



**FACULTY
OF MATHEMATICS
AND PHYSICS**
Charles University

BACHELOR THESIS

Jiří Mayer

Optical Music Recognition using Deep Neural Networks

Institute of Formal and Applied Linguistics

Supervisor of the bachelor thesis: doc. RNDr. Pavel Pecina, Ph.D.

Study programme: Computer Science

Study branch: General Computer Science

Prague 2020

I declare that I carried out this bachelor thesis independently, and only with the cited sources, literature and other professional sources. It has not been used to obtain another or the same degree.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Sb., the Copyright Act, as amended, in particular the fact that the Charles University has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

In date
Author's signature

I want to thank my supervisor, doc. RNDr. Pavel Pecina, Ph.D., for the time and support he gave me. It would be difficult for me to finish this thesis without his guidance. My thanks also belong to my parents and the rest of my family. Their support is what allows me to study computer science.

Title: Optical Music Recognition using Deep Neural Networks

Author: Jiří Mayer

Department: Institute of Formal and Applied Linguistics

Supervisor: doc. RNDr. Pavel Pecina, Ph.D., Institute of Formal and Applied Linguistics

Abstract: Optical music recognition is a challenging field similar in many ways to optical text recognition. It brings, however, many challenges that traditional pipeline-based recognition systems struggle with. The end-to-end approach has proven to be superior in the domain of handwritten text recognition. We tried to apply this approach to the field of OMR. Specifically, we focused on handwritten music recognition. To resolve the lack of training data, we developed an engraving system for handwritten music called Mashcima. This engraving system is successful at mimicking the style of the CVC-MUSCIMA dataset. We evaluated our model on a portion of the CVC-MUSCIMA dataset and the approach seems to be promising.

Keywords: optical music recognition, handwritten music recognition, deep neural network

Contents

1	Introduction	3
1.1	Thesis Outline	5
2	Related Work	6
2.1	CVC-MUSCIMA Dataset	6
2.2	MUSCIMA++ Dataset	6
2.3	End-to-End OMR and the PrIMuS Dataset	7
2.4	HMR Baseline Article	8
3	Deep Neural Networks	9
3.1	Traditional Approaches	9
3.2	Deep Learning Approaches	10
3.3	Our Architecture	10
3.4	Neural Network	12
3.5	Convolutional Neural Network	13
3.6	Recurrent Neural Network	14
3.7	Connectionist Temporal Classification	15
4	Music Representation	17
4.1	PrIMuS Agnostic Encoding	17
4.2	Mashcima Music Encoding	18
4.2.1	Notes And Pitches	18
4.2.2	Rests And Barlines	20
4.2.3	Generic Tokens	21
4.2.4	Attachments	21
4.2.5	Slurs	23
4.2.6	Beams	24
4.2.7	Key Signatures and Time Signatures	25
4.2.8	Clefs and Repeats	25
4.3	Differences to PrIMuS	26
4.4	From PrIMuS Agnostic to Mashcima	26
4.5	Extensibility	29
5	Engraving System	30
5.1	Why Custom Engraving System	30
5.2	Requirements	31
5.3	Token Groups and Canvas Items	31
5.4	Rendering Flow	32
5.5	Slurs and Beams	33
5.6	Multi-staff images	34
5.7	Additional Deformation and Normalization	34
5.8	Parameters	34
5.9	Extensibility	35

6	Experiments and Results	37
6.1	Training Data	37
6.1.1	PrIMuS Incipits	37
6.1.2	Synthetic Incipits	38
6.2	Evaluation Data	39
6.2.1	Manual Annotation Experience	43
6.3	Evaluation Metrics	44
6.3.1	Understanding Model Mistakes	45
6.4	Architecture, Training, and Evaluation	46
6.5	Experiments	48
6.6	Results	51
6.6.1	Language Model	54
6.7	Comparison to Other Works	54
6.8	Evaluating on Printed PrIMuS Incipits	58
7	Conclusion and Future Work	59
	Bibliography	61

1. Introduction

Optical music recognition (OMR) is an interesting subfield of computer vision. It shares a lot of similarities to optical character recognition (OCR) and handwritten text recognition (HTR). It is, however, more challenging as is pointed out in the paper *Understanding Optical Music Recognition* (Calvo-Zaragoza et al. [2020]). For example in OCR, characters are read in one direction, typically from left to right. Musical symbols seem to be similar in that a staff is also read from left to right, but many symbols can be placed above each other. Piano scores can even have symbols that span multiple staves.

Although a musical score can be very complex, many scores are not. We can limit ourselves to scores that are monophonic, have a single voice, and have symbols spanning only one staff. Monophonic scores lack chords, meaning there is only one note playing at a time. This holds, for example, for windblown instruments, since they cannot play multiple notes simultaneously. Sometimes multiple voices (instruments) are engraved in a single staff to save space. We will not attempt to read these scores either. It would be like reading two lines of text simultaneously and the proposed model can output only a single sequence. Also deciding what voice a given note belongs to is in itself a complicated problem.

Deep neural networks have transformed the field of computer vision recently. Especially convolutional networks (CNN), whose architecture is particularly well suited for image processing. Recurrent neural networks (RNN) have been used for sequence processing, like natural language modeling or natural language translation. We can combine these two architectures to create a so-called RCNN network. When trained using connectionist temporal classification (CTC), we get a powerful architecture that is ideal for processing visual sequential data (Puigcerver [2017]). This architecture has been used in handwritten text recognition to yield state-of-the-art results (Scheidl [2018]).

If we limit the complexity of musical scores to the point that a single staff can be represented as a sequence of tokens, we can use this architecture to tackle the problem of OMR. This approach has been tried in 2018 by Calvo-Zaragoza and Rizo (Calvo-Zaragoza and Rizo [2018]). They created the PrIMuS dataset, which contains 87678 real-music incipits. An incipit is the part of a melody or a musical work that is most recognizable for that work. Each incipit is a few measures long, typically shorter than a single staff of printed sheet music would be.

The resulting model has been compared against Audiveris¹, an open-source OMR tool, and has proven to be superior on the PrIMuS dataset. However, the dataset contains printed images only. Since this RCNN architecture is an end-to-end approach, there is a great chance that it would be ideal for reading handwritten scores as well (drawing analogy from HTR).

Therefore the goal of this thesis is to explore the end-to-end approach for optical music recognition of handwritten music scores. More specifically we want to train an RCNN network to yield the best possible results on the CVC-MUSCIMA dataset.

We needed to obtain training data. We explored the *Collection of datasets*

¹<https://github.com/Audiveris>

proposed a new Mashcima representation for the music engraved in a staff. This representation is based on the agnostic encoding proposed by Calvo-Zaragoza and Rizo (Calvo-Zaragoza and Rizo [2018]). Using custom representation makes it yet more difficult to compare our results to other works. That being said, we can still make some comparisons. It seems that having a specialized engraving system is a step in the right direction. The results we obtained when evaluating are comparable to similar works performing similar evaluation (Baró et al. [2019]).

The thesis assignment states that the output of our model will be a MusicXML file. We quickly realized that the problem is far larger than anticipated and so we focused on the core features only. Similarly, the model input is not a plain photo or scan. It is already preprocessed and binarized. This problem has already been solved during the creation of the CVC-MUSCIMA dataset (Fornés et al. [2011]), therefore we did not tackle it either.

Also, there is a Github repository containing all the source code and text of this thesis at <https://github.com/Jirka-Mayer/BachelorThesis>. There is also a release tag corresponding to the time this thesis was submitted and it contains all the trained models for download.

1.1 Thesis Outline

Chapter 1: This chapter provides an introduction to the problem this thesis attempts to solve. It describes the approach we tried and how successful it was.

Chapter 2: This chapter lists work done by other people that we build upon and that will be referenced throughout the thesis. It provides a short overview of each work and of what significance it is to us.

Chapter 3: This chapter describes the specific model we decided to use for our OMR task. It discusses traditional methods and how deep neural networks help us simplify the process. It describes models other people used for similar tasks and how we have been influenced by them.

Chapter 4: This chapter mainly describes the Mashcima music encoding - the encoding we used for our model. It describes how it relates to the PrIMuS agnostic encoding, and why we made certain decisions regarding its design.

Chapter 5: This chapter talks about the Mashcima engraving system. Why we developed this system and what problem it solves. How it works, what are its limitations, and how it can be extended.

Chapter 6: This chapter describes the experiments we performed. These experiments aim to measure the performance of our approach and test hypotheses postulated in previous chapters. We will also attempt to compare our results to other similar works.

2. Related Work

2.1 CVC-MUSCIMA Dataset

CVC-MUSCIMA is a dataset presented in the article: *CVC-MUSCIMA: A ground truth of handwritten music score images for writer identification and staff removal* (Fornés et al. [2011]). This dataset contains 1000 sheets of music, consisting of 20 pages, each written by 50 different musicians. It is the only publicly available dataset containing entire staves of handwritten music. The dataset has been designed for writer identification and staff (staff line) removal tasks. It contains two sets of images. One set for writer identification (containing gray, binary, and staff-less binary images) and one set for staff removal (contains raw, staff-less, and staff-only images, all binary).



Figure 2.1: One sheet of music from the CVC-MUSCIMA dataset. Image taken from the CVC-MUSCIMA website http://www.cvc.uab.es/cvcmusicma/index_database.html.

We will use part of the staff removal set for evaluation. We will also use another part of the staff removal set for engraving, but indirectly via the MUSCIMA++ dataset.

2.2 MUSCIMA++ Dataset

MUSCIMA++ is a dataset developed by Jan Hajič jr. and Pavel Pecina and has been presented in the article: *In Search of a Dataset for Handwritten Optical Music Recognition: Introducing MUSCIMA++* (Hajič jr. and Pecina [2017]). This dataset provides additional information for a subset of the CVC-MUSCIMA dataset. MUSCIMA++ contains 140 sheets of music. Each sheet is annotated at the level of individual symbols (noteheads, stems, flags, beams, slurs, staff lines). Each one of these symbols is classified, contains a bounding box and a pixel mask.

These symbols are then interlinked in a graph that can be traversed to extract higher-level objects (notes, key signatures, beamed note groups).

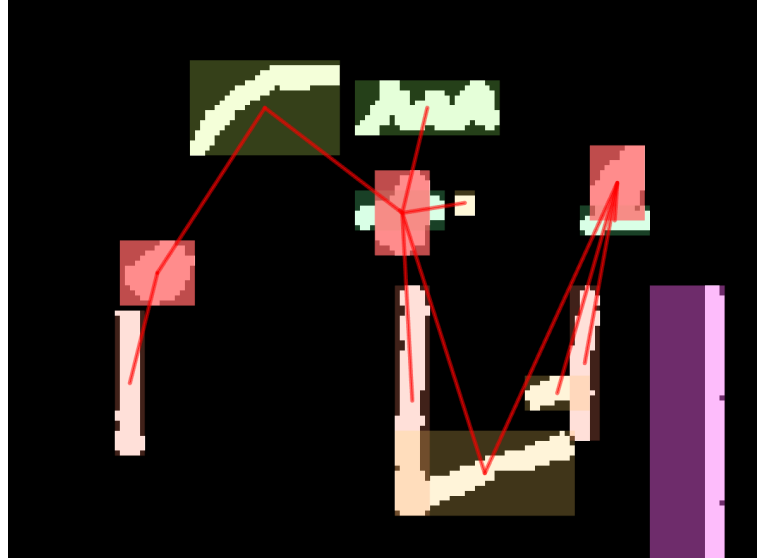


Figure 2.2: Notation graph of the MUSCIMA++ dataset. Image taken from Hajič jr. and Pecina [2017].

We will use the dataset as a collection of musical symbols. We will then place those symbols onto an empty staff to create synthetic training data. The additional data (relationship graph) will help us position certain symbols properly.

2.3 End-to-End OMR and the PrIMuS Dataset

This section refers to the article: *End-to-End Neural Optical Music Recognition of Monophonic Scores* (Calvo-Zaragoza and Rizo [2018]). This article first describes the PrIMuS¹ dataset. This dataset contains 87678 real-music incipits. An incipit is the part of a melody or a musical work that is most recognizable for that work. Each incipit is a few measures long, typically shorter than a single staff of printed sheet music. Each incipit is encoded in a few widely known encodings (MEI, MIDI) and has a corresponding printed image. This image has been engraved using the music notation engraving library Verovio². Each incipit is also encoded using two on-purpose devised encodings — the PrIMuS semantic and agnostic encoding. These encodings are interesting because they are the output of a model proposed in the article, but also the Mashcima encoding described in chapter 4 of this thesis is very similar to the agnostic encoding.

The article also proposes a neural network architecture for an end-to-end solution of OMR. The architecture is very similar to ours, almost identical. It also uses the connectionist temporal classification as the loss function (Graves et al. [2006]), which shapes the PrIMuS dataset encoding formats. Our thesis differs from this article mainly in the focus on handwritten music and the introduction of a custom engraving system for handwritten music. The PrIMuS article focuses on printed music only.

¹<https://grfia.dlsi.ua.es/primus/>

²<https://www.verovio.org/>

3. Deep Neural Networks

This chapter talks mainly about the model we decided to use. First, we describe the full pipeline of a traditional OMR system. Many of these steps are shared between traditional and deep learning approaches. Then we will talk about the deep learning approaches that can be taken. Neural networks can replace parts of a traditional pipeline, or they can be used in an end-to-end setting, where the neural network replaces the most difficult core of the pipeline. We will describe our architecture consisting of a convolutional block, recurrent block, and the connectionist temporal classification (CTC) loss function. The following sections describe in more detail what a neural network is and how the individual blocks of our model work internally. The last section explains how CTC works and what are its pros and cons, compared to the approach described in the HMR baseline article (Baró et al. [2019]).

3.1 Traditional Approaches

A musical score intended for OMR typically begins as a raster image. This image is a photo or a scan of a real-world sheet of paper. The image needs to be prepared first. We need to find the sheet of paper in the image and correct any rotation or perspective distortion. Scanned images are easier to prepare because they do not contain any perspective deformation and lighting artifacts. Searching for the paper in the image can be performed using many approaches, e.g. by using maximally stable extremal regions (Matas et al. [2004]). We can detect staff lines using Hough transform (Hough [1962]). We can then use this information to remove any affine distortion of the image.

The next step is performing some color normalization and binarization. There might be a light-intensity gradient over the image, so we do some automatic contrasting to bring the lightness to a constant level across the image. Median filtering can be applied to remove noise (Huang et al. [1979]). Conversion to a grayscale image is often used since colors are not useful to OMR. The image can then be binarized to further remove unnecessary information. Many thresholding algorithms can be used for this step, many of which are implemented in the OpenCV¹ library. Binarization is important for traditional approaches since they often use methods based on connected components to detect individual symbols. Neural networks could benefit from non-binarized images since binarization can create aliasing artifacts that distort the input image on the pixel level.

The steps described above are shared by both traditional and neural network-based approaches. Traditional approaches now usually perform staff line removal. This step lets methods based on connected components to become useful. Staff localization may be an important part of this step. Symbols then need to be segmented and classified separately. Meaning is then reconstructed by looking at the relationships between all the classified symbols. With the musical score understood at the symbol level, the extracted information can be converted to some final representation (MusicXML, MEI, MIDI).

¹<https://opencv.org/>

3.2 Deep Learning Approaches

Deep learning is a class of machine learning that focuses on deep neural networks. Deep learning has risen over the past two decades and became a very powerful tool for solving many problems, especially classification problems regarding computer vision. Neural networks can be used in many places throughout the pipeline of a traditional OMR system. They can be used for staff line removal (Calvo-Zaragoza et al. [2017]), symbol classification (Lee et al. [2016]), or even symbol detection (Pacha et al. [2018]).

Recently, neural networks have been used to tackle the problem of OMR in an end-to-end fashion (Calvo-Zaragoza and Rizo [2018], Baró et al. [2019]). This approach allows us to replace many stages of the pipeline with a single model. The input sheet of music is usually processed staff by staff, so an initial segmentation of staves is required. This step is, however, very robust and can be performed reliably.

The main steps unified by an end-to-end system are segmentation, symbol classification, and part of the relationship extraction. This means we do not need to explicitly specify the structure of this part of the pipeline, which saves a lot of time and thinking. Also, any intermediate features that would be extracted (like noteheads) need not be specified. The deep neural network can learn, what those features are. Moreover, it can adapt these features to the problem better than a human could.

Deep learning, especially in an end-to-end approach also has some drawbacks. The first is bound to the ability of the model to learn the solution from data. While it is very helpful, that we do not have to design part of our OMR system manually, it is often very difficult to acquire enough high-quality data for the training. Also, the more complex our model is and the more learned parameters it has, the more training data it requires. The data also needs to be of high quality. Ambiguity and mistakes in annotations lead to poor performance of the resulting model. The trained model can only ever be as good as its training data.

The second drawback is the very difficult nature of debugging the model. A neural network is by design a black box and we cannot easily assign specific meaning to any of its internal parts. The process of fixing a mistake the model makes is tedious and requires a lot of experimentation and re-training.

3.3 Our Architecture

As stated in the title of this thesis, we decided to explore the end-to-end approach to OMR using deep neural networks. We were primarily inspired by these three models:

- End-to-End Neural Optical Music Recognition of Monophonic Scores by Calvo-Zaragoza and Rizo (Calvo-Zaragoza and Rizo [2018])
- SimpleHTR by Harald Scheidl²

²<https://github.com/githubharald/SimpleHTR>

- From Optical Music Recognition to Handwritten Music Recognition: A baseline (Baró et al. [2019])

All of these models share the same high-level structure. They combine a convolutional neural network (CNN) with a recurrent neural network (RNN). This combination is sometimes called RCNN architecture. Convolutional neural networks are used in image processing. Their architecture is inspired by the way filters work in computer graphics (convolving a kernel over the source image). They learn to extract edges, corners, and then even more abstract features like noteheads and stems. Recurrent neural networks are used for sequence processing (text and speech). They have been designed to carry state information throughout the input sequence. In our case, they learn to propagate information horizontally — like inferring pitch of an accidental from the pitch of a neighboring note. The CNN block learns to extract features that the RNN block then learns to combine into more abstract features.

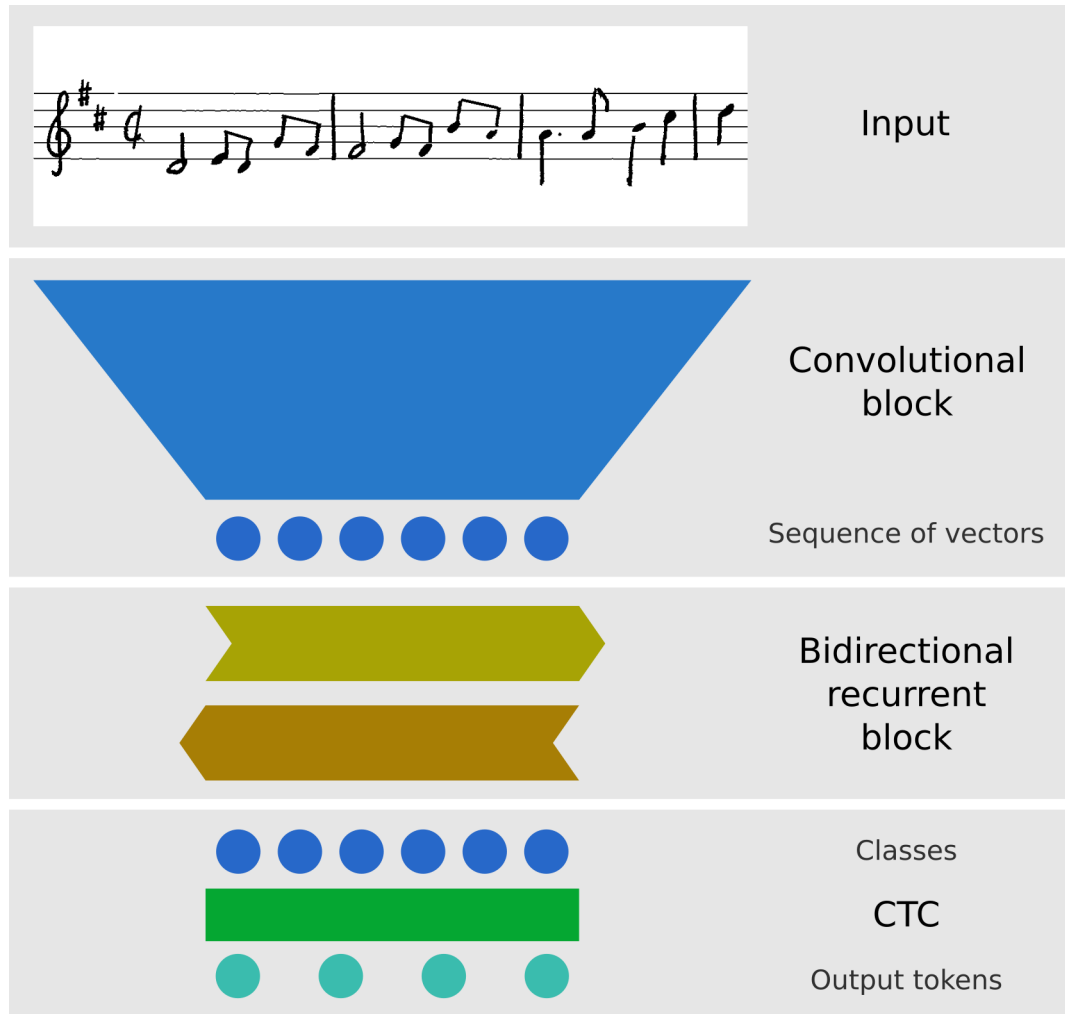


Figure 3.1: Diagram of our model — a RCNN network with CTC loss function. Detailed list of layers with all parameters can be found in table 6.5

The RNN block can be followed by fully connected layers that further refine the result, although these layers are not necessary and our architecture does not contain them. This may be due to the fact that our encoding is very close to

the symbolic visual representation and so most of the heavy lifting is probably performed in the CNN block.

The final layer outputs a sequence of vectors, where each vector represents one time-step (horizontal slice of the input image). Values in such vector correspond to probabilities of individual output classes (tokens) at that given time-step. One additional class *blank* is added, which represents "no symbol present". The most likely class for each time-step is selected and then repetitions of the same class are collapsed into one token. Lastly, all the blank symbols are removed. The remaining sequence of classes is mapped directly onto annotation tokens of the Mashcima encoding explained in the chapter 4. This approach is called greedy CTC decoding (Graves et al. [2006]) and is used during training. For evaluation, a more advanced method is used, called beam search decoding (Hwang and Sung [2016]).

When training, the loss is computed using the connectionist temporal classification. The loss function provides a gradient for improving the network. This gradient is then calculated for the entire network using the backpropagation algorithm (Goodfellow et al. [2016]). Parameters are then updated using the adaptive learning rate optimizer (Adam) (Kingma and Ba [2014]).

The values of all hyperparameters, including sizes and types of all layers, are specified in the section 6.4.

3.4 Neural Network

A neural network is a model inspired by the human brain. Its core building block is a perceptron (analogous to a neuron in the brain). Perceptron is a node that has several inputs (real numbers), combines them, and produces a single output value. The mathematical description is the following:

$$y = \varphi(w \cdot x + b)$$

Vector x contains all the input values. It is multiplied by a vector of weights w and a constant scalar bias b is added. The result is passed through an activation function φ that produces the output value y . The core idea behind this model is that a perceptron activates (fires) when enough inputs activate. Weights and the bias are parameters that allow the perceptron to learn - to detect a specific pattern in the input values. The activation function attempts to model the activation threshold and introduces non-linearity into the system.

One of the first activation functions to be used was the sigmoid function, but it suffered from the problem of vanishing and exploding gradients (Hochreiter [1991]). The hyperbolic tangent function was then used to remedy this problem. Nowadays, rectifier function is often used ($\max(0, x)$), because it is easy to compute. Perceptrons with this function are called rectified linear units (ReLU). It also has some problems, so leaky ReLU can be used to address them (Maas et al. [2013]).

Perceptrons can be interconnected to form neural networks. A typical architecture is a feedforward neural network (FNN). It organizes perceptrons into layers, through which information flows in one direction. The resulting graph is directed and acyclic, which allows us to understand the whole network as a complex mathematical function and lets us train it using the backpropagation and

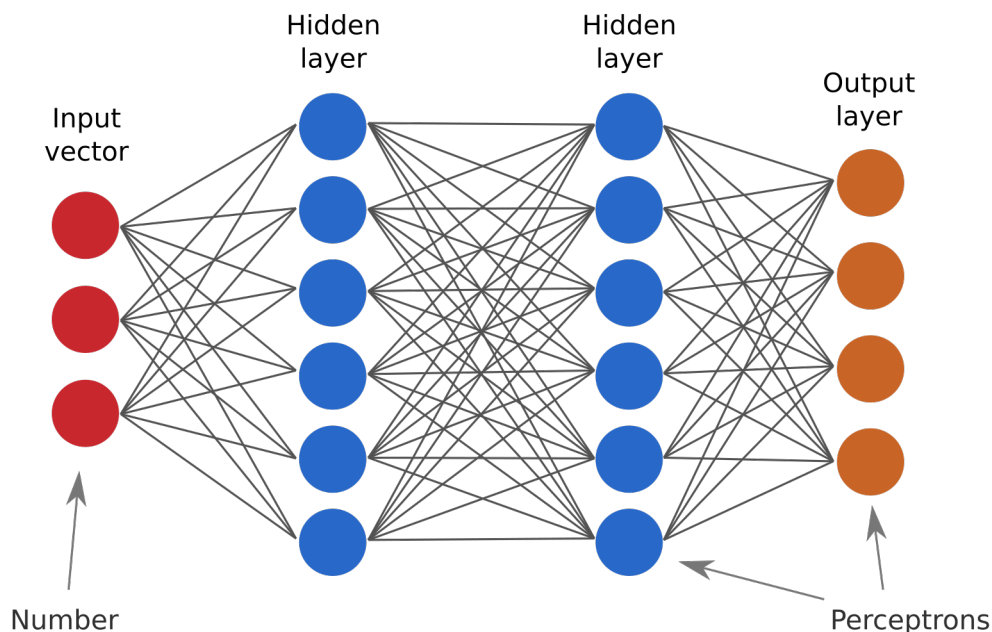


Figure 3.2: Fully-connected neural network with 3 layers.

gradient descent algorithms (Goodfellow et al. [2016]).

The simplest feedforward network is a network with dense (fully connected) layers. Each perceptron in a given layer receives input as the output of all perceptrons in the previous layer.

3.5 Convolutional Neural Network

A convolutional neural network (CNN) is a kind of feedforward network. It is ideal for processing visual data. A CNN is built primarily from two kinds of layers:

Convolutional layer A convolutional layer is similar to a fully connected layer, but the connections are only local. The input and output is a two-dimensional array of numbers (like one channel of an image). Each perceptron takes input from only a small window of neighboring perceptrons in the input layer (3×3 , 5×5 , 7×7). Weights are represented by a kernel of the same size as this neighborhood window. This kernel is shared by all the perceptrons, reducing the number of learned arguments substantially and allowing the layer to process images of variable sizes. There may be multiple input and output arrays (channels). In such a case, the convolutional layer has one kernel for each pair of channels. This architecture is inspired by the convolution filters from computer graphics — hence the name.

Pooling layer A pooling layer typically follows a convolutional layer. Its job is to downsample the image, reducing its spatial resolution, while preserving the number of channels. The downsampling is performed by splitting the input into a set of rectangular regions (that may overlap) and then reducing each region

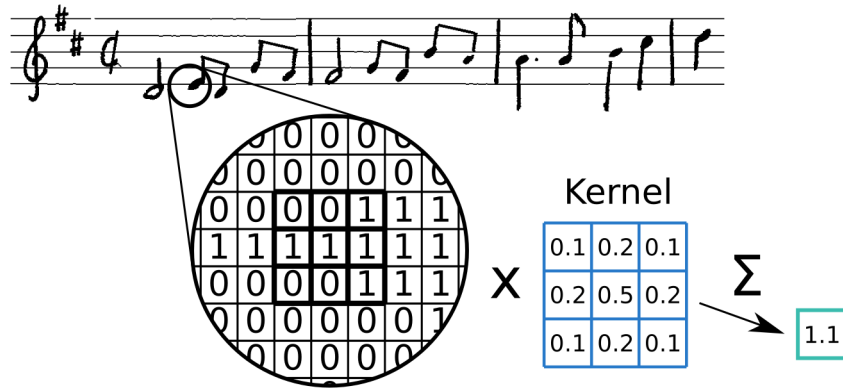


Figure 3.3: Computation of one output pixel in a convolutional layer.

using max, sum, or average operator. Widely used is the maximum operator and the resulting layer is called a *max pooling layer*. The pool size is typically 2×2 .

Fully-connected layer One or more fully connected layers can be added at the end of a CNN, to further refine features extracted by the previous layers. This layer is often present in models performing classification because we want to reduce the number of outputs to the number of classes we are predicting.

A CNN has typically many convolutional layers, combined with pooling layers. Each time the spatial dimensions shrink in a pooling layer, more channels are added to be able to represent more features. This forces the network to create abstractions and convert the visual data into some abstract representation vector.

3.6 Recurrent Neural Network

A recurrent neural network (RNN) is a network intended for sequence processing. Input for the network is a sequence of vectors and the output is a sequence of the same length. The recurrent network can be understood as composed of recurrent units, each with two inputs and two outputs. A unit accepts one vector from the input sequence and an old state vector. It outputs the corresponding vector of the output sequence and the new state vector. These recurrent units can be unrolled along the input sequence and each one passes the state vector to the next one, using it to send information along the length of the sequence. All instances of the recurrent unit share the same learned parameters and so can be unrolled to any length necessary. The sequence dimension, where unrolling happens, is called the time dimension.

The internal architecture of a recurrent unit may vary, but it is often a feedforward network, where the input and output vectors are both split into a sequence part and a state part. The most common recurrent unit architectures are the long short-term memory (LSTM) (Hochreiter and Schmidhuber [1997]) and the gated recurrent unit (GRU) (Cho et al. [2014]). You can join two recurrent layers, passing information in opposite directions, to create a bidirectional recurrent layer.

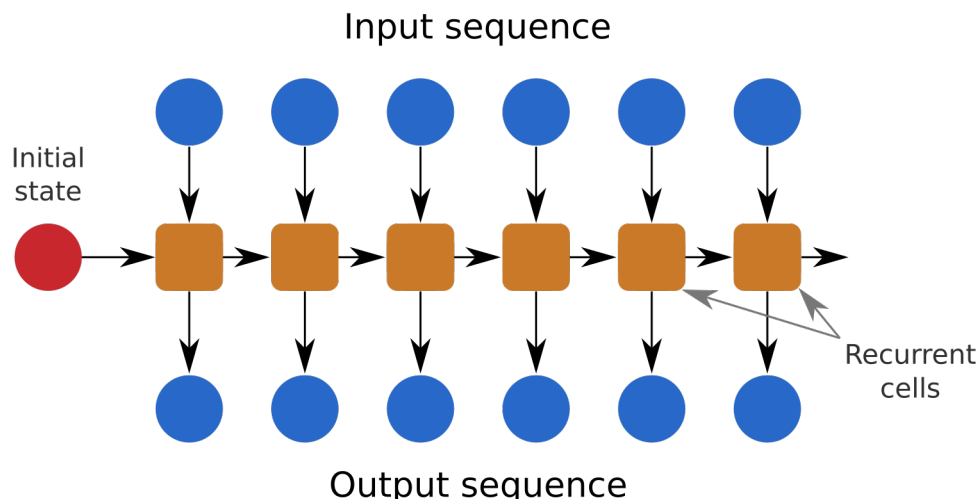


Figure 3.4: Unrolled recurrent units processing a sequence of vectors.

3.7 Connectionist Temporal Classification

A recurrent neural network outputs a sequence of vectors. If these vectors are the final output of the model, they typically represent class probabilities for a given time-step (we solve sequence classification). If we did OCR, we would have a set of characters (an alphabet) and the vector would have the same size as the alphabet. Each value in that vector would correspond to the probability of that given character being present at that given time-step. We could use the softmax function (Goodfellow et al. [2016]) to convert the perceptron activations to probabilities.

This creates a problem when training. In OCR, we want to produce a sequence of letters as the output. But one letter might span multiple time-steps in the input. This means we not only need the gold sequence of letters, but we also need to know at which specific time-steps the letters are present. This mapping of output classes to specific time-steps is called *alignment* and it complicates the creation of a training dataset.

Connectionist temporal classification (CTC) (Graves et al. [2006]) is an approach that solves the alignment issue. The CTC loss function can calculate the loss value over all possible alignments of our gold sequence over the output sequence. This allows us to train the model, without having explicit alignment.

When the model is trained, the greedy decoding algorithm (Graves et al. [2006]) can be used to convert the output vector sequence to the proper prediction. This algorithm takes each vector of the output sequence and selects the most probable output class (letter). It then collapses all repetitions of that class into one occurrence, producing the final sequence. One special output class called *blank* is introduced, to prevent two actual successive occurrences of a given class from being collapsed into one. This symbol is, however, after collapsing removed and will never be present in the final prediction.

There is also the option to use the beam search decoding algorithm (Hwang and Sung [2016]). This algorithm keeps a list of best decodings so far, as the entire decoding is being computed. Greedy decoding is a special case of the beam search

decoding, where the list contains only one item. All partial decodings cannot be considered, because there are exponentially many of them.

The connectionist temporal classification has many benefits regarding the alignment problem, but it has some flaws as well. The loss computation and the decoding rely on the fact, that there will be only one class predicted for each time-step. This means we cannot capture the fact of two symbols appearing in the input at the same time. This happens often with music. We can solve this problem partially by utilizing the recurrent layers. They will allow us to output simultaneous symbols sequentially, by remembering those symbols for a while. This however increases the length of the final sequence, but this sequence can never be longer than the total number of time-steps. We might run out of temporal resolution.

The HMR baseline article (Baró et al. [2019]) opted not to use CTC and performed manual alignment instead. This lets them have a model that can predict multiple symbols simultaneously. This also reduces the complexity of the output sequence and the model thus need not perform much additional work as in our case (see the section 4.2.4 on attachment ordering).

4. Music Representation

This chapter explores how music is represented in this thesis. It looks at the encodings devised for the PrIMuS dataset and how they have been modified to produce our Mashcima music encoding. Then we explore how this encoding can be used for annotating datasets and how it can be extended in the future.

All the encodings explored in this chapter are made for a model that produces a sequence of tokens. An encoding then defines a specific set of tokens and describes how they map onto the musical symbols. In the context of a neural network with a CTC loss function, we take all the tokens of an encoding and represent them as the individual classifier classes. How the tokens get indexed and how the blank symbol is represented is considered an implementation detail of the neural network and is not covered in the encoding specification.

We can provide a short overview of the terms used in this chapter:

- **Token** is a single item of the output sequence produced by a model.
- **Vocabulary** is the set of all tokens in an encoding.
- **Encoding** is a scheme for mapping musical staves onto a sequence of tokens.
- **Annotation** is a specific sequence of tokens.

4.1 PrIMuS Agnostic Encoding

The PrIMuS dataset contains over 87 000 items, each with an image of the printed staff and then multiple files describing the music in that staff. There are two standard formats, namely Music Encoding Initiative format (MEI) and the Plaine and Easie code source. Then there are two on-purpose encodings devised specifically for this dataset. These two encodings are what interests us.



Figure 4.1: An incipit from the PrIMuS dataset.

The first of these two encodings is the *semantic encoding*. It represents what the musical symbols mean. Each symbol has a specific pitch that relies on the clef at the beginning of the staff. This makes the vocabulary much larger and any model using this encoding has to take the clef into account when reading the symbols. It is however much easier to transform this encoding to some well-known format like MusicXML since these formats tend to represent the meaning of a score, not the score appearance.

The second encoding is the *agnostic encoding*. This encoding treats the staff visually as a specific positioning of specific symbols. It tries to capture what is in the staff visually, not what the symbols mean musically. This is comparable to a sentence being thought of as a sequence of letters, whereas the semantic

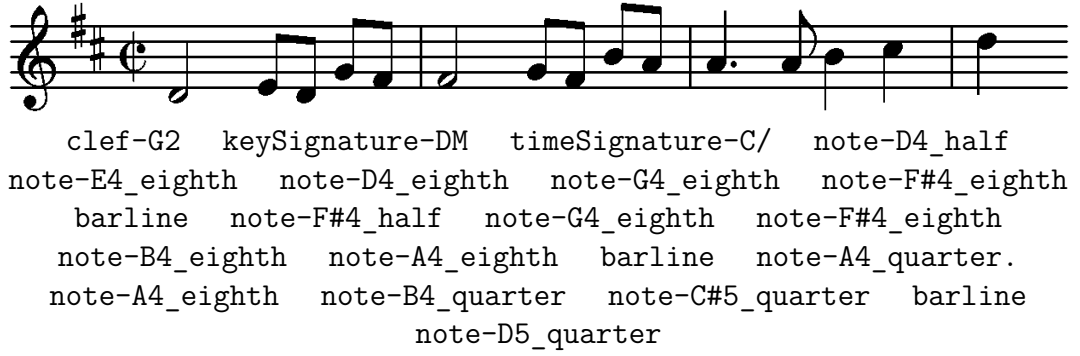


Figure 4.2: Semantic encoding of the incipit from figure 4.1.

encoding could be thought of as the specific sounds a written sentence represents. This makes the encoding harder to convert to a well-known format acceptable by other music software. On the other hand, this encoding is formal-enough to be easily converted to the semantic encoding, if read correctly. So this encoding lets the model do less work, therefore the model should do fewer mistakes.

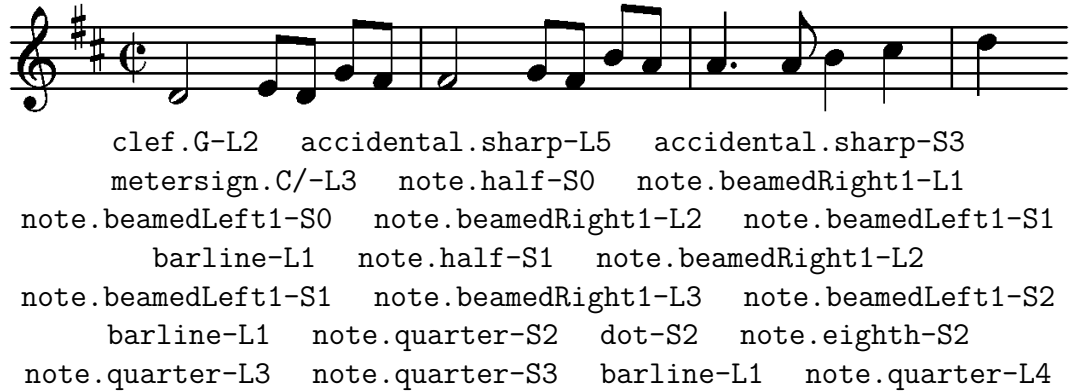


Figure 4.3: Agnostic encoding of the incipit from figure 4.1.

The agnostic encoding has also the advantage, that annotating an image is not as difficult for a human. Annotating an image using the semantic encoding requires the annotator to know pitches for a given key. The situation is even more complicated by key signatures. This means an untrained non-musician has to do a lot of thinking when annotating, which leads to many errors and slow annotation speed.

We have taken this agnostic encoding and modified it slightly to produce our Mashcima music representation.

4.2 Mashcima Music Encoding

4.2.1 Notes And Pitches

Mashcima music encoding is an encoding that attempts to improve upon the PrIMuS agnostic encoding. In the source code, most of the logic regarding this encoding is placed inside the `app/vocabulary.py` file. Each token of this encoding represents some musical symbol.



```
clef.G-2 #4 #1 time.C/ h-5 e=-4 =e-5 e=-2 =e-3 |
h-3 e=-2 =e-3 e=0 =e-1 | q-1 * e-1 q0 q1 | q2
```

Figure 4.4: Mashcima encoding of the incipit from figure 4.1. The top image is taken from the PRIMuS dataset. The bottom image is the same music engraved using our engraving system. Below that is the Mashcima encoding of the music.

The first symbol we need to encode is a note. A note has some duration and some pitch. These two pieces of information can vary independently, so it can seem logical to represent them using two vectors. The problem is that the connectionist temporal classification can output only one vector at a time. To solve this, we take every possible combination of note duration and pitch and create a token for that case.

Mashcima token	Duration	Pitch
w5	Whole note	5
h0	Half note	0
q-8	Quarter note	-8
e-4	Eighth note	-4
s9	Sixteenth note	9
t12	Thirty-second note	12

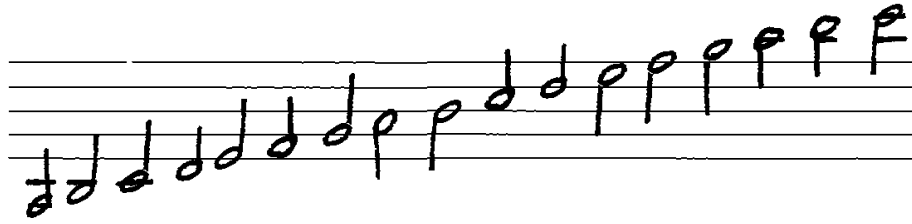
Table 4.1: All possible note durations, each with some pitch.

Combining duration information and pitch information into a single token actually ends up being a reasonable solution. That is because the concept of note duration can be extended to a concept of symbol type in general. This is because not only notes have pitches.

The set of pitches we can choose from greatly impacts the vocabulary size. This is not a major issue, because the vocabulary size will still remain relatively small. Currently, the vocabulary has around 550 tokens. The pitch range we choose spans from -12 to 12 — that is from the fourth ledger line below the staff to the fourth ledger line above the staff.

The pitch encoding is built such that it would be easy to understand for a non-musician. In western music notation, the pitch of a note is represented by the vertical position of that note on the staff. An empty staff is composed of 5 staff lines. Mashcima encoding sets the middle staff line position to be zero. Going

up, the first space is pitch 1 and the first line is pitch 2. Going down, the first space is pitch -1 and the first line is pitch -2.



h-8 h-7 h-6 h-5 h-4 h-3 h-2 h-1 h0 h1 h2 h3 h4 h5 h6 h7 h8

Figure 4.5: Half notes with pitches from -8 to 8 engraved using Mashcima. You can see the stem orientation being randomized for pitches between -2 and 2.

This pitch encoding has the advantage of being vertically symmetric, which speeds up the manual annotation process. The first ledger line above the staff is pitch 6, and the first ledger line below is pitch -6. The second property this system has is that pitches placed on lines are even and pitches placed in spaces are odd.

4.2.2 Rests And Barlines

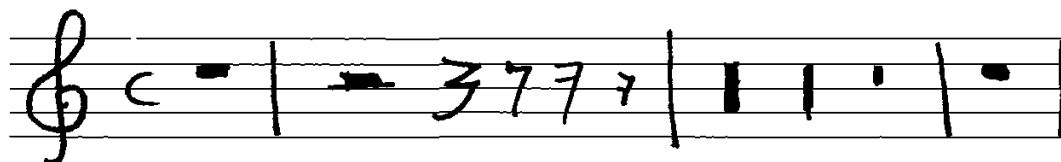
The second most common symbol is probably a rest. A rest has duration, just like a note, but it has no pitch information. Its vertical position may vary, but that does not encode any pitch information.

Mashcima token	Duration
lr	Longa rest (4 measures)
br	Breve rest (2 measures)
wr	Whole rest
hr	Half rest
qr	Quarter rest
er	Eighth rest
sr	Sixteenth rest
tr	Thirty-second rest

Table 4.2: Tokens for all rests representable by the Mashcima encoding.

You may have noticed, that there are two extra durations — longa and breve. Also, there is missing the sixty-fourth rest. It all has to do with the fact that not all durations are used equally frequently. I based Mashcima on the CVC-MUSCIMA and MUSCIMA++ datasets. There is no occurrence of longa or breve notes in those datasets. There are, however, occurrences of longa and breve rests. Similarly, sixty-fourth notes and rests are also not present. The vocabulary can luckily be extended to accommodate these symbols. See the section 4.5 for more details.

Now that we have notes and rests, we can start grouping them into measures (bars). A barline is represented by the "pipe" character (`|`). Barlines separate notes and rests into groups of the same total duration. There are many types of barlines (double barline, repeat signs) and although they are used quite often, they have not yet been implemented into the Mashcima engraving system. This is simply because we wanted to see, whether our approach even works. These special barline types can be easily added in the future.



```
clef.G-2 time.C wr | hr qr er sr sr | lr lr br | wr |
```

Figure 4.6: Rests and barlines engraved using Mashcima. The symbols were taken from all writers and chosen randomly, therefore the two sixteenth rests look very different.

4.2.3 Generic Tokens

There are some tokens that contain pitch information and some that do not. Since the pitch information is often not required when inspecting a token sequence, it is useful to strip it away. This is why we define a *generic token* as a version of a token without pitch information. So for example a generic quarter note is represented by the `q` token.

Generic tokens are not present in the vocabulary and cannot be produced by the model. They should also never appear in the gold data. They are, however, often used when analyzing a given token sequence.

The only exception is tokens that do not contain pitch information (e.g. rests). They are considered to be their own generic token (i.e. the generic token for a quarter rest `qr` is still just `qr`).

The vocabulary file (`app/vocabulary.py`) contains helper methods for working with pitches:

- `to_generic(token: str) -> str`
Obtains generic version of a token unless the given token is already generic.
- `get_pitch(token: str) -> Optional[int]`
Obtains pitch of a token, or `None` if that token has no pitch.

4.2.4 Attachments

It is often the case, that notes are decorated with symbols that slightly modify their meaning. Since these decorating symbols are bound to the note itself, we

call them *attachments*. An attachment token is simply a token that belongs to some other non-attachment token.

Many kinds of musical symbols behave as attachments:

- **Accidentals** are symbols placed before a note and they modify their pitch by a semi-tone.
- **Duration dots** are placed after a note and they extend the duration of a note.
- **Articulation symbols** are usually placed below or above a note and they specify how the note should be played (e.g. staccato, tenuto, accent).
- **Other symbols**, like a tuplet number, fermata, trill, etc.
- **Artificial tokens**, that we have added to encode specific time-spanning symbols. See the section 4.2.5 on slurs to learn more.

You can see, that the term *attachment* is not a musical term and it describes more how a symbol is represented, not what a symbol means.

All of these attachments lack pitch information since the pitch is stored in the note token. The only exception here is accidentals. Accidentals are special because they need not be attached to a note. They can be standalone in a key signature. This means that they need pitch information. This creates some redundancy in the encoding; when a note has an accidental, they both should have the same pitch. This condition is not ideal, because it may cause the model to make unnecessary errors. It is however better than having different tokens for standalone accidentals and attached accidentals.

Mashcima token	Accidental	Pitch
#5	Sharp	5
b4	Flat	4
N-4	Natural	-4
x8	Double sharp	8
bb0	Double flat	0

Table 4.3: All accidentals, each with some pitch.

Attachments come in two kinds:

- **Before attachments** are placed before the target token
- **After attachments** are placed after the target token

By placement, we mean placement in the token sequence. It may not correspond to the visual order of the symbols. The rule of thumb here is that tokens are ordered from left to right and from top to bottom. The problem is that some symbols may be both above and below a note, depending on the note pitch. Therefore we did not make this into a strict rule and instead devised a specific ordering of the attachments:

Here are a few notes regarding the ordering:

Before attachments	Meaning
)	Slur end
fermata	Fermata
trill +	Trill
tuplet.3	Tuplet number
# b N x bb	Accidentals

Table 4.4: Before attachments, properly ordered.

After attachments	Meaning
.	Staccato
-	Tenuto
>	Accent
^	Marcato
* **	Duration dots
(Slur start

Table 4.5: After attachments, properly ordered.

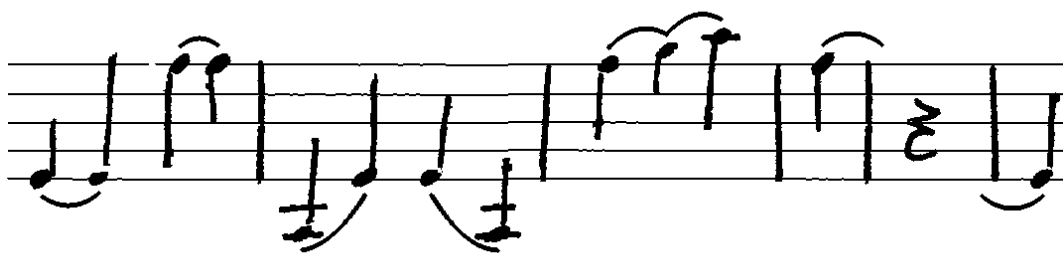
- Slurs are always the first/last attachment.
- Some tokens are mutually exclusive, so they are placed on the same level.
- There are many ornaments and this list is not exhaustive. It is meant to be extended in the future.
- Not all symbols here can be engraved by Mashcima.
- There can be many more tuplet numbers, only triplets are currently present.

4.2.5 Slurs

Slurs and ties are one of the first symbols that make OMR complicated. A slur is a curved line going from one notehead to another. Notes that are under a slur should be played blended without explicit note beginnings. A tie looks exactly like a slur, just the two notes it joins have the same pitch. This means the notes should be played as one long note. So the difference is only semantic, we will consider ties to be just like slurs.

Mashcima encoding does not represent slurs explicitly, but it represents their beginnings and ends. This is accomplished by two attachment tokens, (and). The problem it creates is that sometimes it is impossible to pair beginnings and ends properly. Therefore we can only annotate staves that do not contain nested slurs.

Slurs can also span from one staff to the next. When this happens, the slur ends at a barline. Therefore a barline can also act as a token, on which a slur can start or end.



q-4 () q-4 q4 () q4 | q-8 () q-4 q-4 () q-8 |
q4 () q5 () q6 | q4 () | qr | () q-4

Figure 4.7: Various slur situations, engraved using Mashcima. The first two slurs are actually ties. One note is simultaneously a slur ending and a slur beginning. Two slurs end on a barline.

4.2.6 Beams

Beamed notes pose similar problems as slurs, but they obey some additional constraints that make them easier to deal with. Beams are encoded from the perspective of a single note within the beamed group. The duration of this note depends only on the number of beams passing through its stem. We just need to distinguish the note from its flag variant, therefore we add some information regarding the beam presence. The last problem is, that having a couple of beamed notes in a row does not tell us about the way the beams are grouped. Therefore we modify the beam presence information into two parts — beam ending and beam starting.

Therefore for a given note duration, we get three tokens, that represent all the possible beam situations:



s=-3 =s=-1 =s=2 =s1 e=-3 =e-6 | e=-3 * =s-1 e=6 * =s4

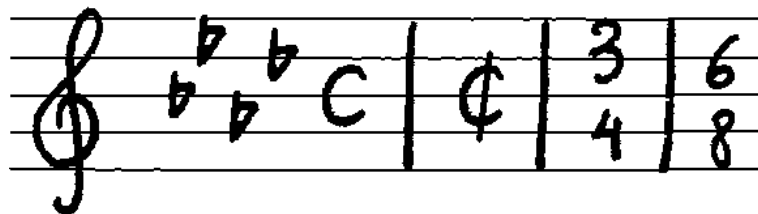
Figure 4.8: Beamed notes use the = sign in token names.

This, however, lets us create annotations that make no sense. We can create un-finished and non-started beams. For this reason, there is the `repair_annotation` function in the file `app/vocabulary.py` that can repair such situations or validate the correctness of an annotation.

4.2.7 Key Signatures and Time Signatures

A key signature is just a group of many accidentals, typically at the beginning of a staff. These accidentals are not attached to any note, so they have special handling in the source code. They are not a problem from the annotation point of view.

A time signature is either a symbol *C* or a pair of two numbers on top of each other. The standalone *C* symbol can be represented easily using the `time.C` token. A pair of numbers is represented by two tokens, e.g. `time.3 time.4`. Numbers have to be paired. Non-paired time number is considered an invalid annotation.



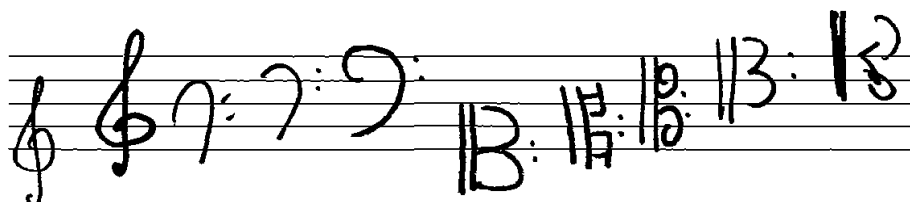
```
clef.G-2 b0 b3 b-1 b2 time.C | time.C/ |
time.3 time.4 | time.6 time.8
```

Figure 4.9: Examples of a key signature and various time signatures.

Time signature numbers are also treated specially since they always come in pairs.

4.2.8 Clefs and Repeats

Clefs are encoded similarly to rests, with the only difference of having a pitch associated. Not all pitches are allowed, though. There are three types of clefs (G, C, F). Each of these clefs has a set of pitches it can have.



```
clef.G-4 clef.G-2 clef.F0 clef.F2 clef.F3
clef.C-4 clef.C-2 clef.C0 clef.C2 clef.C4
```

Figure 4.10: Various clefs by various writers.

There are special barlines that mark a part of music to be repeated. These barlines cannot be engraved yet, but they do have corresponding tokens in the encoding: `:|`, `|:`, `:|:`.

4.3 Differences to PrIMuS

Mashcima encoding is very similar to the agnostic PrIMuS encoding, but there are some deliberate changes introduced. Mashcima encoding has on purpose shorter token names. This aims to aid readability when manually annotating. The goal is to fit one staff of tokens onto the screen. Common tokens, like quarter notes (`q5`), are also quick to type. We also decided to use characters that look like the musical symbols they encode (sharps `#5`, flats `b5`, barlines `|`).

The pitch is also encoded differently. In PrIMuS agnostic encoding, every token has a pitch — even a barline. The pitch might not be useful, but it is present, to simplify working with the encoding. We decided to leave some tokens without pitch information. That complicates the code but makes the annotation more user-friendly.

Also, the specific pitch values are different. PrIMuS indexes lines and spaces separately and line zero is the first ledger line below the staff. We tried to improve on this by putting the line zero at the center of the staff. This makes the pitch values vertically symmetric. We also removed the separation of lines and spaces and we use just an integer value. That simplifies the code that deals with pitch information. This integer value is then odd in spaces and even on lines.

4.4 From PrIMuS Agnostic to Mashcima

Conversion from PrIMuS agnostic encoding to Mashcima encoding could almost be done by a lookup table. There are, however, some differences that make it more complicated. The conversion happens on a token by token basis. First, we convert the pitch of the token according to the table 4.6. All PrIMuS tokens have a pitch but if the corresponding Mashcima token does not, we will not use it. The remaining pitch-less variant of the token can be converted using the table 4.7.

Primus contains symbols that Mashcima encoding cannot represent. Gracenotes are automatically removed since they cannot be engraved. Fermatas are also removed. Incipits containing multi-measure rests are ignored because the logic enabling their parsing has not been implemented. This allows us to treat `digit.4` as a time signature because that is the only remaining place where digits can appear. Dots have to be disambiguated by a simple heuristic to determine whether they are staccato dots or duration dots. Duration double dots have to be merged after the conversion finishes. Incipits containing too long (*longa*, *breve*) or too short notes (*thirty-second*, *5 beams*, ...) are also ignored. This is the reason why we train only on 64 000 incipits and not the full 87 678 incipits.

PrIMuS	Mashcima
S8	11
L8	10
S7	9
L7	8
S6	7
L6	6
S5	5
L5	4
S4	3
L4	2
S3	1
L3	0
S2	-1
L2	-2
S1	-3
L1	-4
S0	-5
L0	-6
S-1	-7
L-1	-8
S-2	-9
L-2	-10
S-3	-11
L-3	-12

Table 4.6: Lookup table for converting PrIMuS pitch to Mashcima pitch.

PrIMuS	Mashcima
barline	
fermata.above	fermata
clef.C	clef.C
clef.F	clef.F
clef.G	clef.G
metersign.C	time.C
metersign.C/	time.C/
digit.0	time.0
digit.1	time.1
...	...
digit.9	time.9
accidental.sharp	#
accidental.flat	b
accidental.natural	N
slur.start	(
slur.end)
dot	. or *
rest.quadruple_whole	lr
rest.double_whole	br
rest.whole	wr
rest.half	hr
rest.quarter	qr
rest.eighth	er
rest.sixteenth	sr
rest.thirty_second	tr
note.whole	w
note.half	h
note.quarter	q
note.eighth	e
note.sixteenth	s
note.thirty_second	t
note.beamedRight1	e=
note.beamedBoth1	=e=
note.beamedLeft1	=e
note.beamedRight2	s=
note.beamedBoth2	=s=
note.beamedLeft2	=s
note.beamedRight3	t=
note.beamedBoth3	=t=
note.beamedLeft3	=t

Table 4.7: Lookup table for converting PrIMuS generic tokens to Mashcima encoding.

4.5 Extensibility

Because the CVC-MUSCIMA dataset contains such a wide range of symbols, we did not want to create an encoding that would capture every possible symbol in the dataset. Most of these unusual symbols are present in only a few places and adding them to the encoding would make everything much more complicated, for little to no return. Therefore the Mashcima encoding contains the ? token. This token should be placed into gold data whenever we encounter a symbol (or a group of symbols) that cannot be represented by our encoding. This ? token acts as a marker, that we cannot fully represent a specific place in the staff. It can then be used to filter out such bars or just to find such places if the encoding was ever extended in the future.

As mentioned in previous sections, there are some missing note and rest durations. These can be easily added when needed. These new tokens would follow the rules that are imposed onto all the current notes and rests.

Similarly, many symbols are not present in the encoding but could be easily added. Dynamics cannot be encoded right now. They are ignored as if they were not present at all. They could be added as *after attachments*. The same applies to additional triplets or similar ornaments.

Grace notes are special. They look like little notes, they do not affect the rhythm and are considered an ornament attached to another note. PrIMuS agnostic encoding can represent them but at the expense of adding a lot of additional tokens. We decided not to bloat our vocabulary with symbols that are not very abundant in the CVC-MUSCIMA dataset. They are present in a few places in the evaluation dataset and are represented by the ? token.

A chord is two or more notes played simultaneously. Currently, there is no way of encoding simultaneous notes. Since chords usually share a stem, they could maybe be represented via after attachments. Maybe if we encoded the top-most note of a chord as a regular note and then added one "notehead" token for every remaining note, we could represent a chord. But there are problems with having multiple accidentals. Either way, it would be interesting to explore in some future work.

Text (like lyrics and tempo) is also ignored. It is not encoded by even the ? token.

5. Engraving System

This chapter describes the Mashcima engraving system we developed. We created a custom engraving system for handwritten music to serve as an advanced data augmentation tool. We opted to write a new system from scratch, because of the flexibility we needed. We will talk about the specific requirements for the engraving system. We will briefly overview the inner workings of the system - how it understands the input (Mashcima encoding), what are the basic building blocks from which a staff image is engraved, what are the most important events that happen during engraving. We will talk about the shortcomings of the system and how it could be extended in the future.

5.1 Why Custom Engraving System

In the thesis introduction (chapter 1) we stated that there is only a single dataset containing handwritten staves of music. There are other handwritten music datasets, but they either contain only symbols, or they are derived from CVC-MUSCIMA. Using this dataset as-is for training is not plausible, because it contains far too few symbol combinations.

We are not the first to realize this issue. The HMR baseline article (Baró et al. [2019]) talks about using data augmentation and transfer learning to solve the lack of training data. They propose a model to be trained on printed music, of which there is abundance. After that, the model is fine-tuned by training on the CVC-MUSCIMA dataset. The results they obtained are impressive, considering the method they used. To help with the process, they used data augmentation, like dilation, erosion, and blurring, and even measure shuffling.

We propose to use more sophisticated data augmentation. Specifically, we want to shuffle the data on the level of individual musical symbols. The reason we choose this approach is that we have access to the MUSCIMA++ dataset (Hajič jr. and Pecina [2017]). This dataset contains a lot of additional information regarding the CVC-MUSCIMA dataset, which includes segmentation masks of individual symbols and their relationships. We want to use these masks to engrave entirely new staves of music.

We will also build our own engraving system because engraving handwritten music is something that existing engraving systems do not focus on. Handwritten music is very different from printed music. Symbol positions vary, notehead shapes may differ from note to note within a single staff, slant angle may also change. We tried looking at Abjad¹ — a python package for engraving music that uses Lilypond² at its core. Lilypond can load custom fonts, but those have to be in vector form, not raster masks. Also, there is no way to introduce a controlled variability and randomness into the engraving process. When we considered the options, we decided to create our own system, since it would be faster and we would have the ability to modify it in the future.

¹<https://github.com/Abjad/abjad>

²<http://lilypond.org/>

5.2 Requirements

We are not trying to produce PDF files as the other engraving systems do. Our goal is to produce a single staff of handwritten music, as a raster image. This image should look as similar as possible to a cropped image of a staff from the CVC-MUSCIMA dataset. This means the image is already binarized, it has comparable resolution and the image has about three times the height of the staff it contains (the height of the staff lines).

This similarity can be accomplished by using symbols from MUSCIMA++. On page 19 of writer 1, there is an empty staff. We take the mask of the staff lines and produce an empty image of proper dimensions containing the empty staff lines. The staff lines are almost perfectly horizontal (they wobble up and down by a few pixels, but do not drift), therefore we will use them to create a lookup table for converting pitch numbers to pixel offset in the y axis. This will act as our blank canvas, from which we will start. *Canvas* is an actual term used in the codebase.

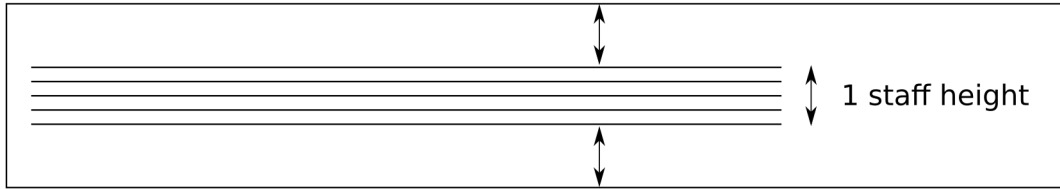


Figure 5.1: Empty staff and the padding to the edge of the engraved image.

By trying to mimic the look of CVC-MUSCIMA, we will remove a lot of problems that we would otherwise face. We do not need to perform any distortion removal, alignment, cropping, preprocessing, and binarization. All this is already performed on the source data we use. This lets us focus only on the engraving system, which is complicated enough by itself.

5.3 Token Groups and Canvas Items

The Mashcima encoding defines the concept of token groups. It is not discussed in the chapter on the encoding, because it is not so important for the encoding itself. A *token group* is a group of tokens, that has one main token and then all the attachments of that token. So for example a quarter note with a sharp and a staccato dot is represented by three tokens, but it is a single token group. A token group is all the tokens that belong to some non-attachment token.

Token groups are used in Mashcima annotation validation. When proper attachment ordering is checked, the annotation is first grouped into token groups and then each group has its attachment order checked. It also helps us to hide away all the attachment tokens and focus only on the important tokens.

Two special kinds of token groups are key signatures and numeric time signatures. It again makes sense to treat a time signature as a single object and since it is a multitude of tokens, it is represented by a token group.

Token groups are important because they map directly onto canvas items. A *canvas item* is something on the staff, that takes up horizontal space and can

be rendered. Canvas item represents a barline, a note, a rest. Attachments, like accidentals and dots, are not canvas items but are part of some given canvas item and they modify its appearance. Canvas items are placed on the staff right after each other with some randomized amount of padding between them. A canvas item has all the vertical space it can have and it decides, where to draw itself vertically. You can see the bounding boxes of individual canvas items in figure 5.2.

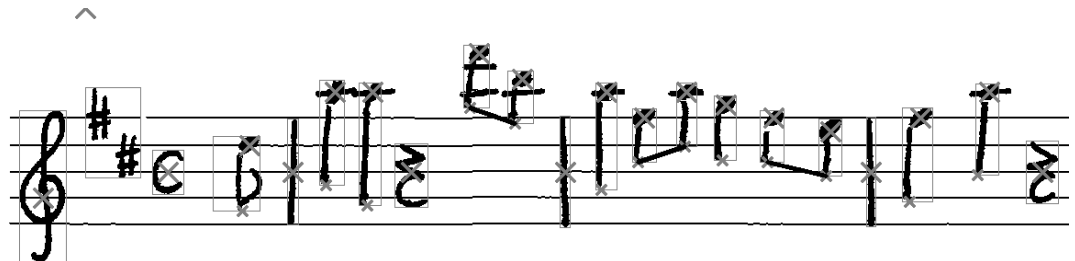


Figure 5.2: Canvas items with bounding boxes. A bounding box encloses all sprites of a canvas item. The large cross is the origin and it used to position the items vertically. Smaller crosses are marking where a stem ends and a beam attaches. A key signature does not use its origin point, so it is left at the top of the image.

5.4 Rendering Flow

The process of converting an annotation to an image has a couple of important steps:

Canvas instance is created An instance of the `mashcima.Canvas` class is created, so that canvas items could be added into it.

Canvas items are added The annotation is grouped into token groups and those groups are mapped onto canvas items. This step just feeds the semantic information from the encoding system to the engraving system. This step is performed by the `annotation_to_canvas` function inside `mashcima/annotation_to_image.py`.

The following steps happen inside the `Canvas.render(...)` method.

Canvas construction is finished This goes over the added canvas items and extracts information about slurs and beams. This step also validates that beams and slurs are properly formed. This extracted information is used later during the rendering of slurs and beams.

Sprites are selected Each canvas item gets to choose specific symbol sprites (images) to use for rendering. These sprites are chosen from a symbol sprite repository, represented by the class `mashcima.Mashcima` inside the file `mashcima/__init__.py`.

Sprites and canvas items are placed With specific sprites selected, dimensions are now known and so everything can be positioned properly. Canvas items are positioned from left to right one at a time. Each time a canvas item is placed, it also places its internal sprites (and attachments and other ornaments) and determines its total width.

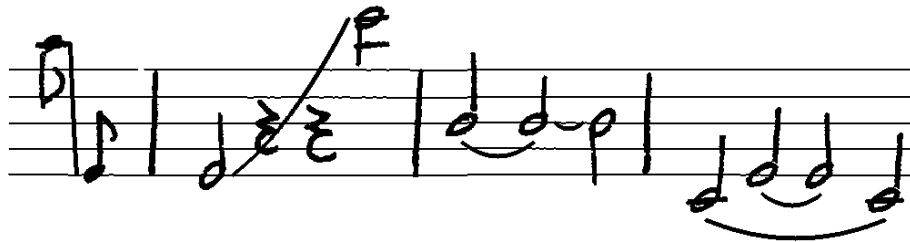
Beams are placed With note positions known, we can calculate the proper position and angle for each beam. Once the beam position is calculated, the length of stems of all corresponding notes is adjusted to match the beam.

Everything is rendered Canvas items, beams, and slurs are rendered. The order is not important, because the resulting image is binary.

5.5 Slurs and Beams

There are only a few symbols that are not taken from the MUSCIMA++ dataset as images but are instead rendered manually. Those are slurs and beams. MUSCIMA++ does contain binary masks of these symbols, but the problem is that they cannot be simply moved to a different position and rendered. They rely heavily on the position of other symbols and therefore they would need to be stretched, rotated, and skewed to render properly.

Slurs have some pre-defined attachment points they can use and the specific attachment point is chosen by the way a note is flipped. Once the two points are chosen, the slur is rendered as a parabola. This approach far too naive and simplified, so it does not capture the whole variability of a handwritten slur. We think this is the reason the model makes so many mistakes regarding slur placement.



e6 () e-4 | h-4 (qr qr) h8 | h0 () h0 () h0 |
h-6 (h-4 () h-4) h-6

Figure 5.3: Sometimes slur rendering does not look nice. Slurs cannot move other items around or create additional space. Slur does not move away when it intersects other symbols. Nested slurs are allowed, although not recommended.

Beams face many similar problems as slurs. They are also rendered manually, as straight lines. This again does not capture the real world. Beams are often curved or they have a gap between themselves and note stems. This again seems to be the source of many mistakes, especially with the writer 49, who leaves a lot of gaps between beams and stems.

5.6 Multi-staff images

Staves are usually so close together, that cropping a single staff with proper space around it will usually crop parts of symbols from the staves above and below. We want to capture this property of real-world data in our synthetic data.

We can take our rendering system and run it three times to render three staves into a single image. We crop out the middle staff we actually want. It can be used with some variations — having staff only above, or only below.

Multi-staff rendering can be performed by the `multi_staff_annotation_to_image` function inside `mashcima/annotation_to_image.py` file.



Figure 5.4: Rendering staves above and below with long barlines.

5.7 Additional Deformation and Normalization

Affine transformation is applied to the produced image to make the model resilient to changes in image positioning, perspective, size, and rotation. This transformation is a subtle, but very important step of the synthetic data preparation.

Finally, before the image is fed into the model, it is normalized to a specified height, while preserving the aspect ratio. The resulting image can look like this:



Figure 5.5: Skewed and normalized to 64 pixels in height. The blurriness is not an artifact of engraving this thesis document, it is actually present in the data. The transformation and normalization turn the image grayscale, so the model takes grayscale images as input, not binary.

5.8 Parameters

There are many places where the engraving system makes an arbitrary decision. This section attempts to list most of them. The list is not exhaustive. The goal is to give you an idea of how the system works.

- Sprites are selected randomly from a repository of all sprites. This selection has the most impact on the variability of the resulting image. The repository contains sprites from all writers so this selection process mixes up symbols from different writers.
- Noteheads and stems are not mixed — a given note sprite will always have its corresponding stem sprite when engraved.
- A canvas item has a pivot that is placed according to the item's pitch. This placement is not randomized (vertically) because sprite selection does a good job of it already.
- A note will have many sprites attached (accidentals, duration dots, staccato) and they are also chosen randomly from the sprite repository. Their position is slightly randomized. Sometimes additional rules are applied (e.g. to prevent duration dots from landing on staff lines and not being visible).
- Canvas items are placed from left to right with random padding inserted in between.
- Barlines can be short or tall. This depends on whether there are staves rendered above and below our staff.
- Note stem orientation depends on the note position. Lower-pitched notes point stems up and vice versa. Notes around the centerline have stem orientation randomized.
- A given note has three slur attachment points — before, below, and after the notehead. When notes on both ends of a slur have stems pointing in the same direction, the below points are used. Otherwise, the before and after points are used. When below attachment points are used, slur curves with the notes. Otherwise, the curvature is randomized.
- Slurs are rendered as parabolas intersecting three points — the attachment points and one midpoint. The midpoint vertical offset defines the curvature and it is proportional to the slur length.
- Beamed group stem orientation is determined in the same way as for a single note, only the average pitch of the group is used.
- The beam is selected as a line that minimizes the distance to all notes while staying on one side of all the notes. Minimal stem length is preserved.
- Stems in a beamed group have their height adjusted (by stretching the sprite) to end up right on the beamline.

5.9 Extensibility

Many symbols currently can be encoded, but cannot be engraved (trills, accents, fermatas). These attachment symbols could be added relatively easily, in a similar fashion to the way staccato dots and accidentals are rendered.

The slur and beam rendering system could be improved to better mimic the real world. The concept of attachment points for slurs is a little bit too digital. It could be made fuzzier. Also, there are certain slur placements, that the current system does not render (like slur above a beam). This kind of extension should not require too much redesign of the system.

It would be interesting to render tuplets. They are similar to beams and slurs in many ways. Also, dynamics and hairpins are maybe even easier to add. But they cannot currently be encoded.

Adding chords is an interesting problem, I think the current system architecture would make it quite difficult. The note canvas item would need to be entirely redesigned.

6. Experiments and Results

This chapter focuses on the experiments we performed. We will first describe the training and evaluation data. How it was chosen, where it comes from, and in the case of evaluation data, also the manual annotation process. Then we will talk about the symbol error rate (SER), a metric used for evaluation, as well as additional metrics we propose to understand mistakes our model makes. We will describe the training and evaluation process in detail and provide values for all hyperparameters. We will describe the setup for each experiment and explore the results from many perspectives. Finally, we will compare our results to the results in the HMR baseline article (Baró et al. [2019]).

6.1 Training Data

Before we can talk about experiments, we have to explain what the training data looks like. In chapter 3 we talked about the network architecture. The model takes an image as the input and produces a sequence of annotation tokens. Chapter 4 describes how these annotation tokens encode the music in an image. Now we just need to obtain enough pairs of images and annotations to train on.

The thesis introduction (chapter 1) stated that the only available dataset is CVC-MUSCIMA (Fornés et al. [2011]). This dataset contains 1000 images of handwritten sheets of music, consisting of 20 pages, each written by 50 writers. Because of this lack of variability, the dataset cannot be used as-is. In chapter 5 we described our Mashcima engraving system. This system can produce an image of one staff, that corresponds to a given Mashcima annotation. It does that by rendering musical symbols present in CVC-MUSCIMA, which in turn were extracted as part of the MUSCIMA++ dataset (Hajič jr. and Pecina [2017]).

We have a system, that can create images for given annotations. All we need to provide are those annotations.

6.1.1 PrIMuS Incipits

We ideally need thousands of annotations to account for all the variability in note types and pitches our encoding can capture. Luckily, the PrIMuS dataset (Calvo-Zaragoza and Rizo [2018]) contains exactly what we need. PrIMuS contains over 87 000 incipits of monophonic music. An incipit is the recognizable part of a melody or a song. The incipits have the ideal length of a few measures. It is not an entire staff, but not a few symbols either. Also, all the incipits are encoded in many formats, but most importantly they are encoded in the agnostic format, which is very similar to the Mashcima encoding.

We can take the PrIMuS dataset, engrave all the incipits using Mashcima, and train on the result. The only obstacle is converting PrIMuS agnostic encoding to Mashcima encoding.

Converting PrIMuS agnostic encoding to Mashcima encoding is mostly a one-to-one mapping of tokens. Pitches have to be encoded differently, tokens have different names. In PrIMuS, all tokens have pitch information, so for some tokens, it gets stripped away.

Some incipits, however, need to be filtered out. PrIMuS contains symbols, that are not present in CVC-MUSCIMA, therefore they cannot be engraved. These symbols are very long or very short notes (longa, breve, thirty-second). PrIMuS also contains many grace notes and similar symbols that the Mashcima engraving system cannot render, so they get removed. There are a couple of other rules and checks that make the conversion slightly more complicated. The exact code for the conversion can be found in the file `mashcima/primus_adapter.py`.

When the conversion finishes, we are left with 64 000 incipits we can use to train on.

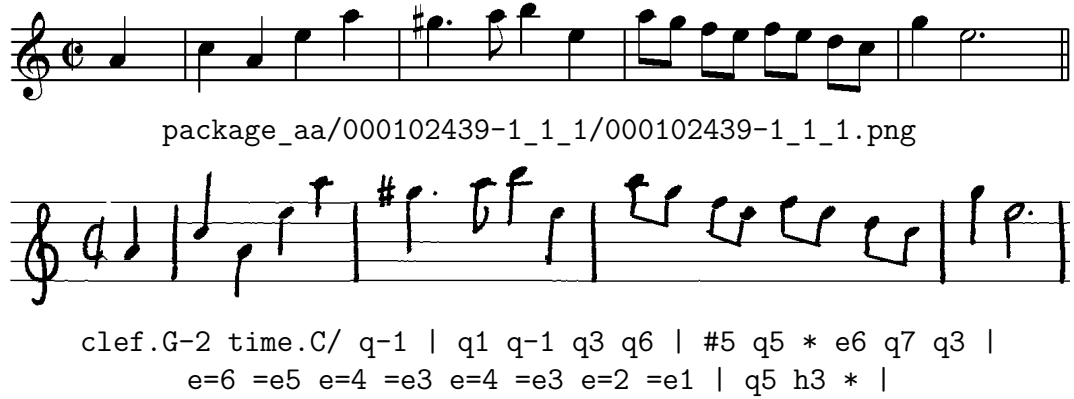


Figure 6.1: An image engraved from an annotation taken from the PrIMuS dataset.

The advantage of this training data is that the music in it comes from the real world. This allows the model to pick up common patterns and possibly learn the language model.

6.1.2 Synthetic Incipits

The other option we have is to just simply randomize the training annotations, to create some synthetic data. We throw away the possibility of learning a language model, but we get a different benefit. We can artificially boost frequencies of tokens that appear less frequently in the real world. This will cause the model to make fewer mistakes on uncommon symbols.

Randomization seems simple at first, but it can be done in many ways. At one extreme, we can simply randomly choose tokens from the vocabulary. This, however, produces sequences that cannot be rendered and are nonsensical. Beamed notes have to have a beginning and an end. We cannot have an unfinished or non-started beam. At another extreme, we can try to mimic the language model by using a lot of rules.

We opted for something in the middle. We make sure, that the synthetic annotation can be engraved, but we do not ensure anything more. Duration per measure is not correct, pitch is almost random, time signatures can be in the middle of a measure. The resulting image looks nothing like what we are used to seeing in sheet music. The code for generating synthetic annotations can be found in file `app/generate_random_annotation.py`.

We will compare these two approaches later in the experiments. It may come

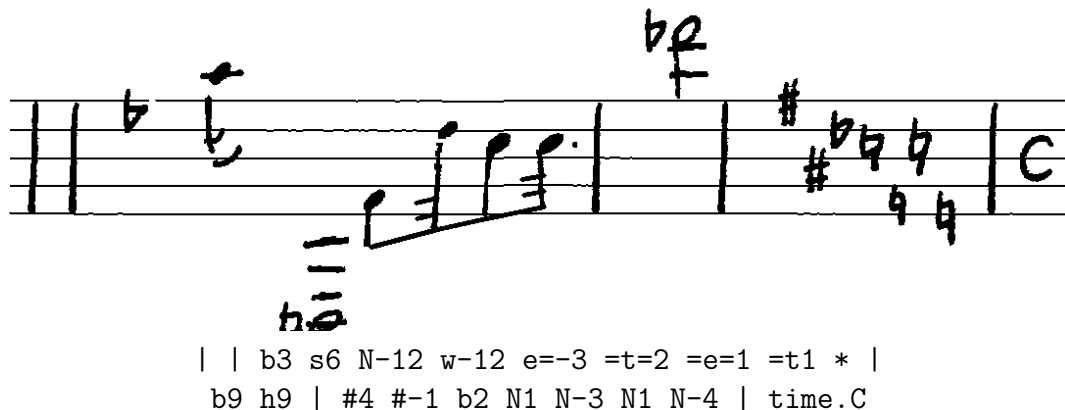


Figure 6.2: An image engraved from a synthetic annotation.

as a surprise, but the best approach will be to combine both synthetic and real-world data, effectively training on both.

6.2 Evaluation Data

When we faced the lack of training data, we resorted to data augmentation. We cannot do that for evaluation, because the evaluation data should be as close to the real-world as possible. Therefore using a well-established dataset is the only option.

By looking at the *Collection of datasets for OMR* by Alexander Pacha (Pacha) we can see that most existing datasets are for printed music. If we focus on the handwritten datasets, most of those contain only musical symbols, not entire staves. When we filter those out, what remains is CVC-MUSCIMA (Fornés et al. [2011]), MUSCIMA++ (Hajič jr. and Pecina [2017]), and Baró Single Stave Dataset (Baró et al. [2019]). We are already familiar with the first two datasets since we used them for training. The last dataset is also derived from CVC-MUSCIMA and it is used in the paper for HMR baseline (Baró et al. [2019]). We will compare our results to the results in the paper in section 6.7.

We decided to evaluate on a portion of the CVC-MUSCIMA dataset. Partly because it is the only dataset available for this purpose, partly because other people use it for evaluation as well. To learn more about the CVC-MUSCIMA dataset, see the section 2.1.

The fact that we devised custom encoding means we have to annotate the evaluation data manually. This is not very difficult, because the evaluation set need not be large. It also means the resulting annotations are of high quality and follow the rules of the Mashcima encoding.

We cannot use the entire CVC-MUSCIMA dataset for evaluation, because we already use it for training. Therefore we need to decide what portion is going to be used for evaluation. We definitely need to evaluate on data from different writers than those we train on. This is because seeing the specific writer’s handwriting style might help the model score higher during evaluation. Avoiding specific music pages is not necessary since the data augmentation process completely destroys any rhythmic or melodic information. The Mashcima engraving system samples

individual symbols, ignoring their placement relative to other symbols in the staff. So the primary concern is to separate writers used for evaluation.

There are additional criteria for selecting the evaluation writers. We want the writer selection to be diverse in terms of handwriting style. Some writers have very clean handwriting, some not so much. Noteheads can be little circles, ellipses, or even little dashes. Some writers have note stems slanted, some have straight, vertical stems. Also, the width and spacing of symbols differ.

We also want to evaluate on pages that are present in MUSCIMA++. This is because pages in MUSCIMA++ have a lot of additional information available and there exist detailed MusicXML transcriptions for them. Both of these facts may become useful in the future. Each writer has 20 music pages in CVC-MUSCIMA, but only 2 or 3 in MUSCIMA++. Additionally, not all pages can be represented in the Mashcima encoding (some are polyphonic or have multiple voices).

First, we sorted the 20 pages by how easily they can be encoded using Mashcima encoding (see table 6.1). This sorting is not perfect, the main goal is to separate pages that we cannot encode at all. Some symbols can be encoded, but since the engraving system cannot render them, they are considered slightly problematic. See the section on extending mashcima encoding (section 4.5).

Page	Acceptable	Notes
03	Yes	perfect
12	Yes	perfect
02	Yes	trills, grace notes
11	Yes	? token
09	Yes	? token, fermata
05	Yes	trills
01	Yes	triplets, fermata, rests in beamed groups
13	Yes	? token
14	Yes	chord, triplets
17	Yes	two staves with chords
15	Yes	rests in beamed groups
16	Yes	beamed notes with empty noteheads, accents
06	Not ideal	trills, many grace notes
04	Not ideal	tenuto, triplets, nested slurs, bar repeat, fermata
18	Not ideal	two staves with chords
07	No	trills, many concurrent notes
08	No	grace notes, unsupported symbols, two voices in bass
20	No	chords in many places
10	No	chords
19	No	multiple voices

Table 6.1: All pages of CVC-MUSCIMA, sorted by how easily they can be represented using the Mashcima encoding.

Then we took all the acceptable pages and found all writers for those pages that are present in MUSCIMA++. We sorted those writers by the number of pages that satisfied our selection (see table 6.2).

All the remaining writers had only two or fewer pages from the selection. We

# Pages	Writer	Handwriting style	Selected
4	49	worse, dash noteheads	Yes
3	06	nice, round noteheads	No
3	13	regular, round noteheads	Yes
3	20	regular, dash noteheads	Yes
3	27	nice, round noteheads	No
3	34	regular, round noteheads, slanted	Yes
3	41	beautiful, round noteheads	Yes

Table 6.2: Writers that were considered for the evaluation dataset.

took 5 writers out of those 7 writers manually, to keep the handwriting diversity high.

Lastly, we wanted to compare our results with the results of *From Optical Music Recognition to Handwritten Music Recognition: A baseline* (Baró et al. [2019]), so we added the writer 17. The final writer and page selection can be seen in the table 6.3.

Writer	Pages
13	02, 03, 16
17	01
20	02, 03, 16
34	02, 03, 16
41	02, 03, 16
49	03, 05, 09, 11

Table 6.3: The evaluation dataset.

We end up with 6 writers, 17 pages (7 distinct), 115 staves, and over 5840 tokens. Annotations for these pages can be found in the file `app/muscima_annotations.py`. These annotations have been created by me — the author of this thesis. Experience regarding the annotation process is described in the following section (section 6.2.1).

Lastly, we want to show a frequency table of the most common tokens and pitches in the evaluation dataset (table 6.4). The table contains generic variants of the tokens.

Pitch	Count
None	2703
4	531
3	411
2	373
5	344
1	243
0	203
6	191
-2	147
7	132
-4	103
-3	89
-6	88
-1	86
8	72
-5	48
9	33
-7	21
-8	20
10	13
-9	1

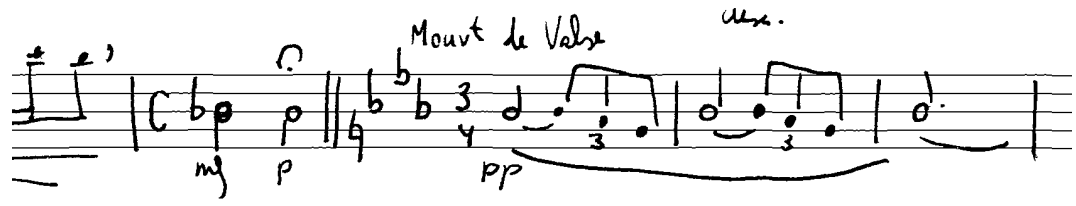
Token	Count
q	981
	843
)	418
(416
=e=	372
e=	357
=e	334
#	287
qr	244
.	236
*	159
e	150
=s	103
=s=	88
s=	80
w	72
h	70
er	67
N	64
wr	62
clef.G	55
b	47
time.4	40
>	40
time.3	39
hr	36
clef.C	34
clef.F	28
s	27
trill	21
lr	16
?	15
tuplet.3	13
**	12
br	8
fermata	4
time.8	3
sr	2
time.6	2
: :	2
:	2
time.5	1
time.7	1
time.C	1

Table 6.4: Frequency tables of pitches and tokens in the evaluation dataset.

6.2.1 Manual Annotation Experience

Although the Mashcima encoding attempts to not be ambiguous, there were some places where we had to make some decisions regarding undefined situations. This section goes over these situations.

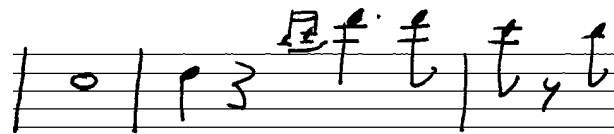
Page 1: The last three measures contain nested slurs. These cannot be represented, so we chose to represent slur beginnings and slur endings as they appear on the page. One note cannot have two slur beginnings, so only one is annotated. The very last slur is maybe not a slur, but some pitch articulation symbol. We annotated it as a slur continuing onto the next staff.



```
=s=8 =e7 | time.C b0 h0 fermata h0 | N-2 b1 b4 b0 time.3 time.4
h0 ( ) e=0 tuplet.3 =e=-1 =e-2 | h0 ( ) e=0 tuplet.3 =e=-1 =e-2
| ) h0 * ( ) |
```

Figure 6.3: Last three measures of page 01, writer 17.

Page 2: The last two staves contain three occurrences of grace notes. They look like regular notes but are smaller. Grace notes cannot be represented yet, so we replaced them with a ? token. We replaced the entire grace note group (two sixteenths with a slur) with a single ? token.

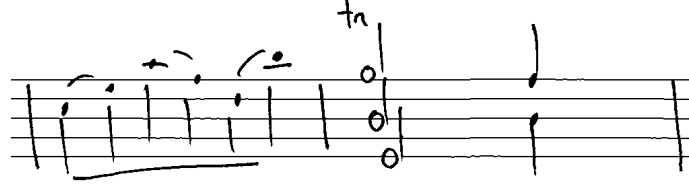


```
| w1 | q2 qr ? q9 * e9 | e8 er e7
```

Figure 6.4: Grace notes in page 02, writer 13.

Page 9: There are two measures with notes playing at the same time. The first three half notes are slightly offset, so we annotated them from left to right. The last two quarter notes are right above each other, so we replaced them with the ? token. We wanted to place at least one ? token inside the measure and then we tried to annotate the rest as good as we could. This way the measure is marked and can be repaired in the future.

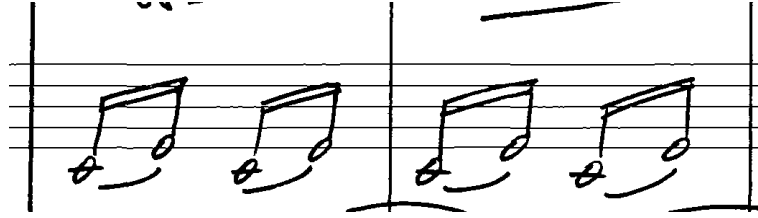
Page 11: One measure has the same problem as page 9.



| e=1 () =e=3 =e=6 () =e=4 =e=2 () =e7 | trill h7 h0 h-6 ? |

Figure 6.5: Simultaneous notes in page 09, writer 49.

Page 16: Third staff contains a bracket symbol in the key signature. The bracket symbol is completely ignored, but the clef and key signature are annotated as usual. The fifth staff contains double-beamed notes with empty noteheads. These are not sixteenth notes, but since they look so similar, we annotated them as such. These symbols are not very common and the trained model treated them as sixteenth notes as well, so we kept it that way.



| s=-6 () =s-4 s=-6 () =s-4 | s=-6 () =s-4 s=-6 () =s-4 |

Figure 6.6: Beamed notes with empty noteheads in page 16, writer 34.

Special thick barlines, double barlines, or barlines with braces at the beginning of a staff are all annotated as simple | token. The only exception is repeat signs that do have their corresponding tokens.

There are many trills or accents throughout the pages. Those are not in the training data but can be represented, so they are annotated just as defined in the chapter on Masheima encoding.

6.3 Evaluation Metrics

Now that we have a model producing some token sequences and we have our gold sequences, we need a way to measure the model performance. There are three goals for these measurements:

- Compare the model against itself to track improvements.
- Get an overall idea of the model performance and compare it to other works.
- Analyze model output to identify common mistakes it makes.

Looking at the work by Calvo-Zaragoza and Rizo (Calvo-Zaragoza and Rizo [2018]) or the HMR baseline article (Baró et al. [2019]) we can see, that the metric they use is *Symbol Error Rate* (SER). This metric is also known as normalized Levenshtein distance or edit distance. The name Symbol Error Rate is used in

contrast to Word Error Rate (WER) in the text recognition community. Since we do not work with text, we are left with the Symbol Error Rate only.

Regular Levenshtein distance (Levenshtein [1965]) is defined as the minimum number of single-character edits that turn our prediction into the gold sequence. We do not work with strings, so we use tokens instead of characters. The basic edit operations are insertion, deletion, and substitution. The lower this number, the better. Zero means a perfect match.

This metric has to be normalized by the length of the gold sequence to allow for averaging over multiple values. Normalized Levenshtein distance produces a number that is typically between 0 and 1, where 0 means the sequence was predicted perfectly and 1 means the sequence was entirely wrong. The normalized distance can be greater than 1 when the predicted sequence is much longer than the gold one, but that happens only when the model is completely useless.

$$SER = \text{LevNormalized} = \frac{\# \text{Insertions} + \# \text{Deletions} + \# \text{Substitutions}}{\text{GoldLength}}$$

Since this metric is also used by other works, we will use it for comparison against these works.

When training, we will use the edit distance function implemented in the Tensorflow¹ library. Although it is claimed to be the normalized Levenshtein distance, the implementation is different from the one used during evaluation. Therefore these two values should not be compared directly. The training edit distance is only meant for tracking the learning process and determining the stopping condition.

6.3.1 Understanding Model Mistakes

We would like to get an idea of the kind of mistakes our trained model makes. The chapter 4 talks about the Mashcima encoding and how it can represent symbols that it cannot yet represent (by the ? token). Having this ? token in the gold data creates an ever-present mistake, that increases our error rate. We would like to get an estimate of how much of the overall error is contributed by such symbols. Also, note that ? is not the only token the model cannot produce. Some symbols cannot be engraved yet, like trills, accents, or fermatas. Being able to measure the error these tokens contribute would give us an idea of how much the model could improve if we implemented these symbols in the engraving system.

During the evaluation, we will take the prediction and remove certain tokens from it. These same tokens will be removed from the gold sequence as well. We will compute the error of these simplified sequences. Comparing this error to the baseline error should tell us how much the removed tokens contribute to the baseline error.

The metric used for computing this error will be the Levenshtein distance but normalized by the number of *important tokens* in the gold sequence. An *important token* is a token that will never be removed. It can be altered, but not removed. This will make sure the normalization term stays constant over all the possible transformations and thus all the error values should be comparable.

¹<http://tensorflow.org/>

Important tokens are notes, rests, barlines, clefs, accidentals, and other similar tokens. What remains as non-important is slurs, ornaments, and the ? token. The specific list of important tokens can be found in the file `app/vocabulary.py`. We will call this metric *Important Token Error Rate* (ITER). Remember that this metric should not be used for comparison against other models using different encodings. It is purely to get an idea of what mistakes contribute to the Symbol Error Rate.

With this metric we propose a set of transformation functions that progressively simplify the sequences:

- **ITER_RAW**
No transformation is applied, corresponds to SER, but normalized by the number of important tokens.
- **ITER_TRAINED**
Tokens that the model has not seen during training are removed (? token, trills, fermatas, etc.).
- **ITER_SLURLESS**
Like the above, but slurs are removed as well ((,)).
- **ITER_ORNAMENTLESS**
Like the above, but most of the non-important attachments are removed (trill, accent, staccato, fermata, ...). What has to remain are accidentals and duration dots. Those are important for correct pitch and rhythm.
- **ITER_PITCHLESS**
Like the above, but all pitch information is removed by converting all tokens to their generic variant.

Each metric builds on the previous one, further simplifying the sequences. This means the error rate should decrease as we go down. The amount by which it decreases can tell us how much the given transformation affected the error, therefore how much the removed tokens contributed to the error rate.

Also please understand, that all these errors are computed on a single trained model. The gold sequence is modified during evaluation. Not during training. We are trying to understand a specific model we have.

6.4 Architecture, Training, and Evaluation

In chapter 3 we provided a short introduction to deep neural networks and described the RCNN architecture. We will use this architecture in the following experiments. The list of layers used in the neural network can be found in table 6.5.

The Mashcima engraving system works with images at the resolution of the CVC-MUSCIMA dataset. Image of an engraved staff is about 400 pixels in height and the width varies from 500 to over 2000 pixels. The neural network however requires the input image to be exactly 64 pixels in height. The image will be scaled down because of this while preserving its aspect ratio.

Layer	Shape	Note
Input	$w \times 64 \times 1$	
Convolution	$w \times 64 \times 16$	Kernel 5x5
Max pooling	$w/2 \times 32 \times 16$	Stride 2,2
Convolution	$w/2 \times 32 \times 32$	Kernel 5x5
Max pooling	$w/4 \times 16 \times 32$	Stride 2,2
Convolution	$w/4 \times 16 \times 64$	Kernel 5x5
Max pooling	$w/4 \times 8 \times 64$	Stride 1,2
Convolution	$w/4 \times 8 \times 128$	Kernel 3x3
Max pooling	$w/4 \times 4 \times 128$	Stride 1,2
Convolution	$w/4 \times 4 \times 128$	Kernel 3x3
Max pooling	$w/4 \times 2 \times 128$	Stride 1,2
Convolution	$w/4 \times 2 \times 256$	Kernel 3x3
Max pooling	$w/4 \times 1 \times 256$	Stride 1,2
Reshape	$w/4 \times 256$	
BLSTM	$w/4 \times 256 + w/4 \times 256$	Droupout
Concatenate	$w/4 \times 512$	
Fully connected	$w/4 \times \text{num_classes} + 1$	No activation function
CTC	$\leq w/4$	

Letter w stands for input image width. BLSTM means bidirectional recurrent network with LSTM cells. The dropout on the BLSTM layer is important because the model does not converge without it. At the end, $\text{num_classes} + 1$ means the number of output classes (vocabulary size) plus the blank symbol required for connectionist temporal classification (CTC). All activation functions are ReLU. Convolutional layers do not have an activation function, only pooling layers do. All the details can be seen in the source code.

Table 6.5: Layers of the neural network.

There will be two datasets used for training. One for the actual training — a *training dataset* and one for validation — a *validation dataset* (or *dev dataset*). The training dataset is fed into the model in batches and each batch is used to perform an update of the learned parameters of the model. This process is called stochastic gradient descent (Goodfellow et al. [2016]). Using the entire training dataset once is called *one epoch*. The validation dataset will be used after each epoch to estimate the true performance of the model (to estimate the generalization error).

Learned parameters will be updated by the adaptive learning rate optimizer (Adam) (Kingma and Ba [2014]), that comes with Tensorflow², with the default parameters (table 6.6).

We have not tried to fine-tune these parameters or any other hyperparameters. Our goal was to try training on engraved handwritten images and see whether this approach is even feasible. Tuning hyperparameters is one of the places where our approach can be improved in the future.

The training will run for a given number of epochs. In each epoch, an average

²<http://tensorflow.org/>

Parameter	Value
Learning rate	0.001
β_1	0.9
β_2	0.999
ε	10^{-8}

Table 6.6: Adam optimizer parameters.

symbol error rate on the validation dataset is recorded. The final trained model is the model, that had the lowest validation symbol error rate, during the whole training. If the number of epochs trained is sufficiently high, this method should return the model at the point, where the generalization error began to rise. Also, note that the symbol error rate here is the edit distance function from Tensorflow. It is a different implementation of SER than the one used for evaluation.

During the evaluation, the beam search decoding algorithm is used with a beamwidth of 100 (Hwang and Sung [2016]). There are two additional steps performed after that. Firstly the produced token sequence is repaired. This means the rules regarding beamed notes are checked and corrected and attachment tokens are sorted properly. This repairing process is relatively simple and completely rule-based. For the details see the `repair_annotation` function inside `app/vocabulary.py`. After the repairing process, leading and trailing barlines are stripped from both gold data and the prediction. This is because barlines at the beginning and at the end of staff convey no additional meaning. It is analogous to trimming whitespace characters around a sentence. Barlines with repeat signs are not stripped away since they are important.

6.5 Experiments

In the section on training data (section 6.1) we hypothesized some differences between training on PrIMuS incipits and synthetic data. The main idea is that training on PrIMuS incipits should allow the model to learn the language model. More generally training on real-world music samples should help the model, since it will be evaluated on real-world music in the CVC-MUSCIMA dataset. Training on synthetic data should allow the model to learn complicated combinations of symbols, that are not as common in real-world music.

To test this hypothesis we propose a set of four experiments (table 6.7).

Experiment	Training data	Validation data
1	63 000 PrIMuS	1 000 PrIMuS
2	63 000 synthetic	1 000 synthetic
3	31 500 PrIMuS, 31 500 synthetic	1 000 PrIMuS
4	63 000 PrIMuS, 63 000 synthetic	1 000 PrIMuS

Table 6.7: Training data (number of incipits) for the proposed experiments.

The first experiment trains a model on real-world incipits, second uses synthetic incipits and the third one combines both approaches in a 1:1 ratio. The

last two experiments validate on real-world incipits since the evaluation will also be performed on real-world music. The second experiment validates on synthetic incipits because we wanted to simulate a scenario where we do not have access to real-world incipits. The fourth experiment is the same as the third one, only utilizing the whole PrIMuS dataset as is available to us.

When training, a random permutation of the size of the entire dataset is generated and then used to access training items at random. This ensures random order of training items and also proper mixing of real-world and synthetic incipits. New permutation is generated for each epoch.

We trained each experiment for 20 epochs (except for the fourth that has been trained for only 10 epochs) and took the model with the lowest edit distance, averaged over the validation dataset. Figure 6.7 shows the training charts — how the training and validation SER (edit distance) evolves. The training took about 24 hours for each experiment on a gaming PC but it did not utilize the full capacity of the machine (probably because recurrent network training cannot be parallelized). The training was done in batches of 10 incipits. One epoch corresponds to 6300 batches. Experiment 4 has twice as large dataset and half as many epochs but the total number of batches is the same. Table 6.8 shows the optimal epochs — the point during training when the model was saved for the last time.

Experiment	Optimal epoch	Batches trained	Validation SER
1	9	56 700	0.0238
2	7	44 100	0.1409
3	6	37 800	0.0292
4	10	63 000	0.0278

Table 6.8: What epochs had the smallest validation error and were chosen as the final model.

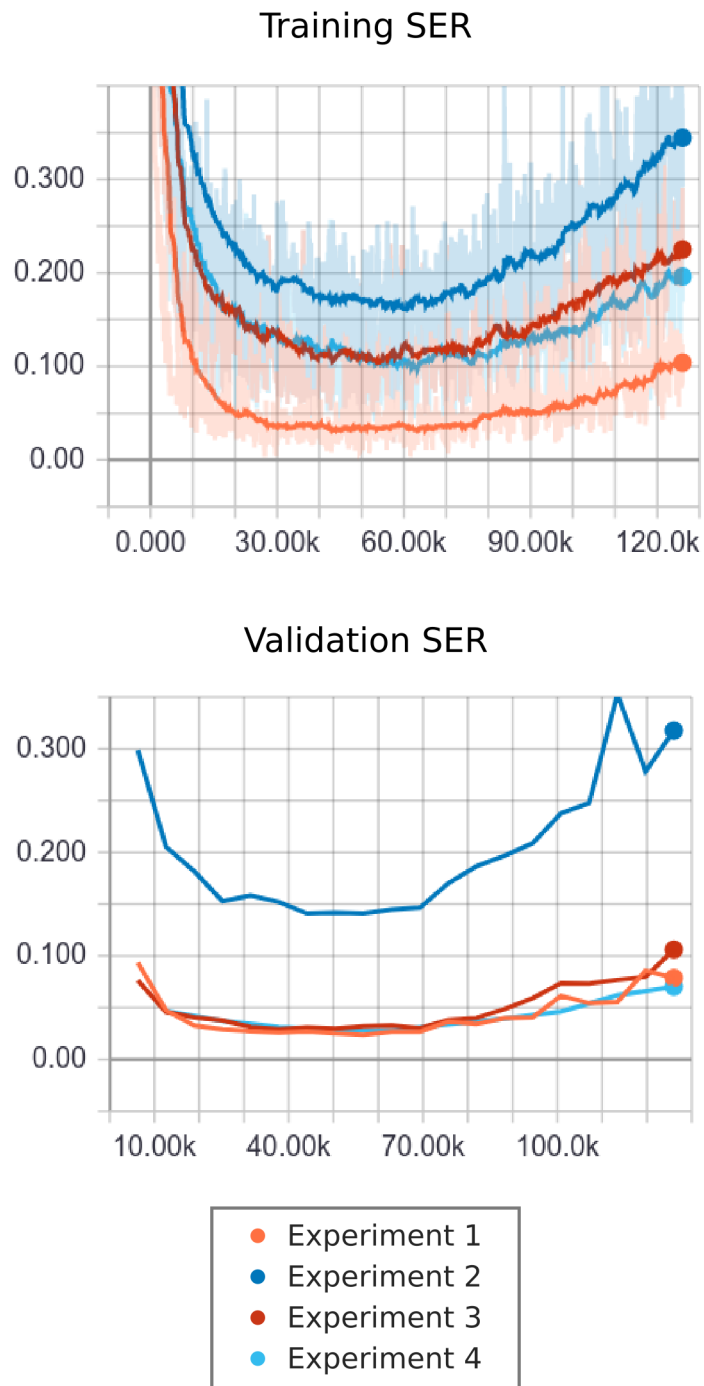


Figure 6.7: Training charts from Tensorboard. The horizontal axis shows the number of trained batches.

6.6 Results

The table 6.9 shows the resulting symbol error rates, averaged over the entire validation dataset.

Experiment	Symbol error rate
1	0.34
2	0.28
3	0.26
4	0.25

Table 6.9: Symbol error rates on the evaluation dataset for each experiment.

It seems that training on synthetic data is better than training on real-world data. But looking at experiment 3, we see that the best approach is to combine both approaches. Synthetic data is probably better than real-world data simply because all the tokens are represented equally. The discussion on the language model is more complicated and is explored in a separate section (section 6.6.1). The experiment 4 is slightly better than the experiment 3 because it has twice as much data to train on.

In section 6.3.1 we proposed a set of metrics, intended to give us insight into the mistakes the model makes. Table 6.10 shows these ITER metrics for each experiment.

Experiment	ITER Metrics				
	RAW	TRAINED	SLURLESS	ORNAMENTLESS	PITCHLESS
1	0.44	0.42	0.30	0.26	0.21
2	0.37	0.34	0.28	0.25	0.17
3	0.34	0.32	0.24	0.21	0.16
3	0.33	0.31	0.23	0.21	0.16
4	0.33	0.31	0.23	0.21	0.16

Table 6.10: ITER metrics (section 6.3.1) on the evaluation dataset for each experiment.

When we compare the `ITER_RAW`, `ITER_TRAINED`, and `ITER_SLURLES`, we can see that reducing our focus to only trained tokens helps slightly, although it is not as big of an impact as we expected. A considerably larger difference happens when we remove slur tokens. This confirms, what can be seen by looking manually at the predictions the model makes. There are a lot of mistakes related to slur classification. This might be caused by the fact that the engraving system does not capture all the variability that exists in the real world with regards to slur engraving.

In a previous section on evaluation (section 6.4) we mentioned, that the prediction, before being evaluated, is repaired by a few rules (attachment sorting, barline trimming). The table 6.11 shows how much impact the repair has on the error rate.

Experiment	Raw SER	Repaired prediction SER
1	0.34	0.34
2	0.28	0.28
3	0.26	0.26
4	0.25	0.25

Table 6.11: Comparison of repaired and non-repaired predictions.

You can see, that there is almost no difference. The repairs were indeed disabled, it is just that the performance difference was so minor, that the resulting average is the same.

Now that we know the experiment 4 performed the best, we will take a closer look at it. The table 6.12 shows metrics for each evaluation page (averaged over all staves in that page).

Page	Writer	SER	ITER Metrics				
			RAW	TRAINED	SLURLESS	ORNAMENTLESS	PITCHLESS
02	13	0.16	0.18	0.17	0.17	0.14	0.12
03	13	0.12	0.13	0.13	0.07	0.06	0.04
16	13	0.30	0.42	0.40	0.33	0.27	0.21
01	17	0.28	0.37	0.31	0.26	0.23	0.16
02	20	0.26	0.32	0.30	0.27	0.24	0.14
03	20	0.11	0.12	0.12	0.07	0.06	0.05
16	20	0.32	0.48	0.44	0.27	0.22	0.15
02	34	0.28	0.34	0.32	0.29	0.24	0.15
03	34	0.06	0.07	0.07	0.03	0.03	0.02
16	34	0.34	0.49	0.47	0.35	0.29	0.19
02	41	0.23	0.27	0.25	0.24	0.22	0.14
03	41	0.14	0.16	0.16	0.07	0.07	0.06
16	41	0.28	0.41	0.37	0.29	0.23	0.14
03	49	0.29	0.33	0.33	0.24	0.24	0.21
05	49	0.24	0.26	0.25	0.21	0.21	0.18
09	49	0.42	0.61	0.59	0.36	0.35	0.33
11	49	0.50	0.67	0.67	0.50	0.48	0.44

Table 6.12: All metrics for each page in the evaluation dataset.

We can average the results for each writer and compare them to the style of their handwriting (table 6.13).

Writer	SER	Handwriting style
13	0.19	regular, round noteheads
41	0.22	beautiful, round noteheads
34	0.23	regular, round noteheads, slanted
20	0.23	regular, dash noteheads
17	0.28	regular, round noteheads
49	0.36	worse, dash noteheads

Table 6.13: Symbol error rates averaged for each writer and handwriting style of each writer.

The first four writers are very much comparable, but the writer 49 has the worst handwriting of all the writers and he ended up last, as expected.

Similarly, we can average over each page and compare it to the annotation difficulty notes we took earlier (table 6.14).

Page	SER	Notes
03	0.14	perfect
02	0.23	trills, grace notes
05	0.24	trills
01	0.28	triplets, fermata, rests in beamed groups
16	0.31	beamed notes with empty noteheads, accents
09	0.42	? token, fermata
11	0.50	? token

Table 6.14: Symbol error rates averaged for each page and annotation notes for each page.

Pages 9 and 11 ended up last because they are only present for writer 49, who ended up as the worst writer. Page 3 is very interesting. It is the only page, that can be fully encoded using Mashcima encoding and all the symbols it contains can be engraved using the Mashcima engraving system. It is, however, also the simplest page in that it does not contain any complicated expressions and contains only a few slurs. This is supported by the fact that page 5 ended up with an also very low error and page 5 is very much comparable in its layout and complexity to page 3.

6.6.1 Language Model

When comparing the results of training on real-world data vs. synthetic data, it is strange that pure synthetic data outperform purely real-world data. But it probably has to do with the fact, that the synthetic dataset is balanced with respect to the individual output class abundance. The learned language model of real-world data helps the first experiment, but it is not nearly enough to beat the benefits of a balanced dataset for the second experiment.

This idea is supported by the fact that the third experiment beats both of the first two. It can benefit from both a balanced dataset and from learning a language model.

Training on real-world data does indeed make the system learn a language model, although it learns only very obvious rules. For example, a time signature is represented by two numbers above one another. The most common time signatures ($4/4$ and $2/2$) are represented by the symbol C and a crossed C . Signatures that remain as numeric are only $3/4$ and $6/8$. Any other signatures, although possible, are almost never used. This means that learning on real-world data should make the model gravitate towards those two common cases. It is explored in table 6.15 and indeed, the more synthetic the data is, the more mistakes the model makes on time signatures.

Experiment	Training data	Time signature mistakes
1	real-world	0/7 (0%)
2	synthetic	5/7 (71%)
3	mixed	1/7 (14%)
4	mixed	2/7 (29%)

Table 6.15: Erroneous time signatures for page 3, writer 49. Page 3 has been chosen because all staves have a time signature of $3/4$. The writer 49 was chosen because his handwriting is the worst and so the models would make many mistakes. Some of the mistakes experiment 2 made are $3/7$, $3/1$, and $3/9$.

6.7 Comparison to Other Works

We wanted to make a comparison against the HMR baseline article (Baró et al. [2019]) because our evaluation datasets overlap. Specifically, we share page 3 for writer 13 and page 1 for writer 17. We both use the symbol error rate metric, although many differences need to be addressed. Their model classifies rhythm and pitch separately, so both error rates are provided. There is also a combined error rate that treats the output symbols similar to our Mashcima encoding — having pitch and rhythm in one token (this number should be analogous to ours). The last column shows our error rate, given by the experiment 4.

You can see, that our model has a much smaller error rate, but we have to consider this result carefully. Their model does not use the CTC loss and the output encoding is very different. While our model might output a sequence of length 50, their model produces a sequence of the same length as the width of the input image, which is in the order of hundreds to a thousand sequence items.

Page	Writer	HMR baseline SERs			Our SER
		Rhythm	Pitch	Combined	
1	17	0.528	0.349	0.592	0.28
3	13	0.226	0.175	0.270	0.12

Table 6.16: Comparison of our model from experiment 4 to the model proposed by the HMR baseline article (Baró et al. [2019]).

Also, their encoding requires perfect alignment. If the model transitions between output classes at slightly different time-steps then the gold data, it produces a lot of error, even though when collapsed, the resulting sequence is the same. And given the temporal resolution, this might contribute a lot.

A more fair comparison would be a qualitative one. Luckily the paper provides a qualitative comparison of one staff from page 3 of their model against a commercial software called PhotoScore³. We can add a prediction by our model and compare all three (see figure 6.8). Note that the image has been produced by manually engraving the predicted Mashcima annotation.

You can see, that the difference is not as pronounced, although this staff is one of the simpler ones. There is, however, also a qualitative comparison on a staff from page 1 (see figure 6.9).

Both models make a lot of mistakes on this more difficult input so it is not easy to tell which one is better. What we can conclude is that their performance is actually very comparable.

It should also be noted, that each model uses a different resolution for input images. The model from HMR article normalizes to a height of 100 pixels, whereas ours normalizes to only 64 pixels. This might be a disadvantage for us. Also, our model cannot read chords by design, but theirs can. This might very well be required for some task and it would make our model unusable. Their model can also detect the presence of dynamics and text.

³<https://www.neuratron.com/photoscore.htm>

Input:



PhotoScore:



HMR baseline:



Our result:



```

clef.C-4 time.3 time.4 qr q3 q4 | q5 ( ) q4 q3 |
q4 e=5 =e=4 =e=3 =e2 | q3 e=4 ) =e3 e=2 ( ) =e3 |
q2 q5 * e4 | q4 e=3 ( ) =e2 q5 | q3 qr qr | qr q2 q3

```

Figure 6.8: Comparison of our results to the HMR baseline article on page 03, writer 13. The upper part (first three staves) are taken from the HMR baseline article (Baró et al. [2019]). The first staff is the input image from CVC-MUSCIMA, page 03, writer 13. The second staff is what produced the commercial software PhotoScore. The third staff is what the HMR baseline article achieved and the last staff is what our model returned, together with the Mashcima annotation. The image was created from the annotation by hand using MuseScore (<https://musescore.org/>).

Input:

PhotoScore:

HMR baseline:

Our result:

```

clef.F2 #2 time.3 time.4 e=-2 N-5 =s-5 e=-2 . =e1 s=0 . =e=1
=s=1 =e-3 | h-4 | e=5 . =t=2 =s2 . e=4 . =e8 . s=7 . =s=8 .
=e4 . | e6 er s5 ( ) q5 | e=5 =s=2 ) =e2 e=4 =e=2 =e4 e1 |
time.6 time.7 w8 * s=-6 ( =e=-7 =e-6 |

```

Figure 6.9: Comparison of our results to the HMR baseline article on page 01, writer 17. The upper part (first three staves) are taken from the HMR baseline article (Baró et al. [2019]). The first staff is the input image from CVC-MUSCIMA, page 01, writer 17. The second staff is what produced the commercial software PhotoScore. The third staff is what the HMR baseline article achieved and the last staff is what our model returned, together with the Mashcima annotation. The image was created from the annotation by hand using MuseScore (<https://musescore.org/>). The blue boxes are part of the original image and they show difficulties with recognizing symbols placed above each other. The red symbols in the last staff show errors introduced during manual engraving. MuseScore requires rhythm to be correct. For the actual output of our model refer to the Mashcima annotation below the staff.

6.8 Evaluating on Printed PrIMuS Incipits

We also wanted to try, how would our model perform on printed music. Models by other people are often pre-trained on printed music and then fine-tuned on handwritten images via transfer learning. Ours is different in that it has never seen an image of printed music. We already have code for parsing PrIMuS dataset and since the dataset contains images as well, we will use those. We just slightly preprocessed the images — inverted them, normalized, and slightly scaled down to have dimensions comparable to what our model trained on. We used the model from experiment 4 since it performed the best. The evaluation was performed on 100 incipits that the model has not seen during training and these are the results:

SER	ITER Metrics				
	RAW	TRAINED	SLURLESS	ORNAMENTLESS	PITCHLESS
0.61	0.64	0.64	0.60	0.59	0.56

Table 6.17: All metrics, evaluated on printed primus incipits. Our model was not trained on printed music so the performance suffers.

You can see, that the performance is not very impressive. We did expect the error rate to be high, but not that high. Although, it is understandable because the printed music is very different from the handwritten. It would be interesting to also train on printed images in the future. This error rate would go down, but maybe the CVC-MUSCIMA error rate would go down as well.

package_ab/211002723-1_1_1/211002723-1_1_1.agnostic



Gold: clef.C-4 #-1 #-4 #0 time.3 time.4 qr qr q1 |
q5 * e6 e=5 =e4 | h3 q3 | q2 . e3 e=4 =e5 | h3

Prediction: clef.C-4 #-1 #-4 #0 time.8 time.2 qr q1 |
q5 * s7 s=5 =e4 | h3 q3 | h2 (time.9 time.4 e=4 =e5 | h3

SER: 0.35

Figure 6.10: Evaluation on printed PrIMuS incipits.

Also, note that ITER_RAW and ITER_TRAINED have the same value. This is expected because we filter out incipits that cannot be engraved by Mashcima.

7. Conclusion and Future Work

Handwritten text recognition is an interesting and unique field of research. We can see it's mainly held back by the lack of training data. It shows the fact, that we were able to achieve state-of-the-art results by writing a simple engraving system. The MUSCIMA++ dataset is a step in the right direction. It provides excessive amounts of information about the music from which we can extract any specific encoding (be it with a little bit of work). But it has to be done this way because commonly used encodings (MusicXML, MEI, MIDI) are too abstract to be useful, whereas any lower-level encodings depend too much on the chosen model architecture. The MUSCIMA++ encodes music on a low level, from which high-level information can be extracted.

End-to-end solutions are powerful since they can learn intermediate features by themselves. The problem is that training an end-to-end model is much more challenging because it requires a lot more data. The MUSCIMA++ dataset has barely enough data to learn symbol classification and segmentation. When we want the model to extract pitch information about notes, the dataset becomes far too small. It is understandable because annotating data at such a high level of precision is very expensive.

Our Mashcima engraving system cannot yet engrave many symbols (chords, trills, repeats, dynamics, text) and this functionality could easily be added. We think that scaling the engraving system now would be a far too early optimization. The system relies heavily on the Mashcima encoding. This is ok for the task we tackled in this thesis — it was ideal for our model architecture. But should this engraving system be extended in the future, there are many more fundamental places it can change:

- What should the input look like? This will depend a lot on the final architecture because the input has to be capable of expressing everything the system can engrave. This would be an API at the source code level for which adapters from other formats (MusicXML) could be added.
- What format and resolution the output image has? Currently, we produced only binary images that are already very refined. We could generate RGB images that look like photos or scans. We could even leave raster graphics and produce a vector output.
- We could provide more information in addition to the output image. We could produce the annotation XML file that MUSCIMA++ uses. Suddenly our system would be useful for symbol segmentation as well as end-to-end learning.
- The way engraving is implemented could be different. Currently, the system moves around sprites. But when it tries to render slurs and beams, sprites become inconvenient. Maybe we could draw everything as curved lines, simulating the pen on the paper (we need it for slurs and beams anyway).

There is also a lot of discussions to be had about the model architecture. It seems that each architecture has pros and cons and there's not yet a fit-all

solution. Most of these differences have been mentioned many times throughout the thesis. We think that an engraving system capable of producing high-quality training data would make for a valuable common ground on which different HMR approaches could be directly compared.

Bibliography

- Arnau Baró, Pau Riba, Jorge Calvo-Zaragoza, and Alicia Fornés. From optical music recognition to handwritten music recognition: A baseline. *Pattern Recognition Letters*, 123, 2019. doi: 10.1016/j.patrec.2019.02.029.
- Jorge Calvo-Zaragoza and David Rizo. End-to-end neural optical music recognition of monophonic scores. *Applied Sciences*, 2018. doi: 10.3390/app8040606.
- Jorge Calvo-Zaragoza, Antonio Pertusa, and Jose Oncina. Staff-line detection and removal using a convolutional neural network. *Machine Vision and Applications*, 28, 2017. doi: 10.1007/s00138-017-0844-4.
- Jorge Calvo-Zaragoza, Jan Hajič jr., and Alexander Pacha. Understanding optical music recognition. *ACM Computing Surveys*, 2020. doi: 10.1145/3397499.
- Kyunghyun Cho, Bart van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using rnn encoder-decoder for statistical machine translation. In *Conference on Empirical Methods in Natural Language Processing (EMNLP)*, 2014.
- Alicia Fornés, Anjan Dutta, Albert Gordo, and Josep Lladós. CVC-MUSCIMA: A ground truth of handwritten music score images for writer identification and staff removal. *International Journal on Document Analysis and Recognition (IJDAR)*, 15, 2011. doi: 10.1007/s10032-011-0168-2.
- Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. <http://www.deeplearningbook.org>.
- Alex Graves, Santiago Fernández, Faustino Gomez, and Jürgen Schmidhuber. Connectionist temporal classification. In *Proceedings of the 23rd international conference on Machine learning (ICML)*, 2006. doi: 10.1145/1143844.1143891.
- Jan Hajič jr. and Pavel Pecina. In search of a dataset for handwritten optical music recognition: Introducing MUSCIMA++. In *14th International Conference on Document Analysis and Recognition (ICDAR)*, 2017. doi: 10.1109/ICDAR.2017.16.
- Sepp Hochreiter. Untersuchungen zu dynamischen neuronalen netzen. Master’s thesis, Technical University Munich, Institute of Computer Science, 1991.
- Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural Computation*, 9, 1997. doi: 10.1162/neco.1997.9.8.1735.
- Paul Hough. Method and means for recognizing complex patterns, 1962. URL <http://www.freepatentsonline.com/3069654.html>.
- Thomas Huang, G. Yang, and G. Tang. A fast two-dimensional median filtering algorithm. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, 27, 1979. doi: 10.1109/tassp.1979.1163188.

- Kyuyeon Hwang and Wonyong Sung. Character-level incremental speech recognition with recurrent neural networks. In *2016 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, 2016. doi: 10.1109/icassp.2016.7472696.
- Diederik Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *International Conference on Learning Representations*, 2014.
- Sangkuk Lee, Sung Joon Son, Jiyong Oh, and Nojun Kwak. Handwritten music symbol classification using deep convolutional neural networks. In *2016 International Conference on Information Science and Security (ICISS)*, 2016. doi: 10.1109/icissec.2016.7885856.
- Vladimir Levenshtein. Binary codes capable of correcting spurious insertions and deletions of ones. *Problems of Information Transmission*, 1965.
- Andrew L. Maas, Awni Y Hannun, and Andrew Y Ng. Rectifier nonlinearities improve neural network acoustic models. In *International Conference on Machine Learning (ICML)*, 2013.
- Jiří Matas, Ondřej Chum, Martin Urban, and Tomáš Pajdla. Robust wide-baseline stereo from maximally stable extremal regions. *Image and Vision Computing*, 22, 2004. doi: 10.1016/j.imavis.2004.02.006.
- Alexander Pacha. Collection of datasets used for optical music recognition. URL <https://apacha.github.io/OMR-Datasets/>.
- Alexander Pacha, Kwon-Young Choi, Bertrand Couasnon, Yann Ricquebourg, Richard Zanibbi, and Horst Eidenberger. Handwritten music object detection: Open issues and baseline results. In *2018 13th IAPR International Workshop on Document Analysis Systems (DAS)*, 2018. doi: 10.1109/das.2018.51.
- Joan Puigcerver. Are multidimensional recurrent layers really necessary for handwritten text recognition? In *2017 14th IAPR International Conference on Document Analysis and Recognition (ICDAR)*, 2017. doi: 10.1109/ICDAR.2017.20.
- Harald Scheidl. Handwritten text recognition in historical documents. Master’s thesis, Vienna University of Technology, 2018.