Facultade de Informática
UNIVERSIDADE DA CORUÑA

TRABALLO FIN DE GRAO
GRAO EN ENXEÑARÍA INFORMÁTICA
MENCIÓN EN COMPUTACIÓN

Deep Learning Language Models for music analysis and generation

Estudante: Daniel Quintillán Quintillán

Dirección: Brais Cancela Barizo

Carlos Eiras Franco

A Coruña, xuño de 2022.

Abstract

In this project, we tackle the problem of predicting the next note in a monophonic musical piece. We choose a symbolic representation and extract it from digital sheet music. The problem is approached as four separate tasks, each of them corresponding to a specific property of the musical note. For each task, we compare the performance of both single and multi-output deep learning algorithms. Despite the severe class imbalance in our dataset, our models manage to generate balanced predictions for the four features.

Resumo

Neste proxecto tratamos o problema de predicir a seguinte nota nunha peza musical monofónica. Escollemos unha representación simbólica e extraémola dun conxunto de partituras dixitais. Afrontamos o problema como catro tarefas de predicción de propiedades inherentes á nota musical. Para cada tarefa, comparamos o rendemento de algoritmos de aprendizaxe profundo dunha e varias saídas. Aínda que o conxunto de datos está moi descompensado, os nosos modelos son capaces de xerar predicións equilibradas nos catro problemas.

Keywords:

- Sequence Learning
- Symbolic representation
- Language Models
- Multi-task learning
- Imbalanced classification
- Self-supervised learning
- Monophonic music
- Deep Learning

Palabras chave:

- Aprendizaxe de secuencias
- Representación simbólica
- Modelos de linguaxe
- Aprendizaxe multi-tarea
- Clasificación descompensada
- Aprendizaxe auto-supervisado
- Monofonía
- Aprendizaxe profundo

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 General context

Music has historically evolved linked to the rest of human languages, and it can even be understood as a communication system on its own. In fact, many of the abstractions used by linguists to understand verbal languages, such as grammar or vocabulary, are also used to study music.

In practice, music composition requires following the constraints imposed by a given grammar and understanding the meaning conveyed by specific patterns across an ever-evolving literature. Nowadays, this task is mostly reserved for highly specialised experts with years of training. Therefore, we aim to develop a program that learns these skills in a more reasonable time frame.

Machine Learning (ML) is devoted to the development of models that can process data to learn their underlying rules and relations. The larger the set of data, the more subtle and complex the patterns. Thus, models need to achieve a higher level of generalisation. This is commonly achieved by combining several layers of non-linear models in an approach called Deep Learning (DL). Recent breakthroughs in DL have shown impressive results in the field of Natural Language Processing (NLP), such as models capable of engaging in human-like conversations [6] or generating images from text descriptions [7].

Given the similarities between music and verbal language, it comes as no surprise that approaches used for NLP, such as Transformer networks or Recurrent Neural Networks (RNN)s, also work well in music [8, 9]. We will take advantage of this by working on a new approach to the problem of musical modelling.

1.2 Problem to solve

This project will tackle the problem of building a language model that can predict the next note in a musical sequence. This task requires a model that can learn both the grammatical rules of the musical language and the semantic relations between different notes.

1.3 Existing solutions

Several projects have already worked on this task, implementing one of two paradigms: end-to-end or symbolic.

- **End-to-end systems** [5] work with raw audio, enabling the model to learn patterns of high complexity such as the frequency spectra of different instruments. However, this approach requires large amounts of data and computational resources.
- **Symbolic systems** [10] use musical notation as a proxy for aural data. NLP models are also symbolic, since they work with words instead of samples of human speech, so a symbolic approach to music generation is similar to a text-based language model. However, a symbolic representation requires a stage of feature engineering, where the researcher must decide which sets of symbols are more suitable to represent the problem.

1.4 Proposal

We propose the use of several state-of-the-art Deep Learning architectures to obtain models that complete symbolic music sequences. Given a set of sequences that represent parts of songs, the model will be trained to predict the next note of each sequence, This approach, known as self-supervised learning, is common practice in the context of sequence models. To achieve good results in its prediction, the model must be capable of extracting "grammatical" and "semantic" relations between the symbols in the sequence.

1.5 Main goals

The main goal of this project is to compare different deep architectures and configurations in their ability to predict the next note of a monophonic tune (a piece of music where only one note is played at a time). Further elaborations on this idea, such as generating novel musical pieces of variable length, lie beyond the scope of this project.

1.6 Report structure

The overall structure of the rest of the document is as follows:

- **Musical rudiments** (chapter 2): Fundamental concepts of music needed to understand the problem of music modelling.
- **Machine Learning** (chapter 3): Fundamental concepts behind the Machine Learning techniques that have been applied in this project.
- **Project Management** (chapter 4): Project planning, costs and implementation details.
- **Problem representation** (chapter 5): Problem formalisation and chosen representation.
- **Related Work** (chapter 6): Analysis of the existing solutions to this problem, their advantages and shortcomings.
- **Dataset** (chapter 7): Analysis of the raw dataset and the different preprocessing pipelines.
- **Methodologies** (chapter 8): Techniques, architectures and approaches used in the learning process.
- **Evaluation** (chapter 9): Discussion of results and error analysis.
- **Conclusions** (chapter 10): Summary, takeaways and future work.

Chapter 2

Music rudiments

Music is the art of arranging sounds in time. Just as verbal languages evolved complex grammars and vocabularies, music has developed different rule sets depending on the region and artistic movement. Indeed, music is said to be driven by systems that resemble the grammar and syntax of verbal languages, even if music rules are not as strict [11, 12]. This chapter will cover basic concepts of music composition and how they are represented in different formats.

2.1 Elements of music

Since music involves the production of sound, studying its fundamental components means understanding the cognitive processes involved in sound perception. Due to the scope of this project, we will address two of them: pitch and duration.

2.1.1 Pitch

Pitch is a perceptual ranking of sound based on its dominant (loudest) frequency [13]. The different intervals between pitch values are fundamental building blocks of music composition. For this project, we are focusing on octaves and semitones.

An octave is an interval whose highest value doubles the frequency of its lowest. This interval is interesting because the human perception of music is robust to octave changes. For example, if we learn the melody of a song and shift every note by one or more octaves, our brain still recognises the patterns of the original song [14]. For this reason, pitch values separated by whole octaves have the same name and are said to share a pitch class.

Octaves can, in turn, be divided into smaller intervals. The standard system for western music, the chromatic scale [15], defines 12 pitch values (also known as pitch classes) per octave, with the intervals between them being called semitones. The exact frequencies of each value depend on the tuning system.

There are several pitch naming systems based on octaves and pitches. In English letter notation, seven of the 12 pitch classes in the chromatic scale are named after their equivalents in the diatonic scale: C, D, E, F, G, A and B (*do*,*re*, *mi*, *fa*, *sol*, *la*, *si* in Spanish). As seen in Table 2.1, some of these values are separated by a whole tone, so the remaining pitch classes are named by appending a suffix (accidental) to shift the pitch of the diatonic value by a semitone.

English name	**Spanish name**
C	*Do*
C#	*Do #*
D	*Re*
D#	*Re #*
E	*Mi*
F	*Fa*
F#	*Fa #*
G	*Sol*
G#	*Sol #*
A	*La*
A#	*La #*
B	*Si*

Table 2.1: Name of the 12 notes in the chromatic scale in both English and Spanish.

To denote the chromatic scale over multiple octaves, the Scientific Pitch Notation (SPN) system [16] names pitch values by combining their pitch class with a number identifying its octave. This number starts from -1 and can increase indefinitely, but frequencies beyond the tenth octave are inaudible for humans. For example, the third pitch class of the third octave is written as D_1.

2.1.2 Duration

Note duration refers, as one would expect, to the length of time that a sound is played [17]. This feature is the key component of rhythm and, for unpitched percussion instruments, the most important pattern to convey both structure and meaning. The basic unit of duration is the beat, and the way instruments emphasise the different beats in a sequence is part of the defining features of specific music genres.

2.2 Western sheet music

Western sheet music is a symbolic representation system that treats sounds as a sequence of discrete events. It is written on staves, which are sets of five parallel lines with four spaces (Figure 2.1).

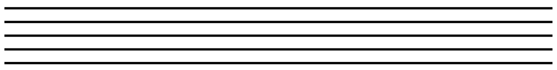


Figure 2.1: A stave.

Western music notation is based around the note, a symbol which conveys both pitch and duration [18]. This section will explore how pitch and duration are represented in written music.

2.2.1 Representation of pitch

In sheet music, pitch is written as a relative scale. The basic pitch reference for a given piece is determined by a clef, a symbol that must be written on the left margin of every stave. The type of clef determines the base pitch at the line on which it is written (Table 2.2).

Clef	**Name**	**Note**
𝄞	G-clef	G_4
𝄡	C-clef	C_4
𝄢	F-clef	F_3

Table 2.2: Clefs and their corresponding pitches.

Once the clef establishes a basic pitch, the pitch of each note depends on the height at which it is drawn with respect to the clef. The pitch interval applied by changing from a line to its neighbouring space depends, once again, on which scale was chosen to write the piece.

There are also special notes called rests that represent the lack of sound. Different symbols are used to denote rests depending on their duration.

2.2.2 Representation of duration

In sheet music, a tempo marking is usually written at the beginning of the document as a reference of pace. For example, (𝅝 = 180) means that a semibreve must last a $\frac{1}{180}$th of a minute. The duration of the rest of the symbols is separated by powers of two (Table 2.3).

Symbol	British	American	Spanish
𝅝	Semibreve	Whole	*Redonda*
𝅗𝅥	Minim	Half	*Blanca*
♩	Crotchet	Quarter	*Negra*
♪	Quaver	Eighth	*Corchea*
𝅘𝅥𝅯	Semiquaver	Sixteenth	*Semicorchea*
𝅘𝅥𝅰	Demisemiquaver	Thirty-second	*Fusa*
𝅘𝅥𝅱	Hemidemisemiquaver	Sixty-fourth	*Semifusa*

Table 2.3: Representation and names of different durations.

In addition, notes can be dotted to denote a duration multiplier depending on the number of dots. A single dot has a duration multiplier of $\frac{3}{2}$.

2.2.3 Other elements

Since metric structures are not relevant to our project, they will not be discussed here. However, we will explain two notational artefacts that are a direct consequence of said structure: slurs and ties. As seen in Figure 2.2, these elements are curved lines that connect sequences of adjacent notes.

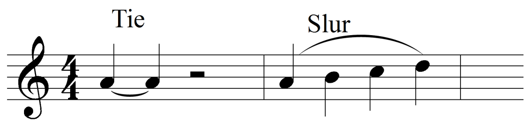


Figure 2.2: A slur and a tie on a pentagram [1].

Although they are written in the same way, they have different implications.

- A slur connects two or more notes of different pitch and indicates that they must be played without separation.
- A tie, on the other hand, indicates that the duration of two notes of the same pitch must be combined into a longer note.

2.3 Digital music files: MIDI and Guitar Pro

When it comes to digital representations of music, a multitude of systems and standards have been developed to meet the needs of music professionals. This section will discuss two of them: MIDI and Guitar Pro.

2.3.1 MIDI

Musical Instrument Digital Interface (MIDI) is a technical standard used for recording, editing and playing music [19]. It supports up to 128 different sounds that can be adjusted to specific pitches, as well as a separate channel for unpitched instruments. MIDI encodes frequency as a note number n, a real value from 0 to 127. Each unit in this scale represents a semitone, while decimal values allow for the representation of microtones (intervals smaller than a semitone). The formula that connects note number n and frequency f is:

$$f = 440 \cdot 2^{(n-69)/12} \tag{2.1}$$

2.3.2 Guitar Pro

Guitar Pro is a multi-track editor of musical scores. It can store, show and play musical information in standard stave format, tablatures or MIDI. Since it supports stave notation, it doesn't support microtones, so it only works with the integer part of the MIDI note number.

Chapter 3

Machine Learning and Neural Networks

3.1 Artificial Neural Networks and Deep Learning

As mentioned before, this project addresses the problem of predicting the next note of a musical composition. Our proposal for this task is based on Artificial Neural Networks (ANN)s. An ANN is a set of connected nodes that mimics the synaptic process of animal brains. Each node (neuron) performs the following computation on the input X:

$$\begin{aligned} z &= W^T X + b \\ \hat{y} &= g(z), \end{aligned} \tag{3.1}$$

where W and b are trainable parameters, g is an arbitrary non-linear activation function and $\hat{y}$ is the output of the node [20]. These models can adjust the weights and biases of each node to replicate the patterns of a given dataset. The simplest ANN architecture is the Feedforward (FFWD) network, where each node only processes information from its previous layer and feeds it to the next [21]. If there are several layers of nodes between the input and the output of the network (Figure 3.1), it is called a deep network. Deep Learning (DL) is the process of training deep networks, an approach that has revolutionised the field of ML by allowing researchers to model increasingly complex functions [22].

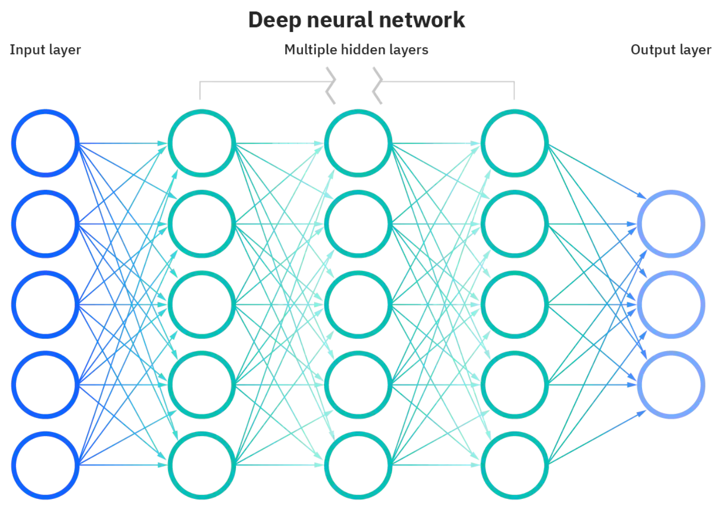


Figure 3.1: Simplified diagram of a deep FFWD neural network [2].

One of the most popular approaches for DL is Supervised Learning (SL). The training process of this approach involves processing a training set of input-output examples. In this project, SL is applied to a multi-class classification problem. This section will introduce this concept and then discuss the most interesting stages of the learning process, namely activation functions, loss functions and optimisation.

3.1.1 Multi-class classification

In a multi-class classification problem, a model is tasked to classify an example into one of three or more classes. In an ANN, this requires using a Softmax function as the activation function of the output layer.

3.1.2 Activation

Choosing the right activation function is key in DL models. The activation function at the output depends on the nature of the problem, whereas the activation functions of the hidden layers affect the speed of the learning process. In this subsection, three activation functions will be discussed: Softmax, ReLU and SeLU.

Softmax

A Softmax function [23] is a generalisation in multiple dimensions of the most common output function for binary classification, logistic regression. It is named *soft* in contrast to the $argmax$ function, which assigns a “hard” binary value indicating whether a position holds the maximum value. Instead, if the size of the output layer of the network matches K (the number of classes of the problem) the function $g_{softmax}$ for a specific element z_i returns

$$g_{softmax}(z_i) = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}}. \tag{3.2}$$

Essentially, the Softmax function returns a probability distribution over K possible classes. Hence, it is quite popular for DL applications.

ReLU

The ReLU activation function [24] is defined as

$$g_{ReLU}(z) = max(0, X) \tag{3.3}$$

Since it sets all negative inputs to zero, it can push multiple neurons to become unresponsive to all inputs, which is known as the dead ReLU problem [24].

SeLU

The Scaled Exponential Linear Unit (SELU) activation function [25] is proposed as a solution to the dead ReLU problem [26]. It is defined as follows:

$$\alpha \approx 1.6733 \tag{3.4}$$

$$\lambda \approx 1.0507 \tag{3.5}$$

$$g_{SeLU}(z) = \begin{cases} \lambda & x \geq 0 \\ \lambda\, \alpha(\exp(x) - 1) & x < 0 \end{cases} \tag{3.6}$$

Interestingly, this activation function has a self-normalising effect, meaning that it pushes values towards a distribution with zero mean and unit variance. For this reason, it is used to improve the performance of FFWD networks.

3.1.3 Loss and cost function

The loss function of a model returns a quantitative estimation of the difference between its own prediction $\hat{y}$ and the expected value y for a single example. For a multi-class classification problem, the most common loss function is Categorical Crossentropy (CCE), which takes two probability distributions y and $\hat{y}$ of K elements each and returns

$$\begin{aligned} \underset{K\times 1}{y} &= [0 \,\ldots\, 0\,1\,0 \ldots\, 0]; \quad \sum_{i=0}^{K} y = 1 \\ \underset{K\times 1}{\hat{y}} &= [0.12 \,\ldots\, 0.49 \,\ldots\, 0.09]; \quad \sum_{i=0}^{K} \hat{y} = 1 \\ L_{cce}(y, \hat{y}) &= -\sum_{i=1}^{K} y_i \, \log \hat{y}_i \end{aligned} \tag{3.7}$$

The function only activates at the position where y is active, the expected output. If $\hat{y}$ assigns a low probability to that class, its logarithm will be a large negative number, so the loss will be large. Conversely, if the expected output has a high probability in y, the logarithm will be close to zero.

The average of the loss function across the m examples of the dataset,

$$J = \frac{1}{m} * \sum_{i=1}^{m} L(\hat{y}^{(i)}, y^{(i)}), \tag{3.8}$$

is often called cost. The cost value of a network is used to adjust its parameters as part of the learning process.

3.1.4 Optimisation: Backpropagation and gradient descent

Finding the set of parameters that allow the network to adjust to training data can be posed as a minimisation problem, namely finding W and b that minimise the cost function J. This problem can be approached as an iterative process where each iteration (*epoch*) requires two stages: backpropagation and gradient descent.

The backpropagation algorithm computes the gradient of the cost function with respect to each of the weights and biases in the network [27]. Then, the gradient descent algorithm [28] updates parameters W and b as

$$\begin{aligned} W &:= W - \alpha \, \frac{\partial J(W, b)}{\partial w} \\ b &:= b - \alpha \, \frac{\partial J(W, b)}{\partial b}. \end{aligned} \tag{3.9}$$

The update is performed in the opposite direction to the gradients $\frac{\partial J(w,b)}{\partial w}$ and $\frac{\partial J(w,b)}{\partial b}$ so as to minimise the cost function. The magnitude of the update is determined by the learning rate

α. This algorithm has been improved by more complex variations that will be discussed in the next section.

3.2 Optimisation algorithms

In this section, we will discuss the different versions of the gradient descent optimisation algorithm that have been used in this project.

3.2.1 Mini-batch and Stochastic Gradient Descent

When dealing with large datasets, processing them entirely before updating the weights is quite costly. Instead, mini-batch gradient descent [29] groups examples into batches and updates the parameters of the network every time a whole batch is processed. The choice of mini-batch size affects the speed of the learning process.

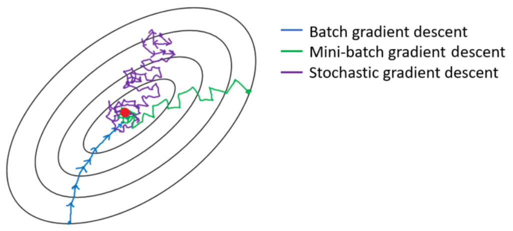


Figure 3.2: Simplified illustration of how mini-batch size affects the learning process [3].

- If it is set to the number of examples, the algorithm is called Batch Gradient Descent (BGD). Each iteration will be slow, but few will be needed to reach the local minimum.
- If it is set to 1, the algorithm is also called Stochastic Gradient Descent (SGD). Each iteration of this algorithm is way faster than BGD, but its progress towards a local minimum is noisier.
- If the mini-batch size is between 1 and the number of examples, the learning process can strike a balance between the speed of each iteration and the number of iterations needed to reach a local minimum.

3.2.2 RMSProp

During the minimisation of the cost function, the parameters are likely to oscillate on unwanted axes, thus hindering the learning process and preventing us from using a higher learning rate. The Root Mean Square Propagation (RMSProp) algorithm [30] solves this by adapting the learning rate for each trainable parameter according to an Exponentially Weighted

Moving Average (EWMA) of its gradients. First, the running average is computed as

$$\begin{aligned}\gamma &= 0.9 \\ v &:= \gamma\, v + (1-\gamma)\left(\frac{\partial J(w)}{\partial w}\right)^2,\end{aligned} \tag{3.10}$$

where γ is the forgetting factor that controls how much recent values affect the average, usually set to 0.9. Then, the parameters are updated as

$$\begin{aligned}\epsilon &= 10^{-7} \\ w &:= w - \alpha\frac{\frac{\partial J(w)}{\partial w}}{\sqrt{v}+\epsilon},\end{aligned} \tag{3.11}$$

where ϵ is a constant added for numerical stability. This approach dampens oscillations and modifies parameters in the general direction of the minimum value, effectively reducing the number of iterations needed for convergence.

3.2.3 ADAM

Adaptive Moment Estimation (ADAM) [31] is an update to the RMSProp algorithm that uses running averages of both the gradients and its second moments. First, the running averages are computed as

$$\begin{aligned}\beta_1 &= 0.9 \\ \beta_2 &= 0.999 \\ m &:= \beta_1 m + (1-\beta_1)\frac{\partial J(w)}{\partial w} \\ v &:= \beta_2 v + (1-\beta_2)\left(\frac{\partial J(w)}{\partial w}\right)^2,\end{aligned} \tag{3.12}$$

where m and v are the EWMAs of the gradients and their second moments, respectively. β_1 and β_2 are the corresponding forgetting factors. Since both m and v are initialised to zero and the forgetting factors are close to 1, the running average is biased towards zero. This problem can be solved by applying the following bias correction:

$$\begin{aligned}\hat{m} &= m\frac{1}{\beta_1^t} \\ \hat{v} &= v\frac{1}{\beta_2^t},\end{aligned} \tag{3.13}$$

where t is the current iteration. Finally, the weight update

$$w := w - \alpha \left(\frac{\hat{m}}{\sqrt{\hat{v}} + \epsilon} \right) \tag{3.14}$$

takes both $\hat{m}$ and $\hat{v}$ into account.

3.3 Hyperparameter tuning

Several techniques can be applied to tune the learning process of a model. This section will explain layer normalisation, dropout and residual connections.

3.3.1 Layer normalisation

This technique [32] normalises the inputs of all the neurons of a specific layer for each example. Given an input mini-batch X of size m where each item contains K elements, the first step of normalisation is to compute the means and variances of a specific input i as

$$\begin{aligned} \mu_i &= \frac{1}{K} \sum_{k=1} K x_{i,k} \\ \sigma_i^2 &= \frac{1}{K} \sum_{k=1} K (x_{i,k} - \mu_i)^2 \end{aligned} \tag{3.15}$$

Then, each sample is transformed to have zero mean and unit variance.

$$\hat{x_{i,k}} = \frac{x_{i,k} - \mu_i}{\sqrt{\sigma_i^2 + \epsilon}} \tag{3.16}$$

A small constant ϵ is added to the denominator for numerical stability. A final transformation guided by two trainable parameters ρ and β allows the data to be normalised to have any mean and variance.

$$y_i = \rho \hat{x_i} + \beta \tag{3.17}$$

Normalising the input to a layer changes the shape of its parameter space and accelerates the learning process.

3.3.2 Dropout

When the performance of a model is significantly lower on the validation set, it is said to be overfitting. This problem can be addressed by limiting the complexity of the model in a process called regularisation. Dropout [33] is a regularisation technique that randomly sets the activation values of neurons to zero during training. Reducing the activation frequency of a neuron limits the complexity of the patterns that it can learn, which in turn reduces the fitting capacity of the whole model.

3.3.3 Residual connections

A residual connection [34] links two nodes of non-consecutive layers. Including them in very deep models is a simple way to get better performance without increasing their computational cost.

3.4 Language Models

Since music can be studied as a language, we can apply techniques from the field of NLP, such as language models, to build our music prediction model. A Language Model (LM) [35] is a probability distribution over a sequence of words. Typically, ANN-based language models are trained as probabilistic classifiers that learn to predict the conditional distribution of a word $w^{<t>}$ given a fixed size window of previous words, namely

$$P(w^{<t>}|w^{<t-1>}\dots,w^{<t-m>}), \tag{3.18}$$

where m is the length of the input window.

We will now introduce the most common alternatives to encode words: one-hot encoding and embeddings.

One-hot encoding

Using One-Hot Encoding (OHE) [36] requires knowing how many different words exist in the training set. Given a vocabulary size K, each word is then represented as a binary-valued vector of size K where every number is set to zero except for one. The index of the active bit is the unique identifier of the word.

Word embeddings

Word embeddings [37] represent each word as a real-valued vector meant to convey its meaning. Words that are close in this vector space are expected to be similar in meaning. The vector representations can be learned by training a model to complete text fragments [38, 39]. Embeddings are often pre-trained on large amounts of easy-to-access data and then fine-tuned to task-specific sets in a process called Transfer Learning [40].

3.5 Sequence Models

Sequence models can be applied in problems where either the input, the output or both are sequences, such as speech recognition, sentiment classification or music generation. The structure of FFWD networks poses several inherent limitations when it comes to modelling sequence data, namely:

- Lack of support for variable-length inputs and outputs.
- The standard network architecture processes each word separately, so it cannot share information learnt at different text positions.
- The input and output layers of a regular ANN need to be of size $n * m$, where n is the size of a single word and m is the length of the sequence. This can be a problem as projects scale in vocabulary size and sequence length.

This section will start by explaining how recurrent neural networks address these problems, to then introduce the LSTM architecture.

3.5.1 Recurrent Neural Networks

Recurrent Neural Networks (RNN)s [41] are a class of ANN that processes sequences one word at a time. The term recurrent is due to the fact that a hidden memory cell shares information that was learnt from processing the previous words of the sequence. To build a deep RNN, the activation vector of a given timestep can be used as the input of another LSTM network.

However, in very deep neural networks, the gradient has a hard time propagating to the first layers. As information traverses many nodes of the network, it can progressively approach zero (vanish) or infinity (explode). In sequence learning, vanishing gradients translate to local influences, where the output of a specific position of the sequence is strongly influenced by its neighbouring inputs but the model has a hard time capturing long-range dependencies.

3.5.2 LSTM

Even though “vanilla” RNNs can theoretically keep track of arbitrarily long dependencies, having a computation graph with too many transformations may cause vanishing or exploding gradients. Long Short-Term Memory (LSTM) networks [42] allow gradients to flow unchanged through the network, thus avoiding the vanishing gradient problem. However, they are still vulnerable to the exploding gradient problem.

Structure

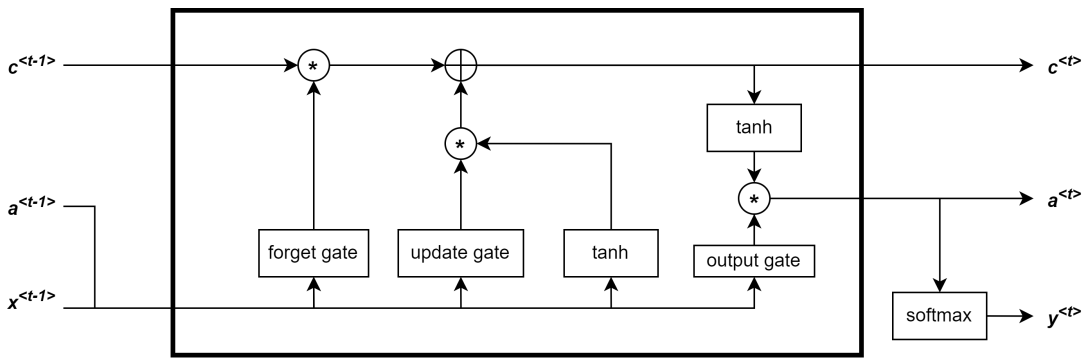


Figure 3.3: Structure of the LSTM network.

At any given time step, the input of the network $x^{<t>}$ and the previous activation value $a^{<t-1>}$ are combined. This value is used to compute a memory cell candidate (D) and three trainable gates: the update gate Γ_u, the forget gate Γ_f and the output gate Γ_o as

$$\begin{aligned} d^{<t>} &= tanh(W_c[a^{<t-1>}, x^t] + b_c]) \\ \Gamma_u &= \sigma(W_u[a^{<t-1>}, x^{<t>}] + b_u) \\ \Gamma_f &= \sigma(W_f[a^{<t-1>}, x^{<t>}] + b_f) \\ \Gamma_o &= \sigma(W_o[a^{<t-1>}, x^{<t>}] + b_o). \end{aligned} \tag{3.19}$$

In this equation, each of the W and b variables is a different weight or bias matrix, so the memory cell and each gate are trained separately. σ is the sigmoid activation function. The gates will control the information of the memory cell before it is passed as an activation value to the next timestep (Figure 3.3).

The memory cell of the current timestamp $c^{<t>}$ combines the current memory cell candidate and the previous memory cell $c^{<t-1>}$. Γ_f discards irrelevant information from the previous memory cell and Γ_u controls which information from the current candidate is relevant enough to update the current one.

$$c^{<t>} = \Gamma_f \, c^{<t-1>} + \Gamma_u \, d^{<t>} \tag{3.20}$$

Finally, the activation vector is computed by applying the output gate Γ_o to the memory cell.

$$a^{<t>} = \Gamma_o \, tanh(c^{<t>}) \tag{3.21}$$

Given an appropriate weighting of the three gates, this architecture allows the contents of the memory cell to flow unchanged through every timestep of the sequence, thus preventing the problem of vanishing gradients.

3.6 Transformers

Even though recurrent models can be modified to prevent the vanishing gradient problem, all RNN architectures have an inherent limitation: to process a word at a given position, every word up to that position must have been processed before. Consequently, capturing long-range dependencies requires transferring information through many transformations. This can cause problems such as vanishing or exploding gradients. The Transformer architecture [43] relies on attention mechanisms to shorten this path and capture arbitrarily long dependencies. This section will explore the building blocks of a transformer network: self-attention and multi-headed attention.

3.6.1 Self-attention

The self-attention technique computes three trainable attributes for each word $x^{<i>}$: Q, K and V as

$$\begin{aligned} q^{<i>} &= W_Q\, x^{<i>} \\ k^{<i>} &= W_K\, x^{<i>} \\ v^{<i>} &= W_V\, x^{<i>}. \end{aligned} \tag{3.22}$$

In this equation, W_Q, W_K and W_V are trainable weight matrices. Then, the attention vectors matrix A is computed as

$$A(Q, K, V) = softmax\left(\frac{QK^T}{\sqrt{d_K}}\right) V \tag{3.23}$$

In this equation, Q, K and V represent the three attributes of the whole sequence: query, key and value. In addition, d_K is the dimension of each key attribute. We will now explain the construction of the attention matrix step by step.

1. The dot product of the query and key attributes gives us an idea of how much a specific word of the input sequence is related to the rest.

2. This value is scaled down to prevent dot product explosion.

3. A softmax function is applied to normalise dot product scores between zero and one.

4. The result is then multiplied by the value matrix.

The result of this operation is that the representation of each word is influenced by the meaning of its neighbours. Whereas word embeddings are context-invariant, the attention vector of the same word can be different depending on its context.

The values of the attention representation can be influenced by the initialisation of the weight matrices W_Q, W_K and W_V. To address this, multiple "heads" of self-attention can be run simultaneously and aggregated into a single representation. A multi-head attention model is the main building block of the transformer encoder.

3.6.2 Transformer encoder

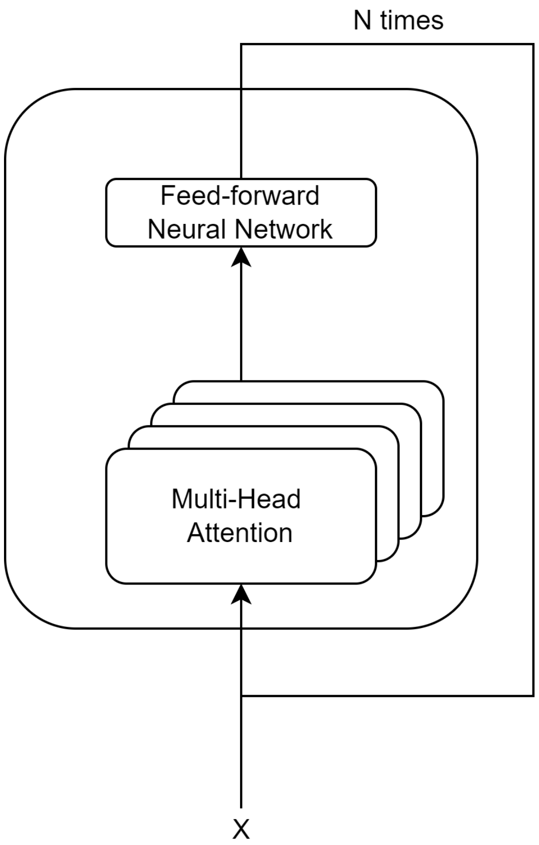


Figure 3.4: Transformer encoder block.

A transformer encoder, as seen in Figure 3.4 is built by appending a FFWD network to a multi-head attention model. This block is run multiple times to fine-tune the encoded output. Our project uses an encoder-only implementation of the Transformer network, so the decoder will not be discussed.

Chapter 4

Project Management

4.1 Planning

As with most ML projects, this one starts with a simple approach and progressively complicates the problem in successive iterations, each of them having roughly the same structure (Figure 4.1). In each iteration, the dataset has to be built before running and evaluating multiple experiments. When the results are deemed sufficient, the next iteration starts.

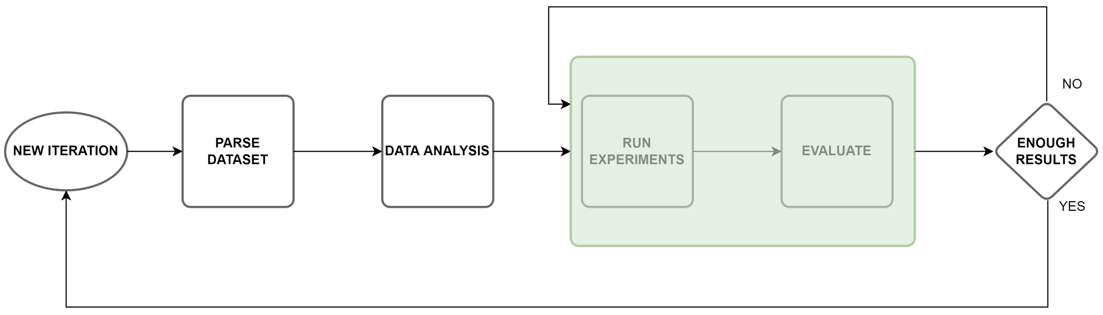


Figure 4.1: Detailed view of an iteration.

The project was intended to be divided into three iterations across the span of four months, as seen in Figure 4.2.

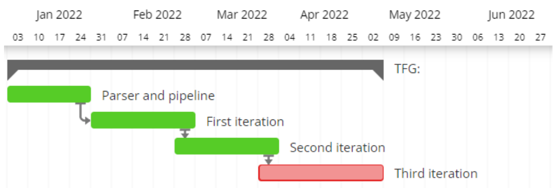


Figure 4.2: Planned tasks of the project on a Gantt diagram.

However, the change from the first to the second iteration required a re-implementation of the whole pipeline, so the last iteration had to be discarded (Figure 4.3).

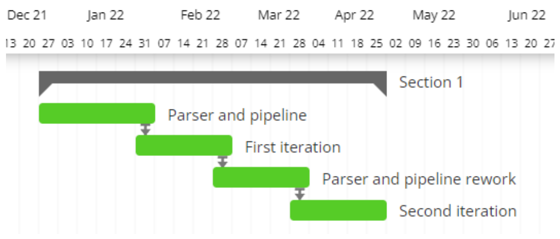


Figure 4.3: Actual tasks of the project on a Gantt diagram.

4.2 Tools and libraries

The main tools and libraries used in this project are listed below.

- The **Python** programming language is used to implement an experimentation pipeline that can support our workflow. It is chosen for its high level of abstraction and many scientific libraries.
- **Tensorflow** is an open-source library for ML projects. The forward propagation of a Tensorflow model is a sequence of symbolic operations that must be compiled before training. This allows the library to check for inconsistencies, optimise the steps and infer the back-propagation.
- **Keras** is another open-source library that works as an interface between Python and Tensorflow. Keras contains implementations for most of the building blocks of any ML project, such as loss functions, optimisers or ANN architectures.
- **Tensorboard** is a data visualisation toolkit that can be used for experiment tracking in Tensorflow projects. It is used to log the evaluation metrics at the end of every epoch and then plot multiple graphs on our web browser.

4.3 Costs

Three people have contributed their time to this project: a junior developer **J1** and two postdoc researchers, **T1** and **T2**. In Spain, the average hourly salary for these roles [44] is 10€ for junior developers and 17€ for post-doc researchers. **J1** worked at a rate of four hours per day, while **T1** and **T2** engaged in weekly meetings to assess the progress of the project. On the whole, the cost of the project is estimated at 6,267€ (Table 4.1).

Resource	Total hours	Cost
J1	560	5,600 €
T1	20	333 €
T2	20	333 €
Total	600	6,267 €

Table 4.1: Detailed costs of the project.

4.4 Implementation details

A swift transition from idea to implementation is key in ML projects. To support multiple architectures and configurations training in parallel, a highly configurable benchmarking pipeline has been implemented [45] following the principles of the strategy pattern [46]. This tool reads preprocessing and model settings from a configuration file and determines the conditions of each experiment. Several instances of the script can be executed in parallel to train multiple models at a time.

Chapter 5
Problem

This project tackles the problem of building a musical language model. Essentially, a language model is a probability distribution over a sequence of semantic units called tokens. A neural network can learn to model such sequences by trying to predict the probability of a token w_t given a specific context, such as a fixed-size window of the k previous tokens.

$$P(w^{<t>}|w^{<t-k>}, ..., w^{<t-1>}) \tag{5.1}$$

The choice of semantic unit is very domain-specific, but it usually relies on abstractions that are already used in linguistics, like words or characters. However, the concept of word does not exist in music, so the first question to address is which unit of language can be used as a token in a musical language model. This chapter will start by defining that unit and discussing which musical features are relevant to the problem. Then, the final formalisation of the problem will be discussed.

5.1 Tokenisation unit

Sheet music uses the note as a discrete unit that conveys both pitch and duration. Notes can be understood as the basic unit of musical text, so they will be used as tokens for this project. However, it is worth noting that note-level tokenisation restricts the scope of the problem to monophonic sequences, which means that only one note can be played at a time. Consequently, the harmony achieved by chords or playing different instruments at the same time cannot be captured by the model.

5.2 Note features

Finding the most appropriate formalisation for note modelling requires exploring several alternatives for representing pitch and duration.

5.2.1 Pitch

Even though sheet music uses a relative pitch notation, this serves no other purpose than increasing readability. Consequently, this representation is discarded in favour of absolute pitch scales.

As discussed in the second chapter, the MIDI standard uses a formula to convert any frequency into a real value between 0 and 127. However, Guitar Pro files only work with the integer part of that value, so microtones are not supported. This discrete representation will be used as a proxy for pitch.

Taking all this into account, there are several ways to represent pitch in the context of a language model. For example, we could adapt pitch prediction into a regression problem. This would involve distributing its discrete values on a continuous linear scale with an infinite range. To discard predictions outside of the Guitar Pro pitch range, the prediction value would need to be truncated to its nearest integer and bounded between 0 and 127. However, this approach is counterproductive. Music intervals are governed by complex rules that do not fit a monotonic linear scale. Thus, being wrong by one semitone is not necessarily better than being wrong by two.

The other alternative is treating pitch modelling as a classification problem, which involves mapping each of the 128 pitch values to a discrete class. This approach makes no assumptions about the relative order of values before training, so it is a better choice for our domain. Moreover, it allows for rests to be represented as a possible pitch value without causing any scale-related problems.

As previously discussed, pitch classes and octaves convey different qualities of sound. Consequently, pitch modelling will be treated as two separate classification problems: octave ($\mathcal{O}$) and pitch class ($\mathcal{PC}$) modelling. Rests will be represented as a separate category in both of these problems.

$$\mathcal{PC} = \{C, C\#, D, D\#, E, F, F\#, G, G\#, A, A\#, B, REST\} \tag{5.2}$$

$$\mathcal{O} = \{-1, 0, 1, 2, 3, 4, 5, 6, 7, 8, REST\} \tag{5.3}$$

5.2.2 Duration

The problem of duration modelling will be simplified in three major ways.

- We will keep duration as a relative fraction of the beat instead of inferring its absolute value from the tempo marking. As a result, the model will have no concept of how tempo affects music composition.

- Even though notes can have an arbitrary number of dots, using more than one is quite uncommon. Therefore, we will assume that any dotted note only has one dot.
- Ties will be ignored. If two notes share a tie, they will remain separate and their duration values will not be added together.

In a similar way to how octaves and pitch classes convey different things, so do both base duration and dotting. As a result, they will also be treated as separate problems. Base duration ($\mathcal{BD}$) will be treated as a classification problem. The reasoning is the same as with pitch: a crochet instead of a semibreve is not necessarily more "correct" than a minim, so a regression problem is not appropriate for this domain.

$$\begin{aligned}\mathcal{BD} = \{&semibreve, minim, crotchet, quaver, semiquaver,\\ &demisemiquaver, hemidemisemiquaver\}\end{aligned} \tag{5.4}$$

Since the dot ($\mathcal{D}$) is reduced to only two possible values (present or absent) it will be treated as a binary classification problem.

$$\mathcal{D} = \{True, False\} \tag{5.5}$$

5.3 Problem formalisation

This section started with the premise of building a language model for notes. However, after analysing the characteristics of sound that a note conveys, the problem will be tackled as four separate classification problems:

$$\begin{aligned}&P(pc^{<t>}|c^{<t-k>}, ..., c^{<t-1>})\\ &P(o^{<t>}|c^{<t-k>}, ..., c^{<t-1>})\\ &P(bd^{<t>}|c^{<t-k>}, ..., c^{<t-1>})\\ &P(d^{<t>}|c^{<t-k>}, ..., c^{<t-1>}),\end{aligned} \tag{5.6}$$

where C is a note representation that combines multiple of these four variables. The different representations of notes will be discussed in chapter 7.

Chapter 6

Related work

This chapter will explore the state-of-the-art in the fields of NLP and music generation, as well as their challenges and limitations.

6.1 Natural Language Processing

Recent advances in the field of NLP are driven by scale in both parameter count and dataset size.

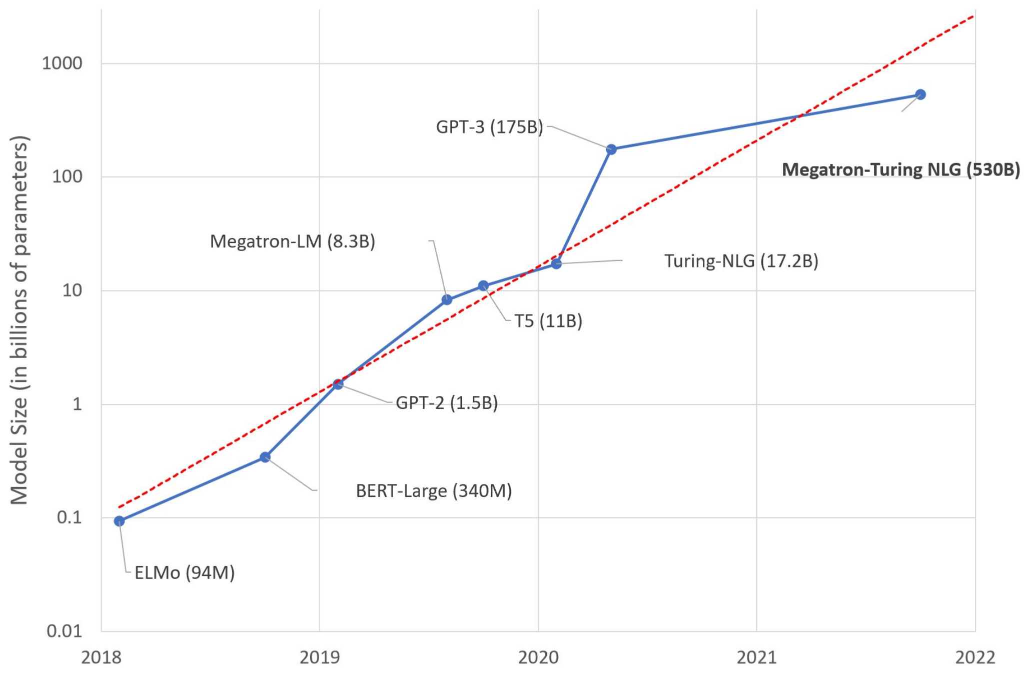


Figure 6.1: Parameter count of state-of-the-art language models over time, on a logarithmic scale [4].

As Figure 6.1 shows, the parameter count of language models presents an exponential increase over the last four years. Large Language Models (LLM)s are pre-trained on large unlabelled text corpora, and can later be fine-tuned to task-specific datasets. This large scale enables such generalisation in LLMs that they have shown great performance in few, one and even zero-shot learning [47]. An example of this paradigm is the Language Models for Dialog Aplication (LaMDA) model [48], which uses a decoder-only architecture to support a conversational language model. This architecture reaches 137B parameters and was trained on multiple datasets with a total of $1.56T$ words. Training such a large model requires advanced parallelisation techniques and, even then, the model had to be trained for 57.7 days. Since our dataset and computational resources are limited, adapting models of this scale to our project would be unfeasible.

6.2 Music generation

In music, repetition is used on multiple time scales to convey structure or meaning. Consequently, musical models must be able to effectively capture long-term dependencies. The Transformer [43] architecture is used in several music generation projects for its ability to maintain long-range coherence, but it requires some modifications depending on the data representation.

Since music can be transcribed and studied as a written language, NLP techniques have been applied to encode and learn music as text. However, some projects have approached music modelling as an end-to-end audio generation problem. This section will discuss both approaches.

6.2.1 End-to-end music systems

Using raw audio as input data allows end-to-end systems to capture the timbres and dynamics of multiple instruments at the same time. The Jukebox [5] model achieves this by using a cascading encoder-decoder architecture. First, the raw audio is processed by three separate VQVAE models [49]. Each autoencoder returns a discrete latent representation of the input with a different temporal resolution.

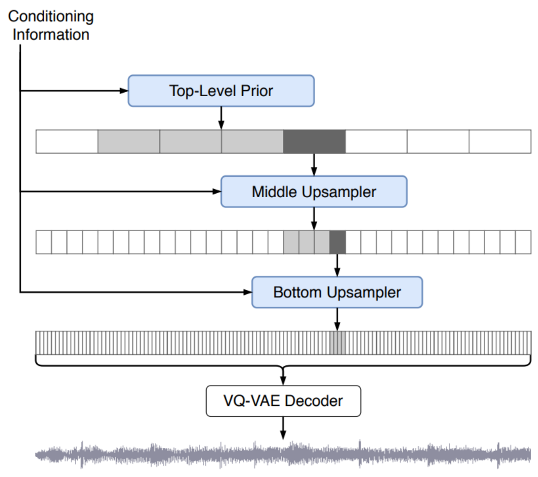


Figure 6.2: Diagram of the decoding stage of the Jukebox model [5]

Then, a cascade of Transformer models (Figure 6.2) is used to train the priors of the VQVAE codes. Each model can be conditioned with genre or artist information, and the upsamplers are also conditioned on the codes from the upper levels. After sampling the VQVAE codes from top to bottom, a decoder converts the bottom-level codes to audio. Even with multiple levels of compression, a model that handles raw audio is too complex for the amount of time and computational resources at our disposal. Instead, a symbolic approach allows us to iterate relatively fast between different approaches and architectures.

6.2.2 Symbolic music systems

Symbolic music systems train on a discrete representation of music (usually based on the MIDI standard). This approach is similar to NLP in that it captures relations between semantic units of a fixed vocabulary. For example, piano music can be encoded as a sequence of events, with a pair of *NOTE_ON* and *NOTE_OFF* events for each of the 128 MIDI note numbers [50]. This representation treats time as an implicit scale, such that the duration of a note is the number of timesteps between its *NOTE_ON* and *NOTE_OFF* events. Since multiple *NOTE_ON* events can be active in the same timestep, this encoding supports polyphonic musical pieces. Consequently, the model can reproduce chords and learn the appropriate intervals that produce vertical harmony. This event-based representation was also used in the Music Transformer project [51], which managed to produce minute-long compositions with thousands of steps by implementing an efficient version of relative position encoding [52]. However, having such a broad input vocabulary increases the complexity of the model. Consequently, we have chosen a compact representation to adapt the approach to our limited time and resources.

Chapter 7

Dataset

THE original dataset consists of 52 552 songs of varying lengths, authors and genres, stored in multiple Guitar Pro versions, namely gp3, gp4, gp5 and gtp. It is free and available for download on a peer-to-peer network. Files are grouped by artist and sorted in alphabetical order. After removing duplicate and empty files, the number of valid songs shrank to 48 323.

Every instrument used in a given song occupies a different track in the corresponding Guitar Pro file. The Guitar Pro instrument system is based on the MIDI standard, which offers both pitched and unpitched instruments. Since notes represent both pitch and duration, including unpitched sequences in the dataset could tamper with the octave and pitch class models. Consequently, only instruments that can be tuned to a specific pitch will be taken into account. In total, the dataset contains over 180 000 pitched tracks.

This chapter will explain in detail the two different datasets that have been extracted from these tracks: D1 and D2.

7.1 Dataset D1: custom parser

The first dataset is meant to represent only the pitch classes ($\mathcal{PC}$) and base durations ($\mathcal{BD}$) of chord-less guitar tracks. The Guitar Pro files are parsed using a custom script provided by the tutors. This section will explain the three stages of preprocessing: pre-parsing, parsing and post-parsing, as well as the feature distributions of the final dataset.

7.1.1 Pre-parsing

The pipeline starts with thousands of multi-track song files grouped by artist. The supervised learning approach demands that we split the data into two different distributions: train and dev-test. The model will try to fit the train distribution in a way that allows it to generalise its knowledge to the dev-test distribution.

The dataset must be split before the songs are parsed to prevent two parts of the same song from being in both distributions. In addition, all songs must be ungrouped and shuffled before splitting them to break any order between them. Otherwise, none of the artists in the dev/test set would appear in the train set. Consequently, the model would need to learn from a subset of artists and generalise to predict songs from another subset that it has never seen. The style difference between artists may be too much of a barrier for the learning algorithm to overcome, so this iteration uses song-level distribution splitting. Still, two songs from the same artist are likely to show quite different patterns, which ensures that the generalisation ability of the models is aptly tested.

After all these procedures, the dataset is split with 90% of the songs in the train distribution and 10% in the dev-test distribution. The rest of the processing pipeline is independently applied to both distributions.

7.1.2 Parsing

For the first approach, a custom parser has been provided by one of the tutors. The parser opens each song file and looks for a single chord-less track whose label contains the word "guitar". Tracks are parsed into chunks C, which are sequences of consecutive notes. Each note $c^{<i>}$ is encoded as

$$c^{<i>} = OHE(pc^{<i>}) \parallel OHE(bd^{<i>}), \tag{7.1}$$

the concatenations of the one-hot encoded pitch class $pc^{<i>}$ and base duration $bd^{<i>}$ (see subsubsection 3.4). If an instrument goes silent for a significant segment of the song, its sequence will have many consecutive rests. To avoid having an over-representation of rests, a threshold *rest_thr* is implemented. If the number of consecutive rests in a chunk exceeds this threshold, the current chunk is saved and the rest of the track is encoded in a different chunk. Since beats are usually 4 to 8 notes long, *rest_thr* is set to 8.

Furthermore, if a track ends too early or has too many consecutive rests, parsing it can produce a short chunk without any interesting musical patterns. This can be solved by setting another threshold *min_notes*, allowing us to discard chunks with less than a given number of notes. It is set to 6 to account for the window size that will later be used on this dataset.

The custom parser is implemented according to outdated documentation. As a result, it has severe compatibility issues and fails to read most of the files in the dataset, as shown in Figure 7.1. Furthermore, guitar tracks are selected based on unreliable hand-written track labels, which proves to be yet another hindrance. In the end, only 633 tracks were successfully turned into 991 chunks, with 121 going into the dev-test set and 870 into the train set.

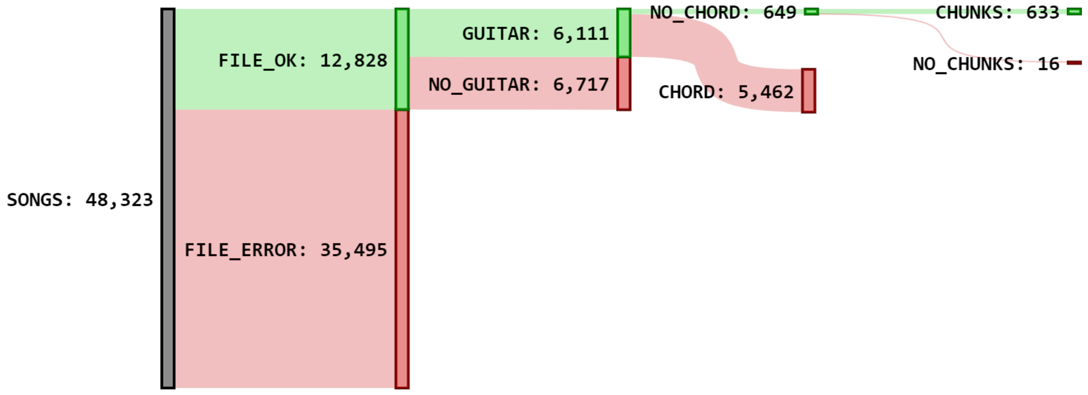


Figure 7.1: Detailed analysis of the parser performance, first iteration.

The distribution of chunk lengths in the training set (Figure 7.2) approaches an exponentially decaying function. This means that long chunks are scarce. However, the longest chunk is about 4 500 notes long, so it will yield far more examples than the short ones. Consequently, there is still a risk of over-fitting the model to long sequences.

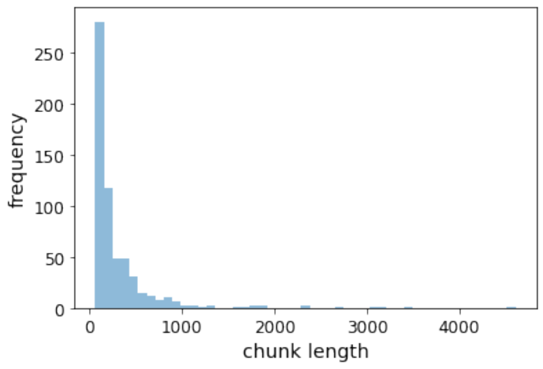
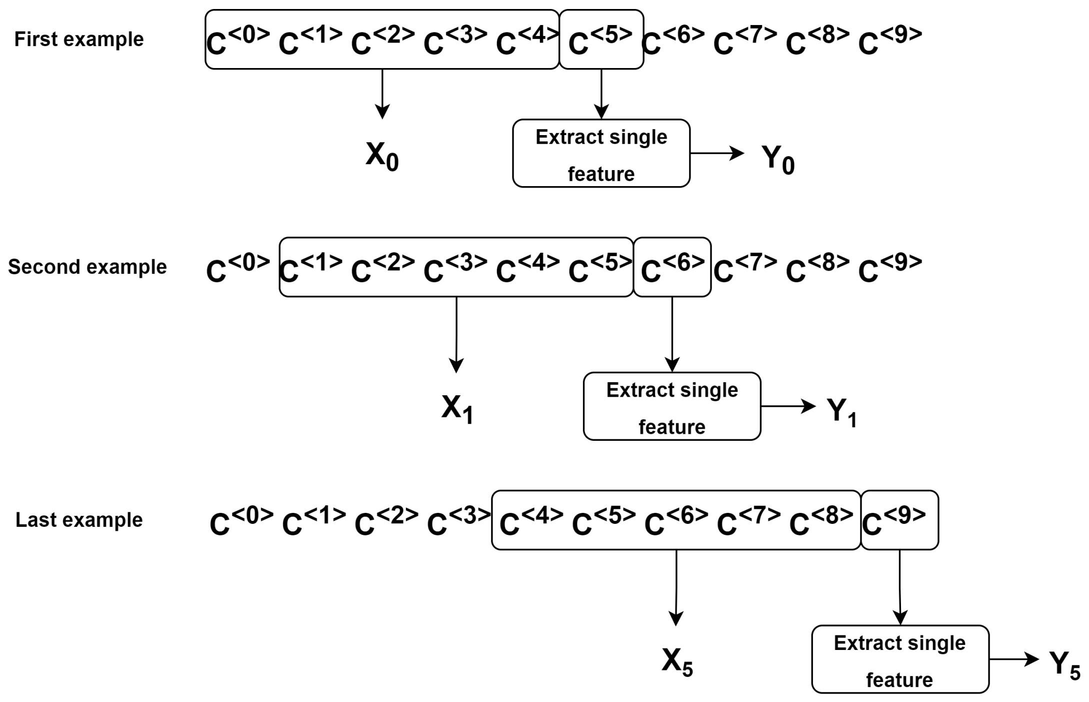


Figure 7.2: Distribution of chunk lengths from the training set, first iteration.

7.1.3 Post-parsing

Dataset examples are created by applying sliding-window indexing on each of the chunks (Figure 7.3).



First example
$C^{<0>}$ $C^{<1>}$ $C^{<2>}$ $C^{<3>}$ $C^{<4>}$ $C^{<5>}$ $C^{<6>}$ $C^{<7>}$ $C^{<8>}$ $C^{<9>}$
X_0
Extract single feature
Y_0
Second example
$C^{<0>}$ $C^{<1>}$ $C^{<2>}$ $C^{<3>}$ $C^{<4>}$ $C^{<5>}$ $C^{<6>}$ $C^{<7>}$ $C^{<8>}$ $C^{<9>}$
X_1
Extract single feature
Y_1
Last example
$C^{<0>}$ $C^{<1>}$ $C^{<2>}$ $C^{<3>}$ $C^{<4>}$ $C^{<5>}$ $C^{<6>}$ $C^{<7>}$ $C^{<8>}$ $C^{<9>}$
X_5
Extract single feature
Y_5

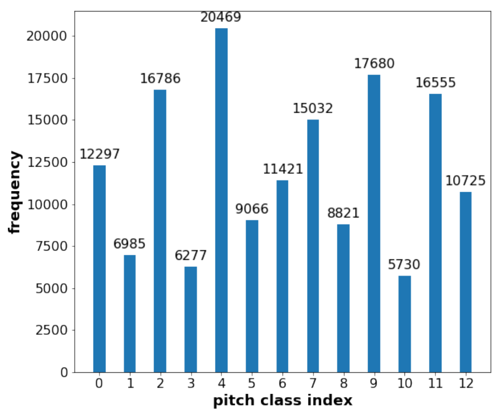
Figure 7.3: Diagram of sliding window indexing applied to a single chunk.

This procedure can be understood as an example of Self-Supervised Learning (SSL)[53] because we are building a labelled dataset from a corpus of unlabelled sequences. Each chunk C produces $length(C) - window\_size + 1$ examples. If a chunk is smaller than the window size, it is discarded.

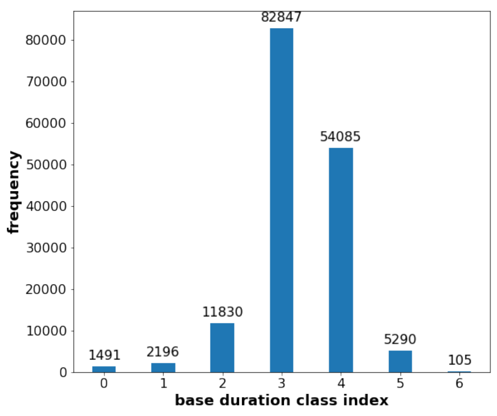
As a last precaution, we check that all classes of both features are represented in the labels of the training set. The model has no way of learning how to predict a class with no representation in the training dataset, so keeping it in the dev-test dataset can contaminate the evaluation on the dev-test distribution. Consequently, the test examples where the expected output is a class with no train set representation are discarded.

It is worth noting that as window size increases more chunks need to be discarded for being too short, so the number of dataset examples decreases. For a window size of 30, this dataset has around 160 thousand examples in training and 30 thousand in dev-test.

7.1.4 Feature distributions of the final dataset



(a) Pitch class frequencies of D1.



(b) Base duration class frequencies of D1.

Figure 7.4: Feature distributions for D1, train set.

Figure 7.4 shows that the dataset presents a class imbalance problem, especially in the base duration prediction. This issue needs to be addressed to prevent a biased model that only predicts the most frequent labels. The measures applied to prevent this are discussed in chapter 7.

7.2 Dataset D2: PyGuitarPro

The compatibility issues of the first parser prevented it from extracting a dataset with enough diversity and volume. Consequently, we implemented a second parsing pipeline based on the PyGuitarPro library. PyGuitarPro is compatible with all the Guitar Pro versions of the dataset and offers an extensive interface that allows us to extract MIDI information directly. In this approach, all features are extracted: pitch class, octaves, base duration and dots. Once again, the processing pipeline is divided into three stages: pre-parsing, parsing and post-parsing. This section will cover each of these stages and show the feature distribution in the final dataset.

7.2.1 Pre-parsing

Since multiple tracks are selected per song, track-level splitting could have been used in this approach. However, the different instruments in a song contribute to the same melody and style, so the song remains the best splitting unit for this problem. Therefore, this stage remains unchanged from the first iteration.

7.2.2 Parsing

The key differences of this parsing stage with respect to the previous approach are:

1. **Track selection:** The selection criteria for tracks are much broader. Any pitched MIDI instrument is valid and several tracks can be selected from the same song. Tracks with chords are still discarded.

2. **Feature extraction:** Pitch classes ($\mathcal{PC}$) and octaves ($\mathcal{O}$) are directly extracted from the MIDI note number NN as:

$$pc^{<i>} = nn^{<i>} \mod 12 + 1 \qquad\qquad o^{<i>} = \lfloor nn^{<i>}/12 + 1 \rfloor \tag{7.2}$$

3. **Note representation:** Each note now conveys four separate features: pitch class ($\mathcal{PC}$), octave ($\mathcal{O}$), base duration ($\mathcal{BD}$) and dot ($\mathcal{D}$)

$$c^{<i>} = (pc^{<i>}, o^{<i>}, bd^{<i>}, d^{<i>}) \tag{7.3}$$

   Since this parsing approach is expected to yield a much larger volume of examples, each feature class is stored as a scalar integer. The encoding of each of these features will be delegated to the model. This process will be explained in detail in chapter 8.

The threshold values of the algorithm were changed. In addition, a new threshold was implemented to discard long chunks.

- *rest_thr* is set to 5. Since most beats have 4 notes, five consecutive rests are often equivalent to a full beat of silence.
- *min_notes* is set to 60 to discard chunks that are too small for the windowing process.
- *max_notes* is set to 1000 to limit the over-representation of long chunks in the dataset.

The new parser yields 64 390 chunks, with their lengths following an exponential decay similar to that of the first iteration, although quite less pronounced (Figure 7.5).

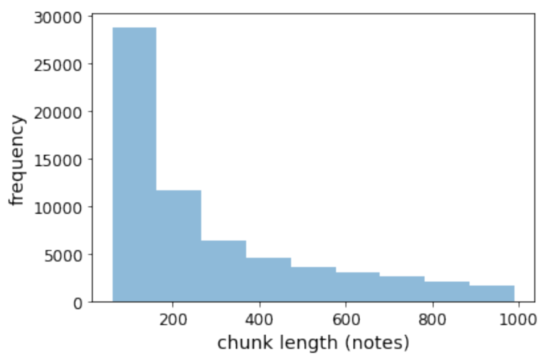


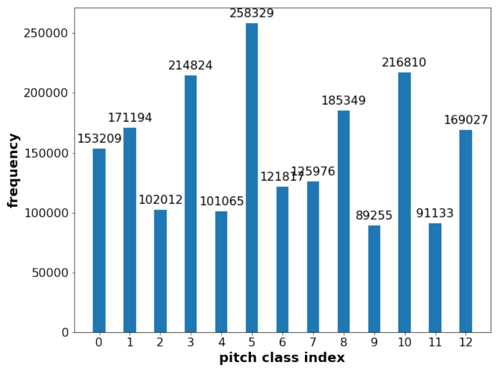
Figure 7.5: Chunk length distribution for the second iteration.

7.2.3 Post-parsing

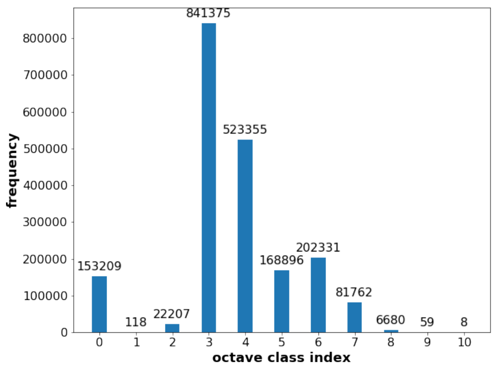
Since the parser selection criteria are much broader than during the first approach, the data set size becomes unwieldy. Therefore, the windowing stage is performed before training and the final dataset is stored on disk. With a window size of 30, this yields over 18 million total examples. To train the models in a reasonable time, both distributions are shuffled separately and trimmed to a total of two million training examples and 200 thousand for the dev-test set.

7.2.4 Feature distribution

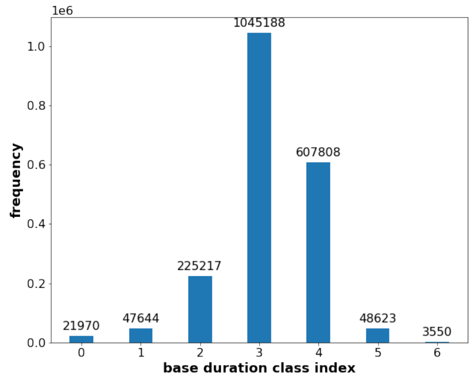
The charts in Figure 7.6 show that increasing the number of examples does not solve our class imbalance problem. Consequently, the training methodologies applied to this dataset will still need to address this issue.



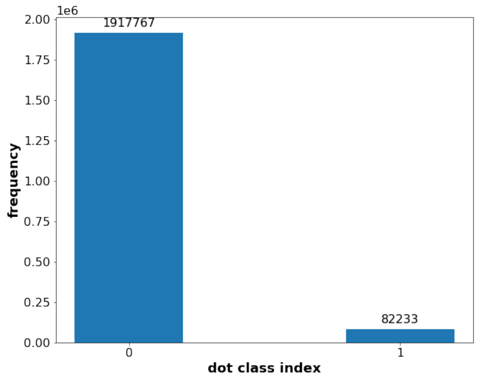
(a) Pitch class frequency distribution.



(b) Octave class frequency distribution.



(c) Base duration class frequency distribution.



(d) Dot class frequency distribution.

Figure 7.6: Feature distributions for the second iteration, train set.

Chapter 8

Methodologies

This chapter will discuss the different techniques and methodologies that have been used in both iterations of this project.

8.1 First iteration: preliminary approach

For the first approach, dataset D1 (see section 7.1) is used to tackle the prediction of pitch classes and base duration. This section will introduce our proposed architecture and how it evolved across multiple experiments.

8.1.1 Encoder architecture

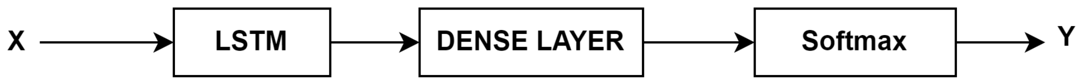


Figure 8.1: LSTM encoder architecture.

The process starts with a basic encoder architecture that will be iteratively tuned according to the evaluation results of each experiment (Figure 8.1). The shape of the input sequence X is $(N, S, 20)$, where N is the number of examples, S the input window size and 20 the depth of the vector that contains the one-hot encoded pitch class and base duration (see subsubsection 3.4). As for the output Y, it is a vector whose size matches the number of classes of the feature being predicted (13 for pitch class and 11 for base duration). From the input data, two models are trained on a many-to-one task to predict the next pitch class and base duration, respectively. Both models are trained and evaluated separately.

More details of the model are explained hereunder, namely the loss function and all other relevant settings that were set before the first experiment.

Loss function

In an attempt to deal with the class imbalance of both features, a weighted version of Categorical Crossentropy (CCE) (see subsection 3.1.3) is chosen as a loss function. The Weighted Categorical Crossentropy (WCCE) loss function is defined as

$$L_{wcce}(\hat{y}, y) = -\sum_{i=1}^{K} W_{\hat{y}}\hat{y}\log y, \tag{8.1}$$

where K is the number of classes of a given feature, y the expected output and $\hat{y}$ the value predicted by the model. The idea is to assign a high loss to the least frequent classes. Consequently, the weights of each class ($W_{\hat{y}}$) are inversely proportional to their frequencies within the data set (F):

$$W_{\hat{y}} = \frac{1}{F_{\hat{y}}} * \frac{K}{\sum_{i=1}^{K}\frac{1}{F_i}}, \tag{8.2}$$

Starting configuration

- The LSTM network is fitted with a ReLU activation function.
- The network is trained using the ADAM optimisation algorithm. The learning rate is set to 10^{-3}.
- Mini-batch size is set to 32.
- The model is trained for 150 epochs.
- The size of the hidden space is 8 “units”.
- The datasets are shuffled before each epoch.

8.1.2 Hyperparameter tuning

Across several experiments, the following hyperparameters have been modified trying to get a better performance.

1. **Loss function**: The performance of the WCCE loss function is compared to the standard CCE implementation.
2. **Window size:** This hyperparameter controls the length of the input sequence and, consequently, the complexity of musical patterns that the model has to capture.

3. **Optimisers:** the performances of three different optimisers were compared: SGD, RMSProp and ADAM. Their learning rates were also modified across several experiments.

4. **Size of the hidden state:** The size of the memory cell of the LSTM network determines how much information it can use in its prediction.

5. **Number of LSTM layers:** For some experiments, two or more LSTM encoders were stacked to produce a deep RNN architecture.

8.2 Second iteration: building embeddings

The second iteration uses dataset D2 (see section 7.2) to tackle all four modelling problems: pitch, octave, base duration and dot classes. To narrow the hyperparameter search space, the input window size is fixed to 30 notes. Furthermore, given the poor results of the first iteration, the sparse one-hot encoding is replaced with four separate embedding layers that map each feature class index to a real-valued vector.

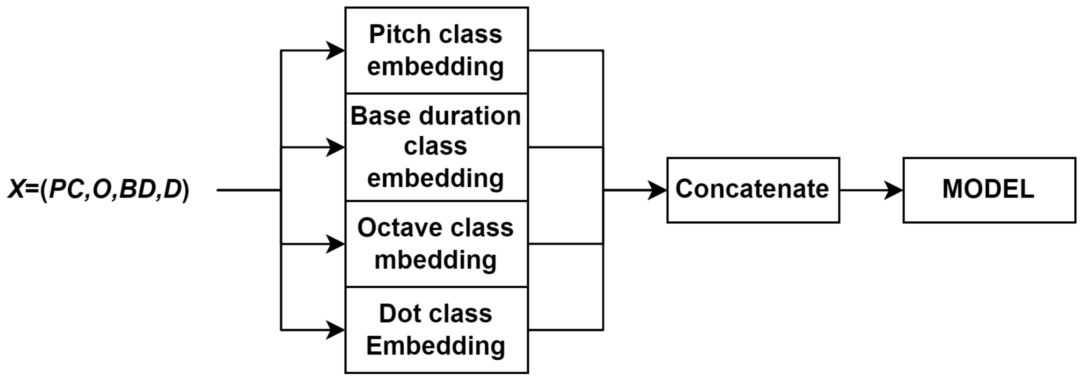


Figure 8.2: Embedding and concatenation of the four input features of X into a window of feature-rich vectors.

The four embeddings are then concatenated and served as the input to the LSTM network (Figure 8.2). Consequently, the input shape of the model is now $(N, 30, 4)$, where N is the number of examples. Each example is a window of 30 notes which, in turn, are encoded as four integer values each. The output shape remains unchanged with respect to the first iteration.

Three separate architectures have been evaluated for this iteration: the original LSTM encoder and two approaches that apply multitask learning to predict the four features with the same model. The multi-output models were built by appending four separate dense layers at the output of the model (Figure 8.3) instead of one.

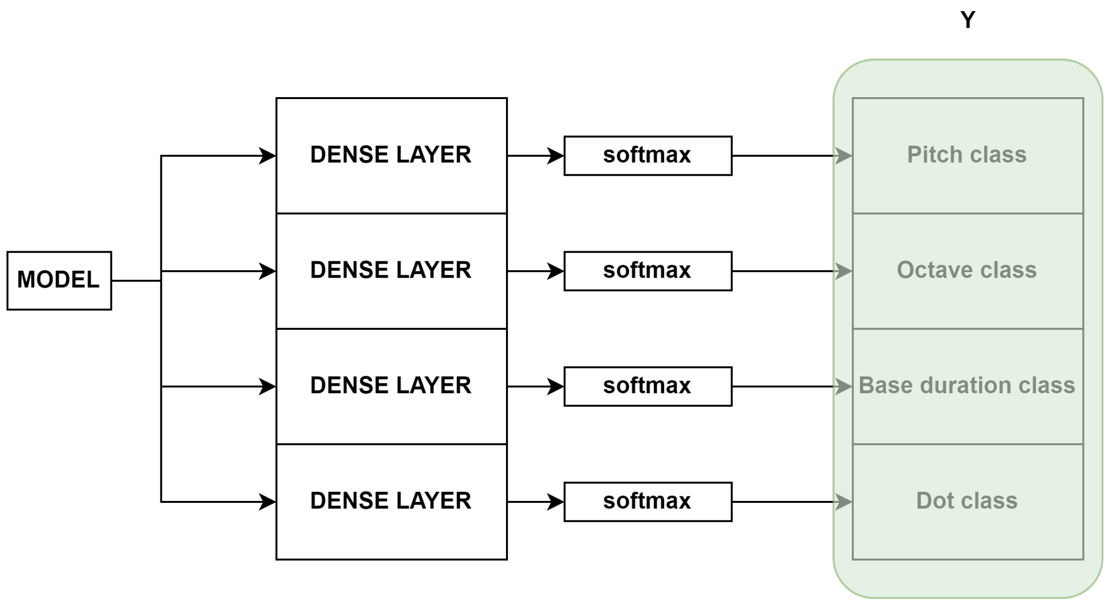


Figure 8.3: Implementation of multitask learning, second iteration.

In multitask learning, the loss function would be computed as the sum of the four individual losses, but the sum was weighted so that the models with less possible classes (and consequently, more severe class imbalance) have a greater contribution to the aggregate loss. By adding yet another level of loss weighting, we aim to compensate the disproportion in number of examples for features with severe class imbalance, such as octaves and base duration.

Some experiment settings of the first iteration are still applied on this one. For example, the datasets are still shuffled before each epoch and mini-batch size is still 32. In addition, all models are now trained for just 100 iterations. The rest of this section will discuss the starting configurations of the three architectures and how they have been tuned.

8.2.1 Multi-output recurrent encoder

The most noteworthy aspects of the starting configurations of the multi-output LSTM models are:

- The embedding size of each input matches the number of classes of that particular feature (13 for pitch class, 11 for octaves, 7 for base duration and 2 for dots).
- The model uses the ADAM optimisation algorithm with a learning rate of 10^{-3}.

- The size of the hidden space is 128 “units”.

Across several experiments, the following hyperparameters have been tuned:

- **Embedding sizes**: The embedding size of each input feature determines the number of parameters that it can directly affect.
- **Learning rate:** The learning rate of the ADAM algorithm was tuned to control the speed of the learning process (see subsection 3.1.4).
- **Number of LSTM layers:** An alternative model of three LSTM layers is used. The hidden spaces of the layers are 128, 64 and 32 “units“ each.
- **Activation function:** The activation function of the LSTM network was changed to a SELU.

8.2.2 Single-output recurrent encoder

The most noteworthy aspects of the starting configurations of the single-output LSTM models are:

- The embedding size of all inputs is set to 8. An equal embedding size ensures that all four features can contribute to the final prediction in the same proportion.
- The model uses the ADAM optimisation algorithm with a learning rate of 10^{-4}.
- The size of the hidden space is 128 “units“.

Across several experiments, the following hyperparameters have been tuned:

- **Number of LSTM layers:** An alternative model of three LSTM layers is used. The hidden spaces of the layers are 128, 64 and 32 “units“ each.
- **Activation function:** The activation function of the LSTM network is changed to a SELU.

8.2.3 Multi-output Transformer encoder

An implementation of a Transformer model for time series classification was taken from the Keras example website [54] and adapted to our multitask approach. This architecture is based on a Transformer encoder block, which applies layer normalisation, multi-head attention and convolutions to the input. Each block (Figure 8.4) also features residual connections.

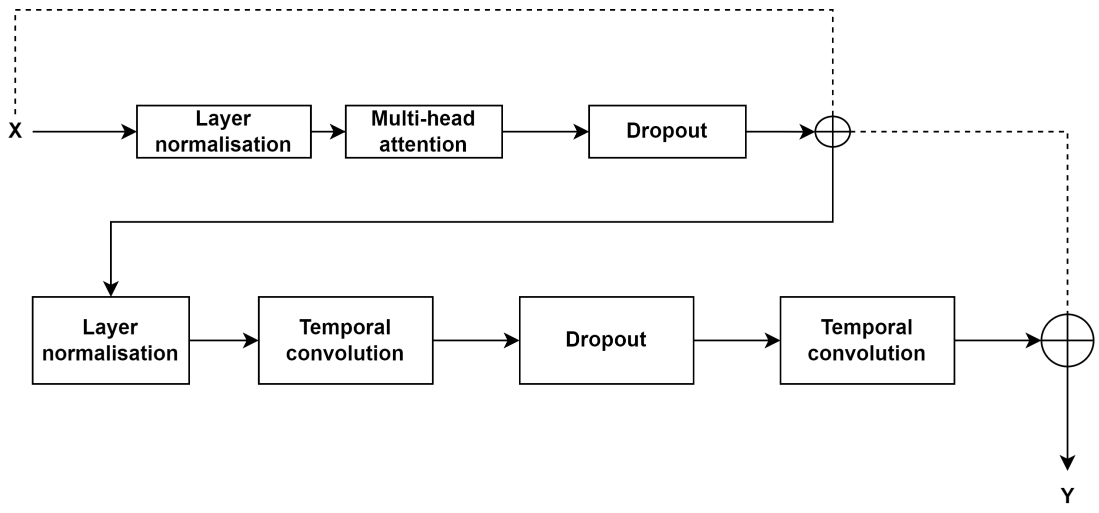


Figure 8.4: Transformer encoder block, second iteration.

Multiple of these blocks can be stacked to increase their fitting capacity. The output of the final block is then pooled and processed by a dense layer. Finally, four additional dense layers from our multi-task implementation return the output probability distributions. To prevent overfitting, several dropout layers are used. Due to time constraints, the hyperparameters

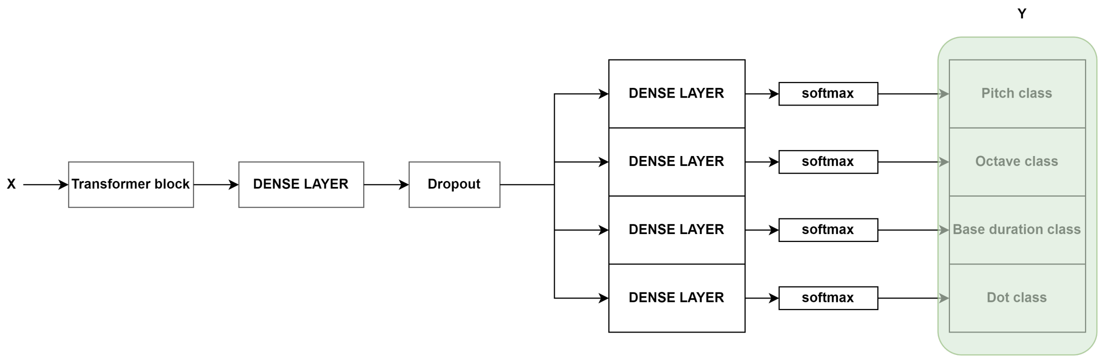


Figure 8.5: Complete Transformer model, second iteration.

of this network could not be tuned. The main hyperparameters of this experiment are listed below.

- Four encoder blocks are stacked together.
- The dropout rate inside each block is 0.25.
- The multi-head attention layer has four heads
- Head size is set to 256.

- The dropout rate of the final dense layer is 0.4.

Chapter 9

Evaluation

This chapter will start by defining the evaluation metric of this project, to then discuss the results of each iteration. Lastly, we will perform an error analysis on the best model configurations.

9.1 Evaluation metric

For multi-class classification problems, accuracy is a common evaluation metric. It is defined as

$$accuracy = \frac{\sum_{j=1}^{N} \hat{y}_j == y_j}{N} \tag{9.1}$$

where N is the total number of samples; and $\hat{y}_j$ and y_j are the predicted result and the true label of the i-th sample, respectively. The accuracy metric shows the fraction of model predictions that match the expected output. However, this metric is not suitable for problems with class imbalance. If the classes of a problem are imbalanced, a trivial model that always returns the same class will yield high accuracy. We need an evaluation metric that addresses this problem.

Balanced precision does this by treating the prediction of each class as a binary classification problem. First, we compute the precision of each class i from the True Positive (TP) and False Positive (FP) values of the confusion matrix:

$$Precision_i = \frac{TP_i}{TP_i + FP_i} \tag{9.2}$$

Then, balanced precision is computed as their average taken across all possible K classes:

$$Balanced\, precision = \frac{\sum_{i=0}^{K} Precision_i}{K} \tag{9.3}$$

Since our implementation uses mini-batch gradient descent, it is worth noting that the recalls are only computed for the classes that were expected labels for any example of the current mini-batch, so K can be smaller than the total number of classes for that feature.

To get high balanced precision, the model must have a good precision on every class that appears on the mini-batch. Because of how it handles class imbalance, this metric is a reliable and robust indicator of the performance of our model. Therefore, it will be used as an evaluation metric. From now on, we will use the acronym ba to refer to this metric, with subscripts $\mathcal{PC}$, $\mathcal{OCT}$, $\mathcal{BD}$ and $\mathcal{DT}$ to denote each of the four problems: pitch class, octave, base duration and dot prediction. Lastly, the results for the train distribution will have the subscripts $train$, while results obtained on the dev-test dataset will have the subscript val.

9.2 First iteration

9.2.1 Baselines

For each approach, two models were implemented to serve as a reference of what performance could be achieved without modelling the sequential nature of examples:

- ***Naïve***: for both classification problems, this model predicts that the output value is the same as it was on the last note of the input window.
- **FFWD network**: a deep neural network with three hidden layers trained for 150 epochs.

Since none of these models show temporal dynamic behaviour, our sequence models should perform better than both of them.

The performance of the baseline models reflects the severity of the class imbalance for each problem. Pitch classes are not uniformly represented, but base duration sequences are far less dynamic. As seen in Table 9.1, trivial strategies such as repeating the last note work much better for base duration modelling than for pitch modelling.

Model	$ba_{pc,val}(\%)$	$ba_{bd,val}(\%)$
Naïve	13.6	51.6
FFWD	26.8	55.0

Table 9.1: Balanced precision (%) of the baseline models on the validation distribution, first iteration.

9.2.2 Loss function

The first approach of this iteration compared the performance of the same recurrent model twice, changing its loss function from CCE to WCCE (see subsubsection 8.1.1). Since the WCCE loss function tackles the problem of class imbalance, it was expected to outperform regular CCE.

Loss function	$ba_{pc,val}(\%)$	$ba_{bd,val}(\%)$
CCE	26.11	43.82
WCCE	27.28	61.13

Table 9.2: Balanced precision comparison between CCE and WCCE, first iteration.

Table 9.2 shows that, even though both problems benefit from a weighted loss function, the performance boost was far more significant in the base duration problem. The reason for this is, once again, the different degrees of class imbalance that both problems present. Since the base duration sequences were so skewed towards a single class, assigning a higher weight to the rest of the classes allowed the model to learn far more complex strategies. The rest of the models in this iteration were also trained on a WCCE loss function.

9.2.3 Window size tuning

The window size was increased hoping that, as the input window got bigger, its patterns grew in complexity and forced the model to refine its strategies.

Window size (notes)	$ba_{pc,val}(\%)$	$ba_{bd,val}(\%)$
5	25.75	51.13
10	26.21	61.24
20	25.12	71.35
30	25.01	69.70

Table 9.3: Balanced precision comparison between different input window sizes, first iteration.

The results in Table 9.3 show that pitch class prediction gained no improvement from this change. In contrast, the base duration model seemed to benefit significantly from this change, boosting performance by 10% with a window size of just 20 notes. This is understandable, since it is likely that all the input notes of a window of five had the same duration, whereas pitch classes are far more dynamic. In short, the base duration prediction model needed a bigger window size to capture meaningful patterns. Considering that increasing window size has an impact on training time, a window size of 10 notes was chosen for both models.

9.2.4 Optimiser

The next hyperparameter to be tweaked in the tuning process was the optimiser. The learning rate of the ADAM algorithm was lowered to 10^{-4}. SGD and RMSProp were also tested with learning rates of 10^{-2} and 10^{-4}, respectively.

Optimiser	$ba_{pc,val}(\%)$	$ba_{bd,val}(\%)$
ADAM (10^{-3})	26.21	65.39
ADAM (10^{-4})	25.29	62.62
SGD	24.91	62.64
RMSProp	26.20	57.27

Table 9.4: Balanced precision comparison between different optimiser configurations, first iteration.

However, Table 9.4 shows that none of these alternatives managed to surpass the performance of the original configuration (ADAM with a learning rate of 10^{-3}). This led us to consider that the problem was not related to the optimisation algorithm. Instead, we suspected the model to be too simple to capture the nuances of the dataset, so we kept the original optimiser and tried to increase the complexity of our model.

9.2.5 Number of layers

The first approach to combat the underfitting problem of both models was a deep RNN. Three LSTM layers were stacked to generate high-level features. In addition, the size of the hidden state was increased to 16 "units" to further increase the capacity of the model.

Number of layers	$ba_{pc,val}(\%)$	$ba_{bd,val}(\%)$
1 (8 units)	26.21	65.39
1 (16 units)	32.16	65.21
3 (16 units)	33.09	57.79

Table 9.5: Balanced precision comparison between shallow and deep LSTM models, first iteration.

The results in Table 9.5 show that increasing the size of the hidden space caused a slight increase in pitch class performance, whereas adding layers proved to be pointless at this stage.

Consequently, the last set of experiments in this iteration increased the number of “units” of the LSTM layer.

9.2.6 Hidden space size

Increasing the size of the hidden space of the network widens the search space of the learning algorithm, so it is a straight-forward way of increasing the complexity of our models.

Units	$ba_{pc,train}(\%)$	$ba_{pc,val}(\%)$	$ba_{bd,train}(\%)$	$ba_{bd,val}(\%)$
20	47.78	32.75	84.38	53.25
50	64.61	36.04	92.73	53.12
100	79.78	43.54	95.92	49.95
150	89.00	50.88	97.15	48.97
175	92.00	52.96		

Table 9.6: Balanced precision comparison between different hidden space sizes, first iteration.

Indeed, 175 “units” was enough to reach over 90% balanced precision on the pitch class training set, while the validation metric reached 50%, significantly higher than any other previous approach (Table 9.6). As for the base duration problem, a similar train metric was achieved with 150 “units”, but the validation metrics oscillated violently below previous values. It is likely that, since base duration sequences are so unbalanced, the most effective way to get better results in this problem is by increasing the window size.

9.3 Second iteration

The metric for this iteration was the same as in the first one. This section will cover the baseline results, describe the conditions of each experiment and then evaluate the four problems separately.

9.3.1 Baselines

The *naïve* and FFWD models were still used as baselines for this iteration. However, the FFWD model was modified to the starting configuration of the single output recurrent models.

Model	$ba_{pc,val}(\%)$	$ba_{oct,val}(\%)$	$ba_{bd,val}(\%)$	$ba_{dt,val}(\%)$
Naïve	49.03	64.57	52.52	59.18
FFWD	62.38	68.44	67.88	82.56

Table 9.7: Balanced precision of the baseline models in the second iteration.

The results of the FFWD model (Table 9.7) showed that even a model with no concept of temporal sequences could get a decent result in a prediction task where the classes are severely unbalanced. In fact, the best result was achieved in the dot prediction problem. Moreover, applying evenly deep embeddings to the four input features seemed to boost the performance of the FFWD network.

9.3.2 Experiments

Six sets of experiments were performed in this iteration. Each of them except for the Transformer compared a single-layer architecture to a deep one. The hyperparameter configuration of each of them must be explained before discussing the results.

1. A multi-output recurrent model with uneven embedding depths, trained with the ADAM algorithm with a learning rate of 10^{-3}.
2. The embedding depth was set to 8 per feature and the learning rate lowered to 10^{-4}.
3. The same configuration as experiment 2, but with a SELU activation function.
4. A Transformer encoder was configured with the same hyperparameters as experiment 2.
5. A single-output recurrent model was configured with the same hyperparameters as experiment 2.
6. The same single-output model was used, but with a SELU activation function.

9.3.3 Pitch class

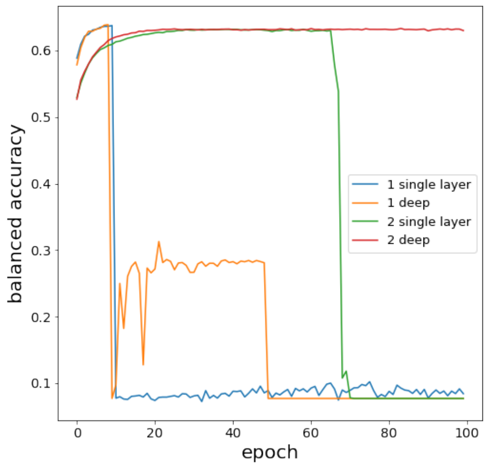


Figure 9.1: Balanced precision comparison between multitask models 1 and 2 on the pitch class prediction problem. Model 1 has uneven embedding depths and trains on a learning rate of 10^{-3}, while model 2 has a constant embedding depth of 8 and trains on a learning rate of 10^{-4}.

Figure 9.1 shows multi-output models with uneven embeddings suffering from sudden performance drops to random performance at the early stages of training. Meanwhile, the ones with evenly-embedded inputs and a lower learning rate exhibit greater robustness. It seems that traversing the gradient in smaller steps with a balanced input representation prevented these abrupt changes. It is also worth noting that the single-layer models appeared to be vulnerable to these drops, while deep models either kept improving or show signs of recovery after them. This led us to believe that having high-level intermediate features also prevented the model from dropping. However, even the best of these models hardly surpassed the baseline performance, so further tuning is required.

Model	$ba_{pc}(\%)$
Baseline	62.38
Best so far	62.99
Transformer	28.66
Single-layer, SELU	63.69
Deep, SELU	64.03

Table 9.8: Balanced precision comparison between different multi-task models on the pitch class prediction problem, second iteration.

The SELU activation function seems to add even more stability to the multi-output models, since none of them suffer performance drops and they both surpass baseline performance (Table 9.8). Given that the single-layer and deep models get nearly identical results, it would seem that using a deep network is not worth the additional computational cost, but the robustness to performance drops that they showed in the previous experiments could be important. On the same table, the multi-output Transformer encoder shows a really poor performance for this problem. It could be that weighting the loss function to prioritise the features with less classes is hindering the pitch class output of this model.

Model	$ba_{pc,train}(\%)$	$ba_{pc,val}(\%)$
Baseline	64.72	62.38
Best so far	67.28	64.03
Single-layer, ReLU	68.73	65.08
Deep, ReLU	71.24	65.04
Single-layer, SELU	68.78	64.85
Deep, SELU	71.83	65.10

Table 9.9: Balanced precision comparison between single output models on the pitch class prediction problem, second iteration.

As for the single output models in Table 9.9, they show such a small improvement with respect to the multi-output approaches (about 1%) that it is hard to justify training them as a separate model. The SELU activation function and the additional layers of the deep recurrent network failed to boost performance, whereas the “shallow” model with ReLU activation manages to avoid any drops.

9.3.4 Octave

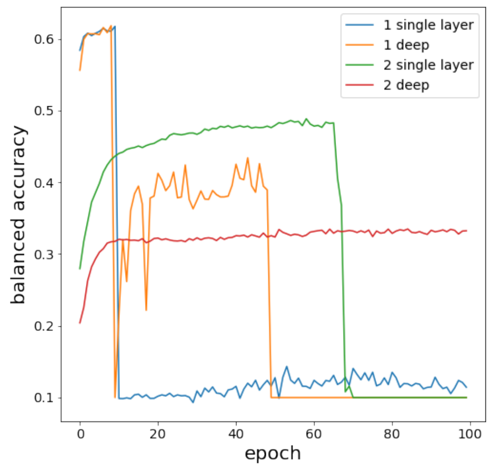


Figure 9.2: Balanced precision comparison between multitask models 1 and 2 on the octave class prediction problem. Model 1 has uneven embedding depths and trains on a learning rate of 10^{-3}, while model 2 has a constant embedding depth of 8 and trains on a learning rate of 10^{-4}.

The multi-output models with unevenly embedded inputs showed a promising trend for the first epochs, but also suffered from performance drops at the early stages of training (Figure 9.2). The models with even embeddings started at a much lower performance but showed a higher level of robustness. A plausible explanation for these observations is that, since the octave prediction problem has eleven classes, the uneven embeddings favoured this problem over the rest. However, the higher learning rate of those same models may have caused them to drop earlier. The deep models are still more robust on average, but plateau at much lower performance values.

Model	$ba_{oct}(\%)$
Baseline	68.44
Transformer	27.00
Single-layer, SELU	53.41
Deep, SELU	33.96

Table 9.10: Balanced precision comparison between different multi-task models on the octave class prediction problem, second iteration.

Using a SELU activation function on the multi-output models seemed to stabilise the optimisation process, but also caused it to plateau at a much lower value than the baseline (Table 9.10). The performance of the deep model was, once again, lower than the “shallow” one. The Transformer model had an ever worse performance, which supports the idea that it is susceptible to the unbalanced loss weights that favour features with less classes.

Model	$ba_{oct,train}(\%)$	$ba_{oct,val}(\%)$
Baseline	70.82	68.44
Single-layer, ReLU	72.72	73.08
Deep, ReLU	66.85	64.29
Single-layer, SELU	85.32	71.55
Deep, SELU	91.00	64.34

Table 9.11: Balanced precision comparison between single output models on the octave class prediction problem, second iteration.

In comparison, single-output models showed relatively good results (Table 9.11), with great robustness and resilience regarding performance drops. The single-layer ReLU model surpassed baseline performance by 4.64%. It is worth noting that, even if the SELU activation function got worse validation results than ReLU, it achieved the highest performance on the training set.

Performance drops seemed to be frequent in problems with high class imbalance, so they may have been caused by how we deal with this problem. Infrequent classes have high loss weights, but some classes are so infrequent that they cause a disproportionate gradient value to be propagated through the model.

9.3.5 Base duration

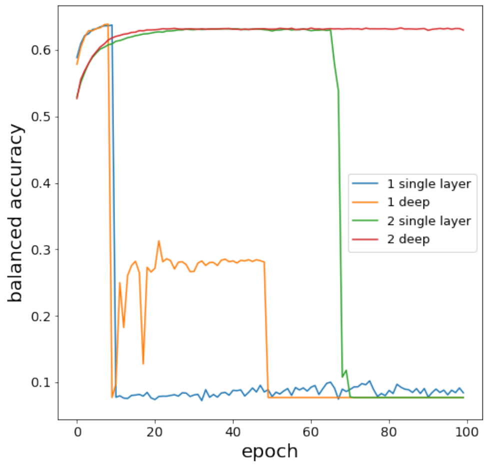


Figure 9.3: Balanced precision comparison between multitask models 1 and 2 on the base duration class prediction problem. Model 1 has uneven embedding depths and trains on a learning rate of 10^{-3}, while model 2 has a constant embedding depth of 8 and trains on a learning rate of 10^{-4}.

The performance of multi-output models with uneven embeddings (Figure 9.3) dropped in the early epochs of training, while the ones with even embeddings showed greater stability. Our intuition of deep models being robust still applies, as we can see them either recovering from drops or not falling at all. In this case, the model benefitted from having evenly-embedded inputs because the base duration problem had less than eight classes. The deep, evenly embedded model performed slightly better than the baseline.

Model	$ba_{bd}(\%)$
Baseline	67.88
Best so far	70.61
Transformer	43.81
Single-layer, SELU	72.12
Deep, SELU	71.67

Table 9.12: Balanced precision comparison between different multi-task models on the base duration class prediction problem, second iteration.

The recurrent multi-output models with SELU activation function (Table 9.12) yielded slightly better results to the ones that had a ReLU, while the Transformer performed, once again, way below baseline.

Model	$ba_{bd,train}(\%)$	$ba_{bd,val}(\%)$
Baseline	72.84	67.88
Best so far	76.12	72.12
Single-layer, ReLU	81.03	75.58
Deep, ReLU	14.37	14.34
Single-layer, SELU	86.85	74.79
Deep, SELU	88.88	74.50

Table 9.13: Balanced precision comparison between single output models on the base duration class prediction problem, second iteration.

The single output models in Table 9.13 showed a significant improvement over the ones with multiple outputs, with the SELU functions slightly lowering the final performance. Surprisingly, the deep ReLU model dropped, while the "shallow" model improved baseline performance by 7.7%.

9.3.6 Dot

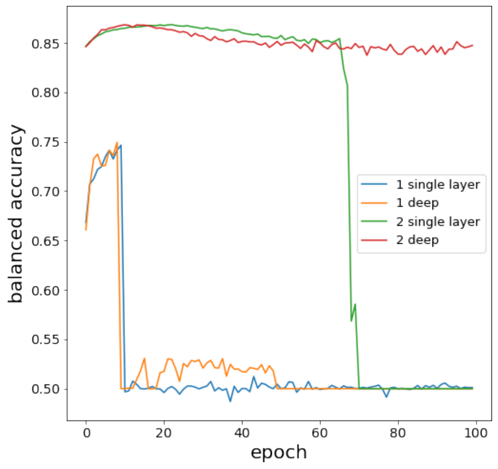


Figure 9.4: Comparison between multitask models 1 and 2. Model 1 has uneven embedding depths and trains on a learning rate of 10^{-3}, while model 2 has a constant embedding depth of 8 and trains on a learning rate of 10^{-4}.

The multi-output models with uneven embeddings (Figure 9.4) not only dropped early, but peaked at a lower performance than the evenly-embedded. Similarly to base duration, the dot prediction problem has less than eight classes, so an even distribution of embedding depth favoured the output of the model for this problem. Yet again, the deep model was the only one to resist the performance drop, in this case finishing 2.19% over the baseline.

Model	$ba_{dt}(\%)$
Baseline	82.56
Best so far	84.75
Transformer	84.23
Single-layer, SELU	84.36
Deep, SELU	82.31

Table 9.14: Balanced precision comparison between different multi-task models on the dot class prediction problem, second iteration.

Both the recurrent model with SELU activation and Transformer surpassed baseline performance, but not the best performance up to that point. Since the aggregate loss function of the Transformer model was weighted to prioritise features with less classes (see section 8.2), the dot prediction problem was the only output where it showed decent results (Table 9.14).

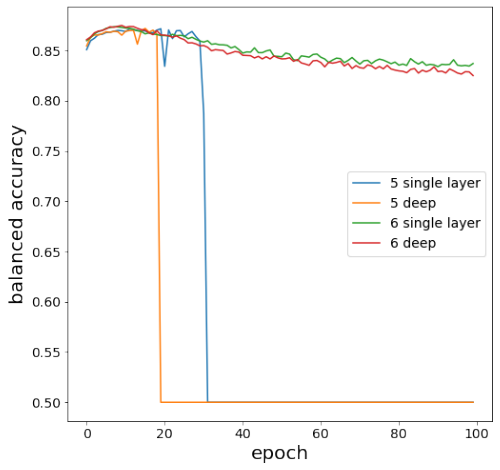


Figure 9.5: Evaluation balanced precision comparison between single-output models for the dot prediction problem, second iteration. Model 5 has a ReLU activation function, whereas model 6 has a SELU activation function.

Finally, the single output models actually performed worse than those with multiple outputs, suggesting that this problem benefits from a multi-task approach. Once again, the SELU activation function added robustness to the model, protecting it from performance drops (Figure 9.5).

9.4 Final error analysis

This section will discuss the errors of the models with the best performance on each of the four problems that have been tackled in this project.

9.4.1 Pitch class

The best performance in the pitch class prediction problem was achieved by the single-output, deep LSTM network with a SELU activation function. It scored 65.10% balanced accuracy, just 2.6% higher than the baseline for that iteration. Such a small improvement may lead us to think that our model still has significant room for improvement, but the confusion matrix in Figure 9.6 shows that the model is already well tuned to the validation set.

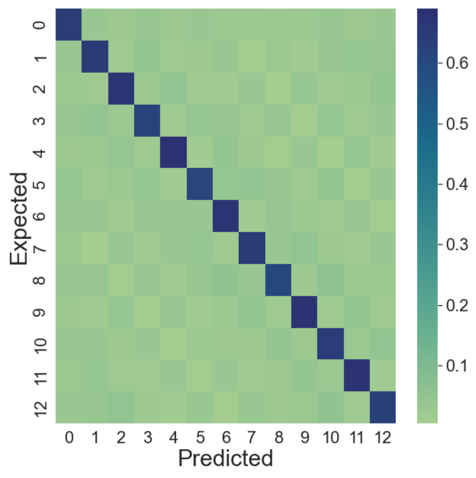


Figure 9.6: Row-normalised confusion matrix of the best model on pitch class prediction.

9.4.2 Octave

As for the octave class prediction problem, the single-layer LSTM model with a ReLU activation function got 73.08%, 4.68% higher than baseline performance. Once again, the confusion matrix (Figure 9.7) shows great class balance in the predictions, with most of the errors staying close to its diagonal. However, the diagonals of this matrix are not as sharp as they were on the pitch class prediction problem. It would seem that the severe class imbalance of this feature causes problems when predicting the octave index zero (the rest) and 10. Moreover, octave index 1 is not accounted for on this diagram, since it was present neither as an expected value of the validation set nor as a value predicted by the model.

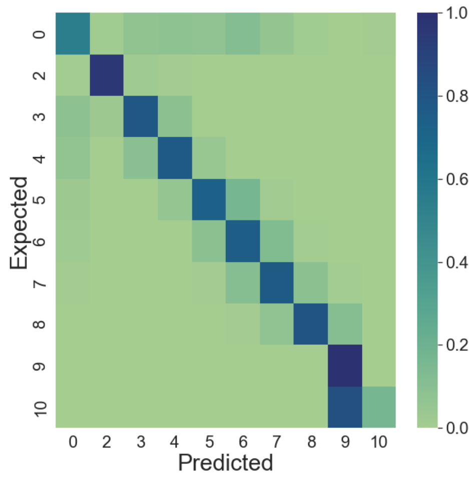


Figure 9.7: Row-normalised confusion matrix of the best model on octave class prediction.

9.4.3 Base duration

The best performing model on the base duration problem had the same configuration as the best octave class predictor: a single-layer LSTM encoder with a ReLU activation function. In this case, it managed to reach 75.58% balanced accuracy, 7.68% higher than the baseline.

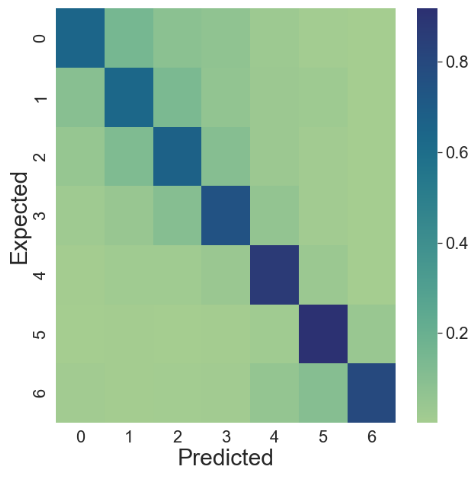


Figure 9.8: Row-normalised confusion matrix of the best model on base duration class prediction.

Although figure 9.8 shows a relatively low performance on long durations (indices 0,1 and 2), most errors were close to the diagonal. From these results we can infer that, in spite of our approach being categorical, our models have learned to order duration on a relative scale.

9.4.4 Dot

The model with the highest performance on the dot prediction problem is the deep LSTM architecture with multiple outputs. Even though the aggregate loss function of this model was weighted to favour dot prediction, it only managed to perform 2.19% better than the baseline with a 84.75% balanced accuracy.

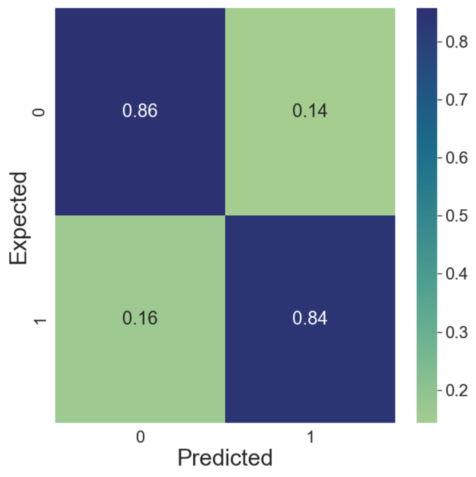


Figure 9.9: Row-normalised confusion matrix of the best model on dot class prediction.

However, the confusion matrix (Figure 9.9 shows that it performs quite well on both classes, which is surprising given the actual frequency of dots in the dataset (see subsection 7.2.4). This is most likely because the loss weighting allowed the model to generate predictions with dots even though they were infrequent in the dataset.

Chapter 10

Conclusions

10.1 Summary

This project has addressed the problem of musical language modelling using SSL on a sheet music dataset. The problem was divided into four separate classification problems: pitch class, base duration, octave and dot prediction. Different approaches have been implemented and evaluated, focusing on class imbalance as the main hurdle to the learning process.

10.2 Conclusions

The baseline performances have been surpassed on all of the four problems. This shows that we have successfully implemented models that can capture the relations between the tokens on a sequence and model their probability. Even though the metric values of our best models are similar to the baselines, the distributions of errors per class show that our models can deal with the class imbalance of the dataset and yield balanced predictions for all features.

However, we have not managed to train a multi-output model that surpasses the baseline performances for the four problems. This approach could significantly reduce training time, so it should not be discarded just yet.

It is also worth mentioning that our approach to the problem of class imbalance seems to be causing performance drops, especially in shallow models with a ReLU activation function. This issue can be addressed by considering different alternatives to loss weighting for future iterations.

10.3 Future work

Since our current approach to the issue of class imbalance seems to cause performance drops, future work on this project should evaluate different alternatives to loss weighting. For example, the dataset could be resampled so that each mini-batch of the dataset contains a balanced representation of all classes. However, the issue of class imbalance for the base duration and octave problem is so severe that this approach on its own could lead to severe overfitting. To prevent this, the dataset could be filtered by instrument to narrow the number of possible classes for these problems.

Even though the results of the multi-output models are not too promising, the advantage of training on multiple problems at the same time cannot be overstated. Consequently, the loss weights for each feature could be tuned for a few experiments before discarding this approach.

Once we are satisfied with the results on these tasks, we could use the hidden states of the models as embeddings of a generative model. These models are usually implemented with Generative Adversarial Networks (GAN)s [55], a technique that involves two networks, a generator and a discriminator, competing against each other. The generator network learns to create a novel output that resembles the dataset, while the discriminator network is fed with examples from both the dataset and the generator and is trained to discern one distribution from the other.

List of Acronyms

OHE	One-Hot Encoding. 18
ReLU	Rectified Linear Unit. vii, 13, 46, 60, 62, 64–67, 71
RMSProp	Root Mean Square Propagation. 15, 16, 47, 56
RNN	Recurrent Neural Networks. 1, 19, 21, 47, 56
SELU	Scaled Exponential Linear Unit. vii, 13, 49, 58, 60–66
SGD	Stochastic Gradient Descent. 15, 47, 56
SL	Supervised Learning. 12
SPN	Scientific Pitch Notation. 6
SSL	Self-Supervised Learning. 40, 71
VQVAE	Vector Quantized Variational Autoencoder. 34, 35
WCCE	Weighted Categorical Crossentropy. viii, 46, 55

Bibliography

[1] R. L. Bowers, “how to get finale 2012 to recognize ties on playback,” Nov 2016, last accessed 2022-06-28. [Online]. Available: https://forum.makemusic.com/default.aspx?f=5&amp;m=474094&amp;g=474103

[2] I. C. Education, “Neural networks,” 2020, last accessed 2022-06-28. [Online]. Available: https://www.ibm.com/cloud/learn/neural-networks

[3] S. Nit, “Batch , mini batch and stochastic gradient descent,” 2020, last accessed 2022-06-28. [Online]. Available: https://sweta-nit.medium.com/batch-mini-batch-and-stochastic-gradient-descent-e9bc4cacd461

[4] S. Smith, M. Patwary, B. Norick, P. LeGresley, S. Rajbhandari, J. Casper, Z. Liu, S. Prabhumoye, G. Zerveas, V. Korthikanti, E. Zhang, R. Child, R. Y. Aminabadi, J. Bernauer, X. Song, M. Shoeybi, Y. He, M. Houston, S. Tiwary, and B. Catanzaro, “Using deepspeed and megatron to train megatron-turing nlg 530b, a large-scale generative language model,” 2022. [Online]. Available: https://arxiv.org/abs/2201.11990

[5] P. Dhariwal, H. Jun, C. Payne, J. W. Kim, A. Radford, and I. Sutskever, “Jukebox: A generative model for music,” 2020. [Online]. Available: https://arxiv.org/abs/2005.00341

[6] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. M. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, and D. Amodei, “Language models are few-shot learners,” 2020. [Online]. Available: https://arxiv.org/abs/2005.14165

[7] G. Marcus, E. Davis, and S. Aaronson, “A very preliminary analysis of dall-e 2,” 2022. [Online]. Available: https://arxiv.org/abs/2204.13807

[8] M. Conner, L. Gral, K. Adams, D. Hunger, R. Strelow, and A. Neuwirth, “Music generation using an lstm,” 2022. [Online]. Available: https://arxiv.org/abs/2203.12105

[9] Y.-J. Shih, S.-L. Wu, F. Zalkow, M. Müller, and Y.-H. Yang, "Theme transformer: Symbolic music generation with theme-conditioned transformer," 2021. [Online]. Available: https://arxiv.org/abs/2111.04093

[10] C.-Z. A. Huang, A. Vaswani, J. Uszkoreit, N. Shazeer, I. Simon, C. Hawthorne, A. M. Dai, M. D. Hoffman, M. Dinculescu, and D. Eck, "Music transformer," 2018. [Online]. Available: https://arxiv.org/abs/1809.04281

[11] M. Baroni, S. Maguire, and W. Drabkin, "The concept of musical grammar," *Music Analysis*, vol. 2, no. 2, pp. 175–208, 1983. [Online]. Available: http://www.jstor.org/stable/854248

[12] J. P. Swain, "The concept of musical syntax," *The Musical Quarterly*, vol. 79, no. 2, pp. 281–308, 1995. [Online]. Available: http://www.jstor.org/stable/742247

[13] A. Klapuri and M. Davy, *Signal Processing Methods for Music Transcription.* Springer Science & Business Media, 01 2006.

[14] E. Applebaum, "Perspectives in music theory: An historical-analytical approach: By paul cooper. new york: Dodd, mead & company, 1973. 282 pp," *Music Educators Journal*, vol. 61, no. 1, p. 16, 1974. [Online]. Available: https://doi.org/10.2307/3394685

[15] A. Forte, *Tonal harmony in concept and practice.* Holt, Rinehart and Winston, 1962.

[16] "Acoustics — standard tuning frequency (standard musical pitch)," International Organization for Standardization, Geneva, CH, Standard, Jan. 1975.

[17] B. Benward and M. Saker, *Music in theory and practice, Vol 1*, 7th ed. New York, NY: McGraw-Hill, 2003.

[18] J.-J. Nattiez, *Music and discourse.* Princeton, NJ: Princeton University Press, Nov. 1990.

[19] d. smith and c. wood, "the 'usi', or universal synthesizer interface," *journal of the audio engineering society*, october 1981.

[20] M. L. Minsky and S. A. Papert, "Perceptrons: expanded edition," 1988.

[21] J. Schmidhuber, "Deep learning in neural networks: An overview," *Neural Networks*, vol. 61, pp. 85–117, jan 2015. [Online]. Available: https://doi.org/10.1016%2Fj.neunet.2014.09.003

[22] L. Deng and D. Yu, "Deep learning: Methods and applications," *Foundations and Trends® in Signal Processing*, vol. 7, no. 3–4, pp. 199–200, 2014, last accessed 2022-06-28. [Online]. Available: http://dx.doi.org/10.1561/2000000039

[23] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016, http://www.deeplearningbook.org.

[24] K. Fukushima, "Visual feature extraction by a multilayered network of analog threshold elements," *IEEE Transactions on Systems Science and Cybernetics*, vol. 5, no. 4, pp. 322–333, 1969. [Online]. Available: https://doi.org/10.1109/tssc.1969.300225

[25] G. Klambauer, T. Unterthiner, A. Mayr, and S. Hochreiter, "Self-normalizing neural networks," *Advances in neural information processing systems*, vol. 30, 2017.

[26] D. C. Marcu and C. Grava, "The impact of activation functions on training and performance of a deep neural network," in *2021 16th International Conference on Engineering of Modern Electric Systems (EMES)*. IEEE, 2021, pp. 1–4.

[27] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning representations by back-propagating errors," *Nature*, vol. 323, no. 6088, pp. 533–536, Oct. 1986. [Online]. Available: https://doi.org/10.1038/323533a0

[28] H. B. Curry, "The method of steepest descent for non-linear minimization problems," *Quarterly of Applied Mathematics*, vol. 2, no. 3, pp. 258–261, 1944. [Online]. Available: https://doi.org/10.1090/qam/10667

[29] S. Ruder, "An overview of gradient descent optimization algorithms," 2016. [Online]. Available: https://arxiv.org/abs/1609.04747

[30] Tieleman and G. Hinton, "Lecture 6.5—rmsprop: Divide the gradient by a running average of its recent magnitude. coursera: Neural networks for machine learning," 2012.

[31] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," 2014. [Online]. Available: https://arxiv.org/abs/1412.6980

[32] J. L. Ba, J. R. Kiros, and G. E. Hinton, "Layer normalization," 2016. [Online]. Available: https://arxiv.org/abs/1607.06450

[33] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: A simple way to prevent neural networks from overfitting," *Journal of Machine Learning Research*, vol. 15, no. 56, pp. 1929–1958, 2014. [Online]. Available: http://jmlr.org/papers/v15/srivastava14a.html

[34] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016, pp. 770–778.

[35] Y. Bengio, “Neural net language models,” *Scholarpedia*, vol. 3, no. 1, p. 3881, 2008. [Online]. Available: https://doi.org/10.4249/scholarpedia.3881

[36] D. M. Harris and S. L. Harris, “Hardware description languages,” in *Digital Design and Computer Architecture*. Elsevier, 2013, p. 129.

[37] D. Jurafsky and J. H. Martin, *Speech and language processing*, ser. Prentice Hall series in artificial intelligence. Upper Saddle River, NJ: Pearson, Jan. 2000.

[38] T. Mikolov, K. Chen, G. Corrado, and J. Dean, “Efficient estimation of word representations in vector space,” 2013. [Online]. Available: https://arxiv.org/abs/1301.3781

[39] T. Mikolov, I. Sutskever, K. Chen, G. Corrado, and J. Dean, “Distributed representations of words and phrases and their compositionality,” 2013. [Online]. Available: https://arxiv.org/abs/1310.4546

[40] L. Torrey and J. Shavlik, “Transfer learning,” in *Handbook of research on machine learning applications and trends: algorithms, methods, and techniques*. IGI global, 2010, pp. 242–264.

[41] A. Karpathy, “The unreasonable effectiveness of recurrent neural networks,” *Andrej Karpathy blog*, vol. 21, p. 23, 2015.

[42] S. Hochreiter and J. Schmidhuber, “Long short-term memory,” *Neural Computation*, vol. 9, no. 8, pp. 1735–1780, Nov. 1997. [Online]. Available: https://doi.org/10.1162/neco.1997.9.8.1735

[43] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, “Attention is all you need,” 2017. [Online]. Available: https://arxiv.org/abs/1706.03762

[44] “indeed “average salaries spain”,” last accessed 2022-06-28. [Online]. Available: https://es.indeed.com/career/salaries

[45] D. Quintillán Quintillán, “Deep learning language models for music analysis and generation,” https://github.com/danielquinti/TFG, 2022.

[46] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Pearson Deutschland GmbH, 1994.

[47] T. Kojima, S. S. Gu, M. Reid, Y. Matsuo, and Y. Iwasawa, “Large language models are zero-shot reasoners,” 2022. [Online]. Available: https://arxiv.org/abs/2205.11916

[48] R. Thoppilan, D. De Freitas, J. Hall, N. Shazeer, A. Kulshreshtha, H.-T. Cheng, A. Jin, T. Bos, L. Baker, Y. Du, Y. Li, H. Lee, H. S. Zheng, A. Ghafouri, M. Menegali, Y. Huang, M. Krikun, D. Lepikhin, J. Qin, D. Chen, Y. Xu, Z. Chen, A. Roberts, M. Bosma, V. Zhao, Y. Zhou, C.-C. Chang, I. Krivokon, W. Rusch, M. Pickett, P. Srinivasan, L. Man, K. Meier-Hellstern, M. R. Morris, T. Doshi, R. D. Santos, T. Duke, J. Soraker, B. Zevenbergen, V. Prabhakaran, M. Diaz, B. Hutchinson, K. Olson, A. Molina, E. Hoffman-John, J. Lee, L. Aroyo, R. Rajakumar, A. Butryna, M. Lamm, V. Kuzmina, J. Fenton, A. Cohen, R. Bernstein, R. Kurzweil, B. Aguera-Arcas, C. Cui, M. Croak, E. Chi, and Q. Le, "Lamda: Language models for dialog applications," 2022. [Online]. Available: https://arxiv.org/abs/2201.08239

[49] A. v. d. Oord, O. Vinyals, and K. Kavukcuoglu, "Neural discrete representation learning," 2017. [Online]. Available: https://arxiv.org/abs/1711.00937

[50] S. Oore, I. Simon, S. Dieleman, D. Eck, and K. Simonyan, "This time with feeling: Learning expressive musical performance," 2018. [Online]. Available: https://arxiv.org/abs/1808.03715

[51] C.-Z. A. Huang, A. Vaswani, J. Uszkoreit, N. Shazeer, I. Simon, C. Hawthorne, A. M. Dai, M. D. Hoffman, M. Dinculescu, and D. Eck, "Music transformer," 2018. [Online]. Available: https://arxiv.org/abs/1809.04281

[52] P. Shaw, J. Uszkoreit, and A. Vaswani, "Self-attention with relative position representations," 2018.

[53] C. Doersch and A. Zisserman, "Multi-task self-supervised visual learning," in *2017 IEEE International Conference on Computer Vision (ICCV)*. IEEE, Oct. 2017. [Online]. Available: https://doi.org/10.1109/iccv.2017.226

[54] N. Theodoros, "Timeseries classification with a transformer model," Jun 2021. [Online]. Available: https://keras.io/examples/timeseries/timeseries_transformer_classification/

[55] I. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio, "Generative adversarial nets," *Advances in neural information processing systems*, vol. 27, 2014.