Handwritten Text Recognition

Armand, Christopher Schuster, Mustafa Fuad Rifet Ibrahim

September, 2019

Contents

1 Introduction

Handwritten Text Recognition (HTR) referrs to the task of recognizing handwritten text from sources such as photographs, scanned documents, etc. and transcribing that to digital text (Figure 1). Traditionally, the task of recognizing handwritten text would involve several different steps and methods such as segmenting and extracting individual characters from the lines (or pages) of scanned text and using intelligently crafted features to recognize those characters. Crafting the features was an arduous task and extracting individual characters is a challenging image recognition task in and of itself. Since 2009, where neural networks developed by Jürgen Schmidhuber and his research group at the Swiss AI Lab IDSIA won multiple international handwritten text recognition competitions[R], neural networks became more and more prominent in the scene of handwritten text recognition. These neural networks consisted of a combination of two different neural network architectures, namely Convolutional Neural Networks (CNN) and Recurrent Neural Networks (RNN). In this project our goal was to implement and expand a model for handwritten text recognition that uses precisely that combination of architectures.

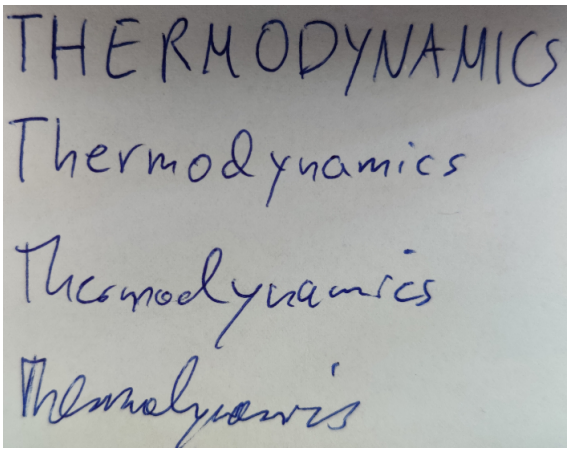


Figure 1: Example of different levels of difficulty for handwritten text recognition. The upper example shows clear separation and writing of the individual characters while the last one shows connected and overlapping characters that are far away from their standard form.

2 Theory

This section assumes basic knowledge about the concept of neural networks and the mathematics behind it. Further specific knowledge about relevant architectures will be explained.

2.1 Convolutional Neural Networks

Convolutional Neural Networks (CNNs) differ from the basic Fully Connected Neural Network (FCNN) in the fact that they are specifically built for processing images. In a FCNN images don't scale well, as the amount of parameters increases rapidly with increasing image size. A CNN handles this problem by having only local spatial connections between neurons of two layers. To understand how this is accomplished, two basic parts ,that are frequently used in CNN architectures, are essential, namely the convolutional layers and the pooling layers.

Convolutional Layers

The convolutional layer consists of filters (or kernels) that have a certain spatial extension, i.e. along the width and height of the image, and a full extension along the depth of the input. The depth of the input describes the number of channels e.g. 3 color channels in a RGB image. During the forward pass, these filters are slidden across the image and a dot product is computed between the entries of the filter and the input values of the image (Figure 2). The result is a 2D activation map (one per filter) and when these maps are stacked along the depth dimension, they represent the output volume of the convolutional layer. The width and height dimension of the output volume are determined by three other hyperparameters: kernel/filter size, stride and zero padding.
The kernel size is the width and height of the filter window described above. The stride referrs to the amount of pixels the filter is moved along the spatial dimensions of the input at each step during the computation of the forward pass. Zero padding or more precisely the size of the zero padding referrs to the size of the optional zero border that can be added to the input if needed. All these hyperparameters determine the width and height of the output volume according to this formula:

$$\frac{D - F + 2P}{S} + 1$$

, where D is the width or height, F is the kernel size along the width or height, P is the amount of padding and S referrs to the stride length. Figure 2 also shows the concept of local connectivity mentioned above. Every entry in the output volume, or in other words every neuron, is connected to only a fraction of the input entries (or neurons).

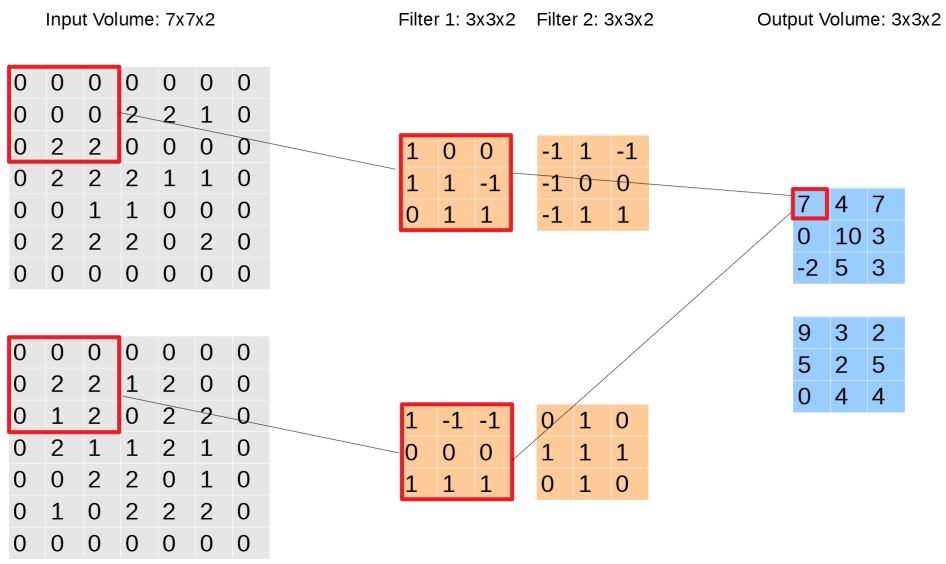


Figure 2: The forward pass of the convolutional layer[R]. On the left is the input with the size 7x7x3. It is padded with a one pixel thick border of zeros. In this graph, the depth dimension or number of channels/filters is depicted by stacking the matrices above one another. In the middle are the two filters applied to the input, in this case we have a filter size of 3x3. This gives us an output of size 3x3x2. The red squares and the black lines show the the entries of the input and the filters applied to those entries to calculate the respective output entry.

Pooling Layers

The pooling layers have the task to downsample the input and thus to decrease the amount of parameters (weights). The mechanics of this layer are somewhat similar to the convolutional layer. It also involves a window that is slidden across the input but in this case the max value among the input values that fall within the spatial extension of the window at the respective calculation step is taken as the output (Figure 3). This is called max pooling. There are also other functions (e.g. average function) but max pooling is the one used most often. All the previously mentioned hyperparameters apply here as well except for the zero padding. In addition to that, the depth dimension remains unchanged as the pooling operation is done on every slice along the depth separately and the resulting downsampled slices are again stacked along the depth dimension to form the output.

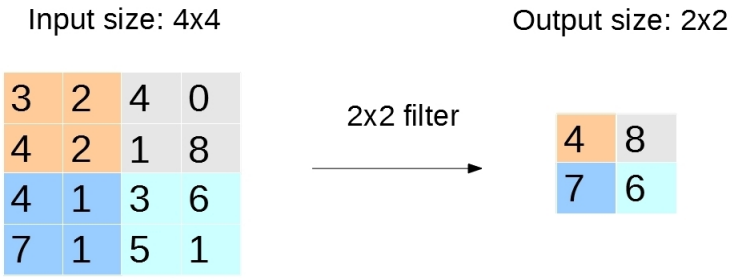


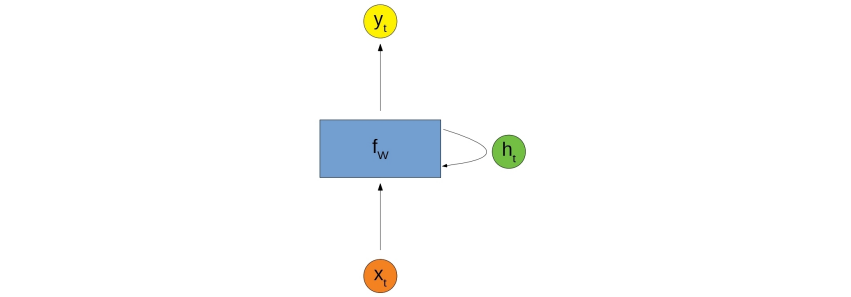
Figure 3: The pooling operation. On the left is one depth slice of an input with width and height equal to 4. The pooling filter in this case has size 2x2 and stride 2 which effectively means that 75% of the image information is discarded.

2.2 Recurrent Neural Networks

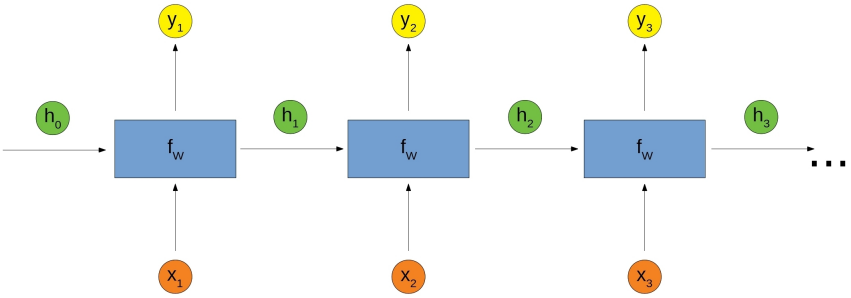
Recurrent Neural Networks (RNN) are a type of neural network architecture that allow for information of a variably sized input sequence to persist, thereby enabling a neural network to solve problems like guessing the most probable last word of a sentence.

Vanilla RNN

The standard RNN can be thought of as having a cell that takes in an input x_t and computes an output based on this input as well as the so called "hidden state" of the cell. This new output is then taken as the new hidden state and is combined with a new input x_{t+1} to generate the next output or hidden state. The result of the calculation can also be used to apply another function to produce an output y_t like for example some kind of class score at all time steps or only certain time steps. During all these calculation steps the same weights are used (Figure 4).



(a) Looping visualization of RNN



(b) Unrolled visualization of RNN

Figure 4: Vanilla RNN visualization. f denotes the non-linear function with parameters W, h is the hidden state, x is the input and y is the output. The upper image is a visualization expressing the feedback nature of the RNN. The lower image elucidates the sequential nature of the RNN by arranging the RNN at different time steps next to itself.

Using a formula the basic operation can be roughly written as:

$$h_t = f_W(h_{t-1}, x_t)$$
$$y_t = f_y(h_t)$$

, where the symbols have the same meaning as in Figure 4 and f_y is some function applied to get the desired output. This means that information of past sequence parts persists in the form of the hidden state. The fact that the parameters W remain unchanged for all sequence elements means that we have parameter sharing. Just like CNNs introduce a form of parameter sharing effective for image inputs, RNNs achieve the same with sequential input. This allows the RNN to not be constrained by fixed input or output size. It is also possible to increase the amount of input information by connecting two layers in opposite direction. This is called Bidirectional RNN (BRNN) and can be thought of as the combination of two RNNs, one getting the sequence input in the normal order and the other getting the input sequence in reverse order. This enables the network to produce an output based on future and past sequence information, e.g. in the case of handwritten text recognition it enables

the prediction based on the letters located before and after the current letter.

Long-Short-Term-Memory Networks

Vanilla RNNs run into problems when it comes to long input sequences where the correctness of the output hinges on the ability to connect pieces of information that are many time steps apart. Long-Short-Term-Memory Networks (LSTM) are a RNN variant that are particularly good at handling these long term dependencies. They differ from standard RNNs in the basic module or cell that accepts the input and hidden state and produces the next hidden state or output (Figure 5). While the standard RNNs have one non-linear function in the module, LSTMs have a more complex structure that involves different functions.

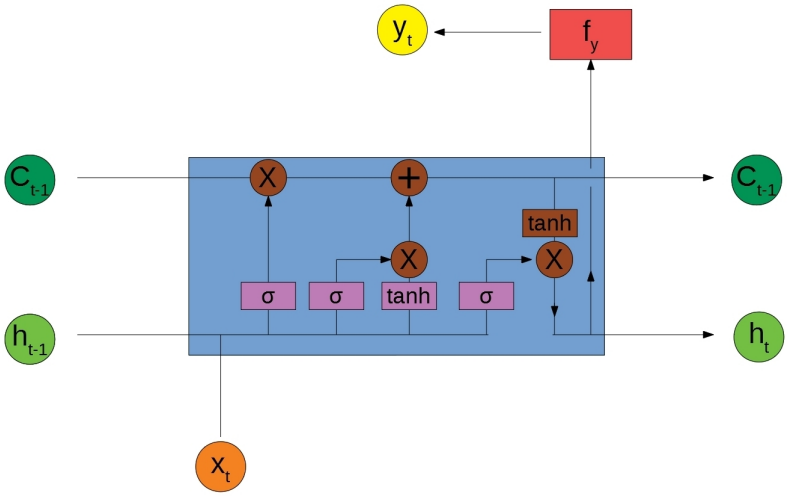


Figure 5: LSTM Visualization. Hidden state, input and output are shown as before. The purple rectangles are neural network layers that use a sigmoid or tanh function. The brown rectangle and brown circles are point wise operations of multiplication 'X', addition '+' or a tanh function. The dark green circles are the cell state and the red square is the function used to get a desired output like for example a class score.

As can be seen from Figure 5 the LSTM has an addition state called the cell state. This cell state flows across the upper stream and can be augmented through information passing through different gates. The first one on the left is the forget gate. Here the information from the previous hidden state and the current input are used to decide which parts of the cell state to forget (values closer to 0) and which parts to keep (values closer to 1):

$$f_t = \sigma(W_f \cdot [h_{t-1}, x_t])$$

This gets combined with the old cell state through point wise multiplication. Next is the input gate. Through a similar mechanism new information that is supposed to be added gets chosen here but before applying the point wise multiplication to the input, the input is pushed through a tanh layer to produce candidate values.

$$i_t = \sigma(W_i \cdot [h_{t-1}, x_t])$$
$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t])$$

This new information is combined with the old cell state through point wise addition. So to summarize the operations, the augmentation to the cell state looks like this:

$$C_t = f_t \cdot C_{t-1} + i_t \cdot \tilde{C}_t$$

, where $\cdot$ denotes point wise multiplication and $+$ denotes point wise addition. The last part of the LSTM module is the output. Similar to the input gate the information is filtered. In this case the new cell state is filtered to form the new hidden state and output at that time step:

$$o_t = \sigma(W_o[h_{t-1}, x_t])$$
$$h_t = o_t \cdot \tanh(C_t)$$

The tanh here is used to squish the values of the new cell state into the range from -1 to 1 before filtering. There are many variants of this basic LSTM architecture that bring certain advantages in specfic applications but a discussion of those augmentations is beyond the scope of this text.

2.3 Connectionist Temporal Classification

Connectionist Temporal Classification (CTC) is the supervised learning algorithm used to train neural networks in sequence problems such as handwritten text recognition. The straightforward way of doing supervised learning in this area would be to manually label every position in the input image to specify how much exactly the letters of the ground truth text extend in the input image. Then, the RNN would output its guess and the loss would punish for each character that was guessed wrong. This however is infeasible with large datasets. CTC provides a way to stop worrying about the alignment of input and output. It introduces a blank symbol that stands for no character recognized at this position to account for gaps between characters or words (or spoken words in speech recognition) and to provide a way to allow properly decode repeat characters, and a way to map many different valid alignments to the ground truth. For a given output of the RNN simply merging repeat characters and then removing any blank symbol (in that order!) provides the actual output that can be compared against the ground truth. Taking this into account, the loss is then calculated by computing the probability for all valid alignments given a ground truth text and the output of the RNN which contains the character probabilities per time step. This is done by multiplying the character probabilities for each

alignment and summing over all such valid alignments. Taking the negative log of this sum gives the loss. The model parameters are tuned to minimize that loss or in other words to maximize the probability the network assigns to the right answer. To illustrate, take the following situation:

- ground truth text: 'bad'
- characters: a,...,z
- time steps(maximum word length): t_0,..t_5
- blank symbol: '-'

The following example of a RNN output matrix is given:

	a	b	c	d	...
t_0	0.02	0.8	0.03	0.1	
t_1	0.9	0.01	0.001	0.03	
t_2	0.1	0.08	0.002	0.78	
$\vdots$					

Table 1: RNN output matrix. Only the first four letters and first three time steps are shown. The blank symbol '-' is also part of the output and has a probability for each time step. In this case the word of the input image could have been more towards the left border of the image since this example output shows high probability for the respective letters of the word appearing right at the start.

With that we can calculate the probability for each valid alignment by multiplying the individual character probabilities. Valid alignments, i.e. alignments that would map to the word 'bad' if we merged repeat characters and removed blank symbols, are for example:

- b - - a - d
- b b b b a d
- b a a a a d
- b a d d d d
- b a d - - -

If we let A be the set of all such valid alignments then the loss is calculated by taking the negative log of the sum over all the alignments probabilities:

$$\text{loss} = -\log\left(\sum_{A} p(A)\right)$$

Inference

To test the model, we need a method to extract a likely output text given the RNN output. The most intuitive way would be to simply take the alignment with the highest probability, which we call 'Best Path Decoding' in our project. This method works well if the most probable alignment has a significantly higher probability than the rest. A problem can arise when this is not the case. For example if we have only three characters: 'a', 'b' and the blank symbol '-' and we have two time steps t_0 and t_1, then given the following RNN output matrix:

	a	b	-
t_0	0.2	0.0	0.8
t_1	0.4	0.0	0.6

Table 2: RNN Output Matrix.

Best Path Decoding would yield: '-' as the output because the respective alignment '- -' $0.8 \cdot 0.6 = 0.48$ has the highest probability out of all alignments. However, even though all alignments corresponding to 'a' are less probable than the one corresponding to '-', summing over them yields a higher total probability for the output text to be 'a'.

$$\begin{aligned} \text{'a-'}&: \ 0.2 \cdot 0.6 = 0.12 \\ \text{'-a'}&: \ 0.8 \cdot 0.4 = 0.32 \\ \text{'aa'}&: \ 0.2 \cdot 0.4 = 0.08 \\ \text{total probability} &= 0.52 \end{aligned}$$

Therefore Best Path Decoding might miss some outputs that are more likely.
A different algorithm called Beam Search Decoding accounts for cases like these. It works by generating possible alignment candidates, expanding them by all characters and calculating the resulting probabilities of these new candidates. In each expansion step it discards all but the most probable ones, e.g. the top 10 candidates will be used in the next expansion step. So if we again have only the three characters 'a', 'b' and blank symbol '-' the first few steps of the beam search algorithm could like the tree in Figure 6.

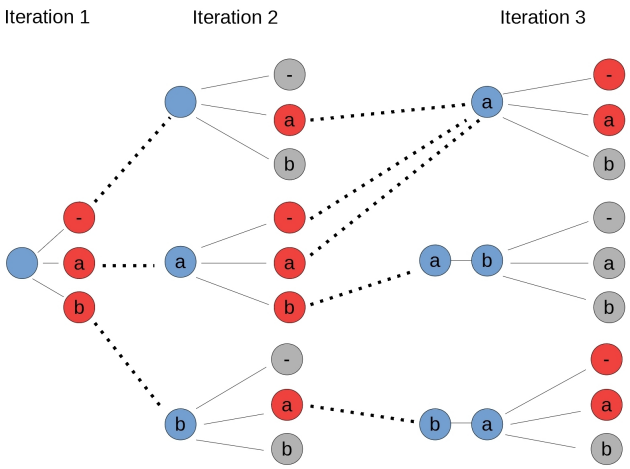


Figure 6: Beam Search visualization with a tree graph. The red circles are the candidate extensions and the blue circles are the current guesses. Empty circles stand for empty strings. The dotted lines stand for the output to which the current guess plus the candidate extension map to. In each step we take the 3 most probable extended words. When multiple alignments map to one output (more than 3 circles like in Iteration 2 and 3) the probabilities are summed and summarized under one new guess.

3 Methods

3.1 Metrics

stuff

3.2 The Model

We based our initial approach on the Handwritten Text Recognition (HTR) system of Harald Scheidl[R]. Our plan was to implement that model with Pytorch as our base line performance model. Using this basic model we then implemented augmentations on different levels of the model to enhance the word recognition accuracy. The base model consists of two main parts, namely a Convolutional Neural Network (CNN) and a Recurrent Neural Network (RNN) with a Connectionist Temporal Classification (CTC) output (Table 3).

Type	Description	Output Size
Input	grey-scale image	1x32x128
CONV+BN+POOL	kernel 5x5, pool 2x2	32x16x64
CONV+BN+POOL	kernel 5x5, pool 2x2	64x8x32
CONV+BN+POOL	kernel 3x3, pool 2x1	128x4x32
CONV+BN+POOL	kernel 3x3, pool 2x1	128x2x32
CONV+BN+POOL	kernel 3x3, pool 2x1	256x1x32
LSTM	bidirectional, 2 layers, hidden size 256	32x512
CONV	kernel 1x1	80x32
CTC		$\leqslant 32$

Table 3: Neural Network architecture of the base line model.

CNN

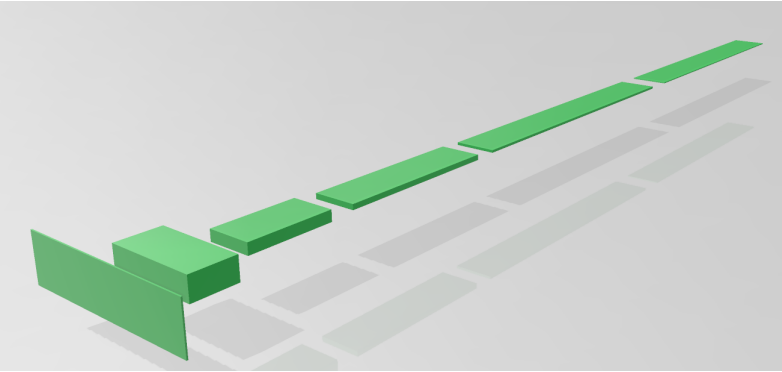


Figure 7: CNN part of the model. The first cube on the left represents the input of size 1x32x128 (channels x height x width) and the last cube on the right represents the output of size 256x1x32.

The CNN is responsible for extracting relevant features from the input images(Figure 7). All input images have the dimension 32x128 (height x width) and the CNN reduces the width to 32 and the height to 1 while extracting 256 features. In other words, the matrix at the end describes each pixel thin slice along the condensed width of the image with 256 feature values. That creates an input for the RNN that consists of 256 features and 32 time steps.

RNN

The RNN takes in the condensed sequence information and uses the extracted features in order to predict the probabilities of the possible characters in the sequence. The RNN used in this case is a Long-Term-Short-Term (LSTM) network that is bidirectional. The output of the RNN consists of a 32x256 map that contains scores for the 256 features for each time step. After the LSTM we used a single CNN layer to map the output of the LSTM to the 80 characters (english alphabet, numbers and a few additional characters plus the CTC blank symbol) that we used, resulting in a 32x80 matrix. We used the CTC algorithm to train the model, as described in the theory section, and decoded the outputs during test time with both Best Path Decoding and Beam Search Decoding. Figure 8 shows how the input matrix is transformed during the feed forward pass.

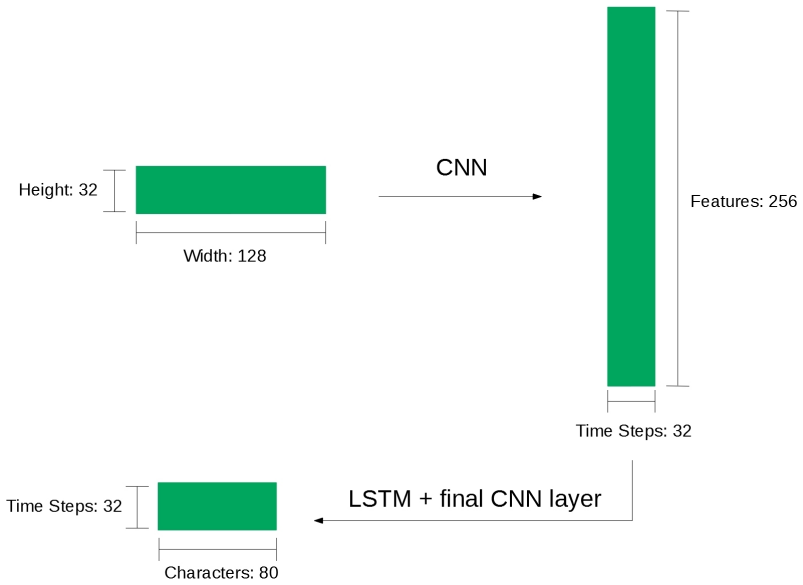


Figure 8: Transformation through the model layers. In the top left is the input image and in the bottom left is the RNN output matrix after it was projected onto 80 characters.

4 Results

5 Discussion

6 References

[R] https://www.kurzweilai.net/how-bio-inspired-deep-learning-keeps-winning-competitions
[R] https://github.com/githubharald/SimpleHTR