Dynamic Handwritten Digits Dataset: Generation and Analysis

Ronny Majani 220201060

A Thesis Report Submitted to the
Faculty of Engineering in Partial Fulfillment
of the Requirements for the Degree of

BACHELOR OF SCIENCE

(Academic Project)

Department: Computer Engineering

Izmir Institute of Technology
Izmir, Turkey

May, 2018

We approve the thesis of Ronny Majani

Date of Signature:

.....

Asst. Prof. Mustafa Özuysal
Supervisor
Department of Computer Engineering

Abstract

The subject of Handwriting recognition, specifically handwritten character recognition, has received a lot of attention, due to the clarity of it's narrow scope, small problem domain, availability of datasets, and the ease of creating these datasets. These datasets are usually characterized as static bitmap images of isolated alphanumerical characters written by people using common mediums such as paper and pen. Due to the static nature of these datasets, there has been little to no open research done on the feasibility of using recurrent types of machine learning algorithms as a solution. This project fills that gap by proposing and evaluating a novel approach to handwritten character recognition that consists of using temporal data trained on a recurrent neural network model. To evaluate this approach, a handwritten Latin numeral *(0,1,2,3,4,5,6,7,8,9)* recognizer was developed then trained and evaluated on a small dataset generated by 36 people, resulting in a model with reasonable accuracy, high durability to occlusions, but high sensitivity to noise.

Contents

1 Introduction **6**
1.1 Scope . . . 6
1.2 Outline . . . 6
1.3 Abbreviations . . . 7
1.4 Terminology . . . 7

2 Literature Survey **7**

3 Research Methodology **8**
3.1 Data Specification . . . 8
3.2 Data Preparation . . . 8
3.2.1 Dataset Preprocessing . . . 9
3.2.2 Dataset Augmentation . . . 11
3.3 Data Recognizer . . . 12
3.3.1 Model Design . . . 12

4 Implementation **14**
4.1 Data Format . . . 14
4.2 Data Collection System . . . 15
4.2.1 Data Collection Mechanism Specifications . . . 16
4.2.2 Data Collection System Software Overview . . . 16
4.2.3 User Interface . . . 19
4.2.4 Program Flow . . . 21
4.3 Collected Data Set . . . 23
4.4 Research Environment . . . 23
4.4.1 Software Requirements . . . 23
4.4.2 Workspace . . . 24
4.4.3 Abstractions . . . 25
4.5 Training and Evaluating the Model . . . 25
4.5.1 Dataset Train/Validation/Test split . . . 25
4.5.2 Training the Recognizer . . . 26
4.5.3 Evaluating Network . . . 26
4.6 Research Tests . . . 27
4.6.1 Effect of Preprocessing Operations . . . 27
4.6.2 Occlusions . . . 30
4.6.3 Noise . . . 31
4.6.4 Number of Samples for Spline Interpolation and Re-sampling . . . 33
4.7 Live Data Prediction Tool . . . 33

5 Conclusion & Impact **34**

6 Discussion & Future Work **35**
6.1 Pitfalls . . . 35
6.2 Future Work . . . 35
6.2.1 Bigger Dataset . . . 35
6.2.2 Use Additional Features . . . 35
6.2.3 Improve the Augmentation and Preprocessing Operations proposed . . . 36
6.2.4 More Dataset Augmentations . . . 36
6.2.5 Extend to alphabetic letters and possibly other symbols too . . . 36
6.2.6 Look for correlations between gender, age and handwriting style . . . 36
6.2.7 Faster Online (Live) Handwritten User Input . . . 36
6.2.8 Generative Model . . . 36

7 References **37**

1 Introduction

The subject of Handwriting recognition, specifically single character recognition, has received a lot of attention, due to the clarity of it's narrow scope, small problem domain, availability of datasets, and the ease of creating these datasets. The first practice exercise many new students of neural networks are faced with is to implement some form of naive optical character recognition, using simple feed forward neural networks, and then convolutional neural networks. It is also quite common for new algorithms developed by researchers in this field to test the performance of these algorithms on a character recognition problem.

Besides its research potential, handwritten character recognition has other more practical applications, such as an alternative digital input method, such as the Windows operating system's Handwriting Input tool.

The datasets used in the problem domain of Optical Character Recognition are usually characterized as static bitmap images of isolated alphanumerical characters written by people using common mediums such as paper and pen. As for word recognition of compound characters many methods exist for dealing with them, most commonly some form of image preprocessing is applied to separate the individual characters in the image and feed them to a recognizer. Due to the static nature of these datasets, there has been little to no open research done on the feasibility of using recurrent types of machine learning algorithms as a solution. This project aims to fill that gap by proposing a novel approach to the handwritten character recognition problem.

The proposed approach consists of representing handwritten characters using temporal sequential data instead of static bitmap images, and then training a character recognizer model designed to exploit the temporal property of the data.

1.1 Scope

The objective of this thesis is to describe a novel approach to character recognition using temporal data by designing and developing a handwritten Latin numeral (digit) recognizer as a proof of concept and then evaluating the performance and properties of this model. The domain of the data used as input to our model consists of handwritten Latin numerals (digits):
0, 1, 2, 3, 4, 5, 6, 7, 8, 9.

1.2 Outline

After conducting a standard literature survey, we begin this report by describing the nature of the data demanded by the proposed approach. Then we define the character recognizer model(s) used and their properties. Next we outline the data preparation process including data normalization, dataset augmentation, and the splitting of the data into train, validation, and test subsets. Then we jump into the implementation details and experimental setup of our research, starting with a detailed explanation of our data file format, the data collection system, and a brief overview of the dataset generated so far, following a description of the research workspace employed to achieve this project's objectives. After that we detail each experiment we ran, and the results they produced, with the hope to be thorough enough that they may be easily reproducible. Finally we conclude this document with a summary of the findings of our research, including a subtle characterization

of the proposed model's properties, followed by a discussion of its pitfalls, and possible extensions to our work that others may wish to pursue.

1.3 Abbreviations

GRU Gated Recurrent Unit

LSTM Long Short Term Memory

OCR Optical Character Recognition

ReLu Rectified Linear Units

RNN Recurrent Neural Network

1.4 Terminology

Data Set A set of Digit Sets.

Digit A single Latin numeral.

Digit Set A set of Samples produced by a single person.

Frame A single snapshot of the state of the digit being draw at a specific point in time.

Sample A sequence of frames that represent a single digit drawn by a user.

2 Literature Review

This project is mostly inspired by LeCun's work on his MNIST database[7] and his popular paper on Gradient based learning for OCR problem [8]. This project aims to explore beyond the classical static data that was used in previous research, and discover the consequences of adding the temporal element to the problem of digit recognition.

There hasn't been much work with dynamic handwritten data. One dataset that worked with dynamic data is the e-BioSign dataset.[11] The folks that worked on that project used a variety of methods for collecting dynamic handwritten data, which motivated the decision that using a Wacom tablet is a feasible option given the project's limited budget and requirements.

A significant portion of this project consists of the design and development of a recurrent neural network. A variety of models have been proposed and assessed in the past 30+ years. To assist with the choosing an appropriate model we referenced [9] and [6]

In conclusion, there seems to be no available research or datasets regarding dynamic (sequential) handwritten digits. Hopefully this project can be the first step in this unnoticed subfield of handwritten recognition.

3 Research Methodology

In the design of a character recognizer (and other types of recognizers) there are 4 main variables that affect performance and the ability to correctly categorize given inputs:

- The preprocessing applied to the data before training
- The way the dataset is split into train, validation, and test subsets and then shuffled.
- The machine learning model's design
- The hyper-parameters used for training the model

In this section we first begin with a detailed description of the nature of the data required by the approach proposed in this thesis, then we dive into the 4 aforementioned variables one at a time, describing the design choices we made and their effect on the final results.

3.1 Data Specification

The Data Set consists of a set of Digit Sets. Each Digit Set set contains a list of samples for each of the 10 Latin numbers (0, 1, 2, ..., 9). Each sample in our dataset consists of a chronological sequence of frames that represent the path of the user's pen as visualized in figure [1]. Each frame consists of a tuple $(x, y, ...)$ where the first 2 elements are the 2D coordinates of the pen's tip relative to the top left corner of the drawing tablet's pad. Optional additional information has also been collected such as the pen's pressure, and the time difference between consecutive frames, but hasn't been employed in our research. The use of such additional features is a potential opportunity for future research.

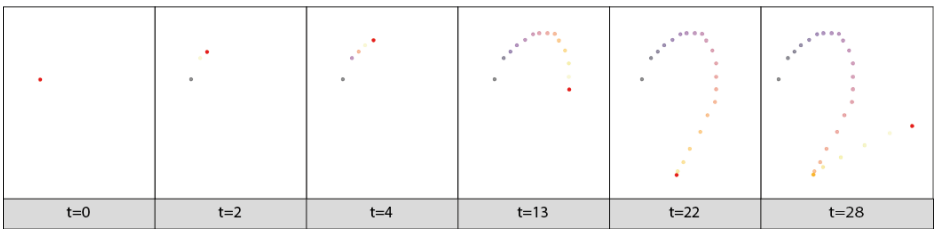


Figure 1: Visualization of Sequential Data

3.2 Data Preparation

Before we dive into designing a neural network and training it on our dataset, we must prepare the data to optimize the training and testing of the machine learning model. This involves applying certain preprocessing operations to normalize the data and reduce feature variance while trying to maintain the relative values of the features, and augmenting the dataset with transformed versions of the original samples to compensate for the lack of more data. The most significant step in the preparation of the dataset is to deal with the varying length of each sample's sequence. There are two options to dealing with this:

1. Appending each sample's sequence with a special *masking* value, so that all samples' sequences have a uniform length, and then designing our recognizer model to include a Masking Layer that will ignore the appended values.

2. Perform spline interpolation on each digit (sample) and then re-sampling an arbitrary number of new frames from the interpolated function.

We chose to go with spline interpolation and re-sampling, as it is computationally less expensive than a masking layer, and bring additional benefits discussed in section 3.2.1

3.2.1 Dataset Preprocessing

we must clean up and normalize the data to minimize the effect of outliers and variance in scale and offset. The following preprocessing operations were performed in order:

1. **Mean Centering:** The position features (X, Y) of each sample are translated such that their mean (the center of the digit) is repositioned at the coordinates (0, 0). This compensated for the variance in the position on the drawing pad where a digit was drawn.



(a) before mean centering (b) after mean centering

Figure 2: Mean Centering

2. **Unit Distance Normalization:** The position features (X, Y) of each digit are scaled (normalized) such that the distance of the point (frame) farthest from the digit's mean will equal 1.0. Hence the distance of all points from their mean will be in the range $[0.0, 1.0]$. This results in scale invariance of the data.



(a) before unit normalization (b) after unit normalization

Figure 3: Unit Distance Normalization

3. **Clean Sequence from Repeated points:** We cleaned the data from points that are too close to each other and hence can be viewed as redundant. The main reason to do this was to ensure compatibility with spline interpolation, and also such repeated points do not contribute to the motion of the pen's drawing and merely represent a state of rest or faster sampling than the pen's motion.

4. **Cubic Spline Interpolation and Re-sampling:** To deal with the variance in the sequence length of each sample (and hence the variance speed), we perform cubic spline interpolation on the position deatures (X, Y) of each digit, then re-sample an arbitrarily number of samples from the resulting spline. This effectively makes the data somewhat invariant to the speed of which samples were made and results in a uniform constant sequence length for all samples, saving us from having to use any masking layer in our neural network (resulting in a faster and more stable network). It also helps smoothen the digits as the new samples are taken uniformly, which helps compensate when a sample was drawn with varying speed (some parts of the digit have a higher density of frames compared to others) and makes it easier for the neural network to compare different samples.

   We chose cubic spline instead of higher orders as cubic interpolation seems to result in more natural looking curves that closer resembled the original digits.

   We chose to resample 50 points from the spline after testing the effect of different values, and calculating the average frames per sample in the dataset (40 50%). We found that too high a sampling rate unnecessarily increases the amount of data to process and increases the data's complexity. Too low a sampling rate leads to loss of information and a more jagged digit. Number of Samples between 25 and 100 seemed optimal, and so we went with 50 as its quite close to the average number of frames per sample in the dataset.



(a) before spline interpolation

(b) after spline interpolation and re-sampling 50 points

Figure 4: Cubic Spline Interpolation and Re-sampling

3.2.2 Dataset Augmentation

In addition to normalizing the data, we compensated for the small size of our dataset by augmenting it with various transformation and manipulations of the original data. The following augmentations were performed:

- **Reverse Sample *(Digit)* Sequences:** For each sample in the dataset, we created a new sample that consists of its sequence in reverse (drawing the digit backwards). This simulated the effect of using a Bidirectional LSTM and helps the neural network learn the relation between both previous and forward points. The maximum attainable accuracy and the speed at which the neural network reached this accuracy improved significantly after doing this augmentation.



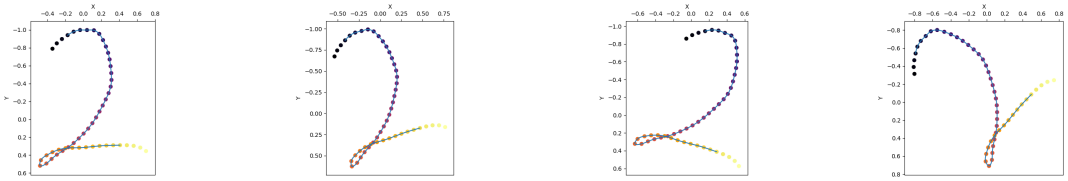
(a) original

(b) reverse digit

Figure 5: Reverse Sample Sequence

- **Rotating Samples:** For each sample (digit) in the dataset, we create a new digit that is equal to rotating all its points' (frames) (X, Y) features a given angle around the digit's center. We augmented the dataset with multiple rotations, where for each digits we generated 8 new digits equal to the following 8 rotations: $5°, 10°, 15°, 45°, -5°, -10°, -15°, -45°$.

This augmentation improved the system's accuracy by another 3% to 4% and reduced the confusion of digits. We believe that by showing the neural network the same digit rotated around it will get better at finding more relative more complex features for identifying digits that are less dependent on the orientation and absolute coordinates.



(a) original (b) 15° rotation (c) ⁻20° rotation (d) 45° rotation

Figure 6: Rotate Samples

3.3 Data Recognizer

3.3.1 Model Design

Recurrent Neural Network Models The first step in the design of our recognizer is to choose a neural network model to build with. Obviously to take advantage of the sequential nature of our data, we opt for a recurrent model. The most prominent and widely known models are vanilla RNNs, nd there gated extensions GRUs[3] and LSTMs. Vanilla RNNs are known to suffer with the exploding and vanishing gradients problem, especially with longer time series, and is the main motivation behind the design of their gated extensions. The final choice comes down between GRUs and LSTMs. LSTMs are much more popular and widely used, but it has been shown that GRUs perform better on smaller datasets[4] and that they are comparable to LSTMs in performance and at times outperforming them in many tasks.[6] Also, due to the simpler design of GRUs (they have only 2 gates compared to LSTMs and their 3 gates), GRUs are computationally less expensive and therefore faster to train with. For these reasons we have chosen to use GRUs as the building blocks of our recognizer model.

Recognizer Architecture The architecture of the recognizer we designed is depicted in figure [7]. The recognizer is divided into two phases: first the recurrent phase, where we stacked 3 GRUs where the first and second GRU pass its entire hidden state to the second and third GRU respectively. The motivation behind this stacking is that the emphasis of our recognizer should be on finding more complex temporal patterns than relying on the actual current values of the features. *The GRUs used the default activations of sigmoid for its recurrent gates, and tanh for its output, and unless otherwise explicitly stated, the remaining parameters were left to the default set by Keras's GRU layer class initializer.* The second phase of the recognizer, is categorization, consisting of one hidden Dense layer with ReLu activate and ending with a standard softmax layer that output the categorical probabilities that a given input sequence corresponds to each of the 10 Latin numerals. ReLu activation also improved the training and validation accuracy, and the speed at which the accuracy was reached (less epochs). From an abstract perspective this architecture is a many-to-one recurrent model, as its input is a sequence, and its output is a single category, and is only valid once the entire sequence has been processed.

Regularization The more complex a model becomes, the higher the risk that it will overfit to a given dataset. To prevent this, and to increase the recognizers ability to generalize to data it has not seen yet, we applied a set of standard regularization techniques. First and most apparent in the figure [7] is the extensive use of Dropout layers between each consecutive layer. The motivation for this design choice, is to add a stochastic nature to the model, preventing it from learning on the full range of weights and values passed between neurons in the network. Initially we experimented with a higher dropout parameter of 50% but found that such a high stochastic nature led to slower training time, and less stable results. Later we decreased the Dropout down to 10% and found the performance to be quite sufficient. Next we also added L2 penalty for weights to also prevent the model from overfitting on the training set, this also improved the stable maximum validation accuracy a bit.

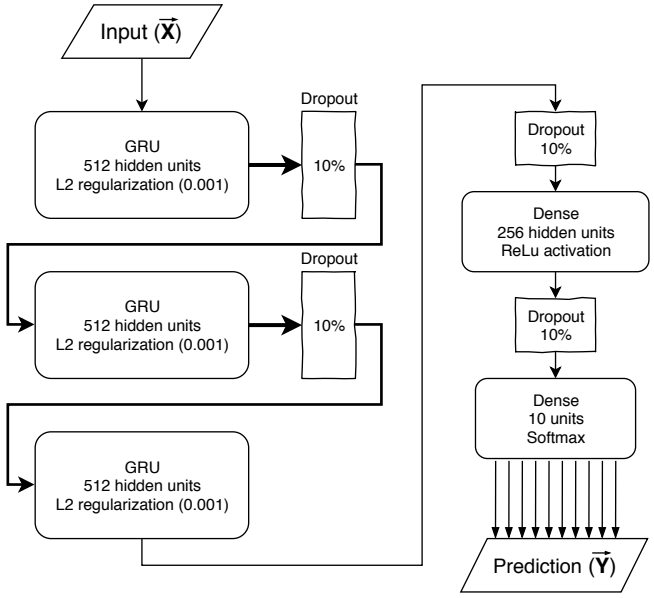


Figure 7: Dataset Recognizer Deep Neural Network model

Variations In addition to the main model stated above, we tried some other variations, such as changing the regularization hyper-parameters, using less layers, increasing/decreasing the number of hidden units in each layer, and even trying LSTMs instead of GRUs, but found that GRUs did indeed perform slighly better and trained significantly faster.

4 Implementation

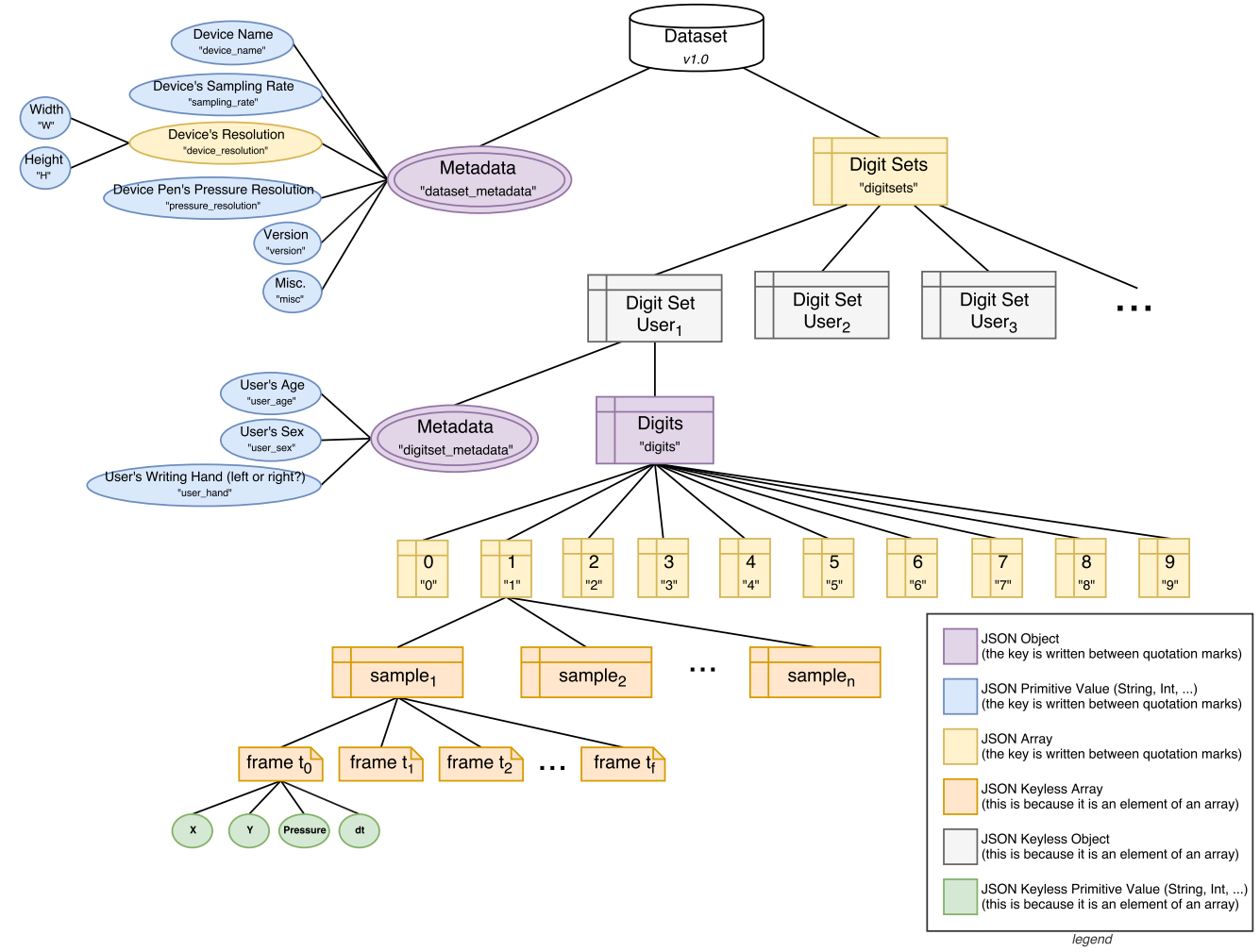
Conceptually, this project can be seen as consisting of two phases:

- **Dataset Generation (Data Collection)** which includes all tasks that result in the the creation of a dataset suitable for the purpose of this project.
- **Research and Analysis** in which we process the generated dataset, train our models, evaluate them, and analyze the properties of this approach.

In this section, we begin by describing the digital format of our dataset, then jump into the implementation details and design of the data collection system we developed to collect the data we need for this dataset. Next, we describe the properties of the final dataset we were able to generate and use for this project. Near the end we describe in detail the software environment and workspace that was used during the research phase. This includes the tools used, abstraction developed to simplify development and research, and the folder and file organization and hierarchy. Finally we end this section explaining the Prediction tool we developed for demonstration purposes.

4.1 Data Format

Our dataset is stored as a JSON[1] file. It's hierarchy and layout is documented and described in figure 8 below. Optionally to keep individual file sizes small, we can opt to save each digit set as a separate JSON file, and then merge them into a unified dataset file if needed.



Device Name *(String)*	Name of the device used
Device's Sampling Rate *(Int)*	The sampling rate of the device measured in "Points per Second" *Valid values: Natural Numbers > 0*
Device Pen's Pressure Resolution *(Int)*	How many distinct and discrete pressure levels the pen can measure *Valid values: Natural Numbers > 0*
Version. *(String)*	The data version of this dataset *Format vX.Y*
Misc. *(String)*	Any other optional note (Not to be used by the software)

W *(Int)*	The virtual width of the tablet's resolution. *Valid values: integers > 0*
H *(Int)*	The virtual height of the tablet's resolution. *Valid values: integers > 0*

User's Age *(Int)*	The age of the person who generated this set of samples *Valid values: integers > 0*
User's Sex *(String)*	The sex of the person who generated this set of samples *Valid values: "male", "female"*
User's Hand *(String)*	The hand that was used to generate this set of samples *Valid values: "left", "right"*

X *(Int)*	The absolute X coordinate where the pen's tip is touching the tablet *Valid values: integers in the range [0, W)*
Y *(Int)*	The absolute Y coordinate where the pen's tip is touching the tablet *Valid values: integers in the range [0, H)*
Pressure *(Int)*	The pressure with which the pen is being pressed onto the tablet *Valid values: integers in the range [0, "pressure_resolution")*
dt *(Float32)*	The elapsed time in ms between two consecutive frames *Valid values: 32 bit floating point numbers >0*

Description of the values we are storing in our dataset

Figure 8: Data v1.0 Specification

4.2 Data Collection System

Clarification

Sometimes we tend to use some terms interchangeably such as drawing tablet and graphics tablet, when referring to our device. For the sake of clarity we list some terms we have used that might

be ambiguous:

- **Wacom:** Even though we only used a Wacom tablet, we use the terms drawing tablet more often to keep the concepts more generic and applicable to other brands and models that could be used.
- **Drawing Pad:** The active area on the drawing/graphics tablet
- **Device:** we use this to refer to the drawing tablet itself
- **Pen:** we use this to refer to the stylus of the drawing tablet

4.2.1 Data Collection Mechanism Specifications

We utilize **Wacom®**[2] drawing tablets as our input devices. Specifically the device we used to generate our current dataset is described as follows:

- **Brand:** Wacom®
- **Tablet Model:** Volito2 (4×5) CTF-420/G tablet.
- **Resolution** $(w \times h)$**:** 5105×3713
- **X Values Range:** $[0, 5104]$
- **Y Values Range:** $[0, 3712]$
- **Stylus Model:** FP-410-0G
- **Stylus Pressure Resolution:** 512
- **Stylus Pressure Values Range:** $[0, 511]$
- **Power Specification:** $DC\ 5V\ 0.14A$
- **Connection Type:** USB

4.2.2 Data Collection System Software Overview

Software Environment Our software was developed using the Python programming language (v2.7) in a Linux environment (Ubuntu 17.04 to be specific). The important thing to note is that although coding in Python offers great portability of our code (we can theoretically run this on any machine that implements the full Python environment), our code requires Linux to work as we are using the Linux kernel's **evdev** driver.

The GUI itself was developed using the **Qt4** framework and the canvas that shows displays a live preview of what the user is writing was made using **pygame**. Basically we do the graphic processing in pygame, and copy over the graphics buffer to a Qt canvas widget embedded in the *drawing window*.

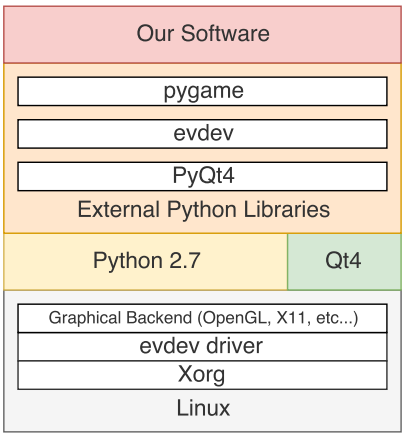


Figure 9: Software Environment Overview

We tried our best to keep the code as modular as possible so it shouldn't be difficult to port the code over to a non Linux system so long as the rest of the dependencies (python 2.7, pygame, qt4, and their respective python libraries) are available. You should only need to rewrite the *"driver"* class that interfaces with the drawing tablet.

System Setup The setup is simple; One Wacom graphics tablet connected to (via USB) an Ubuntu powered computer running our software.

The required software dependencies can be installed straight from your system's package manager (if running Ubuntu).

A configuration file is available and should be modified with the desired parameters. This file is used by our program to determine the settings it should use during execution (the configuration and settings are documented later in this document).

The software should work out of the box, simply run and it should load your specified configuration. Log messages are displayed in the console, and they are the only true indicator of any fault or error.

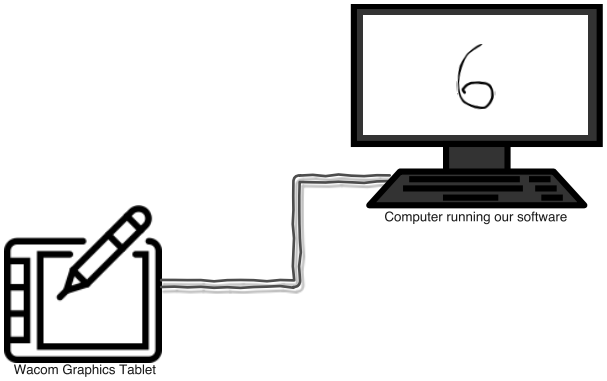


Figure 10: Visualization of our System

Functional Overview We designed our program to separate as much as we could the graphical front-end from the back-end. The GUI is responsible for providing the user interface, initiating the program's phases, and providing a live preview of the currently read data (in the case of the drawing phase). The back-end manages our data structures and interfacing with the drawing tablet (via the evdev driver). The *Server* class of the back-end is also responsible for managing and updating the program state.

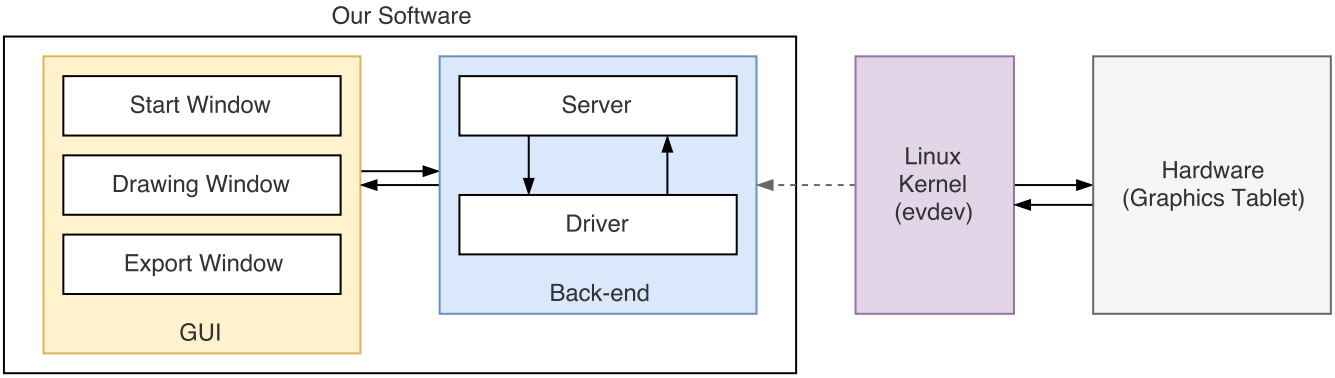


Figure 11: Data Collection Program Functional Overview

Code Structure As shown in figure [12] above, our software is divided into a set of files and classes. The entry point to our program is **main.py** which simply sets up our environment, loads any user configurations and starts our GUI Qt4 Front-end with StartWindow.

globals.py contains the variables that are shared among all the files, such as the instance of our **Server** class *(we only have one instance of the Server class in our program)*

The **Driver** class works as an interface between our program and the Wacom driver *(via the evdev driver)* and runs in a separate thread parallel to our program.

We have some "static" classes that we use to define things like configurations, and constants.

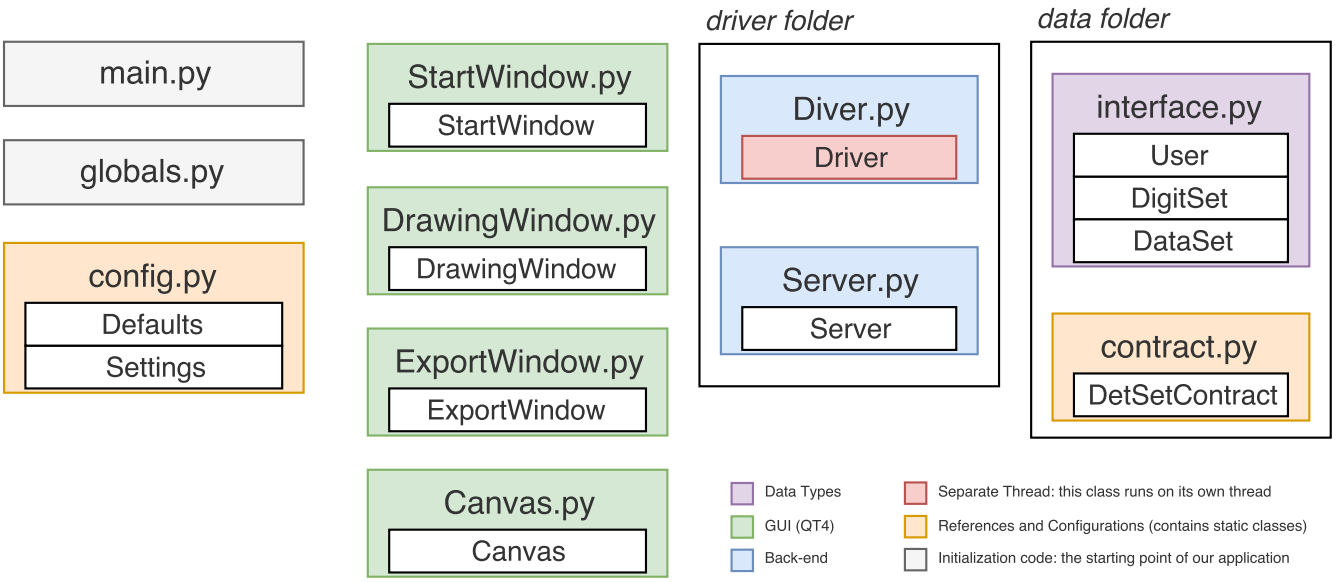


Figure 12: Structure of Project's Code

4.2.3 User Interface

Start Window This is the first window in the application. From this window you can either:

- start collecting data by first filling out the form with the new user's information and then clicking the start button which will open the drawing window and hide this window until the data collection (drawing procedure) has completed
- or export the data that has been collected so far into a dataset file by clicking the menu button in the window's frame and selecting on the "export" option

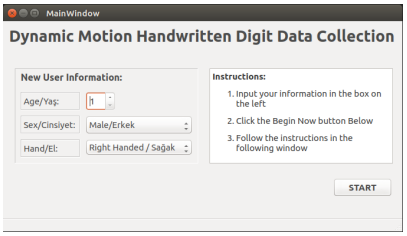


Figure 13: Start Window Screenshot

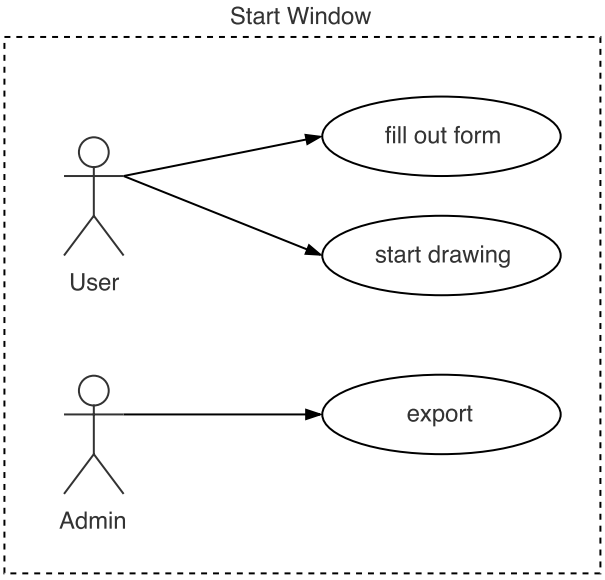


Figure 14: Start Window Use Case Diagram

Drawing Window This window is where the user performs the drawing process.
On creation, the window registers itself with the Server object (defined in **global.py**) and register's the UI buttons with the class's functions. The class also creates a Qt Timer that will repaint the UI at a fixed interval as defined in the configuration. A DrawingWindow instance also contains an instance of a Canvas object.

When this object's repaint object is called, it will first call the draw function on its Canvas object passing in the driver's temporary buffer as an argument, then the object will update its indicators with the last data our Driver thread has received from the Wacom tablet. As for the other indicators related to count and phase, the Server class handles updating them with the correct value.

Using this window, a user will be asked to write down (using the drawing tablet) the digit shown on the screen. This window also contains a canvas to show in real time what the user has written so far. The canvas basically draws the points received from the tablet and connects them with a line of varying thickness. The line thickness is proportional to the stylus pressure value of the next point that is being drawn.

The user is expected to draw each of the 10 digits (0, 1, 2, 3, 4, 5, 6, 7, 8, 9) a specified number of times (as set in the program configuration). These samples constitute a "Digit Set." After all digits have been written, the program will save the digit set to a file in the configured export directory and append this digit set to a data set structure in memory. The naming convention for these saved digit files is:

*HH.MM_dd.mm.yy*_digitset.json
*Where *HH* is the current time in hours, *MM* is the current time in Minutes, *dd* is the current date, *mm* is the current month, and *yy* is the current year
Unless explicitly told to, the program will not export its in-memory dataset to a file.

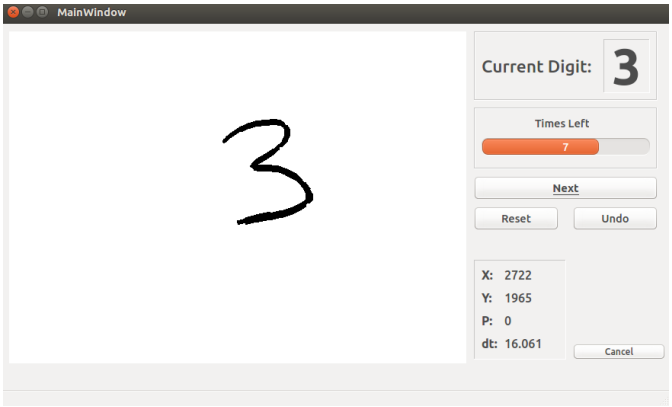


Figure 15: Screenshot of Drawing Window

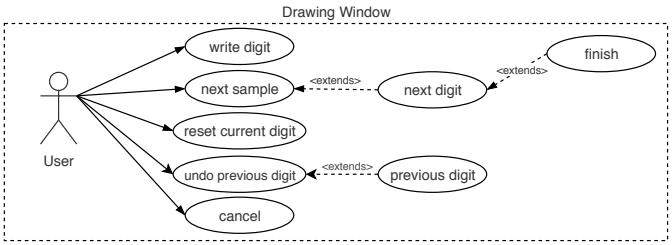


Figure 16: Drawing Window Use Case Diagram

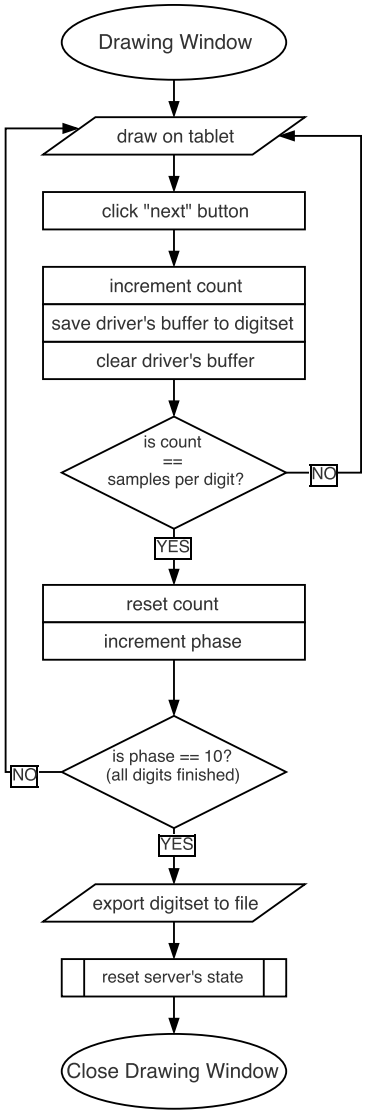


Figure 17: Flowchart of Drawing process

Canvas The canvas class is a Qt widget that wraps inside it a pygame Surface object that provides us with a graphical buffer in the background for us to draw in.

The object's draw method take in a buffer (list) of data points (X, Y, P) and draws each point (scaled to size) on a canvas of fixed dimensions, and connects these dots with lines. The thickness of the dots and lines is proportional to the P value of each point.

When the Canvas object's draw repaint event is called, it will copy its raw graphics buffer and paint its corresponding QtCanvas widget in the UI with this buffer. The reason we do this is because the pygame buffer it contains runs in the background, so all the drawing functionality is done via pygame in the background and then the final result is copied and drawn in the Qt UI manually. This allows more

easier control with drawing since we get to use pygame's graphics capabilities.

Exporting Window The export window's functionality is quite simple, when you open the window, the first thing that the program does is check whether a datset file already exists *(remember that the dataset filename is specified in the configuration)*.

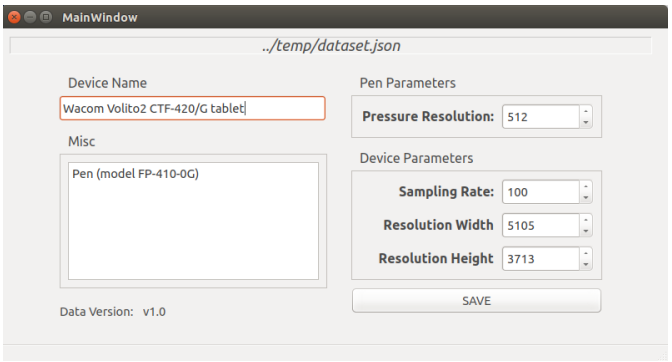


Figure 18: Screenshot of Export Window

If a file already exists, then the pre-existing dataset is loaded into memory and merged with the program's currently loaded dataset. Then the export window's input fields are filled with the loaded datasets metadata.

If no file was found then the form fields will all default to being empty.

Next the admin edits/fills in the desired metadata and then clicks save, which instructs the program to either create a new dataset file or overwrite a previously existing file with the dataset object in memory *(which consists of the data collected during the lifetime of the program merged with the data loaded from a pre-existing dataset)*. After the dataset was successfully exported, the dataset object in memory is deleted and a new empty one is created.

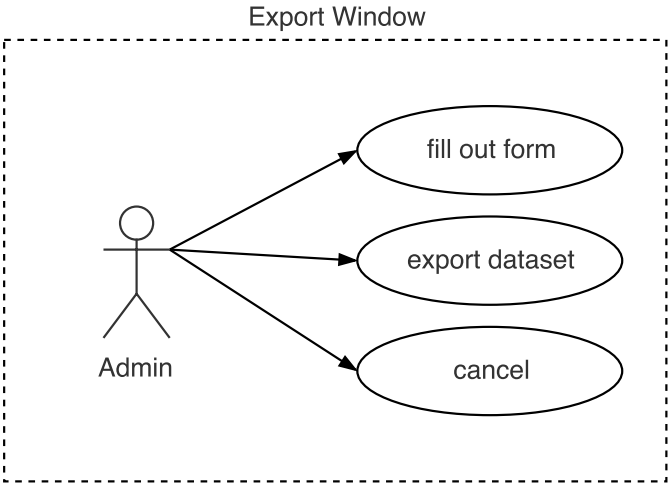


Figure 19: Export Window Use Case Diagram

The program checks whether a dataset file already exists, and if it does it loads its metadata into the form on screen. The system admin is expected to fill out the metadata appropriately and then click on the save button, which will either create a new dataset JSON file and save our in-memory dataset in it, or if a datset file already exists, it will load that dataset into memory, merge the newly loaded data-set with the in-memory dataset and then write back the final dataset into the file overwriting the old version with the newly merged one.

4.2.4 Program Flow

Here we designed two abstract roles: Server and Driver.

Server The Server is the backbone of the entire program. It is also where our data is stored, and where operations on that data are initiated.

Driver The Driver class, as the name suggests, works as the interface between the Server and evdev driver. It runs on a separate thread (using python's threading class) and continuously polls for new events from the Wacom tablet (via the evdev driver) and stores the relevant events (X, Y, Pressure) in a

temporary local buffer. The Server class is responsible for saving the data in that buffer and clearing it as needed. The Driver's role is minimal, as the Server class handles the program logic.

Wacom Tablet evdev Events

- **ABS_X:** *New* Stylus X coordinates
- **ABS_Y:** *New* Stylus Y coordinates
- **ABS_P:** *New* Stylus Pressure value
- **SYN_REPORT:** A synchronization event that comes at a (almost) fixed interval.

Note our emphasis on the word ***"New"*** as the Wacom tablet will not inform us if the values have not changed and will only send events when a new value has been read. Hence we always keep track of the last value read and use the Synchronization event as a signal to package the last read X, Y, and Pressure values into a tuple and append them to our buffer.

Note:

I personally didn't find a standard documentation on which events the Wacom tablet produces and it's button mapping with evdev, but one quick and easy way to gain a good intuition for which events correspond to what actions is to use a tool called **evemu-record** that comes as part of the **evemu-tools** package which can be installed via the *apt* package manager in Ubuntu

Timing: Since **Data v1.0**, the dataset also includes a timing feature for every sample in addition to the X, Y, P features. The evdev driver sends a timestamp with every event, and this program simply extracts thaese values, and measures the elapsed time between *SYN_REPORT* events in milliseconds rounded to 3 decimal points and saves it.

UI Updating As shown in figure [20], the *Driver* class continuously polls for events from the Wacom tablet and saves relevant data to a temporary buffer. The DrawingWindow sets up a Qt timer that guarantees that the UI will be updated (repaint event) at a fixed interval. Whenever a repaint event comes, the Drawing window will first tell its Canvas instance to update itself, then it will update the X, Y, P, dt indicators with the last received event the Driver thread has received. The Canvas object performs its draw method in a hidden pygame graphics buffer, which it will then copy, then convert to a QtImage (bitmap image) object and finally paint it's QtCanvas widget with the created QtImage.

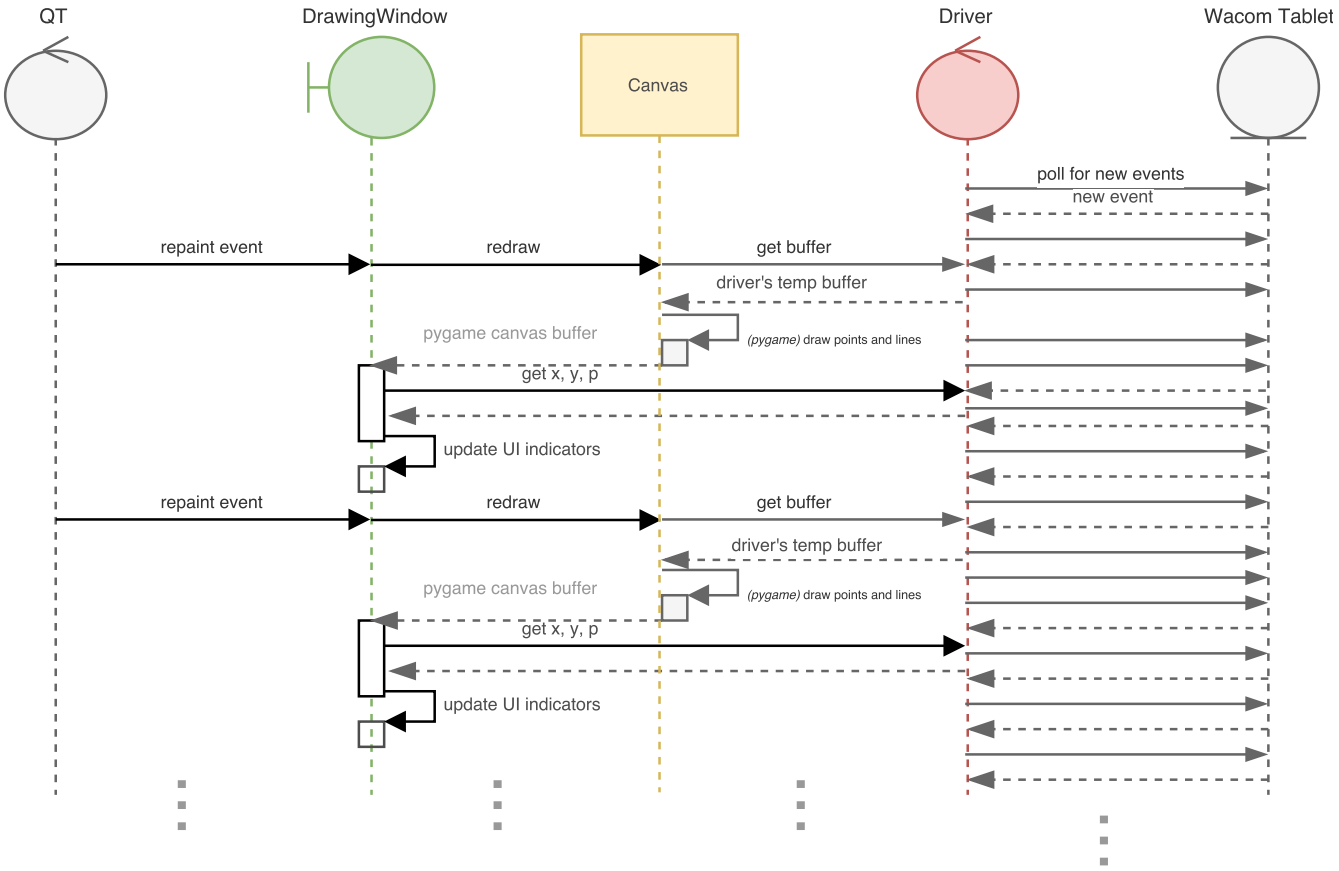


Figure 20: Sequence Diagram showing how the GUI updates itself while the *Driver* class continuously deals with the Wacom tablet.

4.3 Collected Data Set

The final dataset that we collected and used in our research has the following properties:

- **Number of People:** 36
- **Number of Samples per Digit each person generated:** 10
- **Total number of samples:** 3600
- **Average and standard deviation of number of frames per sample:** 42 ($\pm$14)

The population sample of people that contributed to the dataset consisted mainly of Turkish university students between the ages 20 to 30 years old.

4.4 Research Environment

4.4.1 Software Requirements

Since the data collection system was developed in a Linux operating system, we chose to continue to work within the same environment. Unlike the data collection system, we chose to use Python 3 as our development language of choice.

For implementing the neural networks we chose to use one of the popular frameworks, specifically, we used Keras running on the TensorFlow backend. Keras appeared to be a suitable choice due to the simplicity of its syntax and the diversity of modern neural network models it provides built-in. In order for TensorFlow to work, we needed a computer with a Nvidia CUDA capable GPU, running a compatible operating system with the CUDA toolkit and CuDNN drivers installed.

To assist with preprocessing and preparing the raw dataset, we used modules from a range of popular scientific computation libraries for python. Mostly, we benefited from scikit-learn for dataset splitting and stratifying, scipy for spline interpolation, matplotlib for generating useful plots and graphics, and numpy storing the dataset and applying quick vectorized transformations to it. Pandas was occasionally used to format some results in nicely displayed tables when using tools like Jupyter Notebook.

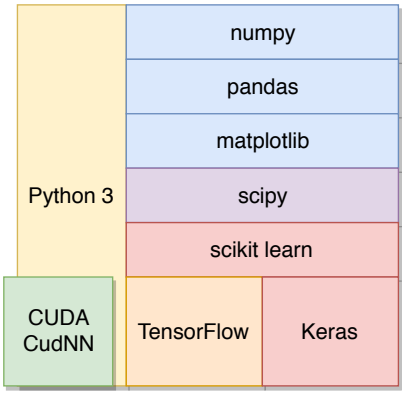


Figure 21: Research Environment Software Requirements

4.4.2 Workspace

Let us define the research workspace as the set of files, abstractions, software modules, and tools we made and interfaced with to achieve our research goals.

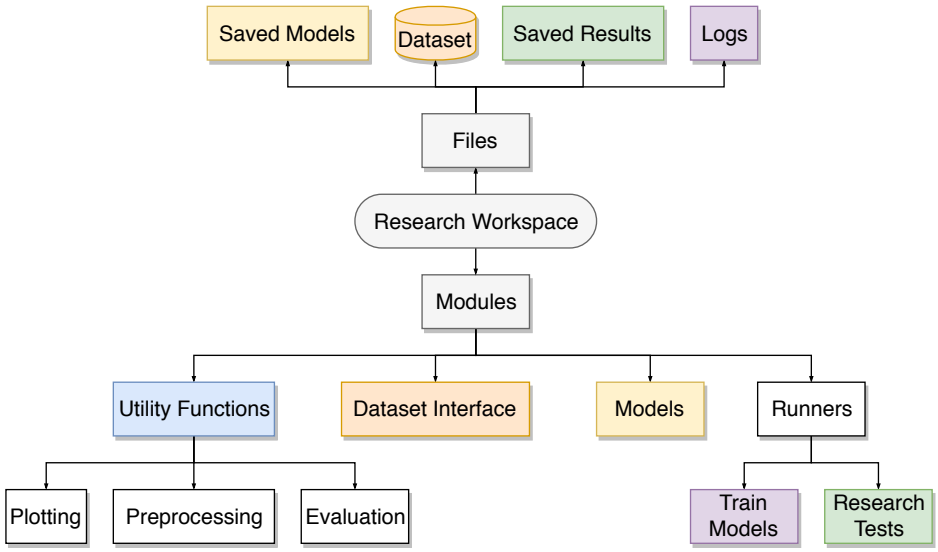


Figure 22: Research Environment Workspace

The workspace files contain the dataset raw files, exported Keras models, log file, generated graphics, plots, and other figures, and basically it works as both, the data dump for all information and files our modules and scripts output, and as an source of input to load saved data and other necessary resources from.

As shown in figure 22, the workspace *Modules* consist of the following

- **Utility Functions:** A set of modules and functions that provide generic operations that can be used and reused in multiple places.

- **Dataset Interface:** These are the set of classes and modules that directly interface with the raw dataset files. They perform operations such as loading data and performing batch transformations on loaded data.
- **Models:** These are classes that define our recognizer models. These classes handle operations such as setting up the model, exporting the models current state, saving a summary of the model.
- **Runners:** These are the *"main"* scripts, where each script achieves a certain task by utilizing the aforementioned models and classes. There are two types of runners that are relevant:

  Train Models: These scripts load a dataset, train a specific recognizer model on them, evaluate the model, and saves the trained model and its results.

  Research Tests: These scripts perform a variety of tasks that aim at answering specific research questions, such as testing the durability of a certain trained model to noise or occlusions. Or determining how many new points should be re-sampled when performing spline interpolation and re-sampling. Some of these scripts train a new model from scratch, while others load a model that was previously trained using the **Train Model** scripts, depending on the situation.

4.4.3 Abstractions

Model Template Interface Class To automate the process of creating and testing new models, certain abstractions were made such as a model template interface class that implements all the default setup procedures and leaves the model definition to the implementing class.

Automated Documentation Keras provides certain utilities that allow the automatic backup of models while they train. The tensorflow backend also comes with a handy visualization tool called TensorBoard that visualizes the weights and accuracy of a model as it trains. We used these two tools, and additionally developed our own semi-automated documentation system that records all the preprocessing operations performed on the dataset and the model design and its parameters, recording them allowing us to keep track of the different setups that we tried.

4.5 Training and Evaluating the Model

4.5.1 Dataset Train/Validation/Test split

A very important aspect of training neural networks is correct splitting and shuffling of the data. There are various approaches to splitting datasets. Some people opt to split their datasets permanently, while others prefer to split them randomly upon initialization. We chose the random approach since our dataset is relatively small, choosing any constant way of dividing the dataset could introduce some bias towards certain styles of writing. To be specific we split the data randomly by users, to ensure that the set of samples produced by one user are uniquely contained in only one of the three subsets (train, validation, and test sets). This way a neural network does not experience the style of writing of any of the users whose data are contained in our validation and test set ensuring a more realistic evaluation of the model's performance. For training and evaluating our model, we split the dataset by the following proportions:

- train set: 53.33%
- validation set: 26.67%
- test set: 20%

4.5.2 Training the Recognizer

For training our model, we used the Nesterov Adam optimizer[5] with the following parameters:

- learning rate: 0.001
- schedule decay: 0.15
- batch size: 500
- β_1: 0.9
- β_2: 0.999
- number of epochs: 30

4.5.3 Evaluating Network

Once we finished training the recognizer, we evaluate it using the following tools and metrics:

- Observation of the validation and categorical sets' accuracy at the end of each epoch, and how many epochs it takes for these accuracies to stabilize (this method relies more on human intuition and is helpful for tweaking hyper-parameters).

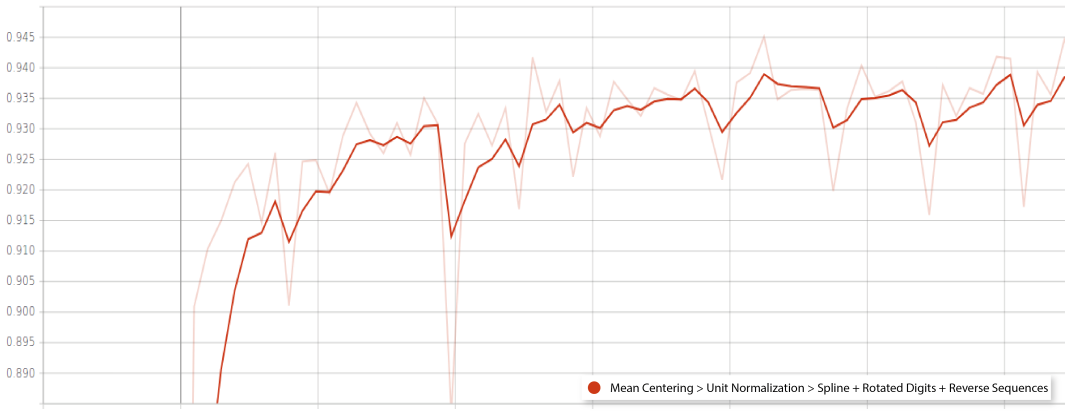


Figure 23: Categorical and Validation Sets' accuracy during training

- Test Set accuracy
- Recall, Precision, F1 Score. We noted that the recall, precision, F1 score, and accuracy were mostly equivalent, and differed by a small fraction.
- Confusion Matrix: helpful for visualizing the network's recall and precision and gives you and idea of which digits are recognized well and which are not. An example confusion matrix evaluated on the test set of the model described in 3.3.1 is shown in figure 24. It is important to note that the confusion matrix values are dependent on the random distribution of the train, validation, and test sets, meaning that if we were to rerun the script from the beginning, this would lead to a different subset of samples and hence a slightly different confusion matrix. The small size of the dataset causes such evaluations such as confusion matrices to be sensitive to the random initialization of the subsets.

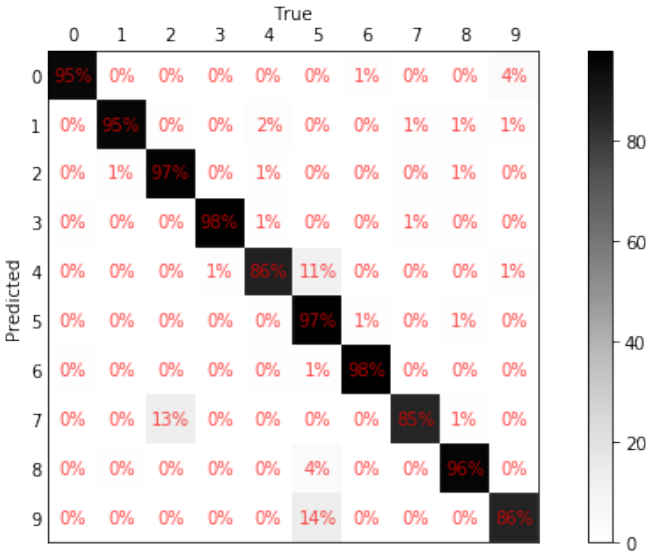


Figure 24: Example Confusion Matrix

- Cross Validation: Not always used due to the long processing time required, but occasionally can be a good tool to evaluate the reliability of the previously reported scores.

4.6 Research Tests

In addition to successfully designing and training a digit recognizer, we were interested in exploring the less trivial properties of the approach proposed in this thesis. To do so, we set up a few tests that helped us assess some of these properties. We tested for are:

- The effect the preprocessing and augmentation operations.
- The effect of occlusions on the recognizer's ability to correctly categorize a given input.
- The effect of adding noise to the input data on the recognizer's ability to correctly categorize a given input.
- The optimal number of points to sample from the interpolated spline of each digit.

The focus of this section is to describe how our tests were performed. The results of these tests are shown and discussed in 5.

4.6.1 Effect of Preprocessing Operations

Human intuition can be a very creative and fundamental tool in any project, but we are very prone to making premature assumptions, biases, and errors. Therefore we must verify the decisions we make preferably through a quantitative analysis. In the scope of this project, the one of the most significant aspects of our approach is the set of preprocessing operation we applied to the dataset. These operations and the intuition behind them were discussed in section 3.2.1. In this section we aim to visualize and discuss the effect of these operations using an empirical approach.

The first thing to note here, is that *spline interpolation and re-sampling* is the always performs as it solves the variant sequence length problem, and removing it requires a change in the recognizer

model designed and therefore we will face difficulty isolating the effect of the model differences from the preprocessing operations. Although this is a potential topic for future research.

To evaluate the effect of the preprocessing operations, we trained a new model for each combination of preprocessing and augmentation operations, evaluated the test set accuracy after training, and plotted the validation set accuracy during trained. (note: the validation set does not contribute to the model training, and only used as a performance metric)

Effect of Preprocessing Operations without Dataset Augmentation As depicted in figure 25, applying *Mean Centering* significantly improves the performance of our recognizer, while *Unit Normalization* seems to slightly degrade it. We might be able to conclude that unit normalization strips the data from the intrinsic scale information of the data, which intuitively makes our model generalize better and even regularizes it such that it doesn't overfit on our current dataset.

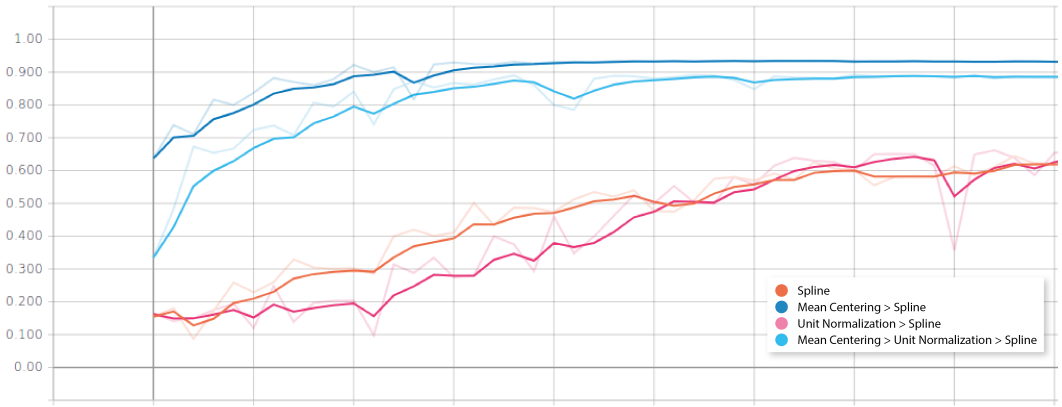


Figure 25: Training model with different preprocessing operation combinations

Effect of Data Augmentation Operations Data augmentation is fundamentally a solution to compensate for the small size of our dataset, and additionally also works to assist the recognizer model in learning patterns that are more invariant to certain transformations (such as rotating the digit for different angles with the hopes that the recognizer will learn to categorize digits despite their orientation).

First we compared augmenting the dataset with just the reverse sequences or rotated digits as shown in figure 26. The first thing to notice is that augmenting the dataset with *Rotated Digits* significantly improves the validation accuracy of our model and the speed at which it plateaus (starting with $\sim 80\%$ accuracy and quickly jumping to 92% by the 5th epoch), while *Reverse Sequence* augmentation has a minimal effect of increasing the accuracy by only 5%. Though it is important to note that *Reverse Sequence* augmentation produces a set of digits that cover certain writing styles that might not be represented in the original dataset, such as certain people choosing to write their *0*s from anti-clockwise. This augmentation is mostly useful for certain digits such as *0, 1, 6, 8,* and *9* that can be drawn in inverse ways.

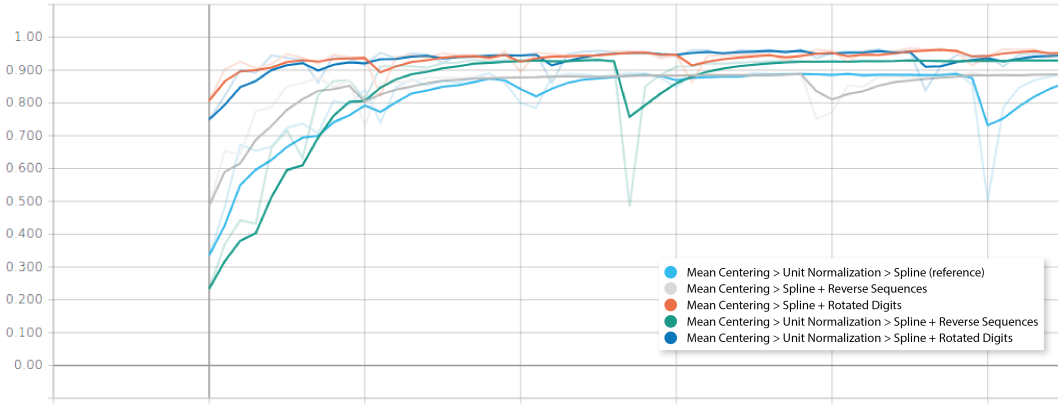


Figure 26: Comparison of augmenting our dataset with Reverse Sequences vs Rotated Digits

Next we look at the performance of using both augmentations instead of just one of them (depicted in figure 27). At this point we see that the performance difference between using mean centering with and without unit normalization minimizes to an insignificant proportion. Of course our choice remains to use unit normalization which potentially eliminates any possible future variations in scale (digits drawn very small and those drawn very big numbers appear equal in scale).

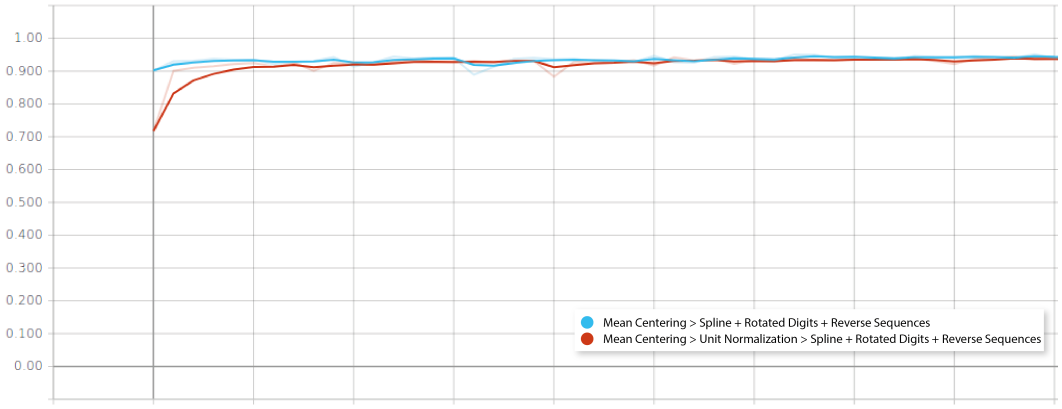


Figure 27: Performance of augmenting our dataset with both Reverse Sequences and Rotated Digits

Summary The best validation accuracy and the test accuracy of the highest performing data preparation combinations are listed in table 1 and their plots while training are shown in 28. The important thing to not is that although the final combination of applying all preprocessing and augmentation operations does not have the highest validation accuracy, it has the highest test accuracy and a good balance between the two, implying that with this combination, the model has learnt to generalize better. The combinations that lead to the highest validation accuracies, seem to do so at the cost of the test set accuracy, from which we can conclude that they lead to a more overfit model.

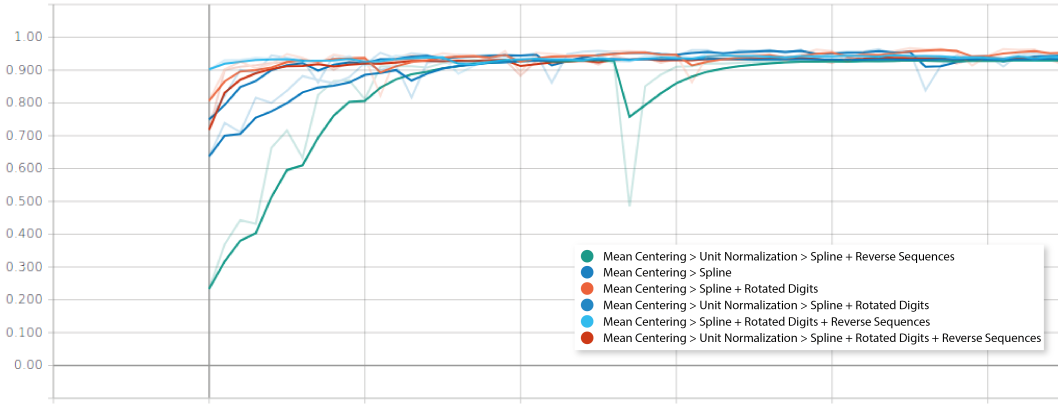


Figure 28: Validation accuracy plot of the most effective data preparation combinations discussed

Preprocessing Operation(s)	Augmentation Operation(s)	Max Validation Accuracy	Test Accuracy
Mean Centering → Spline	None	93.5%	92.57%
Mean Centering → Spline	Rotate Digits	96%	92.59%
Mean Centering → Unit Normalization → Spline	Reverse Sequences	92.9%	94.43%
Mean Centering → Unit Normalization → Spline	Rotate Digits	96%	91.81%
Mean Centering → Spline	Rotate Digits Reverse Digits	95%	94.6%
Mean Centering → Unit Normalization → Spline	Rotate Digits Reverse Digits	94.5%	95.1%

Table 1: Validation and Test Accuracy of top data preparation combinations

4.6.2 Occlusions

For this test, we randomly dropped a chosen percentage of the frames of each sample's sequence, and evaluated the average categorization accuracy of a previously trained recognizer model. We ran this test multiple times, increasing the percentage of frames dropped in each run, and plotted the change in accuracy against the increase in the percentage of data dropped. We started without dropping any of the data (0% occlusions) and iteratively increased the percentage of the frames we randomly dropped in increments of 10% up to 100%. The frames are dropped before any preprocessing operation is applied. We also limited the dropping function so that the occluded digit has at least enough frames (4 frames) for the digit to be successfully interpolated and re-sampled (see spline interpolation and re-sampling in section 3.2.1). As a result, at 100% occlusion, all samples are left with 4 frames randomly selected from the original sequence.

As shown in figure 29 our approach leads to a recognizer that is quite robust against occlusions, and can handle dropping up to 50% of the input frames before the accuracy begins to steep downwards. This could imply that the model does not need many samples to make a good prediction, resulting in less memory usage than standard approaches.

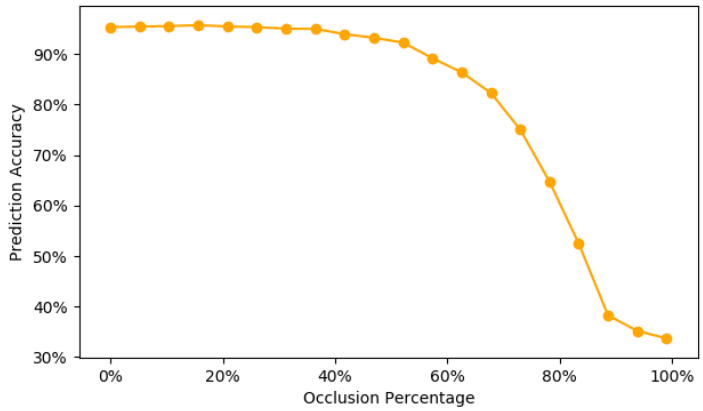


Figure 29: Accuracy Drop vs Percentage of Data Dropped (Occlusions)

4.6.3 Noise

For this test, we randomly added noise sampled from a normal distribution (visualized in figure 30) to the frames in the dataset, and evaluated the average categorization accuracy of a previously trained recognizer model. The noise is added before any preprocessing operation is performed. Each test is ran for multiple iterations where the mean of the normal distribution is fixed. In each iteration we increased the standard deviation of the normal distribution from which the noise is sampled. We ran the test multiple times, changing the mean of the normal distribution each time. We then plotted each test run and the change in accuracy with the increase of the standard deviation.

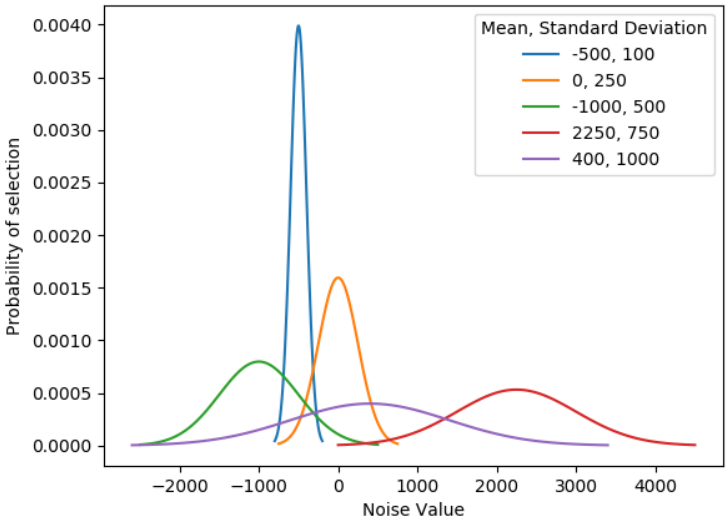


Figure 30: Visualization of multiple normal distributions with different standard deviations and a mean of 0

Intuitively, the mean of the noise's normal distribution represents the offset from each point where noise data will be added, and the standard deviation represents the range of values that the noise function

can randomly sample from. A graphical visualization of the ranges around each frame where the frame can be shifted to after adding noise for 3 different distributions are shown in figure 31

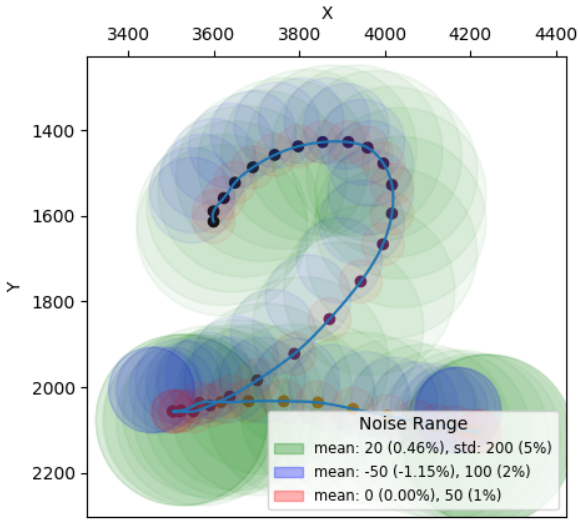


Figure 31: Visualization of the probable range where a noisy point can occur using different normal distributions with different standard deviations and means. The percentage value represents the percentage of the drawing pad's area that the values are equivalent too.

Looking at the resulting plot (figure 32) we observe that that the mean of the Gaussian noise, which represents the offset from each point where noise data will be added, has little to no effect on the accuracy of the model. This can be attributed to the *Mean Centering* preprocessing operation which essentially eliminates any uniform offset in the data. As for the increase in standard deviation, as expected the accuracy drops pretty steeply.

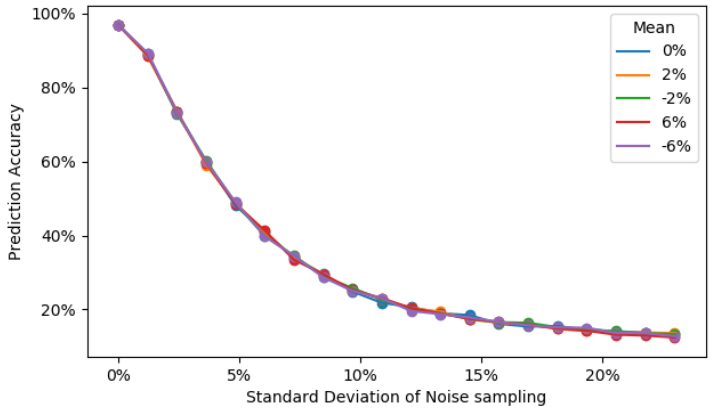


Figure 32: Accuracy Drop vs Strength of Noise Added measured as the percentage of the drawing area that it covers

4.6.4 Number of Samples for Spline Interpolation and Re-sampling

Spline interpolation and re-sampling is an intrinsic part of the approach proposed in this thesis, but the question that remains is how many frames should be sampled from the resulting spline. We set up a test that trains the same recognizer model on different versions of the dataset, where in each version we changed the number of frames to sample from the resulting spline of each digit. As a heuristic we first calculated the average frames per sample in the dataset, and used that as a starting point for our search. All other preprocessing and augmentation operations are still applied in this test. We found that the optimal number of new frames to sample from the resulting spline, is moderately close to, or lower than the average frames per sample of our dataset. Too low a number, led to a drop in accuracy due since we would have lost a lot of the digit's original structure, and too high a number leads to redundant complexity meaning we would need a more complex model that is able to learn from very long time sequences. We chose to sample 50 frames from the spline, which is just higher than the average frames per sample (42), but not too high that we add redundant complexity to our data.

4.7 Live Data Prediction Tool

For the purpose of demonstration, we developed a an application (shown in figure 33) that performs live predictions as a user writes on the drawing pad. The app is built on top of the Data Collection system's code, with some modifications to suit its purpose. In addition to the libraries used by the Data Collection system's code, the prediction tool uses Keras, and the modules we wrote that provide the data interfacing, preprocessing and augmentation functionalities previously discussed in section 4.4.2.

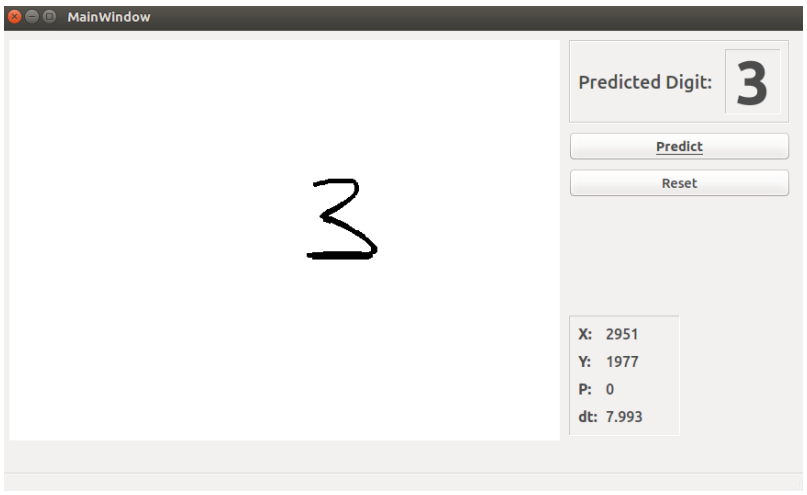


Figure 33: Screenshot of Prediction Tool User Interface

5 Conclusion & Impact

So far we have introduced a new approach to handwritten character recognition, developed a handwritten digit recognizer as a proof of concept, generated a small dataset appropriate for this recognizer model, trained the recognizer on this dataset, evaluated the recognizer and its performance, and analysed the properties of our approach.

We have shown that this approach is handles occlusions well and requires less data points to accurately categorize a given input, but is sensitive to noise. We also showed that after applying the described preprocessing and augmentation operations, the model reaches its maximum accuracy quite fast.

These properties make this approach a suitable fit for both large and small datasets, and for more demanding real time tasks. The small data size requirement of the model makes it feasible for use even in smaller less capable devices.

The research we have done here can be seen as an introduction to a field of handwritten character recognition that has good potential for further exploration and even use in practical scenarios such as Micrososft Windows' handwriting input tool.

6 Discussion & Future Work

6.1 Pitfalls

There are a few weak points that we would like to address in this project:

- The dataset used so far is quite small, and also suffer from a statistical sample bias; the demographic that constitutes the dataset mostly consists of young (age range 20-30) Turkish university students studying engineering. We can not say for certain whether a correlation exists between this demographic and their style of writing. Of course each student comes from a different background and different subcultures, but we can sa with confidence that this sample size is not confidently sufficient to generate a model generic enough to work with handwritten digits of different people from different backgrounds and demographics. This project can be seen more as a proof of concept and a motivation for future generations to continue research in this field picking up from where we left off.
- Another possible pitfall can be seen as the accuracy and the quality of the evaluation metrics and tests used throughout this project. Given more time, resources, and motivation a much more thorough and controlled set of experiments could be conducted with much more easily repeated results.
- The neural network designed in this project can also be criticized as quite naive and primitive. We believe a much more efficient model can be designed.
- Due to the randomness of our datasets splitting, each time we reinitialize a model to train and evaluate it, we end up with a different division of our data. Although this is good in the short run, as it prevents our models from becoming biased towards a certain subset of writing styles, in the long run information about the entire dataset leaks into the model as the researcher. The reason this happens is that as the researcher continuously optimizes and tweaks the recognizer, each time rerunning the model, and evaluating on a different subset of the data, eventually the researcher will end up evaluating on the entire dataset, slowly optimizing the model's parameters to overfit on the entire dataset. This is a common pitfall with machine learning in general, and can be possibly avoided with a constant splitting of the dataset (which is only feasible with a bigger dataset), human intuition, and carefully applying good machine learning practices such as avoiding using the test subset when optimizing the model, and only use it for a final quantitative evaluation.

6.2 Future Work

6.2.1 Bigger Dataset

Generating a much bigger dataset with a more variant demographic would be very beneficial for others who might like to dabble with this type of data and produce interesting and more efficient models and use cases.

6.2.2 Use Additional Features

The in addition to (X, Y) coordinates, the dataset collection system we developed collects pressure information (p) and the elapsed time between consecutive samples (dt) (as shown in figure 8). It would be beneficial to incorporate these features into the proposed approach. We were not able to do due to the lack of data, and adding more features requires a more complex model and more data for the recognizer to successfully learn useful patterns and abstractions.

6.2.3 Improve the Augmentation and Preprocessing Operations proposed

The approach described in this thesis is novel and therefore there is room for improvement in many of its aspects, from the model, to the dataset preparation. A possible topic to invest in is to improve the data preprocessing and augmentation operations, such as possible augmenting more rotations of the original data, or choosing more

6.2.4 More Dataset Augmentations

An interesting idea would be to try some more types of data augmentations that might improve the recognizer model, such as augmenting the training dataset with noisy versions of the original data, which could provide more incentive for the model to find more abstract patterns that are less sensitive to noise and variance in the writing styles.

6.2.5 Extend to alphabetic letters and possibly other symbols too

This thesis focused on latin numerals, but there is no reason why the same concepts could be easily extended to work on other alphabets an symbol sets.

6.2.6 Look for correlations between gender, age and handwriting style

A more complicated research idea would be to study the correlations between the age and gender of users and certain features of the handwritten digits they used.

6.2.7 Faster Online (Live) Handwritten User Input

Currently, the Windows operating system provides a handwriting input tool, that allows users with a graphics tablet to write characters and the software processes the input and transforms it into their digitally equivalent text, allowing users to stick to handwriting in the digital world as an alternative to using a keyboard. The model and data described in this thesis could be used to improve such systems, since the data is much smaller and the recurrent nature of model can be easily trained and used online (online meaning working in real time as the user writes). The software simply needs to add each frame into the recurrent network without resetting the sequence and can continuously provide live predictions. In addition such a model was shown to learn faster and with less data than the conventional models used with static images.

6.2.8 Generative Model

One interesting idea is to use the same dynamic dataset to train a generative model instead of a categorizer. This model in theory would learn how to draw the different characters in a similar way to humans. This model could be implemented in a robot which can utilize its actuators and draw a digit using the sequence of points it generates. The sequential nature of the data itself naturally makes it a suitable fit for producing commands that a robot could implement.

7 References

[1] Javascript Object Notation. URL http://json.org/.

[2] Wacom. URL http://www.wacom.com.

[3] K. Cho, B. van Merrienboer, C. Gulcehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio. Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation. jun 2014. URL http://arxiv.org/abs/1406.1078.

[4] J. Chung, C. Gulcehre, K. Cho, and Y. Bengio. Empirical Evaluation of Gated Recurrent Neural Networks on Sequence Modeling. dec 2014. URL http://arxiv.org/abs/1412.3555.

[5] T. Dozat. Incorporating Nesterov Momentum into Adam. *ICLR Workshop*, (1):2013–2016, 2016. URL http://cs229.stanford.edu/proj2015/054{_}report.pdf.

[6] R. Jozefowicz, W. Zaremba, and I. Sutskever. An Empirical Exploration of Recurrent Network Architectures.

[7] Y. LeCun. The MNIST database of handwritten digits, 1998.

[8] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2323, 1998. ISSN 00189219. doi: 10.1109/5.726791.

[9] Z. C. Lipton, J. Berkowitz, and C. Elkan. A Critical Review of Recurrent Neural Networks for Sequence Learning. *CoRR*, abs/1506.0, 2015. ISSN 9781450330633. doi: 10.1145/2647868.2654889. URL http://arxiv.org/abs/1506.00019.

[10] I. Sutskever, J. Martens, G. Dahl, and G. Hinton. On the importance of initialization and momentum in deep learning. *ICASSP, IEEE International Conference on Acoustics, Speech and Signal Processing - Proceedings*, (2010):8609–8613, 2013. ISSN 15206149. doi: 10.1109/ICASSP.2013.6639346. URL http://www.cs.toronto.edu/{~}fritz/absps/momentum.pdf.

[11] R. Tolosana, R. Vera-Rodriguez, J. Fierrez, A. Morales, and J. Ortega-Garcia. Benchmarking desktop and mobile handwriting across COTS devices: The e-BioSign biometric database. *PLoS ONE*, 12(5):1–17, 2017. ISSN 19326203. doi: 10.1371/journal.pone.0176792. URL https://doi.org/10.1371/journal.pone.0176792.