would affect all other musical features, as the overfitting becomes more prevalent with the addition of more features.

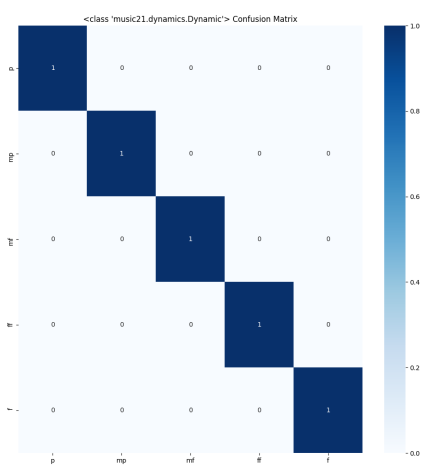


Figure 7.31: Confusion matrix for the discrete dynamics generated from a network taking into account notes, slurs and dynamics.

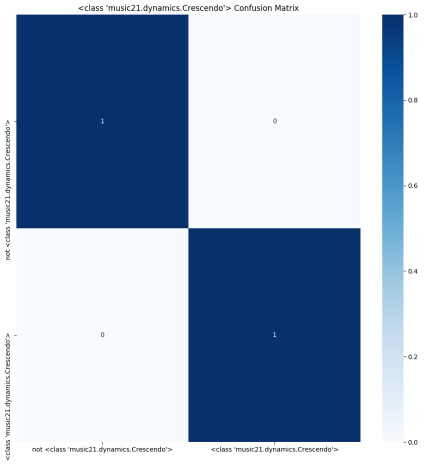


Figure 7.32: Confusion matrix for the crescendo objects generated from a network taking into account notes, slurs, dynamics and spanner objects.

To mitigate this overfitting further, the shuffling step performed before splitting the data into training and test data can be removed. This reduces the overlap between the train features and test features, since they are essentially split by pieces. The effect this has on the network is that the overfitting becomes very obvious from an early stage within the training process. With the same networks defined for this stage, all additions of features show overfitting before the 20th epoch of training - not enough time for the network to learn to produce good quality music. The only network trained with an unshuffled dataset showing any resemblance to its equivalent trained on shuffled data, is the network containing notes and slurs. For this reason, only the overfit networks were considered for the rest of the evaluation as they have been trained on a more uniformly distributed and varied dataset.

The use of the last layer inputs flag decreases the level to which the learning of notes is affected by the overfitting of the other features. This is because the notes are given a number of LSTM layers to learn their own dependencies, with no intervention from other features. Despite the networks with this property performing worse when tested on test data (see Tables 3, 4 and 5 in the appendix), it is conjectured that the dependencies learned are more appropriate since they are more reliant on the notes rather than the overfit extra features. Nevertheless, this is not a perfect solution, and further research would need to be undertaken to solve this problem.

The resultant music generated from the uniformly structured model contains some well-placed features; however, more instances of strangely-placed features occur in the generated music. Figure 7.33 shows an example of this, where the text expression “molto cresc.” appears three times in five bars - two of these instances are within a couple of timesteps. This is relatable to the result of overfitting explained in section 4: “[overfitting] results in poor performance when tested on unknown data”. The inputs to the network during the sliding window generation are completely unknown to the network.

This overfitting also affects the quality of the notes produced; bar 10 of Figure 7.33 contains a dotted-quaver pattern not specified by the ABRSM violin grade 5 sight-reading specification [3]. Another example of the odd placement of features can be seen in Figure 7.34, where there are four dynamic changes in half a bar. This could be attributed to the seeding method, since the artificial seeding method of dynamics is relatively rudimentary. Additionally, in the first note of bar 8, a down bow articulation is placed on the same note as a pizzicato (pluck the string) - these actions are mutually exclusive. The general quality of this piece is poor and is fairly representative of the quality of music in the entire generated dataset of this model.

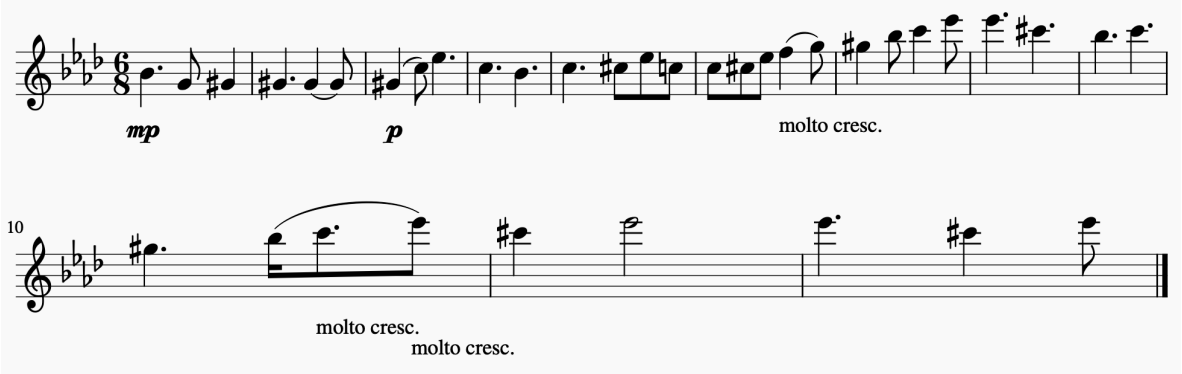


Figure 7.33: Music generated by a model trained with all musical features, using an end seed.

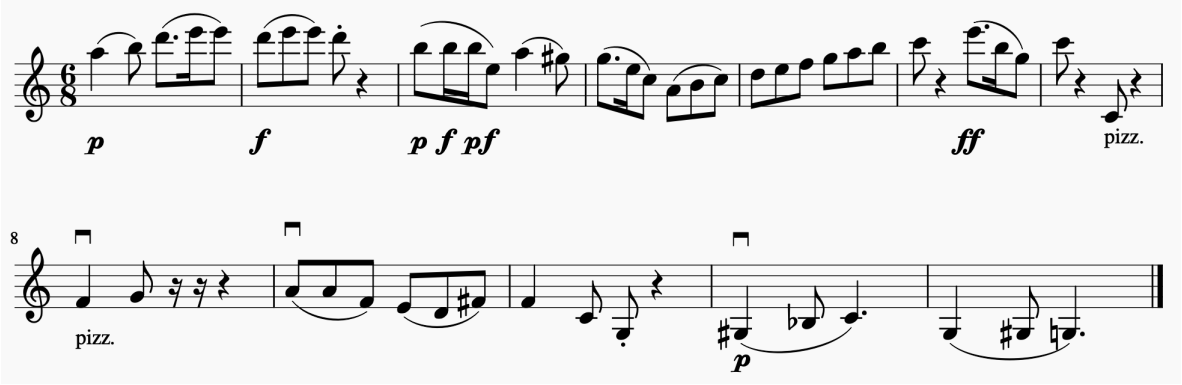


Figure 7.34: Music generated by a model trained with all musical features, using an artificial seed.

A final human evaluation with the network obtaining the highest test accuracy found for notes, slurs and dynamics included three participants. The results of this human evaluation showed similar reported suitability of the music to the previous human evaluation with notes and slurs; these results can be seen in Figure 7.35. An additional question was presented to the evaluators: "How well do the dynamics fit with the piece?", the results of which can be seen in Figure 7.36 with both the artificial-seed-music and end-seed-music performing passably.

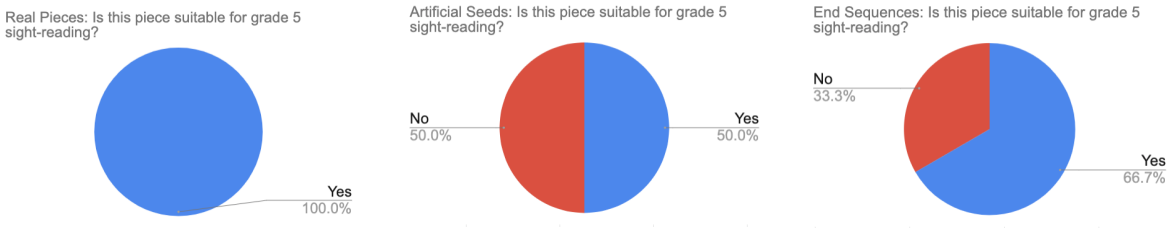


Figure 7.35: Pie charts showing the percentages of pieces suitable for ABRSM grade 5 violin sight-reading reported by test subjects, separated by seeding method. Blue represents suitable. Red represents unsuitable.

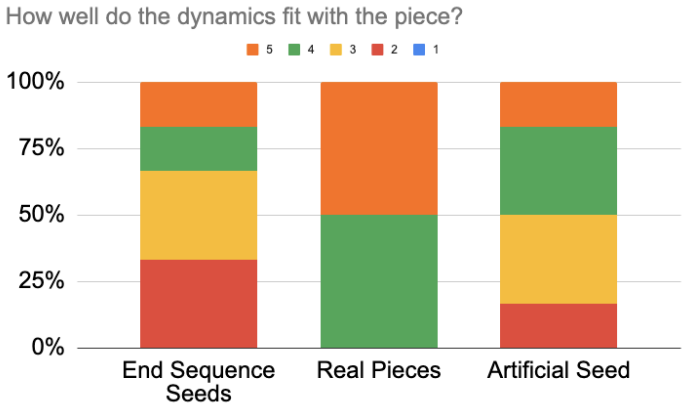


Figure 7.36: Bar chart showing the results to the question: "How well do the dynamics fit with the piece?", separated by the seeding method.

7.4 Results Summary

From a post-project perspective, the results for note-only and note-slur models, with data augmentation, show the best results for generating appropriate sight-reading music. The end seeding method was shown to perform well in both of these model types. The artificial seed method, whilst underperforming with the note-only network, has been proven to perform well with the note-slur network. The artificial seed method is also more sustainable and far less limited than the end seed method, so it would be more suitable for an application of the system. An additional draw-back of the end seed method is the tendency of music to slow down towards the end of the piece. Repeatedly generating from end seeds using the replacement strategy described in section 4.3.1, would almost certainly show degradation in the quality of music. This would be visible in the dataset visualisation metrics, with fewer unique pitches per piece on average, and a higher representation of the longer note lengths in the note length frequency analysis.

The addition of more musical features unearths the substantial issue of the model only memorising the training dataset rather than learning different feature dependencies. This could be attributed to the subjective nature of music and the limited availability of very sparse data to train on. The inclusion of the last layer inputs feature, whilst potentially reducing the impact the feature overfitting has on the

notes, does not actually address the issue. For this reason, it would be naive to call this an appropriate solution. The suitability of musical features for the network to include have been summarised and ranked in the list below.

1. **Notes and slurs**: The results of the experimentation with networks containing just notes or notes and slurs produce the most suitable results.

2. **Dynamics and spanner objects**: Adding dynamics and spanner objects to a network reduces its ability to generalise over the training data. The resultant music is either not varied enough or contains patterns seen in the original data with the occasional violation of music theory.

3. **Note articulations and text expressions:** The problems that arise with the addition of dynamics and spanner objects are amplified with the addition of articulations and text expressions. Additionally, the data augmentation technique used does not complement the inclusion of articulations and text expressions. This is because some articulations and text expressions are reliant on the exact pitches of notes, which are changed during the data augmentation stage. Harmonics and fingerings are examples of these features. The transposition of pitches in the training data causes the network to generate music containing the occasional unintuitive fingering and impossible harmonic.

Discussion

This section discusses the success of this project with reference to other projects. The first subsection draws the comparison between this project and projects focussed on generating music using neural networks. The second subsection discusses the use of the system produced, in comparison to other sight-reading practice tools.

8.1 Generating Music

The results show that LSTM neural networks are capable of generating music suitable for graded sight-reading; this contradicts the results of existing projects discussed in section 2.2 which were unable to fulfil their musical expectations. A proposed reason for this is the reduction in the complexity of music for the network to learn - from Bach fugues [16] and Led Zeppelin ballads [15], to 8 bars of monophonic music. A criticism of past projects could be their attempts to replicate some of the greatest music ever written, with a simple feed-forward neural network. The complexity of the music used for training in these projects far exceeds that of sight-reading music, which can essentially be represented as a sequence of numbers. The increased complexity requires an increase in the number of inputs and outputs to the network, thus creating sparser data and therefore, a more challenging classification problem. Furthermore, these projects find issues with the network's ability to create interesting music for a sustained period of time - this issue is particularly prevalent in Weel [15] and Docevski et al. [17]. This is another issue that does not appear in the results of this project, most likely due to the length of the pieces required being so short. From these observations, the conjecture could be made that in order to generate good music from a simple feed-forward LSTM network, the training data needs to be limited to a small pitch range and basic melodies. A simple change that could be made to these projects is limiting the inputs to a single octave; this would reduce the number of possible classes to 12 (excluding enharmonic equivalence). Despite the fact that it would sacrifice the ability of the network to generate music containing a bassline and a higher-pitched melody, the network would likely see an improvement in its understanding of the relationship between notes in a key. Similar results could be produced through the use of learned embeddings.

8.2 Practice Tool Comparison

The system as it stands consists of a number of scripts capable of preparing data, training a model, and generating pieces of music. The requirements for running the system are a dataset of sample pieces and a JSON file containing discrete requirements set by the examination board. In order to produce music, the user would have to acquire a dataset of practice pieces, scan them, convert them into a digital format, and correct any errors. Following this, there may be the requirement to train a couple of models to see which produces the best music. This is obviously quite impractical and would require the user to have some understanding of programming and neural networks. In comparison, the practice tools mentioned in section 2.1 are able to be used as they are, by any user with minimal technical proficiency. The "Sight-Reading Trainer" app provides a suitable tool for practising sight-reading music for piano. This product is not a direct competitor for the system produced since it only exists for piano, an instrument this project does not account for. The other tool mentioned, "Sight Reading Factory" [13], does contain sight-reading music for a variety of instruments. The user is able to select a number of features to specify the possible contents of the pieces they are given - which would be akin to the set of requirements specified in the JSON input file. The main limitation of this application is the inability to specify a grade or examination board. Despite this, it is the most appropriate tool for practising sight-reading. That is not to say the system produced by this project is useless, since it could be applied in a very similar context. If an examination board were to supply a large dataset of sight-reading pieces, larger datasets of generated pieces could be produced in all instruments and grades. This would require further work on the system as a whole, most importantly solving the issue of how to include extra musical features. Additionally, any nuances of instruments not accounted for in the current system would need to be highlighted and addressed.

Future Work

9.1 Adaptations and Extensions To The Current System

There are many possible areas of experimentation and improvement to the current system that could be implemented. The system, in its current state, has only been tested with one instrument and one grade; this could be expanded by obtaining more datasets and running more evaluations. Some other suggestions of adaptations are listed below.

1. Address the limitation of the system being unable to account for triplet notes. This could involve implementing the additional note length input described in section 4.3.3.
2. Introduce polyphony to the network so that sight-reading music containing more than one note at a time can be generated. This would be required to generate piano sight-reading music.
3. More experimentation could be performed on the integration of extra musical features in the system. An example of this could be looking into mapping the more sparse features onto generated music, rather than generating them sequentially in the same network. Alternatively, having the notes output before the extra features are input could also show improvements, whilst still maintaining a single network.
4. Provide the network with some idea of where in the piece it currently is so that it can produce music with a start, middle and end. A method for introducing piece ends could be generating each piece to the maximum allowed length and then shortening it to the bar containing the most appropriate ending (i.e. when the melody forms a perfect cadence).

9.2 Additions to The System and Similar Ideas

One idea that was floated before the start of the project was to produce a system for classifying music, rather than generating it. This could include scouring through a database of classical pieces and picking out extracts which are deemed suitable for a specific instrument and grade. Another suggestion, mentioned as a potential extension to the project, was to produce a user interface for displaying generated sight-reading music. This could be useful if the system was used as part of a sight-reading music tool. The student could enter their instrument and grade, and the interface would display an appropriate piece.

Conclusion

10.1 Meeting The System Requirements

The system meets all of the functional requirements stated as a "Must" in section 4.1.1. The one "Should" requirement states that "The system should take into account musical features besides just notes."; this is met with the addition of slurring, however, the addition of more musical features does not produce satisfactory results. The "Could" requirements were deemed to be out of scope due to the time constraints of the project, so were not attempted. "The system must produce coherent melodies, suitable for a piece of sight-reading music" is listed as a "Must" non-functional requirement. The feedback from the human evaluation verifies that the system is capable of producing appropriate sight-reading music. The other non-functional requirement references the system's ability to generate music faster than a human composer. On the compute nodes used for the evaluation of the system [55], 200 pieces could be generated in a couple of hours. An exact figure cannot be given since the time taken depends on the current load of the compute node and the size of the network. Nevertheless, this is certainly faster than a person could write 200 pieces of music. On a laptop or personal computer, the generation time could be a lot longer; however, no effort would have to be applied by the user.

10.2 Project Conclusion

The project provides a system capable of generating graded sight-reading melodies at the appropriate level. This is shown in the 50% suitability rate of note-only pieces generated using end seeds, deemed by the human evaluation. The system was extended to include extra musical features, and produced some good results with the addition of slurring. The human evaluation of the note-slur model showed that 50% of end-seed-generated music and 67% of artificial-seed-generated music was suitable for ABRSM grade 5 violin sight-reading. The time constraints of the project meant that the final sprint was left somewhat open-ended with the uncomprehensive evaluation of the addition of extra musical features. The system could certainly be improved through further evaluation, not only with the addition of extra musical features, but also with different musical instruments and grades. The results of this project, along with further work, could be used to produce a functioning sight-reading generation system. This could be used as a learning tool for students taking graded sight-reading examinations.

10.3 Author's Reflection

This project consisted of technical achievement in the field of music generation using neural networks. Complex issues regarding music representation, neural network structure and comprehensive system evaluation were addressed. I have learned so much from this project, coming from a situation of knowing nothing about neural networks, to being able to design and implement a system with a neural network at the heart of it. Neural networks aside, I have experienced an abundance of personal development in time management and resilience, and am in debt to the university for this opportunity to develop my academic ability.

References

[1] *Cambridge Dictionary: English Dictionary, Translations Thesaurus*. 2020. URL: dictionary.cambridge.org.

[2] *Sight-reading*. 2020. URL: gb.abrsm.org/en/our-exams/what-is-a-graded-music-exam/sight-reading/.

[3] ABRSM. *Bowed Strings Syllabus Graded Initial-8 (2020-2023)*. 2019. URL: gb.abrsm.org/media/62975/violin%5C_2020%5C_complete%5C_syllabus.pdf.

[4] *Supporting tests*. 2020. URL: trinitycollege.com/qualifications/music/grade-exams/about/supporting-tests.

[5] *About us*. 2020. URL: gb.abrsm.org/en/about-us/.

[6] *Violin Specimen Sight-Reading Tests ABRSM Grades 1-5 from 2012*. ABRSM (Publishing) Ltd, 2011.

[7] Lorraine Liyanage. *ABRSM Music Exam facts and figures!* July 2011. URL: se22pianoschool.wordpress.com/2011/07/29/abrsm-exam-facts-and-figures/.

[8] Harry Fallows. "3rd Year Project Specification." Dissertation. The University of Warwick, 2019.

[9] Harry Fallows. "3rd Year Project Progress Report." Dissertation. The University of Warwick, 2019.

[10] Neuratron Ltd. *PhotoScore First*. Version 8.8.6. Nov. 2019. URL: neuratron.com/photoscore.htm.

[11] Avid. *Sibelius*. Mar. 2020. URL: avid.com/sibelius.

[12] ABRSM. *ABRSM: Sight-Reading Trainer*. June 2017. URL: gb.abrsm.org/en/exam-support/apps-and-practice-tools/sight-reading-trainer/.

[13] Sight Reading Factory. *Sight Reading Factory*. Oct. 2013. URL: sightreadingfactory.com.

[14] Sigurur Skúli. *How to Generate Music using a LSTM Neural Network in Keras*. Dec. 2017. URL: towardsdatascience.com/how-to-generate-music-using-a-lstm-neural-network-in-keras-68786834d4c5.

[15] Joseph Weel. "RoboMozart: Generating music using LSTM networks trained per-tick on a MIDI collection with short music segments as input." Dissertation. University of Amsterdam, 2016.

[16] Nikhil Kotecha and Paul Young. "Generating Music using an LSTM Network." In: *arXiv preprint arXiv:1804.07300* (2018).

[17] Marko Docevski et al. "Towards Music Generation With Deep Learning Algorithms." In: Apr. 2018.

[18] Daniel D Johnson. "Generating polyphonic music using tied parallel networks." In: *International conference on evolutionary and biologically inspired music and art*. Springer. 2017, pp. 128–143.

[19] Olof Mogren. "C-RNN-GAN: Continuous recurrent neural networks with adversarial training." In: *arXiv preprint arXiv:1611.09904* (2016).

[20] Li-Chia Yang, Szu-Yu Chou, and Yi-Hsuan Yang. "MidiNet: A convolutional generative adversarial network for symbolic-domain music generation." In: *arXiv preprint arXiv:1703.10847* (2017).

[21] Joseph Rocca. *Understanding Generative Adversarial Networks (GANs)*. Jan. 2019. URL: towardsdatascience.com/understanding-generative-adversarial-networks-gans-cd6e4651a29.

[22] Simon Haykin. *Neural networks: a comprehensive foundation*. Prentice Hall PTR, 1994.

[23] Weiren Yu. *CS331 Neural Computing*. University of Warwick. 2020.

[24] Andrej Karpathy. *Andrej Karpathy blog*. May 2015. URL: karpathy.github.io/2015/05/21/rnn-effectiveness/.

[25] Christopher Colah. *Colah's Blog*. Aug. 2015. URL: colah.github.io/posts/2015-08-Understanding-LSTMs.

[26] Yoshua Bengio, Patrice Simard, and Paolo Frasconi. "Learning long-term dependencies with gradient descent is difficult." In: *IEEE transactions on neural networks* 5.2 (1994), pp. 157–166.

[27] Chi-Feng Wang. *The Vanishing Gradient Problem*. Jan. 2019. URL: towardsdatascience.com/the-vanishing-gradient-problem-69bf08b15484.

[28] Sepp Hochreiter and Jürgen Schmidhuber. "LSTM can solve hard long time lag problems." In: *Advances in neural information processing systems*. 1997, pp. 473–479.

[29] Michael Nguyen. *Illustrated Guide to LSTM's and GRU's: A step by step explanation*. Sept. 2018. URL: towardsdatascience.com/illustrated-guide-to-lstms-and-gru-s-a-step-by-step-explanation-44e9eb85bf21.

[30] François Chollet et al. *Keras*. keras.io. 2015.

[31] *music21: a Toolkit for Computer-Aided Musicology*. 2020. URL: web.mit.edu/music21/.

[32] Ian Sommerville. "Agile Software Development." In: *Software Engineering*. 10th ed. Pearson Education Limited, 2016, pp. 72–101.

[33] Guido Van Rossum, Barry Warsaw, and Nick Coghlan. "PEP 8: style guide for Python code." In: *Python. org* 1565 (2001).

[34] Thomas Kluyver et al. "Jupyter Notebooks-a publishing format for reproducible computational workflows." In: *ELPUB*. 2016, pp. 87–90.

[35] Ranko Lazic. *Ethical Consent for Undergraduate Projects*. URL: warwick.ac.uk/fac/sci/dcs/teaching/material/cs310/ethics.

[36] International Institute of Business Analysis. *A Guide to the Business Analysis Body of Knowledge (BABOK Guide), Version 2.0*. International Institute of Business Analysis, 2009.

[37] James Archbold. *CS261 Software Engineering*. University of Warwick. 2019.

[38] Guido Van Rossum and Fred L. Drake. *Python 3 Reference Manual*. Scotts Valley, CA: CreateSpace, 2009. ISBN: 1441412697.

[39] Wes McKinney. "Data Structures for Statistical Computing in Python." In: *Proceedings of the 9th Python in Science Conference*. Ed. by Stéfan van der Walt and Jarrod Millman. 2010, pp. 56–61. DOI: 10.25080/Majora-92bf1922-00a.

[40] Fabian Pedregosa et al. "Scikit-learn: Machine Learning in Python." In: *Journal of Machine Learning Research* 12.85 (2011), pp. 2825–2830. URL: jmlr.org/papers/v12/pedregosa11a.html.

[41] Stéfan van der Walt, S. Chris Colbert, and Gaël Varoquaux. "The NumPy Array: A Structure for Efficient Numerical Computation." In: *Computing in Science & Engineering* 13.2 (2011), pp. 22–30. DOI: 10.1109/MCSE.2011.37. eprint: aip.scitation.org/doi/pdf/10.1109/MCSE.2011.37. URL: aip.scitation.org/doi/abs/10.1109/MCSE.2011.37.

[42] Martın Abadi et al. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. Software available from tensorflow.org. 2015. URL: tensorflow.org.

[43] Jason Brownlee. *3 Ways to Encode Categorical Variables for Deep Learning*. Nov. 2019. URL: machinelearningmastery.com/how-to-prepare-categorical-data-for-deep-learning-in-python/.

[44] Will Koehrsen. *Neural Network Embeddings Explained*. Oct. 2018. URL: towardsdatascience.com/neural-network-embeddings-explained-4d028e6f0526.

[45] Victor Sanchez. *CS342 Machine Learning*. University of Warwick. 2020.

[46] Chigozie Nwankpa et al. "Activation functions: Comparison of trends in practice and research for deep learning." In: *arXiv preprint arXiv:1811.03378* (2018).

[47] Sarang Narkhede. *Understanding Confusion Matrix*. May 2018. URL: towardsdatascience.com/understanding-confusion-matrix-a9ad42dcfd62.

[48] *Google Forms - create and analyze surveys, for free.* 2020. URL: docs.google.com/forms.

[49] *Specimen Sight-Reading Tests for Violin Grades 1-5*. The Associated Board of Royal Schools of Music (Publishing) Limited, 1998.

[50] Paul Harris. *Improve your sight-reading, a workbook for examinations: violin*. Faber Music, 1993.

[51] Michael Good et al. "MusicXML: An internet-friendly format for sheet music." In: *Xml conference and expo*. 2001, pp. 03–04.

[52] *pickle - Python object serialization*. 2020. URL: docs.python.org/3/library/pickle.html.

[53] *music21.corpus - music21 documentation*. URL: web.mit.edu/music21/doc/moduleReference/moduleCorpus.html.

[54] Michiel Schuijer. *Analyzing atonal music : pitch-class set theory and its contexts*. Rochester, NY: University of Rochester Press, 2008. ISBN: 978-1-58046-270-9.

[55] *Compute Nodes*. Nov. 2019. URL: warwick.ac.uk/fac/sci/dcs/intranet/user%5C_guide/compute%5C_nodes/.

[56] Carol L Krumhansl and Mark A Schmuckler. "Key-finding in music: An algorithm based on pattern matching to tonal hierarchies." In: *19th Annual Meeting of the Society of Mathematical Psychology, Cambridge, MA*. 1986.

[57] Tsung-Yi Lin et al. "Focal loss for dense object detection." In: *Proceedings of the IEEE international conference on computer vision*. 2017, pp. 2980–2988.

[58] Umberto Griffo. *umbertogriffo/focal-loss-keras*. Apr. 2020. URL: github.com/umbertogriffo/focal-loss-keras.

[59] Maintainer Trevor L Davis. "Package 'argparse'." In: (2015).

Appendices

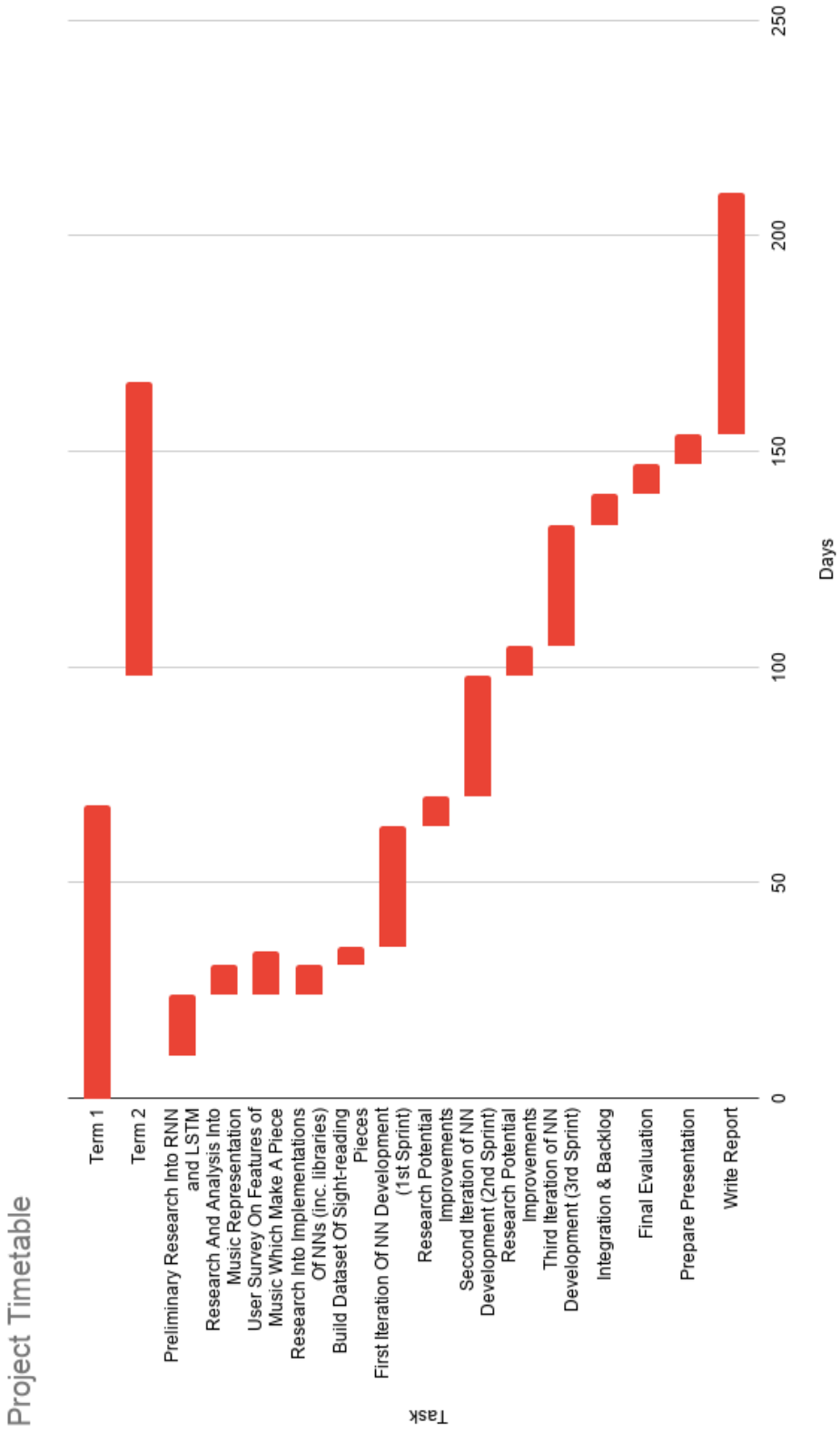


Figure 1: Project Gantt chart designed for use in the project specification [8]

lstm_units	dropout	lstm_layers	loss_function	optimiser	test_accuracy_loss	test_accuracy_val_loss
256	0.2	2	categorical_crossentropy	adam	0.67796	0.670514
256	0.3	2	categorical_crossentropy	adam	0.675354	0.670141
256	0.3	2	categorical_crossentropy	rmsprop	0.684289	0.667908
128	0.3	2	categorical_crossentropy	rmsprop	0.664929	0.665674
128	0.3	2	categorical_focal_loss	rmsprop	0.665302	0.665302
256	0.3	3	categorical_crossentropy	adam	0.664557	0.663068
256	0.3	3	categorical_focal_loss	adam	0.658972	0.662323
128	0.3	2	categorical_crossentropy	adam	0.665302	0.661579
128	0.3	2	categorical_focal_loss	adam	0.658228	0.661206
256	0.2	3	categorical_crossentropy	rmsprop	0.6586	0.661206
256	0.3	2	categorical_focal_loss	rmsprop	0.655249	0.660834
256	0.3	3	categorical_crossentropy	rmsprop	0.663812	0.659717
128	0.2	3	categorical_crossentropy	rmsprop	0.664557	0.659345
128	0.2	2	categorical_crossentropy	adam	0.658972	0.658972
128	0.2	3	categorical_crossentropy	adam	0.658972	0.658972
256	0.2	2	categorical_focal_loss	adam	0.654133	0.658972
256	0.2	2	categorical_focal_loss	rmsprop	0.655622	0.6586
128	0.2	2	categorical_focal_loss	rmsprop	0.655622	0.658228
128	0.3	3	categorical_crossentropy	adam	0.657111	0.657483
256	0.3	3	categorical_focal_loss	rmsprop	0.654505	0.657483
256	0.2	3	categorical_crossentropy	adam	0.650037	0.656739
256	0.2	2	categorical_crossentropy	rmsprop	0.662323	0.656739
256	0.2	3	categorical_focal_loss	rmsprop	0.648548	0.656366
64	0.2	2	categorical_crossentropy	rmsprop	0.656739	0.656366
256	0.3	2	categorical_focal_loss	adam	0.663068	0.655994
128	0.2	3	categorical_focal_loss	adam	0.650037	0.655249
128	0.3	3	categorical_crossentropy	rmsprop	0.650782	0.654877
128	0.3	3	categorical_focal_loss	rmsprop	0.654877	0.654505
128	0.2	2	categorical_crossentropy	rmsprop	0.656366	0.654505
128	0.2	2	categorical_focal_loss	adam	0.654133	0.654133
64	0.2	3	categorical_crossentropy	adam	0.650037	0.654133
128	0.3	3	categorical_focal_loss	adam	0.654133	0.654133
256	0.2	3	categorical_focal_loss	adam	0.661579	0.653388
64	0.3	2	categorical_crossentropy	adam	0.650782	0.652271
64	0.2	2	categorical_crossentropy	adam	0.652271	0.652271
128	0.2	3	categorical_focal_loss	rmsprop	0.652271	0.652271
64	0.2	2	categorical_focal_loss	adam	0.651526	0.651526
64	0.2	3	categorical_focal_loss	adam	0.650782	0.650782
64	0.3	2	categorical_crossentropy	rmsprop	0.650782	0.650782
64	0.3	2	categorical_focal_loss	rmsprop	0.65041	0.65041
64	0.3	2	categorical_focal_loss	adam	0.645197	0.649293
64	0.2	2	categorical_focal_loss	rmsprop	0.647431	0.648176
64	0.2	3	categorical_crossentropy	rmsprop	0.646314	0.646314
64	0.3	3	categorical_crossentropy	rmsprop	0.644453	0.644453
64	0.3	3	categorical_focal_loss	rmsprop	0.641847	0.644453
64	0.2	3	categorical_focal_loss	rmsprop	0.645197	0.644453
64	0.3	3	categorical_crossentropy	adam	0.641474	0.641474
64	0.3	3	categorical_focal_loss	adam	0.637007	0.639613

Table 1: Table displaying the results of a 48-model grid search, ordered by the accuracy of the model with weights saved at the minimum validation loss during training.

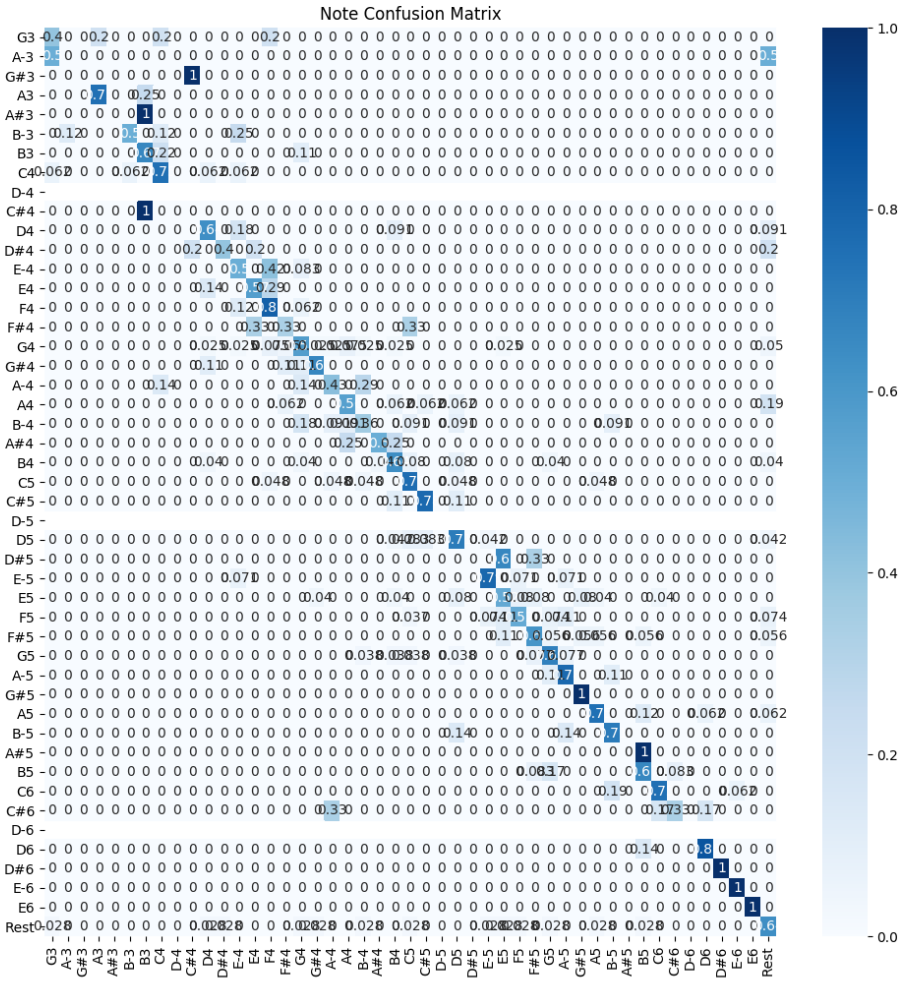


Figure 2: Confusion matrix of notes over the whole note range for the model described in section 7.1.

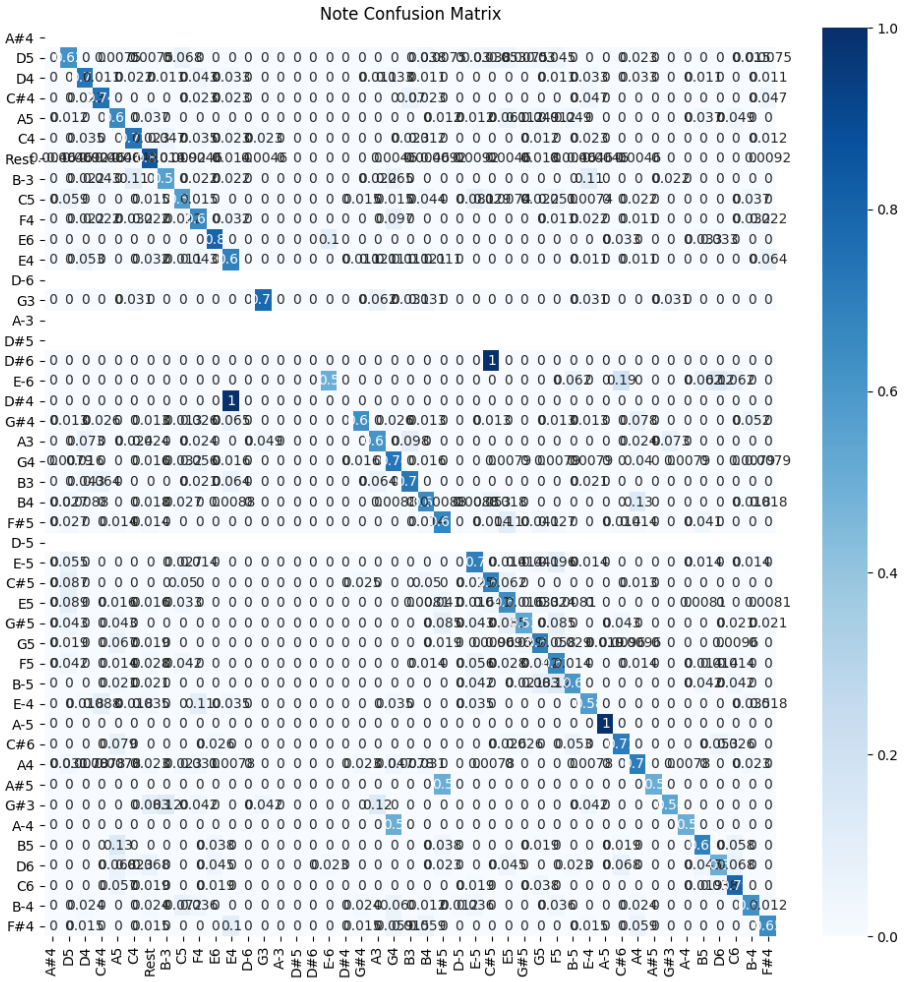


Figure 3: Confusion matrix of notes over the whole note range for the model described in section 7.2.

lstm_units	dropout	lstm_layers	single_loss_function	multi_loss_function	optimiser	test_note_accuracy_loss	test_note_accuracy_val_loss	test_slur_accuracy_loss	test_slur_accuracy_val_loss
256	0.3	3	categorical_crossentropy	binary_crossentropy	adam	0.720402	0.722636	0.948995	0.94825
512	0.3	3	categorical_crossentropy	binary_crossentropy	adam	0.746463	0.718541	0.960908	0.944527
512	0.3	3	categorical_crossentropy	binary_crossentropy	rmsprop	0.743485	0.717424	0.960164	0.94006
256	0.3	2	categorical_crossentropy	binary_crossentropy	adam	0.711467	0.712211	0.937453	0.93373
256	0.3	3	categorical_crossentropy	binary_crossentropy	rmsprop	0.711095	0.711095	0.939687	0.939687
512	0.3	2	categorical_crossentropy	binary_crossentropy	rmsprop	0.717051	0.706627	0.933358	0.926284
512	0.3	3	categorical_crossentropy	binary_crossentropy	rmsprop	0.712211	0.701042	0.943783	0.926657
256	0.3	2	categorical_crossentropy	binary_crossentropy	rmsprop	0.704393	0.701042	0.927774	0.920328
256	0.3	2	categorical_crossentropy	binary_crossentropy	adam	0.705138	0.699181	0.925168	0.922561
256	0.3	2	categorical_crossentropy	binary_crossentropy	rmsprop	0.705138	0.698436	0.923306	0.914743
512	0.3	2	categorical_crossentropy	binary_crossentropy	adam	0.698436	0.698064	0.931124	0.917722
256	0.3	3	categorical_crossentropy	binary_crossentropy	rmsprop	0.695086	0.697692	0.926657	0.919955
512	0.3	2	categorical_crossentropy	binary_crossentropy	adam	0.719657	0.696203	0.939315	0.923306
512	0.3	3	categorical_crossentropy	binary_crossentropy	adam	0.702159	0.696203	0.931497	0.924423
512	0.3	2	categorical_crossentropy	binary_crossentropy	rmsprop	0.69583	0.693224	0.92554	0.915115
256	0.3	3	categorical_crossentropy	binary_crossentropy	adam	0.697319	0.689129	0.927401	0.927774

Table 2: Table displaying the results of a grid search performed on models taking into acccount notes and slurs.

multi_loss_function	last_layer_inputs	test_Note_accuracy	test_Slur_accuracy	test_Dynamic_accuracy
binary_crossentropy	False	0.924051	0.998511	0.998883
binary_focal_loss	False	0.871929	1	0.999628
binary_focal_loss	True	0.846984	1	0.999628
binary_crossentropy	True	0.725242	0.969844	0.992926

Table 3: Grid search results for a model containing 3 layers of 512 units, with a dropout of 0.3, trained with the categorical crossentropy loss function for single-label features, and the Adam optimiser. The model takes into consideration the notes, slurs and discrete dynamics of the pieces of music in the dataset.

multi_loss_function	last_layer_inputs	test_Note_accuracy	test_Slur_accuracy	test_Dynamic_accuracy	test_Spanner_accuracy
binary_crossentropy	False	0.943038	0.999628	1	0.999255
binary_focal_loss	False	0.900596	1	1	1
binary_focal_loss	True	0.832837	1	1	1
binary_crossentropy	True	0.759494	0.98213	0.996277	0.994974

Table 4: Grid search results for a model containing 3 layers of 512 units, with a dropout of 0.3, trained with the categorical crossentropy loss function for single-label features, and the Adam optimiser. The model takes into consideration the notes, slurs, discrete dynamics and spanner objects of the pieces of music in the dataset.

multi_loss_function	last_layer_inputs	test_Note_accuracy	test_Slur_accuracy	test_Dynamic_accuracy	test_Spanner_accuracy	test_TextExpression_accuracy	test_Articulation_accuracy
binary_crossentropy	False	0.971705	0.999628	0.999628	1	0.999714	0.999894
binary_crossentropy	True	0.77513	0.991809	0.997022	0.996091	0.998969	0.998511
binary_focal_loss	False	0.762844	1	1	1	1	1
binary_focal_loss	True	0.526061	1	0.998883	1	1	1

Table 5: Grid search results for a model containing 3 layers of 512 units, with a dropout of 0.3, trained with the categorical crossentropy loss function for single-label features, and the Adam optimiser. The model takes into consideration the notes, slurs, discrete dynamics, spanner objects, note articulations and text expressions of the pieces of music in the dataset.

Evaluation graphics of the model described in section 7.2. The music generated at this stage contains notes only.

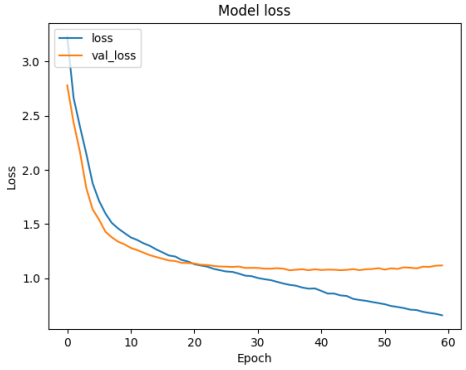


Figure 4: Model loss history from training.

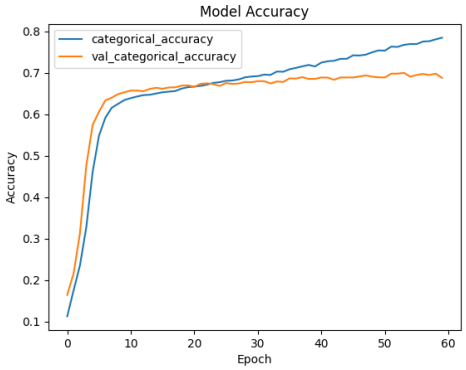


Figure 5: Model accuracy history from training.

Figure 6: Single octave confusion matrix created using the model with weights stored at the minimum loss.

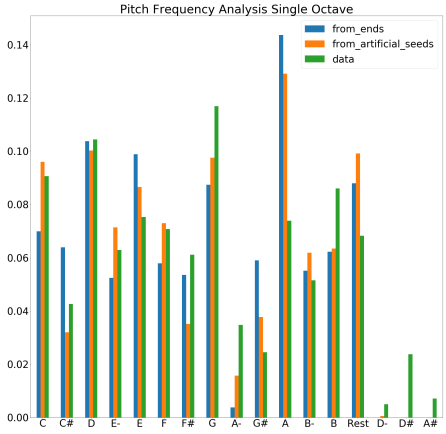


Figure 7: Graph showing the results of the pitch frequency analysis on generated datasets.

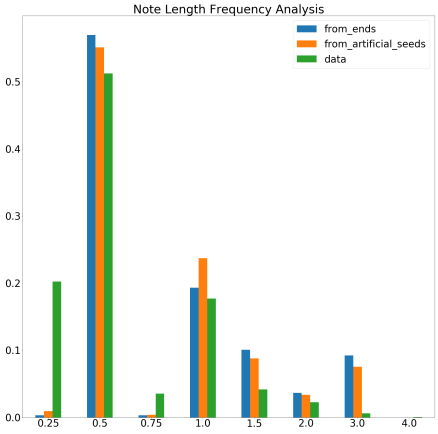


Figure 8: Graph showing the results of the note length frequency analysis on generated datasets.

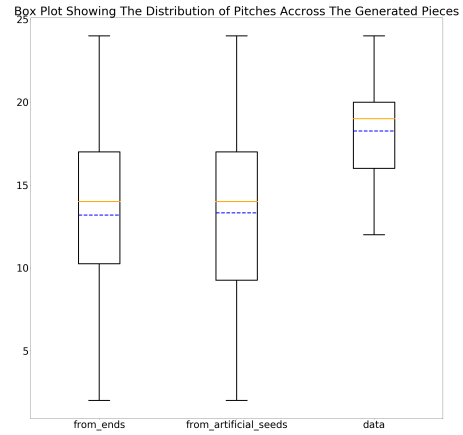


Figure 9: Boxplot showing the distribution in the number of unique pitches per piece of generated music, with different seed methods.

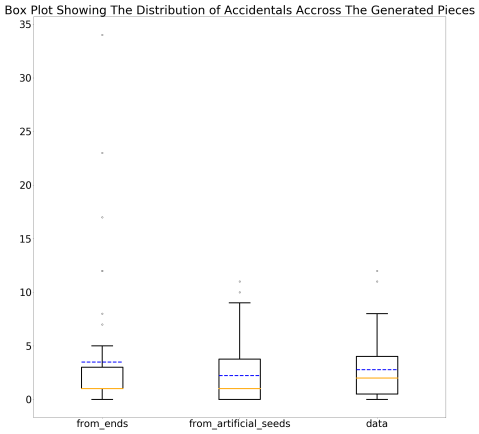


Figure 10: Boxplot showing the distribution in the number of out-of-key-signature notes per piece of generated music, with different seed methods.

Tables containing the human evaluation results of the model described in section 7.2, in accordance with the guidelines set in section 5.2. The music generated at this stage contains notes only. Each row represents a single response from a single piece of music.

How good does the melody of this piece sound?	How well does this piece fit within the time signature?	How well does this piece fit within the key signature?	How difficult is the rhythm to play?	How difficult are the pitch intervals to play?	How difficult is this piece to play in general?	Is this piece of music suitable for a grade 5 sight-reading piece?
5	5	5	2	1	1	Yes
4	5	3	3	3	3	No
2	4	2	2	3	2	No
2	3	2	4	4	4	No
2	2	2	3	4	3	Yes
2	1	3	5	5	5	No
3	5	2	2	4	4	No
1	1	2	5	4	5	No
2	2	2	4	4	5	No
1	3	2	4	2	3	No

Table 6: Results from music generated using artificial seeds.

How good does the melody of this piece sound?	How well does this piece fit within the time signature?	How well does this piece fit within the key signature?	How difficult is the rhythm to play?	How difficult are the pitch intervals to play?	How difficult is this piece to play in general?	Is this piece of music suitable for a grade 5 sight-reading piece?
3	5	4	4	3	3	Yes
4	5	5	2	3	3	Yes
4	5	5	3	2	3	Yes
4	5	4	2	2	1	Yes
5	4	5	1	1	2	No
5	5	4	3	2	2	Yes
5	5	5	3	3	3	Yes
5	5	5	3	3	4	Yes
3	5	3	2	4	3	Yes
4	4	2	3	1	3	Yes

Table 7: Results from real pieces (control dataset).

How good does the melody of this piece sound?	How well does this piece fit within the time signature?	How well does this piece fit within the key signature?	How difficult is the rhythm to play?	How difficult are the pitch intervals to play?	How difficult is this piece to play in general?	Is this piece of music suitable for a grade 5 sight-reading piece?
3	4	2	2	1	3	Yes
4	5	4	2	2	2	No
3	3	4	2	1	1	No
4	5	4	1	3	3	Yes
4	3	2	3	2	3	Yes
2	3	2	4	3	4	No
1	5	1	3	2	3	No
4	3	5	4	4	4	No
3	3	4	2	3	3	Yes
1	1	1	3	1	3	No

Table 8: Results from music generated using end seeds.

Evaluation graphics of the model described in section 7.3.1. The music generated at this stage contains notes and slurs.

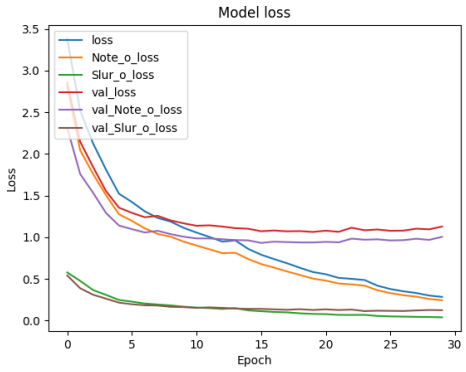


Figure 11: Model loss history from training.

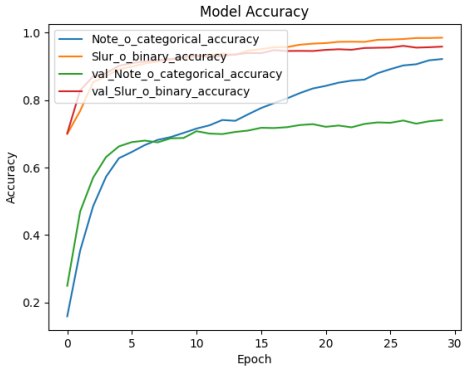


Figure 12: Model accuracy history from training.

Figure 13: Single octave confusion matrix created using the model with weights stored at the minimum validation loss.

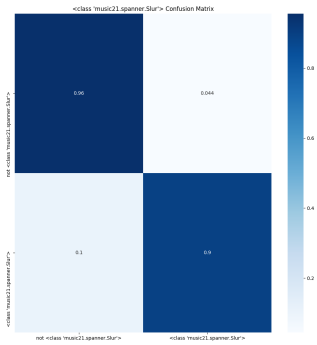


Figure 14: Slur confusion matrix created using the model with weights stored at the minimum validation loss.

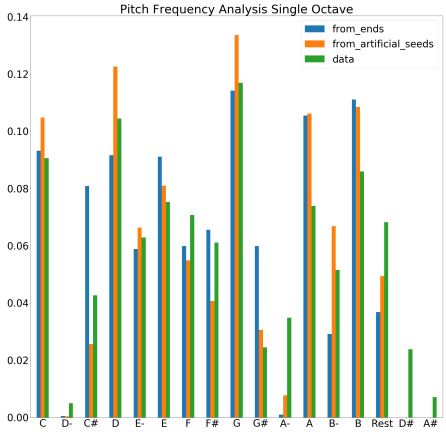


Figure 15: Graph showing the results of the pitch frequency analysis on generated datasets.

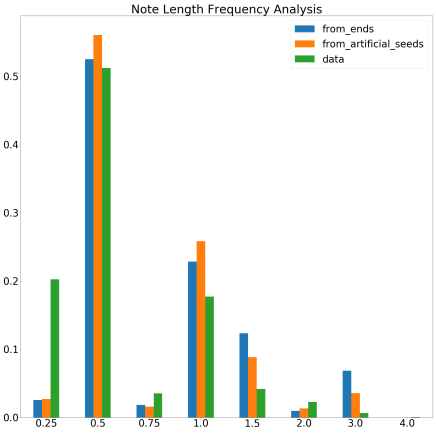


Figure 16: Graph showing the results of the note length frequency analysis on generated datasets.

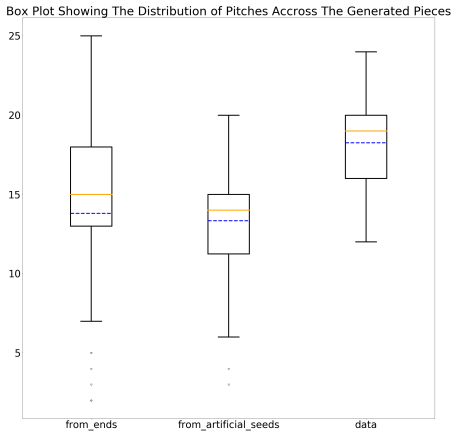


Figure 17: Boxplot showing the distribution in the number of unique pitches per piece of generated music, with different seed methods.

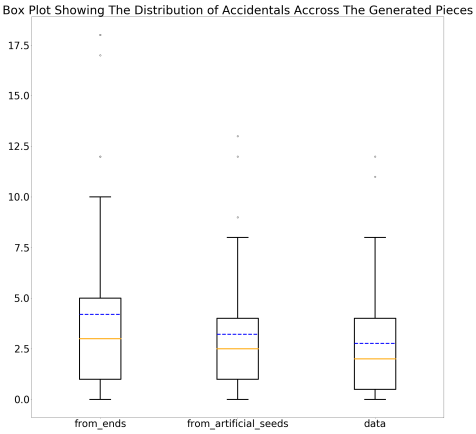


Figure 18: Boxplot showing the distribution in the number of out-of-key-signature notes per piece of generated music, with different seed methods.

Tables containing the human evaluation results of the model described in section 7.3.1, in accordance with the guidelines set in section 5.2. The music generated at this stage contains notes and slurs. Each row represents a single response from a single piece of music.

How good does the melody of this piece sound?	How well does this piece fit within the time signature?	How well does this piece fit within the key signature?	How difficult is the rhythm to play?	How difficult are the pitch intervals to play?	How difficult is this piece to play in general?	Is this piece of music suitable for a grade 5 sight-reading piece?
4	5	5	3	2	4	Yes
4	4	3	4	4	3	Yes
2	2	1	4	3	4	No
4	5	5	2	3	3	No
3	3	3	2	4	3	Yes
4	5	4	3	3	2	Yes

Table 9: Results from music generated using artificial seeds.

How good does the melody of this piece sound?	How well does this piece fit within the time signature?	How well does this piece fit within the key signature?	How difficult is the rhythm to play?	How difficult are the pitch intervals to play?	How difficult is this piece to play in general?	Is this piece of music suitable for a grade 5 sight-reading piece?
5	5	5	4	4	4	Yes
4	4	5	2	3	2	Yes
5	5	4	1	2	1	Yes
5	5	5	5	5	5	Yes
3	4	5	3	2	3	Yes
5	4	5	3	2	3	Yes

Table 10: Results from real pieces (control dataset).

How good does the melody of this piece sound?	How well does this piece fit within the time signature?	How well does this piece fit within the key signature?	How difficult is the rhythm to play?	How difficult are the pitch intervals to play?	How difficult is this piece to play in general?	Is this piece of music suitable for a grade 5 sight-reading piece?
4	5	4	3	5	5	No
3	4	3	2	4	3	Yes
2	5	1	2	2	3	No
3	4	3	3	2	3	No
3	4	4	2	2	3	Yes
4	5	4	2	2	2	Yes

Table 11: Results from music generated using end seeds.