Karlsruhe Institute of Technology

Carnegie Mellon University

On-line recognition of handwritten mathematical symbols

Bachelor's Thesis of

Martin Thoma

At the Department of Informatics
Institute for Anthropomatics and Robotics (IAR)
Karlsruhe Institute of Technology (KIT)
Karlsruhe, Germany

School of Computer Science
Language Technologies Institute (LTI)
Carnegie Mellon University (CMU)
Pittsburgh, United States

Reviewer:	Dr. Stücker
Second reviewer:	?
Advisor:	Prof. Dr. Waibel
Second advisor:	Prof. Dr. Metze

Duration: XX. Monat 2014 – XX. Monat 2014

I declare that I have developed and written the enclosed thesis completely by myself, and have not used sources or means without declaration in the text.

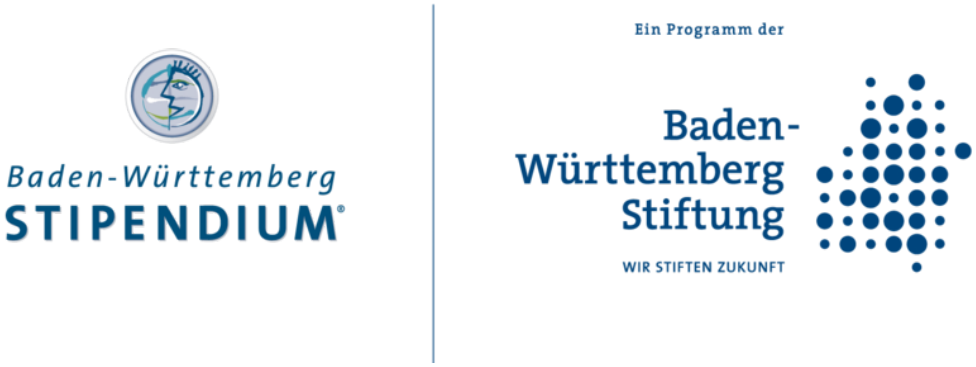
Pittsburgh, DD. MM. 2014

.....

(**Martin Thoma**)

Acknowledgement

TODO



Contents

1. Introduction

Handwriting recognition is the task of finding a proper textual representation given a handwritten symbol or sequence of symbols.

In off-line handwriting recognition, all algorithms have to work on pixel image information of the handwriting. On-line handwriting recognition on the other hand can use the information how symbols were written. This thesis is about on-line handwriting recognition.

1.1. Mathematical notation

I will use the notation $v^{(i)}$ when I want to write about the i-th element of a vector v. Vectors will always be denoted by lowercase latin letters.

Matrices will be denoted by uppercase latin letters.

The number of training examples will be denoted with m, the input vector with x and the output vector with y.

A single training example thus is given by the tuple (x, y).

Parameters that are to be learned will be denoted with θ.

2. Baseline system

A system for symbol recognition was already written and is described in [Kir10]. It uses an algorithm called greedy matching which is similar to Dynamic time warping (DTW).

The Greedy Matching algorithm takes two series of points (A, B) and matches the first points (a, b) of A to B. The distance that point had to be moved is measured. Afterwards, it tries to match the next point a' of A to b as well as the next point b' of B and b' to a. It takes the matching in which the points had to be moved the shortest distance and continues like that.

Pseudocode is on page 23.

3. Data, Preprocessing and Features

The data that was used for all experiments was collected with write-math.com, a website designed solely for this purpose. This website makes use of HTML5 canvas elements. Those elements can be used to track fingers or a mouse coursor touching the canvas, moving and lifting. The origin is at the upper left corner and get bigger to the right (x-coordinate) and to the bottom (y-coordinate).

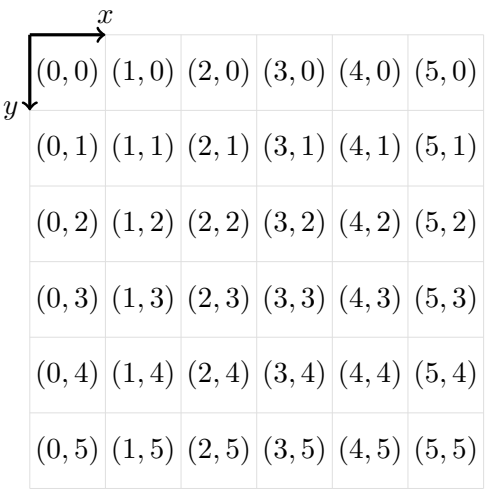


Figure 3.1.: HTML5 canvas plane. Each step is one pixel. There cannot be non-integer coordinates.

The data is stored and shared in JSON format. Each handdrawing is stored as a list of lines, where each line consists of tuples (x, y, t), where x and y are canvas coordinates and t is a timestamp in seconds. This timestamp gives the time in milliseconds from 1970.

The time resolution between points as well as the resolution of the image depends on the device that was used. However, most symbols have time resolution of about 20 ms and are within a bounding box of a 250px × 250px.

3.1. Preprocessing

3.1.1. Scaling and shifting

In many experiments the datapoints were scaled to fit into a unit square while keeping their aspect ratio. Afterwards, the points were shifted to the $(0, 1) \times (0, 1)$ unit square. The algorithm is given in pseudocode on page 24. It was shown in [Kir10, HZK09] that this kind of preprocessing boosts classification accuracy significantly.

3.2. Features

A number of different features have been suggested so far for on-line handwriting recognition. They can be grouped into local features and global features. Local features apply to a given point on the drawing plane and sometimes even only to point on the drawn curve whereas global features apply to a complete line or even the complete image.

3.2.1. Local features

- Curvature[MFW95]
- Speed[HZK09]
- Binary pen pressure[KR98, KRLP99]
- Direction[MFW95, HK06]
- Bitmap-environment[MFW95]

[KR98, KRLP99] suggest that speed is a bad feature, because it "highly inconsistent".

3.2.2. Global features

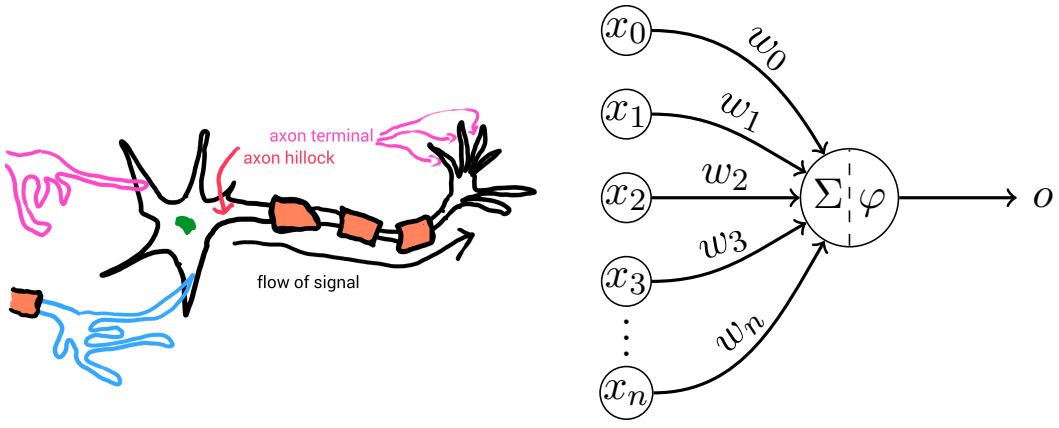
- Re-curvature[HK06] (TODO: Explain (all))
- Center point[HK06]
- Stroke length[HK06]
- Number of strokes[HZK09]
- Pseudo-Zernike Features (TODO: Really global? What is it?)[KC]
- Shadow Code Features[KC]

4. Artificial Neural Nets

Artificial neural networks (ANNs) are models for classification that were inspired by the brain. They consist of artificial neurons and have a lot of different subtypes like Feed Forward Neural Nets.

4.1. Artificial neurons

Artificial neurons are inspired by biological neurons. Signals are send within the cell by charged particles, so called *ions*. But before a biological neuron sends a signal, a threshold charge has to be reached at the axon hillock. This threshold charge is called *action potential*. The action potential can be reached by multiple factors, but the one I want to focus on are charges send by other neurons. Depending on where the other axon terminals are located and how long the distance to the axon hillock is, the signal contributes more or less to reaching the action potential. After that, it simply sends a signal.



(a) Biological Neuron (b) Artificial Neuron

Figure 4.1.: Both neurons receive weighted input, apply a function to that and give output

Artificial neurons are similar. They receive at least one input and give at least one output. Those inputs might get weighted as well as the output.

The neurons apply a function to the sum of all weighted inputs. This function is also called *activation function*.

An artificial neuron using the unit step function (see section 4.7.1) is called a *perceptron*.

The artificial neuron sums all weighted inputs $x_i \cdot w_i$ up and applies its activation function f to it.

4.2. Multilayer Perceptron

Multilayer perceptrons (MLPs) are neural nets whichs neurons are structured in layers. Each layer is fully connected with the next layer, but there are no other connections between neurons.

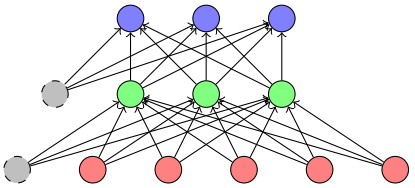


Figure 4.2.: Feedforward artificial neural network

The red neurons in figure 4.2 are input neurons, the green ones are hidden neurons and the blue one is an output node. The gray neurons are bias neurons. Bias neurons have a fixed output of 1.

Usually, you have as many output neurons as you have classes. So in the case of symbol recognition that would be about 1076 neurons.

The number of input neurons is equal to the number of features.

4.3. Notation

Let n_i be the number of neurons in the i-th layer and ℓ be the number of layers of the MLP.

Two neighboring layers of neurons are fully connected and have weights between two layers. This means you can store those weights in form of matrices. So the weights between layer i and layer $i+1$ are

$$W_i = \begin{pmatrix} w_{1,1} & w_{1,2} & \dots & w_{1,n_{i+1}} \\ w_{2,1} & w_{2,2} & \dots & w_{2,n_{i+1}} \\ w_{3,1} & w_{3,2} & \dots & w_{3,n_{i+1}} \\ \vdots & & \ddots & \vdots \\ w_{n_i,1} & w_{n_i,2} & \dots & w_{n_i,n_{i+1}} \end{pmatrix}$$

Let $w_{ij}^{(k)}$ be the value w_{ij} in W_k. So it is the weight of the between neuron i in layer k and neuron j in layer $k+1$.

So $W_i \in \mathbb{R}^{n_i \times n_{i+1}}$ is the matrix denoting the weights between layer i and layer $i+1$.

The unweighted output vector of layer i is denoted by $x_i \in \mathbb{R}^{1 \times n_i}$; the weighted output vector by $\text{net}_i \in \mathbb{R}^{1 \times n_i}$. Instead of x_1 I will write x. The output of the MLP for the input x is denoted by $o_x := x_n$.

In principle each neuron might have a different activation function, but in practice each neuron in one layer has the same activation function. However, activation functions might differ from layer to layer. This is the reason why I denote the activation function of layer i

by φ_i. Although φ_i is defined for single neurons, I will in the following apply it to vectors. In its meant to be applied pointwise.

The activation function is a function

$$\varphi_i : \mathbb{R} \to \mathbb{R}$$

but because of the short-notation it can be applied pointwise to all neurons of layer i it is also a function

$$\varphi_i : \mathbb{R}^{n_i} \to \mathbb{R}^{n_i}$$

4.4. Evaluation

Let $x_1 \in \mathbb{R}^{1 \times n_1}$ be an unweighted output of layer 1. So it's simply the input of our neural net with n_1 features.

Given x_1 one can easily compute the weighted input for layer 2:

$$\begin{array}{ccccc} x_1 & \cdot & W_1 & = & \text{net}_1 \\ \cap & & \cap & & \cap \\ \mathbb{R}^{1 \times n_1} & & \mathbb{R}^{n_1 \times n_2} & & \mathbb{R}^{1 \times n_2} \end{array}$$

After that, you can apply the activation function φ_{i+1} pointwise to net_i.

So the output vector x_3 of a 3-layer (input, hidden, output) neural net can be computed by

$$x_3 = \varphi_3(\varphi_2(x_1 \cdot W_1) \cdot W_2)$$

The output of a general MLP can be computed by

$$\begin{aligned} \Phi(x, n) &: \mathbb{R}^{n_1} \to \mathbb{R}^{n_n} \\ \Phi(x_1, 2) &:= \varphi_2(x_1 \cdot W_1) \\ \Phi(x, n) &:= \varphi_n\left(\Phi(x, n-1) \cdot W_{n-1}\right) \\ \Phi(x, n)^{(p)} &= \varphi_n\left(\sum_{i=1}^{n_{n-1}} \Phi(x, n-1)^{(i)} \cdot w_{ip}^{(n-1)}\right) \end{aligned}$$

It follows: $x_i(x_1) = \Phi(x_1, i)$.

4.5. Supervised Training with Backpropagation

The backpropagation algorithm is a supervised algorithm for training MLPs. This means the trainigset T consists of tuples (x, t_x) where x is input and t_x is the desired output.

To evaluate how good the current MLP is, an error function can be defined:

$$E : \mathbb{R}^{n_1 \times n_2} \times \mathbb{R}^{n_2 \times n_3} \times \cdots \times \mathbb{R}^{n_{\ell-1}, \ell} \to \mathbb{R}_{\geq 0}$$
$$E_T(W) = \frac{1}{2} \sum_{(x,t_x) \in T} \sum_{p=1}^{n_\ell} \left(t_x^{(p)} - o_x^{(p)} \right)^2$$

This function is isomorphic to

$$E : \mathbb{R}^{\sum_{i=2}^{\ell} n_{i-1} \cdot n_i} \to \mathbb{R}_{\geq 0}$$

This error should be minimized. As the error is the sum of non-negative values, we will get a lower error by minimizing the error for a single training example. However, note that those minimizations are not independant. This means, we could get trapped in a local minimum.

The idea is to "go" into the direction in which the error E decreases most. This is the gradient and the process is thus called *gradient descent*.

At this point we need to decide how far we want to go. If we make too big steps in the direction of the gradient, we might overshoot. If we make too small steps, the algorithm will take too long to get to the minimum. As reducing the error is basically learning the number is called learning rate $\eta \in \mathbb{R}_{>0}$.

So the algorithm is

Algorithm 1 Backpropagate

```
function BACKPROPAGATE(T, W)
    while True do
        for all (x, t_x) ∈ T do
            for all node i do
                for all nodes j following i do
                    w_ijk ← w_ijk − η ∂E_{x}/∂w_ijk (W)
```

Computing the partial derivatives $\frac{\partial E_{\{x\}}}{\partial w_{ijk}}$ is not a trivial task. To do that, we have to take a closer look at the error function:

Another approach:

$$E_x(W) = \frac{1}{2}\sum_{p=1}^{n_\ell}\left(t_x^{(p)} - o_x^{(p)}\right)^2 \tag{4.1}$$

$$o_x^{(p)} = \Phi(x,\ell)^{(p)} \tag{4.2}$$

$$= \varphi_\ell\big(\Phi(x,\ell-1)\cdot W_{\ell-1}\big)^{(p)} \tag{4.3}$$

$$= \varphi_\ell\big(\underbrace{\sum_{i=1}^{n_{\ell-1}}\Phi(x,\ell-1)^{(i)}\cdot w_{ip}^{(\ell-1)}}_{\mathrm{net}_{\ell-1}^{(p)}}\big) \tag{4.4}$$

$$\frac{\partial E_x}{\partial w_{ij}^{(k)}} = \frac{\partial E_x}{\partial \mathrm{net}_k^{(j)}}\frac{\partial \mathrm{net}_k^{(j)}}{\partial w_{ij}^{(k)}} \tag{4.5}$$

$$= \frac{\partial E_x}{\partial \mathrm{net}_k^{(j)}}\frac{\sum_{i=1}^{n_k}\Phi(x,k)^{(i)}\cdot w_{ij}^{(k)}}{\partial w_{ij}^{(k)}} \tag{4.6}$$

$$= \frac{\partial E_x}{\partial \mathrm{net}_k^{(j)}}\Phi(x,k)^{(i)} \tag{4.7}$$

Suppose $k = \ell - 1$ (weights to the output layer). Then:

$$\frac{\partial E_x}{\partial w_{ij}^{(\ell-1)}} = \frac{\partial E_x}{\partial \mathrm{net}_{\ell-1}^{(j)}}\Phi(x,\ell-1)^{(i)} \tag{4.8}$$

$$= \frac{\frac{1}{2}\sum_{p=1}^{n_\ell}\left(t_x^{(p)} - o_x^{(p)}\right)^2}{\partial \mathrm{net}_{\ell-1}^{(j)}}\Phi(x,\ell-1)^{(i)} \tag{4.9}$$

$$= \frac{\frac{1}{2}\sum_{p=1}^{n_\ell}\left(t_x^{(p)} - \varphi_\ell(\mathrm{net}_{\ell-1})^{(p)}\right)^2}{\partial \mathrm{net}_{\ell-1}^{(j)}}\Phi(x,\ell-1)^{(i)} \tag{4.10}$$

$$= \left((t_x^{(j)} - \varphi_\ell(\mathrm{net}_{\ell-1}^{(j)}))\cdot(-\varphi_\ell'(\mathrm{net}_{\ell-1}^{(j)}))\right)\Phi(x,\ell-1)^{(i)} \tag{4.11}$$

Suppose $k = \ell - 2$ (last hidden layer). Then:

$$\frac{\partial E_x}{\partial w_{ij}^{(\ell-2)}} = \frac{\frac{1}{2}\sum_{p=1}^{n_\ell}\left(t_x^{(p)} - \varphi_\ell(\mathrm{net}_{\ell-1})^{(p)}\right)^2}{\partial \mathrm{net}_{\ell-2}^{(j)}}\Phi(x,\ell-2)^{(i)} \tag{4.12}$$

$$= \frac{\frac{1}{2}\sum_{p=1}^{n_\ell}\left(t_x^{(p)} - \varphi_\ell(\sum_{i=1}^{n_{\ell-1}}(\varphi_{\ell-1}\big(\varphi_{\ell-2}(\mathrm{net}_{\ell-2}^{(j)})\big))^{(i)}\cdot w_{ip}^{(\ell-1)})^{(p)}\right)^2}{\partial \mathrm{net}_{\ell-2}^{(j)}}\Phi(x,\ell-2)^{(i)} \tag{4.13}$$

$$\tag{4.14}$$

4.6. Parameters

- Number of hidden layers

- Number of neurons per hidden layer
- Epochs
- Momentum
- Weight decay
- Learning rate
- Learning rate decay

4.7. Activation functions

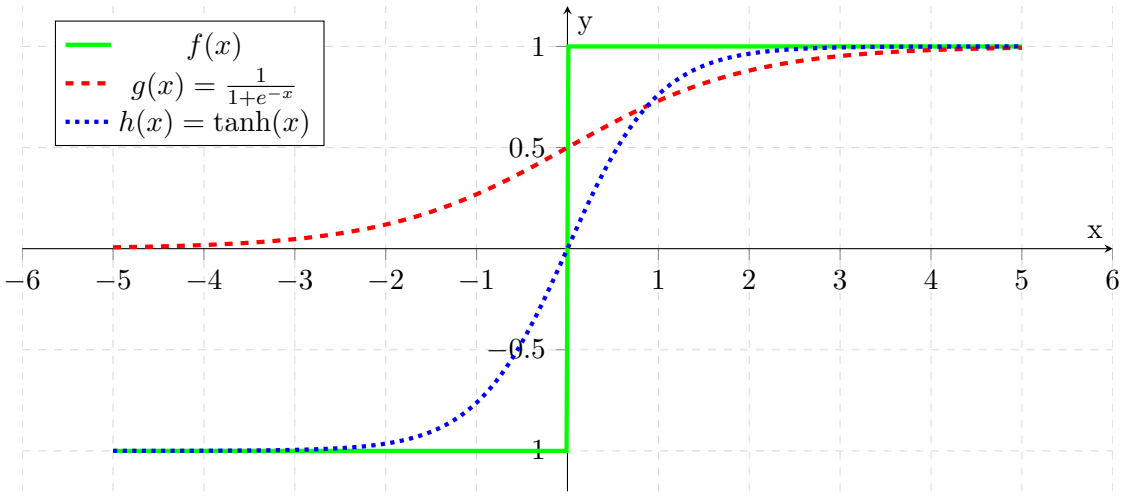


Figure 4.3.: A variation of the sign function f with $f(0) = -1$, the sigmoid function g and the hyperbolic tangend h.

4.7.1. Unit step function

Not so good, because it's not differentiable. Therefore, the backpropagation algorithm cannot be used.

4.7.2. Sigmoid function

Is great because it is infinitely often differentiable.

4.7.3. Hyperbolic tangent

Also differentiable, but gradient descent converges faster (sometimes?)

4.7.4. Softmax

$$\varphi(a_j) = \frac{e^{a_j}}{\sum_k e^{a_k}}$$

5. Formula Recognition

5.1. Nesting Structures

One major issue of formula recognition are nesting structures:

- `\frac{[arbitrary math]}{[arbitrary math]}`
- `\begin{pmatrix}[arbitrary math; structure with & ]\end{pmatrix}`
- `\begin{align}[arbitrary math; structure with & ]\end{align}`
- `\stackrel{[arbitrary math]}{[arbitrary math]}`

Another point that is special about handwritten math recognition compared to natural language text are *decorators*:

- `[symbol]_{[arbitrary math]}`
- `[symbol]^{[arbitrary math]}`

5.2. Segmentation

I assume that writers finish one handwritten symbol before they start the next symbol.

6. Evaluation

A recognition system has two important characteristics: Its recognition accuracy and the time it needs to recognize a new symbol. Recognition accuracy and time are measured with new data which was not seen before.

Tests can be divided into two groups: Tests where some examples of the handwriting of the writer were known at training time and tests were that's not the case.

Known-writer tests are created this way:

Algorithm 2 Creation of k bins of datasets

```
data ← k-dimensional array of Lists
i ← 0
Group labeled datasets by symbol
Filter all symbols that have less than k datasets
for all Group g in datasets do
    for all Dataset (x, t) in g do
        data[i].APPEND((x, t))
        i ← (i + 1)  mod k
```

After that, a k-fold cross validation is run:

I will call result of such a 10-fold cross-validation *classification accuracy*. The first part of the tupel is called *Top-1 accuractiy* and the second one is called *Top-10 accuracy*.

6.1. Baseline system: Greedy matching

The greedy matching algorithm got with scaling and shifting (see page 24) a Top-1 accuracy of 83.11% and a Top-10 accuracy of 97.66%.

Algorithm 3 k-fold cross-validaton

function CROSSVALIDATION(k, grouped dataset d, classificator c)
    correct, wrong $\leftarrow 0, 0$
    c10, w10 $\leftarrow 0, 0$
    for $i \in 0, \ldots, k-1$ **do**
        for $j \in 0, \ldots, k-1$ **do**
            if $i \neq j$ **then**
                c.TRAIN($d[i]$)
        for all $(x, t) \in d[i]$ **do** ▷ List of possible classifications descending by probability
            $L \leftarrow c$.CLASSIFY(x)
            if $L[0] == t$ **then**
                correct $\leftarrow$ correct $+1$
                c10 $\leftarrow$ c10 $+1$
            else if $t \in L$ **then**
                c10 $\leftarrow$ c10 $+1$
                wrong $\leftarrow$ wrong $+1$
            else
                w10 $\leftarrow$ wrong $+1$
                wrong $\leftarrow$ wrong $+1$
    return $(\frac{\text{correct}}{\text{correct+wrong}}, \frac{c10}{c10+w10})$

7. Conclusion

...

Bibliography

[Boa12] E. Board, *Concise Dictionary of Mathematics*, unknown, Ed. V&S Publishers, 2012. [Online]. Available: http://books.google.de/books?id=7OqVdc2LGSUC

[HK06] B. Huang and M.-T. Kechadi, "An HMM-SNN method for online handwriting symbol recognition," in *Image Analysis and Recognition*, ser. Lecture Notes in Computer Science, A. Campilho and M. Kamel, Eds. Springer Berlin Heidelberg, 2006, vol. 4142, pp. 897–905. [Online]. Available: http://dx.doi.org/10.1007/11867661_81

[HZK09] B. Q. Huang, Y. Zhang, and M.-T. Kechadi, "Preprocessing techniques for online handwriting recognition," in *Intelligent Text Categorization and Clustering*, ser. Studies in Computational Intelligence, N. Nedjah, L. de Macedo Mourelle, J. Kacprzyk, F. França, and A. de De Souza, Eds. Springer Berlin Heidelberg, 2009, vol. 164, ch. Preprocessing Techniques for Online Handwriting Recognition, pp. 25–45. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-85644-3_2

[KC] A. Khotanzad and C. Chung, "Hand written digit recognition using combination of neural network classifiers." [Online]. Available: http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=666880

[Kir10] D. Kirsch, "Detexify: Erkennung handgemalter LaTeX-symbole," Diploma thesis, Westfälische Wilhelms-Universität Münster, 10 2010. [Online]. Available: http://danielkirs.ch/thesis.pdf

[KR98] A. Kosmala and G. Rigoll, "Recognition of on-line handwritten formulas," in *In Proceedings of the Sixth International Workshop on Frontiers in Handwriting Recognition*, 1998, pp. 219–228. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.41.9056

[KRLP99] A. Kosmala, G. Rigoll, S. Lavirotte, and L. Pottier, "On-line handwritten formula recognition using hidden markov models and context dependent graph grammars," in *Proceedings of the Fifth Internation Conference on Document Analysis and Recognition (ICDAR)*, 1999, pp. 107–110. [Online]. Available: http://hal.inria.fr/docs/00/56/46/45/PDF/kosmala-rigoll-etal_1999.pdf

[MFW95] S. Manke, M. Finke, and A. Waibel, "The use of dynamic writing information in a connectionist on-line cursive handwriting recognition system," in *Advances in Neural Information Processing Systems 7*, G. Tesauro, D. Touretzky, and T. Leen, Eds. MIT Press, 1995, pp. 1093–1100. [Online]. Available: http://papers.nips.cc/paper/908-the-use-of-dynamic-writing-information-in-a-connectionist-on-line-cursive-handwriting-reco pdf

Glossary

ANN artificial neural network. 7

DTW dynamic time warping. 3

epoch During iterative training of a neural network, an *epoch* is a single pass through the entire training set, followed by testing of the verification set.[Boa12]. 12

learning rate A factor $0 \leq \eta \in \mathbb{R}$ that affects how fast new weights are learned. $\eta = 0$ means that no new data is learned. 12

learning rate decay The learning rate decay $0 < \alpha \leq 1$ is used to adjust the learning rate. After each epoch the learning rate η is updated to $\eta \leftarrow \eta \times \alpha$. 12

MLP multilayer perceptron. 8, 9

Appendix

A. Algorithms

Algorithm 4 Greedy matching as described in [Kir10]

$a \leftarrow$ next from A
$b \leftarrow$ next from B
$d \leftarrow \delta(a, b)$
$a' \leftarrow$ next from A
$b' \leftarrow$ next from B
while points left in $A \wedge$ points left in B **do**
    $l, m, r \leftarrow \delta(a', b), \delta(a', b'), \delta(a, b')$
    $\mu \leftarrow \min\{l, m, r\}$
    $d \leftarrow d + \mu$
    if $l = \mu$ **then**
        $a \leftarrow a'$
        $a' \leftarrow$ next from A
    else if $r = \mu$ **then**
        $b \leftarrow b'$
        $b' \leftarrow$ next from B
    else
        $a \leftarrow a'$
        $b \leftarrow b'$
        $a' \leftarrow$ next from A
        $b' \leftarrow$ next from B
if no points left in A **then**
    for all points p in B **do**
        $d \leftarrow d + \delta(a', p)$
else if no points left in B **then**
    for all points p in A **do**
        $d \leftarrow d + \delta(b', p)$

B. Figures

ein Bild

Figure B.1.: A figure

Algorithm 5 Scale and shift a list of lines to the $(0, 1) \times (0, 1)$ unit square

function SCALE_AND_SHIFT($pointlist$)
  $min_x, min_y = pointlist[0]['x'], pointlist[0]['y']$
  $max_x, max_y = pointlist[0]['x'], pointlist[0]['y']$
  for all *lines* in *pointlist* **do**
    for all p in *lines* **do**
      if $p['x'] < min_x$ **then**
        $min_x \leftarrow p['x']$
      else if $p['x'] > max_x$ **then**
        $max_x \leftarrow p['x']$
      if $p['y'] < min_y$ **then**
        $min_y \leftarrow p['y']$
      else if $p['y'] > max_y$ **then**
        $max_y \leftarrow p['y']$
  $width, height \leftarrow max_x - min_x, max_y - min_y$
  $factor = 1$
  if $width \neq 0$ **then**
    $factor_x = \frac{1}{width}$
  if $height \neq 0$ **then**
    $factor_y = \frac{1}{height}$
  $factor = \min(factor_x, factor_y)$
  $add_x, add_y = 0, 0$
  for all *lines* in *pointlist* **do**
    for all p in *lines* **do**
      $p['x'] \leftarrow (p["x"] - minx) \cdot factor$
      $p['y'] \leftarrow (p["y"] - miny) \cdot factor$
  return *pointlist*