Mathematical Expression Recognizer

ChanMin Kim [*] HeeHoon Kim Sanghyeok Park
Department of Computer Science and Engineering
Seoul National University
kcm1700@snu.ac.kr, csehydrogen@gmail.com, pps987@snu.ac.kr

Abstract

We implemented simple mathematical expression recognizer using Convolutional Neural Network (CNN). Our recognizer takes pictures of one line math expression, and generates $\TeX$form of the expression. It can only recognize digits, some arithmetic operators, parenthesis, π, and e. Our CNN model is modified LeNet, and the network was trained and tested with Caffe, using MNIST and CROHME. MNIST is a data set of handwritten digits, which contains 60000 train and 10000 test images. CROHME is a data set of handwritten digits and other symbols that we need. It has 38248 images. With those data sets, our model shows about 98% accuracy on test set. Our recognizer works with following steps; normalization, segmentation, and recognition. In normalization part, we used local thresholding method. In segmentation part, we applied Flood-Fill and our own merging algorithm. Finally, in recognition part, we obtain the probability for each symbols with our own network; however, we used dynamic programming approach to avoid nonsense expressions. In order to determine which expression flow makes sense, we defined states contains probabilities with respect to a written component, a parenthesis depth, and a kind of guessed symbol. We calculated all states in $O(n^2\sigma^2)$ time using dynamic programming, which is enough fast. With taking the maximum probability of states, we can avoid nonsense expressions, such as $1(+30$. This approach was very important. By our confusion matrix of our model, there exist miss rates over than 10%. However, if symbols that make confusion are similar but in different kind, we can handle with dynamic programming. We used another information of expression: the context of expressions; therefore, we could get better results. We made a web demo application using above approaches at `http://chanmin.kim/image-expr/`.

[*]Author names are listed in alphabetical order. We've contributed to the project equally.

1. Introduction

We implemented a simple mathematical expression recognizer which takes pictures of math expressions and generates $\TeX$form of expressions. Our program has some restrictions. It can only recognize expressions that has only digits, arithmetic operators($+, -, \times, /, \div$), parentheses, π, e. Expressions also should be in one line, and not have superscripts or subscripts. Background should be bright, and the expression should be dark.

We divided this problem into three parts, normalization, segmentation, and recognition. In normalization part, we binarized our gray scale natural image, and rotated it to fit in straight rectangle box. In binarization, we applied local thresholding method. Then we use ternary search method to find best angle we have to rotate.

In segmentation part, basically we split binarized symbols into connected components using Flood-Fill. However, we had to merge components into same symbol. For example, $=$ symbol is one symbol but it has two components. Because our expressions is in only one line, we measured only X-coordinates and overlap ratio of components. If one of the two components overlap ratio is larger than the threshold, we merged those components into same symbol. We chose the threshold 50%.

In recognition part, we determined which symbol each merged component actually is. The probability for each component to be a specific symbol was calculated from neural network. Our neural network model is modified LeNet[3]. The network was trained and tested with Caffe, using MNIST[6] and CROHME[1] data sets. In order to prevent nonsense predictions (consecutive operators, unmatched parentheses, etc.), we used dynamic programming approach.

2. Background

There are a number of prior studies in the field of recognition of mathematical expressions.

Michael Nielsen[4] wrote an introductory book on neural networks. The book describes the basics of neural network.

MNIST[6] data sets are famous for handwritten digits.

One of the most popular mathematical expression recognition competition is CROHME, which stands for 'Competition on Recognition of Online Handwritten Mathematical Expressions'[1]. They provide rich train data set and test data set of handwritten mathematical expressions in InkML. InkML is a file format for representing trace of handwriting.

There was a student project by Vitali Sepetnitsky and Yakir Dahan[5], which recognizes book printed mathematical formulas. There was also a related course project by Alistair Dobke and Mark Mann[2]. The project recognizes handwritten mathematical expressions just as our project.

3. Approach

3.1. Preprocess

Since we use neural network, preprocesses, like data generation, modeling, and training, are as important as our main algorithm. We covered details in this section.

3.1.1 Training and Test Data Generation

Widely used databases of handwritten mathematical symbols have different formats. We needed to convert them in a uniform format which is easy to train and test under Caffe framework. We used MNIST and CROHME, which were enough for our purpose.

MNIST data are available as idx files, which contain 60000 train images and 10000 test images. Each image is a anti-aliased grayscale handwritten digit image, which is normalized to 20 by 20 image preserving aspect ratio, and centered in 28 by 28 image by computing the center of mass of pixels. We used this format as standard for training.

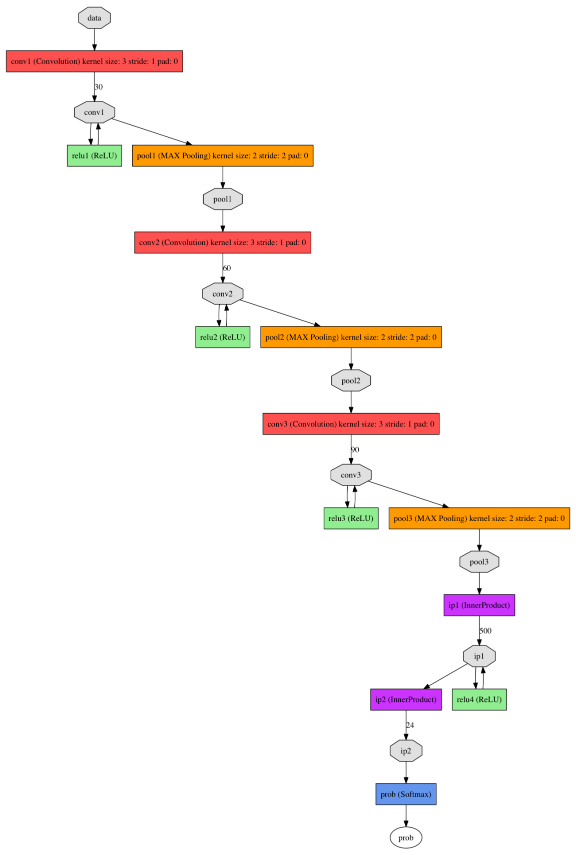
CROHME data was used to cover symbols other than digits. We only used data containing digits, basic operators(+, -, $\times$, $\div$, /, =), parentheses([,], (,), {, }), π, and e. They are formatted as inkml, which contains stroke information only. Lines are generated by linear interpolation from coordinates. Then we applied anti-aliasing with 4 strengths since stroke width is undefined. Finally, we converted them into the same padded-size image as MNIST.

From two databases we get $212992 (= 60000 + 38248 * 4)$ train images and $21772 (= 10000 + 2943 * 4)$ test images. We merged them into LMDB database file for faster input.

3.1.2 Neural Network Modeling

Our network model should be proper to categorize handwriting symbols. We took approach to take existing model, LeNet, and modify for better predictions. Our full model is depicted in Figure 1.

First part of the model consists of 3 convolution steps, compared to 2 in LeNet. Each step has Convolution layer, ReLU layer, and Pooling layer. They output 30, 60, 90



data
conv1 (Convolution) kernel size: 3 stride: 1 pad: 0
30
conv1
relu1 (ReLU)
pool1 (MAX Pooling) kernel size: 2 stride: 2 pad: 0
pool1
conv2 (Convolution) kernel size: 3 stride: 1 pad: 0
60
conv2
relu2 (ReLU)
pool2 (MAX Pooling) kernel size: 2 stride: 2 pad: 0
pool2
conv3 (Convolution) kernel size: 3 stride: 1 pad: 0
90
conv3
relu3 (ReLU)
pool3 (MAX Pooling) kernel size: 2 stride: 2 pad: 0
pool3
ip1 (InnerProduct)
500
ip1
ip2 (InnerProduct)
relu4 (ReLU)
24
ip2
prob (Softmax)
prob

Figure 1: Our neural network model

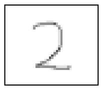
channel images of decreasing size, respectively, which are also larger than LeNet.

Convolution layer picks up overall features of images from previous layers. (See Figure 2) LeNet uses kernel of size 5, but we reduced them to 3. Since we try to categorize twice more symbols from the same size images, large kernel has danger of losing too much information.

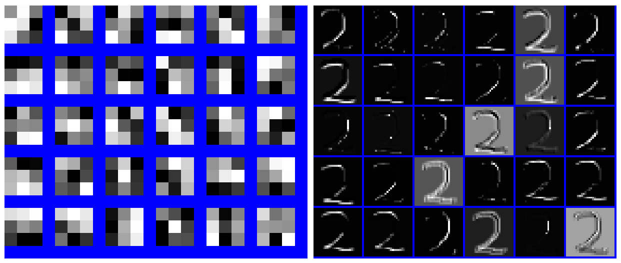
ReLU layers are added for better prediction. We used leaky ones, i.e., negative slopes are 0.01. It is known that ReLU layers increase non-linearity, resulting in better and faster train. Dropout with probability 0.5 is also applied to prevent overfitting and make train faster.

Pooling layers are also placed after ReLU layers. They use max function to decrease variance during detection. Also, kernel of size 2 and stride of 2 effectively decrease image size to half in each dimension.

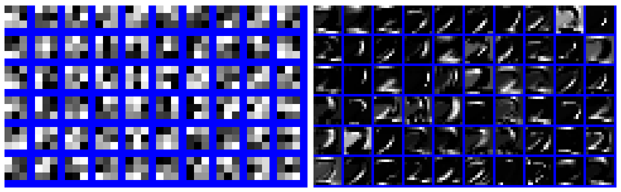
Latter part of the model consists of 2 InnerProduct layers and a Softmax layer. InnerProduct layers are fully-connected layers, which don't have spatiality anymore and prepare for output. Second InnerProduct layer produces 24 dimension vector, which is the same with the number of kinds of symbols. Then Softmax layer takes this vector and outputs estimated probabilities for each symbol.



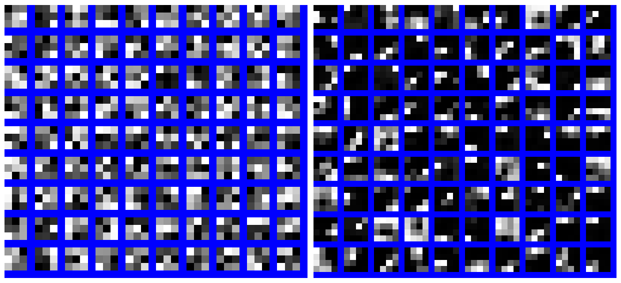
(a) Input Image



(b) First convolution layer (30 channel)



(c) Second convolution layer (60 channel)



(d) Third convolution layer (90 channel)

Figure 2: Visualization of each convolution layer. Left images are trained multichannel masks. Right images are transformed multichannel input images after each convolution step. First step contains general features(lines, edges, ...), while latter step contains higher level features.

3.2. Algorithm

Our algorithm consists of several steps. We first binarize the image and rotate the image. Then we segment the image into characters. The final step is to apply dynamic programming to guess each symbols.

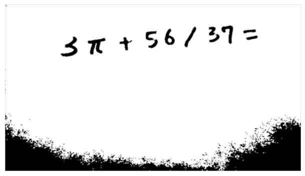
3.2.1 Binarization

The binarization algorithm consists of several steps as shown in **Algorithm 1**. The algorithm calculates nearby pixel average and wider area average for each pixel. Then the function calculates the differences and check the difference against the `threshold`. The result is `height` $\times$ `width` binarized image. W_{large} and W_{small} are parameters to the algorithm. We used 350 and 30 for W_{large} and W_{small} respectively.

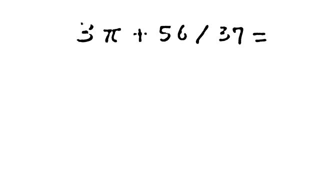
Our binarization method uses only neighborhood information. We were forced to apply local thresholding method instead of global thresholding method. Some parts of the background were sometimes darker than the text. See Figure 3 for example.

Algorithm 1 Binarization

```
1: function BINARIZE
2:     image ← grayscale(image)
3:     ns ← max (width / $W_{large}$, height / $W_{large}$)
4:     nearsum ← image ∗ (ns × ns mean filter)
5:                                  ▷ ∗: convolution operator
6:     fs ← max (width / $W_{small}$, height / $W_{small}$)
7:     farsum ← image ∗ (fs × fs mean filter)
8:     threshold ← $\frac{1}{6}$ min (nearsum - farsum)
9:     return nearsum < farsum + threshold
```



(a) global thresholding



(b) local thresholding and rotation

Figure 3: It is nearly impossible to separate the text by usual thresholding methods.

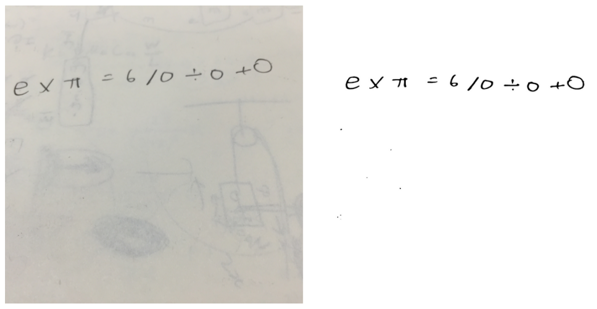


Figure 4: Example image of noisy background

3.2.2 Rotation

We rotated the image by appropriate angle for better segmentation and recognition. We set the object function D:

$$D(\theta) = \text{std}[x_i \sin\theta - y_i \cos\theta] \text{ for all black pixels } i \quad (1)$$

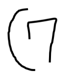


Figure 5: '(' and '7' shares few X-coordinates

This function means the standard deviation of y coordinates of the rotated image. We want to minimize the object function D with regarding to θ.

Our algorithm finds the best angle by using ternary search on range $[-\frac{\pi}{6}, \frac{\pi}{6}]$. Since finding the best angle is straight-forward, we'll skip the details. See Figure 3 and Figure 4 to see the effect of the rotation step.

3.2.3 Flood Fill

Basically we assume that connected pixels are in same symbol. Therefore, using Flood-fill, we decomposed image into components which contain only connected points. However, it is not enough to determine symbols.

3.2.4 Region Merge

There exist two problems using only Flood-fill.

First, some symbols have several disconnected components. For example, $\div$ has at least three disconnected components. Second, noise problem; some connected components in ideal might be written by several disconnected components.

To solve those problems, we should merge some components. One of the simple method of merging is assuming components are in same symbol if and only if there exist points of each components which have same X-coordinates. However, this simple method makes another problem.

In Figure 5, '(' and '7' component shares few X-coordinates; therefore, they will be merged. However, we expect that those components should not be merged.

Therefore, we should restrict our merging method. Our idea is that merge two components if and only if one of the components has common X-coordinate interval more than 50%. In formal, let A and B be two components which we want to determine whether we should merge. Let A_l, B_l be leftmost point of A and B, also A_r, B_r be rightmost point of A and B, respectively. Let $[c_l, c_r]$ be interval that is $[A_l.x, A_r.x] \cap [B_l.x, B_r.x]$ if it exists. Finally, let d be $c_r - c_l$. Then, we merge A and B if and only if

$$\frac{d}{A_r.x - A_l.x} > 0.5 \text{ or} \tag{2a}$$

$$\frac{d}{B_r.x - B_l.x} > 0.5 \tag{2b}$$

If we sort components with decreasing order of leftmost point of each component, we can easily apply merging algorithm in linear time.

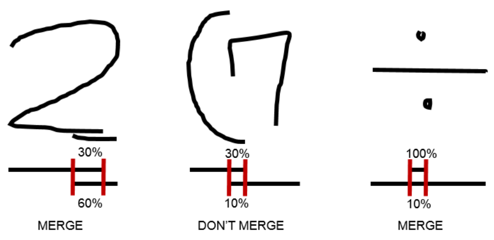


Figure 6: Examples of merging algorithm

See Figure 6 to obtain that how this algorithm works and solves problems.

3.2.5 Single Symbol Recognition

Now we know which symbol each black pixel belongs to. To use our trained neural network, we should transform each symbol into standard input image form, as mentioned in train data generation. For i-th symbol, we extracted minimal area that contains all black pixels of the i-th symbol. Then the area was resized into 20 by 20 image with anti-aliasing technique, and 4 pixel padding was added to each side.

We processed the image with our neural network, and we get P_i, 24-dimension vector for the i-th symbol. The dimension is determined by the number of symbols that we selected. See Table 2 for full list of symbols.

The output layer of our neural network model uses softmax function. Therefore, we can interpret j-th value of the vector, $P_{i,j}$, as the predicted probability of the i-th symbol being j-th class. Taking $argmax$ on j gives us the most probable class for the i-th symbol.

However, probability itself is not enough for parsing whole expression, since we don't benefit from context analysis. Thus, our algorithm goes for another step, Dynamic Programming, with probabilities as some initial values.

3.2.6 Dynamic Programming

We finally guess the symbols in this stage. It might be okay to just guess the segmented symbols independently, but we did more to get better result.

We have put some restrictions on guessing. We know that parentheses of correct mathematical expressions are matched correctly. For example, $(1+4($ doesn't make sense. We also know that duplicate operators doesn't appear in expressions except for unary operators. For example, $3 \times \div 4$ doesn't make sense. There are more restrictions that can be used for guessing.

Here are some of the rules that is used in our algorithm.

- Parentheses are correctly matched.
- There are no digits right after π and e.
- There are no duplicated binary operators.
- There are no binary operator after opening parentheses.
- There are no operator before closing parentheses.
- $\cdots$

We made a dynamic programming algorithm to make use of the rules above. Let's define a function $d(i, open, kind)$ for DP.

$d(i, open, kind) :=$ Maximum probability of the guess when the first i segmented parts are used, the parenthesis is opened $open$ times, and the last character was guessed as $kind$-th symbol.

Then the formula 3 for calculating function $d(i, open, kind)$ look like below.

$$d(i, open, kind) = \max \begin{cases} 1 & \text{if } (i, open, kind) = (0, 0, \square) \\ 0 & \text{if } i, open, \text{ or } kind \text{ is invalid} \\ 0 & \text{if restricted by some rules} \\ P_{i,kind} \cdot d(i-1, open-1, ?) \\ \quad \text{if } kind \text{ is } (, \{, [\\ P_{i,kind} \cdot d(i-1, open+1, ?) \\ \quad \text{if } kind \text{ is }), \},] \\ P_{i,kind} \cdot d(i-1, open, ?) \end{cases} \tag{3}$$

Note that the above recurrence formula is just for a brief description. There are some more cases to consider to implement.

We also have to record what states were used to find out exactly what symbols were guessed. For example, one might want to define the following function. $b(i, open, kind) :=$ Previous d state that is used to make $d(i, open, kind)$ as large as possible.

Not every segmented part is a symbol. Some of the segmented parts could be a noise in the picture. So we have added the probability of whitespace symbol. We have defined the probability of whitespace symbol so that every black pixel decreases the chance exponentially.

Dot symbols are not trained in our neural network model. We have picked two dot symbols, '.' and '·', namely decimal point and multiplication dot. We first guessed the bottom line and the top line of the expression. The lines were picked so that 95% of the black pixels are above/below the line. Let's say the height is the distance between the bottom line and the top line. If width and height of the segmented parts are smaller than $\frac{1}{6}$ of the height, we consider this part as a dot. Deciding whether the dot is a decimal point or a multiplication dot depends on the y position of the segmented part. Internal division of the height in 1:3 ratio worked fine.

The time complexity of the Dynamic Programming step is $O(n^2\sigma^2)$ where n means the number of segmented part and σ means the number of symbols used. Therefore this step is reasonably fast.

3.2.7 Combining the steps

Finally our algorithm looks like the following **Algorithm 2**

Algorithm 2 The whole algorithm

```
 1: Image ← BINARIZE(Image)
 2: $\theta_{best}$ = Find the best $\theta$ of $D$.
 3: Image ← Rotate(Image, $\theta_{best}$)
 4: Sections ← FloodFill(Image)
 5: Sort Sections by the start $x$ coordinate.
 6: for all A ∈ Sections do
 7:     B ← the next element of A
 8:     if formula (2) is satisfied then
 9:         Remove A and B from Sections
10:         $[c_l, c_r]$ ← Merge A and B
11:         Add $[c_l, c_r]$
12: for all S ∈ Sections do
13:     Calculate $P$ by convolutional neural network.
14: Calculate $d(i, open, kind)$ and $b(i, open, kind)$
15: Backtrace the calculations to reveal the guess.
```

3.3. Environments

We used Python 2.7 as our main programming language. We used Caffe and its python binding for neural network. We used Pillow for basic image manipulation and numpy & scipy for faster calculation. We used Django web framework for the web demo.

4. Experiments

4.1. Single Symbol Recognition

Before we run into our algorithm, neural network prediction accuracy should be tested. As mentioned in preprocess section, we used one-symbol images generated from CROHME as train and test data. Caffe framework was used in neural network training, which automates forward pass, compare output with ground truth, and tuning internal variables with backpropagation. For efficient training, minibatch of 64 images were passed at the same time in each step. Learning rate decreased inversely starting from 0.001, and iterations are done 10000 times.

Trained model shows about 98% accuracy on test set. From confusion matrix(Figure 7), we can find out which symbol our neural network didn't predict well. Here are noticeable cases:

- Symbols similar to a vertical line

  /, 1, [,], {, and } fall into this case. Although some of them even show $> 10\%$ error rates, this is not a big problem. Since / is operator, 1 is number, and others are parentheses, dynamic programming can remove improbable cases with context. Among parentheses, we can reduce erroneous prediction by handling all parentheses as (and), *i.e.*, only one set of opening and closing parentheses.

- Similar symbols, but in different kind

  7 with + and =, π with =, and $\times$ with 2 and e fall into this case. Their error rates are relatively low compared to other cases. In addition, they are of different kinds, like operator and numbers. So dynamic programming can handle this case.

- Similar symbols, the same kind

  This case is the most problematic. 8 with 6, and 9 with 4 and 7 fall into this case. This case can be handled only by fixing neural network, since any interpretation is possible without messing context. We tried to deepen and design our model well to handle this case, but still a few error rates remains.

4.2. Full Expression Recognition

To derive quantitative evaluation, we picked 237 expressions from CROHME, which contain only symbols in Table 2. Handwritten images of 4 different line widths are generated using the same method with train data. The result is described in Table 1.

Line width	Total	Correct	Wrong	% accuracy
1	237	147	90	62.0%
2	237	162	75	68.4%
3	237	154	83	65.0%
5	237	143	94	60.3%

Table 1: Full expression recognition result

Most wrong results came from failure of assumption, that different symbol pixels don't meet. Many people write digits and symbols which overlap each other. Other wrong results mostly caused by reasons described in previous subsection. Regarding these noisy test data, our recognizer performs quite well.

4.3. Web Demo

The web demo is available at `http://chanmin.kim/image-expr/`.

5. Conclusion

5.1. Summary

In summary, we made mathematical expression recognizer which takes input as a handwritten expression image and outputs parsed expression in TEXformat. With train and test data generated from MNIST and CROHME, we trained our neural network to 98% accuracy on single symbol recognition, whose model is modified from LeNet. Given real-life images, we binarize, rotate, segment by flood-fill, and merge symbols by overlapping x-coordinates. Finally, we take out improbable candidates and pick the most probable expression using dynamic programming. We also made web demo to prove performance easily.

5.2. Lessons

We reviewed basic image manipulation such as binarization, warping, convolution, etc. And we learned neural network in detail, especially deep learning using convolutional neural network model. As we use Caffe framework, we are accustomed to designing proper neural network model for given tasks, concepts such as forward pass, back propagation, and various layers. Also, we learned how to improve performance using dynamic programming and some ad-hoc techniques.

5.3. Future Work

We want to categorize single symbols better, by trying deeper models similar to GoogLeNet, AlexNet or ImageNet with better equipments (GPU). We want to support multi-line with better segmentation other than merging by x-coordinates. We want to design better way to detect impossible-to-train symbols such as dot and colons. This will also lead to robust recognition result.

References

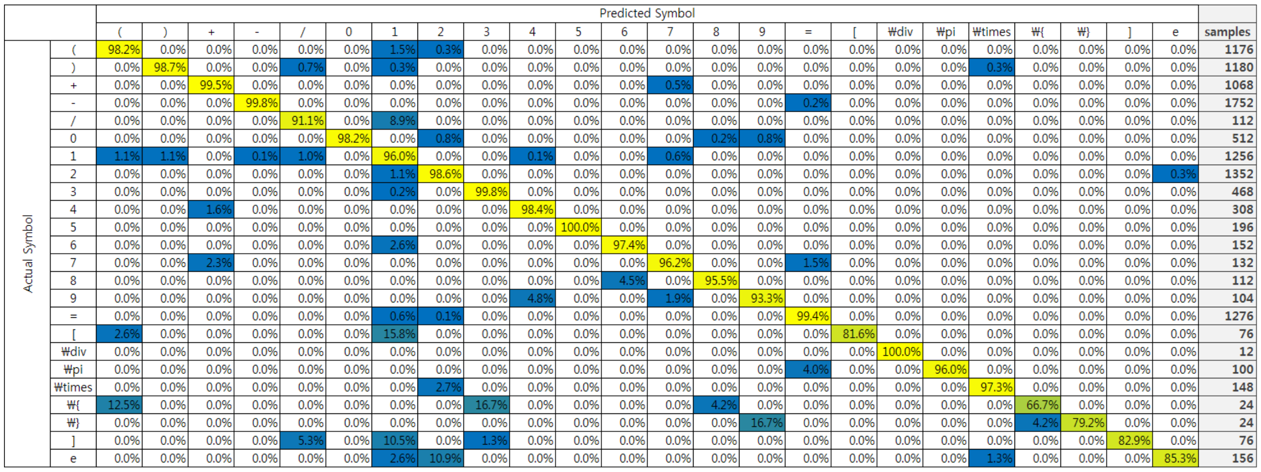
[1] Competition on recognition of online handwritten mathematical expressions. `http://www.isical.ac.in/~crohme/crohme14.html`.

[2] A. Dobke and M. Mann. Ocr for mathematical expressions. `http://www.cs.hmc.edu/~adobke/nn/`, 2012.

[3] Y. LeCun. Lenet, a convolutional network designed for handwritten and machine-printed character recognition. `http://yann.lecun.com/exdb/lenet/`.

[4] M. Nielsen. *Neural Networks and Deep Learning*. 2014.

[5] Y. D. Vitali Sepetnitsky. Ocr of math expressions: Optical character recognition system of mathematical expressions & testing framework. `http://www.cs.bgu.ac.il/~ben-shahar/`



		Predicted Symbol																								
		(	)	+	-	/	0	1	2	3	4	5	6	7	8	9	=	[	₩div	₩pi	₩times	₩{	₩}	]	e	**samples**
Actual Symbol	(	98.2%	0.0%	0.0%	0.0%	0.0%	0.0%	1.5%	0.3%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	**1176**
	)	0.0%	98.7%	0.0%	0.0%	0.7%	0.0%	0.3%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.3%	0.0%	0.0%	0.0%	0.0%	**1180**
	+	0.0%	0.0%	99.5%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.5%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	**1068**
	-	0.0%	0.0%	0.0%	99.8%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.2%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	**1752**
	/	0.0%	0.0%	0.0%	0.0%	91.1%	0.0%	8.9%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	**112**
	0	0.0%	0.0%	0.0%	0.0%	0.0%	98.2%	0.0%	0.8%	0.0%	0.0%	0.0%	0.0%	0.0%	0.2%	0.8%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	**512**
	1	1.1%	1.1%	0.0%	0.1%	1.0%	0.0%	96.0%	0.0%	0.0%	0.1%	0.0%	0.0%	0.6%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	**1256**
	2	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	1.1%	98.6%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.3%	**1352**
	3	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.2%	0.0%	99.8%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	**468**
	4	0.0%	0.0%	1.6%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	98.4%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	**308**
	5	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	100.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	**196**
	6	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	2.6%	0.0%	0.0%	0.0%	0.0%	97.4%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	**152**
	7	0.0%	0.0%	2.3%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	96.2%	0.0%	0.0%	1.5%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	**132**
	8	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	4.5%	0.0%	95.5%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	**112**
	9	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	4.8%	0.0%	0.0%	1.9%	0.0%	93.3%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	**104**
	=	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.6%	0.1%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	99.4%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	**1276**
	[	2.6%	0.0%	0.0%	0.0%	0.0%	0.0%	15.8%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	81.6%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	**76**
	₩div	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	100.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	**12**
	₩pi	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	4.0%	0.0%	0.0%	96.0%	0.0%	0.0%	0.0%	0.0%	0.0%	**100**
	₩times	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	2.7%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	97.3%	0.0%	0.0%	0.0%	0.0%	**148**
	₩{	12.5%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	16.7%	0.0%	0.0%	0.0%	0.0%	4.2%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	66.7%	0.0%	0.0%	0.0%	**24**
	₩}	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	16.7%	0.0%	0.0%	0.0%	0.0%	0.0%	4.2%	79.2%	0.0%	0.0%	**24**
	]	0.0%	0.0%	0.0%	0.0%	5.3%	0.0%	10.5%	0.0%	1.3%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	82.9%	0.0%	**76**
	e	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	2.6%	10.9%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	1.3%	0.0%	0.0%	0.0%	85.3%	**156**

Figure 7: Confusion matrix of trained model. Yellow cells are correct predictions. Blue cells are noticeable error predictions.

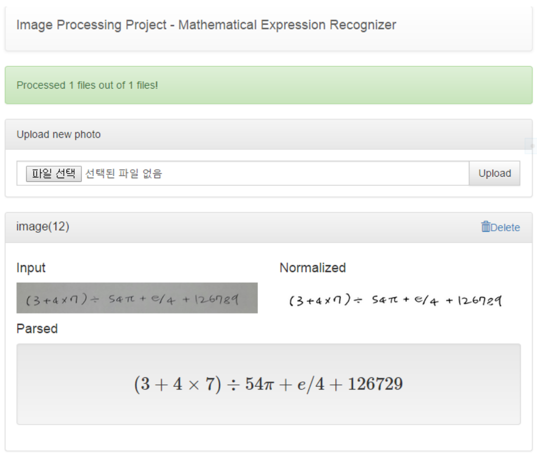


Image Processing Project - Mathematical Expression Recognizer

Processed 1 files out of 1 files!

Upload new photo

$$(3+4\times 7)\div 54\pi + e/4 + 126729$$

Figure 8: Screenshot of web demo. Any user can upload drawn or captured images of handwritten expressions. Then input image, normalized image, and parsed result will appear in a box form. Unwanted results can be removed.

(	)	+	−	/	0	1
2	3	4	5	6	7	8
9	=	[	÷	π	×	{
}	]	e				

Table 2: List of symbols recognized by the neural network

`Teaching/Computational-Vision/StudentProjects/ICBV111/ICBV-2011-1-VitaliSepetnitsky-YakirDahan/index.php`, 2011.

[6] C. J. B. Yann LeCun, Corinna Cortes. the mnist database of handwritten digits. `http://yann.lecun.com/exdb/mnist/`.

6. Appendices