Neural Recognition of Handwritten Mathematical Expressions

Master Thesis
to obtain the academic degree of
Master of Science
in the Master’s Program
Bioinformatics
Abstract

The last decade has been characterized as a revolution of deep learning. Still many mathematical expression recognition engines are built on rule-based systems. In this thesis, we present a model that learns to recognize handwritten mathematical expressions from images using neural networks. We split the problem into two subtasks: symbol detection and symbol parsing. The symbol detection problem is solved by utilizing deep convolutional neural networks and the Faster R-CNN [44] architecture. The detected symbols and their locations are decoded into LaTeX by extending the Transformer [55] model to utilize bounding box information and an improved beam search algorithm. We evaluate the model and discuss its advantages and disadvantages. We conclude the study with discussions of further research.
Kurzfassung

Acknowledgements

I would like to thank my supervisor Dipl.-Ing. Thomas Unterthiner, MSc for his assistance in this study. He was very generous with his time and responded to my emails promptly.

Furthermore, I also want to thank my fiancéé, Klauda Barta, who supported and encouraged me and showed a strong example of industriousness.
## Contents

1 Introduction ........................................... 1  
1.1 Motivation ........................................... 1  
1.2 Problem Statement .................................... 2  
1.3 Project Aims ......................................... 3  
1.4 Literature Review .................................... 4  

2 Artificial Neural Networks ......................... 5  
2.1 Feedforward Neural Networks ..................... 5  
2.2 Convolutional Neural Networks .................... 8  
2.3 Faster R-CNN ......................................... 10  
2.4 Transformer ........................................... 12  

3 Methods .................................................. 15  
3.1 Datasets and Preprocessing ....................... 15  
3.2 Architecture ......................................... 21  
  3.2.1 Symbol Detection ................................ 21  
  3.2.2 Box Transformer ................................ 22  
3.3 Performance Metrics ............................... 26  
3.4 Libraries and Hardware ............................ 27  

4 Experiments and Results ........................... 29  
4.1 Symbol Detection Experiments .................. 29  
4.2 Symbol Parsing Experiments .................... 34  
4.3 Combined Accuracy ................................ 37  
4.4 Results on CROHME 2019 ......................... 37  

5 Discussion and Future Work ....................... 39  

6 Conclusion ............................................. 43  

A Average Precision per Symbol Class ............ 44
# List of Figures

2.1 Visualization of activation functions ........................................ 7  
2.2 The architecture of LeNet ....................................................... 9  
2.3 Faster R-CNN architecture ..................................................... 11  
2.4 Transformer architecture ....................................................... 13  
2.5 Scaled dot-product and multi-head attention .............................. 14  
3.1 Imbalanced dataset .................................................................. 18  
3.2 Training examples .................................................................... 19  
4.1 Metric evolution per step ......................................................... 30  
4.2 Examples of symbol detection .................................................. 31  
4.3 Training and validation losses ................................................ 32  
4.4 Accuracy and learning rate ...................................................... 35  
4.5 Metrics per sequence length .................................................... 36
## List of Tables

1.1 Spatial relationships in mathematical expressions .......................... 3
4.1 The 10 best and worst average precisions ................................. 33
4.2 Confused symbols ......................................................... 33
4.3 Box Transformer parameters ............................................. 34
4.4 CROHME 2019 results .................................................... 38
A.1 Symbols I ......................................................................... 44
A.2 Symbols II ....................................................................... 45
List of Equations

2.1 Layer of a feedforward neural network .......................... 6
2.2 Activation of the layer ........................................... 6
2.3 Softmax ............................................................. 6
2.4 Logistic sigmoid activation function ............................... 6
2.5 Hyperbolic tangent activation function ......................... 7
2.6 Rectified Linear Unit activation function ....................... 7
2.7 Categorical cross-entropy ......................................... 8
2.8 Gradient descent update rule .................................... 8
2.9 Positional encoding ............................................... 12
2.10 Output ............................................................... 14
2.11 Alpha weights ...................................................... 14
2.12 Scaled dot-product ................................................ 14
3.1 The difference matrix .............................................. 23
3.2 The transformed difference matrix ............................... 23
3.3 The modified output ............................................... 23
3.4 The modified scaled dot-product ................................ 23
3.5 Efficient scaled dot-product ..................................... 24
3.6 Beam search ........................................................ 24
3.7 Refined beam search .............................................. 25
3.8 Learning rate scheduling .......................................... 25
Chapter 1

Introduction

In this chapter, we introduce the problem of mathematical expression recognition. We discuss our motivation, state the aims of the project and review the relevant literature.

1.1 Motivation

Graphing calculators have been around for many years, with the first commercially available released in 1985. With the introduction of smartphones into the lives of many students, these calculators have been developed into mobile applications [5] as well. Research suggests that these calculators help increase the achievements of students in algebra [17, 27]. In spite of the developments and successes of these calculators, the user interface to input mathematical expressions remains keyboard based. Most of these phones also have controllable cameras, which opens up the possibility to implement image-based input to these calculators. Commercially available software for detecting mathematical expressions are already available on the market such as Mathpix [1] and Photomath [2] and while they achieve good performance on real-world use cases, the implementation details of the algorithms are trade secrets.

In the last decade, significant successes have also been made in the field of deep learning. Convolutional neural networks learn image features that are helpful in recognition, detection and segmentation tasks. Residual connections enabled researchers to train neural networks with 1000 layers successfully [25]. Recurrent neural networks, like the long short-term memory network [26], can classify, process and make predictions based on time-series data. Deep neural networks can beat professional Go players [11] and attention has been used to solve various natural language understanding tasks like translation and question answering [55].

These successes motivate the development of deep learning architectures that
solve the problem of recognizing handwritten mathematical expressions. In this thesis, we present a possible solution to this problem. A convolutional neural network is trained to detect symbols on the input image while an attention-based neural network provides translations of these symbols and bounding boxes into mathematical expressions.

1.2 Problem Statement

Mathematical expression recognition (MER) is the task of transforming printed or handwritten mathematical expressions as input into machine-readable representations. The input can be online or offline. When data about individual strokes, usually recorded on a tablet with a pen or finger, is available it is called online recognition. When the input is an image showing a printed or handwritten expression, it is called offline (for a list of examples, see figure 3.2).

The mathematical expressions can vary in their vocabulary size, structural complexity and length. The average expression contains digits, uppercase and lowercase letters from the Latin and Greek alphabet, operators like plus, minus, times, divide, fractions, sums and products.

Unlike in optical character recognition, where the symbols (characters or words) only have horizontal relationships, in MER symbols can also have vertical ones. They can be above or below to another symbol, they can constitute as superscripts or subscripts and some symbols, for example, the root, contain others inside them.

In this thesis, we consider offline recognition. For a full list of symbols please refer to table A.1 and A.2 in the Appendix. Symbol relationships are listed in table 1.1.

Symbol Relationships and Representations The relationships between the symbols can be represented in multiple ways. One of the most popular among publishers is called LaTeX, while MathML and label graph representations are also commonly used. LaTeX uses special characters and commands to represent the relationships and is also very compact. An example of LaTeX script can be seen in listing 1.1. MathML is a markup language using XML. This structure is more convenient for parsing by computers. Some browsers can render MathML. Label graphs [59] are directed graphs where the nodes are symbols and the edges are relationships. This representation allows for a simpler evaluation of programs that recognize mathematical expression.

Listing 1.1: The quadratic formula written in LaTeX.
\[ x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a} \]
CHAPTER 1. INTRODUCTION

<table>
<thead>
<tr>
<th>Relationship</th>
<th>Encoding</th>
<th>Image</th>
</tr>
</thead>
<tbody>
<tr>
<td>horizontal</td>
<td>2x</td>
<td>2x</td>
</tr>
<tr>
<td>superscript (power)</td>
<td>x^{2}</td>
<td>x^{2}</td>
</tr>
<tr>
<td>subscript (index)</td>
<td>x_{i}</td>
<td>x_{i}</td>
</tr>
<tr>
<td>vertical (fraction)</td>
<td>\frac{a}{b}</td>
<td>\frac{a}{b}</td>
</tr>
<tr>
<td>vertical</td>
<td>\sum_{i=1}^{n}</td>
<td>\sum_{i=1}^{n}</td>
</tr>
<tr>
<td>inside</td>
<td>\sqrt[3]{x}</td>
<td>\sqrt[3]{x}</td>
</tr>
</tbody>
</table>

Table 1.1: Examples of different spatial relationships between mathematical symbols. In the first column, we name the relationship, the second column lists the LaTeX encoding of that relationship and the third column prints the expression.

Problem analysis Mathematical expression recognition can be split into two tasks: symbol detection and structural analysis. In symbol detection, the task is to recognize and localize the symbols on the input image while the structural analysis part (which we also call symbol parsing) deals with transforming the detected symbols into a representation that encodes the symbols their spatial relationships.

1.3 Project Aims

Earlier studies focused on many possible solutions to the symbol parsing problem. Some used implicit rules [49, 52, 56, 58] some grammar rules [3, 10, 36] while others tried machine learning techniques [4, 8, 18].

This project aims to study the possibility of using deep learning to predict the structure of mathematical expressions. Using deep learning we remove the burden of explicit coding of the rules of the structure and allowing the program to learn them by itself. This opens up the possibility of learning new hidden features, that would have been missed by rule-based systems. We also look at deep learning methods for detecting symbols on image input.

While some work has already been done on mathematical symbol detection [30], implementing symbol parsing using only neural networks has not been explored by other researchers. In this project, we focus more on the symbol parsing, while also addressing the detection problem.
CHAPTER 1. INTRODUCTION

1.4 Literature Review

The study of mathematical expression recognition goes back as far as 1967 when Anderson [6] developed a top-down syntax based parsing algorithm. Another approach was developed by Belaid and Haton (1984) [7] which is based on constructing syntactic trees. In the year 2000, a review of existing expression recognition models shows that most models deal with the problem by splitting it up into two separate tasks: symbol recognition and structural analysis [9]. In some methods, structural analysis and symbol recognition are performed in parallel [13, 31, 32].

In offline expression recognition, Fateman et al. [20] propose a model based on parsing and Ha et al. [24] develop a method that combines segmentation-based and segmentation-free methods for achieving better results.

With the advancements in the field of neural networks, the encoder-decoder model has been proposed for statistical machine translation [12]. It consists of two submodules, the encoder and the decoder. The encoder module encodes the input into a single fixed-length vector, and the decoder module decodes this vector into the output vector with the use of recurrent neural networks (RNN). A similar model has been proposed by Sutskever et al. [48], with the difference of using multilayered Long-Short Term Memory (LSTM) [26] and reversing the order of the source sequences, which introduces short term dependencies between source and target sequences. Xu et al. [57] introduced an attention-based encoder-decoder for image caption generation. Deng et al. [16] extended this model by applying a recurrent neural network over the output of the encoder to capture the temporal layout of the input. This is used to decompile an image into HTML or LaTeX. Zang et al. [62] used very deep convolutions and a coverage-based attention model [53] to decode an image into LaTeX. In 2018 they improve this model by applying Densely Connected Convolutional Networks [28, 61] to strengthen feature extraction and facilitate gradient propagation. It also uses a secondary attention module for higher resolution.

Competition on Recognition of Online Handwritten Mathematical Expressions (CROHME) has been organized since 2011 [37–41]. Its main goal is to provide researchers in this field a common platform for evaluating and comparing their results with others. To enable this, they compiled a collection of datasets that contain online traces of handwritten mathematical expressions together with their respective LaTeX, MathML and label graph representation. The latest version (2019) contains over 12000 mathematical expressions by hundreds of writers from different cultural backgrounds. The challenge comprises of three tasks: online handwritten formula recognition, offline handwritten formula recognition and detection of formulas in document pages. The first two tasks have further subtasks: isolated symbol recognition and structure parsing.
Chapter 2

Artificial Neural Networks

Artificial neural networks are a model class that are inspired by animal brain structure. In a simple model of the brain, it consists of neurons that at some activation level pass information to other neurons. In neural networks, this translates to layers of neurons that do a non-linear transformation of their input and pass this information to the next layer.

2.1 Feedforward Neural Networks

A simple type of neural network is the so-called feedforward neural networks. The idea behind them is that each node in a layer has connections to nodes only in the following hidden layers. In other words, if we consider the neural network as a directed graph, where the nodes of the graph are the neurons, and the edges are the connecting weights, then the network is feedforward if its graph does not contain a cycle.

According to the universal approximation theorem, these feedforward neural networks, with a single layer, given enough hidden units, can represent any function [14]. This is indicative of their strong expressive power. The problem is that the number of hidden units can be very large, and the learning problem of finding the parameters becomes intractable.

Usually, these networks consist of one input layer, one or more hidden layers, and an output layer. Each node in the hidden layers takes its input as the weighted sum of the outputs of the previous layers and applies the activation function (or squashing function). For the $i$th node in the $k$th hidden layer, we calculate the input as $e_i^{(k)} = \sum_{j=1}^{n} w_{i,j} \cdot a_j^{(k-1)} + b_i$ where the activation of the previous node is $a_j^{(k-1)} = f^{(k-1)}(e_j^{(k-1)})$. The input vector is considered as the activation of the 0th
layer. We can convert these into a more compressed matrix notation as

$$
e^{(k)} = W^{(k)} \cdot a^{(k-1)} + b^{(k)} \quad (2.1)$$

and

$$a^{(k)} = f^{(k)}(e^{(k)}) \quad (2.2)$$

where $e^{(k)}$ is the net input vector to the $k$th layer, $a^{(k)}$ is the activation vector, $W^{(k)}$ is the weight matrix, $b^{(k)}$ is the bias vector, and $f^{(k)}$ is the non-linear activation function.

**Output** The last layer in a neural network is usually called the output layer. This layer can be described similarly to a hidden layer, with special kinds of activation function. These include the sigmoid for binary or softmax for multi-label classification.

$$\text{softmax}(x)_i = \frac{e^{x_i}}{\sum_{j=1}^n e^{x_j}} \quad (2.3)$$

**Activation Functions**

In an artificial neural network (ANN), every layer transforms its input via a non-linear activation function. This non-linear property is what allows ANNs to represent any complex function, so careful choice of the activation is important in every learning task. Activation functions must be differentiable, so their gradients can be used for learning.

**Logistic sigmoid** The logistic sigmoid activation function has an s-shape, squashes values between $(0, 1)$ and is defined as follows:

$$\text{sigmoid}(x) = \frac{1}{1 + e^{-x}} \quad (2.4)$$

This is generally used in the output layer of a neural network and can be interpreted as a probability score. Its advantage is that it can be easily computed, but it has several problems. One of the major drawbacks of this function is that if the activation of the neuron saturates (is either close to 0 or 1), its gradient becomes close to zero, making learning very slow.
CHAPTER 2. ARTIFICIAL NEURAL NETWORKS

Figure 2.1: Visualization of the activation functions: sigmoid (blue, top), ReLU (red, middle) and tanh (green, bottom).

**Hyperbolic tangent**  The hyperbolic tangent (tanh) activation function outputs a value between $[-1, 1]$, and is defined as follows:

$$\tanh(x) = \frac{e^{2x} - 1}{e^{2x} + 1}$$ \hspace{1cm} (2.5)

Because it’s centered around 0, optimization is easier using this activation, making it a better choice over the sigmoid function. This function also has a saturation problem in the extremes $(-1, 1)$.

**Rectified Linear Unit**  The Rectified Linear Unit (ReLU) [42] activation function is defined as follows:

$$\text{ReLU}(x) = \max(0, x)$$ \hspace{1cm} (2.6)

This function addresses the saturating gradient (also known as vanishing gradient) problem and converges faster than the tanh activation function [33]. The image of the function can be seen in figure 2.1. A problem of the ReLU function is that some units can “die” during training, that is they essentially become inactive. A solution to this problem is the Leaky ReLU, defined as $x$ if $x > 0$, otherwise $\alpha x$, where $\alpha$ is a small number, which allows for some small negative activation.
CHAPTER 2. ARTIFICIAL NEURAL NETWORKS

Optimization

Loss function  In the supervised learning setting, the loss function measures the performance of the model on a given input. Since we want to maximize the performance (or minimize the loss), this function has to be differentiable, so that the gradient descent algorithm can find the minima. In this setting, we denote $x_i$ as the $i$th input vector from the input set $X$ and $y_i$ as the corresponding label from the target set $Y$.

In regression, the mean squared error loss is the most popular loss function. The formula is defined as $1/n \cdot \sum_{i=1}^{n}(y_i - f(x_i))^2$. The name is self-explanatory, the loss function measures the mean squared error, where the error is the difference between the true output $y_i$ and the predicted output $f(x_i)$.

In classification, the most popular choice for loss functions is the cross-entropy loss in the case of single label classification and categorical cross-entropy in the case of multi-label classification. The formula for the categorical cross-entropy is

$$- \sum_{i=1}^{n} \sum_{c=1}^{m} y_i \cdot \log(f(x_i))$$  \hspace{1cm} (2.7)

where $m$ is the number of class labels. Setting $m$ to 1 we get the cross-entropy loss.

Gradient descent and backpropagation  Gradient descent is the first-order optimization algorithm that is used to find the minima of a differentiable function. Consider the function $f$, the initial value as $x_0$, the derivative $\Delta f$ and a learning rate $\lambda$. With some convergence criteria, the following update rule is applied to the weights:

$$w_i = w_{i-1} - \lambda \Delta f(w_{i-1})$$  \hspace{1cm} (2.8)

Backpropagation is the way how the gradient of a neural network is calculated. It is simply the application of the chain rule for the derivative of the loss. In the loss function (2.7) we have $f(x_i)$ which is a composite function of the activations of the layers. As we have seen in equation (2.8), we calculate the partial derivative of the loss function with respect to the weights of the neural network.

2.2 Convolutional Neural Networks

Convolutional neural network (CNN) is a special type of artificial neural network that has been successfully applied in image and video analyzing tasks. An early
example of this model is called LeNet (LeCun et al. 1998 [34]), of which the basic architecture is still used today. For the architecture see figure 2.2. The ImageNet Large Scale Visual Recognition Challenge (ILSVRC) [45], that has been organized since 2010, has been one of the main driving forces behind the development of complex CNNs. The first introduction of the CNN architecture to this challenge has been in 2012 by Krizhevsky et al. (2012) [33] called AlexNet. They used ReLU for the activation function and applied dropout during training to overcome overfitting. In 2013 Zeiler et al. (2013) [60] won the ILSVRC with ZFNet, and introduced the idea of deconvolutions. With this technique, the activations of the convolutions of the network could be visualized, and intuition gained through analyzing them. In 2014 VGGNet developed by Simonyan et al. (2015) [47] earned first place in the challenge with small 3 × 3 receptive fields and 1 pixel padding and max-pooling with 2 × 2 window size and a stride of 2. They showed that convolutions with small receptive fields can also be effective. The same year Szegedy et al. (2014) [51] developed the inception module in GoogleNet with a depth of 22.

These networks are constructed in such a way that instead of learning all pixel correlates, they learn to correlate pixel intensities in spatial neighbors of the image. Multilayered convolutions learn progressively abstract features of the input image, successively building on the previous layers of knowledge.

Convolutional neural networks are built of multiple types of layers: convolutions, activations, pooling and fully connected layers.

**Convolutions**  Convolutions are the basic building blocks of CNNs and do most of the computations. They scan through the input volume and, based on their training, they activate when the signal they receive is “interesting” to them. This means, for example, that some neurons can learn to activate when they “see” points, lines, circles, different color gradients, or, in deeper layers, they activate when scanning eyes, heads, car tires, etc. [60]. These convolutions are defined by
their receptive field and stride length. The receptive field is the width and the height of the kernel (a common size is $3 \times 3$). The stride length is the step size in both horizontal and vertical directions. Using these parameters, a convolution shrinks the size of the image, for example processing an image of $32 \times 32$ pixels with a kernel size of $3 \times 3$ and a stride of $1 \times 1$ would result in an image of size $29 \times 29$. This shrinking can be mitigated by padding the input image before the convolution.

**Activations** Activation layers are the same layers that are used in feedforward networks. ReLU is the most common choice for activation, as it addresses the vanishing gradient problem, although other functions can be used as well.

**Pooling layers** After some convolutions have been applied, CNNs use pooling layers to reduce the image size, essentially downsampling the image. Max pooling and average pooling are the most common pooling operations, with a $2 \times 2$ size and a $2 \times 2$ stride length, which will half the size of the image. Convolutional layers after pooling layers usually increase the depth of the input.

**Fully connected layers** At the end of convolutional networks there are usually one or more fully connected layers. These layers combine the information coming from the convolutions to produce meaningful results. For example, if the last layer in a convolutional network has 10 units and softmax activation, it can be trained to detect digits. Outputs from convolutional layers are 3-dimensional so they are flattened before passed into a fully connected layer. Interestingly most of the parameters of CNN are in the first fully connected layers. To address this issue GoogleNet used Average Pooling layer to reduce the size of the parameters significantly.

### 2.3 Faster R-CNN

Object detection is a hot research topic and has made a great success in recent years. Its difference to object classification is that instead of trying to classify an image into multiple categories, we are interested in detecting objects in the input image together with their bounding boxes. The number of objects on the image is not known a priori, which makes the detection harder.

Faster R-CNN is the latest development of the R-CNN series. Developed by Ren et al. [44], it builds on the earlier versions of R-CNN [23] which uses selective search for generating region proposals, and Fast R-CNN [22], which uses region of interest pooling layers.
In the earlier methods, object proposals were calculated by another algorithm (e.g. selective search [54] or DPM [21]). Faster R-CNN improves this, by combining region proposals with the convolutional neural network, thus sharing a great deal of computation. The model consists of roughly two parts, which are described below.

**Feature extractor** The so-called backbone of the Faster R-CNN object detection model is the feature extractor. This is usually a convolutional neural network, that runs on arbitrarily sized inputs, without the last dense classification layer. In the original paper [44], the authors experiment with the model introduced by Zeiler et Fergus (ZF) [60] and with another one developed by Simonyan et Zisserman (VGG) [47], but in theory this can be any feature extractor, that transforms an image with dimensions $H \times W \times C$ into $H' \times W' \times D$, where the image height ($H$) and width ($W$) are subsampled and the depth ($C$) is enriched.

**Region Proposal Network** The Region Proposal Network (RPN) takes an image as input and outputs a set of bounding boxes together with objectness scores. This score measures if the region proposed belongs to an object or the background. To increase the throughput of the network, the RPN shares the
CHAPTER 2. ARTIFICIAL NEURAL NETWORKS

computation with the feature extractor. To generate region proposals, it takes the output of this feature extractor and runs an \( n \times n \) convolutional network over it, downsampling the depth of the features to a fixed dimension. This is then fed to two separate fully-connected neural layers. One is the box regression layer (\( \text{reg} \)) and the other is the classification layer (\( \text{cls} \)).

The RPN scans through a grid of locations and multiple anchors are generated at each sliding location. For each anchor, the \( \text{reg} \) layer outputs 4 values, which relative to the anchor describes a bounding box, while the \( \text{cls} \) layer generates two values, which describe the probability of the bounding box containing an object.

2.4 Transformer

The Transformer, introduced by Vaswani et al. [55], is a deep learning architecture used in neural machine translation. It modifies the standard encoder-decoder model, by employing only attention mechanisms on the input and output tokens. This makes the model superior to RNN based models, by being more parallelizable and thus requiring less time to train.

Architecture  The architecture of the transformer (fig. 2.4) can be split into two parts, an encoder and a decoder. The encoder consists of 6 identical stacked layers. Each layer consists of two sublayers: one multi-head attention layer followed by a position-wise fully connected feedforward layer. Around each of these sublayers, a residual connection is applied, followed by a normalization layer. The decoder is similar to the encoder, with the difference of inserting a third layer which applies the multi-head attention to the output of the encoder.

Embedding  The authors use a learned embedding matrix to convert tokens of the input and output to vectors of \( d_{\text{model}} \) size. Since the model does not contain any recurrence or convolution, the position of each token needs to be explicitly encoded. The authors use the sine and cosine functions of different frequencies to encode the position of each token:

\[
PE_{(\text{pos},2i)} = \sin(\text{pos}/10000^{2i/d_{\text{model}}}) \\
PE_{(\text{pos},2i+1)} = \cos(\text{pos}/10000^{2i/d_{\text{model}}})
\]  

(2.9)

where \( \text{pos} \) is the position, \( i \) is the dimension and \( d_{\text{model}} \) is the dimension of the embedding. This encoding is added to the embedding vector.
Figure 2.4: The Transformer consists of an encoder and a decoder module. Both modules use self-attention and share the embedding matrices. To encode the position of the inputs, sinusoidal encodings are added to the embeddings. The decoder uses masked attention to prevent attending to future positions. Source [55]
Figure 2.5: On the left we see the scaled dot-product attention mechanism using query $Q$, key $K$ and value $V$ matrices. On the right, the multi-head attention is visualized. Using a linear projection, the query, key and value are split up, letting the mechanism attend to each separately. Source [55]

**Attention** The authors employ self-attention to capture dependencies between input tokens. They employ a so-called multi-head attention (figure 2.5), where the results of each head is concatenated. Each attention head calculates output $z = (z_1, \ldots, z_n)$ from input $x = (x_1 \ldots, x_n)$, by the following transformation [46]:

$$z_i = \sum_{j=1}^{n} \alpha_{ij}(x_j W^V)$$  \hspace{1cm} (2.10)

The $\alpha_{ij}$ weights are calculated as the softmax function:

$$\alpha_{ij} = \frac{\exp e_{ij}}{\sum_{k=1}^{n} \exp e_{ik}}$$  \hspace{1cm} (2.11)

And the $e_{ij}$ scores are calculated using the scaled-dot product compatibility function (figure 2.5), that compares the query $Q$ with the key $K$:

$$e_{ij} = \frac{(x_i W^Q)(x_j W^K)^T}{\sqrt{d_z}}$$  \hspace{1cm} (2.12)

$W^V$, $W^Q$ and $W^K$ are learned linear transformations, that transform the input into value, query and key.
Chapter 3

Methods

As we have seen in the introduction, the problem of recognizing mathematical expressions from images can be approached from multiple angles. In this thesis, we try to solve it by dividing the problem into two consecutive steps. The first step is recognizing the mathematical symbols from the image together with their bounding boxes and the second step is parsing these symbol, bounding box pairs into a LaTeX string.

3.1 Datasets and Preprocessing

Datasets

For training, validation and testing the datasets from the CROHME competition [37] have been used. These datasets provide a common ground for researchers to compare their results in online and offline handwritten mathematical expression recognition. The latest dataset, released in 2019, has for training in total 9,993 expressions and for testing 986 data points. We remove 1,000 expressions for validation from the training set, leaving us with 8,993 training images.

Data Format

The datasets consists of files formatted in Ink Markup Language (InkML) which are formatted as Extensible Markup Language (XML) and have a predefined structure. Each InkML file contains data about a single mathematical expression [37]. Along with the explanation of the structure of the InkML file, an example will be presented. The root tag is called <ink>, every other tag is the child of this root tag.

The format of the pen strokes is defined under the <traceFormat> tag. Most pen strokes are sequences of 2 dimensional points with \(X\) and \(Y\) coordinates, while some strokes have a third dimension which refers to the timing of the strokes.

\[
\begin{align*}
\text{\langle traceFormat} \\
\quad \text{\langle channel name="X" type="decimal"/} \\
\quad \text{\langle channel name="Y" type="decimal"/>} \\
\text{\rangle traceFormat}
\end{align*}
\]

In our experiments, we have ignored the third coordinate. The <annotation> tags contain information about the ground-truth LaTeX expression, writer relevant information (identifier, left or right handedness, age, gender, etc.) and a unique identifier of the ink. We use the truth annotation to get the LaTeX format of the mathematical expression.

\[
\begin{align*}
\text{\langle annotation type="age"} & 21\text{\rangle annotation} \\
\text{\langle annotation type="gender"} & M\text{\rangle annotation} \\
\text{\langle annotation type="hand"} & R\text{\rangle annotation} \\
\text{\langle annotation type="writer"} & 3\text{\rangle annotation} \\
\text{\langle annotation type="truth"} & \{I_k\}\text{\rangle annotation} \\
\text{\langle annotation type="UI"} & 2013\text{-IVC-CROHME-F3-E18\rangle annotation} \\
\text{\langle annotation type="copyright"} & \text{LUNAM/IRCCyN\rangle annotation}
\end{align*}
\]

In each InkML file, the MathML encoding of the mathematical expression is present, which we ignore. After the general information, recorded traces of pen movements are included in the file under the <trace> tags.

\[
\begin{align*}
\text{\langle trace id="1"} \\
551\ 188,\ 545\ 210,\ 545\ 215,\ 544\ 220,\ 544\ 225,\ 544\ 229,\ 544\ 233,\ 544\ 236,\ 544\ 239,\ 544\ 242,\ 545\ 245 \\
\text{\rangle trace} \\
\text{\langle trace id="2"} \\
520\ 190,\ 549\ 191,\ 555\ 190,\ 560\ 190,\ 565\ 190,\ 571\ 191,\ 577\ 191,\ 581\ 192,\ 586\ 192,\ 590\ 192,\ 593\ 192,\ 596\ 193,\ 599\ 193 \\
\text{\rangle trace}
\end{align*}
\]
These traces are then grouped under the `<traceGroup>` tag, which describes to which symbol do the traces belong to.

```xml
<traceGroup xml:id="7">
  <annotation type="truth">From ITF</annotation>
</traceGroup>
<traceGroup xml:id="8">
  <annotation type="truth">\{</annotation>
  <traceView traceDataRef="0"/>
  <annotationXML href="\{_1"/>
</traceGroup>
<traceGroup xml:id="9">
  <annotation type="truth">I</annotation>
  <traceView traceDataRef="3"/>
  <traceView traceDataRef="1"/>
  <traceView traceDataRef="2"/>
  <annotationXML href="I_1"/>
</traceGroup>
...  
</traceGroup>
```

**Class distribution** In terms of symbols, the training dataset is very imbalanced. In figure 3.1 we can see that the common symbols are the horizontal line, numbers 1, 2, 3, the plus and equals sign, brackets and variables \( x \) and \( a \). Underrepresented samples are the \( \exists \), \( \forall \), \( \in \), Greek letters, curly brackets and the greater than sign.

**Preprocessing**

The training data needs to be preprocessed to be ready to be used. It needs to be prepared for both object detection and symbol parsing. Descriptions of these processing steps are described in this subsection.

**Symbol Detection** For the symbol detection subtask, the data has been prepared to respect the CROHME 2019 rules. The script from the CROHME data package was used to render the mathematical expressions with white strokes on a black background and with dimensions of \( 1000 \times 1000 \) pixels together with a 5 pixel padding on all sides. Information about the symbols and their respective bounding boxes has been extracted from the InkML files. Examples of these images can be seen in figure 3.2.
Figure 3.1: The symbol classes that are over and under-represented in the dataset.
Figure 3.2: Examples of the images generated for symbol detection. There is great variation in the number of symbols per image e.g. in a) and b), in the styles of the symbols e.g. in c) and d) and in the scales of symbols, e.g. for the symbol x in e) and f), but also in the relative scales of the symbols in a) and b).
Symbol Parsing  Symbol parsing takes math symbols and their locations and transforms them into a structured representation. To accommodate this the data has to be preprocessed.

The math symbols and their locations are readily available in the InkML files, with little transformation. For this project, we chose the LaTeX representation, because of its simplicity and compactness. The tokens of the LaTeX expression can be embedded into a vector representation, and this fits nicely with the architecture of the proposed model, as we will see in the next chapter.

The ground-truth LaTeX expressions are available in the InkML files, but they contain a lot of noise and have to be cleaned up. To achieve this, the following rules have been applied:

- The expressions must not be in between $ signs.
- The tokens \left \right \Bigg \Bigg \text{ and whitespaces must be removed.}
- The tokens \lbrack \rbrack \parallel \to ' > < and \log must be replaced by ( ) || \rightarrow \prime \gt \lt and \log respectively

Some formulas could not be parsed due to invalid inputs. These were fixed manually by checking the rendered image of the InkML file.

Augmentation  Some of the image expressions are rotated. This can have a significant effect on symbol parsing, as the bounding boxes do not contain rotation information. To overcome this issue, instead of automatic straightening of expressions, we introduce rotations of the expression in the training set, to train the model to take into account rotations. We rotate expressions that are longer than 10 tokens, and we create 10 rotations each, with angles between -4 and 4 degrees.

Vocabulary  The vocabulary of the LaTeX expressions can be separated into two categories: the symbols that appear on the image and command tokens that are used for specifying the structure of the expression. The symbols are listed in table A.1 and A.2. The vocabulary of the command tokens is listed in table 1.1. The { and } are control tokens grouping expressions together and specifying the parameters of a command. The [ and ] symbols are used to specify the order of the root.
3.2 Architecture

In this section, we describe the general architecture that was used in the experiments. The section is divided into two subsections, each dealing with one of the subtasks.

3.2.1 Symbol Detection

Symbol detection has been implemented using the Faster R-CNN model. For the implementation, we used the framework described in [29]. This framework allows for experimentation with multiple backbones and feature extractors. One of the most promising convolutional neural networks is called Inception Resnet (v2) introduced by Szegedy et al [50], which combines Inception modules with residual connections. The Inception module achieves higher detection quality with a moderate increase in computational requirements [51], while residual connections allow for easier training of wider and deeper neural networks [25].

**Anchor generation**  Object detection models use anchor boxes to represent boxes that might contain objects. The prior generation of these boxes is very important in terms of accuracy, e.g. if these boxes are square shaped, then symbols that are tall and slim like $1$ and $I$, or short and wide like the fraction symbol - can go undetected. To this end, we used scales of $0.25$, $0.5$, $1.0$, $2.0$, $4.0$ and aspect ratios with the same values as scales, to generate the anchor grid.

**Post-processing**  To achieve higher accuracy, the results of the object detection model had to be further processed. On the output of the detector, we applied non-maximum suppression to reduce the number of redundant bounding boxes. After this step, some objects still appeared twice in the result because of the high intersection over union (IoU) threshold. To address this problem, we apply non-maximum suppression but now only on bounding boxes that belong to the same class and a low IoU threshold. Some symbols are comprised of multiple sub-symbols and they can also be falsely detected. Examples of these are $\tan$, $\sin$, $\cos$, and $\lim$. To address this problem we define pairs of symbols, where if the second symbol is found inside the first symbol, it must be excluded. We measure containment as a percentage, calculating the percentage of the area of the smaller symbol that is inside the rectangle of the larger symbol. Instead of using full containment we use a threshold of 98%. Examples of these pairs are $\tan$ and $t$, $\sin$ and $s$ and $\cos$ and $o$.

**Transfer learning**  Training large object detection models from scratch takes a long time. To mitigate this problem, we used transfer learning. We reused
the weights for the CNN feature extractor trained on the COCO dataset, which allowed for faster convergence.

**Optimization** To train this model, the momentum optimizer has been used, with a momentum value of 0.9. We used an initial value of 3e-3, which we decreased to 3e-4 at step 900000 and then to 3e-5 at step 1200000. The network is optimized using a softmax loss function for the class probabilities and a regression loss function for the bounding boxes.

### 3.2.2 Box Transformer

In the first part of the solution, we tried to detect symbols and their respective bounding boxes from an image. In this part, we assume that the results of the first part are perfect and we try to parse them into a LaTeX expression by using deep learning.

**Modified transformer** Natural language processing and symbol parsing are very similar tasks, as in the former we are trying to convert a sequence of words from a source language to a target language, while in the latter we are trying to translate mathematical expressions from symbols and positions to LaTeX tokens. The symbol parsing task matches the Transformer architecture well, because for both input and output the position of the source and target tokens are modeled explicitly by positional encodings. For our input, we can switch this encoding to the relative positional encoding of the bounding boxes.

**Bounding box encoding** The encoder needs to capture the information that is available in the bounding boxes. Based on work done by Shaw et al. on relative position representations [46], we inject relative bounding box information into each layer of the encoder. To achieve this, we constructed a matrix that encodes location information of each symbol relative to every other. The bounding boxes $b = (b_1, \ldots, b_n)$ form a matrix of size $n \times 4$, where $b_i = [x_{\text{min}}, y_{\text{min}}, x_{\text{max}}, y_{\text{max}}]$. From this, we create a tensor $\Delta$ of size $n \times n \times 8$, where each element is defined
as follows:

$$\Delta_{ij} = \begin{bmatrix}
    b_{i,x_{min}} - b_{j,x_{min}} \\
    b_{i,y_{min}} - b_{j,y_{min}} \\
    b_{i,x_{max}} - b_{j,x_{max}} \\
    b_{i,y_{max}} - b_{j,y_{max}} \\
    b_{i,x_{min}} - b_{j,x_{max}} \\
    b_{i,x_{max}} - b_{j,x_{min}} \\
    b_{i,y_{min}} - b_{j,y_{max}} \\
    b_{i,y_{max}} - b_{j,y_{min}}
\end{bmatrix}$$

(3.1)

This contains the differences between the four coordinates of all bounding boxes. From this tensor, the symbols that are close to each other have values close to zero, while elements further away have higher numbers. To increase the signal for elements closer together, we apply the following transformation:

$$\Lambda_{ij} = \begin{cases}
    -1 - \Delta_{ij}, & \text{if } \Delta_{ij} < 0 \\
    1 - \Delta_{ij}, & \text{otherwise}
\end{cases}$$

(3.2)

This $\Lambda$ tensor now contains values of 1 on the main diagonal, values close to 1 for symbols that are closer given a direction (right or down) and values close to -1 for symbols that are closer in the other direction (left or up). This way the matrix encodes relative bounding box information with directionality.

Following the modifications from Shaw et al. [46], we introduced this information into the output of each attention head via a dense layer and an addition. Equation (2.10) is modified as follows:

$$z_i = \sum_{j=1}^{n} \alpha_{ij} (x_j W^V + \Lambda_{ij}^V)$$

(3.3)

where $\Lambda^V = \Lambda W^V + b^V$ is a dense layer applied to $\Lambda$ to get the desired dimensionality.

Similar modifications are applied to (2.12) to consider distances when determining compatibility:

$$e_{ij} = \frac{(x_i W^Q)(x_j W^K + \Lambda_{ij}^K)^T}{\sqrt{d_z}}$$

(3.4)

where again another dense layer is applied to the tensor $\Lambda$ to embed the relative positional information.
Optimized equations The Transformer architecture, without relative positional encoding, allows for efficient implementation of attention using parallel matrix multiplication. This is not possible with the modifications above, because each relative position has a different representation. To overcome this problem, we applied the same modifications as in Shaw et al. [46], that is we split the equation (3.4) as follows:

\[
e_{ij} = \frac{x_i W^Q (x_j W^K)^T + x_i W^Q (A_{ij})^T}{\sqrt{d_z}}
\]  

(3.5)

The first part can be efficiently computed as before, while the second term can be computed using reshape and transpose operations, with \(n\) parallel multiplications of \(bh \times d_z \) and \(d_z \times n\) matrices. This same approach can be used to calculate (3.3).

Beam Search In the end we want to have a model such that we can translate an input sequence of symbols and bounding boxes \(x = (x_1, \ldots, x_n)\), into an output sequence of LaTeX tokens \(y = (y_1, \ldots, y_m)\) with the help of our model that can predict the probability of tokens, given an input and output sequence \(p(y_t | x, y_1, \ldots, y_{t-1})\). With this we can simply apply the following formula to estimate the most likely sequence of LaTeX tokens:

\[
\arg\max_y \prod_{t=1}^m p(y_t | x, y_1, \ldots, y_{t-1})
\]  

(3.6)

A greedy version of this algorithm would in each step take the most likely token. This can lead to errors, as the most likely token is not always the correct token because future tokens are not taken into account.

This algorithm can be improved by keeping multiple hypotheses during the search. After every iteration, for each output sequence, the probabilities of the extended sequences are calculated, they are ordered by their likelihood and the first \(n\) is kept. This algorithm is called Beam search and \(n\) is the beam size.

To keep the algorithm numerically stable, a log transformation is applied so that instead of maximizing the product of probabilities, the sum of log probabilities can be maximized.

A problem with the estimate in equation (3.6), is that it prefers shorter sequences. To counteract this effect the score is normalized by the length of the sequence to the power of \(\alpha\), which is a hyperparameter.
The refined beam search algorithm becomes the following:

$$\arg\max_y = \frac{1}{m^{\alpha}} \sum_{t=1}^{m} \log p(y_t|x, y_1, \ldots, y_{t-1})$$ (3.7)

It can be observed that there is a one-to-one match of input math symbols and output LaTeX tokens in the input-output pairs. This observation helps in further improving the results of the beam search by punishing translations that include or miss symbols that do not appear in the input sequence. The penalty is the number of extra or missing tokens in the output sequence scaled by a hyperparameter $\beta$. With this, we can control the effect of this punishment. In practice, it is good to keep this value relatively high. A caveat in this algorithm is to ignore counting of the control tokens in the output sequence, that do not appear in the input, i.e. { } _ ^ [ ] and \limits.

**Input pipeline** The bounding boxes of each input are normalized between 0.0 and 1.0. In theory, the order of the input data is not important to the Transformer, as the timing signal is calculated from the bounding boxes. Because of this, each input is sorted in increasing order based on the $\min_x$ of the bounding boxes. Then the whole dataset is sorted based on the lengths of the input sequences, and batch sized buckets are created from consecutive elements. This is done to avoid too much padding of the sequences and waste computational power. After this, the order of the buckets is shuffled and the elements in each bucket are shuffled as well. At the end of each sequence, the `<EOS>` token is appended, and the end bounding box of (1.0, 1.0, 1.0, 1.0).

**Optimization** To optimize the symbol parsing network, we used the Adam optimizer. The $\beta_1$, $\beta_2$ and $\epsilon$ parameters were fixed to 0.9, 0.997 and 1e-9 respectively. Similar to the original paper [55], we vary the learning rate over the course of training according to the formula:

$$\text{learning rate} = d_{\text{model}}^{-0.5} \times \min(\text{step}^{-0.5}, \text{step} \times \text{warmup}_\text{step}^{-1.5})$$ (3.8)

where $d_{\text{model}}$ is the hidden dimension of the model and step is the current training step. For warmup_steps the learning rate linearly increases after which it decreases proportionally to the inverse square root of the current training step number. For the loss function, we used the cross-entropy loss.

**Teacher forcing** When training on sequences, early errors in the predicted sequence have a huge impact on predicting the rest of the sequence. To address this
problem, we use teacher forcing, so that during training we always try to predict
the next token based on past ground-truth tokens.

Regularization  To mitigate the problem of a noisy dataset, we apply label
smoothing to the loss function with a value of 0.1. In theory, this makes the
algorithm generalize better. In practice, it reduces the confidence of the training
algorithm in the accuracy of the labels. We also apply L2 regularization with
a value of 0.0001, and dropout to the attention weights, to the encoder inputs
after adding the bounding box information, to the feedforward network, after the
residual connection and to the output of each sub-layer before it is normalized.

3.3 Performance Metrics

To evaluate the effectiveness of our models, we need performance metrics that tell
us how well do they work. We also use them to monitor the training process and
convergence. The performance metrics chosen here are in relation to the errors
that can occur in both types of problems. In this section, we describe the metrics
that we used and their justification.

Symbol Detection

Average precision  In the symbol detection part, we used average precision
(AP) as our performance measure. This metric is also used in the PASCAL Visual
Object Classes Challenge since 2007 [19]. It is defined as the area under the preci-
sion/recall curve. To calculate it, first we run the object detection algorithm on the
input images. This returns a list of tokens and their respective bounding boxes.
Next, we take a threshold value $T$ over the confidence scores of the predictions and
consider predictions over the threshold as the model’s prediction about symbols
on the image. For every symbol prediction, if there is a ground-truth bounding
box, with an area of intersection divided by the area of their union (IoU) larger
than 0.5, it is considered as true positive, otherwise as false positive. Precision is
the ratio between true positives and all of the predicted symbols, while recall is
the ratio between true positives and all the symbols in the test image. Varying the
value of the threshold $T$, one can calculate the precision and recall values and plot
them against each other. The area under this curve is called average precision.
In our case, having multiple symbol classes, we calculate the average precision for
every class, and we call the average of these values the mean average precision.
CHAPTER 3. METHODS

Average recall  We also report values of average recall (AR). AR is the maximum recall given a fixed number of detections over IoU thresholds, ranging from 50% to 100% with an increment of 5%.

Symbol Parsing
For evaluating the transformer architecture on symbol parsing, we used two different metrics. Both of them give a good measure of how well does the model work.

Accuracy  Accuracy is the percentage of sequences that were predicted without any error. We used this metric as our main metric, and we used it to select the best model on the validation set.

Word error rate  The word error rate (WER) performance metric is a common performance metric in machine translation systems. It is derived from the Levenshtein distance, which is used to measure the similarity of two sequences. In our case, we measure the number of insertions, deletions and substitutions required to change the predicted sequence into the ground-truth sequence. We also divide by the number of tokens in the ground-truth sequence, to get a normalized value across all sequence lengths. This measure is useful when we compare the results of sequences of similar length.

3.4 Libraries and Hardware
In this section, we list the tools and libraries that have been used throughout this project. The code was written in Python 3.5.

Tensorflow  Tensorflow is an open-source library used in machine learning. It is developed by Google and has a wide variety of tools that help users in building end-to-end machine learning solutions. It was used to build and train both the symbol detection and the symbol parsing models. It was also used to load and feed the data into these models.

NumPy  NumPy is another open source library, written in python, implementing efficient algorithms for scientific computing, and adding support for large multidimensional arrays. NumPy was used in our experiments to process the dataset before inputting it to the model.
CHAPTER 3. METHODS

**OpenCV** Open Source Computer Vision Library or OpenCV is an open source computer vision and machine learning software library. We used OpenCV to manipulate and display images for the symbol detection task.

**Hardware** We trained our models on a single NVIDIA Tesla K40c GPUs with 12 GB of memory.
Chapter 4

Experiments and Results

In this chapter, we describe the experiments that we ran to find the best performing model on the validation dataset. We also record the results of the models on the test set.

4.1 Symbol Detection Experiments

Our experiments on training the symbol detection model were based on the findings of earlier research \cite{30}. Here, the authors evaluated the Faster R-CNN architecture with different hyperparameters on an earlier CROHME (2016) dataset. They used the models available in Tensorflow’s official GitHub repository \cite{29}.

In their experiment, they have specified a maximum image dimension of 768 and a minimum dimension of 300 pixels. Furthermore, they have found that training on a pre-trained model allowed for much faster convergence. Lastly, they experimented with the number of maximum proposals and found that increasing from the default of 300 proposals did not lead to considerable improvements.

After they fixed these parameters, they tested the algorithm with four deep convolutional neural networks: Inception V2, Resnet 50, Resnet 101 and Inception Resnet v2. They have achieved the highest accuracy with the Inception Resnet v2 feature extractor.

In our experiment, we changed the maximum and minimum image dimension to 1000, which is the expected image size in the CROHME competition. We also reused the weights of a model trained on the MSCOCO dataset\cite{2}. We have

\footnote{The repository can be found at \url{https://github.com/tensorflow/models}, last accessed 29/05/2019.}

\footnote{The pretrained model can be found at \url{http://download.tensorflow.org/models/object_detection/faster_rcnn_inception_resnet_v2_atrous_coco_2018_01_28.tar.gz}, last accessed 03/06/2019.}
also used maximum 300 proposals per image. We left the rest of the parameters unchanged. For the backbone of the algorithm, we chose the Inception Resnet v2 model because of its high accuracy.

We have used minibatch training with batches of size 1 and we let the model learn for 300k steps. Overall the training took ~178 hours in wall clock time.

Results

The mean average precision over 50% IoU threshold is 81.4%. We have included the top 10 best and worst average precision per class in table 4.1. The full table can be found in the appendix, table A.1 and A.2.

In figure 4.1 we can see the mean average precision and average recall broken down by three class sizes. Symbols with an area less than $32^2$ are considered small, between $32^2$ and $96^2$ are considered medium and the rest is considered large. From these figures, we can see that the relative size of symbols greatly influences the precision of the detector: the larger the symbol the more accurate is the prediction.

Examples of the results of the detection model are visualized in 4.2. In 4.2a we can see a result without any mistakes. In 4.2b the s symbol is detected erroneously, as separate from the \cos symbol. In figure 4.2c we see the common error of mistaking the \times symbol with an x.
(a) Detection without any mistakes

(b) Separately detecting the $\cos$ symbol and its composite $s$ inside

(c) Detection with a common error of confusing $x$ with $\times$

Figure 4.2: Three examples of the results of symbol detection.
Figure 4.3: The training and validation losses. In subfigure (a) the training loss has been smoothed using the Savitzky-Golay filter with a window length of 13 and order of polynomial 3 for better visualization.

Loss Curves

In figure 4.3a we can see the training and validation losses over update steps. The fact that the training loss quickly drops under the validation loss hints at overfitting. Still, we let the model train more because in figure 4.1 we can see that the model is improving over time. Decomposing the validation loss into classification and localization loss can be seen in figure 4.3b. Even though classification loss is increasing, we are getting better precision and recall over time (figure 4.1).

Errors

Some symbols have a high precision because they look unique, while others are often confused resulting in a lower precision and recall of the model. Symbols that are confused with another symbol or missed in more than 20% of the cases are listed in table 4.2. Here we can see that the symbol detection model also has problems with small objects like the . symbol.
Table 4.1: The first column shows the 10 best performing symbols in terms of average precision while the second column shows the 10 worst ones. Some of these symbols are often confused with other symbols, which can be seen in table 4.2.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>AP</th>
<th>Symbol</th>
<th>AP</th>
</tr>
</thead>
<tbody>
<tr>
<td>\exists</td>
<td>1.000</td>
<td>o</td>
<td>0.095</td>
</tr>
<tr>
<td>\forall</td>
<td>1.000</td>
<td>S</td>
<td>0.218</td>
</tr>
<tr>
<td>F</td>
<td>1.000</td>
<td>G</td>
<td>0.293</td>
</tr>
<tr>
<td>\in</td>
<td>1.000</td>
<td>l</td>
<td>0.312</td>
</tr>
<tr>
<td>w</td>
<td>1.000</td>
<td>C</td>
<td>0.325</td>
</tr>
<tr>
<td>\mu</td>
<td>1.000</td>
<td>\prime</td>
<td>0.356</td>
</tr>
<tr>
<td>N</td>
<td>1.000</td>
<td>\lambda</td>
<td>0.373</td>
</tr>
<tr>
<td>\rightarrow</td>
<td>1.000</td>
<td>!</td>
<td>0.384</td>
</tr>
<tr>
<td>\sum</td>
<td>0.998</td>
<td>X</td>
<td>0.432</td>
</tr>
<tr>
<td>\int</td>
<td>0.998</td>
<td>P</td>
<td>0.452</td>
</tr>
</tbody>
</table>

Table 4.2: Symbols confused by the object detection algorithm in more than 20% of the cases. Some of these symbols are very similar to one another (e.g. X and x or S and s), some contain other symbols (e.g. \pm and +) and some others are just too small to be detected (e.g. the . symbol and the dots in the \div symbol).
4.2 Symbol Parsing Experiments

In this section, we present the experiments we conducted to find a good parametrization of the symbol parsing model and its results.

Parameter Search

The parameter search was carried out on the Box Transformer architecture. For each parameter, we defined a list of possible values that we considered as a candidate. We then grouped the parameters and carried out grid search on each of them and then fixed them. The list of these parameters can be seen in Table 4.3. We highlighted the parameter values that were chosen as a result.

In the end, we also tested two versions of the Box Transformer: the first one, similar to the original Transformer paper, had different weights in each of the encoder and decoder layers, while the second type, following more recent findings [15], shared the weights in all of the encoding steps, and all of the decoding steps. Compared to the first version, the second one had an increased expression recognition rate of 2%. It is interesting to see, that this constraint can increase the recognition rate, even though the first one had more parameters. We would also like to point out, that the training loss was also lower in the second version, hinting that this is not a case of overfitting.

<table>
<thead>
<tr>
<th>Group name</th>
<th>Parameter name</th>
<th>Candidate values</th>
</tr>
</thead>
<tbody>
<tr>
<td>Model</td>
<td>hidden layers</td>
<td>4, 6, 8, 10</td>
</tr>
<tr>
<td></td>
<td>hidden size</td>
<td>256, 512, 1024</td>
</tr>
<tr>
<td></td>
<td>filter size</td>
<td>256, 512, 1024</td>
</tr>
<tr>
<td></td>
<td>number of heads</td>
<td>8, 16, 32</td>
</tr>
<tr>
<td>Optimizer</td>
<td>learning rate</td>
<td>0.02, 0.2, 2.0</td>
</tr>
<tr>
<td></td>
<td>warm up steps</td>
<td>8000, 10000, 12000, 14000, 16000</td>
</tr>
<tr>
<td>Regularization</td>
<td>dropout</td>
<td>0.0, 0.1, 0.2</td>
</tr>
<tr>
<td></td>
<td>l2 regularization</td>
<td>0.0, 0.0001, 0.001</td>
</tr>
<tr>
<td>Beam search</td>
<td>beam size</td>
<td>3, 20, 40, 100</td>
</tr>
<tr>
<td></td>
<td>alpha</td>
<td>0.5, 0.6, 0.7</td>
</tr>
<tr>
<td></td>
<td>beta</td>
<td>0.0, 0.2, 2.0, 20</td>
</tr>
<tr>
<td>Recurrence type</td>
<td>share weights</td>
<td>false, true</td>
</tr>
</tbody>
</table>

Table 4.3: The parameters of the Box Transformer architecture. Each group was optimized separately.
(a) Training and validation accuracy  
(b) Learning rate scheduling

Figure 4.4: Subfigure (a) shows the evolution of the accuracy on the training and validation set of the best performing model. The training accuracy values have been smoothed using the Savitzky-Golay filter with a window length of 13 and order of polynomial 3 for better visualization. Subfigure (b) shows the scheduling of the learning rate.

Results

For the final results, we trained the Box Transformer algorithm for 500 epochs. Overall this took ~103 hours in wall clock time. In figure 4.4a we plot the accuracy values of the best performing model on the training and the validation dataset. We can see that the model quickly converges and further training does not help that much. Figure 4.4b shows the evolution of the learning rate over the course of training.

The accuracy of the final model is 84.5%, and the word error rate is 2.5%. We have evaluated the model on different sequence lengths. The results are plotted in figure 4.5. Here we can see that the longer the sequence, the more likely it is that it has a lower accuracy.

Errors

Superscript, subscript  A common error that the model makes is missing subscripts and superscripts or introducing one, where there is none. It does this more often on images, where there is a slight rotation of the formula.
Figure 4.5: In the figures above, we can see the evaluation metrics for the symbol parsing task broken down by sequence length. From subfigure (a), we can see that the longer the sequence the less accurate the prediction becomes. This is expected because if in every step of the inference process, there is some probability of introducing an error, then accuracy will decrease with the number of steps necessary to generate the output. On the right, in subfigure (b), we can see the word error rate (WER) per sequence length. The error bars represent the mean absolute deviation calculated for both sides. The WER of the model stays approximately the same across these bin sizes. This means that the length of the sequence does not necessarily influence the errors the model produces. The error bars become larger the shorter the expression is, that is because the WER metric is normalized with the length of the sequence. The bin sizes used in this figure are the following: very short sequences are between length 1 and 3, short between 3 and 7, medium between 7 and 11, long between 11 and 15 and very long between 15 and 95. All of these bins contain approximately 197 expressions.
**Root degree**  The model is unable to learn accurately when to put a symbol as the degree of the root and most of the times it just puts the symbol inside the root. The reason for this is that the stem of the root sign is not always the same size and at the same place, causing the model to be unsure about the degree of the root sign, erring on least likely case based on the distribution.

**Limits**  Another common error that the model makes is missing the \texttt{limits} symbol. This token makes the symbols that come after it as power and subscript to be moved to the top and bottom of the target symbol. The reason for this error is in the dataset: some training examples omit the limits symbol, and so the signal for its presence is not clear.

**Repetitions**  Sequences that have the same symbols repeated over multiple times are also missed in some cases. Interestingly, the error usually consists of moving one of the symbols in the sequence a few locations to the right. For example, $111111+11111$ could be mistranslated by $11111+111111$. In this case, we hypothesized that the number of encoding and decoding steps play an important role in missing such translations. Dynamic step size, like the one in the Universal Transformer, could be helpful in such cases [15].

### 4.3 Combined Accuracy

After performing parameter search on both models and training them, we evaluate the combined model. The accuracy of the system is 43.91%. We can see that most of the errors in the system come from the first part in symbol detection. Symbol parsing does not do any error correction, and it introduces more mistakes further pushing down the accuracy of the system.

### 4.4 Results on CROHME 2019

We also participated in the CROHME 2019 competition, in the task of offline recognition of formulas and structures. This gave us a way of comparing our system to other competing systems. The results of this competition can be seen in table 4.4 [3]. Currently no further information is available about the participating systems.

---

<table>
<thead>
<tr>
<th>Rank</th>
<th>Recognition Rate (%)</th>
<th>Participant</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>76.90</td>
<td>USTC-iFLYTEK</td>
</tr>
<tr>
<td>2.</td>
<td>70.81</td>
<td>PAL</td>
</tr>
<tr>
<td>3.</td>
<td>65.05</td>
<td>chungkwong</td>
</tr>
<tr>
<td>4.</td>
<td>62.47</td>
<td>wangkingkingking</td>
</tr>
<tr>
<td>5.</td>
<td>61.88</td>
<td>hanchao</td>
</tr>
<tr>
<td><strong>6.</strong></td>
<td><strong>47.62</strong></td>
<td>balazs</td>
</tr>
<tr>
<td>7.</td>
<td>24.02</td>
<td>ungquanghuy</td>
</tr>
</tbody>
</table>

Table 4.4: The results for the CROHME 2019 competition. The result of our method is printed in bold.
Chapter 5

Discussion and Future Work

The results of this project show that deep learning can handle complex tasks such as recognizing the structure of mathematical expressions. Still, many opportunities exist to improve the models. Here we discuss some options on how can we further improve the models’ accuracy, and also some possibilities for future development.

Discussion on Symbol Detection

The results from the previous chapter show that there is room for improvement for the symbol detection model. Small symbols, like dots and commas, have a low accuracy, while similar symbols, like $\theta$ and $o$, are often confused. Another issue with the presented solution is that it is using a large CNN as a backbone, and the relatively slow Faster R-CNN method. Further research should look into the accuracy of faster object detection models, like that of Single Shot Detector (SSD) [35] or the YOLO model [43].

Diverging losses It is interesting to see, that in figure 4.3b the validation losses diverge, with localization losses going down over time while classification losses increasing. This points to the fact that longer training makes the model better at localization of the symbols, but worse at correctly identifying them. This could mean that the Faster R-CNN model might not be the best architecture to this problem, as this combines the convolutional neural networks and it does detection and localization at the same time. It is possible that separate networks would not have this counteracting effect.

Data collection Efforts on data collection could also improve symbol detection. In figure 3.1, we have seen that many of the symbols are underrepresented and because of this the probability score of these symbols are not trustworthy (e.g. some of these symbols have 100% accuracy on the test set). Data generation is
also possible in this case since, for every single symbol in the dataset, we have the online information. Using this, the symbols can be augmented and printed on a white background. New formulas can be generated, or the symbols can also be printed in a quasi-random order (given enough space between the individual symbols, such that they remain readable).

**Training effort** As we have noted in the previous chapter, training an instance of the Faster R-CNN model with the Inception ResNet feature extractor takes more than 7 days. This makes a thorough hyperparameter search infeasible in a reasonable time frame without using multiple GPUs. This is a major drawback in our approach as it left us with the unexploited possibility of the large model that we have used.

**Discussion on Symbol Parsing**

We have presented a novel usage of the Transformer model originally designed for machine translation. Still, this approach has some inherent drawbacks. One of them is the degrading accuracy with longer sequences, which can be seen in figure 4.5a. The problem is that in every prediction step, once an error has been introduced, the model has no inherent way to correct for this error. We try to overcome this problem with beam search, but that is still not the perfect solution.

Another issue that arises with the current architecture, is that bounding boxes do not contain all the necessary information needed to generate the mathematical expressions. This can be seen in the model’s inability to correctly learn the base of the root symbol. This is because the location of the stem greatly varies between symbols. Another issue is that due to variance in writing styles, some symbols can have a prolonged tail that makes the bounding boxes unreasonably large. Another important information, the baselines of the characters, are also missing from bounding box information.

It is also a drawback, that there is no explicit coding for the one-to-one relationships between the input symbols and the output tokens in the architecture, except for the beam search. Since it is a black box model, it is also not possible to enforce custom rules, only through more data and learning effort.

**Training effort** Training the model took a considerable amount of time, but the model’s accuracy converges very quickly, giving us early feedback if it is a good candidate.

**Token combining** Some symbols in the training sequences are comprised of multiple symbols. The examples are \(\tan\), \(\sin\), \(\cos\), and \(\lim\). We have observed
CHAPTER 5. DISCUSSION AND FUTURE WORK

that the object detection model can make errors in detecting these characters by also detecting the symbols that comprise the large symbol. For example, detecting $\tan$ can result in the detected symbols $t$, $a$ and $n$ or $\cos$ can return with the full symbol and $s$ at the end (see figure 4.2b). New training entries in the symbol parsing model could be introduced, where such symbols are broken down into constituent symbols.

**Error correction**   Many symbols are very similar to each other and are often confused. For example $\times$ and $x$, $C$ and $(c$ or $5$ and $s$. One could augment the training data set with such input errors, and train the algorithm to correct them. We hypothesize that the algorithm can learn such error correction, as it can utilize contextual information as well. This could improve the overall accuracy.

**Dataset issue**   As already mentioned, the model was unable to correctly learn to generate the $\limits$ token. Upon inspection, we found that the dataset is ambiguous in many instances in using this token. This can be because the data set has been annotated by many different people, and has not been standardized by a set of rules. This could be improved upon, by refining the data set with careful examination.

**Soft classes**   In the symbol parsing phase, it was assumed that the input symbols are correctly identified in the detection phase. This assumption can be eased by allowing for mistranslations, by introducing soft classes in the detection pipeline. This means creating a third class for the symbols that are often confused and using that as the symbol class. Detections with a low score can be transformed into a soft class and parsing could consider both symbols as possible outputs. For example, detections like $5$ and $s$ could be changed to $s$-like, if their scores are lower than a threshold, and then symbol parsing can make use of contextual information to guess the most likely identity of the soft class. Such new entries could be introduced into the training dataset by reusing existing expressions.

**End-to-end training**

One of the drawbacks of our current approach is that we try to solve the problem in two steps and that once an error has been made in the first, symbol detection step, it cannot be fixed in the second, symbol parsing step. This could be overcome to some degree by the introduction of ”soft classes” as mentioned earlier.

However, errors from the symbol detection model could be fully addressed by training the symbol detection and parsing models together. Because of time constraints, we did not test, whether it is even possible to train such a chimera, but
it is possible that such training could result in higher recognition accuracy. We hypothesize that in this case, the errors of the symbol detection could be automatically accounted for without explicit post-processing and data augmentation techniques.

Another possible solution to this, is skipping the intermediate symbol detection output, and passing the features that the convolutional neural network produces to any recurrent neural network, for example, a Long-Short Term Memory network \cite{26} or the decoder module of the Transformer architecture. Some researchers have already tried this, with promising results \cite{61, 62}.

**Productification**

As stated in the introduction, one of the motivation of conducting this research was allowing image input to graphing calculators on mobile phones. To this end, the models presented in this project have to be deployed to mobile phones as well. Furthermore, the development and deployment of these models must be streamlined and automated further simplifying the process. And finally, a user interface must be developed to make the mathematical expression recognition engine accessible to everyday users while also taking care of the needs of people with disabilities. These issues can be addressed in future works.
Chapter 6

Conclusion

In this study, we introduced a novel method for transforming symbols with bounding boxes into LaTeX expression based on the Transformer architecture. We show that it is possible to train it with data coming from handwritten mathematical expressions. We also connect this novel method with an object detection model based on the Faster R-CNN architecture to create an end-to-end pipeline for mathematical expression recognition from offline data.

The symbol detection system has a mean average precision of 81.4%. Errors occur in confusing symbols with one another and in missing small ones.

Symbol parsing was implemented based on a modified version of the Transformer architecture. Bounding box information was encoded and added to the input embeddings. The standard beam search algorithm was extended to account for the fact that all input symbols must appear in the output. The accuracy of the resulting model is 84.5%.

We analyze the systems in terms of its performance and discuss possible further developments. Symbol parsing can be improved by allowing soft-classes that help in resolving errors coming from the detection. Symbol detection could improve in detecting small symbols and could utilize contextual information to reduce the number of false positives in the predictions.

We hope that this study will aid future research looking at deep learning as a possible solution to mathematical expression recognition.
## Appendix A

### Average Precision per Symbol Class

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Image</th>
<th>AP</th>
<th>Symbol</th>
<th>Image</th>
<th>AP</th>
<th>Symbol</th>
<th>Image</th>
<th>AP</th>
</tr>
</thead>
<tbody>
<tr>
<td>\exists</td>
<td>∃</td>
<td>1.000</td>
<td>A</td>
<td>A</td>
<td>0.987</td>
<td>\lt</td>
<td>&lt;</td>
<td>0.953</td>
</tr>
<tr>
<td>\forall</td>
<td>∀</td>
<td>1.000</td>
<td>)</td>
<td>)</td>
<td>0.985</td>
<td>x</td>
<td>x</td>
<td>0.949</td>
</tr>
<tr>
<td>F</td>
<td>F</td>
<td>1.000</td>
<td>=</td>
<td>=</td>
<td>0.984</td>
<td>7</td>
<td>7</td>
<td>0.948</td>
</tr>
<tr>
<td>\in</td>
<td>∈</td>
<td>1.000</td>
<td>3</td>
<td>3</td>
<td>0.984</td>
<td>e</td>
<td>e</td>
<td>0.948</td>
</tr>
<tr>
<td>w</td>
<td>w</td>
<td>1.000</td>
<td>(</td>
<td>(</td>
<td>0.982</td>
<td>b</td>
<td>b</td>
<td>0.944</td>
</tr>
<tr>
<td>\mu</td>
<td>μ</td>
<td>1.000</td>
<td>2</td>
<td>2</td>
<td>0.978</td>
<td>\theta</td>
<td>θ</td>
<td>0.944</td>
</tr>
<tr>
<td>N</td>
<td>N</td>
<td>1.000</td>
<td>d</td>
<td>d</td>
<td>0.978</td>
<td>\sin</td>
<td>sin</td>
<td>0.944</td>
</tr>
<tr>
<td>\rightarrow</td>
<td>→</td>
<td>1.000</td>
<td>B</td>
<td>B</td>
<td>0.976</td>
<td>\pi</td>
<td>π</td>
<td>0.941</td>
</tr>
<tr>
<td>\sum</td>
<td>∑</td>
<td>0.998</td>
<td>\infty</td>
<td>∞</td>
<td>0.974</td>
<td>6</td>
<td>6</td>
<td>0.931</td>
</tr>
<tr>
<td>\int</td>
<td>∫</td>
<td>0.998</td>
<td>\Delta</td>
<td>Δ</td>
<td>0.967</td>
<td>i</td>
<td>i</td>
<td>0.928</td>
</tr>
<tr>
<td>\leq</td>
<td>≤</td>
<td>0.998</td>
<td>y</td>
<td>y</td>
<td>0.965</td>
<td>5</td>
<td>5</td>
<td>0.927</td>
</tr>
<tr>
<td>\sqrt</td>
<td>√</td>
<td>0.995</td>
<td>]</td>
<td>]</td>
<td>0.963</td>
<td>-</td>
<td>-</td>
<td>0.926</td>
</tr>
<tr>
<td>\phi</td>
<td>φ</td>
<td>0.994</td>
<td>a</td>
<td>a</td>
<td>0.962</td>
<td>0</td>
<td>0</td>
<td>0.925</td>
</tr>
<tr>
<td>+</td>
<td>+</td>
<td>0.990</td>
<td>\gt</td>
<td>&gt;</td>
<td>0.957</td>
<td>m</td>
<td>m</td>
<td>0.919</td>
</tr>
<tr>
<td>\lim</td>
<td>lim</td>
<td>0.988</td>
<td>[</td>
<td>[</td>
<td>0.955</td>
<td>H</td>
<td>H</td>
<td>0.908</td>
</tr>
<tr>
<td>E</td>
<td>E</td>
<td>0.988</td>
<td>4</td>
<td>4</td>
<td>0.953</td>
<td>\log</td>
<td>log</td>
<td>0.908</td>
</tr>
<tr>
<td>8</td>
<td>8</td>
<td>0.988</td>
<td>\neq</td>
<td>≠</td>
<td>0.953</td>
<td>f</td>
<td>f</td>
<td>0.895</td>
</tr>
</tbody>
</table>

Table A.1: Symbols, their printed images and average precision per class, first part.
<table>
<thead>
<tr>
<th>Symbol</th>
<th>Image</th>
<th>AP</th>
<th>Symbol</th>
<th>Image</th>
<th>AP</th>
<th>Symbol</th>
<th>Image</th>
<th>AP</th>
</tr>
</thead>
<tbody>
<tr>
<td>n</td>
<td>n</td>
<td>0.893</td>
<td>,</td>
<td>,</td>
<td>0.797</td>
<td>\sigma</td>
<td>\sigma</td>
<td>0.592</td>
</tr>
<tr>
<td>z</td>
<td>z</td>
<td>0.890</td>
<td>p</td>
<td>p</td>
<td>0.792</td>
<td>}</td>
<td>}</td>
<td>0.571</td>
</tr>
<tr>
<td>k</td>
<td>k</td>
<td>0.888</td>
<td>\div</td>
<td>\div</td>
<td>0.790</td>
<td>s</td>
<td>s</td>
<td>0.548</td>
</tr>
<tr>
<td>\alpha</td>
<td>\alpha</td>
<td>0.887</td>
<td>c</td>
<td>c</td>
<td>0.785</td>
<td>Y</td>
<td>Y</td>
<td>0.538</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0.886</td>
<td>I</td>
<td>I</td>
<td>0.760</td>
<td>r</td>
<td>r</td>
<td>0.500</td>
</tr>
<tr>
<td>u</td>
<td>u</td>
<td>0.871</td>
<td>{</td>
<td>}</td>
<td>0.756</td>
<td>.</td>
<td>.</td>
<td>0.463</td>
</tr>
<tr>
<td>\cos</td>
<td>\cos</td>
<td>0.868</td>
<td>h</td>
<td>h</td>
<td>0.750</td>
<td>P</td>
<td>P</td>
<td>0.452</td>
</tr>
<tr>
<td>9</td>
<td>9</td>
<td>0.865</td>
<td>v</td>
<td>v</td>
<td>0.738</td>
<td>X</td>
<td>X</td>
<td>0.432</td>
</tr>
<tr>
<td>j</td>
<td>j</td>
<td>0.861</td>
<td>\geq</td>
<td>\geq</td>
<td>0.734</td>
<td>!</td>
<td>!</td>
<td>0.384</td>
</tr>
<tr>
<td>M</td>
<td>M</td>
<td>0.847</td>
<td>\tan</td>
<td>\tan</td>
<td>0.731</td>
<td>\lambda</td>
<td>\lambda</td>
<td>0.373</td>
</tr>
<tr>
<td>t</td>
<td>t</td>
<td>0.844</td>
<td>g</td>
<td>g</td>
<td>0.700</td>
<td>\prime</td>
<td>\prime</td>
<td>0.356</td>
</tr>
<tr>
<td>R</td>
<td>R</td>
<td>0.821</td>
<td>\times</td>
<td>\times</td>
<td>0.667</td>
<td>C</td>
<td>C</td>
<td>0.325</td>
</tr>
<tr>
<td>\beta</td>
<td>\beta</td>
<td>0.820</td>
<td>q</td>
<td>q</td>
<td>0.639</td>
<td>l</td>
<td>l</td>
<td>0.312</td>
</tr>
<tr>
<td>\pm</td>
<td>\pm</td>
<td>0.809</td>
<td>V</td>
<td>V</td>
<td>0.637</td>
<td>G</td>
<td>G</td>
<td>0.293</td>
</tr>
<tr>
<td>\ldots</td>
<td>\ldots</td>
<td>0.801</td>
<td>/</td>
<td>/</td>
<td>0.605</td>
<td>S</td>
<td>S</td>
<td>0.218</td>
</tr>
<tr>
<td>L</td>
<td>L</td>
<td>0.800</td>
<td></td>
<td></td>
<td>0.601</td>
<td>o</td>
<td>o</td>
<td>0.095</td>
</tr>
<tr>
<td>T</td>
<td>T</td>
<td>0.799</td>
<td>\gamma</td>
<td>\gamma</td>
<td>0.593</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table A.2: Symbols, their printed images and average precision per class, second part.
Bibliography


[57] Kelvin Xu, Jimmy Ba, Ryan Kiros, Kyunghyun Cho, Aaron Courville, Ruslan Salakhutdinov, Richard Zemel, and Yoshua Bengio. Show, attend and tell: Neural image caption generation with visual attention.


Statutory Declaration

I hereby declare that the thesis submitted is my own unaided work, that I have not used other than the sources indicated, and that all direct and indirect sources are acknowledged as references.

This printed thesis is identical with the electronic version submitted.

Balázs Bencze