System for Recognition and Evaluation of Handwritten Arithmetic Expressions

Vlado Galić, Tomislav Hrkać
University of Zagreb, Faculty of Electrical Engineering and Computing
Department of Electronics, Microelectronics, Computer and Intelligent Systems
Unska 3, 10000 Zagreb, Croatia
vlado.galic@fer.hr, tomislav.hrkac@fer.hr

***Abstract*—Recognition of mathematical expressions today is a very interesting area of application of deep learning. The problem is quite complex and requires a complex system to solve it. The problem becomes even more complex if arithmetic expressions are handwritten. The use of convolutional neural networks yields satisfactory results but leaves some room for improvement. In this paper, a simplified variant of the above problem is solved by using an approach that consists of extraction and classification of individual symbols using convolutional neural networks followed by syntactic parsing of the expression taking into account the symbol positions.**

I. Introduction

Recognizing and evaluating of handwritten arithmetic expressions is an enviable ability for people and it would be desirable to implement it on a computer. Some potential application areas for such systems include creating devices that speed up the tedious calculation of mathematical expressions, for example in context of "smart blackboards" that offer automatic solving of handwritten expressions or equation-solving on mobile devices. Another potential application is related to improving productivity of scientific writing by offering direct and easy way of input of mathematical expressions into computer. Because the problem is quite complex, it is necessary to use state-of-the-art computer vision technologies combined with formal rules designed to deal with the fuzzy syntax of handwritten arithmetic expressions. In this paper, we use only a subset of all mathematical expressions, namely the arithmetic expressions, in order to simplify the problem.

The objective of this paper is to implement a system for recognition and evaluation of handwritten arithmetic expressions by generating LaTeX code and the corresponding numeric result. In order to show the whole process as intuitively as possible, an analogy with the program language translator can be used. The whole process, from loading an image with an expression to generating LaTeX code and expression results, can be compared to the process of translating the program written in a high-level programming language. Therefore, this system is analogous to the compiler. As it is known, the compiler translates a program written in a higher-level language into a language comprehensible to the processor or, to put it another way, translates the code file into machine commands. Similarly, this system translates an image with an arithmetic expression into the LaTeX code and generates the expression evaluation result. The compiler has 3 stages of code analysis: lexical, syntax and semantic analysis. Analogously, our system has 3 phases which correspond to the mentioned phases. They are:

- **Lexical analysis** - Determining the locations of symbols and their type. A convolutional neural network classifier and some well-known computer vision algorithms are used to perform this phase. Details about this phase are described in section III.
- **Syntax analysis** - At this stage, previously determined positions of the symbols become important. This procedure ensures that the symbols are in the right positions and in the right structure. For example, if we have a fraction bar symbol, it must have other symbols above and below it because otherwise the fraction is not valid. Each trigonometric operation must necessarily have parentheses behind it, otherwise a syntax error occurs.
- **Semantic analysis** - The difference between this phase and the syntax analysis is that this phase takes care of logical constraints that do not depend on the position and structure of the symbol. For example, a denominator must not be zero, logarithm must not have a negative number as an argument etc.

Generation of the LaTeX code and the numeric result of the expression evaluation corresponds to the code generation phase, which usually follows above discussed three phases of analysis in higher-language compilers.

In order to speed up the process and make more efficient use of computing resources, syntax and semantic analysis are performed in parallel. The code generation is also performed in parallel to these two phases, by inserting appropriate predefined LaTeX expression into appropriate positions of the final LaTeX code. More about the specific ways of applying these phases in the system is described in the section IV. It is important to note that dividing the process into such phases allows us to effectively monitor errors. Therefore, it can always be determined at what phase the error occurs and why.

II. Related work

Mathematical expression recognition has been a subject of research for a long period of time, the first attempt appearing as early as 1967 [1]. However, the problem remains challenging, especially for handwritten expressions, due to numerous factors, such as high variability in handwriting among different people, large number of mathematical symbols, and high variability in mathematical expression structure (two-dimensional spatial arrangement of symbols).

A large number of methods has been proposed to deal with this task. These methods can generally be categorized either as offline or online. Offline recognition systems start from a static image of the expression, obtained either by scanning a paper document, taking an image of the written expression by a camera, or drawn on a touch screen of a device. Online recognition systems, on the contrary, enter the expression as a sequence of strokes, i.e. the sequences of points recorded along the pen trajectory. In this way, in online recognition system, the time information is available in addition to spatial arrangement of the expression parts.

Generally, mathematical expression recognition involves three tasks [2]: (1) segmentation of individual symbols – grouping of pixels or strokes belonging to the same symbol; (2) symbol recognition – the task of labeling symbol candidates to assign each of them a corresponding symbol class; and (3) structural analysis – identification of 2D spatial relations between symbols to produce the interpretation of the expression as a whole.

Standard survey papers (e.g. [2], [3]) give an overview of classical computer vision and machine learning techniques used to address these tasks up to 2000. and 2012, respectively. More recently, approaches based on deep learning and convolutional neural networks (CNNs) have achieved impressive results in many computer vision applications, so it is not surprising that they are more and more commonly applied in handwritten mathematical expression recognition systems, either to recognize the expression image as a whole, or to detect and/or classify individual symbols before applying some more traditional method for structural analysis and generation of the result.

For example, Tran et al. [4] employ a single shot multi-box detector (SSD) – a state-of-the-art deep convolutional neural network architecture for object detection and classification, trained to detect and recognize individual symbols in a handwritten mathematical expression image. This is followed by structure analysis based on open-source DRACULAE parser [5], since it has high accuracy when the symbols are correctly detected. LaTeX strings corresponding to the input image are created as a final result.

Several other works attempt to design systems that can be learned end-to-end, usually to generate LaTeX expressions corresponding to the input images. Due to variable number of symbols in expression images and variable length of the generated expressions, recurrent neural networks (RNNs) are usually used in these situations. Zhang et al. [6] propose an end-to-end approach called Watch, Attend and Parse (WAP), an encoder-decoder type deep convolutional architecture in which a standard convolutional neural network is employed as an encoder that takes handwritten expression images as input, and a RNN decoder equipped with an attention mechanism as the parser to generate LaTeX sequences. In the subsequent work [7], they improve the encoder by employing densely connected convolutional networks to strengthen feature extraction and facilitate gradient propagation, and introduce a novel multi-scale attention model employed to deal with the recognition of mathematical symbols in different scales. Similarly, Shan et al. [8] propose a robust encoder-decoder learning framework for offline handwritten mathematical expression recognition – another encoder-decoder architecture based on improved DenseNet network as encoder and a gated recurrent unit (GRU) RNN with the attention model as decoder.

In contrast to the above works, this paper adopts a simpler approach that consists in first segmenting the individual symbols, then recognizing them using a CNN-based classifier, and finally parsing the expression by analyzing 2D spatial arrangement of the symbols. The adopted approach has some limitations, mainly consisting in the requirement to the user to pay special attention that individual symbols are carefully written (they may not contain disconnected parts and may not touch each other) in order to facilitate the segmentation process; however, if this requirement is satisfied, the system shows very good performance in practice.

III. Symbol recognition

The input to the expression recognition system is a binary image containing handwritten arithmetic expression consisting of one or more supported mathematical symbols. The first phase of the proposed algorithm is to extract the individual symbols, determine their positions and recognize their types or classes. The term ”symbol” in this paper also refers to the area of pixels in the image that make up that symbol. This section explains how to find each symbol position in the image and determine its type. First of all, the equation image must be properly prepared for the recognition process, otherwise significant errors can occur. Therefore, it is very important to follow the main rules, which are:

- **Writing symbols in one piece** - this rule is one of the limitations that must be adhered to in this problem-solving approach. Individual symbols must not contain multiple parts (curves). An example of correctly and incorrectly written symbols according to this rule is shown in the first row of Fig. 1.
- **Symbols must not touch each other** - accidental touching of symbols may cause inability to predict the type of a symbol. An example of correctly and incorrectly written symbols according to this rule is shown in the second row of Fig. 1.
- **Using binary images** - i.e. the expressions are white on black background; this increases the accuracy of finding a continuous monochrome group of pixels (correctly written symbol).

Adherence to these rules is necessary to correctly determine the position and type of all symbols. It is important to note that the expression image can be of arbitrary dimensions.

A. Position

Determining the positions of the symbols in the image is performed by looking for contours of the connected pixel

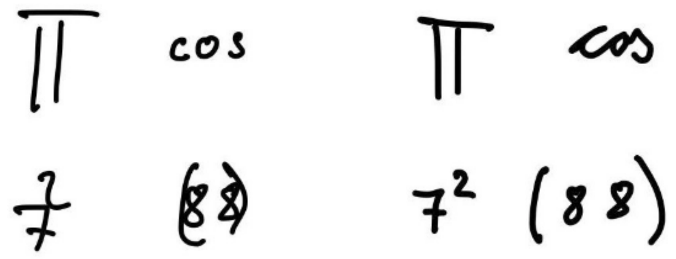


Fig. 1: Correctly (right) and incorrectly (left) written symbols.

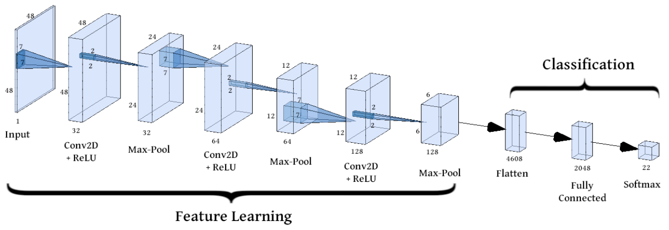


Fig. 2: Architecture of used model for symbol classification

groups in the binary image. In simple language, a contour can be defined as a curve joining all the continuous points along the edge of the connected group of pixels, i.e. of the group of pixels having same intensity or color in binary image. This is the reason for using binary pictures. In binary images, each pixel is represented as 0 or 1. The pixel area of each contour (symbol) is determined using well-known algorithms for binary images, such as border-following algorithm of Suzuki and Abe [9] implemented in the OpenCV library's *cv2.findContours()* method.

After finding the pixel area for each symbol, the areas of their bounding boxes are cut from the image and each of them represents a new image. Important features of the new images that will be used later are: *center of the image*, *minimum/maximum x and y*, *width*, *height*. These features will be crucial for creating expression syntax.

B. Symbol types

Table I shows the 22 types (classes) of symbols used in this paper. It is important to note that for the multiplication operation the symbol '$\times$' is used and for the division operation the symbol '$-$' which is also a minus sign. So one symbol represents two different mathematical operations. Determining whether it is a minus or a fraction bar is performed during the expression evaluation (syntax analysis), by considering the surrounding above and below the symbol, as explained in section IV-A. The system supports basic arithmetic operations with integer numbers and two basic constants. The decimal point is not supported because the corresponding symbol would be too small in the image and easily confused with noise. Decimal numbers, however, can be used indirectly, by expressing them as fractions of integers.

Determining the type of symbol from a piece of an image can be solved by an appropriate type of classifier. The classifier used in this paper was implemented using a convolutional neural network. Its network architecture is shown in Fig. 2. The first part of the network is a feature extractor that is used to extract features from the image. The features are then passed to the input of a fully connected neural network. The output of this network consists of 22 neurons corresponding to 22 classes (one class for each symbol type). Based on the input features, this network predicts what type of symbol is in the image. It is important that the images for which prediction is made are in the same format as the training images. The previously found symbol areas can be of different dimensions, so they must be scaled to the agreed dimensions to satisfy the correct image format. Our classifier architecture requires 48x48 pixels image as the input layer, so each new image is resized to 45x45 pixels and then expanded to the required dimension by adding a blank frame of the appropriate size.

If the classifier misclassifies a symbol, the result is a lexical error that cannot be corrected. Therefore, the classifier must be very accurate. More about the results of the classifications are discussed in the section V-A.

TABLE I: Supported symbols

Category	**Samples**
Numbers	0, 1, 2, 3, 4, 5, 6, 7, 8, 9
Constants	e, π
Parentheses	) , (
Trigonometry	sin, cos, tan
Basic operators	$+$, $-$, x
Other operators	$\sqrt[n]{}$, log

IV. Construction and evaluation of expression

The construction and evaluation of the expression are performed in parallel. The goal of this process is to generate a LaTeX code and a numeric result for the given arithmetic expression. In order to accomplish this task, the arithmetic expression must be divided into several smaller arithmetic expressions. After that, each of the expressions is solved and the results are eventually merged into the final solution. Therefore, the algorithm used in this process is a "divide and conquer" type of algorithm. Generation of Latex code and the result of an arithmetic expression is realized through a function implemented in the program. Fraction, parentheses and exponentiation can be nested so our function must support such nesting. This is achieved by using a recursive function. Multi-branched recursion is usually used in "divide and conquer" algorithms. So our recursive function will process the expression piece by piece, and for each part of the expression that is nested it will call again itself, but this time it will process the nested part. In addition to being robust to nesting, the function must also support the processing of various mathematical operations and signs. The body of such a function consists of the sequentially implemented processing features described in the subsections A-G. The processing features must be implemented in the same order as listed below.

For better understanding of the algorithm and subsections A-F, an example of arithmetic expression processing using a recursive function is shown in equations (1) - (7).

A. Fraction

First of all, it has been said earlier that the minus sign belongs to the same class as the fraction bar sign. Therefore, it is necessary to examine all the detected ”minuses” and determine if any of them is a fraction bar. To do this, it is first necessary to sort all detected minuses in descending length. They are then processed one by one, by counting the symbols above and below the line (taking into account the previously determined positions of individual symbols). There are three possible outcomes of this analysis:

- **Zero symbols above and below** $\rightarrow$ Minus sign
- **Non-zero symbols above and below** $\rightarrow$ Fraction bar
- **Else** $\rightarrow$ Syntax error

Once fraction bars have been detected, all fractions need to be resolved and replaced with constants that correspond to their values. In order to resolve a single fraction, it is necessary to produce two expressions, namely $Expression_1$ with all the symbols above the line and $Expression_2$ containing the symbols below the line. Then both expressions are resolved recursively and their results are divided by each other to give the result of the fraction, as illustrated in equations (6) - (7). The whole fraction is then replaced by the division result and the LaTeX expression of that fraction is `\frac{<Expression_1>}{<Expression_2>}`. Of course, it is also important to check whether the divisor is non-zero. This check belongs to the semantic analysis and if the expression does not satisfy this rule the system reports a semantic error.

B. Parentheses

At the beginning of this and the following stages, it is important to sort all the symbols each time by their position, from left to right. Parentheses are resolved by replacing the whole expression within the parentheses by the result of resolved expression inside the parentheses. The LaTeX expression for this type of expression is `(<Expression>)`. Parentheses can be nested, so care must be taken when determining the start and end parenthesis, e.g. Eq. (3) - (4). For an open parenthesis, its corresponding closed parenthesis is determined by counting the open and closed parentheses between them. The number of these parentheses must be equal. Of course, incorrect use of parentheses leads to a syntax error and empty parentheses represent a semantic error.

C. Exponentiation

The proper form of the exponentiation expression looks like this: $< Expression_1 >^{<Expression_2>}$. The main rule used here is that all subsequent characters whose minimum y coordinate is greater than the y coordinate of the center of the last character in $Expression_1$, represent $Expression_2$. Exponentiation can also be nested, so it is necessary to first resolve $Expression_2$. Both expressions are replaced by the (constant) resultant exponentiation of the two resolved expressions and the LaTeX expression is: `{Expression_1}^{Expression_2}`. The semantic rule that applies here is that the expression zero to the power of zero is forbidden.

D. Trigonometry

The syntax rule for trigonometric functions is that there must be parentheses behind the function symbols that contain the function argument. Also, parentheses must not be empty. This phase is very similar to the parenthesis phase. The only difference is that in front of the parenthesis stands the symbol of some trigonometric function. So after the expression in parentheses is resolved, the trigonometric function is applied to it, and the whole expression, including the function symbol and parentheses, is replaced by the result of the function. Also, a LaTeX expression is generated such as for example `\cos{<Expression>}`. The semantic rule for this case is that the value of the argument must be in the domain of the function.

E. Numbers and constants

After the previous stages, in the expression such as $24+34$, the parts 24 and 34 are represented only by sets of individual symbols: 2, 3 and 3, 4. At this stage, groups of symbols consisting of digits, are converted to numbers (constants). There are also special symbols that can be replaced with a constant without any processing. In this paper, these are π and e, and they are replaced by their true values.

F. Multiplication

Multiplication is a mathematical operation that takes precedence over addition and subtraction, so it must be performed before these two operations. Multiplication by more than two factors requires performing multiplication operations sequentially from left to right. Therefore, at this stage, the symbols are processed sequentially from left to right to find the multiplication sign and then the multiplication is performed. Then the whole expression is replaced by the result of the expression (constant) and the corresponding LaTeX expression is generated.

Furthermore, it is necessary to consider when two or more constants are adjacent to each other (for example 32π). This is also considered as two constants that multiply.

G. Addition and subtraction

Only constants, plus signs and minus signs are left at the end of the process. Solving such expressions is very simple. First, the cumulative variable (accumulator) is initialized and then sequentially from left to right the new constants are summed/subtracted with the accumulator. A syntax error may occur if there is more than one operator side by side or if the expression begins or ends with an operator. This leaves room for future implementation of unary operations. The result of this processing is also the final result and represents the LaTeX code together with the numerical result as a solution of a given arithmetic expression.

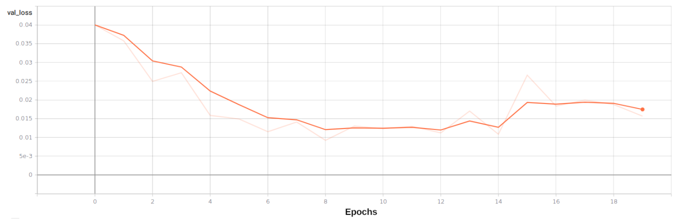


Fig. 3: Validation loss function (lighter line – original; darker line – smoothed)

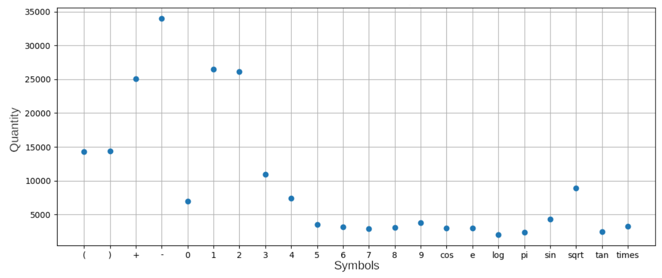


Fig. 4: The distribution of symbol classes in the dataset

Equations (1) - (7) show an example of evaluating an expression by using the function *solve*. In these equations, $solve_d(<Expression>)$ represents a recursive function with expression as argument and d as a depth.

$$solve_0(\frac{3\times(2+(4+3))}{3+7}) \tag{1}$$

$$solve_0(\frac{solve_1(3\times(2+(4+3)))}{solve_1(3+7)}) \tag{2}$$

$$solve_0(\frac{solve_1(3\times solve_2(2+(4+3)))}{solve_1(3+7)}) \tag{3}$$

$$solve_0(\frac{solve_1(3\times solve_2(2+solve_3(4+3)))}{solve_1(3+7)}) \tag{4}$$

$$solve_0(\frac{solve_1(3\times solve_2(2+7))}{solve_1(3+7)}) \tag{5}$$

$$solve_0(\frac{solve_1(3\times 9)}{solve_1(3+7)}) \tag{6}$$

$$solve_0(\frac{27}{10}) = 2.7 \tag{7}$$

V. Experimental results

A. Training results

For training the dataset Handwritten math symbols dataset from Kaggle [10] was used. It contains 22 symbols and consists of a total of 211239 images. We used 18% of the images for validation and 10% for testing. The symbols quantity distribution is shown in Fig. 4. The entire training took place in 20 *epochs* with a *batch size* of 100 images. Fig. 3 shows the validation loss function. The figure shows that the loss function eventually reached very small values, which means that the classifier is quite precise in determining the types of symbols in the validation dataset. In order to verify that the classifier model is truly accurate, it is necessary to test the accuracy of the model on the test dataset. The size of the test dataset is 21114 images. After testing, the accuracy of the classifier model is 99.78%, which means that the model made a wrong prediction for 49 images. Considering that the symbols are handwritten and there are many symbols that are illegible, the accuracy of the model is more than satisfactory. Additional experiments carried out on the model have shown that the samples which obey the above mentioned rules were predicted with very high accuracy.

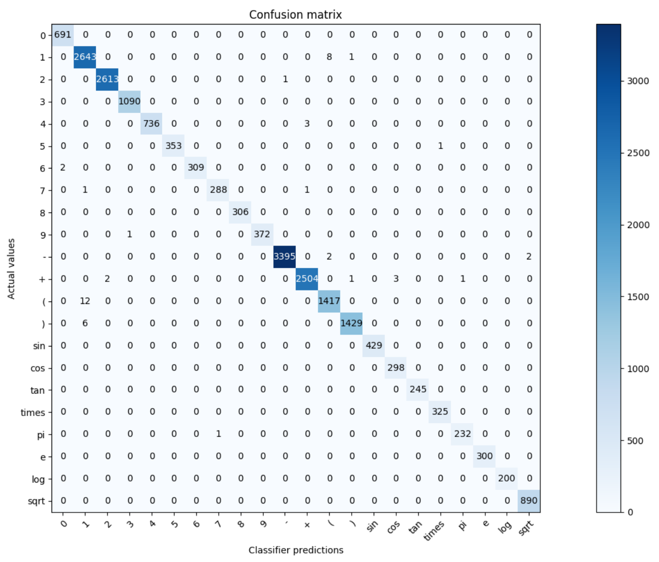


Fig. 5: Confusion matrix

Another good way to show the accuracy of a classifier is the *confusion matrix*. The confusion matrix for the trained classifier tested on the test dataset is shown in the Fig. 5 and shows how many times the classifier correctly predicted the class and how many times it has missed and selected another (wrong) class. From our confusion matrix one can see that the classifier rarely makes mistakes, and even when it is wrong, in most cases it misclassifies visually quite similar characters as, in our example, number 1 and parenthesis.

B. An illustrative example

This section presents the test results of a developed system as a whole, as described in the section III. This means recognizing the position and type of the symbol in the image, and then constructing and evaluating the expression. An example result of the position detection of individual symbols is shown on the left side of the Fig. 6 and the result of symbol classification on the right side of the same image. The final processing result of Fig. 6 is:

LaTeX: `\frac{15^{2}+\sin(\frac{\pi}{2})^{4+7}}{226}`

Result: 1.0 (Time: 0.38 seconds)

The system was extensively tested on a number of inputs of different complexity. Some of the successful and interesting

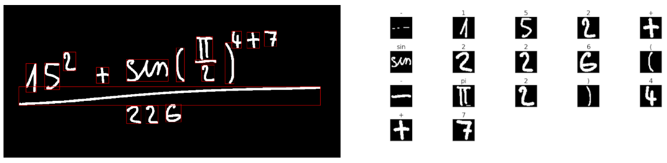
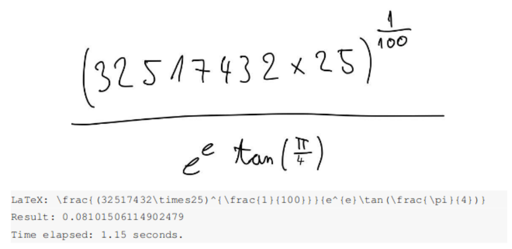
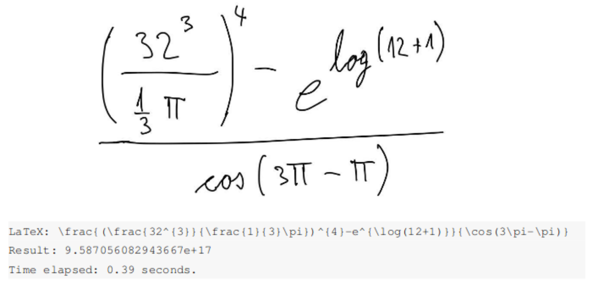


Fig. 6: An example of an arithmetic expression with found symbols positions (left) and symbol types (right)



```
LaTeX: \frac{(32517432\times25)^{\frac{1}{100}}}{e^{e}\tan(\frac{\pi}{4})}
Result: 0.08101506114902479
Time elapsed: 1.15 seconds.
```

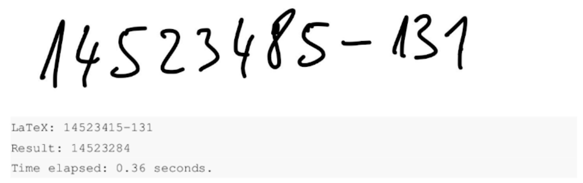
(a)



```
LaTeX: \frac{(\frac{32^{3}}{\frac{1}{3}\pi})^{4}-e^{\log(12+1)}}{\cos(3\pi-\pi)}
Result: 9.587056082943667e+17
Time elapsed: 0.39 seconds.
```

(b)

Fig. 7: Both examples show a solution for combining various mathematical operations into different structures



```
LaTeX: 14523415-131
Result: 14523284
Time elapsed: 0.36 seconds.
```

Fig. 8: An example of a system (lexical) error - wrong classification of the sign '8'

examples of system testing are shown in the Fig. 7. One unsuccessful example is shown in Fig. 8 where digit 8 is wrongly recognized as 1 during the classification, resulting in lexical error. Overall, when care is taken to enter the expressions carefully, the system performs well in most cases, although there is room for some improvements.

VI. Conclusion

We implemented and described a computer vision system for recognition and evaluation of handwritten arithmetic expressions, capable of generating LaTeX expressions and numeric results from binary images. The described system imposes certain restrictions to users, specifying the way the symbols must be written, to facilitate the crucial first step of the individual symbol segmentation, but if care is taken of these restrictions, the system performs reasonably well. There are some possible improvements, however, that we leave for future work. In its current form, the system expects binary image without noise as input, for example made by drawing the expression in black color on white background using a touch screen device. One improvement would be to implement good noise removal and binarization technique to enable entering less constrained images, e.g. taken by a camera. The current system is also sensitive to the thickness of the lines in the input images, since the lines in the training images are very thin. The problem could be addressed either by augmentation of training set with lines of different thickness or by thinning of the lines in the input images. Finally, using recurrent neural networks could help with restrictions that are currently imposed to the way the symbols must be written.

References

[1] R. H. Anderson, "Syntax-directed recognition of hand-printed two-dimensional mathematics," in *Symposium on Interactive Systems for Experimental Applied Mathematics: Proceedings of the Association for Computing Machinery Inc. Symposium*. New York, NY, USA: ACM, 1967, pp. 436–459.

[2] R. Zanibbi and D. Blostein, "Recognition and retrieval of mathematical expressions," *Int. J. Doc. Anal. Recognit.*, vol. 15, no. 4, pp. 331–357, Dec. 2012.

[3] K. Chan and D. Yeung, "Mathematical expression recognition: a survey," *IJDAR*, vol. 3, no. 1, pp. 3–15, 2000.

[4] G. S. Tran, C. Huynh, T. Le, T. Phan, and K. Bui, "Handwritten mathematical expression recognition using convolutional neural network," in *2018 3rd International Conference on Control, Robotics and Cybernetics (CRC)*, Sep. 2018, pp. 15–19.

[5] R. Zanibbi, D. Blostein, and J. R. Cordy, "Recognizing mathematical expressions using tree transformation," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 24, no. 11, pp. 1455–1467, Nov 2002.

[6] J. Zhang, J. Du, S. Zhang, D. Liu, Y. Hu, J. Hu, S. Wei, and L. Dai, "Watch, attend and parse: An end-to-end neural network based approach to handwritten mathematical expression recognition," *Pattern Recognition*, vol. 71, pp. 196–206, 2017.

[7] J. Zhang, J. Du, and L. Dai, "Multi-scale attention with dense encoder for handwritten mathematical expression recognition," in *24th International Conference on Pattern Recognition, ICPR 2018, Beijing, China, August 20-24, 2018*, 2018, pp. 2245–2250.

[8] G. Shan, H. Wang, and W. Liang, "Robust encoder-decoder learning framework towards offline handwritten mathematical expression recognition based on multi-scale deep neural network," 2019.

[9] S. Suzuki and K. Abe, *Computer Vision, Graphics, and Image Processing*, no. 1, pp. 32–46.

[10] [Online]. Available: https://www.kaggle.com/rtatman/handwritten-mathematical-expressions