UNIVERSITATEA POLITEHNICA BUCUREȘTI
FACULTATEA DE AUTOMATICĂ ȘI CALCULATOARE
DEPARTAMENTUL CALCULATOARE

PROIECT DE DIPLOMĂ

TOPIC: Transformator Optic pentru Partituri Interpretate Complet

Ștefan Vodiță

Coordonator științific:
Conf. dr. ing. Irina Mocanu

BUCUREŞTI

2021

UNIVERSITY POLITEHNICA OF BUCHAREST
FACULTY OF AUTOMATIC CONTROL AND COMPUTERS
COMPUTER SCIENCE DEPARTMENT

DIPLOMA PROJECT

SMITE: Sheet Music Interpretation using Transformers (End-to-end)

Ștefan Vodiță

Thesis advisor:
Conf. dr. ing. Irina Mocanu

BUCHAREST

2021

CUPRINS

SINOPSIS

În domeniul recunoașterii optice a muzicii (OMR, din engleză, Optical Music Recognition), au existat dificultăți în ce privește dezvoltarea unei soluții directe și cuprinzătoare care să fie utilă consumatorului final. Metoda propusă preia transformatorul din învățarea automată și îl aplică în transcrierea partiturilor, urmărind să obțină o soluție practică și simplă din punctul de vedere al dezvoltatorului, și să investigheze fezabilitatea unei arhitecturi bazate pe transformatoare în rezolvarea problemei OMR. Modelul prezentat este competitiv, îmbunătățind rezultate precedente în ce privește eficiența, pasul de inferență fiind realizat de trei ori mai repede. Se confirmă astfel utilitatea transformatoarelor pentru probleme OMR și este deschisă o cale pentru cercetări ulterioare.

ABSTRACT

The field of Optical Music Recognition (OMR) has faced difficulties in coming up with a straight-forward comprehensive solution that is useful to the end customer. The proposed method takes the transformer from machine learning and applies it to the problem of transcribing sheet music, aiming to build a solution that is practical and simple from the developer's standpoint, and to investigate the feasibility of a transformer architecture for solving OMR. The developed model is shown to produce competitive results, improving upon previous work in terms of efficiency, with inference being done three times faster. This confirms the usefulness of transformers for OMR tasks and opens a path for future work.

ACKNOWLEDGEMENTS

Many thanks to Conf. dr. ing. Irina Mocanu for her advice as a thesis coordinator on developing this project and paper.

1 INTRODUCTION

This chapter will first make a case for the relevance and the utility of researching and attempting to solve OMR problems. Next, the two contexts that intersect throughout this paper will be introduced: that of sheet music and its recognition from images and that of deep learning, specifically through transformers. Afterwards, the purpose and objectives of this work will be discussed in more detail.

1.1 The Case for OMR

This section will argue for the utility of attempting optical music recognition (OMR) problems. There are two areas where OMR research can be applied:

1. Commercial applications: Ever since the 1990s OMR software has been made available for personal use for non-technical people. Some use cases are for students of music that need help with their sight-reading, or who may find it easier to learn a piece by listening to it and then replicating, instead of relying on the score exclusively. Another use case is for musicians where OMR software could replace missing accompaniment by translating sheet music directly to audio [1]. The amount of OMR applications on the market [2] [3] [4] [5] [6] [7] suggests that demand exists, and current solutions do not satisfy it, sometimes because of their monetization strategy [2] [3], other times because they are only available on some platforms [4] [7], and possibly most often because, as they admit [8], recognition may be impressive on some scores and useless on others, depending on score type, on whether the score is typeset or handwritten, on quality of the image, and myriad other factors.
2. Cultural preservation: There exist large libraries of sheet music that has never been recorded, but the musical content is not accessible directly once those are scanned, and a manual interpretation process would be highly impractical. Thus, a computerized solution is in demand [9].

One should also take note of the multidisciplinary nature of OMR. A success in OMR often entails a positive development in other related computer science or music fields. This is graphically represented through Figure 1. Recent work in OMR (see section 1.2.3 and section2), as well as this paper, will also add machine learning to the number of disciplines involved in OMR.

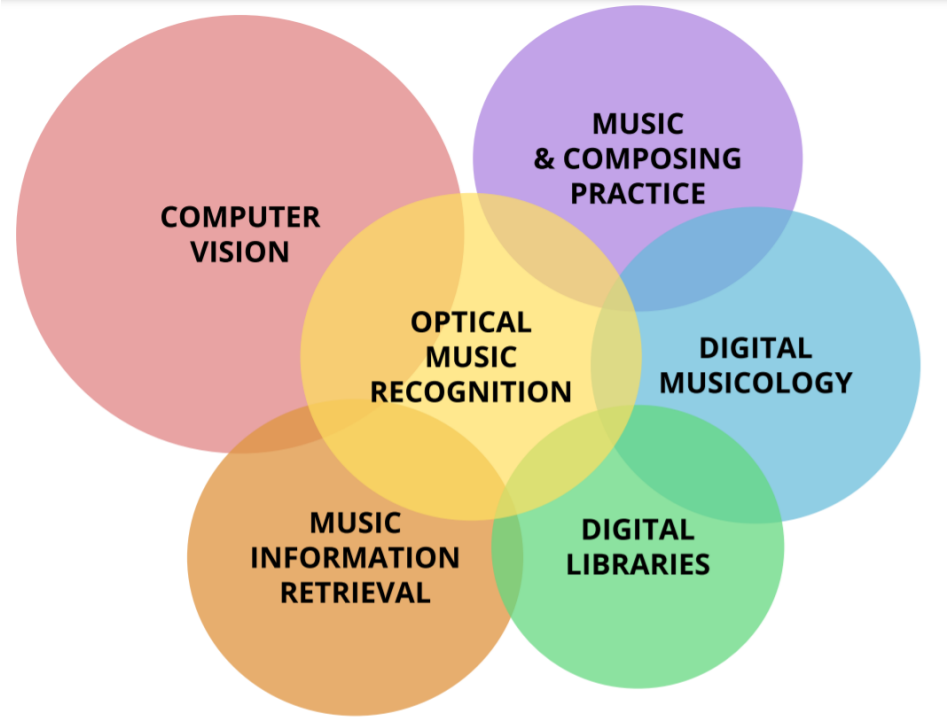

Figure 1 The multidisciplinarity of OMR [1]

Evidently, the arguments presented have already sparked the interest of many in the past, ever since OMR emerged as a field of research in 1966 [10]. If the reader has found their own curiosity piqued, they are invited to read on and strongly encouraged to study the bibliography as well.

1.2 Context

The fields of OMR and deep learning intermingle throughout this paper. As such, it is necessary to introduce the relevant aspects of each of these fields, starting with some music basics, then making an overview of problems in music recognition, and finishing with the machine learning model that is the cornerstone of the solution presented – the transformer.

1.2.1 Basics of Western Classical Music Notation

To follow this paper, the reader will need to first familiarize themselves with some essential principles of music notation but should feel free to skip this section if they are confident in their musical knowledge. One can always come back and reference this section if the need arises.

Western classical music notation refers to the music notation system that started developing in Europe around the year 1000 A.D. and evolved into the system used by the famous classical composers, like Bach, Beethoven, or Mozart. Although historically a feature of European

music, it has, in modern times, grown to achieve international adoption. Most popular music one would hear today can be transcribed in western classical notation.

A prominent characteristic of a musical piece written in western notation is the staff or stave. This consists of 5 parallel, equidistant, horizontal lines. Most likely, a piece will take up multiple staves. The staves are also parallel and equidistant. A note may be written on a staff line, between 2 staff lines, above or below the staff (on what is called a ledger line). The vertical positioning of a note denotes its pitch, with notes written lower on the staff having a lower pitch, and the horizontal positioning of the note placing it in a sequence of musical symbols which are interpreted and played from left to right. In Figure 2, a sequence of 11 notes, in order of increasing pitch is depicted. The first note is on a ledger line, below the staff.

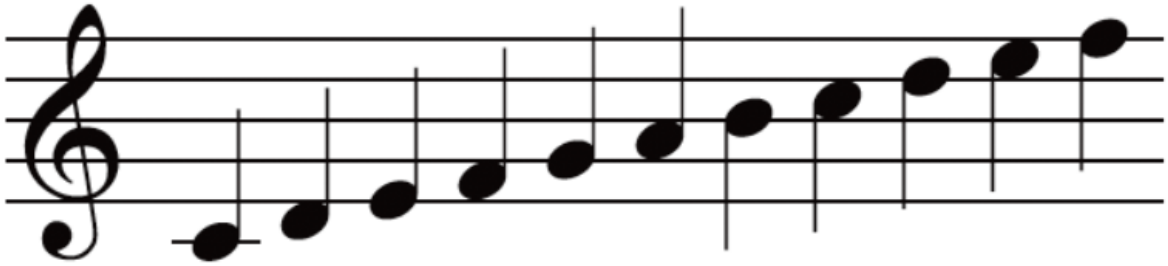

Figure 2 Staff with notes in ascending pitch[1]

Another distinctive characteristic the reader may have already spotted in Figure 2 is the symbol on the leftmost edge of the staff, called a clef. Although the staff is only 5 lines, we can picture it extending indefinitely with more horizontal lines below and above, as the ledger lines suggest (meaning more pitches). In that infinite spectrum of staff lines, the clef marks the 5 lines that will constitute the staff, so it influences the pitch of all notes that follow. For example, in Figure 2, a G clef is shown, which marks the 4th line from top to bottom as the line that G notes will be placed on.

A clef may or may not be accompanied by a key signature. In Figure 2, the key signature is implicit, the default is assumed (C major). In Figure 3, the key signature is explicit. The role of the key signature is to alter the pitch of a specific subset of notes.

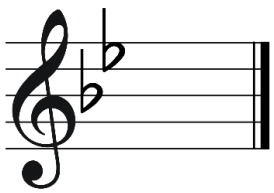

Figure 3 Staff with key signature[2]

[1] © https://www.pianote.com/blog/read-ledger-lines/ [Accessed on 16 06 2021]
[2] © https://en.wikipedia.org/wiki/Key_signature#/media/File:B-flat-major_g-minor.svg [Accessed on 16 06 2021]

Having described the important pitch notations, we move on to duration notation. The clef and key signature that we have seen at the start of the staff can be followed by a time signature. This describes the rhythm of the piece. Figure 4 shows a 4/4 time signature, which is the most common. This notation is similar to a fraction. The bottom number represents a type of note (in this case a fourth, see Figure 6), and the number on top represents the number of notes in a measure (more information below).

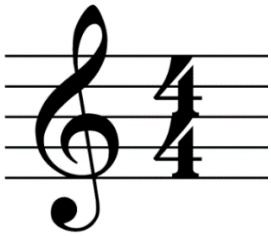

Figure 4 Staff with time signature[3]

After specifying the time signature, the music itself can begin to be described. This will take the shape of a sequence of measures (or bars), which represent units of time, separated by vertical bar lines. Each measure contains notes (times when a pitch is sounded) and rests (times when no pitch is sounded). Figure 5 shows a staff with 4/4 time signature and then 2 measures, each with 4 notes, separated by a vertical bar line. The very last measure of the piece will be followed by a double bar line, as can be seen in Figure 3.

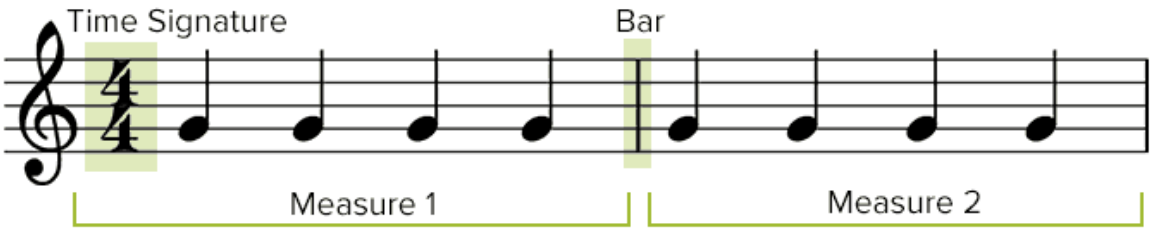

Figure 5 Two measures in 4/4 time[4]

The last thing to mention is that each note is in fact a symbol for a duration of time, different note shapes representing different durations. You can see the most used note shapes in Figure 6.

[3] © http://digitalsoundandmusic.com/3-1-5-musical-notation/ [Accessed on 16 06 2021]
[4] © https://www.musicnotes.com/now/tips/a-complete-guide-to-time-signatures-in-music/ [Accessed on 16 06 2021]

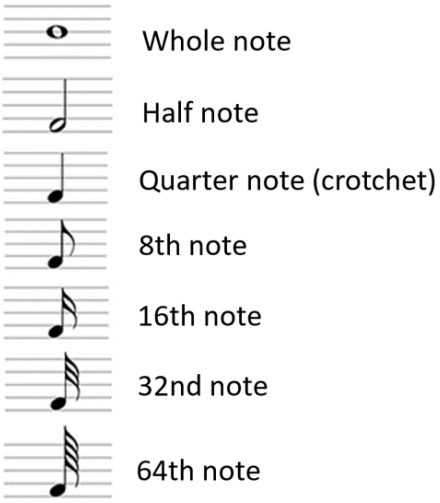

Figure 6 Note shapes and corresponding durations[5]

1.2.2 OMR vs OCR

When looking at Optical Music Recognition (OMR), a natural comparison might be made with Optical Character Recognition (OCR). At first glance, one would be tempted to think both fields address the same problem, with OCR working on text alphabets (for example the Latin alphabet, or the Sanskrit alphabet, or kanji) and OMR working on musical notation alphabets (western classical notation for example).

One difference to consider, as pointed out by Nichols and Byrd [11], is that in an OCR problem all characters are basically equivalent as far as an accuracy measurement is concerned. Misclassifying a letter, whichever one it may be, will have a similar impact. Meanwhile, in OMR, it is not expected for all characters to be equal in this regard. For example, misclassifying a note would have a lower impact on the musicality of the resulting piece than misclassifying a clef, because the clef changes how all notes following it should be interpreted. An OMR software could assign weights to each type of mistake to more accurately estimate its performance [12].

This reveals another difficulty OMR algorithms face – context. If an OCR application can digitize a character in absence of the larger context, say the syllable, or the word (although taking these into account may help), an OMR application can never do the same. Context is crucial in music. To reiterate on the example from earlier, a clef introduces a new context that will modify the meaning of all musical notes appearing after it. Key signatures, time signatures, bar lines can all influence how characters on the staff should be interpreted. This means that a mistake on one of these symbols can propagate and accumulate errors on later symbols. For a correct interpretation to be achieved, several nested variable length contexts must be implemented. The top-level context might be that of the clef, upon which a key

[5] © https://www.simplifyingtheory.com/note-values-in-sheet-music-examples/ [Accessed on 16 06 2021]

signature context is built (but there may be no explicit key signature provided). In parallel, there is a time signature context, and then a measure context which inherits properties from both the clef/key (pitch) context and the time context (nested contexts). A score reader of any kind (human or computer) cannot know the number of measures in the piece until it has reached the end in its sequential reading of measures (variable length contexts). There is no notation that announces the length of the piece from the start. Moreover, each measure can contain from 1 to a theoretically infinite number of musical symbols (variable length context).

Going back to Nichols and Byrd's report [11], there is a third point to be made about OMR. The difficulty of doing recognition on musical scores has high variance, meaning a particular piece of software may have very strong or very disappointing results depending on the type of score it is run on (the example they give is of a choral score versus an idiomatic piano score). Meanwhile, a character recognition program is expected to deliver the same performance whether it is run on a historical or a sci-fi novel (assuming the same typeset).

A fourth difference is in the placement of the symbols [10]. In text languages, the letters are placed along one dimension (left to right in European languages, denoting time), while in western music notation, symbols are placed along 2 dimensions. Left to right denoting time and up or down denoting pitch. While the letter 'a' will always appear at similar positions on the vertical axis, the note Do is not required to do so.

In short, the similarities between OCR and OMR are evident, but we should not think of them both as the same problem. The language structure of music itself poses challenges that are not normally encountered in languages written in text. Perhaps a better way of reasoning through the OCR-OMR distinction is to view OMR as a subset of OCR, inheriting general OCR characteristics, but bringing forth its own particularities.

1.2.3 OMR Approaches and Issues

Ever since 1966 [10], researchers have worked rather chaotically [1] at OMR-related problems. For most of this time, the solutions offered tended to break down the extended OMR problem into sub-tasks that could be more easily approached and evaluated, such as: staff detection, staff removal, symbol detection, symbol classification, musical semantics [13] [14], and even lower level tasks [15]. Because of this focus on sub-tasks and the difficulty of assembling all processing stages into a coherent whole, there are very few complete solutions in the world of research [16].

Evaluation also was a daunting task [10], since, as discussed in 1.2.2, not all mistakes are made equal in OMR, and the impact of a mistake is not self-evident, as it depends on the larger context. There was also a lack of agreed upon datasets, output formats, metrics, and evaluation procedures, so it was difficult to compare results [17].

The approach chosen also had to consider the type of musical score it was to be applied to, and what form the output should take. Some of the questions to be asked would be: Is the

score typeset or handwritten? What era was the score written in [18]? Are there lyrics [19]? Is the score polyphonic? Should the result be MusicXML [20]? A custom encoding? A MIDI file [21]? The step of defining the problem is beautifully illustrated by Calvo-Zaragoza et. al. [1] in Figure 7.

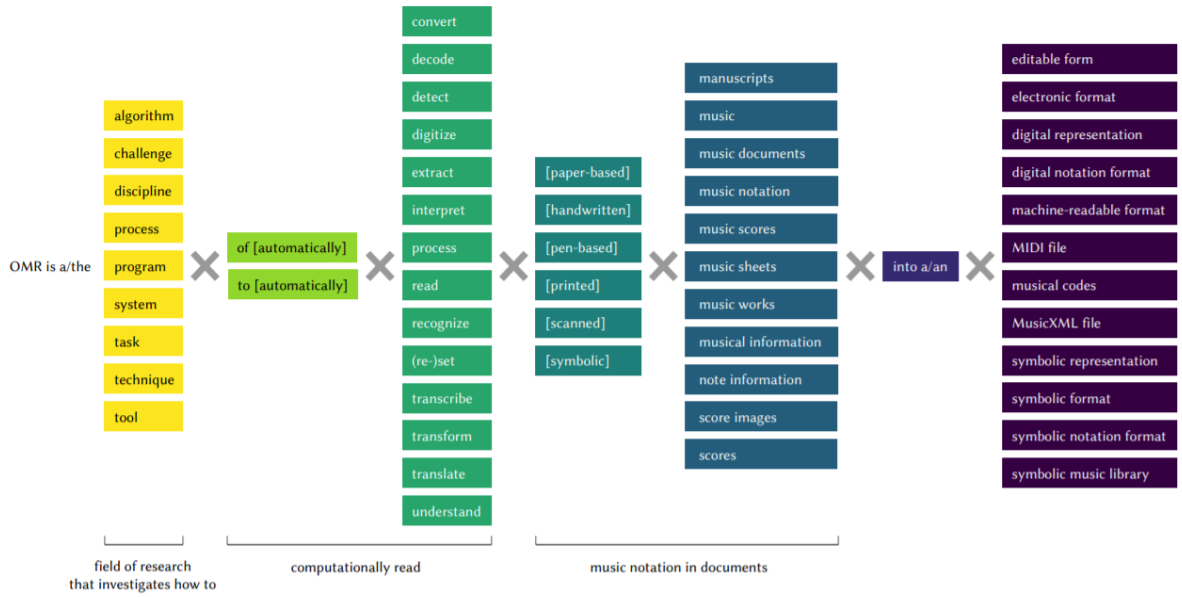

Figure 7 How OMR is defined/understood [1]

Although many problems persist, either because they have not been focused on yet, or because they are intrinsic to music recognition, some are currently being solved. Large public datasets [22] allow for more standardization in evaluation., Meanwhile, advances in machine learning are allowing for a paradigm shift in OMR, with some of the old preprocessing steps [14] now considered solved and getting collapsed into larger stages [1], permitting simpler modular solutions, and even complete end-to-end solutions to become viable [16].

1.2.4 Transformers

The transformer is an attention-based encoder-decoder deep neural network architecture. Its story started in 2017 when the researchers at Google came out with an article titled "*Attention is all you need*" [23]. The innovation that the transformer brought with it was an attention mechanism that could provide context across the entire input, which no longer needed to be processed sequentially, as with a recurrent neural network (RNN, the transformer's natural competitor), but could be done in parallel, speeding up training. Since then, the transformer has suffered multiple iterations and developments [24] [25], is now state of the art on multiple natural language processing tasks [26] [27] and has proven to be a useful tool in other fields too [25] [28] [29]. This paper investigates its applicability to OMR.

1.3 Focus of this Work

This paper will discuss musical notation recognition using transformers on typeset monophonic scores.

The transformer was chosen because of its recent success in sequence-to-sequence tasks, a class of problems that includes the one this paper aims to solve. The advantages of the transformer in natural language processing, in terms of efficiency and context management through the attention mechanism, should still hold for the OMR task being proposed.

The chosen subclass of music scores (typeset monophonic) is one that is easier to manage for recognition tasks but should suffice for a proof of concept. The dataset used is PrIMuS [16], which stands for printed (meaning typeset) images of music staves. The set consists of over 80 000 grayscale images, each of a single staff, with symbols spanning the whole breadth of musical notation. The solution will be evaluated with respect to all musical notation present in the dataset and is expected to show a level of semantic understanding and a grasp of the musical context (e.g.: Current key signature should be considered when interpreting a musical note).

1.4 Objectives

The objectives of this work are:

1. To investigate the feasibility of a transformer-based approach for OMR

   The problem will be viewed as a translation task, from sequential objects in an image to computer-readable music notation in a custom format. The criterion for success will be for performance to surpass that of random guessing.

2. To develop a simple, but comprehensive end-to-end solution for OMR

   The solution should solve the large OMR problem, translating images into musical notation without dividing the work into multiple sub-tasks on a pipeline, as has been customary until recently (see section 1.2.3), but by approaching the problem as one unified task. All musical notation should be interpreted, and it should all be done within the context of the entire score.

1.5 Structure of this Paper

Having prepared the groundwork necessary for understanding the problem at hand, the next sections can now expand on possible solutions:

1. Chapter 2, State of the Art and Related Work, will look at other attempts on similar problems, from the initial involvement of machine learning in OMR, to a recent and novel transformer approach, which provides the perfect comparison for the work presented in this paper.
2. In chapter 3, Proposed Solution, the approach being aimed at will be further concretized and divided into its composing parts.
3. Chapter 4, Application Architecture and Implementation, is concerned with actualizing the ideas from section 3. Particularities of the dataset and its processing, the model's implementation, the metrics being evaluated, the software and hardware that the solution is built upon will all be considered.
4. The purpose of chapter 5, Results, is to present the metrics recorded in a meaningful way, providing comparisons with the work of others.
5. We end with chapter 6, Conclusions and Future Work, which will interpret the results from the previous section and indicate opportunities for improvement.

2 STATE OF THE ART AND RELATED WORK

Asking generally about the state of OMR is underspecified [1] because of the large differences in understanding of what is meant by OMR (as illustrated in section 1.2.3) and because of a lack of agreement on evaluation methods (also in section 1.2.3). Instead, we should ask about a specific OMR task. For the purposes of this paper, we want to know about the state of deep learning end-to-end solutions on monophonic typeset scores.

2.1 RNN on PrIMuS

The first results to be considered should be those presented in "End-to-end Neural Optic Music Recognition of Monophonic Scores" [16] (the very reason for this diploma project's existence). This paper's spot in the timeline of OMR research is early in the development of deep learning for OMR tasks. The authors were interested in investigating whether this approach was promising or not. They did not aim to build the best model for their task, but simply one that would prove the usefulness of neural networks for the OMR field.

Their architecture consisted of a convolutional network meant to learn how to process the input image, which fed its outputs into a recurrent neural network (RNN) that would model the sequential nature of the task. With the training data not being segmented (this would require identifying the musical symbols beforehand) they chose to use connectionist temporal classification [30] for the loss function, which is meant to solve the issue of RNNs requiring segmented data. This is all condensed in Figure 8.

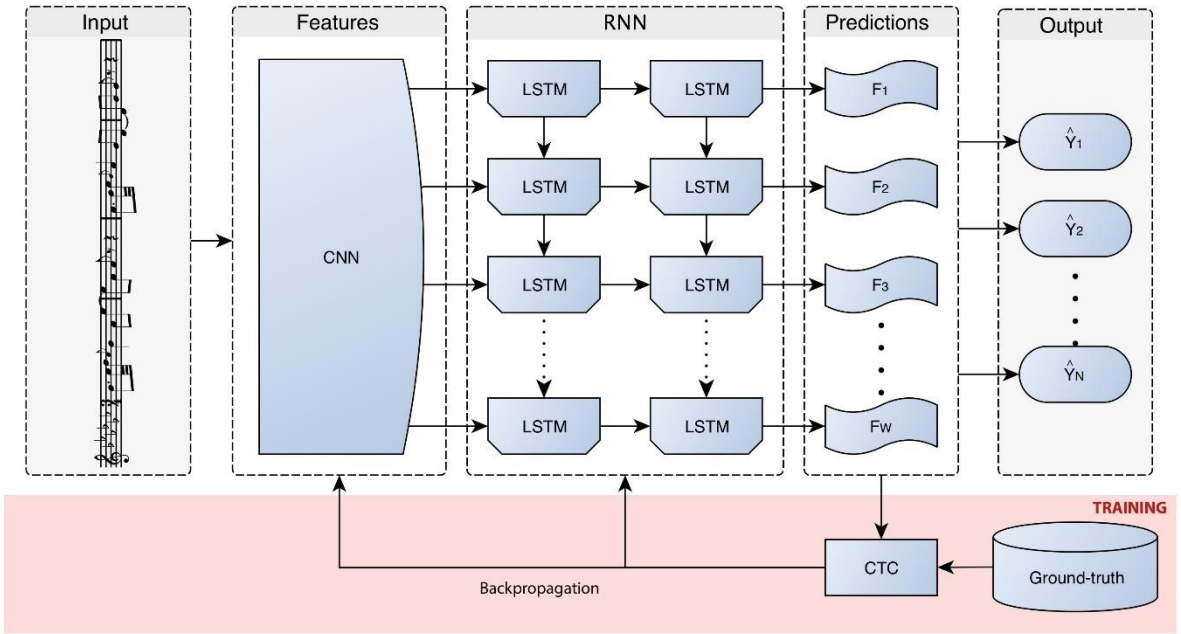

Figure 8 CNN + RNN with CTC loss architecture [16]

They trained their model on a dataset of their own making, PrIMuS (which is also used in this paper), and considered both an agnostic (each symbol is to be interpreted on its own) and a

semantic solution (each symbol is interpreted within the context hierarchy in which it appears; section 1.2.2 explains more about context).

The metrics they reported were the Sequence Error Rate (corresponding to the ratio of samples with at least one error) and the Symbol Error Rate (average number of elementary edit operations (insertions, modifications, or deletions) needed to convert the prediction into the target sequence). Their results are shown in Table 1.

Table 1 Results for agnostic and semantic models [16]

	Representation	
	Agnostic	Semantic
Sequence Error Rate (%)	17.9	12.5
Symbol Error Rate (%)	1.0	0.8

Very interesting is that the semantic model outperforms the agnostic one, which is the opposite of what one would expect, since the semantic model is effectively solving the same problem as the agnostic one with some additional requirements. The authors speculate on this point but give no definitive answer. Accuracy is not calculated, but an estimation is given: "around 1% of the symbols predicted need correction to get the correct transcriptions of the images".

A comparison was made with Audiveris [5], which is a popular OMR tool. Audiveris had a 44.2% symbol error rate and an inference time 15 times longer than the 1 second necessary for the convolutional recurrent neural network. This is unfortunate for Audiveris, since the dataset is a relatively easy one for OMR, but it is possible that more complicated images are where Audiveris can shine.

The authors also looked at the most frequent classification mistakes, first among which is bar line misclassification. They attribute this to samples that do not end with a completed measure, so the model would need a higher level of understanding to decide whether to end the sequence with a bar line or not. It should also be added that the bar line is the most frequent symbol in the dataset, so it is natural for bar lines to have a larger share in the total number of errors. Other frequent errors (though, not close to being as frequent as bar line errors) are produced by grace notes. Grace notes are musical ornamentation graphically represented as a scaled down musical note, sometimes with a slash going through the note. Evidently, these graphical elements may interfere with the recognition task, leading to errors. Grace notes are also much less frequent than normal notes (and other musical symbols for that matter), both in the dataset and in music in general. The model could have trouble learning grace notes because of their rarity. The smaller print symbols in Figure 9 are grace notes.

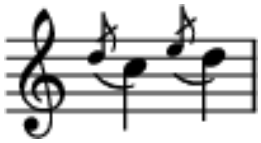

Figure 9 A staff with a clef, two grace notes, and two standard notes[6]

The last relevant topic is that of the dataset's size. As the article [16] shows in Figure 10 and Figure 11, both the symbol error rate and the sequence error rate decrease drastically by adding samples to the dataset up to 10 000 samples. Afterwards, adding more samples does not have as large of an impact. This information proved useful when developing the model presented in this paper (see section 4).

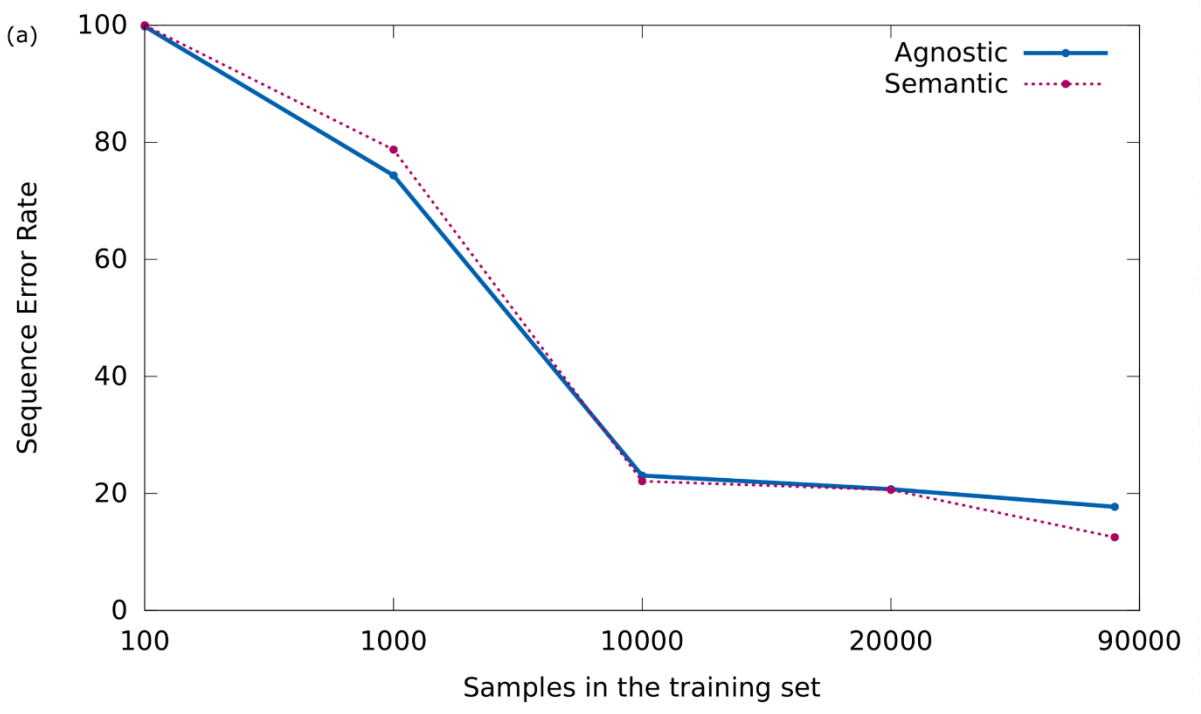

Figure 10 Sequence error rate for CRNN as a factor of dataset size [16]

[6] © https://upload.wikimedia.org/score/8/d/8dfuph84l92irhrknwh6q48l6o6br1x/8dfuph84.png [Accessed 19 06 2021]

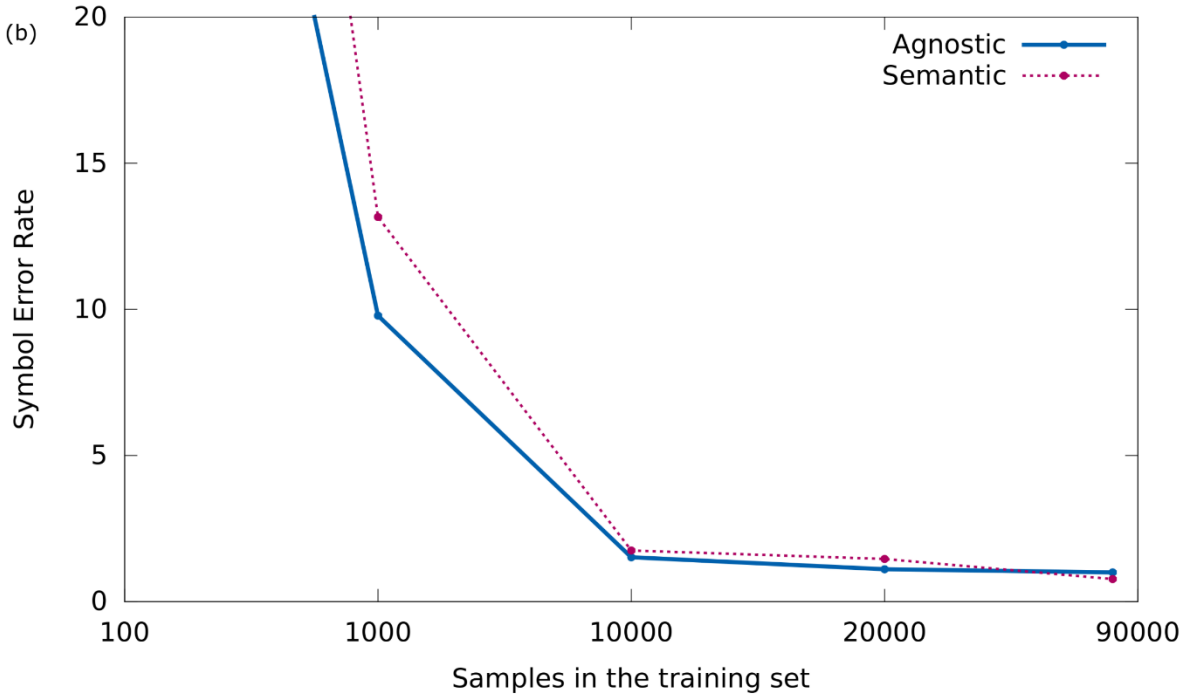

Figure 11 Symbol error rate for CRNN as a factor of dataset size [16]

2.2 RNN on Camera-PrIMuS

The second paper to consider, "Camera-PRIMUS: Neural End-to-end Optical Music Recognition on Realistic Monophonic Scores" [31], comes as a natural development of the first [16]: Now that we know a recurrent neural network is a feasible solution for OMR, does it work in real conditions?

Real conditions in this case meant altering the images of the PrIMuS dataset (created from software) with artifacts that may be present when taking a picture with a camera – this new dataset was deemed Camera-PrIMuS. The model is almost the same as before, only the size of some polling layers having been changed.

Figure 12 shows the results. The model trained on distorted images was able to recognize both clean and distorted images afterwards. This shows that the original model described in the first paper [16] was able to generalize for realistic images without requiring much in the way of architecture changes, although, admittedly, performance has decreased slightly.

		Evaluation			
		Clean		Distortions	
		Agnostic	Semantic	Agnostic	Semantic
Training	Clean	1.1 / 21.7	0.8 / 12.5	44.3 / 94.1	59.7 / 97.9
	Distortions	1.4 / 24.9	3.3 / 44.6	1.6 / 24.7	3.4 / 38.3

Figure 12 Symbol Error Rate (%) / Sequence Error Rate for all training/evaluation combinations on the Camera-PrIMuS dataset [31]

2.3 Transformer on Camera-PrIMuS

The third paper to consider, "Applying Automatic Translation for Optical Music Recognition's Encoding Step" [32], is the only research work on the usage of transformers for OMR as of June 2021. It was published in April 2021, while work on this project was already under way, and the usage of transformers can be taken as some confirmation that our premise is worth investigating. However, the paper does not address the problem of music recognition from an image, but of converting an agnostic sequence obtained from an image to a semantic representation. To some extent, the model devised for this project will have to do this as well, since the final output should show understanding of semantics. Nevertheless, if the model does recognize individual symbols and then adds information from the context, this process would be abstracted away from the developer.

In contrast to the works discussed up until now, 3 machine translation models will be evaluated this time and a comparison made between them. The 3 approaches are:

1. Statistical Machine Translation [33]
2. An encoder-decoder architecture with attention, using a recurrent neural network
3. Transformer

Of these, the 3rd approach is of particular interest here. The authors trained and evaluated their models on 3 datasets, one of which is the aforementioned Camera-PrIMuS [31]. They report a 0.5% symbol error rate for the transformer on Camera-PrIMuS, which outperforms the other models significantly (23.7% for SMT and 2.4% for the RNN). This is shown in Figure 13.

	Camera-PrIMuS	FMT	Zaragoza
SMT	23.7 ± 0.3%	**9.6 ± 3.8%**	69 ± 12%
Seq2Seq-Attn	2.04 ± 0.09%	9.8 ± 3.2%	89.5 ± 1.6%
The Transformer	**0.50 ± 0.06%**	15.3 ± 3.9%	86.3 ± 5.1%

Figure 13 Symbol error rate and standard deviation with 10-fold cross validation for SMT, RNN and transformer models on Camera-PrIMuS, FMT and Zaragoza datasets [32]

To explain the weaker performance of the neural approaches on two of the datasets, the graph in Figure 14 was computed, representing the mean symbol error rate for the SMT and transformer models as a function of the dataset's size. Here, subsets of PrIMuS were used, leading to the conclusion that the reduced performance can be ascribed to insufficient samples in the FMT and Zaragoza datasets.

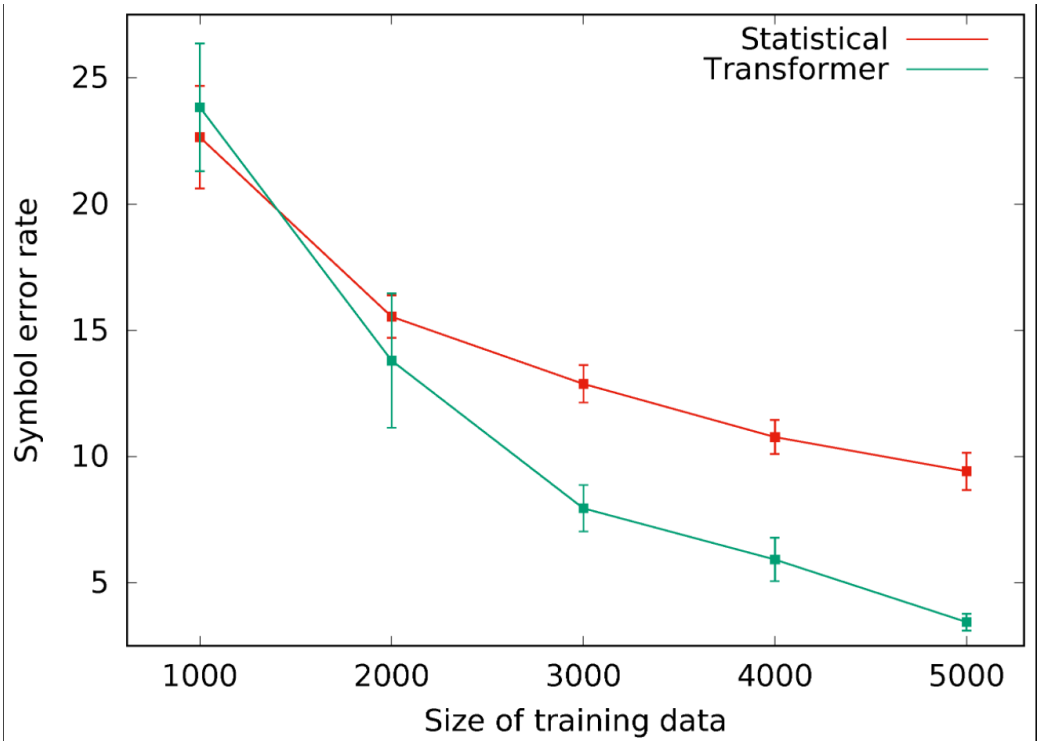

Figure 14 Mean symbol error rate as a function of dataset size for SMT and transformer with standard deviation

The authors conclude that their results should be taken as a baseline for future work and that the transformer is the most promising translation technology for OMR.

3 PROPOSED SOLUTION

To reiterate on points made in sections 1.3 and 1.4, the aim of this project is to develop an end-to-end transformer-based solution for music recognition.

The first decision that needs to be made is on how to encode the predictions. The most straight-forward way will be to follow the example [16] of those that created the dataset and have each musical symbol correspond to an integer. Then, the output for an image will be a sequence of integers representing the symbols on the staff in the image, from left to right. To encode symbols, a dictionary can be used, with strings representing the symbol names as keys and the encodings as values. For decoding, a list of strings is sufficient, where the name of the symbol is found at an index equal to its encoding.

The second decision is the model to be used for the learning task. The transformer is most often found in language processing, translating one sequence to another, but this work requires a transformer that acts on images. One of the sequences in the translation is the musical sequence in encoded form. This is the output. It follows that the input image needs to be formatted as a sequence, which can be done by cutting the image into patches. The resulting sequence of image patches is a viable input format. Through this operation, the task has been modeled as a sequence-to-sequence problem, translating a list of image patches into a list of integers (the symbol encodings). A transformer which uses the input scheme described above has already been developed [25], but its use is in classification tasks. This shortcoming can be bypassed by replacing the last layers in the transformer with ones more favorable for the problem at hand.

Passed through the transformer, each image outputs a matrix of probabilities P where P[i][j] is the probability that the musical symbol with index i in the image (from left to right) corresponds to the symbol whose encoding is j. An alternative formulation would be that P[i][j] is the probability that j is the encoding of the i^{th} symbol on the staff. This will allow experimenting with both a connectionist temporal classification (CTC) loss function [30] and a cross entropy loss function. The CTC loss function will format the output as subsequences of repeating symbols, interspersed with blank symbols. This effectively doubles the output size, because of the space required for blanks used as delimitation between symbols. However, the entire output will be filled without respect to the length of the input sequence. Meanwhile, for a cross entropy loss implementation, the model needs to learn how to mark the end of the sequence. The part of the output following that mark will be wasted.

Figure 15 shows the architecture of the proposed solution. The sample shown in the upper left, representing one staff of music, goes into a preprocessing step, where the image can be resized, normalized, and cut into consecutive patches. Then, the processed image, now in the form of a sequence of image patches, passes through the transformer, which outputs the matrix of probabilities mentioned above. Afterwards, the output can be interpreted, and the final prediction is reached, a sequence of musical symbols.

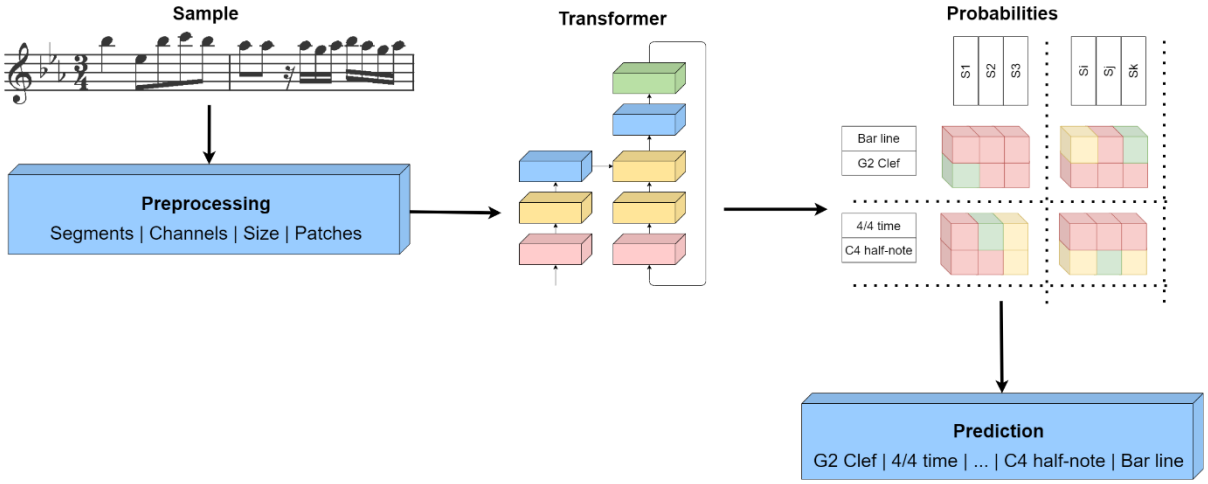

Figure 15 Overall structure of the proposed solution

4 APPLICATION ARCHITECTURE AND IMPLEMENTATION

The sections that follow will describe implementation details of the solution's different aspects, starting from the inputs, moving on to the model, then looking at how loss and other metrics are calculated, ending with a section on the specific libraries needed to run the solution, and some information about the hardware used. There will also be some discussion on various attempts at improvements and optimization.

4.1 Dataset

As already mentioned, the dataset to be used is PrIMuS [16]. This consists of monophonic single-staff grayscale images of real music incipits. 78 755 such samples were made available.

The targets' symbols are encoded as integers through a predefined association. For example, 0 stands for a bar line, 10 stands for the G2 clef, 1779 stands for common time and 790 stands for a C4 quarter note. Thus, 10 1779 790 790 790 790 0 would be a correct sequence describing a full bar of C4 quarter notes, in common time, in C major. There are 1781 symbols in total in the dataset's vocabulary. As stated above, the samples are of real music, so the symbols are not distributed evenly. There are symbols with just one appearance in the dataset, and more symbols with only a few appearances.

An example from the dataset is shown in Figure 16. The target is human-readable:

clef-G2
keySignature-EbM
timeSignature-3/4
note-Bb5_quarter
note-Eb5_eighth
note-Bb5_eighth
note-C6_eighth
note-Bb5_eighth
barline
note-Ab5_eighth
note-Ab5_eighth
rest-sixteenth
note-Ab5_sixteenth
note-G5_sixteenth
note-Ab5_sixteenth
note-Bb5_sixteenth
note-Ab5_sixteenth
note-G5_sixteenth
note-Ab5_sixteenth
barline

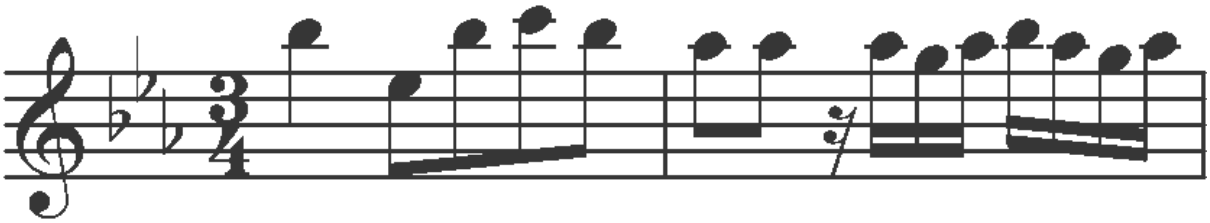

Figure 16 First sample from PrIMuS [16]

The length of the musical sequence in each image is relevant when setting the model's output size, since the output size must always be at least as large as the target's length. All sample sequence lengths are plotted in Figure 17. The mean length is 23.8928 with a standard deviation of 5.9585 and the maximum length is 58 (not visible in the figure, since there is only one sample that long).

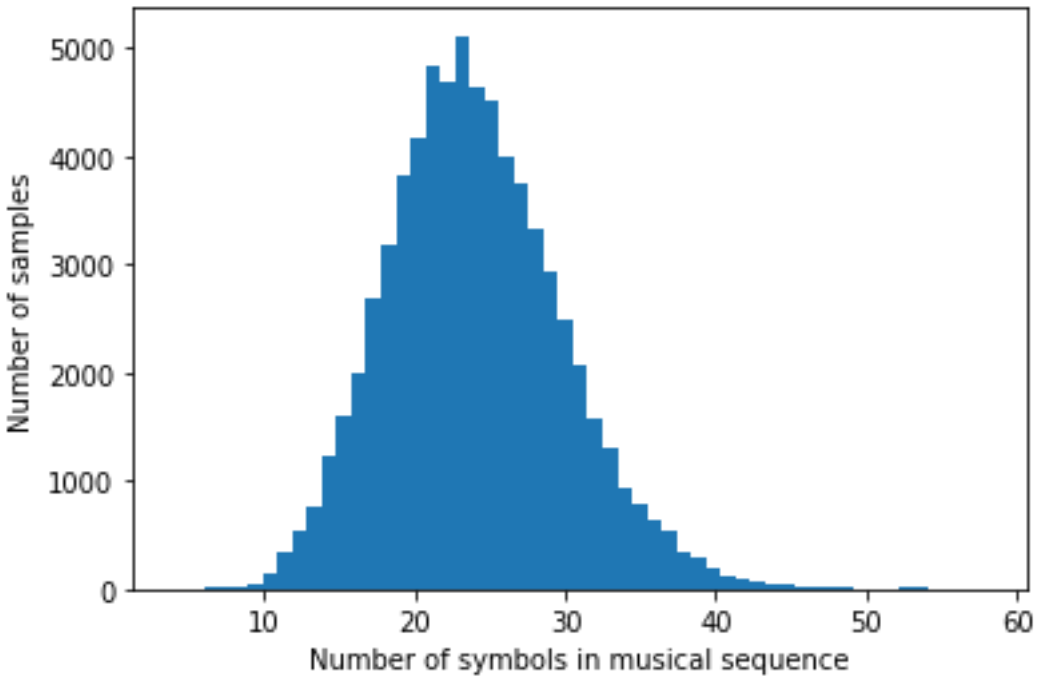

Figure 17 Histogram of the length of musical sequences in the dataset

Image width and height are also important for the preprocessing step because all images must be resized to the same dimensions. The heights are shown in Figure 18 and the widths in Figure 19. There is more variance in width than in height, and a significant proportion of samples (approximately 1/7) have widths of around 2000 pixels, while most of the other samples are distributed in the width region of 1000 to 1500 pixels.

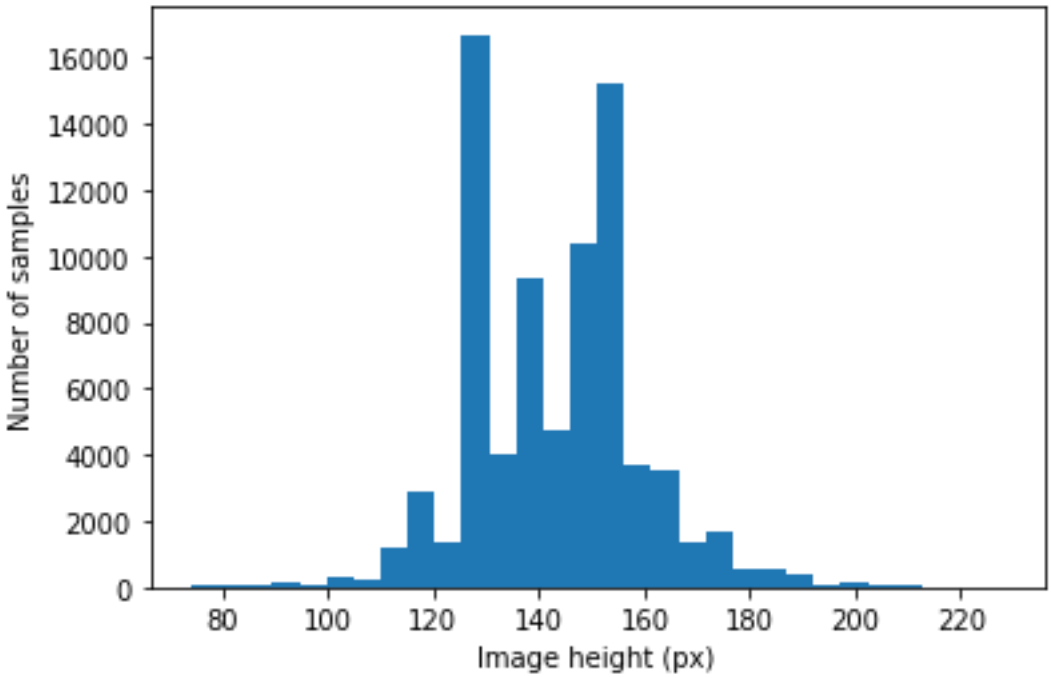

Figure 18 Histogram of the height of images in the dataset

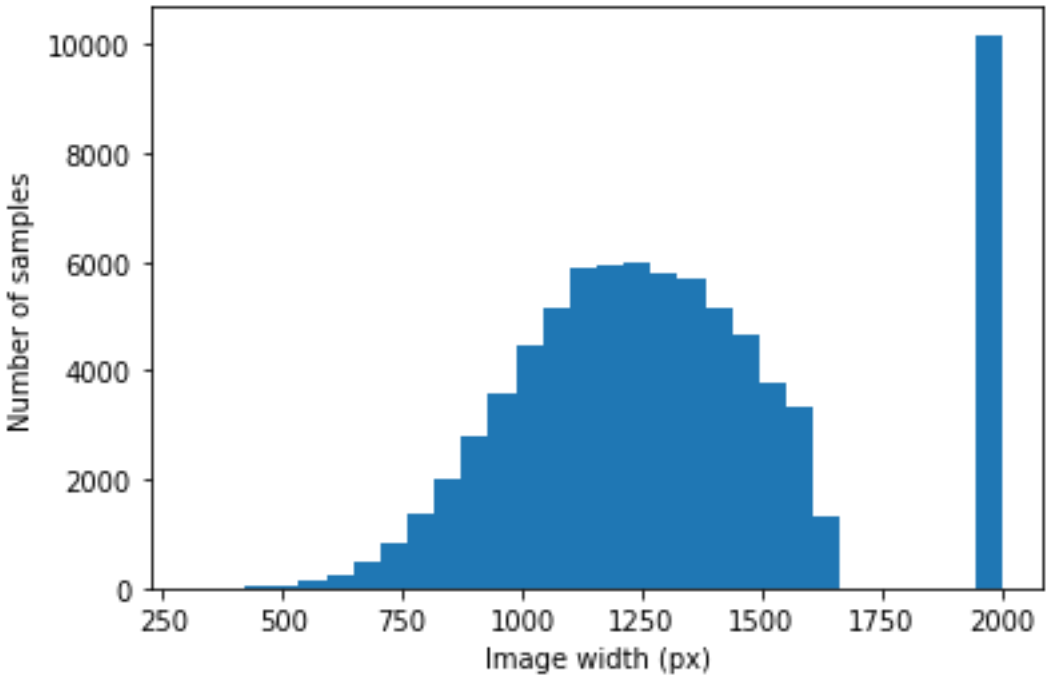

Figure 19 Histogram of the width of images in the dataset

4.2 Application Architecture

Figure 20 shows the principal components of the implementation, each of which is explained and discussed in a subsequent section (except for some aspects which have to do with the

dataset and have already been discussed in section 4.1). The components concerned with image preprocessing are depicted in blue. These are, in order of operation:

1. Segmentation: Cutting the images and reattaching the resulting segments – section 4.3.1
2. Channels: PrIMuS images are grayscale, but the model expects RGB images – section 4.3.2
3. Normalization / Resizing / Patches grouped under the feature extractor: Images are further processed to ensure successful learning; cutting the image into patches creates the sequence-shaped input that is required – section 4.3.3

The components which refer to metric computation and collection are the ones in red:

1. Loss – section 4.5
2. Accuracy – sections 4.7.1 and 4.7.2
3. Levenshtein distance (normalized) – section 4.7.3
4. Run time – section 4.7.4

In addition to the listed components, the optimizer (section 4.6) and aspects of the model (section 4.7) will also be discussed.

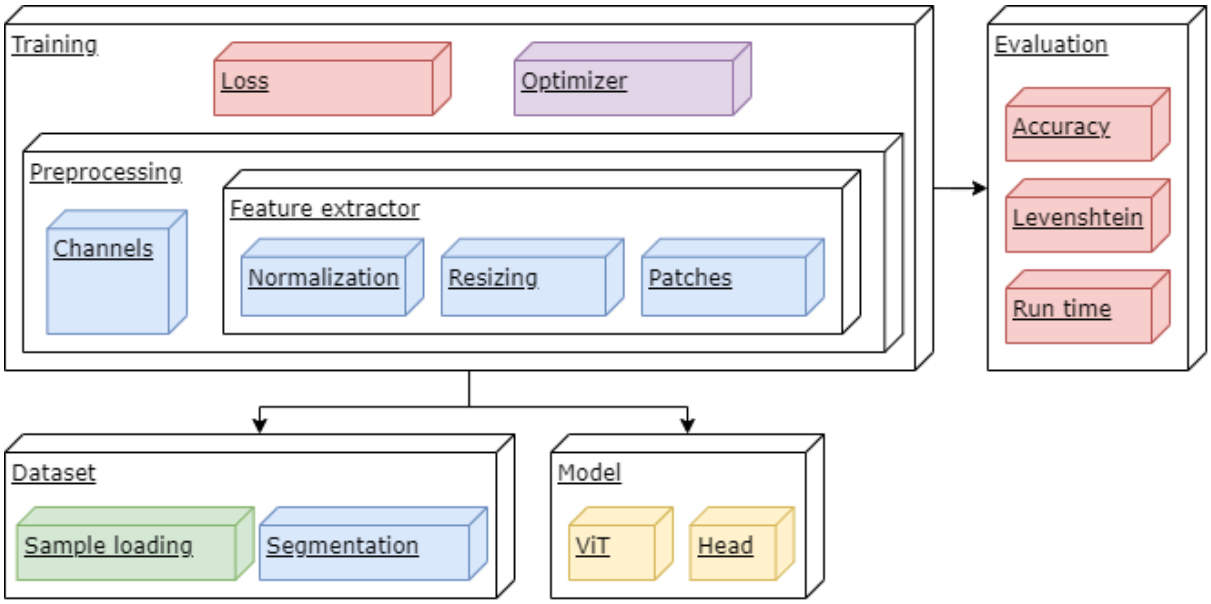

Figure 20 Block diagram of conceptual components in the solution

4.3 Preprocessing

This section concerns all the operations needed to make a sample image acceptable to the model, the most involved of which is the segmentation and subsequent reshaping of the image.

4.3.1 Image Segmentation

The title of this section should not be taken to mean that objects will be extracted from the image. In this case, segmentation only refers to a part of the process of resizing the samples. Since the images are each of a single staff of real music their dimensions are not consistent but tend to skew towards a short and wide aspect ratio, with heights around 150 pixels and widths around 1200 pixels, sometimes going over 2000 pixels, or down below 500 (width and height distributions are plotted in section 4.1). This size inconsistency is even more of a problem when we consider that the model runs on square images (see section 4.4 for a discussion of the model). If the images were resized directly, information could be lost, especially in the case of tall narrow symbols like bar lines or note stems. To prevent this, the image is cut into several horizontally aligned sequences, which are then stacked vertically. This brings the image closer to a square shape, while conserving features in the image.

The process is illustrated below. Figure 21 shows a complete, unaltered image from the dataset. Figure 22 shows the same sample cut into 3 segments, and Figure 23 shows the 3 segments rearranged vertically, forming a more desirable shape. Reading the score in Figure 23 would still be intuitive for a human, reading each row from left to right, starting from the top row.

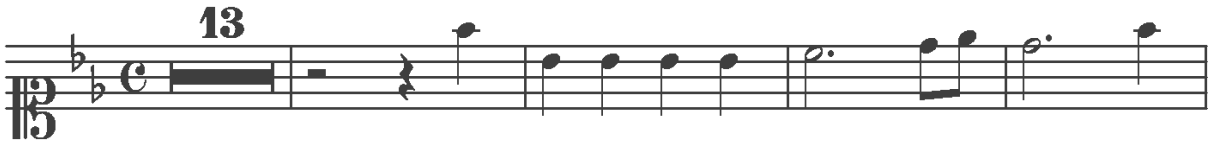

Figure 21 A sample image from PrIMuS

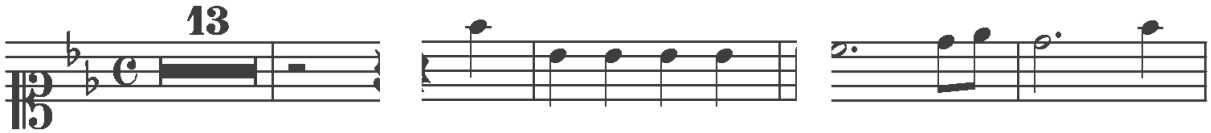

Figure 22 The staff from Figure 21 segmented horizontally

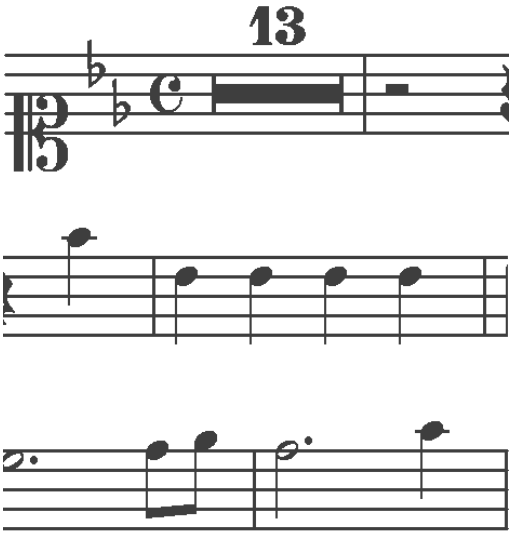

Figure 23 The segmented staff from Figure 22 stacked vertically

The perceptive reader may have noticed that some symbols can be cut through during this process. In the example above, a quarter rest was split in two by the first cut and a note was separated from its stem by the second. We might expect this to lead the model to recognize no symbols in place of the one that has been cut through, or to recognize two symbols instead. To counteract this issue, an algorithm was devised that would only cut through places where the staff is empty:

```
1 find N – 1 equidistant cuts
2 for each cut:
3         while cut would go through a non-empty staff:
4                 move cut one pixel to the right
```

The empty staff is identified per image (since each image can be scaled differently) by looking for a repeating column of pixels, because symbols tend to not have identical columns one after another, while the staff is always consistent.

The proposed algorithm is very simple and can be intuitively described as follows: If a cut would go through a symbol, move that cut to the right until the symbol is passed.

However, doing this creates a new problem. To showcase this problem, a new example is taken, shown in Figure 24. This time we will use 2 segments, just to better illustrate the point at hand. Figure 25 shows the cuts that would have been made if symbols were not avoided, and Figure 26 shows the cuts made while avoiding symbols.

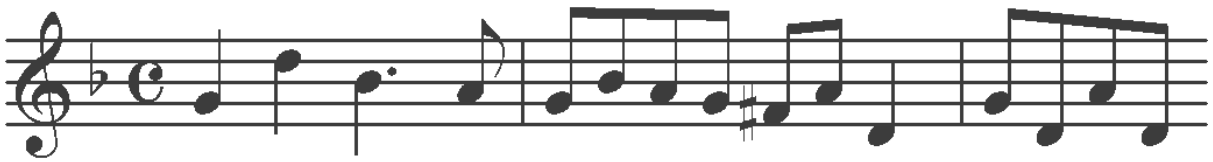

Figure 24 A sample image from PrIMuS with beamed notes

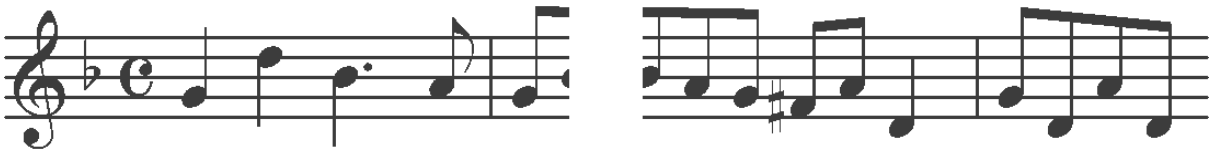

Figure 25 Horizontal segmentation of the image from Figure 24, without avoiding symbols

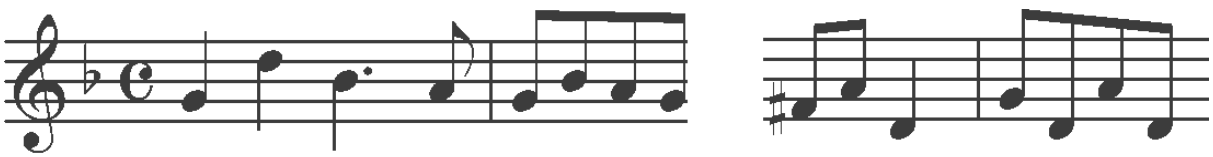

Figure 26 Horizontal segmentation of the image from Figure 24, avoiding symbols

If the two segments are now stacked as previously described, we get the result depicted in Figure 27.

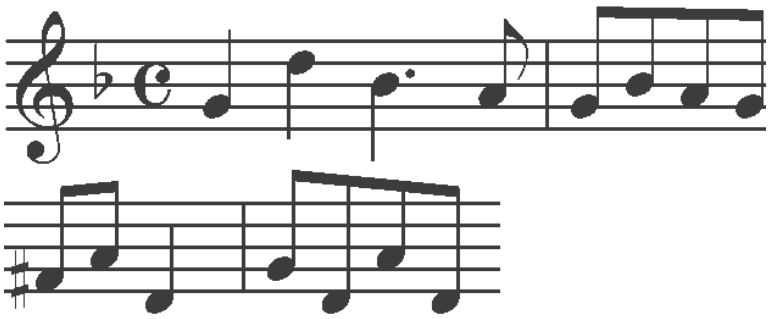

Figure 27 The staff in Figure 24, after reshaping

One could argue that the blank space in the lower right corner of the image is undesirable. In that case, the end of the second row can be padded with empty staff without altering the musical interpretation of the sequence. This is exemplified in Figure 28.

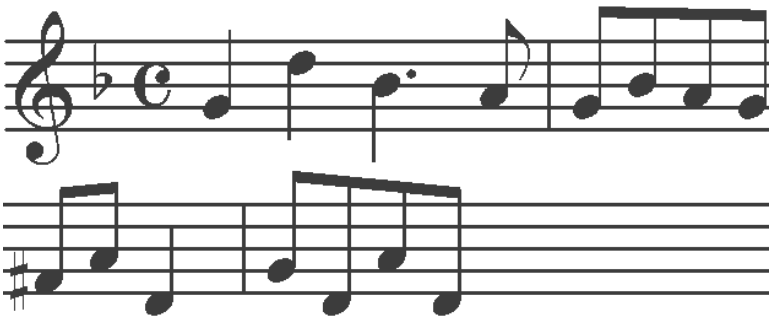

Figure 28 The staff in Figure 24, after reshaping and padding

The new problem is that the image has effectively been made larger by filling it with blank space devoid of musical information that will still need to be processed by the model. This will happen to a certain degree whenever there are beamed notes or horizontally close symbols on a cut. If we had gone left instead of right in the example above, we would have gotten more even segments, but, obviously, this is not always the case. The algorithm could also just choose whichever side is closest, but this can lead to even worse results, by having a middle segment encroached upon from both sides. Padding with empty staff does not ameliorate the problem – results were worse than just leaving the spot as whitespace, albeit not significantly worse.

Figure 29 and Figure 30 plot the accuracy and loss of a run where segments are cut while avoiding symbols (clean) and a run where segments are cut without concern for what may be cut through (dirty). Both runs are using 3 segments per image. A similar case can be observed in Figure 31 and Figure 32. The difference is that this time, 6 segments were employed for each image. In both cases, loss and accuracy improve less over time for clean segmentation.

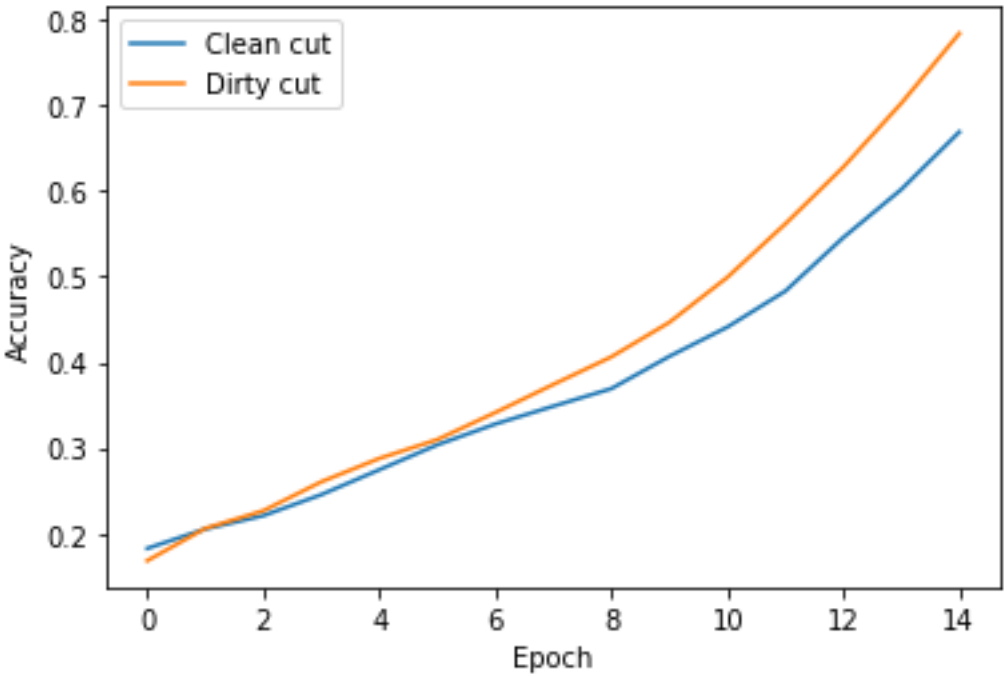

Figure 29 Accuracy for clean and dirty cuts on 3 segments

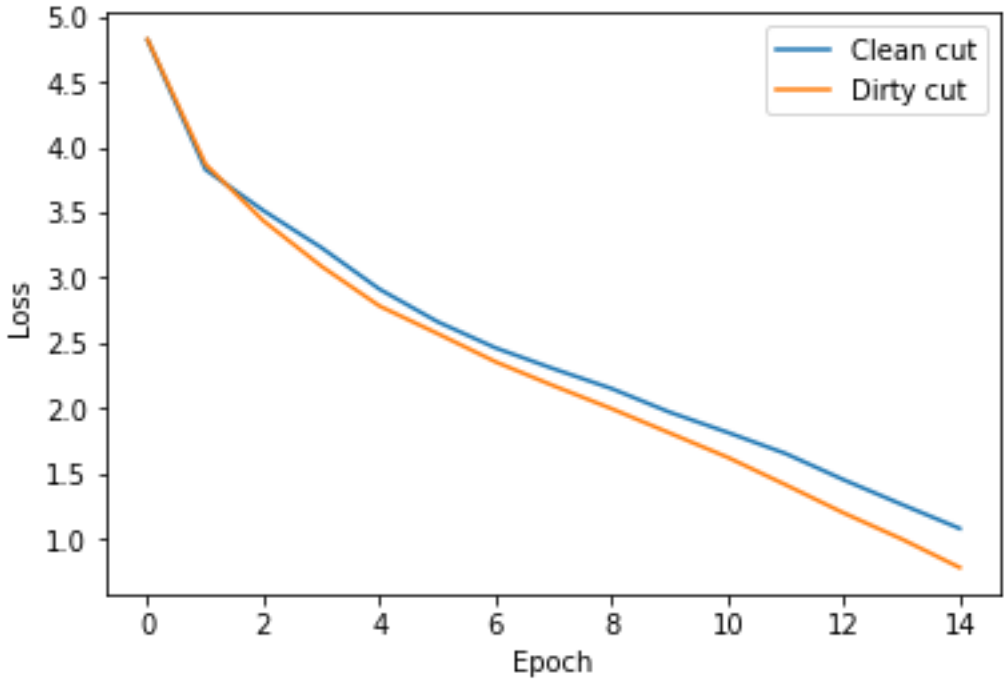

Figure 30 Loss for clean and dirty cuts on 3 segments

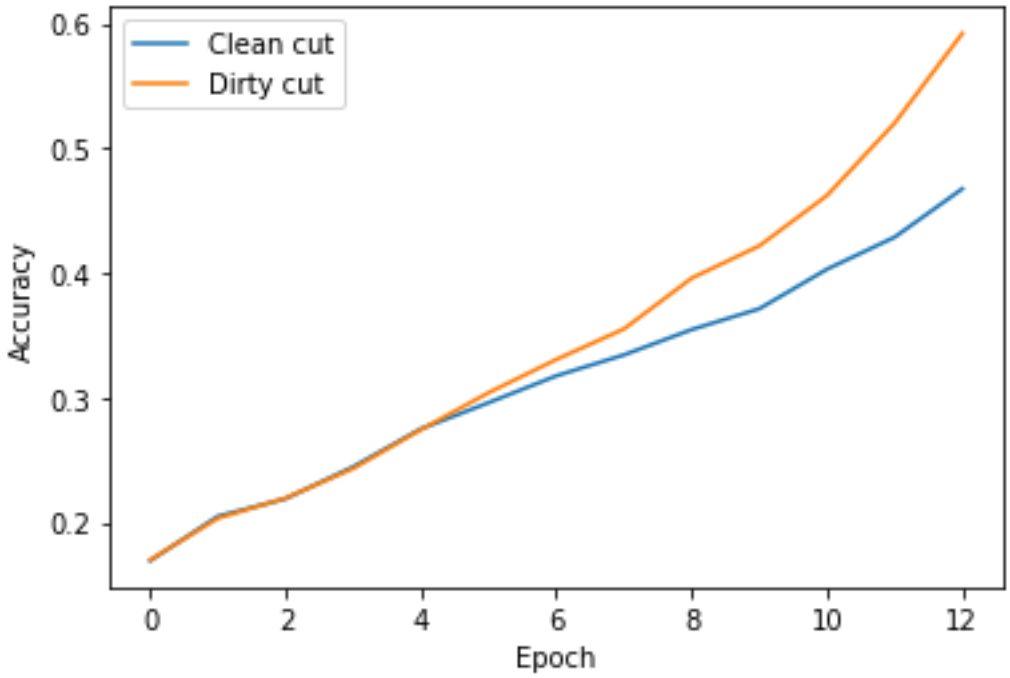

Figure 31 Accuracy for clean and dirty cuts on 6 segments

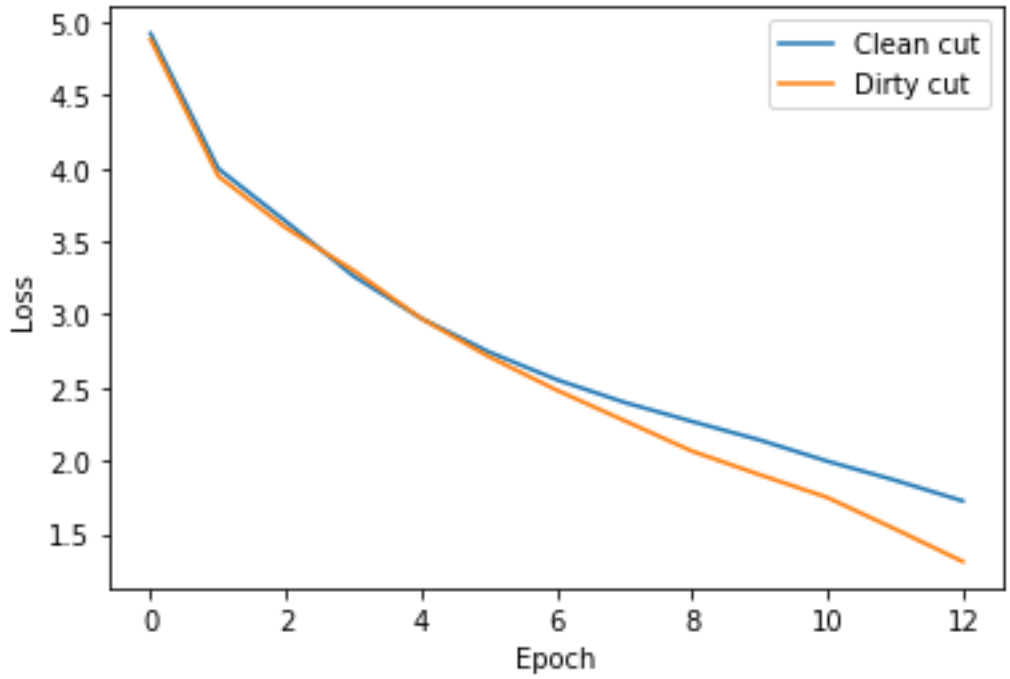

Figure 32 Loss for clean and dirty cuts on 6 segments

Overall, learning went worse when avoiding cutting through symbols, so the idea was scrapped, and images are cut into roughly equal segments. They are not exactly equal in all cases, since an image's width might not be divisible by the number of segments we want to obtain, but the difference is no more than one pixel in width.

Having settled on how segmentation should be done, the next question is on the number of segments. Examples with 2 (Figure 27) and 3 (Figure 23) segments have been shown and one can see how the image split into 3 had achieved a better shape at the end. The optimal number of cuts will need to balance two goals:

1. Bringing images as close as possible to a square shape
2. Doing 1. for as many images in the dataset as possible

An empirical search for this number ensued. Figure 33 plots the training accuracy and Figure 34 the loss obtained with the model when images are cut into a different number of segments. The following two lines, where the number in parenthesis symbolizes the number of segments, summarize the results after the last epoch:

Accuracy(2) < Accuracy(1) < Accuracy(4) < Accuracy(6) < Accuracy(5) < Accuracy(3)

Loss(1) > Loss (2) > Loss (4) > Loss (6) > Loss (5) > Loss (3)

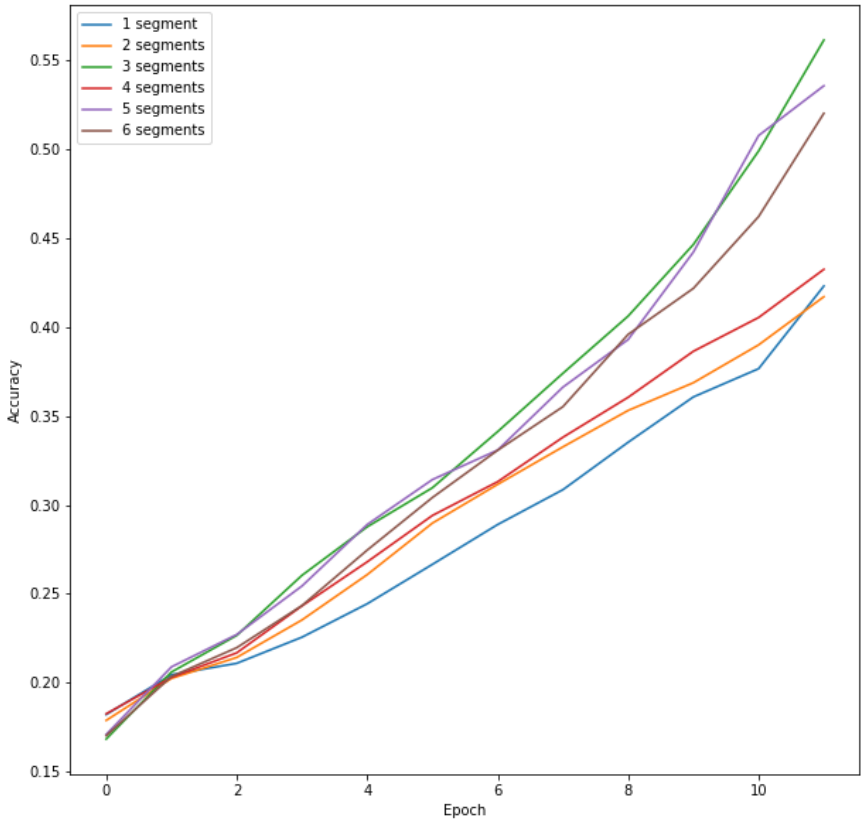

Figure 33 Accuracy for 1 to 6 segments

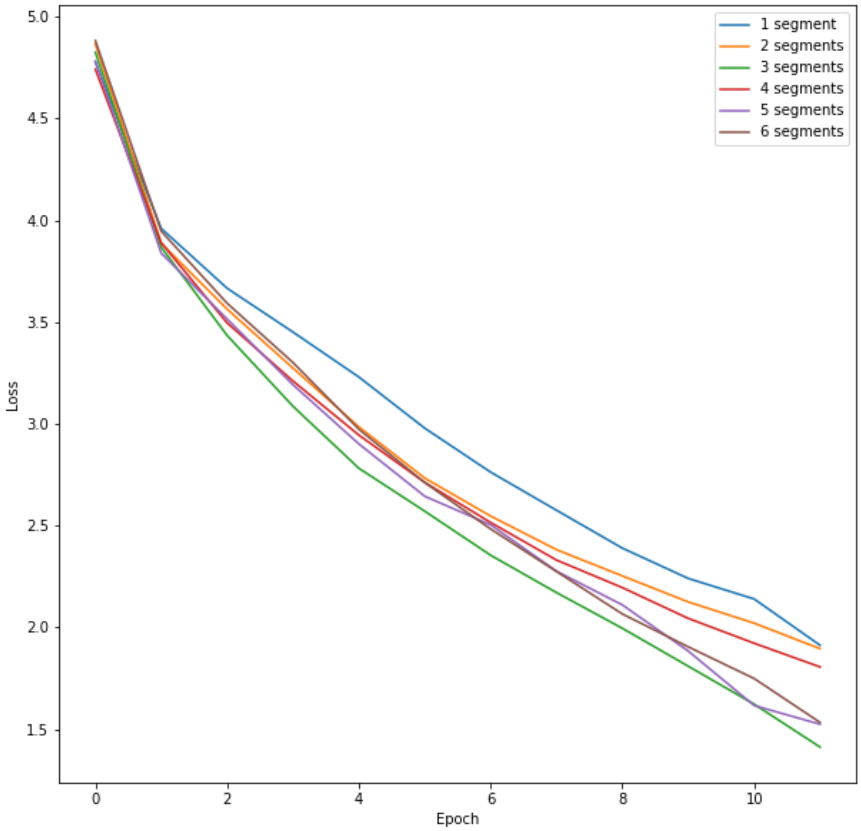

Figure 34 Loss for 1 to 6 segments

Many factors come into play here. To investigate further, the top 2 performers have been run for a longer time and plotted again. This is shown in Figure 35 and Figure 36. The 5-segment implementation ends up overtaking the 3-segment one, becoming the best performer by a small margin. This could indicate that the model prefers slightly tall images that it can expand horizontally, since most musical symbols are tall and thin, so horizontal expansion could ensure that information does not get lost, by making it more prominent (e.g.: A stem or a bar line gets thickened). From here on, the 5-segment method is employed.

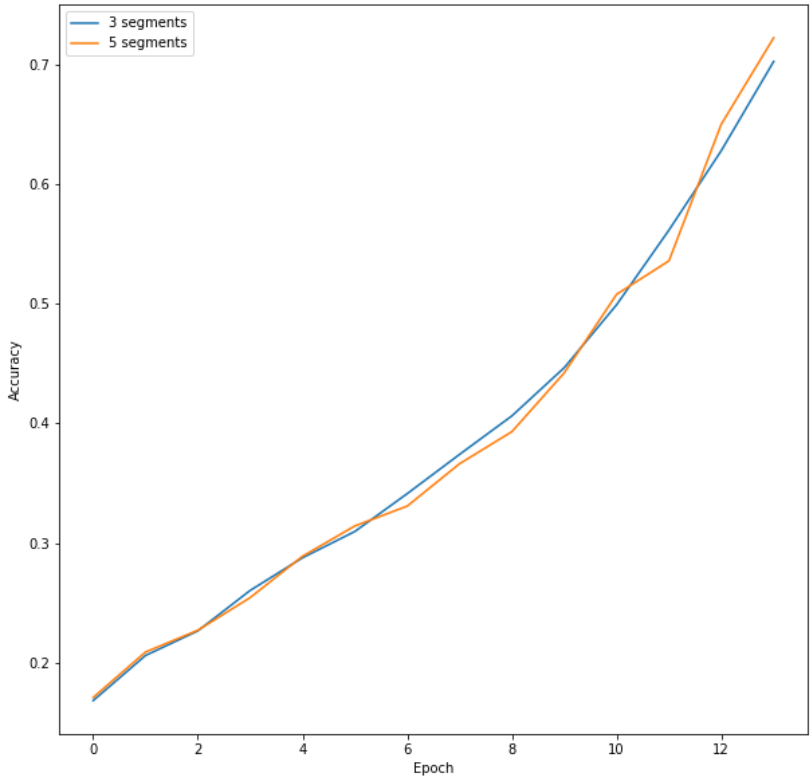

Figure 35 Accuracy over time for 3 and 5 segments

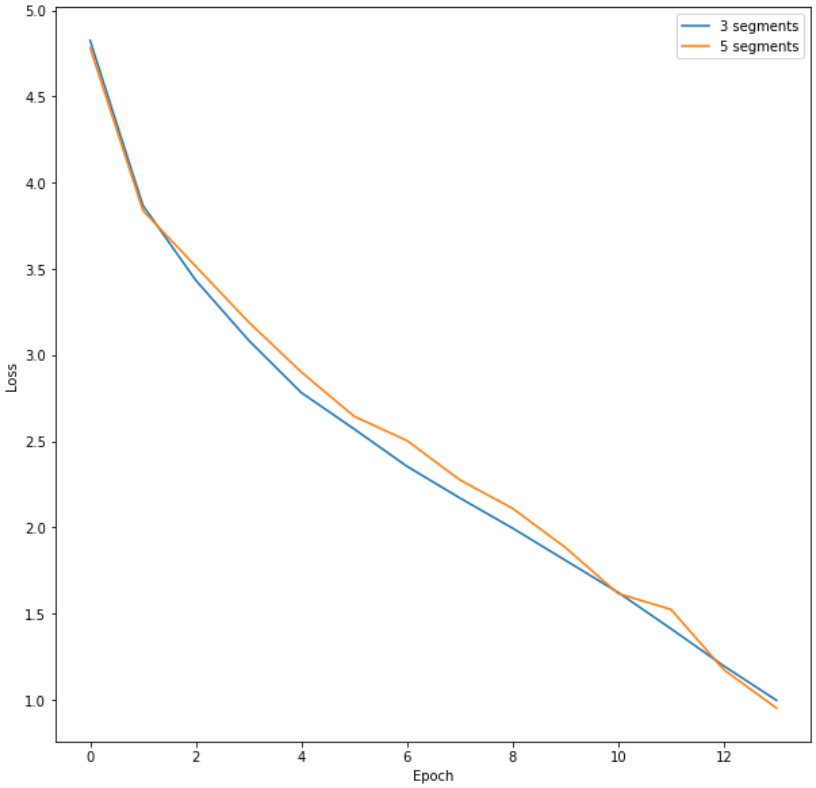

Figure 36 Loss over time for 3 and 5 segments

4.3.2 RGB Channels

The model used (see section 4.4) requires 3-channel RGB images, but PrIMuS [16] only has grayscale images. To get around this, the 3 channels were all filled with the same data from the single channel of the grayscale image. A different approach was also considered, where the model would be preceded by a convolutional layer that would learn what the 3 channels should be. This increased complexity without showing any tangible benefits, so the idea was discarded.

4.3.3 The Feature Extractor

Together with the vision transformer, the Hugging Face library [34] also makes available a feature extractor that works in conjunction with the transformer. The name may be a bit misleading; the feature extractor resizes the image to the dimensions taken in by the transformer, it normalizes the image, and it separates the image into patches (more on this in section 4.4).

4.4 Model

Initial attempts at developing a transformer-based model had slow learning speeds. To combat this, and speed up development, a pre-trained vision transformer (ViT) was employed [25]. The classification layer was replaced so the output would be shaped into a matrix of probabilities, as described in section 3. This configuration appended after the transformer will be referred to as the head of the model. Together, the two parts have 87 771 486 learnable parameters. The complete model is available in the appendix.

4.4.1 Vision Transformer

ViT [25] works on 3-channeled square images. It processes the image in square consecutive patches, which are flattened and passed through the original transformer [23].

4.4.2 Model Size

The developers of ViT have investigated, trained, and made public multiple versions [35] of ViT, trained on different datasets (ImageNet-21k, JFT-300M), different image sizes (224x224, 384x384), different patch sizes (14x14, 16x16, 32x32), and with different architecture complexity (base, large, huge). All these parameters were varied one by one and the most successful learning in terms of speed and minimum loss obtained was done by the base model with image size 224x224 and patch size of 16x16. This is also the model that the researchers had the best results with. It is denoted at the link above as *google/vit-base-patch16-224*.

4.4.3 Head

The head needs to reshape the 197x768 output of the ViT (this shape changes with the size of the ViT; see section 4.4.2 for ViT sizes) into a *NUM_TOKENS* x *VOC_SIZE* tensor, where *NUM_TOKENS* is the maximum length of a sequence (i.e., the maximum number of musical symbols in an image) and *VOC_SIZE* is the size of the musical vocabulary (i.e., the total number of symbol types in the dataset (1781) plus added symbols, like blanks or symbols marking the end of the sequence (more about these in section 4.4.4)).

This was achieved with a linear layer and a one-dimensional convolutional layer. The kernel size of the convolutional layer was tested for values 1, 3, 5, 7, and 9, each with corresponding padding. There were no significant differences. This is expected, as the bulk of the learning should take place in the ViT, and the last layer serves only to reshape the output.

Another option was to use a pooling layer instead of the convolutional layer. This slowed down learning and the idea was discarded.

4.4.4 Biased Predictions

Since all samples in the dataset are incipits, they all start with a clef, an optional key signature, and a time signature. The setup was altered to only make predictions for the first symbol from the vocabulary of clefs and only make predictions for the second symbol from the vocabulary of key signatures and time signatures. Neither of these changes influenced the accuracy of the model. This means that the model learns on its own to predict clefs, key signatures, and time signatures at the start of sequences.

4.5 Loss

Two approaches to loss measurement were investigated, one with cross entropy loss, which showed some benefits, and another using connectionist temporal classification (CTC) loss [30], as advocated by previously discussed papers [16] [31] [32].

4.5.1 Connectionist Temporal Classification Loss

CTC addresses the problem of unsegmented input data. In our case, segmenting the data would have meant identifying the musical symbols in the image in a separate step, and then passing those (perhaps as patches taken from the image) to a model that would classify them. The first step would not have been a simple one, and not as straightforward to separate from the second step as might seem from the description, for example in the case of multiple symbols that are on the same vertical axis (like a tie and a note), or symbols that are not graphically well separated (beamed notes, time signatures, handwritten symbols). Doing separate identification and classification steps would also not conform to our goal of having a simple end-to-end solution without a pipeline. These are the reasons CTC has been chosen

in the past for music recognition with RNNs [16] [31], and more recently with transformers [32].

For the implementation, using CTC loss means that the model's predictions will be structured as a list of repeating symbols, interspersed with blanks:

S1, …, S1, BLANK, S2, … S2, BLANK, …

S1 and *S2* from above are musical symbols encoded as described in section 4.1. *BLANK* is a symbol added to the alphabet that has no correspondence in the score, and only serves to mark the separation between consecutive symbols (necessary in case *S1* and *S2* are the same, i.e. have the same encoding).

4.5.2 Cross Entropy Loss

The advantage we got from CTC was that the prediction would be spread across the whole output size, effectively making it so that the model did not have to learn to compute the output sequence's length. Compare the prediction structure from above to the one being used with cross entropy loss:

S1, S2, …, Sn, END, IGNORE, IGNORE, …, IGNORE

S1 to *Sn* are musical symbols. *END* is a symbol added to the alphabet to represent the end of the sequence. Musically, *END* corresponds to the double bar marking the end of a piece. If a double bar is not present at the end of the sequence in the image, *END* will still be added to the output. All symbols after *END*, marked above as *IGNORE*, are ignored. A particular image will only fill the minimum amount of the output space that is needed (depending on the number of symbols in the image).

4.5.3 Loss Function Comparison

CTC is an elegant solution that lifts the burden of identifying the sequence length from the model. However, it ends up leading to slower learning in the end. Figure 37, Figure 38, and Figure 39 show the accuracy, normalized Levenshtein distance and loss (section 4.7 explains more about these metrics) that the model achieves with cross entropy loss and CTC loss on a subset of the dataset. As can be seen in the graphs, it takes more than twice the number of epochs for CTC loss to reach the results achieved with cross entropy loss. The running time holds to the same proportion, at around 2.35 longer running time with CTC, meaning an epoch takes just as long to be processed, irrespective of the chosen loss function.

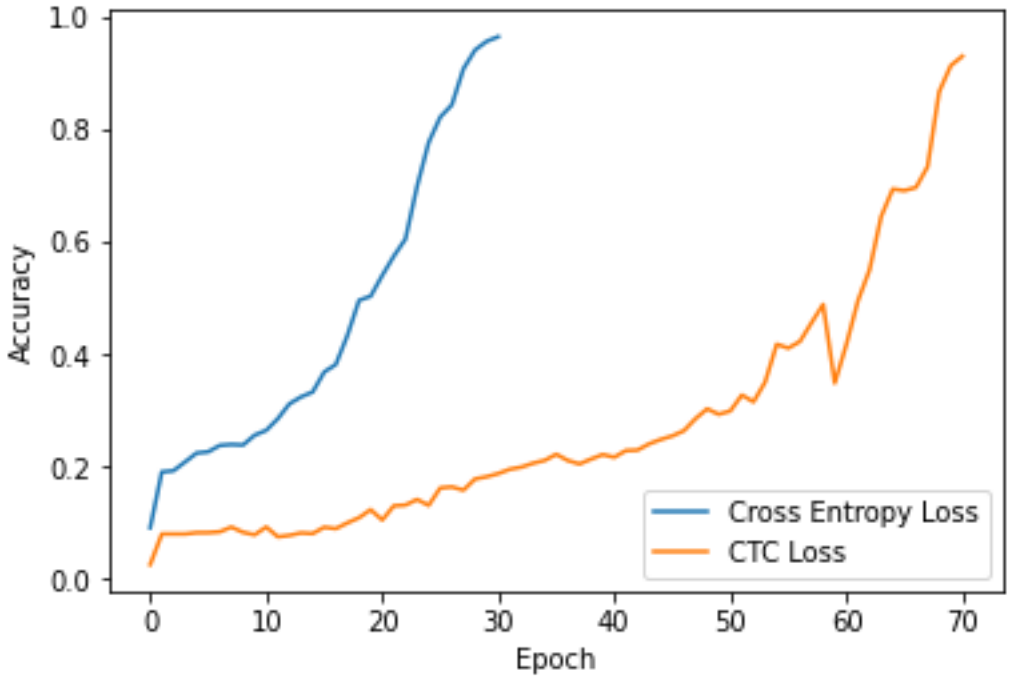

Figure 37 Accuracy for model with cross entropy loss and CTC loss

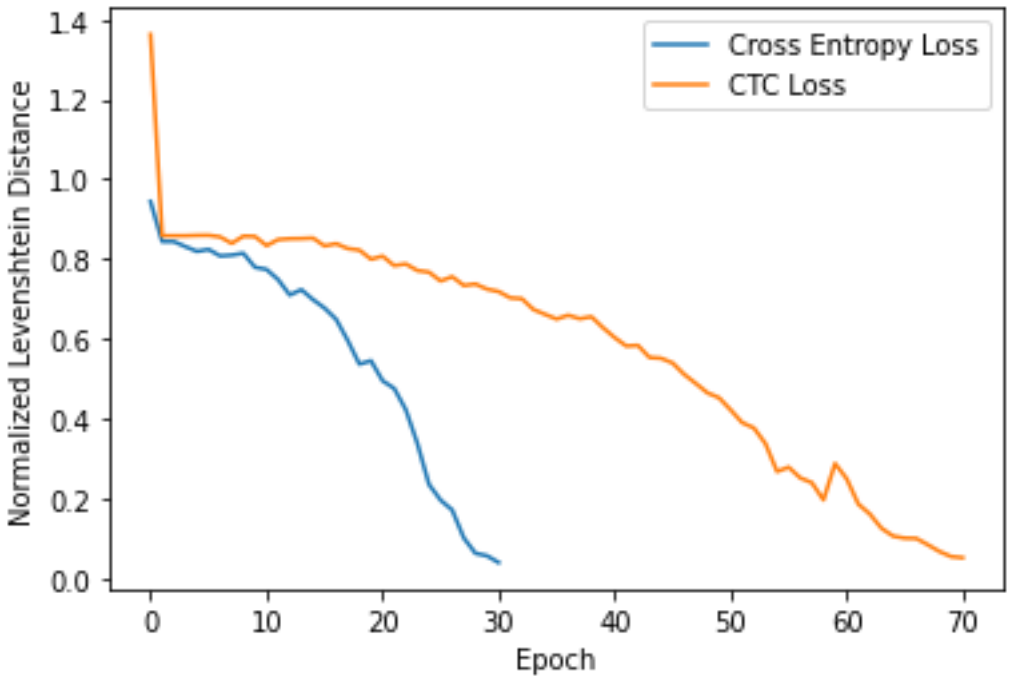

Figure 38 Normalized Levenshtein distance for model with cross entropy loss and CTC loss

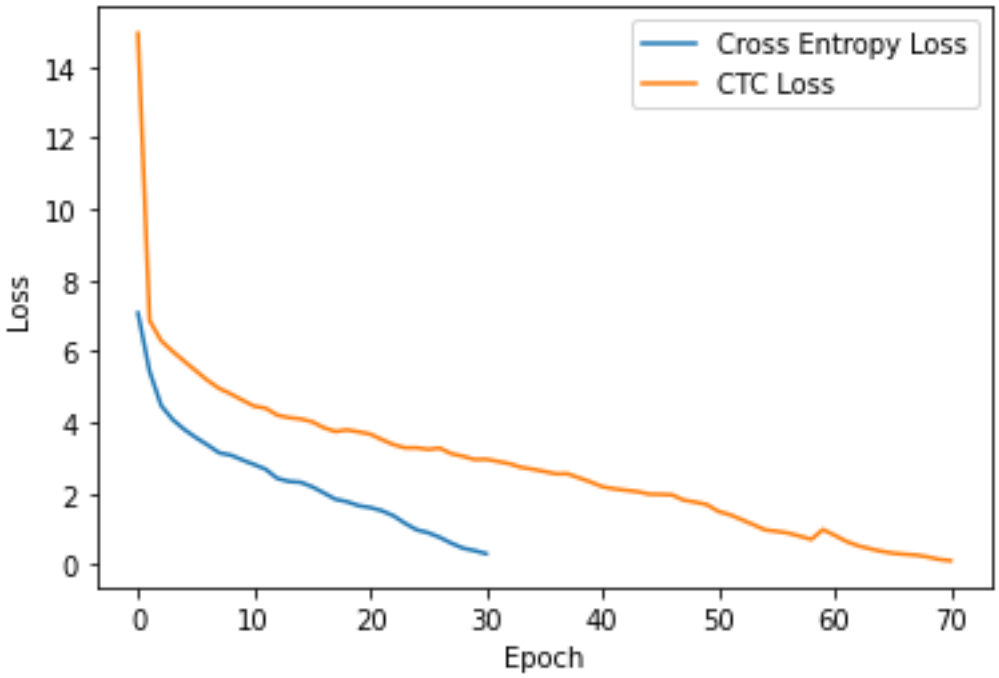

Figure 39 Loss for model with cross entropy loss and CTC loss

Having evaluated the performance of both methods, the verdict is clearly in favor of cross entropy loss, which is the loss function used throughout the rest of the paper.

4.6 Optimizer

The optimizer used is the weighted Adam optimizer from Hugging Face [34]. This was chosen because of good performance with little involvement from the developer. The learning rate was experimented with and settled at 5e-5.

4.7 Metric computation

The next few sections will each describe one of the metrics computed for evaluation of the implementation.

4.7.1 Accuracy

Accuracy has been computed as the number of correctly classified symbols in a prediction up to the length of the target, divided by the length of the target, averaged across multiple samples.

4.7.2 Top-n Accuracy

Top-n accuracy has been computed as the number of times in a prediction (up to the length of the target sequence) that the target symbol is in the top n predictions divided by the length of the target sequence and averaged across multiple samples. This metric will allow us to gauge whether the model is guessing randomly when it makes mistakes or is instead finding it difficult to distinguish between a group of similar symbols.

4.7.3 Edit Distance

Previous work [16] [31] [32] computed a symbol error rate as the normalized Levenshtein distance [36] between the prediction and the target. To facilitate comparison, this was also done here.

The Levenshtein distance is part of a larger class of functions computing edit distances. It takes 2 sequences as arguments and returns the minimum number of basic edits necessary to make the sequences identical. A basic edit can be:

1. Inserting an element at a specific position (e.g.: “aa” -> “aba”)
2. Deleting an element from the sequence (e.g.: “aba” -> “aa”
3. Replacing an element in the sequence with another (e.g.: “aba” -> “aca”)

For each target-prediction pair the Levenshtein distance is computed, then divided by the length of the target, normalizing it. Normalized Levenshtein distances are averaged across samples when reported.

4.7.4 Inference time

Inference time was computed by averaging 1000 runs. One run consisted of loading an image, preprocessing it, passing it through the model and interpreting the prediction. This is the inference time a user would experience, and not just the time spent on the processing done by the model. For GPU measurements, Amnon Geifman’s indications were followed [37].

4.8 Software

Python 3.6.9[7] was used, and together with it, the following libraries:

- NumPy 1.18.5[8]
- PyTorch 1.8.1[9] - First developments used TensorFlow 2.3, following Calvo-Zaragoza and Rizo [16], but upgrading from TensorFlow 1. Later, the project was moved to

[7] https://www.python.org/downloads/release/python-369/ [Accessed on 16 06 2021]
[8] https://pypi.org/project/numpy/ [Accessed on 16 06 2021]
[9] https://pytorch.org/ [Accessed on 16 06 2021]

PyTorch because it provided better compatibility with other libraries, i.e., Hugging Face [34] and X-Transformers.

- Hugging Face 4.5.1 [34] - Through this library the user gets a convenient API to work with the vision transformer developed by Google [25]. An honorable mention should be made for X-Transformers[10], which was initially used in development, but later dropped in favor of the pretrained models made available by Hugging Face.
- Cuda 10.1[11]
- OpenCV-Python 4.4.0.44[12]
- Polyleven 0.7[13] - Used to compute Levenshtein distance. It only works on strings, but our sequences of symbols, represented as lists of integers, could be easily converted: *“”.join(map(chr, sequence))* .

4.9 Hardware

Initial developments took place on the university’s cluster. Because of some missing libraries (libcudnn.so.7, PyTorch, Hugging Face [34]) and with installation not being made possible for standard users, a cloud provider was sought out for further development. Google Cloud Platform, Google Colab, Amazon Web Services, Gradient, vast.ai, FluidStack, Q Blocks, Azure and Kaggle were considered. Azure was chosen in the end and all subsequent development took place on a Standard NC6_Promo virtual machine[14], which is one of the weaker, but cheaper options. Microsoft does not state what GPU type you can get for an NC6 instance, but whenever this was checked during development, the GPU was a Tesla K80 [38]. Depending on availability, one could expect to be assigned different GPUs. A similar situation is that of the CPU. The timings reported for CPU runs in section 5 are on an Intel Xeon CPU E5-2690 v3 @ 2.60GHz.

[10] https://github.com/lucidrains/x-transformers [Accessed on 16 06 2021]
[11] https://developer.nvidia.com/cuda-10.1-download-archive-base [Accessed on 16 06 2021]
[12] https://pypi.org/project/opencv-python/ [Accessed on 16 06 2021]
[13] https://pypi.org/project/polyleven/ [Accessed on 16 06 2021]
[14] https://docs.microsoft.com/en-us/azure/virtual-machines/nc-series [Accessed on 16 06 2021]

5 RESULTS

This section will present the performance of the model, first from a quantitative perspective, then from a qualitative one. It will also provide comparison with existing solutions discussed in section 2.

5.1 Training Process

The training process is summarized in Figure 40, Figure 41, and Figure 42. In all cases, the x axis represents the number of samples passed through the model. Dividing this by 70 000 (the number of samples in the train set) will result in obtaining the epoch at that point in time. In total, 5 epochs are represented in the plot. One epoch took around 4.6 hours to execute in the conditions described under section 4.9. An evaluation step was run periodically, computing the metrics on a test set. The occasional spikes on the test set plots are due to the positive/negative influence of the samples the model has processed since the last report. Those would not be present if the graphs were made as a function of the number of epochs, but it would result in small unrepresentative plots due to the low number of elapsed epochs.

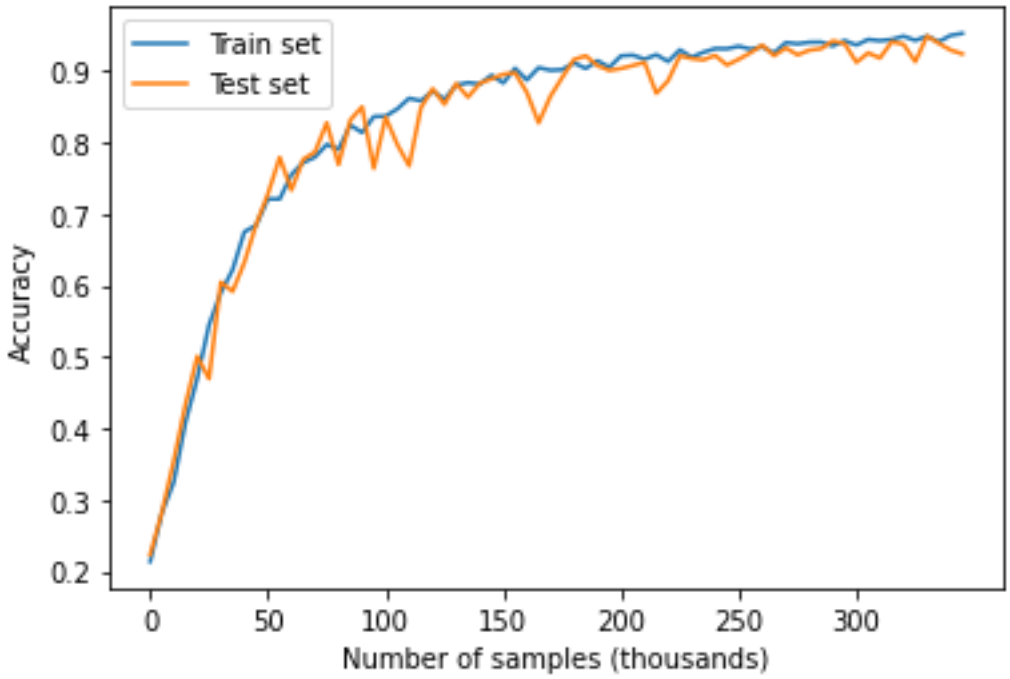

Figure 40 Accuracy for the train and test sets as a function of the number of samples the model is exposed to

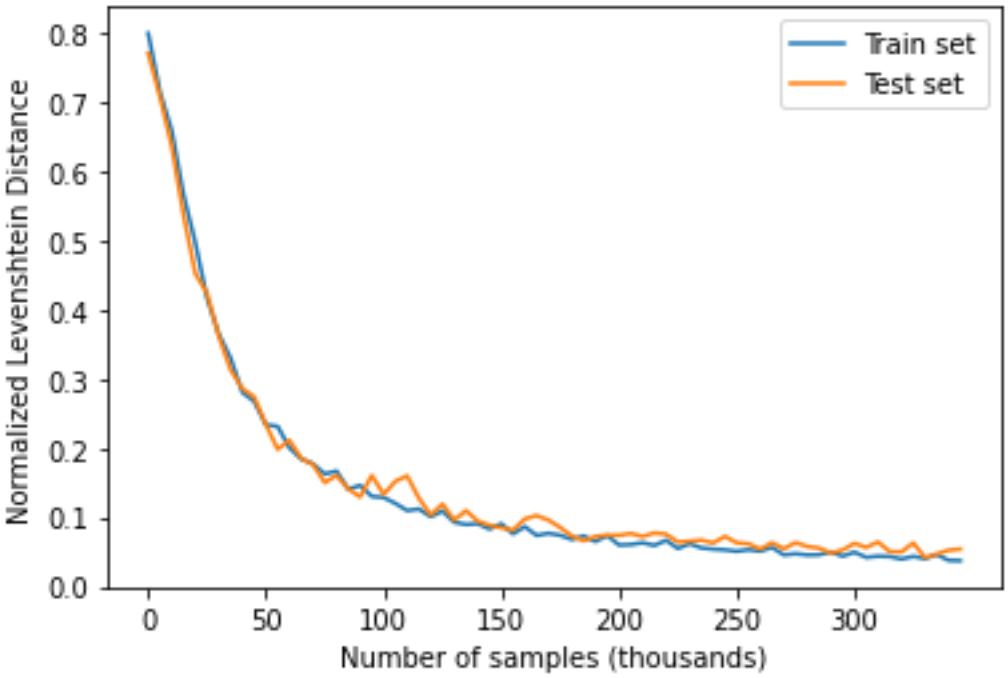

Figure 41 Normalized Levenshtein distance for the train and test sets as a function of the number of samples the model is exposed to

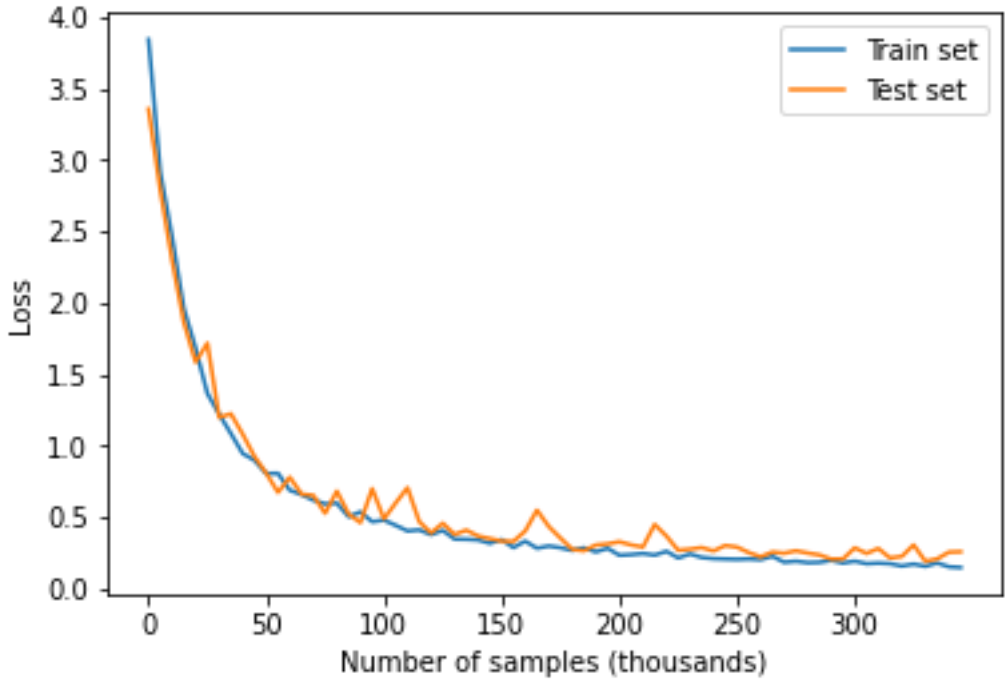

Figure 42 Loss for the train and test sets as a function of the number of samples the model is exposed to

The final results, averaged over the full test set of 7000 samples, are 0.9277 accuracy, 0.9948 top-5 accuracy, 0.0502 normalized Levenshtein distance and 0.2318 loss. Inference time stands at 0.12 seconds with a standard deviation of 0.0127 when using the Tesla K80 and 0.282 seconds with a standard deviation of 0.006 when doing inference on the CPU. The top-

5 accuracy could suggest that, had training been kept going for longer, the model would have continued to improve. It should be noted that overfit had not been reached, as was seen in Figure 42.

The metrics are presented more succinctly in Table 2.

Table 2 Final results obtained with ViT

Accuracy	0.9277
Top-5 accuracy	0.9948
Levenshtein distance (normalized)	0.0502
Loss	0.2318
Inference time (CPU)	0.282
Inference time (GPU)	0.12

5.2 Confusion Matrix

To better understand the model, normalized confusion matrices were calculated for both the test set and a train set subset of equal size. The vocabulary is too large to represent confusions as actual matrices or to enumerate all confusions here. Instead, some significant cases will be listed, under the following format:

```
(
        target encoding,
        target name,
        prediction encoding,
        prediction name,
        number of confusions divided by target occurrences,
        number of occurrences
)
```

The first meaningful results are the most frequent confusions for a target. These happen mostly for symbols that are rare in the dataset (e.g.: one appearance), which tend to be classified as bar lines. This can be explained through the fact that the bar line is the most frequent symbol in the dataset, so the model gets better results by guessing bar line when it does not recognize a symbol. This phenomenon can be observed in the following sequence:

```
(398, 'note-A4_double_whole.', 0, 'barline', 1.0, 1)
(489, 'note-Ab3_whole', 0, 'barline', 1.0, 1)
(574, 'note-B3_whole.', 0, 'barline', 1.0, 1)
(675, 'note-Bb3_whole_fermata', 0, 'barline', 1.0, 1)
(718, 'note-Bb5_whole.', 0, 'barline', 1.0, 1)
(913, 'note-Cb5_half.', 0, 'barline', 1.0, 1)
(936, 'note-D2_thirty_second', 0, 'barline', 1.0, 1)
(1054, 'note-D#5_whole_fermata', 0, 'barline', 1.0, 1)
(1081, 'note-D6_whole.', 0, 'barline', 1.0, 1)
(1260, 'note-Eb2_quarter._fermata', 0, 'barline', 1.0, 1)
(1397, 'note-F#4_double_whole', 0, 'barline', 1.0, 1)
```

(1401, 'note-F4_double_whole_fermata', 0, 'barline', 1.0, 1)
(1590, 'note-G3_whole_fermata', 0, 'barline', 1.0, 1)
(1622, 'note-G4_quarter._fermata', 0, 'barline', 1.0, 1)
(1682, 'note-G5_whole_fermata', 0, 'barline', 1.0, 1)
(1687, 'note-Gb3_half', 0, 'barline', 1.0, 1)

Another interesting case is that of grace notes, previously discussed in section 2.1 when it was mentioned that they are relatively rare in the dataset and in music in general, and that they are graphically difficult to identify. Those characteristics are also evident here:

(200, 'gracenote-G4_eighth.', 28, 'gracenote-A4_sixteenth', 1.0, 1)
(207, 'gracenote-G4_thirty_second', 28, 'gracenote-A4_sixteenth', 1.0, 1)
(206, 'gracenote-G#4_thirty_second', 30, 'gracenote-A4_thirty_second', 1.0, 1)
(31, 'gracenote-A#5_eighth', 32, 'gracenote-A5_eighth', 1.0, 1)
(64, 'gracenote-B5_eighth', 32, 'gracenote-A5_eighth', 1.0, 1)
(48, 'gracenote-Ab5_sixteenth', 34, 'gracenote-A5_sixteenth', 1.0, 1)
(848, 'note-C#5_quarter._fermata', 119, 'gracenote-D5_eighth', 1.0, 1)
(144, 'gracenote-E#5_eighth', 145, 'gracenote-E5_eighth', 1.0, 2)
(179, 'gracenote-F#4_thirty_second', 177, 'gracenote-F#4_sixteenth', 1.0, 1)
(158, 'gracenote-Eb4_thirty_second', 178, 'gracenote-F4_sixteenth', 1.0, 1)
(191, 'gracenote-G#3_eighth', 192, 'gracenote-G3_eighth', 1.0, 2)
(171, 'gracenote-F3_sixteenth', 195, 'gracenote-G3_sixteenth', 1.0, 1)
(180, 'gracenote-F4_thirty_second', 207, 'gracenote-G4_thirty_second', 1.0, 1)

A fascinating case is that of the long rests (Figure 43 shows a 15-measurement rest, also called a tacet). The model effectively recognizes the digits in the symbol but, perhaps being unfamiliar with a particular configuration, its prediction ends up being a more common rest that contains the recognized digit (e.g.: instead of 55, the output is 15; instead of 71, it is 17):

(298, 'multirest-53', 250, 'multirest-13', 1.0, 1)
(300, 'multirest-55', 253, 'multirest-15', 1.0, 1)
(316, 'multirest-71', 256, 'multirest-17', 1.0, 1)
(304, 'multirest-59', 258, 'multirest-19', 1.0, 1)
(299, 'multirest-54', 277, 'multirest-34', 1.0, 1)
(329, 'multirest-94', 277, 'multirest-34', 1.0, 1)
(326, 'multirest-89', 282, 'multirest-39', 1.0, 1)

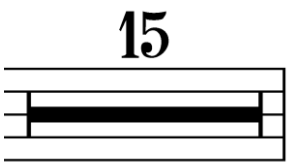

Figure 43 Multirest of 15 measures[15]

[15] © https://upload.wikimedia.org/wikipedia/commons/e/ef/15_bars_multirest.png [Accessed 19 06 2021]

The bottom of the confusion list concerns the more frequent symbols, which are sometimes misclassified, but not in a consistent manner. For example, the bar line is often confused with notes, but not the same notes every time:

(0, 'barline', 373, 'note-A3_half', 3.8315644277558524e-05, 26099)
(0, 'barline', 392, 'note-A3_thirty_second', 3.8315644277558524e-05, 26099)
(0, 'barline', 424, 'note-A#4_sixteenth', 3.8315644277558524e-05, 26099)
(0, 'barline', 448, 'note-A5_half.', 3.8315644277558524e-05, 26099)
(0, 'barline', 454, 'note-A5_quarter.', 3.8315644277558524e-05, 26099)
(0, 'barline', 461, 'note-A5_sixteenth.', 3.8315644277558524e-05, 26099)
(0, 'barline', 501, 'note-Ab4_sixteenth', 3.8315644277558524e-05, 26099)
(0, 'barline', 504, 'note-Ab4_thirty_second', 3.8315644277558524e-05, 26099)
(0, 'barline', 509, 'note-Ab5_eighth', 3.8315644277558524e-05, 26099)
(0, 'barline', 515, 'note-Ab5_quarter', 3.8315644277558524e-05, 26099)
(0, 'barline', 556, 'note-B3_half', 3.8315644277558524e-05, 26099)
(0, 'barline', 591, 'note-B4_half.', 3.8315644277558524e-05, 26099)
(0, 'barline', 620, 'note-B5_eighth', 3.8315644277558524e-05, 26099)
(0, 'barline', 627, 'note-B5_quarter', 3.8315644277558524e-05, 26099)

5.3 Comparison with Previous Work

The proposed implementation compares favorably with past work. Overall accuracy is lower, but a better inference time is achieved in exchange. The normalized Levenshtein distance of 0.0513, corresponding to a symbol error rate of 5.13%, is above that of 0.8% achieved with an RNN [16] (see section 2.1 for more information), but their reported inference time of 1 second on an Intel Core i5-2400 CPU @ 3.10 GHz has been improved on with a 0.282 second inference time on an Intel Xeon CPU E5-2690 v3 @ 2.60GHz. Although the i5 is a weaker processor overall, single core performance is similar (according to the Cinebench 23 benchmark[16]) and does not account for this large of a difference in inference time. It is also unclear if the reported 1 second inference time also includes image loading and preprocessing, as the measurements in this paper do. In that case, other factors would come into play (such as disk speed).

The comparisons from above and all metrics reported in section 5.1 are summarized in Table 3. Missing fields are due to the original work not reporting that metric. Note that despite the ten times larger edit distance, accuracy is well above 90%.

[16] https://www.cpu-monkey.com/en/compare_cpu-intel_core_i5_2400-1450-vs-intel_xeon_e5_2697_v3-508 [Accessed on 28 06 2021]

Table 3 Comparative results across all reported metrics

	SMITE / PrIMuS	**RNN / PrIMuS** [16]
Accuracy	0.9277	
Top-5 accuracy	0.9948	
Levenshtein distance (normalized)	0.0502	0.008
CPU inference time (s)	0.282	1
GPU inference time (s)	0.12	

5.4 Inference Example and Comparison with Audiveris

To better showcase the results obtained, an example is provided. Figure 44 shows the sample.

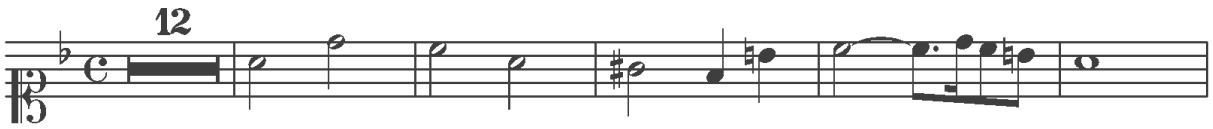

Figure 44 Example score

Passing this image to the trained model, we would get the encoded prediction:

> 1, 231, 1779, 245, 0, 408, 1026, 0, 832, 408, 0, 1604, 1426, 597, 0, 832, 1741, 824, 1044, 823, 583, 0, 434, 0, 1781, 1781, 1781, 1781, 1781, 1781, 1781, 1781, 1781, 1781, 1781, 1781, 1781, 1781, 231, 231, 1781, 1781, 1781, 231, 1781

As previously discussed in section 4.5.2, the true prediction is only up to the first END symbol. END here is the number 1781, which can be seen repeating in the second half of the prediction. By removing this appendage, we get the clean prediction:

> 1, 231, 1779, 245, 0, 408, 1026, 0, 832, 408, 0, 1604, 1426, 597, 0, 832, 1741, 824, 1044, 823, 583, 0, 434, 0

This can now be transformed to a human-readable format by simple association from the encoding to the full symbol name. In this case, the recognized musical sequence is identical to the target (i.e., the accuracy on this sample is 1 and the Levenshtein distance is 0):

> clef-C1
> keySignature-FM
> timeSignature-C
> multirest-12
> barline
> note-A4_half

note-D5_half
barline
note-C5_half
note-A4_half
barline
note-G#4_half
note-F4_quarter
note-B4_quarter
barline
note-C5_half
tie
note-C5_eighth.
note-D5_sixteenth
note-C5_eighth
note-B4_eighth
barline
note-A4_whole
barline

If the same image is passed to Audiveris [5] (version 5.1.0 was used), the result shown in Figure 45 is obtained.

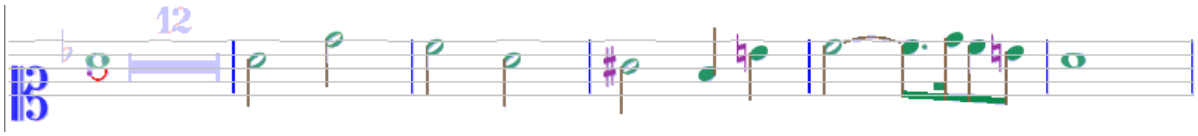

Figure 45 Audiveris output on the sample from Figure 44

As stated in section 2.1, Audiveris' focus is not on the type of score found in PrIMuS. As an end-user product, the set of scores that Audiveris handles must be much broader, so any errors in this example should not be taken to mean that Audiveris overall performs poorly.

The errors Audiveris makes on this sample are on the key signature, time signature and the initial 12 measure rest. The flat of the key signature is interpreted as a 6, the common time signature is interpreted as a whole note with a fermata underneath, and the 12 from the multirest is recognized, but not the horizontal bar indicating the rest itself. With the encoding discussed in section 4.1, this would correspond to an accuracy of 87.5%. This can be explained by considering that Audiveris deals with complete scores, while the model discussed in this paper was only trained on incipits, so better performance on types of symbols that are frequent in incipits but not in complete scores is understandable.

6 CONCLUSIONS AND FUTURE WORK

This paper has argued for the implementation of a transformer model for OMR tasks. Specifically, a modified ViT model [25] was employed, leading to results comparable to those of the most recent developments in deep learning OMR. At the price of reduced accuracy, inference time was improved, which is essential for commercial usability. The modest goals set out in section 1.4 have been achieved and surpassed - those were for the solution to prove itself better than random guessing and for the implementation to be simple and complete.

The first objective has been fulfilled decisively, with expected random guessing accuracy below 0.001, and achieved accuracy above 0.9. Even when training the model on just 5000 images, accuracy was above 0.55.

As for the second objective, completeness in both the sense of the entire score being interpreted in context, and in the sense of having the model encapsulate or replace all intermediary operations, has been delivered.

Simplicity, on the other hand, is harder to give a binary answer to. Most of the complexity in the implementation arises from the need to preprocess the samples. Engaging with more difficult datasets (such as sets of complete scores, with titles, lyrics, multiple staves) is bound to complicate the preprocessing step further. However, most of the preprocessing operations are simple and many are already encapsulated under the feature extractor (section 4.3.3). The most problematic operation remains the segmentation (section 4.3.1). This is tied to both particularities of the dataset (single staff, short wide images) and particularities of the model (requires square images). Alternatives are to build models specific to the targeted score type, or models that do not prefer certain image shapes. This may pose some problems with positional encoding in the transformer, but this issue needs to be investigated further.

In any case, the ViT model is not a perfect fit for the task. Its advantage was the pre-training it had received. If a model were designed from scratch for this task, it should be made compatible with grayscale images, and different patch cutting strategies should be experimented with, for example by making vertical cuts through a staff, which could result in easier to learn sequences.

It should be noted that the metrics reported in section 5 are not the result of a cross-validation process. This is simply because of a lack of time and resources but should be done in the future, as the metrics are expected to show some variance over multiple training runs. Two training runs were attempted with different test sets and the results suggest that the reported metrics are indeed robust, but a more complete validation is still necessary. More time would also allow for more thorough training. The model's learning potential had not been exhausted when training was stopped (although learning had slowed down), so the peak performance of this architecture has most likely not been reached.

Further developments should also consider training this model on more realistic images, such as the ones in Camera-PrIMuS. As seen in section 2.2, there is a good chance the model will

still perform reasonably well. It would also be interesting to get an accuracy metric from other work (this was not reported in the articles discussed under section 2), to confirm that the performances of this model are competitive.

A more difficult task that could be proposed is the recognition of polyphonic scores. However, the current encoding will lead to an exponential increase of the vocabulary if polyphonic scores are introduced, since all combinations of simultaneous notes will be assigned a new symbol. The current encoding also reduces the dimensionality of the score, forcing symbols into a sequence corresponding to the left-right orientation of the score, but having no way to distinguish symbols on the vertical axis. If the task of polyphonic scores is approached, it will have to be done with a rethought encoding scheme.

Another OMR problem mentioned before is that of handwritten scores. This problem could be approached with the solution described in this paper, as long as the score is monophonic.

Larger sheets also present an interesting challenge. The problem with their size can be solved by simply cutting the image into staves and passing each of them through the model separately. The real issue is that of losing contextual information from previous staves if this information is not rewritten on every staff (e.g.: the key).

7 BIBLIOGRAPHY

[1] J. Calvo-Zaragoza, J. Hajič and A. Pacha, "Understanding Optical Music Recognition," *ACM Computing Surveys,* 2020.

[2] "info-capella-scan," capella, [Online]. Available: https://www.capella-software.com/us/index.cfm/products/capella-scan/info-capella-scan/. [Accessed 07 06 2021].

[3] "PhotoScore," NEURATRON, [Online]. Available: https://www.neuratron.com/photoscore.htm. [Accessed 07 06 2021].

[4] "Arsupix," Arsupix, [Online]. Available: https://www.aruspix.net/index.html. [Accessed 07 06 2021].

[5] "Audiveris," Audiveris, [Online]. Available: https://github.com/Audiveris/audiveris. [Accessed 07 06 2021].

[6] "OpenOMR," greenjava, [Online]. Available: https://github.com/greenjava/OpenOMR. [Accessed 07 06 2021].

[7] "PlayScore 2," PlayScore, [Online]. Available: https://www.playscore.co/. [Accessed 07 06 2021].

[8] "Eighth rest or smudge," capella, [Online]. Available: https://www.capella-software.com/us/index.cfm/products/capella-scan/eighth-rest-or-smudge/. [Accessed 07 06 2021].

[9] J. Hajič, M. Kolárová, A. Pacha and J. Calvo-Zaragoza, "How current optical music recognition systems are becoming useful for digital libraries," in *5th International Conference on Digital Libraries for Musicology*, Paris, 2018.

[10] P. Bellini, I. Bruno and P. Nesi, "Assessing Optical Music Recognition Tools," *Computer Music Journal,* vol. 31, no. 1, p. 68–93, 2007.

[11] E. Nichols and D. Byrd, "The Future of Music IR: How Do You Know When a Problem is Solved?".

[12] M. Droettboom and I. Fujinaga, "Micro-level groundtruthing environment for OMR," in *5th International Conference on Music Information Retrieval*, Barcelona, 2004.

[13] D. Bainbridge and T. Bell, "The Challenge of Optical Music Recognition," *Computers and the Humanities,* vol. 35, pp. 95-121, 2001.

[14] A. Rebelo, I. Fujinaga, F. Paszkiewicz, A. R. S. Marcal, C. Guedes and J. S. Cardoso, "Optical music recognition: state-of-the-art and open issues," *International Journal of Multimedia Information Retrieval,* no. 1, pp. 173-190, 2012.

[15] K. Ng, D. Cooper, E. Stefani, R. Boyle and N. Bailey, "Embracing the Composer: Optical Recognition of Handwritten Manuscripts," in *ICMC*, Beijing, China, 1999.

[16] J. Calvo-Zaragoza and D. Rizo, "End-to-End Neural Optical Music Recognition of Monophonic Scores," *Applied Sciences,* vol. 8, no. 4, 2018.

[17] D. Byrd and J. G. Simonsen, "Towards a Standard Testbed for Optical Music Recognition: Definitions, Metrics, and Page Images," *Journal of New Music Research,* vol. 44, no. 3, pp. 169-195, 2015.

[18] I. Fujinaga and G. Vigliensoni, "The Art of Teaching Computers: The SIMSSA Optical Music Recognition Workflow System," in *European Signal Processing Conference*, A Coruna, Spain, 2019.

[19] G. S. Choudhury, T. DiLauro, M. Droettboom, I. Fujinaga, B. Harrington and K. MacMillan, "Optical Music Recognition System within a Large-Scale," 2000.

[20] "MusicXML," W3C, [Online]. Available: https://www.w3.org/2021/06/musicxml40/. [Accessed 07 06 2021].

[21] "Standard MIDI-File Format Spec. 1.1," [Online]. Available: http://www.music.mcgill.ca/~ich/classes/mumt306/midiformat.pdf. [Accessed 07 06 2021].

[22] apacha, "OMR-Datasets," [Online]. Available: https://apacha.github.io/OMR-Datasets/. [Accessed 07 06 2021].

[23] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser and I. Polosukhin, "Attention Is All You Need," in *31st Conference on Neural Information Processing Systems*, Long Beach, CA, USA, 2017.

[24] N. Kitaev, Ł. Kaiser and A. Levskaya, "Reformer: The Efficient Transformer," in *ICLR*, 2020.

[25] A. Dosovitskiy, L. Beyer, A. Kolesnikov, D. Weissenborn, X. Zhai, T. Unterthiner, M. Dehghani, M. Minderer, G. Heigold, S. Gelly, J. Uszkoreit and N. Houlsby, "An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale," in *ICLR*, 2020.

[26] J. Devlin, M.-W. Chang, K. Lee and K. Toutanova, "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding," 2018.

[27] 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 and Da, "Language Models are Few-Shot Learners," 2020.

[28] D. Noever, M. Ciolino and J. Kalin, "The Chess Transformer: Mastering Play using Generative Language Models," 2020.

[29] A. Nambiar, M. Heflin, S. Liu, S. Maslov, M. Hopkins and A. Ritz, "Transforming the Language of Life: Transformer Neural Networks for Protein Prediction Tasks," in *11th ACM International Conference on Bioinformatics, Computational Biology and Health Informatics*, 2020.

[30] A. Graves, S. Fernández, F. Gomez and J. Schmidhuber, "Connectionist temporal classification: labelling unsegmented sequence data with recurrent neural networks," in *23rd international conference on Machine learning*, Pittsburgh, PA, USA, 2006.

[31] J. Calvo-Zaragoza and D. Rizo, "Camera-PrIMuS: Neural End-to-End Optical Music Recognition on Realistic Monophonic Scores," *ISMIR,* 2018.

[32] A. Ríos-Vila, M. Esplà-Gomis, D. Rizo, P. J. P. d. León and J. M. Iñesta, "Applying Automatic Translation for Optical Music Recognition's Encoding Step," *Advances in Music Reading Systems,* 2021.

[33] P. Kohen, Statistical machine translation, Cambridge University Press, 2009.

[34] "Hugging Face," Hugging Face, Inc., [Online]. Available: https://huggingface.co/. [Accessed 14 06 2021].

[35] "HuggingFace - Visual transformers by Google," HuggingFace, [Online]. Available: https://huggingface.co/models?search=google%2Fvit. [Accessed 14 06 2021].

[36] V. I. Levenshtein, "Binary codes capable of correcting deletions, insertions and reversals," *translated by P.S. Novikov from Doklady Akademii Nauk SSSR,* vol. 163, no. 4, p. 845–848, 1965.

[37] A. Geifman, "The Correct Way to Measure Inference Time of Deep Neural Networks," towards data science, 05 05 2020. [Online]. Available: https://towardsdatascience.com/the-correct-way-to-measure-inference-time-of-deep-neural-networks-304a54e5187f. [Accessed 29 06 2021].

[38] "NVIDIA TESLA K80," Nvidia, [Online]. Available: https://www.nvidia.com/en-gb/data-center/tesla-k80/. [Accessed 17 06 2021].

APPENDIX A. THE MODEL

The model used – a vision transformer (google/vit-base-patch16-224) provided by Hugging Face [35], with the classification layer replaced:

```
ModViT(
 (vit): ViTModel(
  (embeddings): ViTEmbeddings(
   (patch_embeddings): PatchEmbeddings(
    (projection): Conv2d(3, 768, kernel_size=(16, 16), stride=(16, 16))
   )
   (dropout): Dropout(p=0.0, inplace=False)
  )
  (encoder): ViTEncoder(
   (layer): ModuleList(
    (0): ViTLayer(
     (attention): ViTAttention(
      (attention): ViTSelfAttention(
       (query): Linear(in_features=768, out_features=768, bias=True)
       (key): Linear(in_features=768, out_features=768, bias=True)
       (value): Linear(in_features=768, out_features=768, bias=True)
       (dropout): Dropout(p=0.0, inplace=False)
      )
      (output): ViTSelfOutput(
       (dense): Linear(in_features=768, out_features=768, bias=True)
       (dropout): Dropout(p=0.0, inplace=False)
      )
     )
     (intermediate): ViTIntermediate(
      (dense): Linear(in_features=768, out_features=3072, bias=True)
     )
     (output): ViTOutput(
      (dense): Linear(in_features=3072, out_features=768, bias=True)
      (dropout): Dropout(p=0.0, inplace=False)
     )
     (layernorm_before): LayerNorm((768,), eps=1e-12, elementwise_affine=True)
     (layernorm_after): LayerNorm((768,), eps=1e-12, elementwise_affine=True)
    )
    (1): ViTLayer(
     (attention): ViTAttention(
      (attention): ViTSelfAttention(
       (query): Linear(in_features=768, out_features=768, bias=True)
       (key): Linear(in_features=768, out_features=768, bias=True)
       (value): Linear(in_features=768, out_features=768, bias=True)
       (dropout): Dropout(p=0.0, inplace=False)
      )
      (output): ViTSelfOutput(
       (dense): Linear(in_features=768, out_features=768, bias=True)
       (dropout): Dropout(p=0.0, inplace=False)
      )
     )
     (intermediate): ViTIntermediate(
```

```
    (dense): Linear(in_features=768, out_features=3072, bias=True)
   )
   (output): ViTOutput(
    (dense): Linear(in_features=3072, out_features=768, bias=True)
    (dropout): Dropout(p=0.0, inplace=False)
   )
   (layernorm_before): LayerNorm((768,), eps=1e-12, elementwise_affine=True)
   (layernorm_after): LayerNorm((768,), eps=1e-12, elementwise_affine=True)
  )
  (2): ViTLayer(
   (attention): ViTAttention(
    (attention): ViTSelfAttention(
     (query): Linear(in_features=768, out_features=768, bias=True)
     (key): Linear(in_features=768, out_features=768, bias=True)
     (value): Linear(in_features=768, out_features=768, bias=True)
     (dropout): Dropout(p=0.0, inplace=False)
    )
    (output): ViTSelfOutput(
     (dense): Linear(in_features=768, out_features=768, bias=True)
     (dropout): Dropout(p=0.0, inplace=False)
    )
   )
   (intermediate): ViTIntermediate(
    (dense): Linear(in_features=768, out_features=3072, bias=True)
   )
   (output): ViTOutput(
    (dense): Linear(in_features=3072, out_features=768, bias=True)
    (dropout): Dropout(p=0.0, inplace=False)
   )
   (layernorm_before): LayerNorm((768,), eps=1e-12, elementwise_affine=True)
   (layernorm_after): LayerNorm((768,), eps=1e-12, elementwise_affine=True)
  )
  (3): ViTLayer(
   (attention): ViTAttention(
    (attention): ViTSelfAttention(
     (query): Linear(in_features=768, out_features=768, bias=True)
     (key): Linear(in_features=768, out_features=768, bias=True)
     (value): Linear(in_features=768, out_features=768, bias=True)
     (dropout): Dropout(p=0.0, inplace=False)
    )
    (output): ViTSelfOutput(
     (dense): Linear(in_features=768, out_features=768, bias=True)
     (dropout): Dropout(p=0.0, inplace=False)
    )
   )
   (intermediate): ViTIntermediate(
    (dense): Linear(in_features=768, out_features=3072, bias=True)
   )
   (output): ViTOutput(
    (dense): Linear(in_features=3072, out_features=768, bias=True)
    (dropout): Dropout(p=0.0, inplace=False)
```

```
  )
  (layernorm_before): LayerNorm((768,), eps=1e-12, elementwise_affine=True)
  (layernorm_after): LayerNorm((768,), eps=1e-12, elementwise_affine=True)
)
(4): ViTLayer(
  (attention): ViTAttention(
    (attention): ViTSelfAttention(
      (query): Linear(in_features=768, out_features=768, bias=True)
      (key): Linear(in_features=768, out_features=768, bias=True)
      (value): Linear(in_features=768, out_features=768, bias=True)
      (dropout): Dropout(p=0.0, inplace=False)
    )
    (output): ViTSelfOutput(
      (dense): Linear(in_features=768, out_features=768, bias=True)
      (dropout): Dropout(p=0.0, inplace=False)
    )
  )
  (intermediate): ViTIntermediate(
    (dense): Linear(in_features=768, out_features=3072, bias=True)
  )
  (output): ViTOutput(
    (dense): Linear(in_features=3072, out_features=768, bias=True)
    (dropout): Dropout(p=0.0, inplace=False)
  )
  (layernorm_before): LayerNorm((768,), eps=1e-12, elementwise_affine=True)
  (layernorm_after): LayerNorm((768,), eps=1e-12, elementwise_affine=True)
)
(5): ViTLayer(
  (attention): ViTAttention(
    (attention): ViTSelfAttention(
      (query): Linear(in_features=768, out_features=768, bias=True)
      (key): Linear(in_features=768, out_features=768, bias=True)
      (value): Linear(in_features=768, out_features=768, bias=True)
      (dropout): Dropout(p=0.0, inplace=False)
    )
    (output): ViTSelfOutput(
      (dense): Linear(in_features=768, out_features=768, bias=True)
      (dropout): Dropout(p=0.0, inplace=False)
    )
  )
  (intermediate): ViTIntermediate(
    (dense): Linear(in_features=768, out_features=3072, bias=True)
  )
  (output): ViTOutput(
    (dense): Linear(in_features=3072, out_features=768, bias=True)
    (dropout): Dropout(p=0.0, inplace=False)
  )
  (layernorm_before): LayerNorm((768,), eps=1e-12, elementwise_affine=True)
  (layernorm_after): LayerNorm((768,), eps=1e-12, elementwise_affine=True)
)
(6): ViTLayer(
```

```
  (attention): ViTAttention(
    (attention): ViTSelfAttention(
      (query): Linear(in_features=768, out_features=768, bias=True)
      (key): Linear(in_features=768, out_features=768, bias=True)
      (value): Linear(in_features=768, out_features=768, bias=True)
      (dropout): Dropout(p=0.0, inplace=False)
    )
    (output): ViTSelfOutput(
      (dense): Linear(in_features=768, out_features=768, bias=True)
      (dropout): Dropout(p=0.0, inplace=False)
    )
  )
  (intermediate): ViTIntermediate(
    (dense): Linear(in_features=768, out_features=3072, bias=True)
  )
  (output): ViTOutput(
    (dense): Linear(in_features=3072, out_features=768, bias=True)
    (dropout): Dropout(p=0.0, inplace=False)
  )
  (layernorm_before): LayerNorm((768,), eps=1e-12, elementwise_affine=True)
  (layernorm_after): LayerNorm((768,), eps=1e-12, elementwise_affine=True)
)
(7): ViTLayer(
  (attention): ViTAttention(
    (attention): ViTSelfAttention(
      (query): Linear(in_features=768, out_features=768, bias=True)
      (key): Linear(in_features=768, out_features=768, bias=True)
      (value): Linear(in_features=768, out_features=768, bias=True)
      (dropout): Dropout(p=0.0, inplace=False)
    )
    (output): ViTSelfOutput(
      (dense): Linear(in_features=768, out_features=768, bias=True)
      (dropout): Dropout(p=0.0, inplace=False)
    )
  )
  (intermediate): ViTIntermediate(
    (dense): Linear(in_features=768, out_features=3072, bias=True)
  )
  (output): ViTOutput(
    (dense): Linear(in_features=3072, out_features=768, bias=True)
    (dropout): Dropout(p=0.0, inplace=False)
  )
  (layernorm_before): LayerNorm((768,), eps=1e-12, elementwise_affine=True)
  (layernorm_after): LayerNorm((768,), eps=1e-12, elementwise_affine=True)
)
(8): ViTLayer(
  (attention): ViTAttention(
    (attention): ViTSelfAttention(
      (query): Linear(in_features=768, out_features=768, bias=True)
      (key): Linear(in_features=768, out_features=768, bias=True)
      (value): Linear(in_features=768, out_features=768, bias=True)
```

```
      (dropout): Dropout(p=0.0, inplace=False)
    )
    (output): ViTSelfOutput(
      (dense): Linear(in_features=768, out_features=768, bias=True)
      (dropout): Dropout(p=0.0, inplace=False)
    )
  )
  (intermediate): ViTIntermediate(
    (dense): Linear(in_features=768, out_features=3072, bias=True)
  )
  (output): ViTOutput(
    (dense): Linear(in_features=3072, out_features=768, bias=True)
    (dropout): Dropout(p=0.0, inplace=False)
  )
  (layernorm_before): LayerNorm((768,), eps=1e-12, elementwise_affine=True)
  (layernorm_after): LayerNorm((768,), eps=1e-12, elementwise_affine=True)
)
(9): ViTLayer(
  (attention): ViTAttention(
    (attention): ViTSelfAttention(
      (query): Linear(in_features=768, out_features=768, bias=True)
      (key): Linear(in_features=768, out_features=768, bias=True)
      (value): Linear(in_features=768, out_features=768, bias=True)
      (dropout): Dropout(p=0.0, inplace=False)
    )
    (output): ViTSelfOutput(
      (dense): Linear(in_features=768, out_features=768, bias=True)
      (dropout): Dropout(p=0.0, inplace=False)
    )
  )
  (intermediate): ViTIntermediate(
    (dense): Linear(in_features=768, out_features=3072, bias=True)
  )
  (output): ViTOutput(
    (dense): Linear(in_features=3072, out_features=768, bias=True)
    (dropout): Dropout(p=0.0, inplace=False)
  )
  (layernorm_before): LayerNorm((768,), eps=1e-12, elementwise_affine=True)
  (layernorm_after): LayerNorm((768,), eps=1e-12, elementwise_affine=True)
)
(10): ViTLayer(
  (attention): ViTAttention(
    (attention): ViTSelfAttention(
      (query): Linear(in_features=768, out_features=768, bias=True)
      (key): Linear(in_features=768, out_features=768, bias=True)
      (value): Linear(in_features=768, out_features=768, bias=True)
      (dropout): Dropout(p=0.0, inplace=False)
    )
    (output): ViTSelfOutput(
      (dense): Linear(in_features=768, out_features=768, bias=True)
      (dropout): Dropout(p=0.0, inplace=False)
```

```
        )
       )
       (intermediate): ViTIntermediate(
        (dense): Linear(in_features=768, out_features=3072, bias=True)
       )
       (output): ViTOutput(
        (dense): Linear(in_features=3072, out_features=768, bias=True)
        (dropout): Dropout(p=0.0, inplace=False)
       )
       (layernorm_before): LayerNorm((768,), eps=1e-12, elementwise_affine=True)
       (layernorm_after): LayerNorm((768,), eps=1e-12, elementwise_affine=True)
      )
      (11): ViTLayer(
       (attention): ViTAttention(
        (attention): ViTSelfAttention(
         (query): Linear(in_features=768, out_features=768, bias=True)
         (key): Linear(in_features=768, out_features=768, bias=True)
         (value): Linear(in_features=768, out_features=768, bias=True)
         (dropout): Dropout(p=0.0, inplace=False)
        )
        (output): ViTSelfOutput(
         (dense): Linear(in_features=768, out_features=768, bias=True)
         (dropout): Dropout(p=0.0, inplace=False)
        )
       )
       (intermediate): ViTIntermediate(
        (dense): Linear(in_features=768, out_features=3072, bias=True)
       )
       (output): ViTOutput(
        (dense): Linear(in_features=3072, out_features=768, bias=True)
        (dropout): Dropout(p=0.0, inplace=False)
       )
       (layernorm_before): LayerNorm((768,), eps=1e-12, elementwise_affine=True)
       (layernorm_after): LayerNorm((768,), eps=1e-12, elementwise_affine=True)
      )
     )
    )
    (layernorm): LayerNorm((768,), eps=1e-12, elementwise_affine=True)
    (pooler): ViTPooler(
     (dense): Linear(in_features=768, out_features=768, bias=True)
     (activation): Tanh()
    )
   )
   (head): Sequential(
    (0): Linear(in_features=768, out_features=1782, bias=True)
    (1): Conv1d(197, 60, kernel_size=(1,), stride=(1,))
   )
  )
```