TALLINN UNIVERSITY OF TECHNOLOGY
Faculty of Information Technology
Department of Software Science

Henri Vasserman, 040683 IAPB

DEEP LEARNING BASED ANALYSIS OF THE SENTENCE WRITING TEST TO SUPPORT DIAGNOSTICS OF PARKINSON'S DISEASE

Bachelor's Thesis

Supervisor: Sven Nõmm, PhD

Tallinn 2019

TALLINNA TEHNIKAÜLIKOOL
Infotehnoloogia teaduskond
Tarkvarateaduse instituut

Henri Vasserman, 040683 IAPB

SÜGAVÕPPEPÕHINE ANALÜÜS PARKINSONI TÕVE DIAGNOOSI LAUSE KIRJUTAMISE TESTI TOEKS

Bakalaureusetöö

Juhendaja: Sven Nõmm, PhD

Tallinn 2019

Author’s declaration of originality

I hereby certify that I am the sole author of this thesis. All the used materials, references to the literature and the work of others have been referred to. This thesis has not been presented for examination anywhere else.

Author: Henri Vasserman

01.05.2019

// Abstract

This bachelor's thesis is about trying to create a method for facilitating automatic diagnosing of Parkinson's disease from a handwriting test.

The first step is to create a handwriting recognizer that can detect individual letters from previously collected handwriting samples. For this end I wrote a sample labeling tool in JavaScript and a deep recurrent neural network recognition engine using TensorFlow and Keras.

The resulting letter labels I use to extract several different motion mass features for each detected letter. The features I use then to train several different classifiers to see if it is possible to detect if a sample is part of the Parkinson's disease group or the healthy control group.

Using cross-validation, I show that it is possible to achieve an accuracy up to 75% with using just a few features in a simple classifier such as a decision tree.

This thesis is written in English and is 37 pages long, including 7 chapters, 11 figures and 4 tables.

Annotatsioon

Sügavõppepõhine analüüs Parkinsoni tõve diagnoosi lause kirjutamise testi toeks

Selle bakalaureusetöö teema on ehitada meetod, mis toetaks Parkinsoni tõve automaatset diagnoosimist käekirjatestiga.

Esimene samm on luua käekirja tuvastaja, mis suudaks leida üksikuid tähti kogutud eksemplaridelt. Selle jaoks kirjutasin tähtede lahterdamistööriista JavaScripti abil ja sügava rekurentse närvivõrgu peal töötava mudeli, mis on ehitatud Keras ja TensorFlow peal.

Tuvastatud tähtedele arvutan välja erinevad liikumismassi parameetrid, mida kasutan erinevate klassifikaatorite treenimiseks, et näha, kas on võimalik tuvastada, kas antud käekirjaeksemplar kuulub Parkinsoni tõve või tervesse kontrollgruppi.

Kasutades ristvalideerimist, saan ma näidata, et on võimalik saavutada isegi kuni 75% täpsuse, kui kasutada vaid mõnda üksikut parameetrit sellises lihtsas klassifikaatoris nagu otsustuspuu.

Lõputöö on kirjutatud inglise keeles ning sisaldab teksti 37 leheküljel, 7 peatükki, 11 joonist ja 4 tabelit.

List of terms

PD	Parkinson's disease
HC	healthy control
RNN	recurrent neural network [1]
LSTM	long-short term memory: an advanced form of RNN [2]
JSON	JavaScript object notation: a standard data interchange format
Python	Scripting language that is widely used in data science and machine learning
Numpy	Multidimensional computation library for Python [3]
Pandas	Data analysis library for Python [4]
Tensorflow	Automatic differentiation framework by Google [5]
Keras	Neural network and machine learning library that runs on Tensorflow [6]
C#	C-like language by Microsoft
API	Application programming interface

Contents

1 Introduction 11
1.1 Parkinson's disease 11
1.2 Previous research 11
1.3 Data source. 12
1.4 Problem statement 12
2 Implementation 13
2.1 Data description 13
2.2 Workflow 13
2.3 Directory structure 14
2.4 Data format 15
3 Handwriting recognition 16
3.1 Data preprocessing 16
3.1.1 Coordinates 16
3.1.2 Vertical location 16
3.1.3 Relative coordinates 18
3.1.4 Training targets 18
3.2 Neural network design 18
3.2.1 Recurrent neural networks 19
3.2.2 LSTM 19
3.2.3 Dense layer. 20
3.2.4 Softmax 20
3.2.5 Categorical cross-entropy 21
3.2.6 Adam optimizer 21
3.2.7 Training 21
3.3 Label detection 22
4 Data manager 23
4.1 Letter classification 23
4.2 Result view. 23
4.3 Confidence 24
5 Extracting kinematic features 26
6 Validation 28
6.1 Scikit-learn classifiers 29
6.1.1 Cross-validation 29
6.1.2 Decision trees. 29
6.1.3 Multi layer perceptron 30
6.1.4 Stochastic gradient descent 30
6.1.5 Nearest neighbor 30

7 Conclusions 31
7.1 Shortcomings and future improvements 31
References 34
Appendix 1 – Readme file 35

List of Figures

1	Previous studies	12
2	Example of data	13
3	Workflow of the data	14
4	Example of similarities of motion whether the pen is lifted or not	17
5	Preprocessed letter example	18
6	Recognition network design	19
7	Hyperbolic tangent and the logistic sigmoid function	20
8	Label reading state machine	22
9	Data manager web view	25
10	Results view	25
11	Decision tree with two features.	30

List of Tables

1 Model weights 21
2 Classification labels 23
3 Top 10 features 28
4 Validation 29

1 Introduction

The goal of this thesis work is to create a system that can isolate characters from handwriting samples and classify them into the healthy control group (HC) or Parkinson's disease (PD) group. The handwriting samples have been collected during research with PD patients.

1.1 Parkinson's disease

Parkinson's disease affects 1% of the population over 60 years of age in industrialized nations. Current diagnosis rely on several clinical symptoms yet there are no universally accepted set of criteria for diagnosis. There is also no known cure or any one known cause [7].

Patients exhibit symptoms such as bradykinesia (slow movement), tremor and rigidity in motion. This affects manual dexterity, including handwriting; it becomes less legible and smaller on the page, called micrographia [8].

1.2 Previous research

The main research methods that are relevant have been analyzing various kinematic features from when the subject draws on a graphics tablet or smart device equipped with a digitizer, dating back to the 1990-s by Marquardt [9] and Eichhorn [10].

Studies in 2016 by Kozhenkina [11] and Nõmm in [12], [13] showed results by analyzing kinematic features in the Luria's [14] alternating series which look like zigzags or waves. Drotár [15], used similar kinematic analysis to analyze handwriting and spiral drawing.

Mašarov [16] in 2017 used a clock-drawing test collected using an iPad Pro with the Apple pencil. He used MNIST database-trained [17] type of classifiers to detect numbers of the clock and to extract features from different measurements of the drawing as well as motion mass parameters.

Bardõš [18] with Nõmm [19] in 2018 analyzed Luria's alternating series for *anomaly detection* and of the same kind of kinematic features as before in the other studies. They also used cross-validation for their results, up to even top 90 features combined.

Toodo [20] in the same year used the sentence writing test from the same iPad app as [16], [18] to extract features of individual words using Google's Inception v-3 [21] deep convolutional network, analyzing several kinematic motion mass and character measurement features using several classifiers.

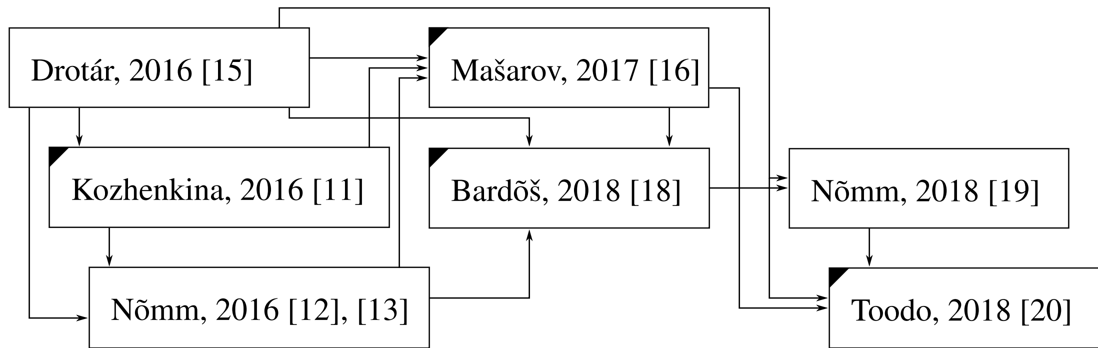


Figure 1: Previous studies. Master's theses marked in corner.

1.3 Data source

The information system and app to collect the testing data from patients was started during normal coursework and finished in the thesis of [16].

In addition to the alternating series and other geometric drawing tests there is a handwriting test. This test was used for [20], as well. The subjects are asked to write the sentence, in Estonian: "*Kui Arno isaga koolimajja jõudis, olid tunnid juba alanud.*"

This sentence is the opening line from a classic Estonian novel and was chosen because of its high cultural collective memory of the subjects in the study. It was hoped that this would give the test a more common baseline.

1.4 Problem statement

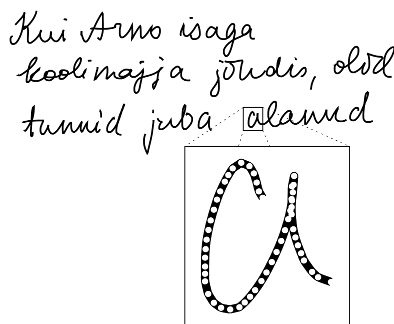
To create a software system that uses machine learning to learn how to read letter labels from handwriting samples and extracting individual kinematic features from the detected letters to use them for classification of the subject into two groups: HC or PD; verifying the accuracy of the classification using different subsets of data for training and testing.

2 Implementation

The software was written in various languages. Most of the machine learning and analysis in Python, the small web service in C# using .NET Core, the user interface in JavaScript using Vue.js.

2.1 Data description

The data input is in the format of JSON files [16] that record each stroke of the handwriting as a series of points (up to 240 per second). Each point contains six floating point values $\{x, y, t, a, l, p\}$, where x, y are the Cartesian coordinates (x increases from right to left and y from top to bottom), a, l are the azimuth and longitude of the pen, p is the pressure applied to the pen, t is a timestamp in the iOS format that represents seconds from the beginning of 2001 [22].



x	y	t	a	l	p
595.2969	391.7930	572198502.634109	0.695058	0.934383	0.333333
594.5000	390.4688	572198502.684318	0.695058	0.934383	0.153108
594.0313	388.6055	572198502.721226	0.695058	0.934383	0.191631
593.4375	386.6836	572198502.721343	0.695058	0.934383	0.164456
592.5000	385.1602	572198502.721384	0.695058	0.934383	0.291993
590.7813	384.4336	572198502.738869	0.695058	0.934383	0.348021
588.9844	385.6289	572198502.756594	0.695058	0.934383	0.411446
588.0625	387.0234	572198502.756924	0.695058	0.934383	0.424279
587.3281	388.4063	572198502.756983	0.695058	0.934383	0.449387
586.3438	390.4688	572198502.772691	0.695058	0.934383	0.460670
585.6719	392.1914	572198502.772790	0.695058	0.934383	0.471061
585.0156	394.3125	572198502.772817	0.695058	0.934383	0.462629
584.3438	396.5742	572198502.772842	0.695058	0.934383	0.470390
583.8750	398.5664	572198502.789255	0.695058	0.934383	0.483450
⋮	⋮	⋮	⋮	⋮	⋮
609.6875	409.5117	572198503.019896	0.695058	0.934383	0.422518

Figure 2: Example of data. Left: a sample of the written test with the letter *a* highlighted. Right: the data points of the highlighted *a* letter. Some rows have been omitted for brevity.

2.2 Workflow

The user interface (chapter 4) is used for defining character ranges in the handwriting samples, which is required for the training, and also to configure if a sample is used for the recognition 'training' or 'testing' phase. It also has options for defining the sample group of HC or PD.

After the sentences have their letters marked using the UI, there point sequences are preprocessed using Python and Numpy [3] before the training step.

The deep learning machine training for the character recognition is built using Tensorflow [5] version 2 alpha. Even if it is in alpha, it is quite stable and also is much easier to use and develop for.

The training script loads all the samples that are in the 'training' set. The model from the training is saved to disk in a portable JSON format in the subdirectory `model`.

After the model is trained, there is a Python script that can load the model and run the recognition step on all the sample files that are in the 'test' set.

The classification of the letters is done with scikit-learn [23], where the data is processed with Pandas [4] and Numpy [3] for analysis.

The analysis first finds what are the best features and then uses several classifiers that scikit-learn includes, using 2 .. 10 of the best features in a combination, selecting the 'testing' samples using k-fold cross-validation, with $k = 6$.

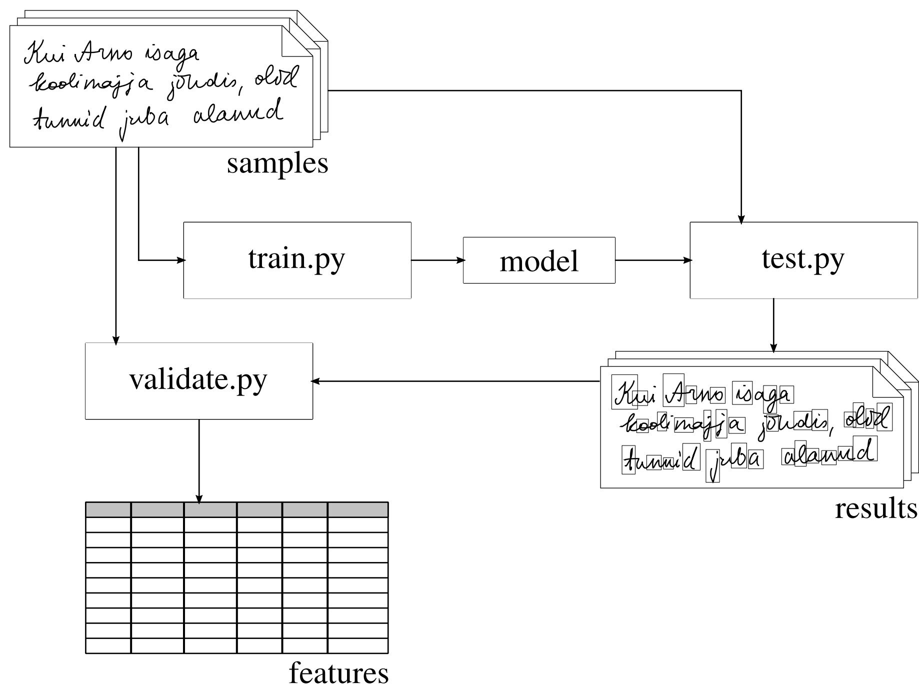


Figure 3: Workflow of the data, from sentences to results to features

2.3 Directory structure

There are several files and directories in the root of the software project:

`sentences` – Input files in JSON format.
`metadata` – Metadata for each input file.
`results` – Output from the handwriting recognizer, contains letter labels for each input file.
`model` – Saved recognition model.
`DataMgr` – The data manager web (see ch. 4).
`train.py` – Handwriting recognizer training script (ch. 3). results saved to `model`.
`test.py` – Handwriting inference, reads `model` and `sentences` and writes into `results`.
`validate.py` – Extracts kinematic features from `results` and checks the accuracy of the classification.

Some of these scripts contain helper functions:

`data.py` – Preprocesses data for input to `train.py` and `test.py`.
`features.py` – Extracts feature data into pandas structures [4] for `validate.py`.
`model.py` – A Python class encapsulating the trained model loading from the `model` directory.

For a more in-depth understanding of the software setup, see appendix 1.

2.4 Data format

For saving the data, a JSON-based format is used in the intermediate steps. I chose this because of the the original incoming data was also in JSON format. The Tensorflow model has support for saving and loading in JSON format but the weights file is usually not in this format just because of the size overhead of saving thousands or millions of floating point numbers in text format.

However, even with over 20,000 numbers for the weights, the saved model is less than a megabyte in size and can be significantly reduced with compression.

3 Handwriting recognition

One possible recognition engine that I tried to use was Microsoft's Tablet PC APIs. it did give some results as a text string; however, it does not support Estonian and it was also not very accurate, while also there seems to be no way to extract individual letter ranges.

Using recurrent neural networks (RNNs) was inspired by Karpathy [24] where he introduces and analyzes their ability to generate sequences of text.

The handwriting recognition was inspired by Graves who analyzes using RNNs to generate various sequences such as text [25] and handwriting (recorded points as opposed to static images). In [26] he also analyzes different RNNs' abilities to classify sequences, such as speech and handwriting.

3.1 Data preprocessing

The samples are highly variable in the amount of points pre time unit and also the the difference of time units per point (writing speed). A simple processing step that I have used in this work is just a distance filter from the last point.

This proves sufficient but a future improvement would be to represent the strokes as a two-dimensional parametric curve $[x, y] = \mathbf{f}(t)$, this way the data can be re-sampled into any number of data points. Recent work by Google [27] has used least-squares estimation to reduce the points into Bézier curves.

3.1.1 Coordinates

If the input stroke S consists of points of $\mathbf{p}_i = [p_{x_i}\ p_{y_i}]$ coordinates, then the output set of points S' is a subset where each point has a distance from the last point that is greater than the distance D:

$$\begin{aligned} \forall \mathbf{p}_i \in \mathbf{S}, &\quad \text{where } i \in [0, N) \\ \mathbf{S}' \ni \mathbf{p}_i, &\quad \text{if } \operatorname{dist}(\mathbf{p}_i, \mathbf{p}_{i-1}) > D \end{aligned}$$

Where the function dist gives the distance between two points in two dimensions, as derived from the Pythagorean theorem:

$$\operatorname{dist}(\mathbf{a}, \mathbf{b}) = \sqrt{(a_x - b_x)^2 + (a_y - b_y)^2} \tag{1}$$

3.1.2 Vertical location

This processing gives a list of points of two dimensions, x, y, but not the z-axis, or rather the information of the fact that the pen is touching the surface.

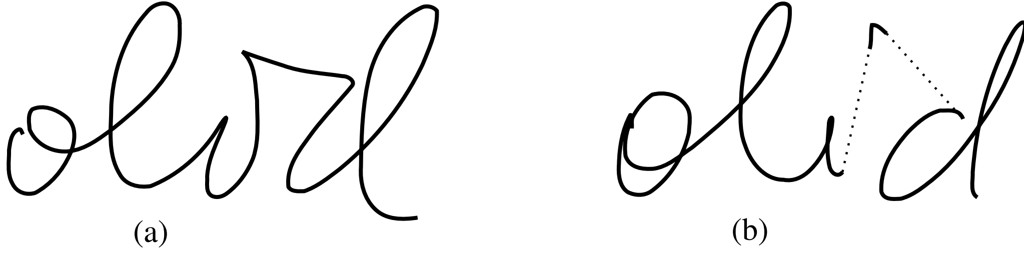


Figure 4: Example of similarities of motion whether the pen is lifted or not: (a) the dot in *i* is written connected to the main main body of the letter and also the next letter, (b) the dot is separated.

I had an insight after looking at the images of the samples where, for example the letter *i*, the dot is drawn after the main body of the letter. However, in some cases, the writer does not lift the pen before the dot, leaving them connected (figure 4).

Since the x, y-coordinate movement between them is almost the same, I thought that this would make it easier for the RNN to learn that the letters are, in fact, the same. Doing this would also maybe solve some difficulties with discontinuous sections, such as in [25].

I also added a third dimension d for 'down', as in, 'is the pen down'.

Since the input data does not contain the pen movement while the pen is lifted off the drawing surface, I have synthesized the points as a simple linear interpolation between the ending of one stroke and the next.

If the stroke $\mathbf{S}_i$ is the next stroke after $\mathbf{S}_{i-1}$, I get the points $\mathbf{p}_1$ and $\mathbf{p}_2$:

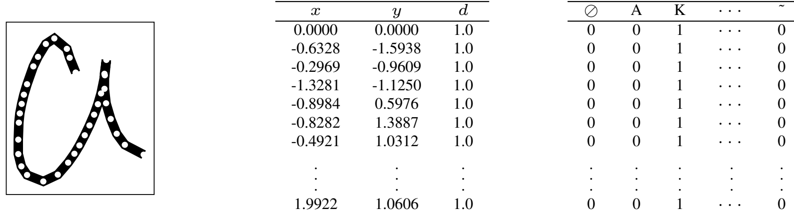
$$\begin{aligned}\mathbf{p}_1 &= \mathbf{p}_{\mathbf{S}_{i-1}[N-1]}\\ \mathbf{p}_2 &= \mathbf{p}_{\mathbf{S}_i[0]}\\ \Delta\mathbf{p} &= \mathbf{p}_2 - \mathbf{p}_1\end{aligned}$$

Where $\Delta\mathbf{p}$ is the difference of coordinates between the stroke end and the next's beginning. I calculate the count of points to generate c using (1):

$$\begin{aligned}c &= \left\lfloor \frac{\mathrm{dist}(\mathbf{p}_2, \mathbf{p}_1)}{D} \right\rfloor\\ G_k &= \left[\frac{\Delta p_x}{c}, \frac{\Delta p_y}{c}, 0\right]\end{aligned}$$

Where $\mathbf{G}$ is the generated point set and $k \in [0, c)$.

The filtered coordinates $\mathbf{S}'$ and generated sequences $\mathbf{G}$ are then concatenated into a single sequence $\mathbf{A}$, where $\mathbf{A} = \{\mathbf{S}'_0, \mathbf{G}_1, \cdots, \mathbf{G}_N, \mathbf{S}'_N\}$. Each point $\mathbf{p}$ in $\mathbf{A}$ contains the x, y, d-coordinates from above.



x	y	d
0.0000	0.0000	1.0
-0.6328	-1.5938	1.0
-0.2969	-0.9609	1.0
-1.3281	-1.1250	1.0
-0.8984	0.5976	1.0
-0.8282	1.3887	1.0
-0.4921	1.0312	1.0
⋮	⋮	⋮
1.9922	1.0606	1.0

∅	A	K	···	˜
0	0	1	···	0
0	0	1	···	0
0	0	1	···	0
0	0	1	···	0
0	0	1	···	0
0	0	1	···	0
0	0	1	···	0
⋮	⋮	⋮	⋮	⋮
0	0	1	···	0

Figure 5: Preprocessed letter example. This is the same letter as in figure 2 but the preprocessing has removed some points. The middle table is the point data that is fed into the neural net. The right table is the table that is used for training targets (3.1.4) that correspond to the labels in table 2

3.1.3 Relative coordinates

Because it is possible to write the sentence on anywhere on the surface, the absolute coordinates in $\mathbf{A}$ are calculated into relative coordinates $\mathbf{R}$. For any point $\mathbf{a} \in \mathbf{A} = [a_{x_i}, a_{y_i}, a_{d_i}]$, where $i \in [0, N]$ (N being the length of $\mathbf{A}$) we can calculate the point $\mathbf{r} \in \mathbf{R}$:

$$\mathbf{r}_j = \left[\frac{(a_{x_j} - a_{x_{j-1}})}{D}, \quad \frac{(a_{y_j} - a_{y_{j-1}})}{D}, \quad a_{d_j} \right] \quad j \in [1, N) \tag{2}$$

Since all the points are scaled by D, they will all have values roughly in the interval $[-1, +1]$, which is the interval of the $\tanh$-activation that the LSTM uses. [24]

3.1.4 Training targets

The label of the letter for every point is looked up and the targets are one-hot encoded for the training. That is the have a 1 at the index of the matching label and 0 elsewhere. While, also, the index 0 is special and it encodes empty spaces and non-mapped regions.

3.2 Neural network design

The network is built using a sequential Keras [6] model in Tensorflow 2.0 [5].

The input of the model is a matrix of size $N \times 3$, that is, N is the count of points and 3 is the size of $\mathbf{R}$ from 3.1.3. The output is a matrix of the size $N \times K$ where K is the count of labels in training targets (4.1).

The intermediate layers are three LSTM layers with an output size of 32. This number of layers and their sizes was determined experimentally and seems to be sufficient. The LSTM layers are followed by a dense fully-connected layer that transforms the 32 outputs to the amount of letter classes that are to be recognized. A softmax layer follows, it normalizes the probabilities into a sum of 1.

This means that for every $[x, y, d]$ vector that is fed into the model, it will output a vector $[l_0, l_1, \cdots, l_{K-1}]$. Each element in the vector represents a probability in the interval $[0, 1]$ that the output label is detected at the input, also called a *one-hot* representation.

3.2.1 Recurrent neural networks

Recurrent neural networks are a type of neural network that contain an internal state vector that keeps some about previous activations.

In the classic Elman RNN [1] there is only one state vector $\mathbf{h}$. If the input vector at time-step t is $\mathbf{x}_t$ and the output $\mathbf{y}_t$, then the calculation can be summarized like:

$$\mathbf{h}_t = \tanh(\mathbf{W}_{xh} \times \mathbf{x}_t + \mathbf{W}_{hh} \times \mathbf{h}_{t-1} + \mathbf{b}_h) \quad (3)$$

$$\mathbf{y}_t = \tanh(\mathbf{W}_{hy} \times \mathbf{h}_t + \mathbf{b}_y) \quad (4)$$

Where $\mathbf{W}_{xh}$, $\mathbf{W}_{hh}$, $\mathbf{W}_{hy}$ are the weight matrices between $\mathbf{x}$, $\mathbf{h}$, and $\mathbf{y}$ respectively. The operator $\times$ is matrix multiplication. The vectors $\mathbf{b}_h$ and $\mathbf{b}_y$ are bias vectors that have a linear effect.

The weights an biases are all trainable by a gradient descent algorithm because $\tanh$ (figure 7) is a differentiable function [28]:

3.2.2 LSTM

Long short-term memory is a special type of recurrent that adds more weights to calculate 'input', 'output' and 'forget' gate vectors [2]. The input being $\mathbf{x}$ and output $\mathbf{h}$; there is also

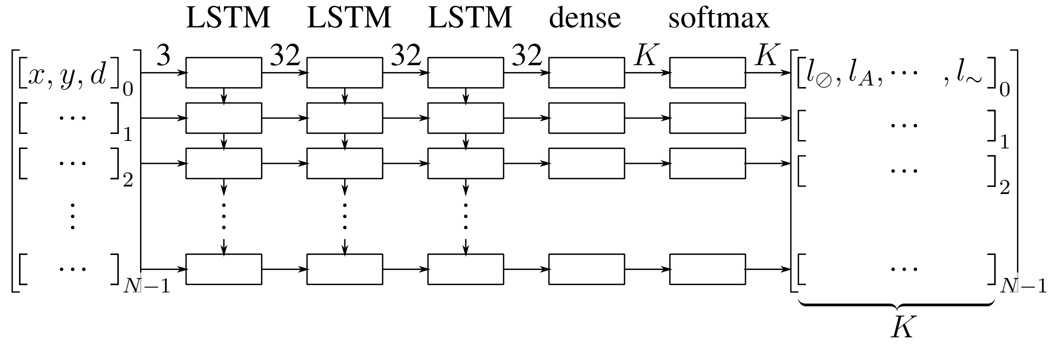


Figure 6: Recognition network design. The LSTM layers all have output sizes of 32, the dense layer resizes it to the number of classes K. The number of time-steps is N. Before the output a softmax layer is used to normalize the probabilities.

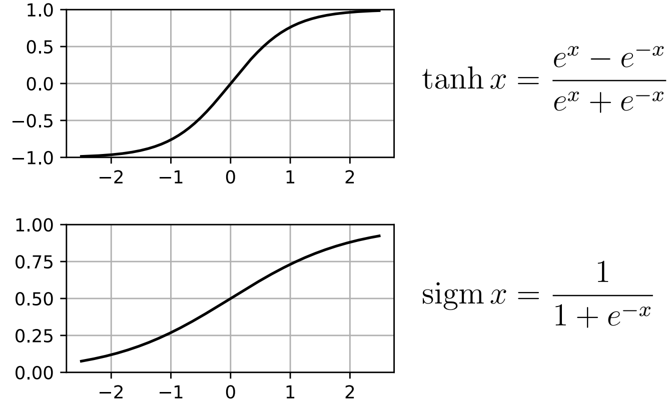


Figure 7: Hyperbolic tangent and the logistic sigmoid function

a stateful 'cell' vector $\mathbf{c}$:

$$\mathbf{f}_t = \operatorname{sigm}(\mathbf{W}_{xf} \times \mathbf{x}_t + \mathbf{W}_{hf} \times \mathbf{h}_{t-1} + \mathbf{b}_f) \tag{5}$$

$$\mathbf{i}_t = \operatorname{sigm}(\mathbf{W}_{xi} \times \mathbf{x}_t + \mathbf{W}_{hi} \times \mathbf{h}_{t-1} + \mathbf{b}_i) \tag{6}$$

$$\mathbf{o}_t = \operatorname{sigm}(\mathbf{W}_{xo} \times \mathbf{x}_t + \mathbf{W}_{ho} \times \mathbf{h}_{t-1} + \mathbf{b}_o) \tag{7}$$

$$\mathbf{c}_t = \mathbf{f}_t \circ \mathbf{c}_{t-1} + \mathbf{i}_t \circ \tanh(\mathbf{W}_{xc} \times \mathbf{x}_t + \mathbf{W}_{hc} \times \mathbf{h}_{t-1} + \mathbf{b}_c) \tag{8}$$

$$\mathbf{h}_t = \mathbf{o}_t \circ \tanh(\mathbf{c}_t) \tag{9}$$

Where the $\mathbf{W}_{vg}$ matrices are weight matrices of the gate g for the input vector v, the vectors b_g are biases for the gate g and the operator $\circ$ is element-wise multiplication.

3.2.3 Dense layer

The dense layer is a simple matrix multiplication with a weight $\mathbf{W}$ and a linear bias vector $\mathbf{b}$ where $\mathbf{x}$ is the input to the layer and $\mathbf{y}$ is the output:

$$\mathbf{y} = \mathbf{W} \times \mathbf{x} + \mathbf{b} \tag{10}$$

3.2.4 Softmax

The softmax function is used on the output as it is useful for classification problems. It takes a vector of size N and returns another vector of the same size:

$$\operatorname{softmax}(\mathbf{a})_i = \frac{e^{a_i}}{\sum_{j=0}^{N-1} e^{a_j}}, \quad \mathbf{a} = \{a_i | i \in [0, N)\} \tag{11}$$

Table 1: Model weights

Size	Weight description
3×128	LSTM1 input to $\mathbf{W}_{xf}, \mathbf{W}_{xi}, \mathbf{W}_{xo}, \mathbf{W}_{xc}$ combined
32×128	LSTM1 hidden to $\mathbf{W}_{hf}, \mathbf{W}_{hi}, \mathbf{W}_{ho}, \mathbf{W}_{hc}$ combined
1×128	LSTM1 biases $\mathbf{b}_f, \mathbf{b}_i, \mathbf{b}_o, \mathbf{b}_c$ combined
32×128	LSTM2 input from LSTM1
32×128	LSTM2 hidden
1×128	LSTM2 biases
32×128	LSTM3 input from LSTM2
32×128	LSTM3 hidden
1×128	LSTM3 biases
32×18	Dense layer to $K = 18$ labels
1×18	Dense layer biases
$21,842$	Total weight count

3.2.5 Categorical cross-entropy

Categorical cross-entropy [29] is used as the loss function for the training step. This is a loss function that takes in the predicted labels $\mathbf{l}$ of size N and training target labels that should be the actual results $\mathbf{t}$:

$$\mathrm{crossentropy}(\mathbf{l}, \mathbf{t}) = -\sum_{i=0}^{N-1} t_i \ln(l_i) \tag{12}$$

Since it takes a logarithm of the predicted label it goes naturally with the output of the softmax function (11), which uses an exponent, as also shown by Facebook [30].

3.2.6 Adam optimizer

Adam is a stochastic optimizer [31] that is particularly good at deep learning tasks. It is easy to use and only requires getting the gradient of the loss function (12) over the trainable variables, which can easily be obtained from the TensorFlow automatic differentiation engine [5].

3.2.7 Training

For performance purposes the sequence is split into 64 point long batches and the recurrent layers are unrolled. This number was chosen during experimentation and seems to work best across most platforms.

All the training samples are used to train for 200 epochs, where the order is shuffled on each epoch. In the end of every epoch, the model weights are saved if the average loss of the epoch is less than the previous loss.

The model weights and their sizes are in table 1.

3.3 Label detection

After the model outputs a one-hot encoded vector of probabilities $\mathbf{l}$, the label decoder finds ranges of letters that correlate to the input.

The decoder works as a simple state machine (see figure 8), it reads the output labels $\mathbf{l}_i$ until it finds a class that has $\mathrm{argmax}(\mathbf{l}_i) > 0.9$, where i is the point index of $\mathbf{r}$ (2).

The state machine then reads backward from that point to find a starting point where the class probablility was < 0.2.

It then reads until $\mathrm{argmax}(\mathbf{l}_i) \leq 0.2$, this i is then the ending index of the letter and the detected label is written to the output.

1: **procedure** GET_LABELS($\mathbf{l}$)
2: $\quad$ $\mathbf{r} \leftarrow []$
3: $\quad$ $c \leftarrow \oslash$ ▷ indeterminate class or `None`
4: $\quad$ **for** $i \in [0, \mathrm{len}(\mathbf{l}))$ **do**
5: $\quad\quad$ $\mathbf{v} \leftarrow \mathbf{l}_i$
6: $\quad\quad$ $c_{\mathrm{max}} \leftarrow \mathrm{argmax}(\mathbf{v})$ ▷ get the maximum class index for $\mathbf{v}$
7: $\quad\quad$ **if** $c = \oslash$ **then** ▷ No class found so far?
8: $\quad\quad\quad$ **if** $v_{c_{\mathrm{max}}} > 0.9$ **then** ▷ Start reading
9: $\quad\quad\quad\quad$ $c \leftarrow c_{\mathrm{max}}$
10: $\quad\quad\quad\quad$ $i_{\mathrm{start}} \leftarrow i$
11: $\quad\quad\quad\quad$ **for** $j \in (i_{\mathrm{start}}, i_{\mathrm{start}} - 10]$ **do** ▷ Read backwards to find class start
12: $\quad\quad\quad\quad\quad$ **if** $pr_{j_c} > 0.2$ **then**
13: $\quad\quad\quad\quad\quad\quad$ $i_{\mathrm{start}} \leftarrow j$
14: $\quad\quad\quad\quad\quad$ **end if**
15: $\quad\quad\quad\quad$ **end for**
16: $\quad\quad\quad$ **end if**
17: $\quad\quad$ **else** ▷ We have already found class c
18: $\quad\quad\quad$ **if** $v_c < 0.2$ and $c \neq 0$ and $i - i_{\mathrm{start}} > 10$ **then**
19: $\quad\quad\quad\quad$ $i_{\mathrm{end}} \leftarrow i$
20: $\quad\quad\quad\quad$ $\mathbf{r} \Leftarrow \{i_{\mathrm{start}}, i_{\mathrm{end}}, c\}$ ▷ Append to results $\mathbf{r}$
21: $\quad\quad\quad$ **end if**
22: $\quad\quad$ **end if**
23: $\quad$ **end for**
24: $\quad$ **return** $\mathbf{r}$
25: **end procedure**

Figure 8: Label reading state machine

This algorithm is quite simplistic and it sometimes fails such as when the start of the letter looks like another letter. One such exapmle is the letter *g* which in the beginning can look like the letter *a*. I have not added a way to detect these overlapping charaters like that.

4 Data manager

The data manager is a simple website that allows categorization of the samples and defining letter regions for training.

4.1 Letter classification

The user can select with a mouse a rectangle around a letter and then type the Unicode text that it corresponds to. If the selection contains some parts of unwanted letters, it is possible to fine tune the selection with sliders (figure 9).

The classified letters are just the same letters that exist in the sentence, with some exceptions (table 2). The first letter class (class 0) is the empty set class. This class is ignored by the label reading state machine (3.3).

One difference is the capital letters *A* and *K* are very different from the lower case. The letter *õ* is split into ˜ and *o*.

4.2 Result view

Results show the detected labels on the sample and the class probabilities as a graph. Each detected label is outlined with a box of a certain color and a small text label of the letter is shown.

The bounds of the box is a simple minimum-maximum of all the coordinate points that are between the detected start and end of the label.

Table 2: Classification labels

index	letter	index	letter
0	$\emptyset$	10	l
1	A	11	m
2	K	12	n
3	a	13	o
4	b	14	r
5	d	15	s
6	g	16	t
7	i	17	u
8	j	18	˜
9	k		

4.3 Confidence

The detector in `test.py` also calculates a 'confidence' level where it averages the maximum of each row of the neural net output. For a perfect result each row would contain a 1 in a single output class, so the average would be 1, or 100%. Since the softmax layer (11) assures that the maximum value in each row is ≤ 1, any lower average would indicate that the model is having difficulties in 'choosing' the right class (as can be seen on figure 10).

The confidence calculation for the result labels $\mathbf{l}$ is:

$$\text{confidence}(\mathbf{l}) = \frac{\sum_{i=0}^{N-1} \max \mathbf{p}_i}{N} \tag{13}$$

Where $\mathbf{l}$ is a two-dimensional array of $\mathbf{r} = \{\mathbf{p}_i | i \in [0, N)\}$, N being the length of the results. $\mathbf{p}_i = [p_k | k \in [0, K)]_i$ is the probability of the output at time step i of the class k in the interval $[0, 1]$. The $\max$ function gives the maximum value of all the input.

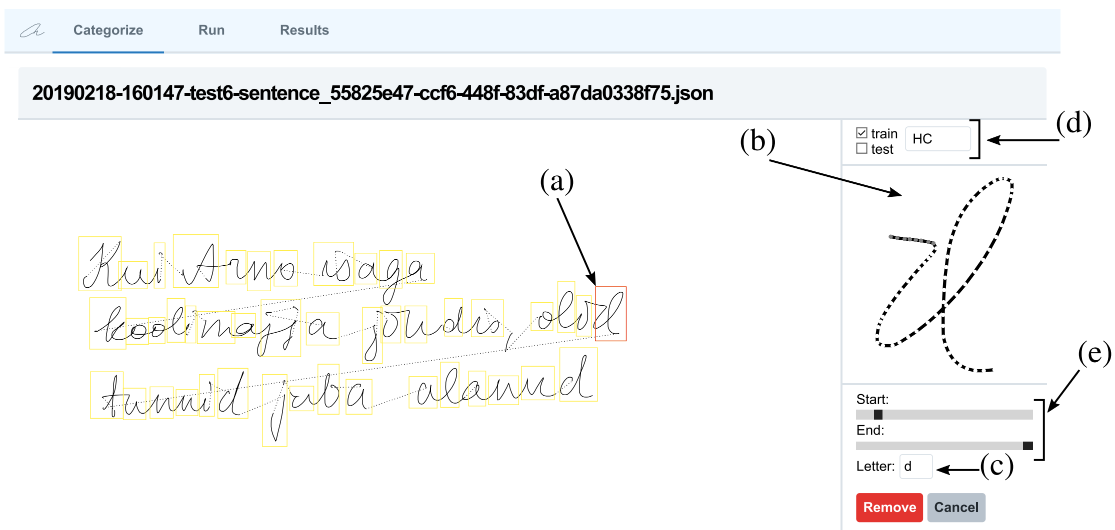


Figure 9: Data manager web view. The currently selected letter is *u* in the (a) handwriting, (b) shows a closeup of the selection, (c) is the Unicode text equivalent of the letter, (d) is the sample's categorization ('train' – handwriting training, 'test' – feature extraction; possible to mix but isn't done), (e) fine-tuning of the letter start and end point.

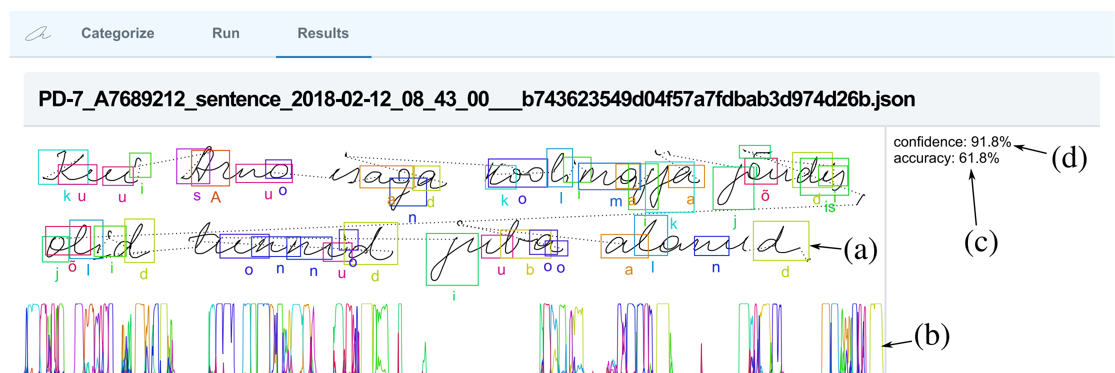


Figure 10: Results view. (a) shows the detected letter labels with surrounding boxes and text labels, (b) is the probability graph of all letters over time, (c) gives the accuracy rating compared to the definitions in figure 9, (d) the confidence (4.3) of the result.

5 Extracting kinematic features

From the recognized letter labels I extract a number of kinematic features for each letter. Using these features I hope to classify the samples into two groups: PD and HC.

The kinematic parameters I am using are mostly the same as the motion mass parameters in [12], where they are used for analyzing Luria's alternating series, but also for handwriting in [15].

The definitions of which are velocity mass V, acceleration mass A, jerk mass J and pressure mass P. They are calculated from all the points of recorded input per letter (that is not the preprocessed data in 3.1).

From the $\mathbf{p}_k = [x, y, p, t]$ vectors, where $k \in [0, N)$ to the length N of the letter's points and x, y being the coordinates of the point, p the pen pressure and t the timestamp, I calculate the vectors $\hat{\boldsymbol{\theta}}, \mathbf{d}$ (angle change and distance):

$$\Delta x_i,\ \ \Delta y_i = (\mathbf{p}_{(i-1)_x} - \mathbf{p}_{i_x}),\ \ (\mathbf{p}_{(i-1)_y} - \mathbf{p}_{i_y}) \tag{14}$$

$$\theta_i = \text{arctan2}(\Delta y_i, \Delta x_i) \tag{15}$$

$$d_i = \sqrt{\Delta x_i^2 + \Delta y_i^2} \tag{16}$$

$$\Delta t_i = t_{i-1} - t_i \tag{17}$$

Where $i \in [1, N)$ and the function arctan2 is from the Numpy library [3] (the same as the C language function `atan2`) that calculates the angle between (x, y) and the x-axis. To keep the arrays all the same length N, I define $\Delta x_0, \Delta y_0, \theta_0, l_0, \Delta t_0 = 0$.

Calculating the angles:

$$\Delta\theta_i = \theta_{i-1} - \theta_i \tag{18}$$

$$\widehat{\Delta\theta}_i = \big((\Delta\theta_i + \pi) \bmod 2\pi\big) - \pi \tag{19}$$

Where $\Delta\theta_i$ is the angle change between two points and $\widehat{\Delta\theta}_i$ is the angle change normalized between $[-\pi, +\pi]$. The operator mod is the modulo or remainder operator and defining $\widehat{\Delta\theta}_0 = 0$.

For the vectors $\mathbf{v}, \mathbf{a}, \mathbf{j}$ (velocity, acceleration, jerk):

$$v_i = d_i / t_i \tag{20}$$

$$a_i = v_i / t_i \tag{21}$$

$$j_i = a_i / t_i \tag{22}$$

Once again $i \in [1, N)$ and $v_0, a_0, j_0 = 0$ to normalize the length of the vectors.

From these definitions I calculate the scalar motion mass sums Θ_S – angle, V_S – velocity, A_S – acceleration, J_S – jerk, and P_S – pressure:

$$\Theta_S = \sum_{i=0}^{N-1} |\hat{\theta}_i| \tag{23}$$

$$V_S = \sum_{i=0}^{N-1} |v_i| \tag{24}$$

$$A_S = \sum_{i=0}^{N-1} |a_i| \tag{25}$$

$$J_S = \sum_{i=0}^{N-1} |j_i| \tag{26}$$

$$P_S = \sum_{i=0}^{N-1} p_i \tag{27}$$

Since the writing of each letter is variable in length and time, similarly to [12], I normalize the motion masses in regard to the L_S – total length – using $\mathbf{d}$ from (16):

$$L_S = \sum_{i=0}^{N-1} d_i \tag{28}$$

The scalar values Θ, V, A, J, P are a simple ratio with the length L_S:

$$\Theta = \Theta_S / L_S \tag{29}$$

$$V = V_S / L_S \tag{30}$$

$$A = A_S / L_S \tag{31}$$

$$J = J_S / L_S \tag{32}$$

$$P = P_S / L_S \tag{33}$$

It is possible to also look at the ratio with regard to time but the results were quite similar.

6 Validation

To validate the motion mass features, I use cross-validation over all the letters that are detected.

To choose which features/letters to use, I calculate the Fisher score for each letter-feature pair:

$$F = \frac{\sum_j^k p_j(\mu_j - \mu)^2}{\sum_j^k p_j \sigma_j^2} \tag{34}$$

Where k is the count of classes and p_j is the proportion of the class to the number of samples, μ_j is the mean of the feature across class j, μ is the mean of all classes, σ_j^2 is the variance of the class j.

Since there are only two classes, HC and PD, the calculation simplifies to:

$$F = \frac{p_{\mathrm{hc}}(\mu_{\mathrm{hc}} - \mu)^2 + p_{\mathrm{pd}}(\mu_{\mathrm{pd}} - \mu)^2}{p_{\mathrm{hc}}\sigma_{\mathrm{hc}}^2 + p_{\mathrm{pd}}\sigma_{\mathrm{pd}}^2} \tag{35}$$

The ten best features ordered by the Fisher score are in table 3.

With a combination of these features I trained several scikit-learn classifiers and used k-fold cross-validation to check the accuracy of the features to classify between the HC and PD groups. The results are in table 4.

Table 3: Top 10 features

n	feature	letter	F	p	N
1.	Θ	n	0.18	7.03×10^{-1}	128
2.	P	k	0.15	7.12×10^{-1}	66
3.	P	n	0.13	7.03×10^{-1}	128
4.	Θ	k	0.12	7.12×10^{-1}	66
5.	Θ	o	0.09	7.13×10^{-1}	122
6.	P	d	0.09	7.18×10^{-1}	117
7.	P	u	0.07	7.37×10^{-1}	156
8.	Θ	d	0.07	7.18×10^{-1}	117
9.	Θ	u	0.07	7.37×10^{-1}	156
10.	P	o	0.06	7.13×10^{-1}	122

6.1 Scikit-learn classifiers

Scikit-learn [23] is an easy to use library for Python that comes with many built in classifiers. The basic API is the same for all of them, which means I can declare them in an array and invoke them in a loop.

I chose to use the following classifiers for this task:

SGD Support vector machine [32] trained using stochastic gradient descent

DT Decision tree

3NN Nearest neighbor, $n = 3$

5NN Nearest neighbor, $n = 5$

MLP Multi layer perceptron, hidden layers 5×2

For each classifier I also use a varying number of the top features (table 3). The numbers range from 2 to 10, giving a total of 40 combinations.

6.1.1 Cross-validation

Cross-validation is a method to validate classifiers by taking dividing the data into 'folds'. Then there are multiple train-test cycles, typically equal to the amount of folds. In each cycle, a fold is excluded from training the classifier and is used for testing only.

I decided to choose the number of folds to be 6 because of the number of samples available.

6.1.2 Decision trees

Decision trees are binary trees that compare a feature in every node and reach a decision in the end. They may not give the best accuracy but they are easy to interpret using a graph such as in figure 11.

Table 4: Validation

n-features	classifier	accuracy
6	MLP	75.8%
5	MLP	73.3%
8	MLP	73.3%
4	SGD	73.3%
5	DT	72.5%
9	MLP	70.0%
4	MLP	70.0%
3	MLP	70.0%
7	MLP	70.0%
9	DT	70.0%

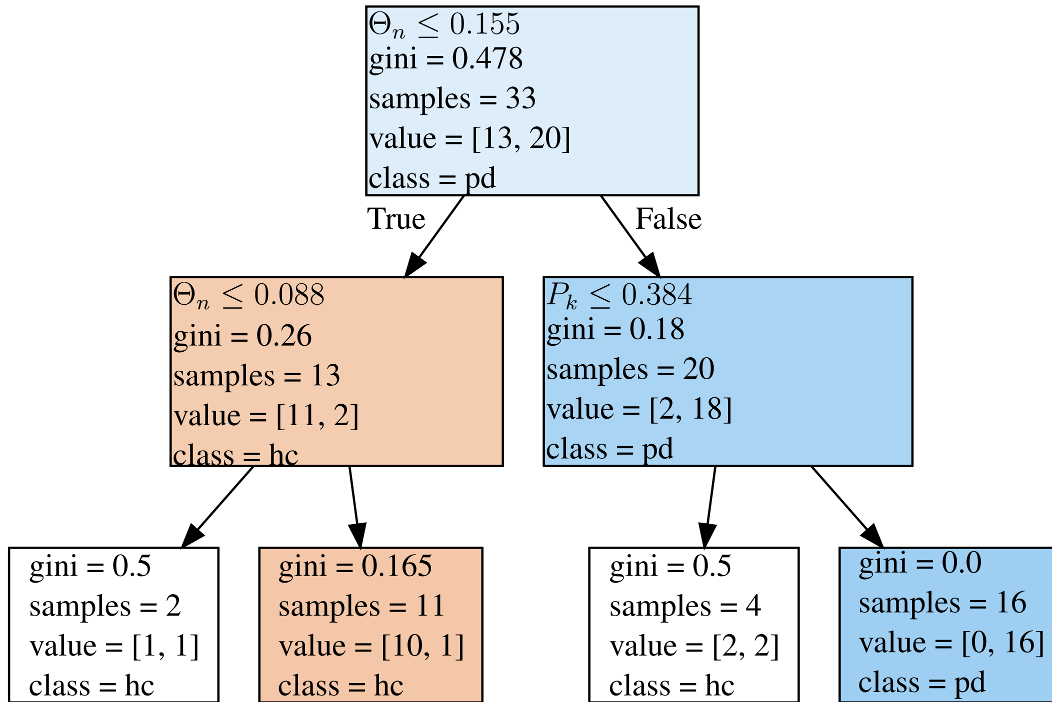


Figure 11: Decision tree with two features. Θ_n is the angle mass sum over the letter n. P_k is the pressure mass sum over the letter k. This graph is generated automatically by Graphviz [33].

6.1.3 Multi layer perceptron

Is a multi-layer neural net using the quasi-Newton Broyden–Fletcher–Goldfarb–Shanno (BFGS) algorithm for its optimizer. The results are promising, however, with large number of features required.

6.1.4 Stochastic gradient descent

Is actually using a support vector machine as its model. Overall it gives a pretty good accuracy with only two features.

6.1.5 Nearest neighbor

The nearest neighbor classifiers perform very poorly, so they are not considered.

7 Conclusions

I started this task trying to create a handwriting recognizer that could find just some letters. But the end result was surprisingly even more effective than I hoped for.

The handwriting recognizer is functional enough to be able to use its output for diagnosis purposes.

7.1 Shortcomings and future improvements

It should be possible to improve the training performance significantly if there were more samples being processed in parallel. This would probably allow better acceleration by with GPU based training.

The recurrent model is currently linear and forward only, it could be augmented by a reverse model which can 'see' letters ahead. I tried using a bidirectional LSTM, however it did not give good results. Experiments with adding a delay to network output were not fruitful.

Adding a way to automatically detect and normalize the writing scale and skew before training and inference should improve accuracy considerably.

Having a separate model for block letters (all caps) would help recognition because currently the model is not very good at it. Choosing appropriate letter labels would be too.

It should be possible to create an acquisition page into the management web, which would allow collecting new samples easily. The letter categorization interface could use a function to use the current model to pre-generate labels for human checking.

There is no way handle disjointed strokes. Some writers may – for instance – only cross the *t*'s and dot the *i*'s at the end of a word. It is also hard to make sense of corrections if the writer goes back to fix a mistake.

The max-length input filter works well for input that has a high sampling rate, but for lower quality input it will fail. Better would be a way to resample the input using some kind of interpolation, such as linear or Bézier splines [27].

The model does not currently detect white-space and there is no way to add labels for whitespace in the labeling tool (4). There is every reason to expect it would be very accurate, since it does seem to avoid the spaces between words and also the 'carriage' return to the beginning of the next line.

References

[1] Elman, J. L.: Finding Structure in Time. *Cognitive Science* 14, 179–211 (1990)

[2] Hochreiter, S. and Schmidhuber, J.: Long Short-Term Memory. *Neural Computation* 9, 1735–1780 (1997)

[3] Oliphant, T.: NumPy: A guide to NumPy. USA: Trelgol Publishing (2006–). [Online; accessed 2019-05-16]

[4] McKinney, W.: Data Structures for Statistical Computing in Python. In van der Walt, S. and Millman, J., eds., *Proceedings of the 9th Python in Science Conference*, 51–56 (2010)

[5] Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., Corrado, G. S., Davis, A., Dean, J., Devin, M., Ghemawat, S., Goodfellow, I., Harp, A., Irving, G., Isard, M., Jia, Y., Jozefowicz, R., Kaiser, L., Kudlur, M., Levenberg, J., Mané, D., Monga, R., Moore, S., Murray, D., Olah, C., Schuster, M., Shlens, J., Steiner, B., Sutskever, I., Talwar, K., Tucker, P., Vanhoucke, V., Vasudevan, V., Viégas, F., Vinyals, O., Warden, P., Wattenberg, M., Wicke, M., Yu, Y., and Zheng, X.: TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems (2015). Software available from `https://tensorflow.org`

[6] Chollet, F. et al.: Keras. `https://keras.io` (2015)

[7] de Lau, L. M. and Breteler, M. M.: Epidemiology of Parkinson's disease. *The Lancet Neurology* 5, 525–535 (2006)

[8] Yarnall, A., Archibald, N., and Burn, D.: Parkinson's disease. *Medicine* 40, 529–535 (2012). Neurology: Part 3 of 3

[9] Marquardt, C. and Mai, N.: A computational procedure for movement analysis in handwriting. *Journal of Neuroscience Methods* 52, 39–45 (1994)

[10] Eichhorn, T. E., Gasser, T., Mai, N., Marquardt, C., Arnold, G., Schwarz, J., and Oertel, W. H.: Computational analysis of open loop handwriting movements in Parkinson's disease: A rapid method to detect dopamimetic effects. *Movement Disorders* 11, 289–297 (1996)

[11] Kozhenkina, J.: *Quantitative Analysis of the Kinematic Features for the Luria's Alternating Series Test*. Master's thesis, Tallinn University of Technology, Tallinn (2016)

[12] Nõmm, S., Toomela, A., Kozhenkina, J., and Toomsoo, T.: Quantitative analysis in the digital Luria's alternating series tests. In *2016 14th International Conference*

on Control, Automation, Robotics and Vision (ICARCV), 1–6. IEEE (2016). ISBN 9781509035496

[13] Nõmm, S., Bardõš, K., Mašarov, I., Kozhenkina, J., Toomela, A., and Toomsoo, T.: Recognition and Analysis of the Contours Drawn during the Poppelreuter's Test. In *2016 15th IEEE International Conference on Machine Learning and Applications (ICMLA)*, 170–175. IEEE (2016). ISBN 9781509061679

[14] Luria, A. R.: *The Higher Cortical Functions in Man* (1962)

[15] Drotár, P., Mekyska, J., Rektorová, I., Masarová, L., Smékal, Z., and Faundez-Zanuy, M.: Evaluation of handwriting kinematics and pressure for differential diagnosis of Parkinson's disease. *Artificial Intelligence in Medicine* 67, 39–46 (2016)

[16] Mašarov, I.: *Digital Clock Drawing Test Implementation and Analysis*. Master's thesis, Tallinn University of Technology, Tallinn (2017)

[17] LeCun, Y., Cortes, C., and Burges, C. J.: The MNIST database of handwritten digits. `http://yann.lecun.com/exdb/mnist/` (1998–)

[18] Bardõš, K.: *Analysis of Interpretable Anomalies and Kinematic Parameters in Luria's Alternating Series Tests for Parkinson's Disease Modeling*. Master's thesis, Tallinn University of Technology, Tallinn (2018)

[19] Nõmm, S., Bardõš, K., Toomela, A., Medijainen, K., and Taba, P.: Detailed Analysis of the Luria's Alternating SeriesTests for Parkinson's Disease Diagnostics. In *2018 17th IEEE International Conference on Machine Learning and Applications (ICMLA)*, 1347–1352 (2018)

[20] Toodo, T.-B.: *Assessment of parameters from the handwritten sentence test used to diagnose Parkinson's disease*. Master's thesis, Tallinn University of Technology, Tallinn (2018)

[21] Szegedy, C., Vanhoucke, V., Ioffe, S., Shlens, J., and Wojna, Z.: Rethinking the Inception Architecture for Computer Vision. *arXiv e-prints* arXiv:1512.00567 (2015)

[22] Apple: *Apple Developer Documentation: NSDate*. `https://developer.apple.com/documentation/foundation/nsdate`

[23] Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., Vanderplas, J., Passos, A., Cournapeau, D., Brucher, M., Perrot, M., and Duchesnay, E.: Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research* 12, 2825–2830 (2011)

[24] Karpathy, A.: The Unreasonable Effectiveness of Recurrent Neural Networks. `http://karpathy.github.io/2015/05/21/rnn-effectiveness/` (2015)

[25] Graves, A.: Generating Sequences With Recurrent Neural Networks. *arXiv e-prints* arXiv:1308.0850 (2013)

[26] Graves, A.: *Supervised Sequence Labelling with Recurrent Neural Networks*. Studies in Computational Intelligence. Springer Berlin Heidelberg, University of Toronto, Toronto, Canada (2012). ISBN 9783642247972

[27] Carbune, V., Gonnet, P., Deselaers, T., Rowley, H. A., Daryin, A., Calvo, M., Wang, L.-L., Keysers, D., Feuz, S., and Gervais, P.: Fast Multi-language LSTM-based Online Handwriting Recognition. *arXiv e-prints* arXiv:1902.10525 (2019)

[28] Karpathy, A., Johnson, J., and Fei-Fei, L.: Visualizing and Understanding Recurrent Networks. *arXiv e-prints* arXiv:1506.02078 (2015)

[29] de Boer, P.-T., Kroese, D. P., Mannor, S., and Rubinstein, R. Y.: A Tutorial on the Cross-Entropy Method. *Annals of Operations Research* 134, 19–67 (2005)

[30] Mahajan, D., Girshick, R., Ramanathan, V., He, K., Paluri, M., Li, Y., Bharambe, A., and van der Maaten, L.: Exploring the Limits of Weakly Supervised Pretraining. *arXiv e-prints* arXiv:1805.00932 (2018)

[31] Kingma, D. P. and Ba, J.: Adam: A Method for Stochastic Optimization. *arXiv e-prints* arXiv:1412.6980 (2014)

[32] Cortes, C. and Vapnik, V.: Support-vector networks. *Machine Learning* 20, 273–297 (1995)

[33] Gansner, E. R. and North, S. C.: An open graph visualization system and its applications to software engineering. *Software—Practice and Experience* 30, 1203–1233 (2000)

Appendix 1 – Readme file

Handwriting recognition and by recurrent NN and analysis

This is the software part of my thesis.

Requirements

The software should run on all modern Windows and Linux 64-bit machines. However, the Tensorflow library requires that the processor support AVX instructions. GPU support is limited to NVIDIA graphics cards with CUDA, currently.

Installation

The scripts run on Python version 3.6 or 3.7.

To run the Python scripts you should first create a virtual environment `venv`:

```
$ python3 -m venv venv  # in Linux
PS> python -m venv venv # in Windows
```

After that you need to activate the virtual environment:

```
$ . venv/bin/activate            # in Linux
PS> . venv/Scripts/Activate.ps1  # in Windows Powershell
```

You can see if the virtual environment is active when there is `(venv)` prefixed on the prompt. Then you can run pip to install the required packages:

```
(venv) $ pip install -r requirements.txt
```

Scripts

Make sure to activate the Python virtual environment for each new shell/terminal session before trying to run any script.

These are the scripts:

train.py Trains the model using the input data in `sentences/` and `metadata/` saving the output model to JSON files in `model/`.

The training is done by default on the CPU, if you want to try using GPU then you can install `tensorflow-gpu`:

```
(venv) $ pip install tensorflow-gpu==2.0.0-alpha0
```

Training the model can take some time, so that's why I have included a pre-trained model in the repository already.

test.py Runs inference using the saved model in `model/` and saves the results into `results/`

data.py Contains functions that load and process data from JSON files.

model.py Contains the handwriting recognizer as a reusable class.

features.py Contains functions that read data from `results/` and convert them to pandas `DataFrame` objects.

validate.py Contains code that reads the results from `test.py` and calculates F-test scores for the features it finds. Then runs cross-validation on the extracted features.

Data manager website

The website runs on ASP.NET core. To install that you need the .NET Core 2.2 SDK.

Then you can build and run the web, it is recommended to have the python venv active so the python scripts can run properly.

```
(venv) DataMgr$ dotnet run
```

By default the site opens on the URL `http://localhost:50695/`.

Assets If you change any of the Tailwind CSS options or otherwise add custom CSS, then you need to run `gulp` to rebuild the assets. The built assets are included in the source already.

Node.js is required to build the assets. Run `npm` and `gulp` to build the assets:

```
$ cd DataMgr
DataMgr$ npm install
DataMgr$ node_modules/.bin/gulp
```

All JavaScript files are included as ES6 modules, they don't need any sort of build step.

Using DataMgr The *Categorize page* is used to select sentences for training or testing. You can select rectangles around letters and categorize them, also to fine tune the start and end of a letter.

The *Run page* is just to run scripts remotely and display output.

The *Results page* is to show the output from `test.py`. It overlays recognized letters over the original input sentence and shows the probability graph of each output class from the neural network outputs.

Thesis draft

The thesis is in the `draft` subdirectory. It is built with the 2018 version of TeX Live. To build run these commands:

```
$ cd draft
draft$ latexmk -pdf
```

The output is in `main.pdf`. I recommend using Visual Studio Code, there is an excellent extension called LaTeX Workshop that can generate the .pdf output automatically on save and then refresh the preview window.