Build a Handwritten Text Recognition System using TensorFlow

A minimalistic neural network implementation which can be trained on the CPU

Harald Scheidl Jun 15, 2018 · 7 min read

Offline Handwritten Text Recognition (HTR) systems transcribe text contained in scanned images into digital text, an example is shown in Fig. 1. We will build a Neural Network (NN) which is trained on word-images from the IAM dataset. As the input layer (and therefore also all the other layers) can be kept small for word-images, NN-training

Fig. 1: Image of word (taken from IAM) and its transcription into digital text.

Get code and data

1. You need Python 3, TensorFlow 1.3, numpy and OpenCV installed
2. Get the implementation from GitHub: either take the code version this article is based on, or take the newest code version if you can accept some inconsistencies between article and code
3. Further instructions (how to get the IAM dataset, command line parameters, …) can be found in the README

Model Overview

We use a NN for our task. It consists of convolutional NN (CNN) layers, recurrent NN (RNN) layers and a final Connectionist Temporal Classification (CTC) layer. Fig. 2 shows an overview of our HTR system.

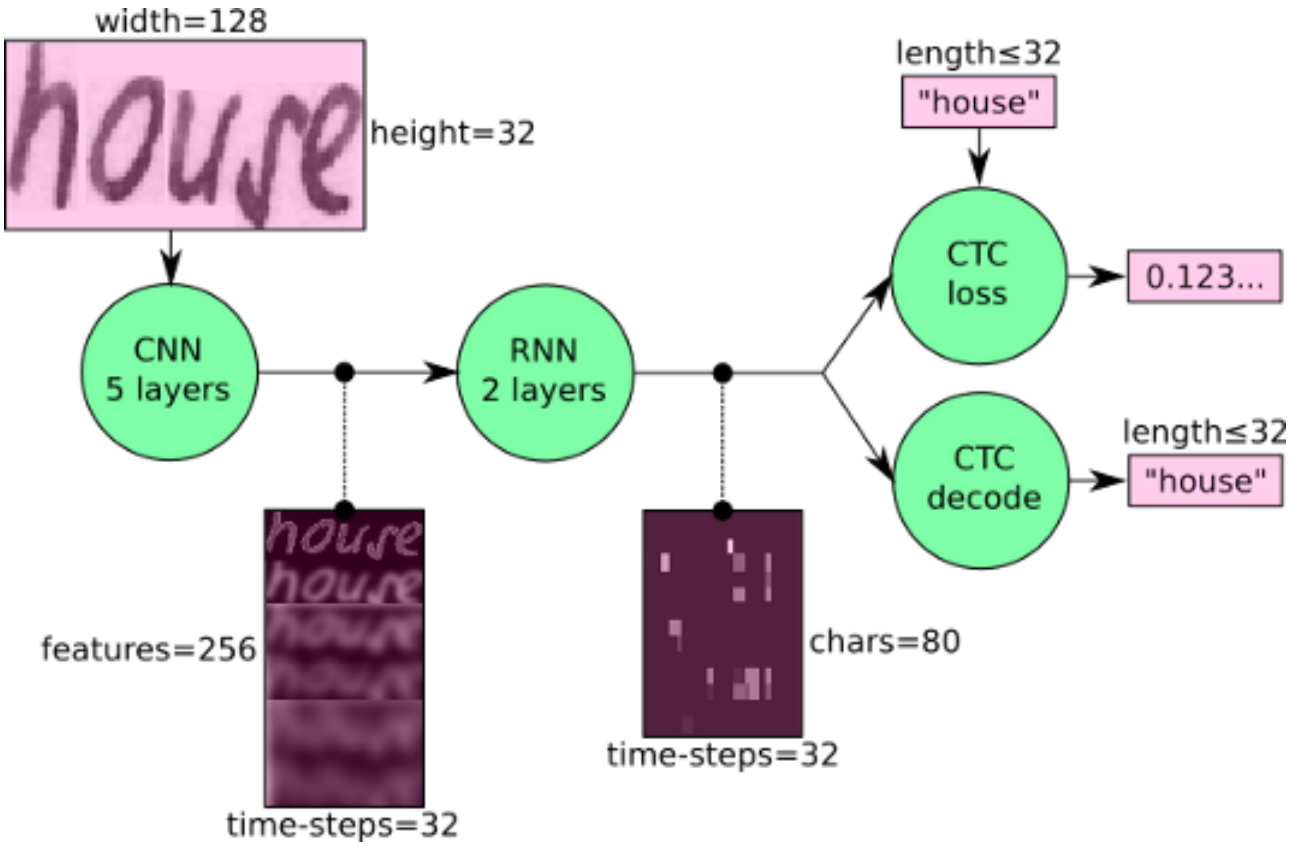

Fig. 2: Overview of the NN operations (green) and the data flow through the NN (pink).

between 0 and L. As you can see, the text is recognized on character-level, therefore words or texts not contained in the training data can be recognized too (as long as the individual characters get correctly classified).

$$NN: \underset{W \times H}{M} \rightarrow (c_1, c_2, \ldots, \underset{0 \leq n \leq L}{c_n})$$

Eq. 1: The NN written as a mathematical function which maps an image M to a character sequence (c1, c2, ...).

Operations

CNN: the input image is fed into the CNN layers. These layers are trained to extract relevant features from the image. Each layer consists of three operation. First, the convolution operation, which applies a filter kernel of size 5×5 in the first two layers and 3×3 in the last three layers to the input. Then, the non-linear RELU function is applied. Finally, a pooling layer summarizes image regions and outputs a downsized version of the input. While the image height is downsized by 2 in each layer, feature maps (channels) are added, so that the output feature map (or sequence) has a size of 32×256.

RNN: the feature sequence contains 256 features per time-step, the RNN propagates relevant information through this sequence. The popular Long Short-Term Memory (LSTM) implementation of RNNs is used, as it is able to propagate information through longer distances and provides more robust training-characteristics than vanilla RNN. The RNN output sequence is mapped to a matrix of size 32×80. The IAM dataset consists of 79 different characters, further one additional character is needed for the CTC operation (CTC blank label), therefore there are 80 entries for each of the 32 time-steps.

CTC: while training the NN, the CTC is given the RNN output matrix and the ground truth text and it computes the **loss value**. While inferring, the CTC is only given the matrix and it decodes it into the **final text**. Both the ground truth text and the recognized text can be at most 32 characters long.

Data

Input: it is a gray-value image of size 128×32. Usually, the images from the dataset do not have exactly this size, therefore we resize it (without distortion) until it either has a width of 128 or a height of 32. Then, we copy the image into a (white) target image of size 128×32. This process is shown in Fig. 3. Finally, we normalize the gray-values of the image which simplifies the task for the NN. Data augmentation can easily be

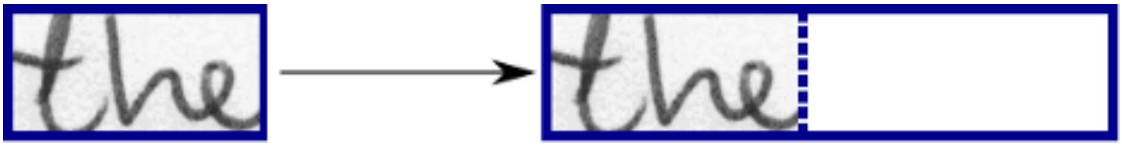

Fig. 3: Left: an image from the dataset with an arbitrary size. It is scaled to fit the target image of size 128×32, the empty part of the target image is filled with white color.

CNN output: Fig. 4 shows the output of the CNN layers which is a sequence of length 32. Each entry contains 256 features. Of course, these features are further processed by the RNN layers, however, some features already show a high correlation with certain high-level properties of the input image: there are features which have a high correlation with characters (e.g. “e”), or with duplicate characters (e.g. “tt”), or with character-properties such as loops (as contained in handwritten “l”s or “e”s).

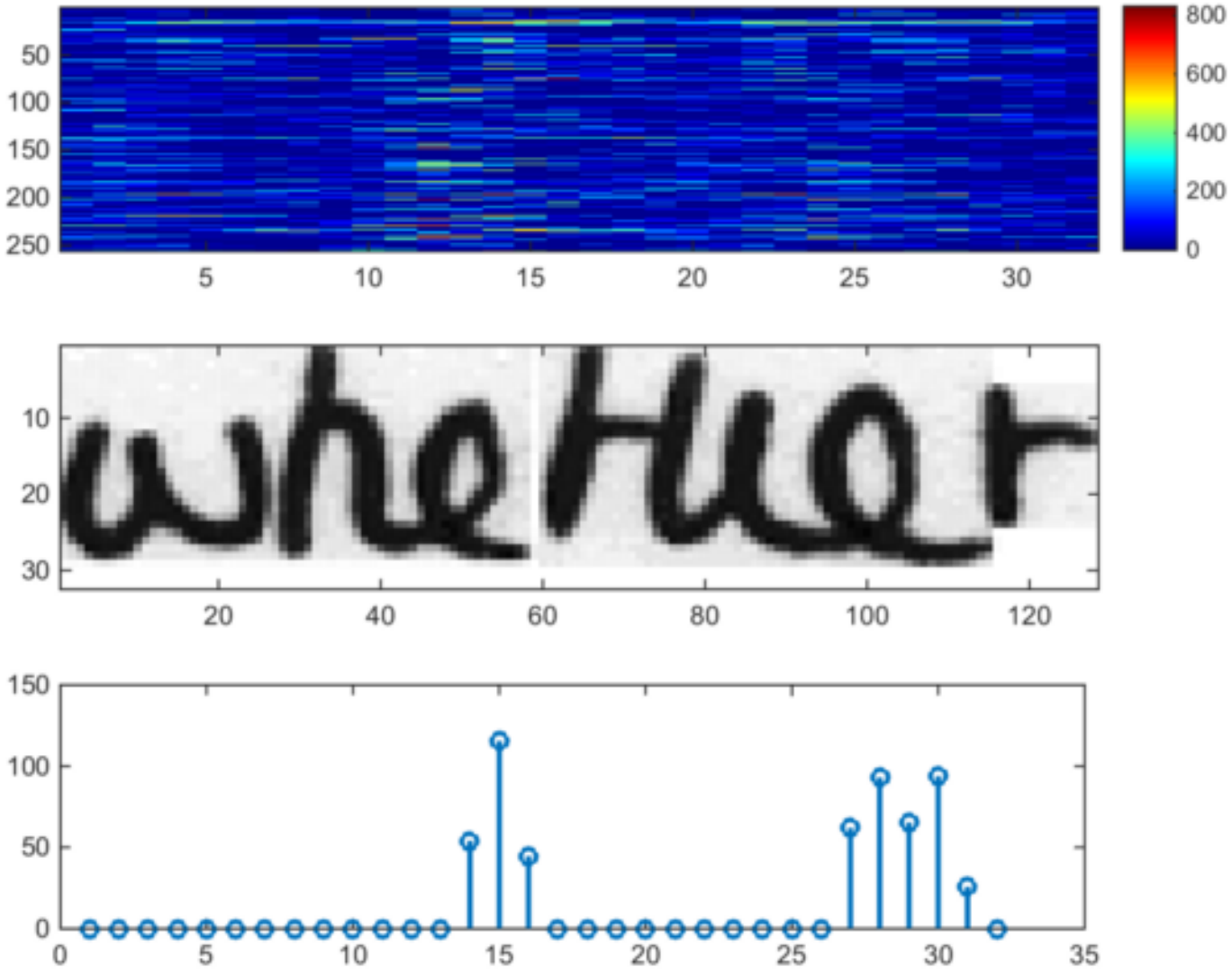

Fig. 4: Top: 256 feature per time-step are computed by the CNN layers. Middle: input image. Bottom: plot of the 32nd feature, which has a high correlation with the occurrence of the character “e” in the image.

RNN output: Fig. 5 shows a visualization of the RNN output matrix for an image containing the text “little”. The matrix shown in the top-most graph contains the scores

!”#&’()*+,-./0123456789:;?
ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz”. It can be seen that most of the time, the characters are predicted exactly at the position they appear in the image (e.g. compare the position of the “i” in the image and in the graph). Only the last character “e” is not aligned. But this is OK, as the CTC operation is segmentation-free and does not care about absolute positions. From the bottom-most graph showing the scores for the characters “l”, “i”, “t”, “e” and the CTC blank label, the text can easily be decoded: we just take the most probable character from each time-step, this forms the so called best path, then we throw away repeated characters and finally all blanks: “l---ii--t-t--l-…-e” → “l---i--t-t--l-…-e” → “little”.

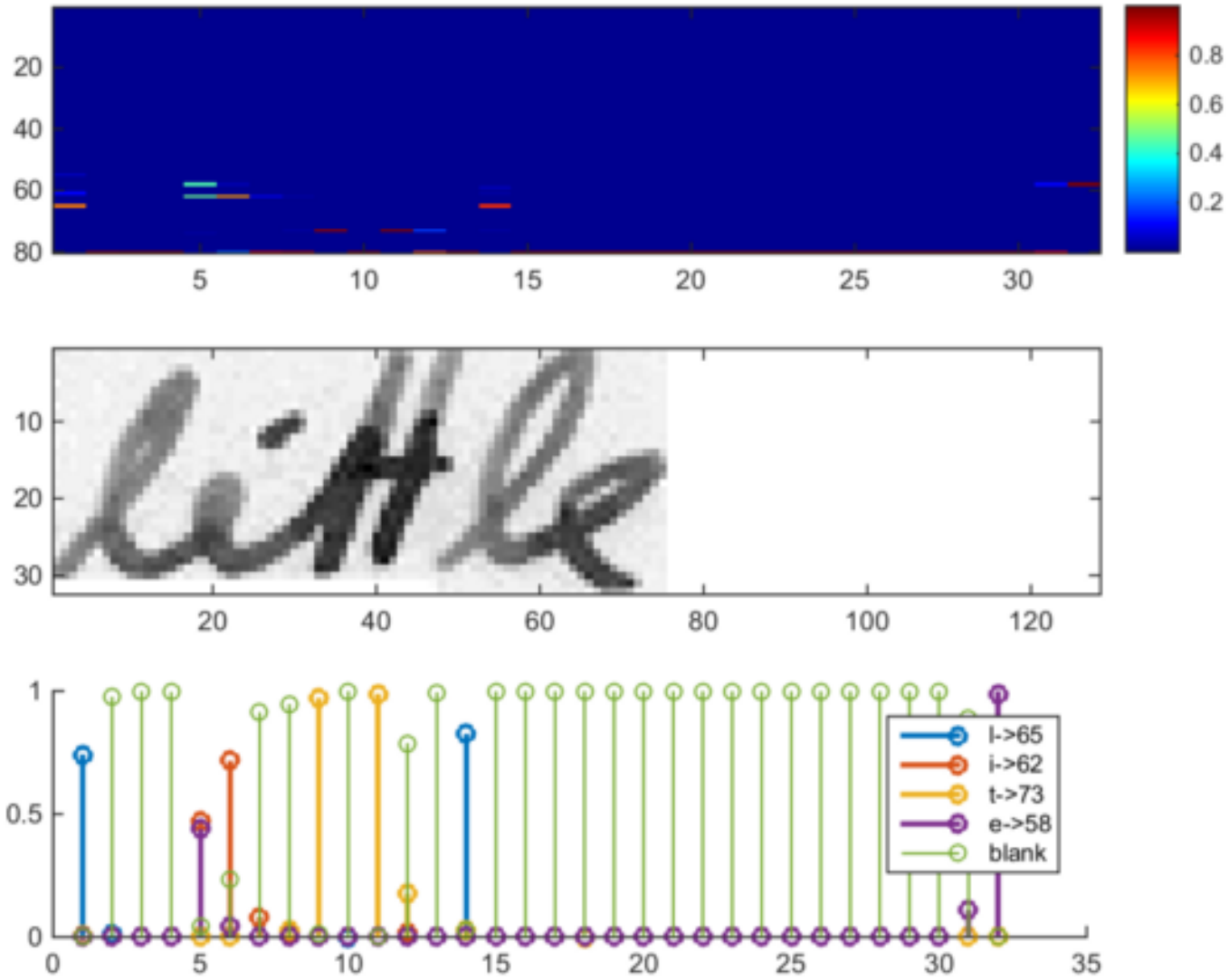

Fig. 5: Top: output matrix of the RNN layers. Middle: input image. Bottom: Probabilities for the characters “l”, “i”, “t”, “e” and the CTC blank label.

Implementation using TF

The implementation consists of 4 modules:

1. SamplePreprocessor.py: prepares the images from the IAM dataset for the NN

3. Model.py: creates the model as described above, loads and saves models, manages the TF sessions and provides an interface for training and inference
4. main.py: puts all previously mentioned modules together

We only look at Model.py, as the other source files are concerned with basic file IO (DataLoader.py) and image processing (SamplePreprocessor.py).

CNN

For each CNN layer, create a kernel of size k×k to be used in the convolution operation.

```
kernel = tf.Variable(tf.truncated_normal([k, k, chIn, chOut], stddev=0.1))
conv = tf.nn.conv2d(inputTensor, kernel, padding='SAME', strides=(1, 1, 1, 1))
```

HTR_1.py hosted with ♡ by GitHub view raw

Then, feed the result of the convolution into the RELU operation and then again to the pooling layer with size px×py and step-size sx×sy.

```
relu = tf.nn.relu(conv)
pool = tf.nn.max_pool(relu, (1, px, py, 1), (1, sx, sy, 1), 'VALID')
```

HTR_2.py hosted with ♡ by GitHub view raw

These steps are repeated for all layers in a for-loop.

RNN

Create and stack two RNN layers with 256 units each.

```
cells = [tf.contrib.rnn.LSTMCell(num_units=256, state_is_tuple=True) for _ in range(2)]
stacked = tf.contrib.rnn.MultiRNNCell(cells, state_is_tuple=True)
```

HTR_3.py hosted with ♡ by GitHub view raw

Then, create a bidirectional RNN from it, such that the input sequence is traversed from front to back and the other way round. As a result, we get two output sequences fw and bw of size 32×256, which we later concatenate along the feature-axis to form a sequence of size 32×512. Finally, it is mapped to the output sequence (or matrix) of size 32×80 which is fed into the CTC layer.

```
((fw, bw),_) = tf.nn.bidirectional_dynamic_rnn(cell_fw=stacked, cell_bw=stacked, inputs=inputTens
```

CTC

For loss calculation, we feed both the ground truth text and the matrix to the operation. The ground truth text is encoded as a sparse tensor. The length of the input sequences must be passed to both CTC operations.

```
gtTexts = tf.SparseTensor(tf.placeholder(tf.int64, shape=[None, 2]), tf.placeholder(tf.int32, [No
seqLen = tf.placeholder(tf.int32, [None])
```

HTR_5.py hosted with ♡ by GitHub view raw

We now have all the input data to create the loss operation and the decoding operation.

```
loss = tf.nn.ctc_loss(labels=gtTexts, inputs=inputTensor, sequence_length=seqLen, ctc_merge_repea
decoder = tf.nn.ctc_greedy_decoder(inputs=inputTensor, sequence_length=seqLen)
```

HTR_6.py hosted with ♡ by GitHub view raw

Training

The mean of the loss values of the batch elements is used to train the NN: it is fed into an optimizer such as RMSProp.

```
optimizer = tf.train.RMSPropOptimizer(0.001).minimize(loss)
```

HTR_7.py hosted with ♡ by GitHub view raw

Improving the model

In case you want to feed complete text-lines as shown in Fig. 6 instead of word-images, you have to increase the input size of the NN.

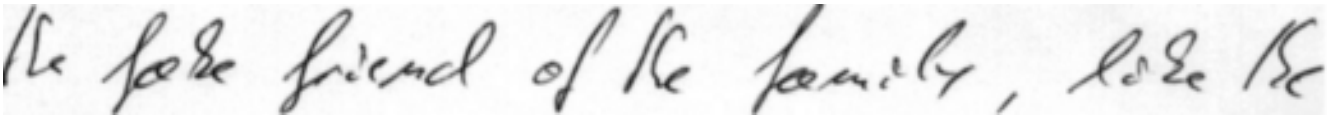

Fig. 6: A complete text-line can be fed into the NN if its input size is increased (image taken from IAM).

If you want to improve the recognition accuracy, you can follow one of these hints:

- Data augmentation: increase dataset-size by applying further (random) transformations to the input images
- Remove cursive writing style in the input images (see DeslantImg)

- Add more CNN layers
- Replace LSTM by 2D-LSTM
- Decoder: use token passing or word beam search decoding (see CTCWordBeamSearch) to constrain the output to dictionary words
- Text correction: if the recognized word is not contained in a dictionary, search for the most similar one

Conclusion

We discussed a NN which is able to recognize text in images. The NN consists of 5 CNN and 2 RNN layers and outputs a character-probability matrix. This matrix is either used for CTC loss calculation or for CTC decoding. An implementation using TF is provided and some important parts of the code were presented. Finally, hints to improve the recognition accuracy were given.

FAQ

There were some questions regarding the presented model:

1. How to recognize text in your samples/dataset?
2. How to recognize text in lines/sentences?
3. How to compute a confidence score for the recognized text?

I discuss them in the FAQ article.

References and further reading

Source code and data can be downloaded from:

- Source code of the presented NN
- IAM dataset

These articles discuss certain aspects of text recognition in more detail:

- FAQ
- What a text recognition system actually sees
- Introduction to CTC
- Vanilla beam search decoding

A more in-depth presentation can be found in these publications:

- Thesis on handwritten text recognition in historical documents
- Word beam search decoding
- Convolutional Recurrent Neural Network (CRNN)
- Recognize text on page-level

Machine Learning Ocr Deep Learning Recurrent Neural Network Image Processing