Escuela
Politécnica
Superior

Recognition of Japanese handwritten characters with Machine learning techniques

Bachelor's degree in Multimedia Engineering

Bachelor's Thesis

Author:
José Vicente Tomás Pérez

Supervisor:
José Manuel Iñesta Quereda

Universitat d'Alacant
Universidad de Alicante

July 2020

Recognition of Japanese handwritten characters with Machine learning techniques

A comprehensive research on the difficulties for the recognition of Japanese handwritten characters and the application of Machine learning techniques used to approach this problem

Author
José Vicente Tomás Pérez

Supervisor
José Manuel Iñesta Quereda
Department of Software and Computing Systems (DLSI)

Bachelor's degree in Multimedia Engineering

Escuela Politécnica Superior

Universitat d'Alacant
Universidad de Alicante

ALICANTE, July 2020

Abstract

The recognition of Japanese handwritten characters has always been a challenge for researchers. A large number of classes, their graphic complexity, and the existence of three different writing systems make this problem particularly difficult compared to Western writing. For decades, attempts have been made to address the problem using traditional OCR (Optical Character Recognition) techniques, with mixed results.

With the recent popularization of machine learning techniques through neural networks, this research has been revitalized, bringing new approaches to the problem. These new results achieve performance levels comparable to human recognition. Furthermore, these new techniques have allowed collaboration with very different disciplines, such as the Humanities or East Asian studies, achieving advances in them that would not have been possible without this interdisciplinary work.

In this thesis, these techniques are explored until reaching a sufficient level of understanding that allows us to carry out our own experiments, training neural network models with public datasets of Japanese characters.

However, the scarcity of public datasets makes the task of researchers remarkably difficult. Our proposal to minimize this problem is the development of a web application that allows researchers to easily collect samples of Japanese characters through the collaboration of any user.

Once the application is fully operational, the examples collected until that point will be used to create a new dataset in a specific format. Finally, we can use the new data to carry out comparative experiments with the previous neural network models.

Resumen

El reconocimiento óptico de caracteres japoneses manuscritos ha sido siempre un reto para los investigadores. El gran número de clases, su complejidad gráfica y la existencia de tres sistemas de escritura diferentes hacen este problema particularmente difícil en comparación a la escritura occidental. Durante décadas, se intentó abordar el problema mediante técnicas tradicionales de OCR (Reconocimiento Óptico de Caracteres), con resultados variados.

Con la reciente popularización de las técnicas de aprendizaje máquina a través de redes neurales esta investigación se ha revitalizado, aportando nuevos enfoques al problema. Estos nuevos resultados alcanzan niveles comparables al reconocimiento humano. Además, estas nuevas técnicas han permitido la colaboración con disciplinas muy diferentes, tales como las Humanidades o los estudios de Asia Oriental, logrando avances en las mismas que no habrían sido posibles sin este trabajo interdisciplinario.

En este trabajo se exploran dichas técnicas, hasta alcanzar un nivel de comprensión suficiente que nos permite realizar nuestros propios experimentos, entrenando modelos de redes neuronales con conjuntos de datos públicos de caracteres japoneses.

Sin embargo, la escasez de dichos conjuntos de datos públicos dificulta enormemente la tarea de los investigadores. Nuestra propuesta para minimizar este problema es el desarrollo de una aplicación web que permita a los investigadores recolectar ejemplos de caracteres japoneses fácilmente mediante la colaboración de cualquier usuario.

Una vez la aplicación se encuentre plenamente operativa, se utilizaran los ejemplos recogidos hasta el momento para crear un nuevo conjunto de datos en un formato concreto. Finalmente, podemos utilizar los nuevos datos para realizar experimentos comparativos con los anteriores modelos de redes neuronales.

抄録

日本語の手書き文字の認識は、常に研究者にとって課題でした。多数のクラス、グラフィックの複雑さ、および 3 つの異なる書記体系の存在により、西洋の書記に比べて特に複雑とされています。何十年もの間、従来の OCR（光学式文字認識）を使用して問題に対処する試みが行われており、結果はさまざまです。

ニューラルネットワークを介した機械学習の最近の普及により、この研究は活性化され、問題に新しいアプローチがもたらされました。これらの新しい結果は、人間の認識に匹敵するパフォーマンスレベルを達成しています。さらに、これらの新しい方法により、人文科学や東アジア研究などとのコラボレーションが可能になり、この学際的な研究なしには不可能だった進歩を達成することができました。

この卒論では、日本語の文字の公開のデータセットを使用してニューラルネットワークモデルをトレーニングし、独自の実験を実行しました。

しかし、公開データセットが不足しているため、研究者の作業は著しく難しいになります。この点への対策として、任意ユーザーの貢献を通じて日本語の文字のサンプルを簡単に収集できる Web アプリケーションの開発を行いました。

アプリが完全になると、その時点までに収集された例を使用して、特定の形式で新しいデータセットが作成されます。最後に、新しいデータを使用して、以前のニューラルネットワークモデルとの比較実験を実行できます。

Acknowledgments

Although this is only the start of a new phase in my life, I have to thanks all this people for the help along the way until now, without their support this would have been truly difficult, if not impossible.

For my university classmates: To my teammates from our group TAKO･KO: *Alberto, Irene, María and Eduardo.* For the hard-working hours creating and pushing forward our last-year PBL project, a result of all the skills we acquired along this degree. And, most importantly, for being the best friends anybody could have, even when we disagree about everything possible just for fun. To the rest of you, though you all know who I'm talking about: to *Mónica*, putting up with me even before university, to *Esther*, for listening to me about every thinkable topic during our walks, to *Martín, Moisés, Mario, Javier, Yera, Carlos, Roque* to all of you.

To my friends from Japanese lectures, *David and Raúl*, for sharing my passion for this language and culture, supporting each other during this long 7 semesters of classes, even if after all of this we only reached intermediate level of this infinite ocean called Japanese language. To *Dani*, for making me laugh and for standing my horrible football skills. Hope he keep on it. And, of course to my Japanese teacher *You Ozawa* (in Japanese (JP): 小澤陽), for teaching me almost all my Japanese during these years, for showing us the real culture as much as possible, for not giving up on us, and, last but not least, for being a true friend for life. Without her, I wouldn't have the required knowledge to write this thesis.

To my lifelong friends from my home village. *Evora, Rebeca, Grego, Jonathan, Rafa, Josean*, even if we don't meet for weeks because of schedule problems, you are always there willing to listen to my stupid tales and useless facts. Hope you can do it even for a longer time because I'm not giving up soon.

To my supervisor teacher, *José Manuel*, for believing in me for this project, even when he have more than enough students to care about.

And of course, to my parents, my brother and sister, for supporting me throughout my life

Of course, I am interested, but I would not dare to talk about them. In talking about the impact of ideas in one field on ideas in another field, one is always apt to make a fool of oneself. In these days of specialization there are too few people who have such a deep understanding of two departments of our knowledge that they do not make fools of themselves in one or the other.

Richard P. Feynman.

When the ego dies, the soul awakes.

Mahatma Gandhi.

Contents

List of Figures

List of Tables

Listings

1 Introduction

This chapter serves as introduction to the main topic of the thesis. It is made up of 5 different sections: Section 1.1 sets out the problem and how we expect to undertake it, Section 1.2 explains the personal and circumstantial motivations to carry out this research, Section 1.3 shows some of the currently most important related works in the field of Japanese language recognition, Section 1.4 defines the objectives for this project and Section 1.5 details the structure of this document.

1.1 Overview

The main objective of this Bachelor's Thesis is to research the neural network structures capable of obtaining state-of-the-art results for the recognition of modern Japanese handwritten characters. In order to understand this task's challenge, we will study the structure and peculiarities of the Japanese language itself, starting from its history and written form until these days. Firstly, we will use previous existing datasets to develop the model, and afterwards, we will create a new dataset with a small number of character classes from samples obtained via crowd-sourcing. This will be achieved with a web application where the users will input the characters requested using the device's touchscreen, after which they will be processed to a standard format following current conventions. This way we hope to address the current lack of accessible and sizeable corpuses of Japanese language, creating a medium for researchers to easily obtain open training data.

Finally, we can compare the results of the model using the same classes from previous datasets and the ones obtained via crowd-sourcing. With this data, we can get to meaningful conclusions about the situation of the field nowadays and remark ideas for future paths to follow which may lead to improvements.

1.2 Motivation

Fulfilling the objective of a thesis, this document presents the work that was carried out to prove the knowledge acquired during the Bachelor's degree in Multimedia Engineering taken at the University of Alicante from 2016 to 2020. From this experience stems the technological part of the motivation to create this work.

On the other hand, Japanese culture and society have been from many years ahead of university a particular interest of mine. I found their lifestyle and language, as an evolution from their geographical and sociolinguistic environment, especially unique from my, at that time, western view of the world. Therefore, I decided soon to turn this into my second specialization. On this purpose, simultaneously to my degree, I have studied the Japanese language every semester at this same University.

I have always wanted to believe that having two very different specializations should not be an obstacle but an advantage in today's world, as long as you hold a similar level of expertise in both fields. Nowadays, the chances to study a degree with the presence of different specialities are growing, yet this is still a work in progress.

By creating this thesis, I wish to close a circle by combining my two fields of study during these four years: Multimedia engineering and Japanese language, at first glance areas very far apart from each other, yet sharing a world of possibilities when they get together. I hope that by taking a different approach to this language, which has come to be one of my particular passions, I could attract people from technical careers to it, and, eventually, help growing an inter-disciplinary environment.

Apart from this, the techniques used in this work will help me learn about the theory and tools used for Machine Learning in general, and Optical Character Recognition in particular.

1.3 Related works

As with other major languages, the first attempts to recognize Japanese characters used traditional OCR methods, such as pattern recognition or feature extraction. Soon these techniques were replaced with the coming of modern machine learning ones, later we will develop on both of them. In this section, we will show the main related works about recognition of Japanese characters using machine learning, as it is this thesis's topic.

Although the recognition of Japanese characters is a significant challenge due to their own nature, the recognition of typewritten characters was considered already a "solved" problem when using a standard font. Because of this, most of the works and research papers nowadays about Japanese ideogram recognition only deal with the handwritten script, if not single characters. Some works going towards this direction are Grębowiec & Protasiewicz (2018) and Tsai (2016). In both papers, a neural network model is suggested as an attempt to improve current recognition success statistics.

In recent years, data science has become one of the main research fields in Computer Science. The reason behind this growth is the need to manage and process large quantities of information, a.k.a "*Big data*", and machine learning plays a key role inside this data pipeline. This trend to move any existing data into the digital world have naturally attracted scholars from very different fields into this area of informatics. One of the most important ones is the Humanities. For many years ahead, Humanities have used data-related technologies such as databases to properly store huge amount of curations information and other types of documents, but now, with the popularization of data science, the scholars of this field have seen the opportunity to create new approaches to their research through applying data science techniques. The result of this processing not only eases their work, but also informs them about details or interpretations they didn't notice at first glance. Of course, Japanese studies are not an exception.

We call this union between humanities and informatics "*Digital Humanities*". Nowadays one of the main institutions which are pushing forward the boundaries of Japanese character recognition is the Center for Open Data in the Humanities (CODH) from the Research Organization of Information and Systems (ROIS) in Japan. Apart from their work with other cultural or art-related elements, their need to efficiently process old Japanese literature have required them to create models and datasets aimed to Japanese script far better from previous ones. The most famous project until now is Clanuwat et al. (2018), where a dataset of old cursive Japanese characters and its corresponding recognition model via a neural network is suggested. We will develop further on this language variances and its drawbacks later.

All in all, these are some of the related works until today where Japanese handwritten characters processing is the main topic, sometimes as the primary topic, others as derived one. Using the knowledge acquired from these readings, I wish to develop the ability to judge my own model. I hope that this presented work bridges further the research on humanities and machine learning.

1.4 Proposal and Goals

The main purpose of this thesis is to research the current field of Japanese language recognition, both from western and native sources, and to compare various preexisting models and datasets in order to understand where the field may lead for future improvements. Therefore, the main objectives of this Bachelor's thesis are:

- Review the current state of the art of OCR techniques using Machine learning in general, and Japanese character recognition in particular.

- Understand the challenges of Japanese handwritten recognition by explaining the language structure and peculiarities, from historical to today's speech.

- Compare the previously existing models of recognition and their different success rates depending on their focus.

- Analyse current datasets available of Japanese language and create our own one using a crowd-sourcing web application that can also be repurposed for future corpus creation.

- Use a neural network model based on previously studied ones with similar purpose and compare its performance with our dataset and others.

1.5 Schedule

This project has been developed during the 4th year of the degree in Multimedia Engineering. The thesis topic was proposed in November 2019, starting work soon after.

The work pace of the project has not always been the same throughout the year due to different circumstances. Until the end of May, the project has been developed at the same time as the PBL project of my degree's last year. During part of this period, I worked on the project twice a week at the university itself, at 4 hours per session, researching and doing experiments of Japanese dataset classification. This situation changed abruptly from March due to the COVID-19 pandemic, working from that moment until now from home. The web application was developed from late March until the end of April. During the peak workload of the PBL project, the time spent on this final degree project was reduced, but we tried to never interrupt it. Finally, most of this Bachelor's thesis has been written during the months of June and July, at the same time that I was doing the external internship for my degree.

1.6 Outline

This document's structure is conceived to follow the natural flow of the research, avoiding to put chapters with previously unexplained concepts before other ones: Chapter 1 serves as an introduction to the main topic of the thesis, its motivations and some related works. Chapter 2 intents to be a comprehensive research about traditional Optical Character Recognition, the use of Neural Networks for this purpose and the peculiarities of the Japanese language on this field. Chapter 3 is a record of the materials and technologies employed during the project, and the methodology followed when using each of them. Chapter 4 presents the results of our experiments and the development of our crowd-sourcing application. Chapter 5 describes our conclusions in summary and proposes future improvements based on our experience.

2 State of the Art

This chapter intents to be a comprehensive explanation about the use of Neural Networks for Optical Character Recognition and the peculiarities of Japanese language on this field. It is made up of 3 different sections: Section 2.1 introduces us to the world of OCR and its origin, Section 2.2 gives a brief introduction to Japanese language itself and the challenges it presents for OCR and Section 2.3 show us a general view of the Neural Network technologies and its ins and outs, with details of how these models are applied for OCR.

2.1 Introduction and history of OCR

We call Optical Character Recognition, a.k.a OCR, to the conversion of images of typed, printed or handwritten nature into machine-readable data, whether for converting it into a digital document or for machine internal use. This images can come from scanned documents, photos, subtitle text from television broadcast... etc. OCR is one of the most successful applications of technology in the field of pattern recognition and artificial intelligence.

It has reached a level of accuracy which allow us to use it on a daily basis with a small margin of error. The applications range from digitization of printed texts to the recognition of passport documents data. This process is often only a part from a bigger system, aiming to objectives such as machine translation, text-to-speech or text mining.

Many commercial systems for performing OCR exist, although the machines are still not able to compete with human capabilities. This said, near-perfect performance is already achieved in some environments. According to the *Annual Test of OCR Accuracy* by the Information Science Research Institute from the University of Nevada, modern OCR technologies performance range from 81% to 99%. Although the term OCR is a general definition for a wide range of models, we can subdivide it into 4 categories:

- **Optical Character Recognition** is supposed to refer only to typewritten text, one character at a time, although in reality is used as a generic term for these 4 technologies.
- **Optical Word Recognition** refers to typewritten text, one word at a time, when the language uses spaces between words. This is an interesting fact for us, as this is not the case of Japanese.

- **Intelligent Character Recognition** refers to handwritten script or cursive text, one character at a time.
- **Intelligent Word Recognition** targets the same media as above, but one word at a time. Both models usually involve the use of machine learning.

OCR lies in the field of Automatic Identification. We call Automatic Identification to a wide range of solutions used when a traditional input into a computer is not the best nor the most efficient way to handle the processing of data. Automatic Identification technologies are around us everyday, such as bar codes in supermarkets, speech recognition in smartphones or magnetic stripes in credit cards. Compared to other automatic identification technologies, OCR is unique in that it does not require control over the original information. It is true though that the performance is directly dependent upon the quality of input data, whether is typewritten or handwritten script.

To enable machines to perform human functions is an old ambition, and so is OCR history. In the next section we will overview each significant milestone in OCR history and their contribution to today's technology.

2.1.1 The history of OCR: An overview

The first attempts of OCR-related technologies can be traced back to the 1870s. In this year the American inventor Charles R. Carey created the first retina scanner using a mosaic of photocells. Another important predecessor was the 1885's Nipkow disk. Invented by the Prussian technician Paul G. Nipkow, it consisted of a mechanically spinning disk with a series of equally distanced holes of the same size drilled in it. The Nipkow disk was the main component of mechanical televisions. Inside them, the disk take the role of a primitive "raster". The "screen" of these televisions was a small circular sector of the disk, keeping the rest of it hidden behind another opaque layer. When spinning, each hole appears to be a full line of the visible circular sector. This allowed dividing the light coming from the image we wanted to transmit into small pieces, coming through each hole. This way anything could be scanned line by line. In the other side of the disk, a light sensor was placed (like a photodiode or photocell), which converted the light coming from each hole individually into an electric signal. In the receiver side, the same process was performed but in the opposite way, with disk spinning at strictly the same velocity. Although a really smart concept, the Nipkow disk is greatly limited by its size and curvature.

Moving to the 19th century, the development of OCR predecessors were boosted by the creation of devices to aid the blind. In 1912, Edmund Fournier d'Albe, an Irish physicist, develops the Optophone, a handheld scanner that when moved across a text, generated a time-varying chord of tones to identify letters. The tool was supposed to be aimed at the blind, but reading was slow and required of training. Moving toward mid-century, electronic data processing was becoming an important field due to its potential applications in the business world. In 1954, the first true OCR machine was installed in the offices of American magazine *Reader's Digest*, handling the conversion of typewritten sales reports into punched cards. By mid-1950's, the first OCR machines became commercially available. Although these machines had a more business-related purpose, the research in OCR technology for the disabled didn't pause. In the 1960's an influential device for the blind was developed. The Optacon, as it was called, was an electromechanical equipment with which the user moved a scanner over a text while a finger pad translated the words into vibrations that could be felt in the fingertips. It was a great leap forward for blind people, until then limited to braille script reading.

A common problem of the first generation of OCR technologies, until 1965, was the input material used. The font used for the texts was specially designed for machine-reading and had a constrained shape. Later on, the number of supported fonts grew to 10. This number was determined by the pattern recognition method applied, mostly because it had to compare each character with an entire library of prototype images for each character of each font.

By the early 1970's, the second generation of OCR came in. Though limited in the number of supported characters, hand-printed documents were now readable for these machines, like the IBM 1287. Japanese companies also played a role, like Hitachi or Toshiba, with a letter sorting machine. Most importantly, a font standard for OCR was pursued. As a result, the American OCR-A font was created, followed by the European OCR-B. While the first was developed with OCR requirements in mind, the second intended to feel more natural under human eyes. These fonts plus some special characters from Magnetic Ink Character Recognition (MICR) code are included in the Unicode standard since 1993.

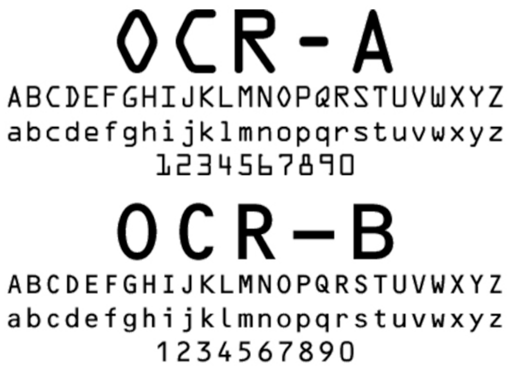

OCR-A
ABCDEFGHIJKLMNOPQRSTUVWXYZ
abcdefghijklmnopqrstuvwxyz
1234567890
OCR-B
ABCDEFGHIJKLMNOPQRSTUVWXYZ
abcdefghijklmnopqrstuvwxyz
1234567890

Figure 2.1: Fonts OCR-A y OCR-B

The third generation of OCR machines appeared around the mid 1970's. As always, low cost and high performance were the ideal objectives for the devices, even more now due to the market inquiry. The uniform print spacing and the small number of fonts made OCR devices really useful in the age before personal computers. One could create documents' drafts on typewriters and then fed them into the computer through OCR, cutting costs substantially in a time when word processors were still an expensive tool. In 1974, Kurzweil Computer Products Inc. develops the first omnifont OCR software, opening the doors to the development of font-independent devices.

Starting in the mid 1980's, OCR devices and software became available to the wide public, getting consequently cheaper until today's standards. Nowadays, there is commonly no such things as "OCR machines", but a wide variety of devices using OCR libraries and software. Some famous ones are the free library Tesseract, sponsored by Google, the PDF reader from Adobe Acrobat or the one used in Google Drive files.

The research in OCR is no more focused in the typewritten script, and have reassigned every effort in the field into the handwritten script and real-time recognition, like with street signs or menus translation using a smartphone camera. Usually, these solutions include the use of machine learning techniques, as we will detail later. Moreover, human crowdsourcing is recently used as a recognition technique, especially to correct texts already passed through an OCR software.

2.1.2 The OCR pipeline

In this section, we will explain the general recognition process and the techniques typically employed in each step, but first, we will review some basic OCR terms. In OCR, we call characters or patterns to the letters, numbers or special symbols that make up a dataset. Inside a dataset, each class correspond to a different character. The training of an OCR model is performed by showing the software examples of characters of all the different classes. Based on this, the machine builds a common description of each class of characters by defining a series of distinctive features. Usually in commercial devices the training is already stored inside the software, but some of them give the chance to train the model with new classes added by the user.

Although there are some systems where a step is not required or where the process is shortened, a general OCR pipeline consists of 5 steps. First, we digitize the text or character through optical scanning. Then, the regions of the image with text are located, extracting them through a segmentation process. Once we have each part clearly delimited, preprocessing is performed if necessary to get rid of imperfections, like noise. After this, the system tries to identify the most important features of each image, in order to compare them with similar features found during the training process and therefore achieve an accurate classification. We will now describe each step with more detail.

During the **optical scanning**, a digital image of the original material is captured. Printed documents usually consist of black print on a white background, so it is common to convert the resulting image into a bilevel image of black and white. This process is called thresholding and is performed to facilitate the recognition in the following steps of the process. Usually, a fixed threshold is used, where grey-levels below it are considered black, and every other thing white. One popular option for this is the *Otsu's Binarization.* With correctly scanned documents this tends to be sufficient, but documents usually have a rather large degree of contrast. For these cases, an adaptive threshold is recommended. This method varies the threshold depending on the local properties of contrast an brightness, though is more computationally intensive. In the past, the range of devices used for this process was limited, mainly scanners, and therefore it was easier for developers to control the result. Nowadays, with the coming of smartphones, images are no longer always high-contrast clear, neither with fixed color background. Because of this, modern techniques not only include thresholding, but a wide variety of filters.

The **location and segmentation** of the text inside the image is a step inside the process that has gained importance in modern days, as OCR is often performed over images where the text is only a small part of a wider scene, often with a complex background. A fresh example of this is self-driving cars, equipped with cameras that must take into account traffic signs. Focusing on the texts, segmentation is the isolation of characters or words. This way characters are treated in a one by one manner. In a perfect environment, this technique should be easy to implement, as it only consists in isolating each connected black area, but in reality, several problems may arise. The most common one is the extraction of touching and fragmented characters. When two characters are touching, the software may try to treat them as a single one, leading to recognition problems. These imperfections are often a result of dark images or low thresholds. Serif fonts are also a source of problems, as a minimal threshold deviation can cause them to touch. If we change to the realm of handwritten script, variances are so common that word by word recognition is also a popular scope. We may also want to reduce noise in the background, without neglecting the tracking of small characters like commas or docs, sometimes mistakenly treated as noise.

The next phase in the process, and the last image-treatment one, is the **preprocessing**. Due to a low scanning resolution or an inadequate threshold selection, characters may be smeared or broken at this stage. This process aims to smooth the digitized characters before feature extraction and classification. The smoothing implies both filling and thinning. For this task we use two basic techniques: **dilation** and **erosion**, respectively, part of the so-called "Morphological operations". These techniques are performed using a structuring element, a predefined kernel, over an input image. A kernel is a small matrix with certain values used to apply effects over an image. To do this, the center value of the kernel is lied over a pixel, with the rest of the values over the surrounding ones. We must then multiply each pixel by the corresponding entry of the kernel and then take the sum. That sum becomes the value of the center pixel in the output image. This process is repeated over each pixel of the input image. We call this operation "convolution". One subtlety of this is what to do with the edges of the images, where kernel values may not have a corresponding pixel. A common fix for this is to extend the edge values of the image out by one, keeping the output image the same size. There are quite a large range of kernels used in computer vision, each with a specific name. Sobel kernel shows the differences in adjacent pixel values in a particular direction, usually for edges contrast, while blur kernel de-emphasizes these differences. As we will see later, convolution is also used in machine learning for feature extraction. In addition, preprocessing usually applies **normalization** to obtain characters with uniform slant, size and rotation. For this, the angle of rotation must be found. While scanning a text or document, a skew might get into the scanned image. In this case, a skew is an image oriented at a certain angle within horizontal. Usually a Hought transformation method is used for skew correction, but the simplest one is the so-called projection profile method. In this method, we project the image horizontally to get a histogram of pixels (a bar graph representing the distribution of numerical data). Then the image is rotated in different angles separated by a delta interval. Wherever we find the maximum difference between peaks in the histogram, that will be the skew angle of the image. The following correction is simply an image rotation through an angle equal to the skew angle in the opposite direction. However, if needed, single character rotation can be performed after recognition.

The last step previous to classification is the **feature extraction**. This is one of the most difficult problems of pattern recognition and OCR. In this step, we aim to capture the essential characteristics of the characters. The most basic way to describe a character is by the actual image, but usually more advanced methods are used to shorten the process. These techniques are designed to extract only the features that characterize each symbol, leaving out the unnecessary ones. We could divide these methods in the next three categories:

- **Distribution of points** is a group of techniques based on the extraction of a statistical distribution of points. Their main advantage is the tolerance to distortions and style variations. Some of the more used ones are:

  Zoning: The image is divided into several overlapping or non-overlapping regions, within which the densities of black points are processed as features.

  Moments: We choose a center for the image, like the center of gravity of the character, and then we use the moments of inertia of black points about it as center.

  Crossings and distances: The number of times a character shape is crossed by vectors along certain directions is used as a feature in this technique. Usually the length of vectors crossing within the boundary of the character is considered another feature.

  N-tuples: The joint occurrence of black and white points in a certain specific order is used as feature.

  Characteristic loci: The number of times a vertical line is intersected by the line segments describing the character.

- **Transformations and series expansions** such as Fourier, Walsh, Haar, Hough...etc. Techniques like these are used because they help to reduce the dimensions of the feature vector and can be invariant to rotations and translations. These transformations are usually based on the curve describing the contour of the character. Because of this, they are susceptible to noise around the character's border, but not to noise inside of the character.

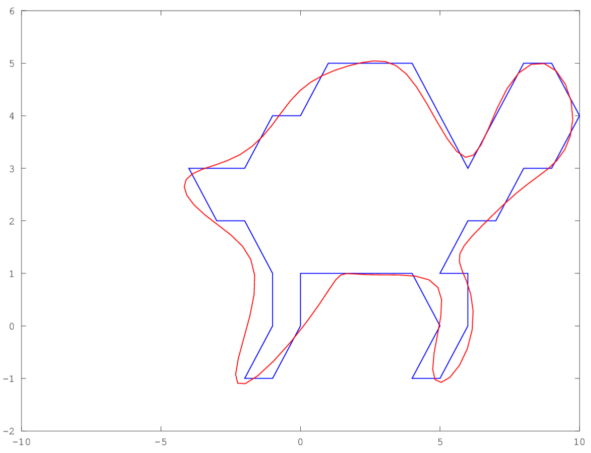


Figure 2.2: Elliptical Fourier Descriptor used over a cat shape from Mathworks, by Auralius Manurung (2016)

- **Structural analysis** is the extraction of features based on the topological and geometric structures of a character. This features usually include strokes, end-points, bays, intersections and loops. The extraction of these is not trivial, but once obtained, they are very tolerant to noise and style variations, but not to translation or rotation.

One last approach to this step is the **template-matching techniques**. Surprisingly, in these no features are actually extracted. Instead, the image is matched with a set of prototype images for each class. We then quantify the distance between each pair, choosing the best match as the correct answer. Although is simple to implement, this technique is easily sensible to noise, rotation or variations.

The most important step of the process is, of course, the **classification**. Here we aim to identify each character and assign to it the correct character class as accurately as possible. We could divide classification models into two categories: decision-theoretic methods and structural methods. The first ones are used for characters numerically represented in a feature vector, while the others are used for characters described by more physical patterns which are not as easily measured. First, we will take a look into the main decision-theoretic methods:

- The **minimum distance classifiers or matching techniques** are the groups of methods where the distance between the feature vector of the character and the class is measured, usually using the Euclidean distance, to find out the correct character identity. They tend to work well if the classes' means are separated between each other by a large distance. If the entire character is used as input, a template-matching approach is taken, instead of feature extraction.

- The **optimum statistical classifiers** are a variety of probabilistic approaches to recognition, with the objective of having, on average, the lowest probability of making classification errors.

  To build a probabilistic approach, we start defining our variable. First, we have to choose a random variable that represents the statistical feature of a character image, we will call this Z. A good example of a random feature would be the fill percentage of the image, in other words, the proportion of foreground pixels in the binary image. Using the random variable, we can build a probability model for that feature in the form of $Prob(Z|C)$, where C are the character classes $C = A, B, 1, 2....$ Given this, the probability that the random variable Z belongs to class C is computed for all classes $C = 1...N$. Seen as a function for each event C, the probability model $Prob(Z|C)$ is known as the likelihood function $L(C) = Prob(Z|C)$. All in all, we want to find the probability that the character is, in fact, C.

  If there is no prior knowledge of what C might be, then we usually assume it to be normally distributed, like $Prob(C) = 1/36$ (as for a set formed by every English alphabet letter plus numbers). This probabilistic approach is called a *Bayesian approach*. The closer this assumption is to reality, the closer a Bayes' classifier comes to optimal prediction. The most popular and simple example of such classifier is a **Naive Bayes classifier**. This family of techniques are based on the assumption of strong independence between the features, i.e., that the value of a particular feature is independent of the value of any other feature, given the class variable. Though this may be seen as an oversimplified view of a problem, as sometimes there are obvious correlations between features, the naive Bayes classifier has demonstrated to be useful in practice, even in complex situations. To obtain these parameters necessary for the classifiers, a training process is undertaken, after which a description of each class is obtained.

  In general, these probabilistic predictions, classifications and inferences may be simple in principle, but they rely on the proper selection of the random variable Z, that is, to choose the feature that best represents every character. Aboura (2016)

  Although these classifiers have been used along with neural networks for predictions, they also have proved to work really well for their own.

- **Neural Networks** are technically also part of the family of decision-theoretic methods, but we will cover them more extensively in a specific chapter, as they are a central part of this thesis.

On the other hand, **Structural methods** take into account the relationship between character features as an element of importance when deciding on the class membership. The most common of them is the **Syntactic methods**, where this relationship between character is formulated by using grammatical concepts. We say that each class is defined by a grammar, therefore a character is more similar to a class if is more likely to be generated by the grammar of this class than others. Likewise, we say a grammar is usually made-up of the structural components of the character.

The last step in the OCR pipeline is the **post-processing**. This step is compound of some techniques used to improve the OCR final result after individual characters' classification. The most common one is grouping, where characters already identified are associated into strings, words, using the characters' location in the document as parameter. When letters are found enough close to each other, they are grouped together. This process is fairly simple when using typeset characters, as fonts usually have a fixed white-space between them. The real problem comes when we are dealing with handwritten characters, where no standard exists.

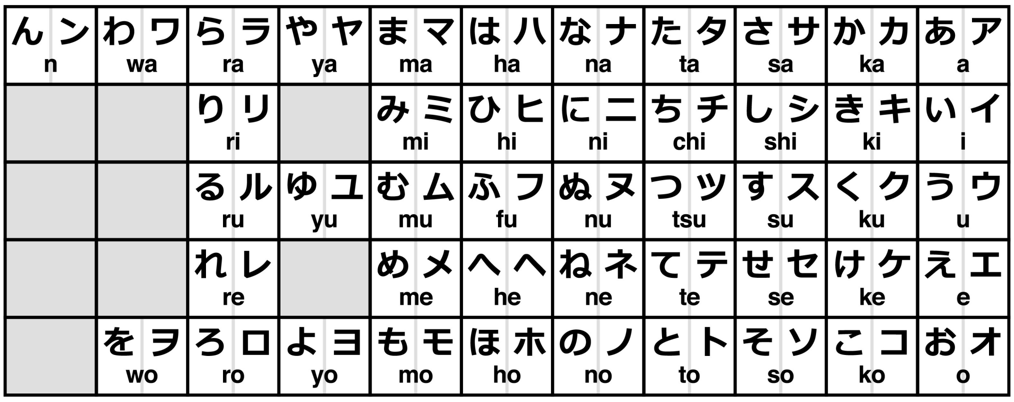
Other error-detection or correction technique common in post-processing is the use of context for improvements. **Context** is one of the main ideas to come to mind when thinking about how to improve OCR results, though the actual implementation may be fairly complex. One approach is to calculate the possibility of sequences of characters appearing together, so unlikely or impossible letters' pairs could be quickly discarded. This requires to specify certain rules for each language. Although this may be a fast method, the direct use of dictionaries to look up full words has been proven to be the most efficient method for error detection. Nevertheless, such dictionary methods can not detect an error in which classification resulted in another legal word, as it would see it as a perfectly correct string.

2.2 The Japanese language and its recognition challenges

Note: As for this section and the rest of the thesis standard, transcriptions of Japanese words into Latin alphabet will be written in italics. We call this system rōmaji (JP: ローマ字).

In order to approach this thesis's topic properly, first we have to understand to a certain extent what we are dealing with. In general terms, Japanese (JP: 日本語) is an East Asian language spoken mainly in Japan and its surrounding islands, by around 128 million speakers. Officially, Japanese is inside an isolated language family alongside the Ryukyuan (from Okinawa Islands in southern Japan (JP: 沖縄口)), though it holds important relations with Korean, mainly because the later one uses a large number of loanwords from Japanese. This said, Japanese is not a uniform language by any means. Dozens of dialects are spoken in the country, differing usually in the pitch accent, the vocabulary or the particle usage. The most popular ones are the Kanto dialect (JP: 関東弁) and the Kansai dialect (JP: 関西弁), but we can even find dialects in northern Japan which may be unintelligible for the rest of the country's speakers.

The Japanese writing system is among one of the most complexes in the world. Due to historical reasons that will be explained later, the Japanese language has come to be written with a mixture of three different systems at the same time. First we have *hiragana* and *katakana* systems. Both systems are similar, as each of their symbols corresponds to a phonetic syllable, with the exception of consonant n, which is written alone. In addition to the common or "pure" sounds, the syllabaries also provide the speaker with the ability to represent mixed combinations. For this purpose, we can add a smaller version of characters「や」,「ゆ」or「よ」behind any "i" ending character to create a joint syllable. For example, if we take the *hiragana*「に」(pronounced *ni*) and「や」(pronounced *ya*), we can create the syllable「にゃ」(pronounced *nya*). Furthermore, the sounds of the consonants in each syllable can be modified to create the so-called "impure" sounds. To do this, we add the diacritic markers *dakuten*「゛」or *handakuten*「゜」over a character. The first one is used to turn any sound into a voiced consonant: "h" to "b", "k" to "g"...etc, for example, if we add *dakuten* to *katakana*「カ」(pronounced *ka*), it transforms into「ガ」(pronounced *ga*). The second marker is used to turn a "h" consonant into a "p", for example with the *hiragana*「ひ」(pronounced *hi*) and its accentuated version「ぴ」(pronounced *pi*). There are also specials ways to modify the pitch accent for the reader if the word requires it, but we will not cover them here. The main difference between them is that while *hiragana* is often used to write native Japanese words, *katakana* is used to write loanwords from foreign languages. *Katakana* is also often used to write onomatopoeias.



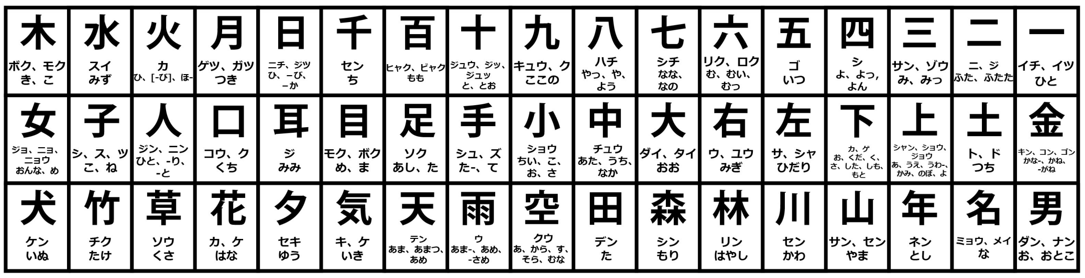
ん ン n	わ ワ wa	ら ラ ra	や ヤ ya	ま マ ma	は ハ ha	な ナ na	た タ ta	さ サ sa	か カ ka	あ ア a
		り リ ri		み ミ mi	ひ ヒ hi	に ニ ni	ち チ chi	し シ shi	き キ ki	い イ i
		る ル ru	ゆ ユ yu	む ム mu	ふ フ fu	ぬ ヌ nu	つ ツ tsu	す ス su	く ク ku	う ウ u
		れ レ re		め メ me	へ ヘ he	ね ネ ne	て テ te	せ セ se	け ケ ke	え エ e
	を ヲ wo	ろ ロ ro	よ ヨ yo	も モ mo	ほ ホ ho	の ノ no	と ト to	そ ソ so	こ コ ko	お オ o

Figure 2.3: Combined *Hiragana* and *Katakana* chart. Left character of each box is *Hiragana*, while right one is *Katakana*. Phonetic lecture is included in *rōmaji*, only pure sounds included. Chart by u/Danilinky (2019)

The last system is the *kanji* (JP: 漢字), Chinese characters integrated into the

Japanese writing system that are essential to language understanding. They are used mostly for writing nouns and the stem form of verbs, where they are used jointly with *hiragana*. In this case, *hiragana* serves as a suffix to create each different verb tense and they receive the name *okurigana* (JP: 送り仮名) or "accompanying letters". For example, in the sentence「今は水を飲んでいます」, meaning "Right now I'm drinking water", we are using *kanji* for both the noun "water"「水」(pronounced *mizu*), and the verb "drink"「飲む」(pronounced *nomu*). As you see, 「飲む」is already using an *okurigana* termination by default to represent the informal present tense, changing it with the one in the sentence we obtain "drinking"「飲んでいます」(pronounced *nondeimasu*), as this is the informal present continuous termination.

It is fair to say that this is not always the rule, as we can find both nouns and verbs in full *hiragana* in the common written script. Japanese is a language that tends to adapt itself to each situation, and therefore is also a language plenty of exceptions. One such case is the different kanji reading. In Japanese, each kanji usually have two or more readings: *on'yomi* (JP: 音読み) and *kun'yomi* (JP: 訓読み). The first one is the Chinese originated reading, used usually when two or more *kanji* are put together, as in nouns, while the second one is used when a *kanji* is used along with *hiragana* characters, as in verbs. This is far to be a strict rule and you can commonly find *kanji* with more than two readings, even five. Moreover, some *kanji* were created by the Japanese themselves and therefore they lack an *on'yomi* reading. *Kanji* system is undoubtedly complex, however, it serves one more specific purpose. In Japanese, homophones (words that share the same pronunciation, regardless of the meaning) are remarkably frequent. This is due to the low number of sounds in the Japanese phonology and the use of loanwords from Chinese. In China, these words could be told apart using a tone system, but they were phonetically uniformed at their arrival to Japan, where there is no tone system. *Kanji* helps us to differentiate between such words efficiently. In the spoken speech this is not possible and every word meaning is guessed by the context, that is why Japanese is often referred to as a highly contextual language.



木 ボク、モク き、こ	水 スイ みず	火 カ ひ、[-び]、ほ-	月 ゲツ、ガツ つき	日 ニチ、ジツ ひ、−び、−か	千 セン ち	百 ヒャク、ビャク もも	十 ジュウ、ジッ、ジュッ と、とお	九 キュウ、ク ここの	八 ハチ やっ、や、よう	七 シチ なな、なの	六 リク、ロク む、むい、むっ	五 ゴ いつ	四 シ よ、よっ, よん	三 サン、ゾウ み、みっ	二 ニ、ジ ふた、ふたた	一 イチ、イツ ひと
女 ジョ、ニョ、ニョウ おんな、め	子 シ、ス、ツ こ、ね	人 ジン、ニン ひと、-り、-と	口 コウ、ク くち	耳 ジ みみ	目 モク、ボク め、ま	足 ソク あし、た	手 シュ、ズ た-、て	小 ショウ ちい、こ、お、さ	中 チュウ あた、うち、なか	大 ダイ、タイ おお	右 ウ、ユウ みぎ	左 サ、シャ ひだり	下 カ、ゲ お、くだ、く、さ、した、しも、もと	上 シャン、ショウ、ジョウ あ、うえ、うわ-、かみ、のぼ、よ	土 ト、ド つち	金 キン、コン、ゴン かな-、かね、-がね
犬 ケン いぬ	竹 チク たけ	草 ソウ くさ	花 カ、ケ はな	夕 セキ ゆう	気 キ、ケ いき	天 テン あま、あまつ、あめ	雨 ウ あま-、あめ、-さめ	空 クウ あ、から、す、そら、むな	田 デン た	森 シン もり	林 リン はやし	川 セン かわ	山 サン、セン やま	年 ネン とし	名 ミョウ、メイ な	男 ダン、ナン お、おとこ

Figure 2.4: 51 Basic *kanji* chart. Under each one, Chinese *on'yomi* readings are written in *katakana*, while Japanese *kun'yomi* readings are written in *hiragana*, as is provided in the standard. Chart by u/Danilinky (2019)

As for basic grammar notions, Japanese is said to have a SOV morphology, i.e, to have a subject-object-verb word structure. In reality, the only strict rule is to keep the verb at the end of the sentence, with the rest of elements mostly freely positioned. One Japanese peculiarity previously mentioned are the particles. These suffixes are used to mark the grammatical function of every sentence element, or as modifiers. For instance, in the sentence 「これはパソコンです」, meaning "This is a computer", the syllabic character「は」(pronounced *wa*), serves as the particle of topic, which is in this case「これ」("This", pronounced *kore*). Following these we have the word「パソコン」(pronounced *pasokon*), written in *katakana* because it is a phonetic abbreviation for the English loanword "Personal Computer". Finally we put the verb 「です」(pronounced *desu*), meaning "to be", at the end of the sentence, as is always required.

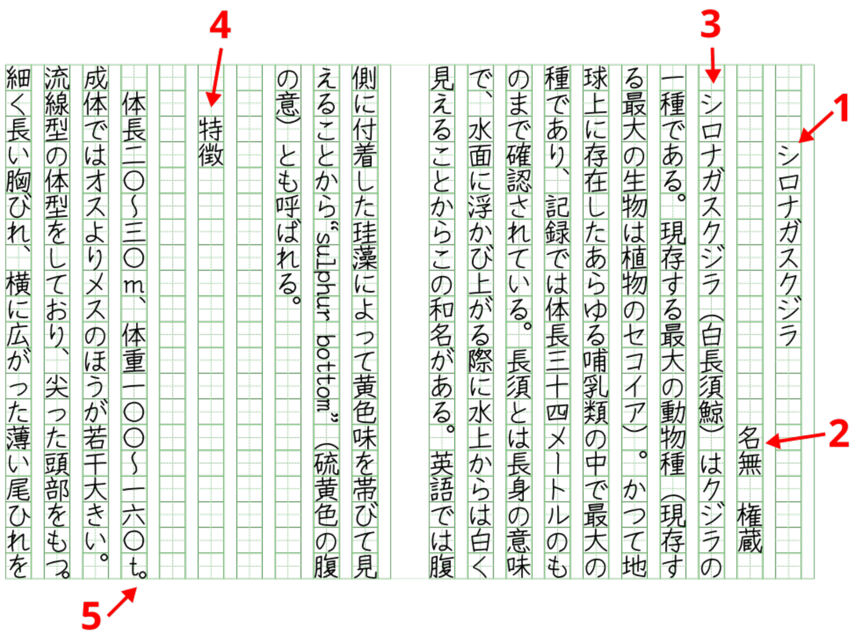
In this section, we will give a comprehensive explanation about modern Japanese script, both typed and handwritten, as is directly related to this thesis's premise. We will not further cover other language fields, such as grammar or phonetics, as they are not directly related, though they could be eventually brought up if part of the written system needs of them to be properly explained.

2.2.1 Standards of Japanese writing

The most important part of the Japanese language for this thesis topic is the Japanese writing style, as we aim to accomplish a recognition rate as accurate as possible. Therefore, we need to consider as many aspects as possible of the Japanese writing conventions. This knowledge will not only be useful for recognition, but also for the dataset compilation process, as native speakers will be more willing to contribute if the setting they see is more familiar.

Perhaps the most popular aspect of Japanese writing from abroad is the **stroke order**, maybe because of the prevalence in the media of traditional calligraphy with brush as a characteristic art from Japan or China. Stroke order is a concept that is taught from the very beginning in Japanese schools and have a few exceptions, but in general terms, the rules are: to go always top to bottom, left to right; horizontal before vertical; and vertical strokes passing through others must be written at the end. All of this in one go without any partial correction. These rules are important not only for *kanji*, but also for *hiragana* and *katakana*. The concept of stroke order may be seen as far-fetched from countries that use the Latin alphabet, but there are some cases where the order is so important that affects the readability of the word. The most obvious example of this occurs with the pairs of katakana characters「シ」and「ツ」(pronounced *shi* and *tsu*, respectively) or with「ン」and「ソ」(pronounced *n* and *so*). Past OCR recognition models for Chinese and Japanese characters were dependent on the stroke order to work, this was inconvenient because the program relayed in the user to correctly input every symbol. Using machine learning techniques we are no longer dependent on them, though overly similar characters could be problematic.

When talking about rules and style codes, **Japanese handwritten script** has proved to be an excellent example. Stroke order may be the most popular instance of this, but is far to be the only one. It is standard in Japan to write any type of manuscript in a special type of paper layout called **Genkō yōshi** (JP: 原稿用紙). In modern day Japan, the use of this "manuscript paper" has declined enormously due to the preferred use of word processing software, although some computer programs still have *genkō yōshi* templates included. However, they are still very widely used (for example, in newspapers) and primary and secondary students are required to write their essays or assignments on *genkō yōshi*, and therefore they have to learn the correct way to use it. The standard layout is designed for vertical writing, from right to left. Each column is composed of 20 squares, with usually ten columns per page. Each box contains a single character. Between consecutive columns a vertical white space is left for *furigana* characters (JP: 振り仮名). These small-size characters, known as "ruby" in English, are annotations in *hiragana* or *katakana* over the main characters to hint the lecture of rare *kanji* for native readers, or for students. In this case, they are placed on the right side of each character. Each character is supposed to be centered inside their respective box, some versions even have a cross dashed grid inside each individual box to help the writer in this task. The same type of grid is used for individual *kanji* practice. Apart from characters; commas, full stops or small *hiragana*/*katakana* are supposed to be placed in the top right corner of their own box, except when this would mean placing it as the first symbol of the next column, in which case it would be placed in the bottom right corner of the last box with a character inside. These are in summary the basic writing rules. Information from Ｚ会作文クラブ (2012).



シロナガスクジラ

名無　権蔵

　シロナガスクジラ（白長須鯨）はクジラの
一種である。現存する最大の動物種（現存す
る最大の生物は植物のセコイア）。かつて地
球上に存在したあらゆる哺乳類の中で最大の
種であり、記録では体長三十四メートルのも
のまで確認されている。長須とは長身の意味
で、水面に浮かび上がる際に水上からは白く
見えることからこの和名がある。英語では腹
側に付着した珪藻によって黄色味を帯びて見
えることから"sulphur bottom"（硫黄色の腹
の意）とも呼ばれる。

特徴

　体長二〇～三〇m、体重一〇〇～一六〇t。
成体ではオスよりメスのほうが若干大きい。
流線型の体型をしており、尖った頭部をもつ。
細く長い胸びれ、横に広がった薄い尾ひれを

Figure 2.5: An example of correct use with *genkō yōshi* layout. We can see in point 5 how the rule for full stops placement is applied. (Gus Polly from Wikimedia, 2016)

We will take advantage of this knowledge about Japanese writing customs to apply it into the design of our dataset crowd-sourcing web application, using the grid layout for our writing canvas, as we will see in the implementation chapter.

Of course, the presence of strict rules does not mean they will be followed by everyone. In the end, each person writes following their own style, with a series of details and nuances in each stroke that are difficult to predict. Some hiragana characters even have a variety with a different number of strokes in its handwritten form. This “free style” is accentuated when it comes to annotations reserved for oneself. However, during elementary school, children are taught a specific common style called **Kaishotai** (JP: 楷書体). The *Kai* from the first character means “regular”. This traditional square style is characterized by its horizontal lines slating upwards going from left to right. Another styles of writing include: *Gyōshotai* (JP: 行書体), a semi-cursive flowing form of writing, and *Sōsho* (JP: 草書), a fully cursive style of calligraphy. The last one is often performed with a single curving brushstroke and may be difficult to understand for an untrained eye. This style deeply roots with the history of Japanese writing and has been studied for the creation of recognition tools using Machine Learning techniques, as we will see later. As one can imagine, if the recognition of *sōsho* style has been researched successfully, *kaishotai* standard writing should not be a problem if the model is adapted to the environment where it will be used and we have quality training data.

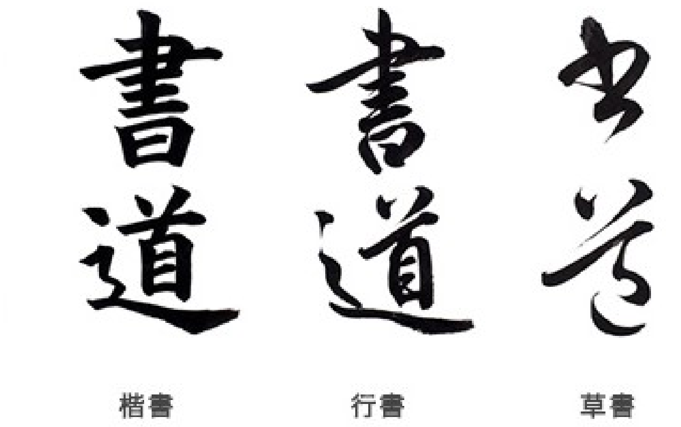


Figure 2.6: The three main styles of Japanese writing. From left to right: *kaishotai*, *gyōshotai* and *sōsho*. Image by 日本書道協会 (n.d.)

There are several others calligraphy styles of Japanese, often linked to a specific context or use, such as *kaishotai* (JP: 勘亭流), reserved for theatrical arts like *kabuki*, or the *kakuji* (JP: 角字) style, used for seals; but we will not cover them as they are far off from regular written script.

2.2.2 Relevant written script for Japanese datasets and the Sino-Japanese correlation

Once we understand the basic concepts and writing systems of Japanese, we are ready to start collecting character samples. The question we should ask ourselves now is: What characters classes? *Hiragana* and *katakana* syllabaries are the obvious starters for a Japanese charset (yet we could also decide if they should include diacritics variants), the real problem comes with *kanji*. According to the *Dai Kan-Wa Jiten*, probably the most relevant compilation of Sino-Japanese characters, there are around 50000 *kanji* in existence. Most of them are obscure, used for really specific things and rarely used even in China. Then, what number of characters is necessary to better represent the modern use of Japanese? After World War II, the Allied Forces of Occupation of Japan issued indications to the Japanese government for an integral reform of the writing system. The objective was to simplify the language learning for children and the access to literature and newspapers. An initial position based on the complete abolition of *kanji* evolved into a more conservative reform. The number of characters in circulation were reduced, some of them simplified, others completely discouraged. The result of this is a list of 2136 characters called **Jōyō kanji** (JP: 常用漢字), meaning "regular use letters". They are supposed to be the necessary *kanji* to have literacy in Japanese. In reality, the number of *kanji* used in modern written Japanese is more, especially in fields such as medicine or family names. For the last case a special set was created, the *Jinmeiyō kanji* (JP: 人名用漢字), literally "people given names". Children in elementary school learn a subset of the *jōyō kanji* of 1026 characters called *Kyōiku kanji* (JP: 教育漢字), distributed throughout all 6 grades. JIS digital encoding, that will be detailed in the next section, goes beyond this limit and reach the 6000 *kanji*. All in all, *jōyō kanji* is a good representation of modern Japanese and a more than decent reference for dataset creation. This does not mean problems should disappear instantly, as *kanji* is a world of exceptions. For instance, one particular issue that might be sensible for datasets is the existence of the so-called *itaiji* (JP: 異体字), a variant form of a *kanji* with the same meaning but different strokes. If we are to include that kind of variations in our dataset, we have to take care of the use of duplicated class names.

Apart from the selection of *kanji* classes, we have to care about the so-called **obsolete characters**. In the previously mentioned *kanji* reforms, the characters that were not included had the status of "officially discouraged", however, we can also find characters in a similar state outside the *kanji* domain. For instance, we have the characters 「ゐ」, pronounced *we*, and 「ゑ」, pronounced *wi*, in *hiragana*, and their counterparts in *katakana* 「ヰ」 and 「ヱ」. These exist because the sounds they represent existed in Japanese when they were created. After the sounds *we* and *wi* disappeared from the spoken Japanese, the characters continued to be used to represent *e* and *i* sounds. Nowadays their use is sparse but sometimes they are useful to represent foreign words containing the "v" or "w" sounds, like whisky or wine, though these can also be represented using the "u" sound with *dakuten* 「ヴ」. For datasets aiming modern Japanese, whether include them or not is a decision that only concern the researcher.

Finally, to be really meticulous, we must take into account the existence of some symbols that do not belong to either the *kanji* category or the two syllabaries. We could classify them as **repetition marks** or *odoriji* (JP: 踊り字), and they are fairly common in written Japanese. The most important one is called *noma*「々」, as looks like a fusion between *katakana* characters for *no* and *ma*. It is basically used to show the repetition of the previous *kanji*, *hiragana* or *katakana* character.

Not all datasets created are intended to be used for the recognition of modern Japanese. Recently, the use of machine learning techniques has been decisive for the recognition of old Japanese literature texts and other written materials from the same era. In this case, a recognition model is fundamental because we are talking about texts hardly understandable in modern times even for native Japanese speakers. The difficulty to read these works stems directly from the history and origin of the Japanese syllabaries. *Hiragana* and *katakana* are, at their core, simplified forms of *kanji*. Chinese characters were introduced into the Japanese archipelago around the first or second century A.D. Back when Japanese had no written system, Chinese characters were used to represent Japanese sounds. This is the oldest native Japanese writing system, often called by scholars **Man'yōgana** (JP: 万葉仮名), in reference to a homonymous ancient book of poetry written using this system. The *man'yōgana* system was extremely complex and convoluted. Under this system, some *kanji* were used for their meaning, but others were only read as a phonetic syllable, even two. Both of these were often used simultaneously in the same texts, without guidance apart from context. Even today a similar phenomenon to *man'yōgana* called *ateji* (JP: 当て字) occurs with some foreign words that can be written in *kanji*, only because of their phonetic resemblance. Both *hiragana* and *katakana* are a result of the development of *man'yōgana* into a simpler form. *Hiragana* was developed from *man'yōgana* written in the previously mentioned *sōsho* (JP: 草書) cursive style. *Hiragana* was easier to learn and use, but was not widely accepted by the educated elites or scholars, who preferred to use *kanji*. *Hiragana* became popular among women, who were not generally allowed to access the same education as their male counterparts. Court women used it regularly for writing informal correspondence and literature. This is the main reason why *hiragana* was at first know as *onnade* (JP: 女手), "women's writing", and also why even nowadays *hiragana* is considered "feminine". On the other hand, *katakana* was created by Buddhist monks who were seeking a way to reduce the workload needed to transcribe their ancient oral teachings. This task was time-consuming, as they had to write every single syllable as a *kanji*. They developed *katakana* as a shorthand, creating them by taking only a few strokes of each *man'yōgana* character. The evolution from *man'yōgana* characters to *hiragana* and *katakana* ones was not immediate, but rather progressive. In the middle point between *man'yōgana* and *hiragana* we encounter the *hentaigana* (JP: 変体仮名) characters. The meaning of this term is "variant kana", as in the past *hiragana* took more than one possible form for each sound. They were officially discouraged by the language reforms, but they are still sometimes used on traditional shop signs. In contrast, *katakana* has seen much less variation in its characters along the time. As cited in the “Related works” section, the most important research concerning the recognition of classical Japanese is carried over by the CODH from the ROIS. These texts, not only from literature but also from all kinds of fields, are written in *kuzushiji* (JP: 崩し字), “simplified characters”, which is another way to call the *sōsho* (JP: 草書) cursive style. With the language reforms of the 1900’s, *kuzushiji* was ruled out from Japanese schools curriculum. As a result, most native Japanese nowadays can not read books written or printed just 150 years ago. This center works in collaboration with the National Institute of Japanese Literature (NIJL) for the recollection of *kuzushiji* samples from bookstores, auctions and their own archives. Their most notorious work is Clanuwat et al. (2018), where a *kuzushiji* direct replacement of the famous MNIST dataset is created. The distinct point is that *kuzushiji* contains also *hentaigana*, which means many of the characters in the dataset have multiple ways of being written, therefore successful models have to be able to capture the multi-modal distribution of each class. This is the main reason why *kuzushiji* MNIST can be more challenging than the original MNIST, serving as an alternative benchmark for machine learning algorithms (Lamb, A., Clanuwat, T. Kitamoto, A., 2020). We could say this work is the most similar to the objective of this thesis, referring to the creation of a new dataset following the MNIST format. Their most recent project “KuroNet” is an end-to-end recognition model for entire pages of text in *kuzushiji* without preprocessing, making it easy to apply for real-world data (Clanuwat et al., 2019).

As we already know, Japanese *kanji* originated in China, and therefore there are many similarities between both written scripts. However, the pass of time and history have created significant **differences between Chinese and Japanese characters**. Traditional Chinese *hanzi* (the spelling of *kanji* in Chinese) were first introduced in Japan around the first century A.D. Nevertheless, it is believed that the first texts were not written in Japan until the fifth century A.D, probably through bilingual Korean officials. One concept used by both parts is the one of the “radicals”. A “radical” (JP: 部首), *bushu*, is a part of the *kanji* which is found in more than one character, sometimes in relation to a similar meaning. For example, the radical「言」,“statement”, can be found in both「話」, “to talk”, and in「語」, “language”. The radical system was made popular by Chinese *hanzi* dictionaries. For this reason, radicals are a common way of classifying characters for searching. There have been even attempts to use these sub-character elements as a model to improve Neural Language Modeling in Japanese. In Nguyen et al. (2017), a predictive language model using neural networks is built under the premise that sub-character information could improve the ability to capture generalizations. It is concluded that the effectiveness of this does depend on the properties of the *kanji* dataset. We also have in Grębowiec & Protasiewicz (2018) an example of the use of sub-character features for *kanji* recognition. In this case, each time a stroke is written by the user, the model tries to classify it based on a *kanji* dictionary. The process is repeated with every stroke until the number of possible *kanji* is reduced to one.

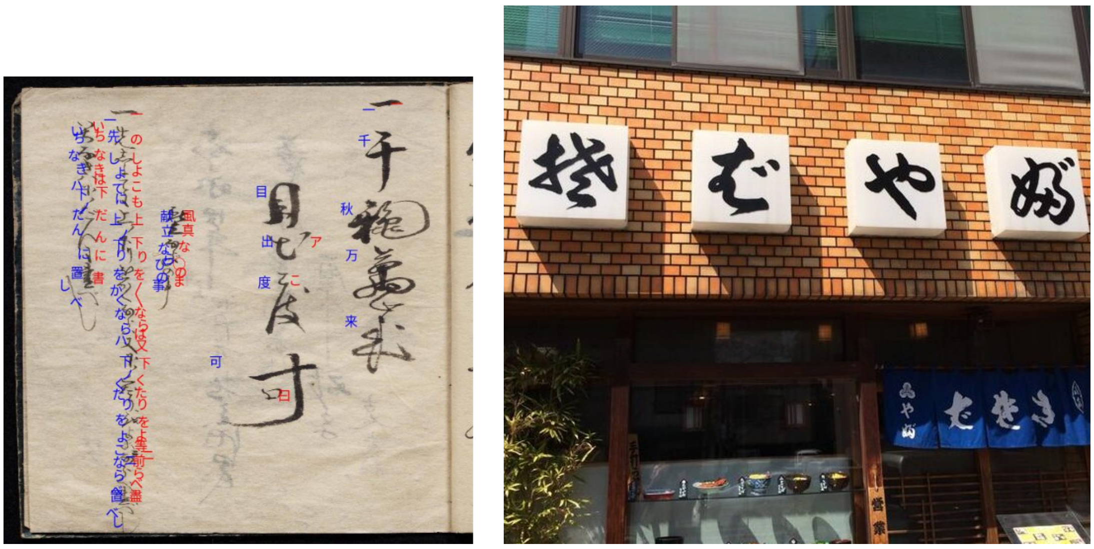


Figure 2.7: Left: An example of KuroNet being used over a full *kuzushiji* written page, from Clanuwat et al. (2019). **Right**: A soba noodles restaurant with a sign written in *hentaigana*. These type of signs are also a problem for smartphone's OCR software, typically used by tourists for machine translation. This one reads as 「生そばやぶ」, *kisoba yabu*, though *hentaigana* forms are not typeable in this document. Image from 変体仮名 bot (2020).

The main difference between Japanese *kanji* and Chinese *hanzi* was introduced when Mainland China decided to reform their writing system in the 1950's. They simplified the shape of the most common characters, some from existing variants, others completely from scratch. If a common radical was simplified, so was every character which contained it. The resulting script of this process is what we call Simplified Chinese. As a result, a large portion of Japanese *kanji* were no longer similar to their Chinese counterparts, with the exception of Taiwan and Hong Kong, where Traditional Chinese is still used. This distinction made the creation of potential bilingual OCR models much more burdensome, as most of the characters have to include at least an alternative form.

2.2.3 Japanese typesetting and the graphical density

Once we have profusely explained the details of the Japanese handwritten script, in this section we will give a brief introduction to **Japanese typesetting**, since it is of interest for some aspects of our work that will be explained here. Before the advent of computers, typesetting in Japan was done using a type of long, squared woodblocks with each character carved in one of the block sides. Because of this blocks, early printed characters where designed to fit in the shape, giving them a characteristic squared style. Some of the most important printed typesets are *Minchō* (JP: 明朝体), with triangular endings to the lines, similar to serif in Latin typefaces, *Shimbun shotai* (JP: 新聞書体), created in smaller sizes for newspapers, and *Goshikkutai* (JP: ゴシック体), in this case similar to English sans-serif. Fast-forward to nowadays, computers have monopolized the world of typesetting. We call **encoding** to the way a character set is mapped one-to-one inside our computer memory. One direct way of doing this is storing each character as a single integer in memory. The problem is that each integer is usually stored as a whole byte and for Latin alphabets, which take around 26 characters, it seems rather a waste of memory. Because of this, during the years encoding schemes have been constantly optimized and convoluted, resulting in a messy environment, and Japanese is no exception. In general, there are two main characters sets used to write Japanese: the **JIS** and **Unicode**. JIS stands for "Japanese Industrial Standard" (JP: 日本工業規格) and is a blanked term to name every character set published by the Japanese Standards Association. They were the first sets to allow the use of Japanese characters in computers, using only *katakana* back when machines were not powerful enough to support *kanji*. The development process of JIS has been rather chaotic, resulting in the creation of multiple ramifications of the standard along the years. Still, this has been the most used encoding in Japan for most of the time. Unicode is an effort to assign a unique code to every living writing system and unlike others, it follows a policy that each revision must be a strict set including the previous ones, avoiding any conflicts with older standards. The migration from JIS and Shift-JIS (a Microsoft variant) to Unicode is still in progress, mostly because of legacy and compatibility issues. This know-how about Japanese typesetting is not only useful for the design of our crowd-sourcing app and dataset's labeling, but also for a key concept: the **graphical density**. When designing printed or digital typesets for Japanese or Chinese, one important factor to care of is the legibility of *kanji* characters. If a *kanji* have too many strokes for the block within is contained, the white space between them could be too small, even resulting in overlapping strokes. We call this a density problem and in the worst cases could make the text incomprehensible. Although this might be seen only as a visual perception issue, is also a potential barrier for recognition (Qu J., Lu X., Liu L., Tang Z., Wang Y, n.d.). After collecting samples for our dataset, it is usual to resize the images to reduce data input. Even though this is a useful custom, we have to be careful that the general shape of the character is preserved. It is general knowledge that Japanese fonts should at least use 9 pixels of height to be legible. There have been already multiples complaints about this in videogames from the Japanese themselves. Some games designed for portable devices with small screens have *kanji* fonts so tiny that in some cases text could not be understood without the context. In printed media, it is specified that characters should measure 9 points or 8 points (a letterpress printing unit around 0.3 mm) tall for optimal readability. (W3C Standard for Japanese, 2012)

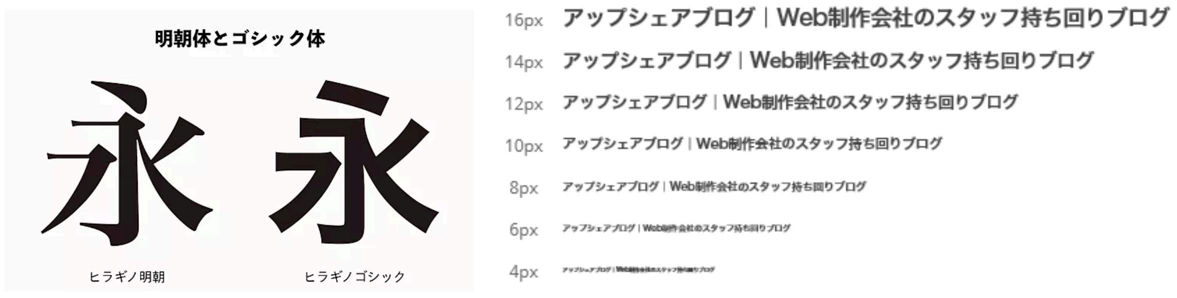

明朝体とゴシック体
永 永
ヒラギノ明朝
ヒラギノゴシック
16px アップシェアブログ | Web制作会社のスタッフ持ち回りブログ
14px アップシェアブログ | Web制作会社のスタッフ持ち回りブログ
12px アップシェアブログ | Web制作会社のスタッフ持ち回りブログ
10px アップシェアブログ | Web制作会社のスタッフ持ち回りブログ
8px アップシェアブログ | Web制作会社のスタッフ持ち回りブログ
6px
4px

Figure 2.8: **Left**: Difference between *Minchō* and *Goshikkutai* typesets Image by 中の人（2 号） (2017). **Right**: Comparison of font sizes in Internet Explorer, we can notice that stroke density of *kanji* turn the text unreadable between 10px and 8px. Google Chrome caps size to at least 10px to guarantee legibility. Image from アップシェア Blog (2017).

As a further comment, **recognition of machine-typed Japanese characters** such as those covered in this section is a long-standing research path with many years of history. The reason behind this is that the automatic recognition of document data for businesses can significantly lower operation costs and increase productivity. This need is pre-eminent in banking, insurance, postal service, etc. In Japan, the research on this topic is carried out by many research laboratories (such as Nippon Telegraph and Telephone (NTT) and ElectroTechnical Laboratory (ETL)), universities (such as Tokyo, Osaka and Nagoya Universities) and a few large companies (like Toshiba and Ricoh). The process follow the typical OCR pipeline described in the OCR section of this chapter. The first difference with English OCR starts with the location and segmentation step. In Japanese text images, a text line is segmented directly into a sequence of characters as word boundaries are not distinguishable. Feature descriptors for Japanese characters typically have large dimensionality due to the complexity of *kanji* and their stroke structure. Moreover, several characters are structurally similar. Regarding the classification step, structural analysis and pattern matching are examples of methods used traditionally before neural networks were widely used (S.N. Srihari, G. Srikantan, T. Hong and S.W. Lam, 1996). A long pursued effort of Japanese OCR was the creation of an omnifont or at least multi-font recognition model. For example, in S.N. Srihari, T. Hong and Z. Shi (1997) a general purpose recognition model for Japanese documents is proposed. It use a minimal error subspace and fast nearest-neighbour classifiers. The model is even said to have been developed with multi-lingual support, being possible it adaptation for similar oriental languages such as Chinese. Although machine-typed characters are not the main objective of this thesis, it is worth mentioning it as an example of previous efforts carried out in the field of Japanese OCR.

2.3 Introduction to Neural Networks

An **Artificial Neural Network (ANN)** is, in short, a system composed of interconnected nodes arranged in layers. We call each node a "neuron" because they are loosely inspired by the neurons of a biological brain. Neural networks are capable of, given a data input, learn through example and generalize from that point for the sake of making predictions. In this section we will detail the structure and in and outs of Neural Networks to later describe how they are used for actual Machine Learning. But first, we will provide a brief historical background.

2.3.1 Historical background

The concept of neural networks is older than it may seem. The history of these systems kickoff with the work of Warren McCulloch and Walter Pitts in 1943. These two researchers in computational neuroscience created a model for neural networks that is consider a foundation for both the understanding of the biological neural process and the application of neural networks to artificial intelligence. Later on that same decade, the neuropsychologist Donald O. Hebb developed a hypothesis based on the concept of neural plasticity that became known as Hebbian learning. Neural plasticity is the potential ability of the brain to undergo physiological changes. Hebb theorized, following previously proposed Santiago Ramón y Cajal's ideas, that neurons may grown new connections between them to store new memories. This hypothesis of unsupervised learning is nowadays well established and served as an inspiration for future computational models. From 1954, researchers started to use for the first time computational machines to simulate a Hebbian network. In 1958, the psychologist Frank Rosenblatt developed the **perceptron**, a key concept for neural networks. The Russian mathematician Alexey Ivakhnenko created jointly with other researchers the first **multilayer perceptron networks**, a model which will define how functional neural networks look like even today. At that point, machine learning stagnated due to two fundamental issues. The first was that it was discovered that perceptrons were not able to process exclusive-or logic, but most importantly, 1960's computers were incapable of handling large neural networks. The interest in neural networks were not revitalized until the discovery of the **backpropagation** algorithm by Paul Werbos in 1975. This algorithm, which will be explained later, has allowed the training of multilayer networks and keep being a standard nowadays. By the 1980's the computational power of digital electronics had grown enough to continue the development of feasible neural networks. In 1992, max-pooling is introduced to help in 3D object recognition in 2D images. Max-pooling is a technique of non-linear down-sampling, in other words, it reduces the dimensions of the data, the "samples", by combining the outputs of certain neurons. This algorithm is usually used in the design of **Convolutional Neural Networks**. Unlike the multilayer perceptron model, CNN employs the convolution operation in at least one of its layers. This type of model have proven to be really effective in processing visual and two dimensional data. Although CNNs were already described in the 1980s, they became relevant around this time, when processing power allowed their practical implementation. Other important model is the Recurrent Neural Network, where data is not only propagated forward, but also backwards. In 1997, the research team of Jürgen Schmidhuber used a variation of Recurrent Neural Networks to overcome the famous "vanishing gradient problem". Nevertheless, CNNs constitutes today the state of the art in machine learning as they have been the first to achieve human competitive performance on certain contexts. These ANN's related concepts will be explained with more detail in the subsequent sections.

As we can see, neural networks date back to decades, but research did not gain traction until recent years. The question we might ask is: Why are we living a resurgence now? The first factor is that we live in the age of *Big Data*, we need to handle huge amounts of data on a daily basis, and these algorithms need similar amounts to succeed. Secondly, these algorithms are massively parallelizable and can benefit from modern GPU technologies that simply did not exist when they were invented. Finally, we now have access to open source toolboxes like Tensorflow, which have extremely streamlined the implementation of these models for developers (Alexander Amini, 2020).

2.3.2 The multilayer perceptron model and backpropagation

The **multilayer perceptron model** is at the origin of every neural network structure nowadays. In order to understand this model, we have to understand first the concept of the **perceptron** itself. The perceptron is a linear binary classifier, or in other words, an algorithm that decides whether or not an input belong to an specific class. We call it linear because its predictions are based on a linear function or combination of the feature vector. In fact, we could say perceptrons are the "neurons" of our network. Before multilayer perceptron, the simplest model of neural network is the single-layer perceptron, another way for terming the perceptron algorithm. In Figure 2.9 we can see the different parts of this algorithm. First we have the inputs, which are all the numerical values we are going to feed our perceptron with. Linked with each value are the weights, which will be multiplied to the its corresponding input. These weights must be initialized from the start. If we need it, in this step we can add a bias value to be used, with the objective of shifting away the decision boundaries from the origin. In this case such value is 1. Once we have the results of the previous multiplications, in the next step we sum all of them to obtain what could be consider the perceptron output. However, usually this value is passed through an activation function before the end of the process. This function maps our value inside a range where the final output is expected to be. Heaviside step function is usually used in single-layer perceptron as activation function. Heaviside maps every positive value to 1 and every negative value to 0. Activation functions are particularly important in multiple layers models, as we will explain later.

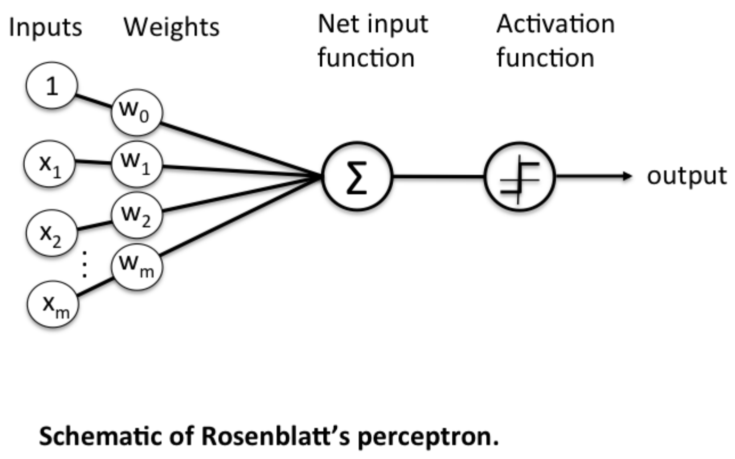


Figure 2.9: A depiction of the single-layer perceptron model. From left to right, we can see the inputs, labeled from x_1 to x_m, the weights, defined from w_0 to w_m, a Summation $\sum$ and an Activation function with its corresponding output. Image by Nazanin Delam (2016)

Once we know how the basic structure of a perceptron works, we can explain the multilayer perceptron model, and more importantly, how the training process is used to actually achieve what we know as Machine Learning. But before all this, what are we referring to exactly when talking about Machine Learning? We could say **Artificial Intelligence** is the field that focuses on building algorithms that can process information to inform future decisions. In short, these are the techniques that enables computers to mimic human behaviour. **Machine Learning** is just a subset of Artificial Intelligence that focuses in teaching an algorithm how to do a task without explicitly being programmed to do so. Also, the term **Deep Learning** has recently become popular in this field. Following the sequence, Deep Learning is a subset of Machine Learning that aims to automatically extract the useful patterns from data in neural networks that are needed to inform future predictions. (Alexander Amini, 2020)

Now we are ready to deepen in how a neural network *actually* works. We will try to explain it in the most intuitive way possible, assuming no background. Let us use as an example a seemingly simple task: to recognize a number from a binary image between 0 and 9. Our objective is not only to explain the general working of a multilayer neural network structure, but to actually understand what is behind the process and by what reasons the structure is motivated. First, we will explain the structure of the model, and afterwards, the training process algorithm.

We can define each "neuron" or node from our network simply as something that holds a number from 0 to 1. For our first layer, each neuron corresponds to the input of an image's pixels. If our image is 28×28 pixels size, then we have a layer of 784 neurons.

Each of them holds a number that represents the greyscale value of its corresponding pixel, from 0 for black pixels, to 1 for white pixels. We call ”activations” to each of these numbers inside the neurons of the network.

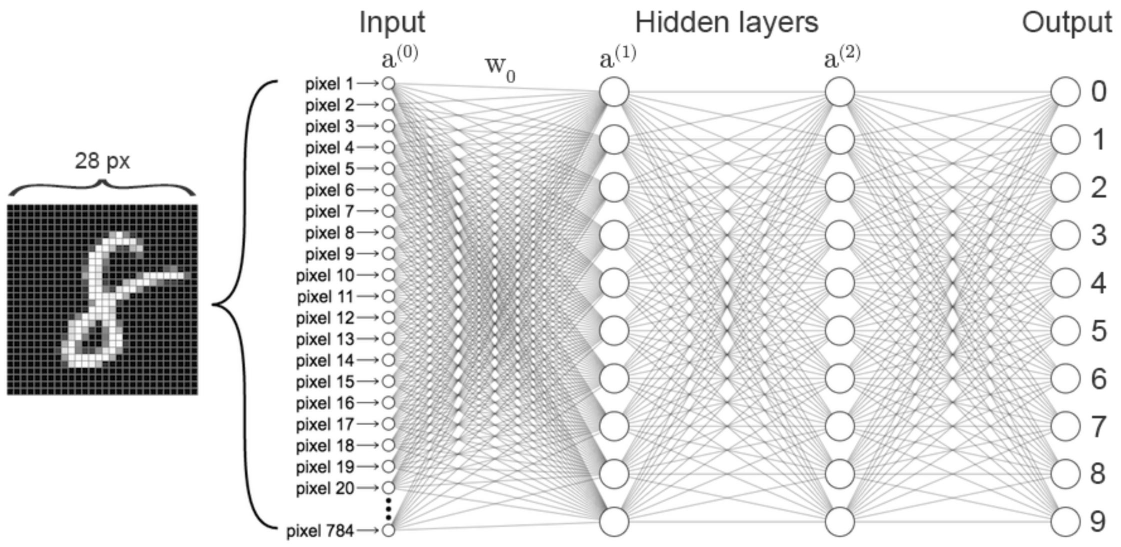


Figure 2.10: The described multilayer perceptron model. The digit's image is decomposed into an input layer, followed by two hidden layers and an output. We use the notation $a^{(layer)}$ to refer to each neuron activation, likewise the weights are represented by a w. Author's own creation based on the work of Andreas Refsgaard (2020)

Jumping over to our last layer we can see it is made up of ten neurons, each representing one of the digits we are willing to classify. The activation in each of these neurons, again ranging from 0 to 1, represents how much the system thinks that a given image corresponds with each digit. We call “hidden layers” to the ones laying in between the input and the output.

The activations in one layer determine the activations of the next layer through its connections. The heart of the network, as an information processing mechanism, comes down to how activations in one layer bring about activations in the next layer. This process is similar to the one taking place inside our brain, where when some group of neurons are fired, it can cause others to fire. Assuming our network is already trained, if it is fed with a certain digit's image, activation in each successive layer will provoke the activation in the final output layer, where the biggest activation should be that of the neuron representing the digit we used as input.

But why we expect our layers to behave like that? What are they supposed to be doing? When our brains recognise digits, it does so by breaking down them in smaller components. For example, an eight is made of two loops, while a nine only have one plus a straight line. In a perfect world, we hope each neuron in the hidden layers corresponds with one of these features. This way, a neural network only have to know which specific combination of features made up each digit. Of course, these features could be broken down into other subfeatures, such as dividing the loop into 4 edges. Again, in a perfect model, we could hope that each neuron from our first hidden layer corresponds to some subfeature, while the second hidden layer's neurons represent the bigger features we described first. In the end, whether or not our network decomposes the digits this way is difficult to know without knowledge of the training process. Breaking down things into layers of abstractions is not only a model for image recognition. For example, in speech recognition certain sounds can be taken as features, which eventually will be joined together in syllables adding another layer of abstraction to the equation. The network will combine these tokens into an output of words or sentences for recognition. Extrapolating to this thesis's topic, we could even identify these features as the *kanji's* radicals we cover in the previous section, if not a subset of them.

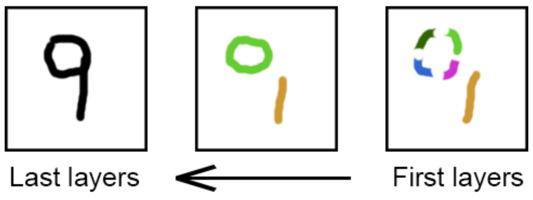


Figure 2.11: An example of a nine divided into features and subfeatures. The first hidden layer is responsible of the smaller subfeatures, like edges. Successive ones start to made up more complex features like loops, until the whole digit is computed. Author's own creation.

Once we understand what to expect our neural network to do, we can deepen in the behaviour of the layers themselves. As we said, a layer objective could be to know if a certain type of edge is present in one region of the image. The parameters we should tweak to make one layer express the presence of a feature have already been mentioned in our introduction to the perceptron concept. Giving an input layer with all its neurons connected to one neuron of the next layer, we assign a "weight", positive o negative, to each of these connections. These layers are also called "dense layers", as each neuron is connected with every neuron of the next layer. We then multiply each input neuron activation to its connected weight, and sum the results. If we wanted to detect whether or not an edge is present in one region, we could assign negative weights to the pixels around that area, so the sum is larger because the edge pixels are brighter than the surrounding pixels. The sum of these multiplications could be any number, but as we want every neuron to hold a value between 0 and 1, an Activation function is required. In short, we are measuring ”how positive” the weighted sum is. Sometimes we don't want a neuron to light up when this sum is only bigger than 0, maybe we need it to only meaningfully start from 5, for example. In that case, a bias value of 5 is added before using the activation function. All in all, this is how our single-layer perceptron model fits inside the multilayer perceptron one, repeating it for each neuron of each layer.

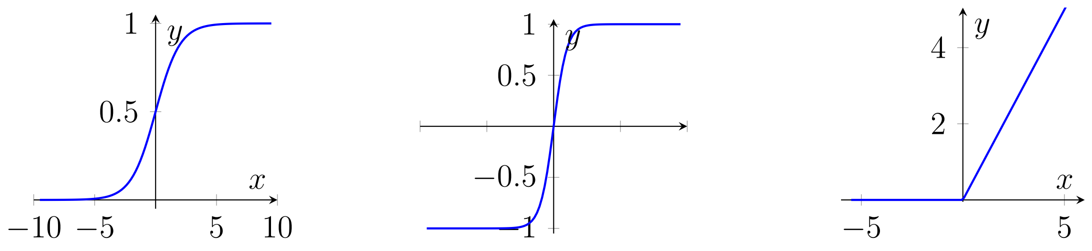
$$a^{(1)} = \sigma(w_{0,0}a_0^{(0)} + w_{0,1}a_1^{(0)} + \cdots + w_{0,n}a_n^{(0)} + b_0) \tag{2.1}$$

This expression represents the weighted sum of the input layer neurons activations. The current layer activation is represented with $a^{(0)}$, with its corresponding weights $w_{0,x}$. Our chosen bias is represented with b_0. Finally, the Greek letter sigma σ is commonly used to illustrate an activation function. The result of this operation is the activation $a^{(1)}$ of a neuron in the next layer.

A more notationally correct way of representing the 2.1 formula is by using matrices. We can take all the input layer activations into one single column and all the weights as a matrix, with each row corresponding to the connections between one layer and a particular neuron of the next layer. This way, the weighted sums of the activations in one layer can be calculated using the vector product of the matrix and the column. Also, we can organize the biases in another column, so we can sum them directly to the result of the previous vector product. We can then proceed to apply our activation function to each component of the resulting vector. This type of notation is very relevant, as modern GPUs optimize matrix calculations for us. (Grant Sanderson, 2018)

$$a^{(1)} = \sigma\left(\begin{bmatrix} w_{0,0} & w_{0,1} & \cdots & w_{0,n} \\ w_{1,0} & w_{1,1} & \cdots & w_{1,n} \\ \vdots & \vdots & \ddots & \vdots \\ w_{k,0} & w_{k,1} & \cdots & w_{k,n} \end{bmatrix} \begin{bmatrix} a_0^{(0)} \\ a_1^{(0)} \\ \vdots \\ a_n^{(0)} \end{bmatrix} + \begin{bmatrix} b_0 \\ b_1 \\ \vdots \\ b_n \end{bmatrix}\right) \tag{2.2}$$

Activation functions could be basically classified in two types: linear and non-linear functions. Linear functions have the equation $f(x) = x$ and therefore their output is not confined between any range. This type does not help with the complexity of usual data that is fed to the neural network. The most interesting and used type is the non-linear functions. They help the model to generalize or adapt to a variety of data and to differentiate between the output. Some functions commonly used nowadays are Sigmoid, Tanh and ReLU. The **Sigmoid** function is a S-shape looking curve that exists between 0 and 1. Its use was motivated by the biological analogy of neurons being either active or inactive. This type of range is also useful when we want to predict a probability as an output. A popular generalization of Sigmoid is the **Softmax** function, as plain Sigmoid use is not that common anymore. The **Tanh** function, or hyperbolic tangent, exists in the range between -1 and 1. The advantage in this case is that negative inputs will be mapped strongly negative, while zero inputs will be mapped near zero in the graph. The **ReLU** function, or Rectified Linear Unit, maps to 0 if our output is less than zero, keeping the value otherwise. ReLU is right now the most used activation function in the world since it is a common part of Convolutional Neural Networks. One of its main advantages is that it has a computationally low cost in comparison with Sigmoid and Tahn. Usually activation functions are set to be used on a layer level. While ReLU is used for the hidden layers, Softmax is the standard for the output layer. (Sagar Sharma, 2017)



(a) Sigmoid **(b)** Tanh **(c)** ReLU

Figure 2.12: The main activation functions. Tikz graphs plots by ambs (2019)

Every neuron from the input layer is connected to every neuron of the next layer, each connection with its own weight and bias. This adds up to a respectable 784×10 weights, not taking into account the successive layers of the network. With them, we are talking about (784×10)+(10×10)+(10×10)=8040 weights and 10+10+10=30 biases. We can already deduce why machine learning is always considered a computing intensive task.

Previously, we have mentioned that the weights of the network should be initialized by the start of the process. In fact, when we talk about training our model, we are referring to finding the right weight and biases.

The first concept to come to mind when thinking about training a neural network is the method to follow. What we want in our case is, essentially, an algorithm where you could show a certain amount of training data, which comes in the form of images of handwritten digits along with labels for what are supposed to be, and it will adjust those 8040 weights and 30 biases in order to improve its performance on the training data. Hopefully, this layer structure will mean that what it learns also generalizes to images beyond that training data. In this section, we will explain the process to come out with this algorithm and its functioning. To start things off, we are going to initialize all the weights and biases randomly. Needless to say, in this state the network is going to perform worse than expected, with a kind of messy output. To correct this, we need to define a cost function between the expected network's output and the real one. In other words, what we do is add up the squares of the differences between those two outputs. This is the cost of a single training example. This sum is small when the network confidently classifies the image correctly, but is large when the network does not know what it is doing, like in our case. If we consider the average cost over all the training data, we get a measure of how lousy our network is. Therefore, what we want to do to improve the performance of our network is to reduce the value of our cost function. Sometimes, we can find the input that minimizes the value of a function explicitly, but this is not feasible for a function with thousands of inputs like ours. What we do in this case is to choose any input in the function and decide in which direction to step to make the output lower. In a 2d function this could be done easily following the slope reference, but in a function with multiple inputs like ours we have to find whatever direction that decreases the cost most quickly. In multi-variable calculus, the "gradient" of a function gives you the direction of steepest ascend, i.e., the one that increases the function most quickly. On the contrary, if we take the negative of that gradient, it gives us the direction to step that decreases the function most quickly. This is called **gradient descent**. Eventually, we will reach a minimum. The problem is that it is not guaranteed to be the global minimum, as it may be simply a local minimum. Finding the global minimum is much more difficult than finding a local one. In other words, we could have found a valley in our function, but not the deepest valley. This process of taking small steps until reaching a minimum is also the reason why neurons have continuously ranging activations rather than simply on and off states, so these steps are much more smooth.

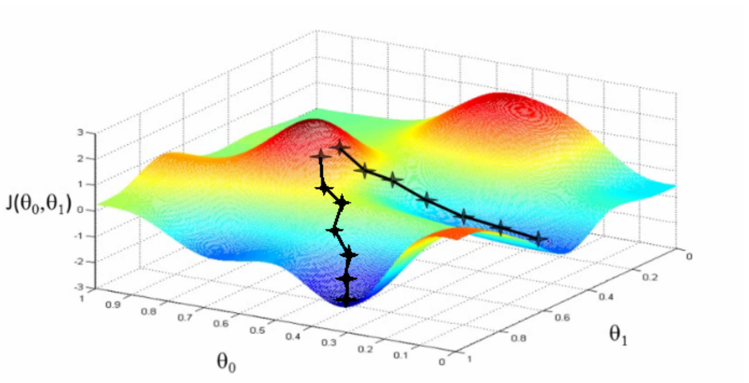


Figure 2.13: A 3D representation of the cost function value $J(\theta)$ with two inputs θ_1 and θ_2. We can see that two starting points can lead to two different local minimums, without the certainty of either of them being a global one. Original image by Andrew Ng (2018), modified by Albert Lai (2018)

Notation wise, we can organize all the weights and biases of our network into a large column vector. The negative gradient of the cost function is just another column vector. Adding or subtracting the second one values from its corresponding inputs of the first vector will minimize the average cost, and therefore, improve our network's output for all the samples. Using this, we can represent the gradient descent in a non-spacial way, avoiding the constraints of 3D space. The algorithm for computing this gradient efficiently, which is the core of how neural networks learn, is called **backpropagation**.

$$\vec{W} = \begin{bmatrix} 2.4 \\ -1.1 \\ 1.7 \\ \vdots \\ -2.16 \\ 3.8 \end{bmatrix} \qquad -\nabla C(\vec{W}) = \begin{bmatrix} 0.18 \\ 0.45 \\ -0.5 \\ \vdots \\ 1.3 \\ -0.4 \end{bmatrix} \quad \vec{W} + (-\nabla C(\vec{W})) = \begin{bmatrix} 2.58 \\ -0.65 \\ 1.2 \\ \vdots \\ -0.86 \\ 3.4 \end{bmatrix} \tag{2.3}$$

where: $\vec{W}$ $\rightarrow$ Column vector of weight values.
$-\nabla C(\vec{W})$ $\rightarrow$ Column vector of negative gradient of the cost function.

Because thinking about the gradient vector as a direction in 8040 dimensions is beyond the scope of our imagination, we can think about it in another way: The magnitude of each component of the negative gradient vector is telling us how sensitive the cost function is to each weight and bias. For example, if the negative gradient for a specific weight is 4.20 and for other is 0.10, it means that the cost of the function is x42 times more sensitive to changes in the first weight than in the second.

Because the cost function involves averaging a certain cost per example over all the ten of thousands of training examples, the way that we adjust the weight and biases for a single gradient descent step also depends on every single example (Grant Sanderson, 2018). For the sake of simplification we will explain the backpropagation algorithm through a single example. Starting with an untrained network, the results in the output may look almost random. We can not change this directly, as we only have influence over the weights and biases. However, we can at least hope how the changes in the output activations should be. In short, we want the activation representing our image's digit to go up and the rest to go down to correct the error. The change should be proportional to how far away is the activation to its target value. Let us focus on the neuron representing our image's digit, and therefore the one we wish to increase. Looking at the connections to this neuron coming from the previous layer, we would want to increase the weights of the connections with higher activation neurons, as they are the ones which could have a stronger influence over our target. The next thing to look at could be the previous layer activations. If everything connected to our target neuron with a positive weight increases its activation, and decrease when the weight is negative, then the target neuron will become more active. Of course we can not change directly the activation values, but it is worth to keep track of them. The desired changes determined by the target neuron are added together with the desired ones determined by the rest of neurons of the output layer, again, in proportion to the corresponding weights and in proportion to how much each of those neurons needs to change. This way, we get a list of adjustments that we want to happen to the layer. Once we have this, we can recursively apply the same process to the relevant weights that determine those values, repeating the algorithm and moving backwards through the network. This is what we know as **backpropagation**. As this is only for one sample, we have to repeat the same back-prop routine of updating weights for every training sample, and then average together all the changes. This average is, in fact, the negative gradient of the cost function.

In practice, computers do not calculate the influence of every training example over the network's weight and biases at once, as it would take too much time. Instead, usually the training data is shuffled and divided in "mini-batches" of a certain number of samples. Then, we compute in each gradient descent step only one of these subdivisions each time. Each "mini-batch" gradient descent is only an approximation of the full training dataset, but it also gives us a significant computational speed up. In the end, you will still find a local minimum, only that following a less direct path.

But, what is exactly a batch? A **batch** is compromised of a certain number of samples. While training, a for-loop iterates over these samples, making predictions for each one. At the end of the batch, we compare the predictions with the expected outputs to obtain an error we can use to calculate the gradient descent. The training dataset can be divided into one or more batches. If the batch size is equal to the dataset size, i.e, all the samples are used to create a single batch, the algorithm is called **Batch Gradient Descent**. If the batch size is equal to only one sample, is called **Stochastic Gradient Descent**. If the batch size is more than one sample but less than the size of the full dataset, then is called **Mini-Batch Gradient Descent**. Some popular batch sizes for this variety are 32, 64 and 128.

Another important concept in training is the **epoch**. An epoch is a complete pass of the learning algorithm through the training dataset, and therefore is compromised of one or more batches. Likewise, one epoch pass means that every sample of the dataset has had the chance to update the model parameters. Usually, the training process involves performing more than one epoch through the dataset. This means that a typical training for an ANN is compromised of two for-loops: one iterates over the number of epochs, where each loop proceeds over the dataset, and the other, within the latter one, iterates over each batch of samples. (Jason Brownlee, 2018a)

Both the number of batches and the number of epochs are considered **hyperparameters**. We call hyperparameters to the variables that are set before training and determine the network structure or the training routine (Pranoy Radhakrishnan, 2017). For example, the number of epochs tends to be traditionally large, around hundreds or thousands, so the model's error can be sufficiently minimized. Other important hyperparameters are the number of neurons in the hidden layers and the number of hidden layers themselves. The choice of the number of neurons in each of these layers is important for our network performance. Even the number of hidden layers is not an evident choice. In practice, these are chosen by experimental tests, adding more when the number of layers seems not being able to capture a dataset's complexity. In general, usually one hidden layer is sufficient for the majority of problems. The more hidden layers are present, the more difficult the training becomes. However, there do exist some guidelines to choose the number of hidden neurons. Using too few neurons in the hidden layers will result in **underfitting**. Underfitting occurs when there are too few neurons to model the training data nor generalize to new data. On the other hand, having too many neurons in the hidden layers may result in **overfitting**. Overfitting occurs when the model has so much information processing capacity that learns the detail and noise in the training data to the extent that it affects negatively the model's performance (Jason Brownlee, 2016). Usually, we want the number of hidden neurons to be around 2/3 the size of the input layer, plus the size of the output layer, but never more than twice the size of the input layer. (*How many hidden layers should I use?*, n.d.)

Overfitting is often regarded as the most serious issue, as underfitting can be easily detected given a good performance metric. One straightforward way to detect if our model is overfitting is if the error in the training set is low but not so in the test set. There are different techniques to limit overfitting. One is to hold until the end of the training a validation dataset to test the model with unseen data. However, **Dropout** is probably the most important technique to avoid overfitting. It consist in randomly dropping out some nodes and its connections to other layers during the training based on a probability. In effect, each update to the layer is performed with a different "view" of it (Jason Brownlee, 2018b). Although dropout makes the training process noisier, it has proven to significantly reduce overfitting. Dropout is also considered a hyperparameter, taking usually a value between 20% and 50%. A lower value will have a minimal effect, while a higher one could result in under-learning.

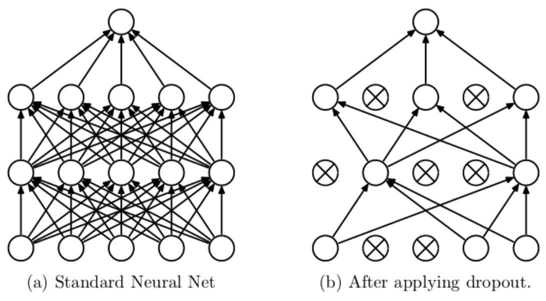


Figure 2.14: A depiction of an ANN before and after dropout. Image from Srivastava et al. (2014)

2.3.3 Convolutional Neural Networks

Convolutional Neural Networks, **ConvNets** or simply **CNN** are a class of neural networks that have proven to be very effective in the field of image recognition and classification. For this reason, is in the best interest of us to employ them for our character recognition scope. CNNs are also inspired by the biological neural organization of the Visual Cortex, in the sense that individual neurons respond to stimuli in a restricted area of the visual field known as the Receptive Field. In short, a CNN is a neural network that uses convolution in place of matrix multiplication in at least one layer. The CNN's capture of images' spacial dependencies stems from a better fitting to image datasets thanks to the reduction in the number of parameters and reusability of weights. CNNs have also been used for non-images, though not in the same quantity. The very first significant iteration of a Convolutional Neural Network was the LeNet architecture, developed by Yann LeCun in the 1990s. At that time, LeNet was used mainly for characters recognition tasks such as ZIP codes or digits. Although recently new architectures that improve performance have been proposed, such as VGGNet, the LeNet architecture remains to be a foundation for current models, as they all use the same main concepts.

The structure of a CNN is typically comprised of 3 main sections: the Convolution layers, the Pooling layers and the Fully Connected layers. Understanding how these basic building blocks work is essential to understand modern CNN models. We will now delve into each of them following the network order.

The way of organizing the input image for CNNs is no different than in standard neural networks. Pixel values are plotted along a matrix with the same size as the image, only that this time we will not turn it into a layer yet. RGB images have three channels of color and therefore use three pixel matrices.

We have already previously explained how the **convolution** operation is performed over an image using a kernel in Section 2.1.2, for OCR preprocessing. Here in CNNs the process is similar, but the intention is not. The objective of convolution in CNNs is to extract high-level features from the input image. The advantage over traditional ANNs is that we don't take features individually, such as pixels, but within a receptive field. This way the spatial information of the features is preserved. Conventionally, CNNs use a first convolution layer for low-level features, such as edges or orientation, and a second layer for high-level ones. Each custom structure adds a certain number of layers until the network holds a proper understanding of the images in the dataset (Sumit Saha, 2018).

Mathematically, a convolution is an operation on two functions that produces a third function expressing how the shape of one is modified by the other, which can be represented as follows:

$$y(t) = (f * g)(t) = \int_{-\infty}^{\infty} f(\tau)g(t-\tau)d\tau \tag{2.4}$$

where: $f \rightarrow$ Input function.
$g \rightarrow$ Kernel function.
$y \rightarrow$ Feature map.

This expression means that the resulting feature map y is defined as the integral of the product of input f and kernel g after one of them is reversed and shifted. In computers the inputs are discrete, so a discrete convolution is defined in that case.

In CNN terminology, the result matrix from the dot product between the input and the kernel is called “Feature Map”, “Activation Map” or “Convolved Feature” (Ujjwal Karn, 2016). This feature map is always smaller than the input image, as edges are not computed in this case. We call “Stride Length” to the number of pixels the kernel moves each step from left to right until it parses the complete row. Moving on, it hops to the start of the next row and repeats the process until the entire input is traversed. In case we are dealing with a RGB image, the process is carried out simultaneously for the three matrices of each color. The results of the dot product between the input matrices and the kernel over the same position are summed and added to its corresponding position in the output feature map. Although there some padding techniques to output a feature map the size of the input, usually it is kept small to save computational cost.

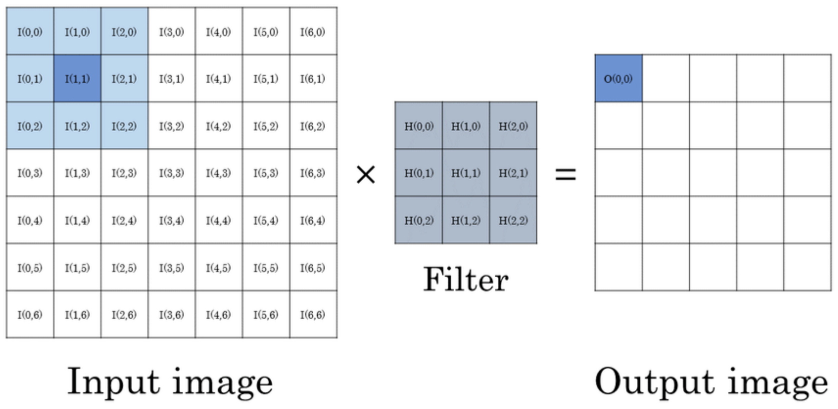

I(0,0) I(1,0) I(2,0) I(3,0) I(4,0) I(5,0) I(6,0)
I(0,1) I(1,1) I(2,1) I(3,1) I(4,1) I(5,1) I(6,1)
I(0,2) I(1,2) I(2,2) I(3,2) I(4,2) I(5,2) I(6,2)
I(0,3) I(1,3) I(2,3) I(3,3) I(4,3) I(5,3) I(6,3)
I(0,4) I(1,4) I(2,4) I(3,4) I(4,4) I(5,4) I(6,4)
I(0,5) I(1,5) I(2,5) I(3,5) I(4,5) I(5,5) I(6,5)
I(0,6) I(1,6) I(2,6) I(3,6) I(4,6) I(5,6) I(6,6)
×
H(0,0) H(1,0) H(2,0)
H(0,1) H(1,1) H(2,1)
H(0,2) H(1,2) H(2,2)
Filter
=
O(0,0)
Input image
Output image

Figure 2.15: Example of convolution operation between a 7x7 input image and a 3x3 kernel, resulting in a 5x5 output matrix, also called feature map. Image retrieved from Baskin et al. (2017)

We know that convolution is used in computer vision to highlight certain features of images, such as edges or depth, by using some specific kernels. In CNNs, however, kernels are not predefined, but trained. To understand how is this implemented, we have to look at how a **convolutional layer** works. We could say that a convolutional layer is comprised of three main components: the local connectivity, the shared weights and the convolutional operation itself.

Local connectivity is a key concept for understanding how this layer works. Usually in traditional ANNs, every neuron of the input layer is connected to every neuron of the next hidden layer, following the dense layer structure. Local connectivity means that one single neuron from the convolutional layer is only connected to a certain quantity of neurons of the previous layer, taking only their values into consideration. This way, we are defining a weights’ kernel over the previous layer, whose size is the number of neurons connected with the next one. If we follow this local connectivity structure for every neuron of the convolutional layer, we can see that we have in fact defined a convolutional operation where the resulting feature map is made up of the values contained in each neuron from the said layer. Unlike ANNs, where weights are different for each connection between layers, CNNs use a system of **shared weights** to force every connection of a hidden layer to use the same weights. Weights in this case behave as a particular kernel, whose values change when the network is trained, allowing the network to adjust them to detect certain features. Since weights are shared, the network does not have to learn the kernel values of each neuron in the layer separately to detect the features, as they all use the same ones. As the size of this kernel depends on the number of connections to each neuron in the convolutional layer coming from the previous layer, we can define it as a hyperparameter.

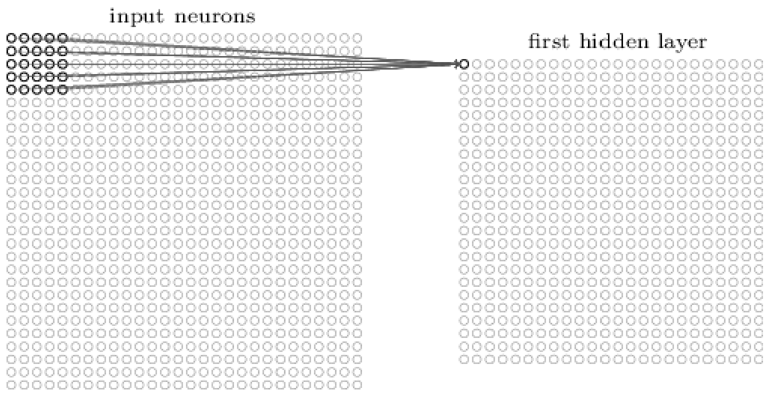


Figure 2.16: An representation of local connectivity. In this case, we see the connections create a kernel of size 5x5. Each of these carry a weight that is multiplied by its correspondent input layer activation to output the hidden layer neuron, also called feature map. Image retrieved from Michael Nielsen (2019)

On the contrary, if every single neuron of the hidden layer is connected to every neuron of the input layer, we call the structure “global connectivity”.

Another new kind of layers that are introduced by CNNs are the **Pooling layers**. In general terms, a pooling layer is a form of non-linear downsampling. In other words, they use dimensionality reduction to decrease the computational power required for processing the data. The reduction of output dimensionality may seem detrimental to CNNs’ purpose of taking into account spatial information, but at last, the exact location of a feature is less important than its rough location relative to other features. Moreover, this also helps to extract the dominant features which are invariant to rotation or position.

The process of pooling is done using a filter that slides over the input matrix. Usually a kernel of size 2×2 with a stride length of 2 is used for this purpose. Once the kernel is positioned, we can use two types of pooling: Average Pooling or Max Pooling. Average pooling returns the average of all the values covered by the kernel. On the other hand, Max pooling returns the maximum value of the ones covered by the kernel. Max pooling also offers the advantage of noise suppression, thus it has become the most common technique to use. It is a custom to periodically insert these layers between successive convolutional layers in the CNN architecture.

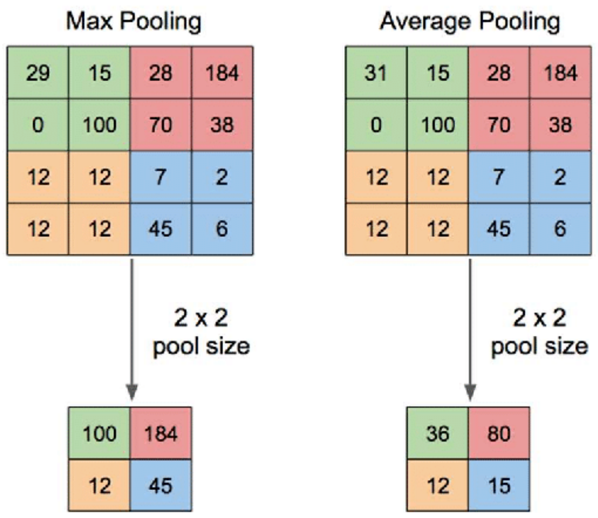


Figure 2.17: An example of Max Pooling and Average Pooling with a kernel of size 2x2 and a stride of 2 performed over a 4x4 feature map. Image retrieved from Yani et al. (2019)

The last parts of a traditional CNN structure are the **Fully connected layers**. These type of layers are the same as those used in the multilayer perceptron model, with a softmax activation function in the output. The name “fully connected” points to the fact that every neuron of the layer is connected with every neuron of the previous layer. These layers are used after the succession of convolutional and pooling layers to perform the classification using the extracted features. Additionally, fully connected layers are an easy way of learning non-linear combinations of these features. Most features coming from convolutional layers are adequate for classification, but a combination of them might be even better. Before using these layers, we shall flatten the output matrix from the previous layers into a column vector. We can then fed this vector to the fully connected layers for classification. Softmax is used in the final output layer as activation function to ensure that the sum of output probabilities is equal to 1. Meanwhile, ReLU is often used as activation function in the hidden layers.

The use of this function helps to mitigate the effects of the famous **Vanishing gradient problem**. The vanishing gradient problem happens when the use of certain activation functions make the gradients of the loss function vanishingly small, thus making the network difficult to train. The problem becomes worse while more layers are added, as the error propagates backwards from the output and makes the gradient decrease exponentially. While Sigmoid can potentially provoke this problem, ReLU is a good countermeasure due to its limited range. (Chi-Feng Wang, 2019)

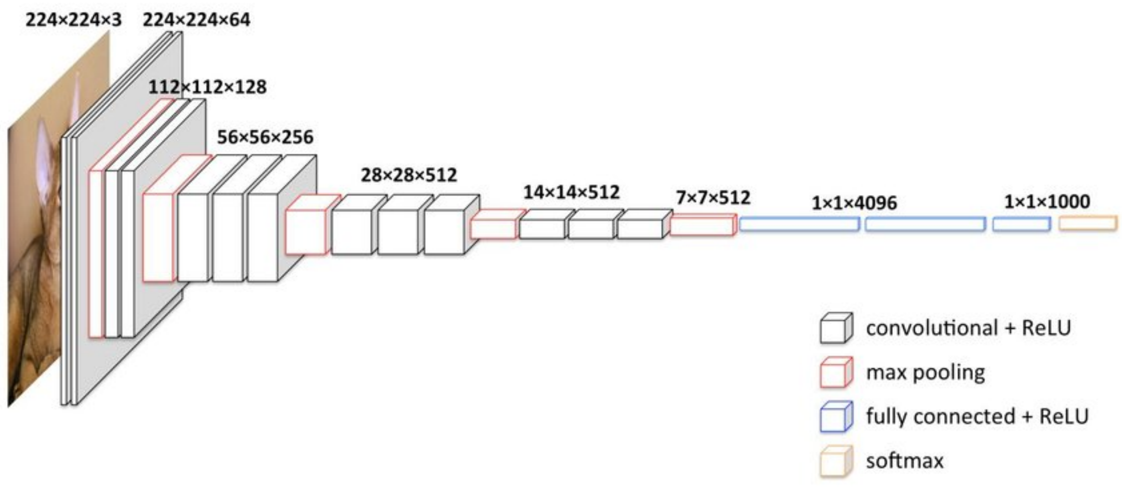


Figure 2.18: A representation of the VGG-16 architecture following the described Convolutional Neural Network model in this section. We can see how the input dimensionality get smaller with each successive layer. Image retrieved from Shi et al. (2018)

All in all, CNNs create a pipeline that is capable of segmenting images' features through different convolutional and pooling layers to be properly used in classification. We can tune this structure for our needs through some hyperparameters such as the number of filters, the filter shape and the pooling filter shape. Layers nearer the output tend to have more filters, as feature map size increases with depth. The filter shape is usually chosen based on the dataset content. We must also be careful with the size of the pooling filter, as larger sizes could lead to excessive information loss.

3 Methodology

This chapter is a record of the materials and technologies employed during the project, and the methodology followed when using each of them. It is made up of 4 different sections: Section 3.1 summarizes the materials that will be described, Section 3.2 lists the Japanese Characters Datasets used for the experiments and details its contents, Section 3.3 gives a brief background about the tools used to build the Neural Networks models and Section 3.4 describes the technologies chosen for the development of the web application and the reasons behind it.

3.1 Introduction

In the following sections we will review the materials and technologies used for the implementation of our project. We could divide these into three categories: the datasets, the ANN's software and the technologies used for the crowd-sourcing web app development. Datasets could be labelled as materials, though they are collected using digital devices and software. We will briefly cover the background of each of these technologies, followed by the purpose of using them. It is important to address each of these categories in order to understand the implementation that will be explained in the next chapter.

3.2 Review of Japanese Characters Datasets available

Datasets are the primary source material of our research, without them, nothing would make sense. As we previously mentioned, the success of a recognition model depends mostly on the use of quality and considerable training data. In this sense, Japanese is an additional challenge. This is due to the lack of accessible, sizeable and public training datasets for this language. This is because the creation of datasets for far-east languages such as Japanese or Chinese is an extremely time and money-consuming process due to the high number of different classes to take into account. Although recently new public datasets have been made available, we are still far from the numbers of western languages. (Velek & Nakagawa, 2002)

First, we will cover the previously mentioned **Kuzushiji-MNIST dataset**. In section 2.2.2, we already explained the concept of the *kuzushiji* writing style, yet we still don't know about the meaning of the MNIST part. The MNIST dataset (Modified National Institute of Standards and Technology) is a dataset of handwritten digits released in 1998. It includes 60000 examples for the training set and 10000 for the test set. This data was retrieved initially from mailed paper forms, then normalized into a 20×20 pixel box and, at last, centered in a 28×28 image using the center of gravity. With the pass of the years, it has become a standard to test ANN's models, and thus we are talking about it today (Michael Garris, 2019). The Kuzushiji-MNIST dataset is designed as a drop-in replacement for the MNIST dataset, following its same format (28×28 images in 10 classes). Released in 2018, it has a total of 70000 examples. Although limited to 10 classes, Kuzushiji-MNIST is still a substantial challenge for ANN's models due to the multiple ways of writing a single character class in cursive style. The fact that we will be using old handwritten script to test a model does not really matter, as this multi-modal distribution for each class is also common in modern Japanese and could be beneficial for our testing.

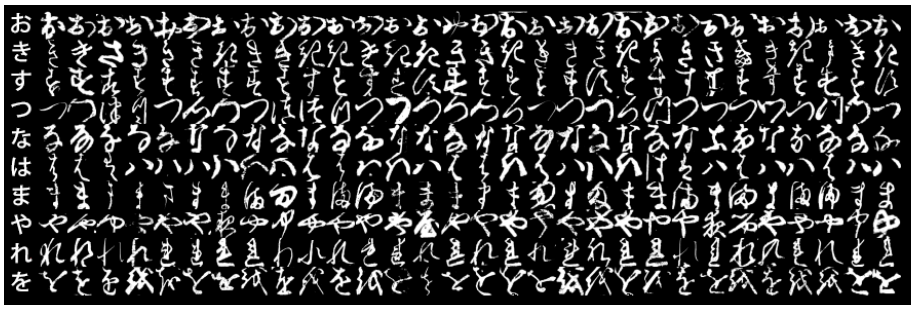


Figure 3.1: The 10 classes of Kuzushiji-MNIST with its modern counterpart on the left. Image from Mikel Bober-Irizar (2019)

The ROIS-CODH has also developed the Kuzushiji-49 and Kuzushiji-Kanji datasets simultaneously. Kuzushiji-49 consist of 49 hiragana classes, while Kuzushiji-Kanji is made up of 3832 *kanji* classes. Despite having a larger amount of samples, these datasets are highly imbalanced (each class has a very different number of examples), and thus they are less optimal for training experiments. It is interesting to note though, that Kuzushiji-Kanji images have a larger size of 64×64. This is an example of trying to avoid a problem of *kanji* graphical density, as described in section 2.2.3. We will see in a moment how this size pick is shared by other datasets.

Apart from these newly created *kuzushiji*-specific cases, public Japanese databases keep being scarce. Japanese companies that develop OCR systems usually have their own datasets, but they are not open to the public. That leaves us with only two well-known, public and sizeable datasets: the JEITA-HP dataset and the ETL dataset (Kitamura et al., 2015). It is our hope that the development of our open-source crowd-sourcing web application will help researchers to easily create new public Japanese language datasets in the future and thus improving the current situation.

The **JEITA-HP dataset** was originally collected by Hewlett-Packard Japan and later released by the Japan Electronics and Information Technology Industries Association (JEITA). JEITA-HP consists of DATASET-A, with 480 writers, and DATASET-B, with 100 writers. The two datasets were collected under different conditions, with DATASET-B being written more neatly than DATASET-A. The entire database consist on 3214 character classes, with 3306 images per writer. Each one of these have a resolution of 64×64 pixels, which are encoded into 512 bytes. In general, JEITA-HP has not been used as much as others, despite having been created many years ago. Sadly, it seems like JEITA-HP is no longer easily accessible in spite of being public.

On the other hand, the **ETL dataset** is still available and has been the standard for OCR research of Japanese *kanji* characters for many years. The ETL database is a group of datasets collected by the Electrotechnical laboratory (nowadays known as the AIST) under the cooperation with the JEITA, universities and other research organizations for character recognition from 1973 to 1984. The database is comprised of 9 datasets. Each one of them consists of a combination of *hiragana*, *katakana* or *kanji*. In total, we are roughly talking about 1.2 million handwritten and machine-printed characters. For our research, we will take a look at ETL-1, ETL-8 and ETL-9 datasets. ETL-1 contains handwritten *katakana* samples from 1445 writers, apart from other ASCII characters. ETL-8 contains handwritten *hiragana* and *kanji* from 160 writers. As there are 75 *hiragana* classes and 878 *kanji* classes, they add up to a total of 152,480 samples. ETL-9 is similar to this last one, including 3036 *hiragana* and *kanji* classes from 4000 writers. The characters of these datasets are labelled by their unique Shift-JIS codes. Each image is an isolated grey-scale character of size 64x64. Preprocessing wise, all samples have been pre-segmented and centered, and, to maximise contrast, binarized using Otsu's method. (AIST, 2014)

Luckily, we have been granted access to the ETL datasets after requesting it from AIST, therefore we will be able to use it in our experiments.

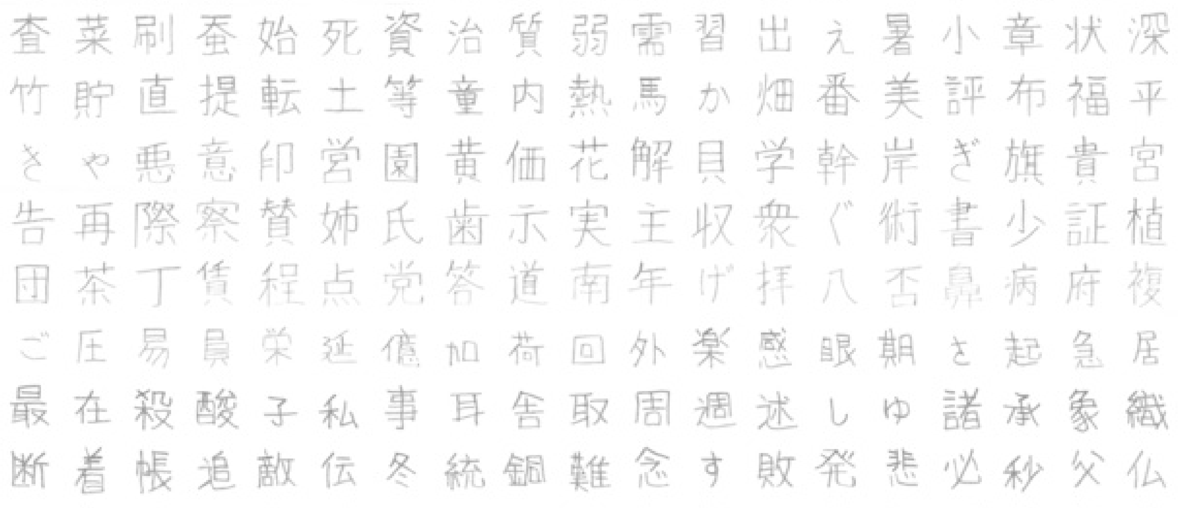


Figure 3.2: *Kanji* and *kana* samples taken from ETL-8G dataset. Image from AIST (2014)

These datasets are rather big in terms of samples. The quality of data and its preprocessing is important, but so is the quantity of it. As it is remarked in Velek & Nakagawa (2002), having large training sets significantly increases the recognition rate for Japanese character classifiers. We will carry out our classification experiments using Kuzushiji-MNIST and ETL in the next chapter. If an ANN structure is capable of accurately handling both Kuzushiji-MNIST and ETL, we could confidently say that it is prepared for most of the Japanese script given proper training data.

Both Kuzushiji-MNIST and ETL are off-line character pattern datasets, which means we only have the final visual image of the character. It is traditionally related with OCR, as characters are scanned directly from a paper or other physical medium. On the other hand, an on-line character pattern is information about the movement that created a character. This means no visual information is included in the dataset. An on-line representation makes no sense for printed characters, and for handwritten ones the information has to be captured during the writing process. If Japanese off-line pattern datasets are already scarce, on-line ones are even more rare.

There are other Japanese datasets who deserve an honourable mention. These datasets are not comprised of handwritten characters, but they have been used in some recognition experiments. For instance, the Center of Excellence for Document Analysis and Recognition (CEDAR) from the State University of New York has a database of machine-printed Japanese character images. The images are extracted from documents, books, faxes, magazines, newspapers... etc. This dataset is available for purchase. Other notable datasets are the ones derived from KanjiVG. KanjiVG is an educational open-source project for teaching people *kanji*. Each *kanji* is stored in the form of an SVG file that gives the shape and direction of each of its strokes. Some machine learning projects have used this vector data in a simplified manner to train and test their ANN models.

3.3 Software for Neural Network implementation

In this section we will summarize the main features of the tools and libraries used for testing the neural network models in our own machine. Rather than being differentiated tools, this software works jointly in a sort of machine learning pipeline, so that the final user only needs to worry about his ANN structure.

3.3.1 Tensorflow

Tensorflow is an end-to-end open-source machine learning library that helps to develop and train ML models. It was developed initially by the Google Brain Team for internal use, but was soon used across diverse Google companies in both research and commercial applications. The first implementation, by the name of "Disbelief", dates back to 2011, but it was not until 2017 that we saw an initial release version of Tensorflow. Its popularity stems from being highly optimized for deep learning in ANNs. The name Tensorflow derives from the tensor, which is a vector or matrix of n-dimensions that can represent all types of data. We call "tensor's shape" to the dimensionality of this matrix or array. In machine learning, tensors are the operations that neural networks perform in multidimensional array data. For example, the feature vector will usually be the primary input to populate a tensor. Tensorflow was developed with C++ and Python and can run in multiple CPUs and GPUs. It is available on Windows, Linux, macOS, Android and iOS. For the user, Tensorflow provides stable Python, C, C++, Go, Java, Javascript and Swift APIs. Additionally, we also have Tensorboard available, which is a visualization tool to make easier to understand the execution process of Tensorflow.

3.3.2 Keras

Keras is a machine learning API written in Python, running on the top of Tensorflow 2.0. Created by François Chollet in 2015, it is conceived as an interface with a focus on enabling fast experimentation. Keras offers a series of high-level, intuitive instructions that serve as an abstraction of machine learning models. These implementations represent the "building blocks" of neural networks, such as layers, activation functions or optimizers. Both standard and convolutional neural networks are supported, along with some common utilities like dropout or pooling. This way, we can easily develop and try new models, hence increasing our productivity. Keras can run both on top of Tensorflow or Theano backend, which is yet another machine learning framework. Other backends are available, but Tensorflow is the officially recommended one. Starting with Tensorflow 2.0, Keras is highly integrated into the Tensorflow ecosystem, taking it as an inspiration for its high-level API. Keras let us build our models in two different ways: using the Sequential API or the Functional API. The Sequential API is the simplest type of model, allowing us to create a linear stock of layers step by step. We simply add the layers one by one from input to output. On the other hand, the Functional API allows us to create models with non-linear topology, shared layers and multiple inputs or outputs. For example, we can create a model with part or all the inputs directly connected with the output, so we can choose to train it following the intermediate layers or taking the bypass between input and output. Additionally, Keras includes some well-know datasets to debug models or create code examples, such as the previously mentioned MNIST.

3.3.3 Python and the Anaconda environment

For this thesis, we managed our versions of Python, Tensorflow and Keras with Anaconda. Anaconda is an open-source distribution of Python and R for data science and machine learning. Anaconda distribution includes over 250 basic packages by default, but thousands more can be download using its own package management system *conda*.

Unlike Python's default package manager pip, conda manages package dependencies in a way that is favourable for data science. For example, if we try to install a new package using pip, it will install all the dependencies last versions regardless of the requirements of the already installed packages. This could potentially break previously working code, or even worse, appears to work but with different results. In contrast, conda tries to create a compatible set of dependencies, warning you if not possible. Moreover, conda allows you to create and maintain different environments containing a specific Python version, files, packages and dependencies that will not interact with other environments. This is the fastest way to set up a machine learning configuration without installation hassles. Anaconda also includes a GUI called Anaconda Navigator where you can launch applications and manage packages without using the command line. Among these applications, we have the well-known Visual Studio Code or RStudio.

3.4 Software for Web application development

Like all web applications, we can divide ours into server-side and client-side. The server-side has been developed using the Python-based web framework Flask, so we can use the library OpenCV for preprocessing of the samples in the server before storing. The client-side has been developed mostly with plain HTML, CSS and Javascript, though with special attention to the Canvas API, as we will see in the next chapter.

3.4.1 Flask and the WSGI server

Flask is a micro web framework written in Python. The word "micro" here means that Flask aims to maintain the application core simple but extensible. By default, Flask does not include any utilities or libraries that other frameworks may include, such as a database layer or form validations. Instead, Flask supports different extensions to add these functionalities to your application. This said, Flask does have some conventions, such as storing all the templates and static files in homonymous subdirectories within the application source tree. Flask uses the **Jinja template engine** to embed variables with data from the server into our HTML. A Jinja template usually contains variables or expressions that get replaced with values when the template is rendered on page load, also using tags to control the template's logic. A minimal Flask application would be a Python file with a Flask import, followed by the declaration route to tell Flask what URL should trigger whatever function we may declare after that. Moreover, Flask follows the **WSGI standard**. The Web Server Gateway Interface (WSGI) is a specification that describes how a web server forwards requests to web applications written in Python. The web server is in charge of serving the client with the files it needs, such as HTML and CSS, using the HTTP protocol. When the web server receives a request from the client, it in turn asks the web application to generate the page by running the predefined Python code. Once generated, the page is forwarded to the web server which in turn sends it to the client. We could say the WSGI is the language used by the web server and the Python web application to communicate with each other. In our case, we use Flask for the Python web application, but there are other popular frameworks, such as Django. Currently, the most popular WSGI Web server is Gunicorn, but for our project we use Phusion Passenger, as we will see in the next chapter. Additionally, an Nginx server is usually used between the client and the web server to retrieve static files such as CSS or images.

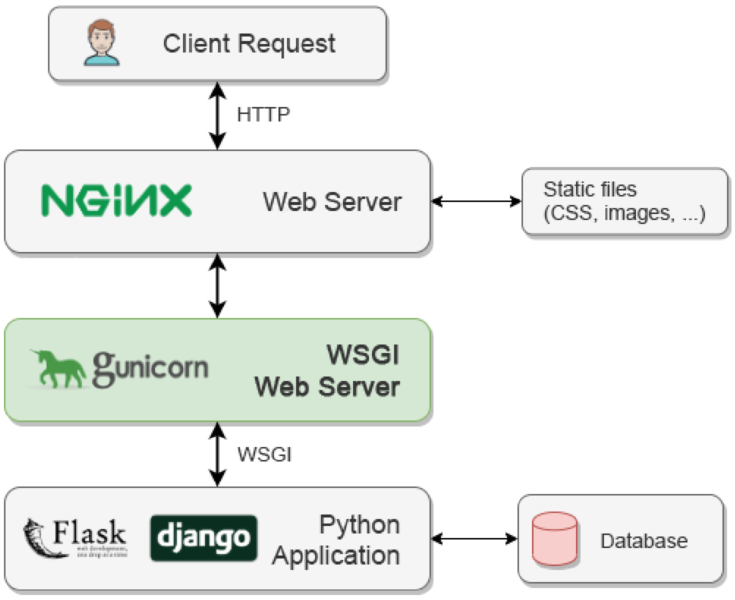


Figure 3.3: Representation of the Python WSGI server pipeline. Original image from Nacho Alonso (2019), translated by this thesis' author.

3.4.2 PyMySQL

Previously we have mentioned that Flask relies on extensions to include basic utilities, such as the database layer. For this purpose, we used the PyMySQL package. PyMySQL is a pure-Python MySQL client library, or in other words, an interface for connecting to a MySQL database server from Python. It implements the Python Database API, which is the Python standard for database interfaces and can support a wide range of database servers. The goal of PyMySQL is to be a drop-in replacement for MySQLdb, which is another database connector for Python. There are a few reasons to use PyMySQL over MySQLdb. First, while MySQLdb is a C extension module, PyMySQL is written in pure Python. For this reason, end-users of MySQLdb need to wait for new binaries to be compiled for each new release of Python. In contrast, PyMySQL does not have any dependency and therefore is easier to run in some systems as you do not need headers or compiled C components. Another important advantage is that PyMySQL supports Python 3. Although some forks of MySQLdb do support Python 3, until today no official support is available. For some users it is also important that PyMySQL works with PyPy, which is a faster Python compiler. All in all, the main reasons to use PyMySQL have to do with compatibility, even being slower due to its pure Python implementation.

3.4.3 OpenCV

OpenCV is a real-time computer vision software library. The Open Source Computer Vision Library (OpenCV) was originally developed by Intel and released in 2000, being nowadays maintained by a non-profit foundation. OpenCV is written in C++, though it started as a C library. New developments and functions are now written in C++, with bindings for Python, Java and Matlab. There are also several wrappers for other languages. Due to its open-source BSD license, OpenCV is used extensively in all kind of companies, research groups and government bodies. Moreover, it supports GPU's hardware acceleration through the use of CUDA and OpenCL for parallel computations. Some of the main modules of OpenCV are: image processing, video analysis, camera calibration, 2D features detector, object detection, ...etc. In our project we make use of the first of these modules to perform resize operations and content checks on the received character samples from our crowd-sourcing web application. Thanks to the OpenCV Python binding we can make use of all of these functions in our Flask app, though in a slower manner than the native C++ implementation. OpenCV has also evolved to include statistical machine learning utilities such as ANNs implementation, the Naive Bayer classifier or k-nearest neighbour.

4 Experiments and Implementation

This chapter describes the results of our experiments and the development of our crowd-sourcing application. It is made up of 4 different sections: Section 4.1 gives a brief introduction about the structure of this chapter, Section 4.2 presents the results of employing multilayer perceptron models over pre-existing Japanese character datasets, likewise, Section 4.3 shows the results of using Convolutional Neural Networks for the same purpose, Section 4.4 summarizes the development and design of our crowd-sourcing web application, Section 4.5 describes the process of collecting and creating our own Japanese character dataset and Section 4.6 compares the results of the previous datasets with our own one using the same models.

4.1 Introduction

In this chapter, we are going to describe the experiments and implementation following the natural flow of our project. First, we have to define multilayer perceptron and CNN models that fit the previous Japanese datasets, based on already studied models and our knowledge about ANNs. At the same time, we can briefly see how these are implemented in Keras. After this, we will describe the process of planning, creating and setting up our own web application for collecting new character data, and, once we count with a significant amount, turn it into a suitable dataset to feed the previously proposed models. Along with the character itself, the application asks the user some relevant questions, with the purpose of creating subsets of characters depending on the answers. Finally, we can compare the results between both datasets and figure out the basis of their differences.

4.2 Multilayer perceptron model for character recognition

In this section, we will test the performance of multilayer perceptron models over the datasets listed in the last chapter. Multilayer perceptron are basic ANN models, so they are not expected to perform better than more advance models like CNN. However, multilayer perceptron models often show better results than expected in multiples situations, and it is very interesting to see to what extent we can improve their performance. For this section and the next we will use three datasets as training data: **KMNIST**, *hiragana* characters from **ETL-8B2** and *kanji*, again from ETL-8B2.

Kuzushiji MNIST is a drop-in replacement of the MNIST digits dataset, and therefore it works without any special treatment. We only have to download the .npz files, which are Numpy's compressed array format, and proceed as with any other dataset. These files are openly provided by the ROIS-CODH in their *GitHub* repository. Usually when training ML models, datasets are divided in two subgroups: *train* and *test*. *Train* is the actual data with which we train our model, while *test* is a part of the dataset we hold apart to test the accuracy of the model with previously unseen data. The Kuzushiji MNIST dataset already provides these two subsets in separated compressed files, so is even faster to get down to business.

On the other hand, the **ETL** requires a considerable amount of work to be usable in a real model. The ETL is an unusually old dataset. Back in its days, it was distributed by sending recorded media like magnetic tapes and CD-Rs with the postal mail. This is why each record in it is organized completely linearly, byte by byte. To make it a suitable Keras input, we have had to create some specific functions in Python to convert it from its original format to the usual *train* and *test* data. First, we unpack the original files by providing the exact byte length of each record. Following the record structure that we can see in Table 4.1, we iterate through each record, extracting the images and labels from each sample and reshaping the images into its 64×64px size. Once we have each of them, we proceed to *shuffle* the data. This function from the *sklearn* package does random permutations of the data collection in a consistent way. Lastly, we use the *train_test_split* function to separate a *test* subset from the rest of the training set. The size of this *test* set is specified as a percentage parameter in the function, in our case 20%. After following these steps, the ETL dataset is now in the same state as Kuzushiji MNIST.

bytes	type	contents
1-2	Integer	Serial Sheet Number
3-4	Binary	JIS Kanji Code (JIS X 0208)
5-8	ASCII	JIS Typical Reading (ex. ‘AI.M’)
9-512	Packed	Binary image of 64 x 63 = 4032 pixels

Table 4.1: The ETL-8B2 dataset structure.

Before coding the models, we still have to do a couple of steps. First, we are going to normalize the data input from 0-255 to 0-1, for the reasons we explained in Chapter 2. Secondly, we convert our labels to one-hot encoding format. Most machine learning algorithms require numerical input and output variables. Usually our labels are categorical data, such as "dog" or ours "あ". One-hot encoding is used to convert categorical data into integer data, so we can safely input our data into the ML models.

Before proposing some models, we may want to know how we can judge their performance objectively. In machine learning we usually use the **loss** and **accuracy** to evaluate the performance of our model after each epoch. Both are calculated for the training and validation sets. The loss is the summation of the errors made for each example. The accuracy is the percentage of correct classifications made by the model after the learning is complete. These parameters are concrete and a good indicator, but sometimes they fail to offer a more global perspective of the model behaviour. To achieve this we can use **learning curves**. These plots allow us to observe where the training is progressing towards and diagnose its behaviour. Observing the curves we can deduce what is happening inside the network, something that would be otherwise impossible to analyse parameter by parameter. In the Figures 4.1, we can see the reasons that may be causing each different trajectory of the curve.

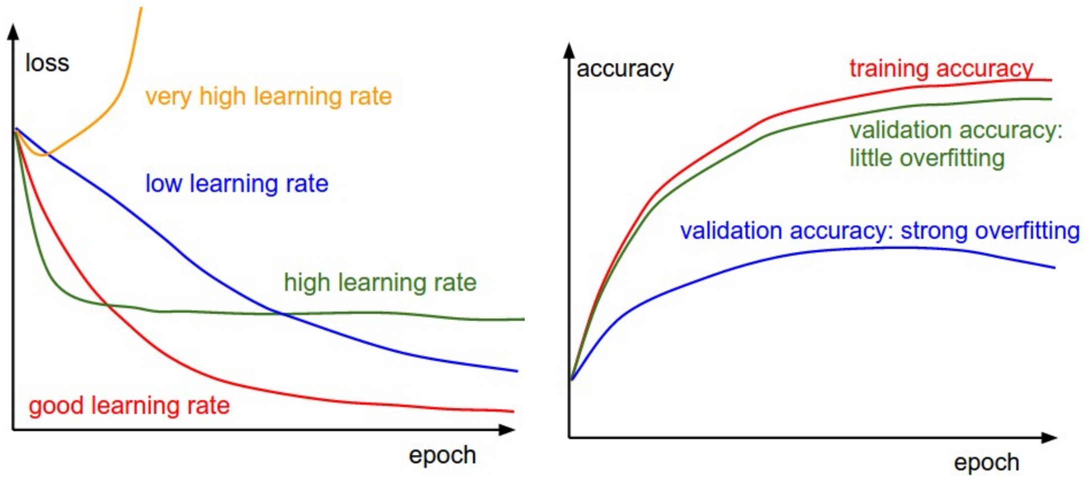


Figure 4.1: Left: Loss curve portrayal **Right**: Accuracy curve portrayal with difference between training and validation sets. Image retrieved from George V Jose (2019).

In general terms, the gap between training and validation accuracy is a clear sign of overfitting. The larger the gap, the higher the overfitting (George V Jose, 2019). Of course, this is relative to the scale used to plot the y-axis in the graph. We will use the same type of accuracy plot to diagnose the behaviour of our models.

This said, the most important factor to take into account keeps being **the data itself**. Without proper training data, the model is ill-fated from the beginning. Many of the problems that arise during training actually have to do with the data itself, primarily due to an imbalanced dataset, but sometimes simply because there is not enough data to support the problem statement.

Lastly, we also have calculated a baseline error based on the accuracy returned by the evaluation of our *test* data after the network learning is complete. It is included at the top of each accuracy curve.

First, let's take a look at the **KMNIST** results. KMNIST has only 10 classes, but with a large number of samples. Each class have 6000 training samples and 1000 test samples. In theory, having such a large number of samples for only 10 classes makes easier for a model to generalize the data, even with a simple multilayer perceptron ANN. First, we started with a network with no hidden layers, only with the input layer of 784 neurons (since 28×28 image = 784 pixels), and the output layer. As you can see in Figure 4.2(a), after 10 epochs we got a pretty decent baseline error of 8.93% with a training accuracy of 0.99 and validation accuracy of 0.92 at the last epoch. We tried then to add a hidden layer. Following the rule explained in Chapter 2, a good reference size for hidden layers is 2/3 of the input layer, in our case 522 neurons. This model is featured in Figure 4.2(b), trained during 50 epochs and obtaining a better baseline error of 7.38%. However, we see how the values stay in a range around the same as the previous model, suggesting that we have reached a limit.

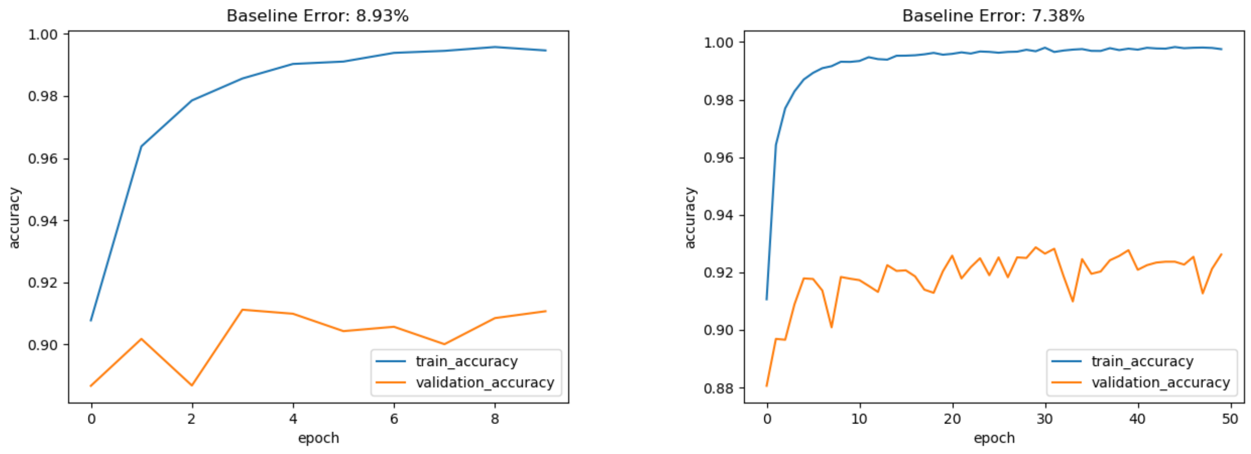


Figure 4.2: Left: KMNIST with no hidden layers during 10 epochs **Right**: KMNIST with 522-neuron hidden layer during 50 epochs.

Next is the turn of the *hiragana* characters from **ETL-8B2**. This dataset is comprised of the 71 hiragana classes, including the *dakuten* and *handakuten* diacritics variances, with 160 samples each. In this case, we have more classes and fewer samples, so logically we can expect the model to perform worse. A first test without any hidden layers during 10 epochs returns a considerable 21.96% baseline error. Following the same logic as the last model, we now try with a hidden layer of 2730 neurons (2/3 of a 64×64 layer). This returns at the end a slightly better baseline error of 14.71%. We can see the evolution of this model from 10 to 50 epochs in Figures 4.3(a) and 4.3(b), respectively. These show clearly how the validation accuracy gets stuck around 0.8, while the training accuracy almost gets to 1.0. This is a clear sign of moderate over-fitting. In other words, at some point the model has started memorizing the training set, failing to generalize the data, so the validation accuracy can not further improve.

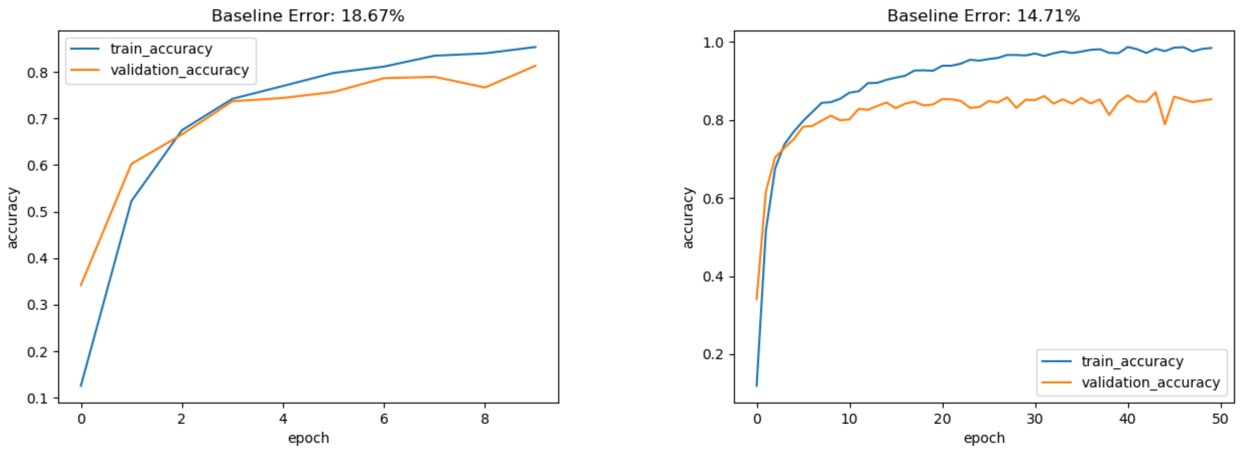


Figure 4.3: Left: ETL-8B2 *Hiragana* with 2730-neuron hidden layer during 10 epochs **Right**: ETL-8B2 *Hiragana* with 2730-neuron hidden layer during 50 epochs.

Lastly, we have the *kanji* characters from **ETL-8B2**. This dataset contains thousands of classes, but because of time constraints we have limited it to 100 classes, which is a similar number to the rest of datasets and the one we are going to create. The number of samples per class remains 160. In this case, after some trial and error guesses, the model with the best performance turns to be the one with a single 2730 neuron hidden layer, having a 26.05% baseline error. We can see it during 10 epoch in Figure 4.4.

These multilayer perceptron models are interesting in the sense of learning to interpret the parameters that define the behaviour of neural networks, but performance wise we could expect something much better. For that reason we are going now to delve into CNN models.

4.3 CNN model applied to Japanese datasets

As it was previously explained in Chapter 2, Convolutional Neural Networks are very effective in the field of image classification. We will now test the accuracy of this statement. In this section we will use two models of CNNs for our experiments. The first one is an example CNN intended for the MNIST dataset that is included in the

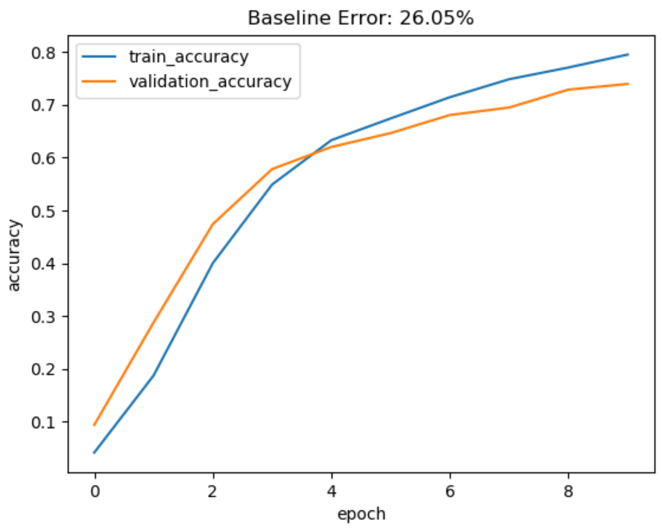


Figure 4.4: ETL-8B2 *Kanji* with 2730-neuron hidden layer during 10 epochs.

Keras documentation (François Chollet, 2015). It is comprised of two convolutional layers (64 and 32 neurons) with its respective MaxPooling layers and a Flatten layer. This is a good starter to see how the datasets perform inside CNN models. The second one has been created by ourselves, based on the research of Tsai (2016) and in turn in the VGGNet model. Using the modern Keras-Tensorflow syntax, we have tweaked the basic VGG structure to follow the convolutional layer ordering that gave the best results in the experiments carried out by Tsai (2016). This CNN structure code is featured in Listing 4.1.

Listing 4.1: New CNN code in Keras (Python)

```python
def CNN_deep_model():
    model = keras.Sequential()

    model.add(keras.Input(shape=(64, 64, 1)))
    model.add(layers.Conv2D(32, (3, 3), activation='relu'))
    model.add(layers.MaxPooling2D(pool_size=(2, 2)))
    model.add(layers.Dropout(0.25))

    model.add(layers.Conv2D(64, (3, 3), activation='relu'))
    model.add(layers.MaxPooling2D(pool_size=(2, 2)))
    model.add(layers.Dropout(0.25))

    model.add(layers.Conv2D(128, (3, 3), activation='relu'))
    model.add(layers.MaxPooling2D(pool_size=(2, 2)))
    model.add(layers.Dropout(0.25))

    model.add(layers.Conv2D(256, (3, 3), activation='relu'))
    model.add(layers.MaxPooling2D(pool_size=(2, 2)))
    model.add(layers.Dropout(0.25))

    model.add(layers.Flatten())
model.add(layers.Dense(4096))
model.add(layers.Activation('relu'))
model.add(layers.Dropout(0.5))
model.add(layers.Dense(num_classes, activation='softmax'))
# Compile model
model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
return model
```

Something that has not been mentioned previously and that we can be found in the previous code are the optimizers. The role of **optimizers** inside the machine learning pipeline is to update the weight parameters to minimize the cost function. The cost function tells the optimizer if it is moving in the right direction to reach the global minimum. The one we are using, "Adam" (Adaptive Moment Estimation), is one of the most popular gradient descent optimization algorithms and is very computationally and memory efficient.

We will now test both of our CNN models with **KMNIST** dataset. In Figure 4.5(a) we can see the results of the Keras CNN with a batch size of 16 during 20 epochs. We obtain a good 5.30% baseline error, with 0.9785 of training accuracy and 0.9460 of validation accuracy at the last epoch. In this case, using our own CNN returns similar or slightly lower results than the Keras model. This last one can be seen in Figure 4.5(b), with a baseline error of 6.37% during 20 epochs. The values for this model at the last epoch were 0.9535 of training accuracy and 0.9363 of validation accuracy.

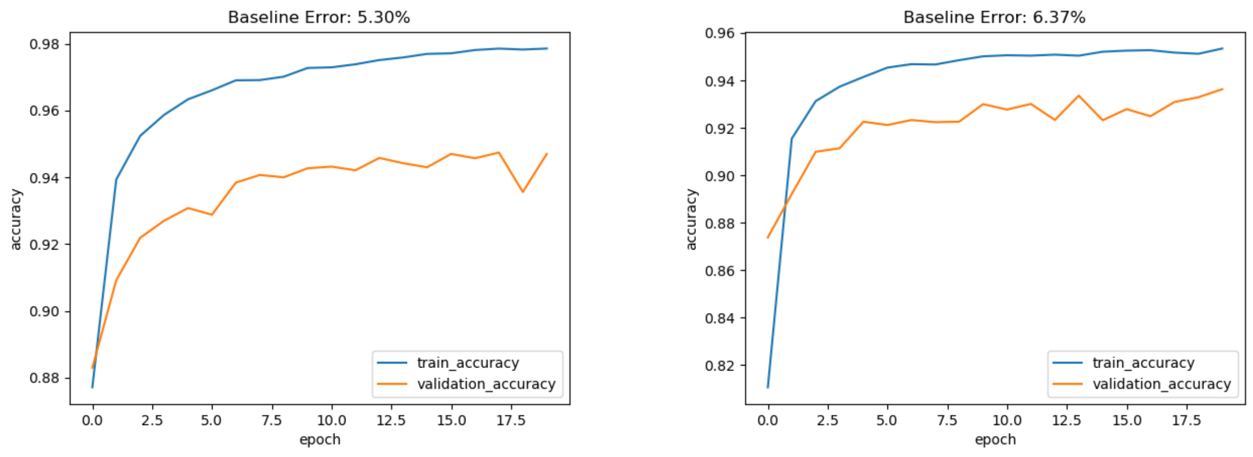


Figure 4.5: Left: KMNIST with Keras CNN, batch of 16 and 20 epochs **Right**: KMNIST with New CNN, batch of 16 and 20 epochs

In the next experiments using **ETL-8B2**, we will see how optimal curves are supposed to look. First, is the turn of *hiragana*. The Keras CNN experiment of Figure 4.6(a) still has a curve similar to the rest, trained with a batch size of 16 during 20 epochs. It returns a 8.29% baseline accuracy, with 0.9914 of training accuracy and

0.9171 of validation accuracy at the last epoch. On the other hand, the experiment using our CNN in Figure 4.6(b) shows how both the training and validation curves converge, which is a sign of having a good fitting model. The model ends with a 5.58% baseline error, with a training accuracy of 0.9886 and a validation accuracy of 0.9421.

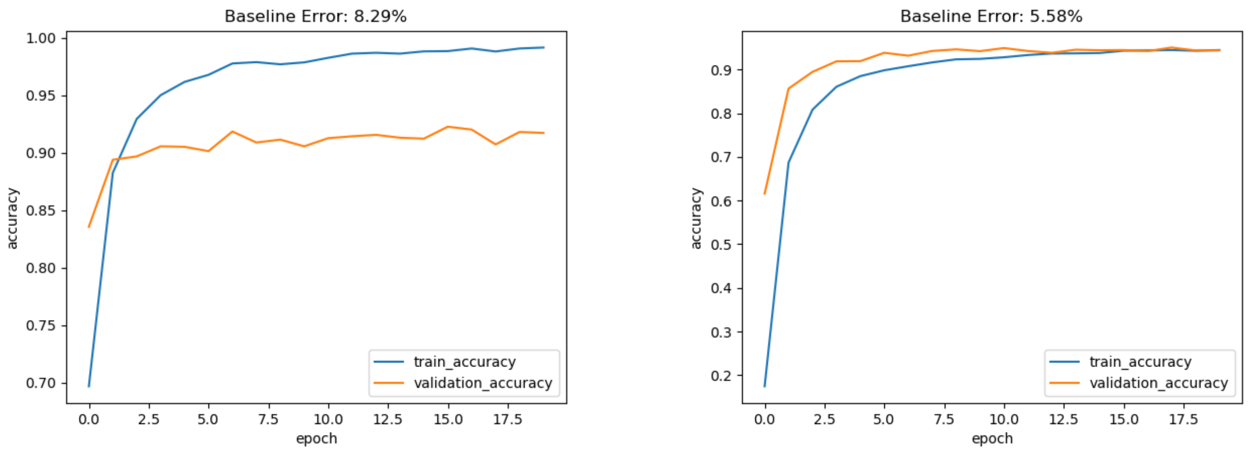


Figure 4.6: Left: ETL-8B2 *Hiragana* with Keras CNN, batch of 16 and 20 epochs **Right**: ETL-8B2 *Hiragana* with New CNN, batch of 16 and 20 epochs

Lastly, we will cover the experiments with **ETL-8B2** *kanji*. In theory, these experiments should show superior performance compared to their *hiragana* counterpart despite having the same number of examples per class, since in general *kanji* have more differential characteristics between them due to their complexity. In this case, the Keras CNN obtains a really good result, with a 1.88% baseline error, a training accuracy of 0.9989 and a validation accuracy of 0.9812, as shown in Figure 4.7(a). Finally, our own CNN achieves an excellent result with a 0.78% baseline error. As shown in Figure 4.7(b), the last epoch gives us a training accuracy of 0.9796 and a validation accuracy of 0.9922.

With this, we conclude our experiments on publicly available Japanese datasets. A human equivalent recognition rate of 96.7% for *kanji* has been reported in Yin et al. (2013). Some of our models therefore outperform the human recognition rate.

4.4 Development of Japanese Crowd-sourcing web application

This section details the process of development of our crowd-sourcing web application from its initial definition to its final validation. The process starts with the description of the functional and non-functional requirements of the web application, so we can

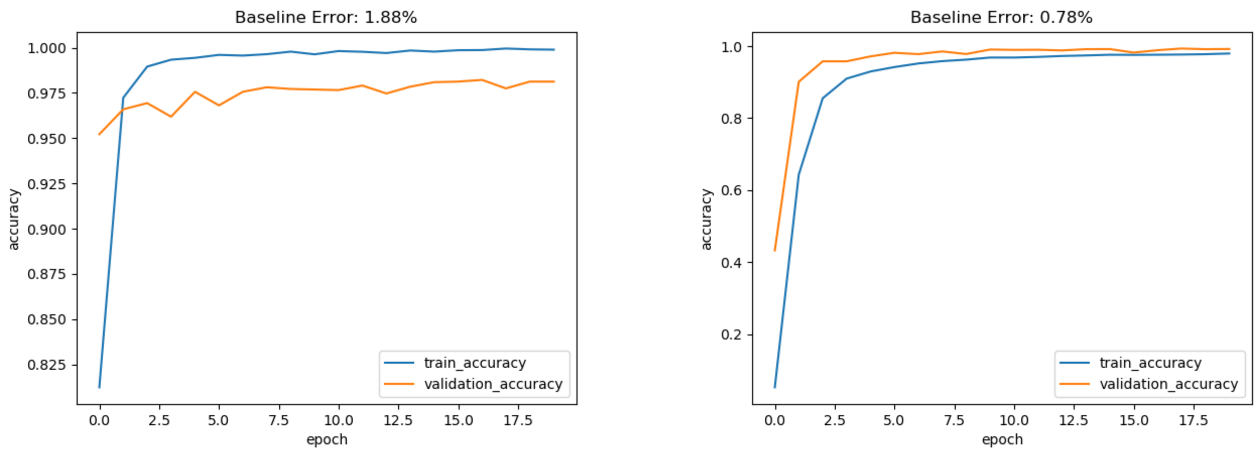


Figure 4.7: Left: ETL-8B2 *Kanji* with Keras CNN, batch of 16 and 20 epochs **Right**: ETL-8B2 *Kanji* with New CNN, batch of 16 and 20 epochs

clearly set the guidelines of development for the project. Moving on, the design section not only includes definitions for the style, but also architectural and back-end design. After that, the characteristics described in the last sections are put into practice during the implementation and deployment process. Finally, we carry out a series of tests and validations with different users to assure the proper working of the application.

4.4.1 Requirements and specification

The creation of our own crowd-sourcing web application is one of the main parts of this project. Above all, the application aims to be straightforward to use, yet completely functional. We could not achieve this without proper planning beforehand. We will establish the restrictions, requirements and user profiles for this app through the standard of IEEE 830 (IEEE, 2008). This standard establishes a series of good practices for the specification of software requirements. Since the organization and format of the standard are not fixed, we can adapt it to describe the parts we are interested in.

First, we will describe the general features of the users that our application is target to. These **user profiles** are organized in Table 4.2, including for each one a description of themselves, their skills and their role inside the application.

User kind	Features	
Administrator/Researcher	Profile	Owner of the system. Usually a researcher or professional interested in Japanese OCR.
	Skills	High technical and linguistic educational level. Should have reasonable knowledge in web technologies and distributed systems, as well as enough comprehension of the Japanese language inner working.
	Role	The administrator is in charge of managing the proper working of the whole system, solving any eventual issue. Fulfilling the role of researcher, the administrator must also define the form of the Japanese dataset the app will gather.
User	Profile	Any person who uses the application. Usually interested in Japanese OCR or willing to collaborate.
	Skills	No technical education is required. The user should have at least basic knowledge about the Japanese writing system to correctly use the application. Either English or Japanese understanding is required to use the application.
	Role	The user only task is to collaborate to the dataset creation through the client-side interface, where they can input one character at a time and send it after answering some basic questions. They can repeat this process as many times as they want.

Table 4.2: Description of the different user profiles for the web application.

Secondly, we have to define the **restrictions** we will be facing during the development of the application. We explain each of them in Table 4.3, along with a reference code that will be used in later sections to cite each restriction if necessary. A description and type for each restriction are also included.

Reference code	Type	Description
RES-HW-01	Hardware	We have **server memory limitations**. The application will be deployed in a shared hosting, where one physical server hosts multiple websites. Therefore, we only get a percentage of that individual server. Specifically, we have 20GB of SSD disk space and around 1GB of RAM.
RES-SW-01	Software	Our back-end must be **capable of running our OpenCV scripts** for character images preprocessing. This implies that our server application must be able of executing code in an OpenCV supported language, in this case Python.
RES-SH-01	Schedule	This is the most important restriction. Our web application must reach a state of **minimum viable product well ahead of schedule**, before this thesis' deadline, so it can collect enough character samples to create a dataset that can be used for comparative testing.

Table 4.3: Detail of the restrictions we have to take into account during the development.

We will now define the different **requirements** we will have to fulfil during development to achieve the objectives of our application. We can divide them into two types: functional and non-functional requirements. These will be presented in two different tables. Similarly to the restrictions, each requirement has a reference code for citing later if required. Each reference code is made up of two parts that refer to the content of the requirement: whether is a functional or non-functional one (FR or NFR), and if it concerns the user, the administrator or the system itself (USR, ADMIN or SYS).

The **functional requirements** are the ones which make reference to the system functionalities. They define the basic system components, generally divided into system input, processing, and output. These are listed in Table 4.4.

Reference code	Profile	Description
FR-ADMIN-01	Administrator	The administrator/researcher defines the form of the Japanese dataset that will be collected by the web application. This includes the character classes and the number of samples per class and category.
FR-USR-01	User	The user draws the requested character inside the HTML Canvas in the web application interface.
FR-USR-02	User	The user has to answer three questions related to his birthplace, language ability and method used to input the character before sending one.
FR-USR-03	User	Once the user sends the character sample, a confirmation modal will pop up acknowledging his contribution and asking to press the return button to input a different character class.
FR-USR-04	User	The user can choose in which language the application interface is displayed, either English or Japanese.

Table 4.4: Table of functional requirements for the web application.

Finally, the **non-functional requirements** are the ones most related to the system itself. They are regarded as quality requirements to define the behaviour of the system as planned, establishing some limits if necessary. These are listed in Table 4.5.

Reference code	Profile	Description
NRF-SYS-01	System	Once the character is received by the server app, preprocessing is performed over each sample to adapt it to our desired format.
NRF-SYS-02	System	Each character sample and its questions answers are stored in the file system and the database, respectively.
NRF-SYS-03	System	The system will always request a character based on the number of character records in the database, prioritising the classes with fewer samples and maintaining the dataset balanced.
NRF-SYS-04	System	The system must reject any post from the client with a blank canvas input.
NRF-SYS-05	System	The application interface will adapt to any screen size or device. This includes support for character input through mouse, touchscreen or stylus pen.
NRF-SYS-06	System	The application must be 24/7 available through a fixed web domain. Users from all around the world must be able to send samples without restrictions apart from an internet connection.
NRF-SYS-07	System	All the application and the material collected must be under an appropriate open-source license, so both the application and the datasets created under it will be public and free for everyone, forever.
NRF-SYS-08	System	The application code will be highly commented and under a version code system to easily allow contributing developers to add functionalities or fork the project.

Table 4.5: Table of non-functional requirements for the web application.

4.4.2 Design guide

The design section is one of the most relevant for the development. Here we are going to solve the problems stated in the requirements section. These solutions are the ones that will be implemented in the next section to successfully develop a web application that covers our expectations. Not all of these requirements can be addressed using a single technology, in fact, is more common to use multiple technologies to implement each of them. For better clarification, we will divide this section into various subsections that represent each part of the design process in a more concrete way.

4.4.2.1 Conceptual design

Here we define the building blocks of our application. This structure can be useful in the future to organize the actual implementation of the project. All the application logic is centralized in the server, leaving the client-side interface only as the input for data. A representation of this structure is shown in Figure 4.8.

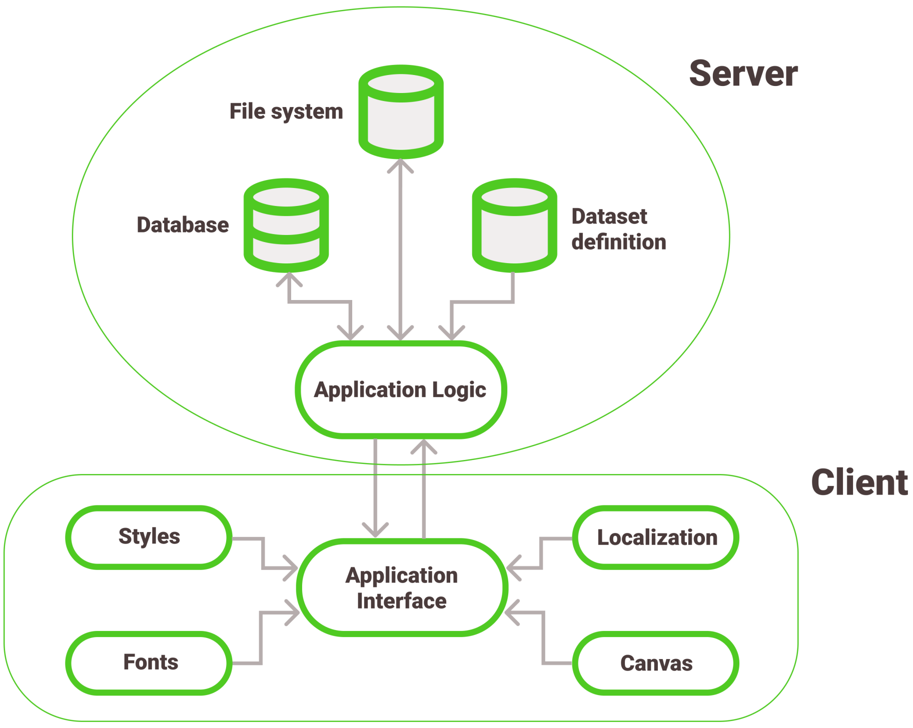


Figure 4.8: Schema representing the application conceptual design. Author's own creation.

Each of these building blocks can be defined as follows:

- The **Application Interface** is where all the information is displayed to the user from the web browser, including the requested character and the three questions for the user.
- Our chosen **Styles and Fonts** are loaded into the application interface to compose the application visual aspect.
- The **Localization** are the versions of the static texts in the application in English and Japanese.
- The **Canvas** is the part of the application interface from which the user inputs the character.
- The **Application Logic** controls the storage of data sent by the client, the preprocessing of the character images and the sending of data to the client for display, all from the server.
- The **Database and File system** are the so-called “persistent” parts of the application, where we store all the data coming from the client for later examination.
- The **Dataset definition** is a static section of the server storage where the new dataset content is defined.

4.4.2.2 Persistence design

We call “persistence” to the part or parts of the application that are used to store data. In our case, this data is received from the client each time the application collects a new sample. The persistence of our application can be divided into two different sections: the database and the file system.

For our application, it was decided to store the character images inside the server file system, instead of inside the database itself. This is because file systems are already optimized to store files, including web server file systems. On the other hand, databases require additional code to extract and stream images and the latency may be slower than direct file access. We therefore decided to store them in the server file system, at the same time reducing the load on the database server.

Inside the database two tables were created: *Dataset* and *Classes*. The *Dataset* table holds every record made to the dataset, with a character name column, an image URL to reference the path of each image in the server file system and the answers of each user to the three questions (*NRF-SYS-02*). The *Classes* table holds the number of records that the application has collected for each character up to that point. We keep this register in order to avoid having to recount the number of records of each character from the *Dataset* table each time the application decides which character to request (*NRF-SYS-03*). Our relational database server uses MariaDB and is managed with PyMySQL, as explained in the last chapter.

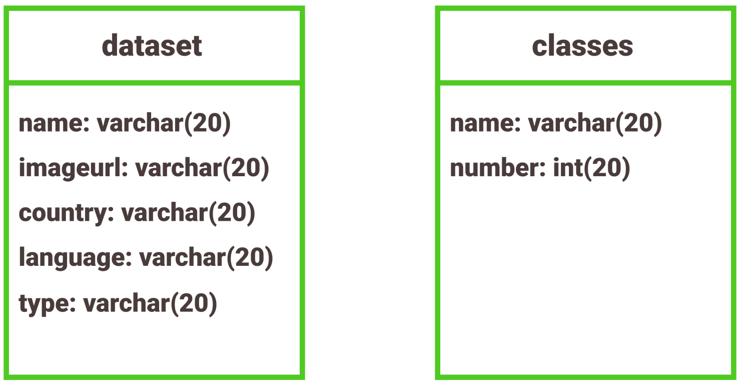


Figure 4.9: Database tables structure, along with each column's datatype. Author's own creation.

4.4.2.3 User experience

Usability and accessibility are very important factors to take into account when considering our application structure, but so is the user experience. We could define the user experience as the "feeling" an application transmits while using it. One good way of representing the user experience is through a *User Journey Map*, like the one in Figure 4.10.

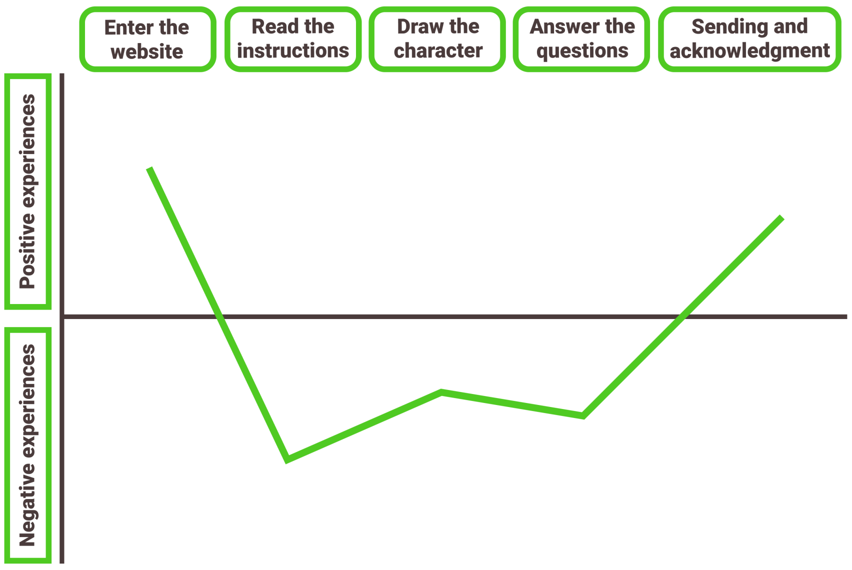


Figure 4.10: User Journey Map of our web application. Author's own creation.

Studying this type of user experience patterns is truly relevant for our application because, in principle, the user does not get any trade back from using our web application, apart from an appreciation message. We can see in our *User Journey Map* that a user using our web application is most of the time dealing with “negative experiences”, that is doing actively things without feedback. Therefore, our mission is to reduce the time these tasks take from the user as much as possible. At the same time, we want the user to send as many character samples as possible, so we must also encourage him to repeat the process. We should then design an interface structure that fulfils all these requirements.

4.4.2.4 Interface design

For our application, we have opted for a simple and minimal design that follows our objective of reducing the time for sending samples as much as possible. This type of design not only improve the user’s navigation flow, but also makes our app more usable and stylized.

In order to design and visualize the interface structure before the implementation phase, we created a number of *wireframes* sketches, which are visual guides that represent the skeletal framework of a website, app or program. First, we will take a look at the general proposed structure:

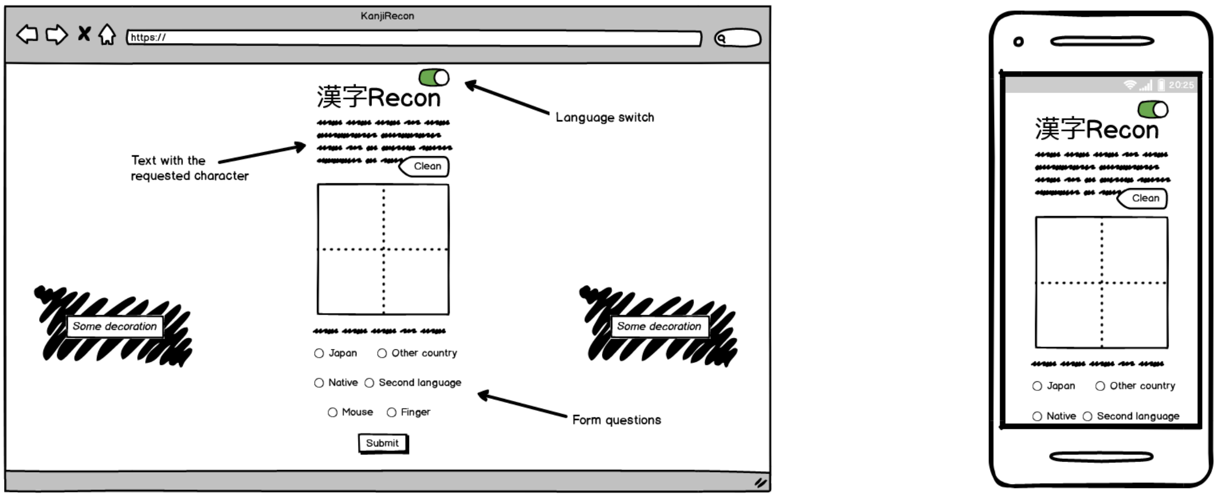


Figure 4.11: *Wireframe* of the web application interface for desktop and mobile devices. Some annotations were made over the image with pointing arrows for further clarification. Author’s own creation using Balsamiq tool.

The first thing that someone usually notices from any web application is the title: **KanjiRecon**. We decided to name our tool like this after the words *Kanji* and Recognition, which is the purpose behind collecting character samples publicly. From the beginning we thought about properly designing an accessible application that adapts to any device screen (*NRF-SYS-05*), to the point that the *wireframe* already included a mobile version of the website at that time. As we can see, all the application content is organized in a y-axis column, making it easier to adapt to mobile screens. The subsequent space left blank in the desktop version margins was planned to be filled with symmetrical background decorations. The **HTML Canvas** is placed at the centre of the interface, as is the most important part of the application, along with a "clean" button to erase the canvas content. An intuitive addition to this writing interface could be the typical "undo" and "redo" buttons, but, as stated in Chapter 2, Japanese characters must be written in one go, so this option is discarded. In Chapter 2 we also mentioned the inclusion of a *genkō yōshi*-like grid in the canvas. This way we may save many operations over the images to centre the characters when creating our dataset. Apart from this, adding a reference to write the character is very convenient for the user, since it increases the usability of the page. Following the Japanese topic, a **language switch** is placed over the tool logo, to change between English and Japanese. Under said logo we encounter the **instructions**. We tried to maintain them as minimal as possible, explaining briefly why we are interested in retrieving character samples over the internet a which character is being requested for that session. It is worth saying that this type of design is completely opposite to traditional Japanese web design, which often included as much information as possible in a given space, generally growing into visually overloaded websites. This tendency is nowadays changing on the Japanese internet, though many old websites still keep this style. As one target public of this tool are Japanese people, it will be interesting to receive feedback from them about this topic. Lastly, we have the **three form questions** just over the submit button (*FR-USR-02*). It is interesting to note that originally each question had only two possible answers, as opposed to the final questions with three answers. We decided on the content of these questions based on what factors might most influence the user's habits and writing style. The final questions are: **Where are you from?** (Japan/Other), **What is your level of Japanese language?** (Native/Second language/Learner) and **What have you used to write the character?** (Mouse/Finger/Stylus pen).

Wireframes can also be used for other things apart from designing the application interface. User experience designers use *wireframes* to show navigation paths between pages. In Figure 4.12, we can see a *wireframe* that represents our application flow:

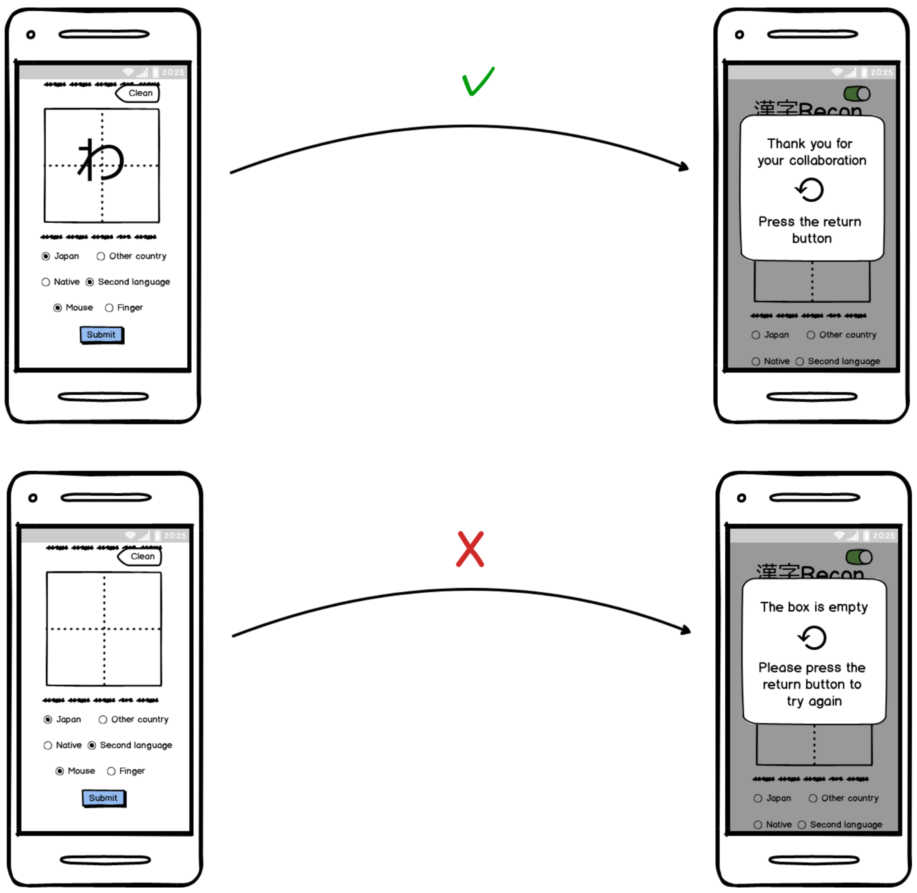


Figure 4.12: *Wireframe* of the web application navigation flow. A mobile version is used to save space. Author's own creation using Balsamiq tool.

This image represents what happens when the user presses the submit button in different situations. If the inputs are all filled and the canvas has any content, an acknowledgement modal is triggered with a "Thank you for your collaboration" message. As we want to encourage the user to send more samples, the application does not redirect to any other page, instead, it pops up the modal in the main one, with a "return" button to close it. This way we transmit the user the feeling of being inside a cyclical process, instead of redirecting to an endpoint page. On the other hand, if the canvas is left blank, the server will reject the post and issue a message asking the user to try again (*NRF-SYS-04*).

4.4.2.5 Colours, logo and typography

The **colour palette** we used for our web application have two main shades. *Light gray* is the main colour, while *Lime green* is the accent colour. With these "light colours" we want to give a floating feel to the interface, along with its plain white background.

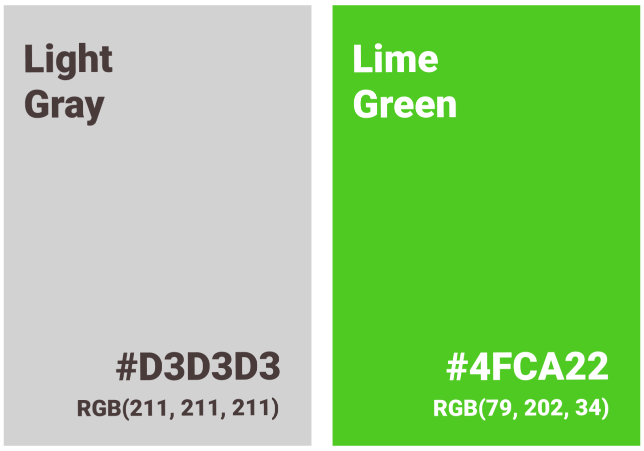


Figure 4.13: Colour palette of the KanjiRecon tool. Author's own creation.

The **logo** of our KanjiRecon tool is featured in the header, being most probably the first thing any user notice from the app. The design of our logo is inspired by the fact of KanjiRecon being a bilingual app, featuring texts in English and Japanese. For that reason, half of the logo is written in Japanese (meaning *kanji*), and the other half being an abbreviation of the word "Recognition" in English. This strengthens the international scope of the application, appealing the public from both Japan and the rest of the world. The logo also includes the two colour shades of our palette, so it stays in tune with the rest of the interface.

漢字Recon

Figure 4.14: The KanjiRecon logo. Author's own creation.

This said, our application is after all an open-source tool and therefore our logo can not be as relevant as the ones from commercial targeted applications. Nevertheless, having a distinctive symbol is always beneficial for the diffusion of any medium. In this manner, the public has an easy way to identify a certain element, and perhaps, share it with their peers, which is exactly what we want.

The next thing to cover is the **typography**. We can play with the fact of having to represent two languages by using a different font for each of them. This way our application can transmit a different sensation while being in English or Japanese. For the English texts, we use the *Nunito* font. *Nunito* is a sans serif rounded typeface created by Vernon Adams. With its rounded terminals, a “soft” feeling is transmitted, which is in line with the light style of the application. For the Japanese texts, we use the *Noto Serif JP* font. *Noto* is a font family created by Google that aims to support all languages in the world. We use the serif Japanese family to give the app a more “calligraphic” sensation while it is in Japanese, since handwriting is one of the main actions performed in the application. Both fonts are used under the Open Font License and retrieved from the *Google Fonts* service.

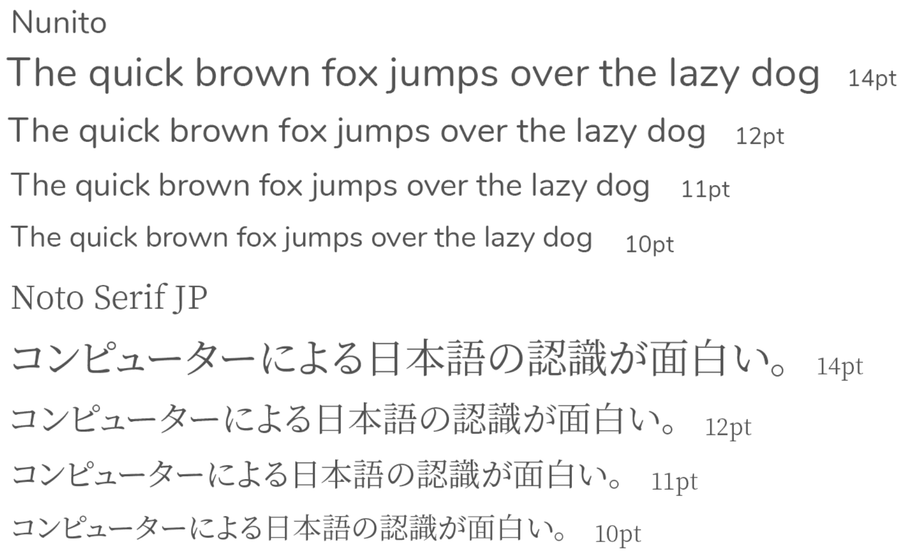

Nunito
The quick brown fox jumps over the lazy dog 14pt
The quick brown fox jumps over the lazy dog 12pt
The quick brown fox jumps over the lazy dog 11pt
The quick brown fox jumps over the lazy dog 10pt
Noto Serif JP
コンピューターによる日本語の認識が面白い。 14pt
コンピューターによる日本語の認識が面白い。 12pt
コンピューターによる日本語の認識が面白い。 11pt
コンピューターによる日本語の認識が面白い。 10pt

Figure 4.15: *Nunito* and *Noto Serif JP* fonts. The example sentence in Japanese includes parts in *katakana*, *hiragana* and *kanji*. Author’s own creation.

Moreover, we have to assure any font in the interface to be bigger than 10px to avoid the graphical density problem in Japanese described in Chapter 2. This way we guarantee the legibility of the interface and improve the app usability.

4.4.3 Implementation and deployment

In this section, the process of development is detailed step by step, carrying out the solutions that were outlined in the design section. As stated in the Schedule section from the Introduction, the app was developed from late March to the end of April. This is due to the restriction of having to leave a time frame before this thesis’ deadline for collecting enough character samples (*RES-SH-01*). We will now explain each step of the process in the same order that they were developed. After that, the same will be done with the deployment process. Although a description of each part of the development will be presented, we will pay special attention to the problems we encountered, describing their origin and the solution we came up with.

The development has been carried out in my personal desktop computer, using the *Windows 10* operating system. The IDE used was the popular *Visual Studio Code*, under an *Anaconda Environment*, as explained in Chapter 3. The version control software used was *GitHub*, along with the GUI client *GitKraken*. A repository was created to hold all the material for this thesis, including the whole KanjiRecon code (*NRF-SYS-08*).

The first step was to set up the Anaconda Environment. We had already done this for the ANN models experiments, so installing the specific Python packages for the app development was the only step left. The most important are: the Flask framework, PyMySQL, Yaml, OpenCV, JSON, Pillow...etc. As explained in Chapter 3, we used the *conda* package manager to install and maintain all of these. Regarding the database, we used *XAMPP* with *Apache* and *MySQL* for debugging. Before any back-end or server application function, the first thing to create was a basic HTML skeleton file without any CSS style attached. This page was created to serve as a provisional interface to communicate with the server. But before that, the canvas was to be correctly configured. Using the **HTML Canvas API**, we created a blank 300×300px canvas in the basic HTML interface. The first basic function we wanted to create was to be to able to draw anything inside using the mouse. To do this, we needed to detect the mouse position relative to the canvas. In a newly created Javascript file, we used *onmouse* events to detect whether or not the mouse was clicked and whether it was moving or not. When the canvas is clicked, a line path is initialized. If then the mouse starts moving, we proceed to detect the mouse position. First, the canvas bounding relative to the page is determined using a JS build-in function. Now we have two options: we could use the *onmouse* property *pageX/Y*, which returns the mouse position relative to the top left corner of the whole page, or we could use the property *clientX/Y*, which returns the mouse position relative to the top left corner of the visible page at that moment. The correct option is to use *clientX/Y*, because in mobile browsers there is usually a top search bar that is constantly hidden when you move the page, causing the top left corner to be repositioned. We must then always take the mouse position relative to the visible page, so these changes do not affect us while drawing. Lastly, we have to subtract the offset of the canvas calculated before to obtain the correct position inside the canvas. The line path then follows each obtained position while the mouse keeps moving. In this state, the canvas drawing input already works, but we had yet to make sure this would work in mobile devices. For this purpose, each *onmouse* event was adapted to similar *touch* events that also work in touchscreen devices using the same mouse logic.

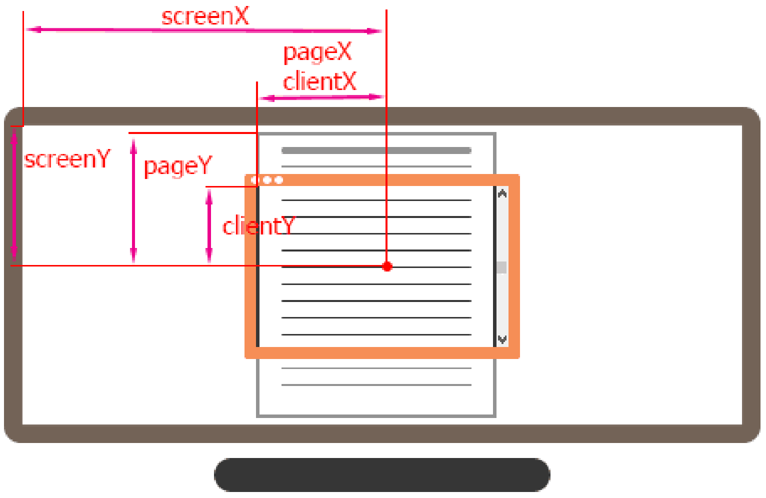


Figure 4.16: Representation of the differences between the *pageX/Y* and *clientX/Y* properties. Image retrieved from webdiz.com.ua (2019)

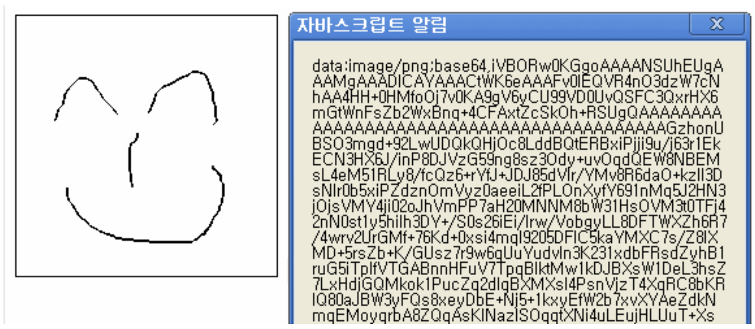
As planned in the design section, a cross-shaped dashed grid was also added to the canvas, traced in grey behind the drawing layer. To complete the canvas drawing tool we also added the functional clean button, though without any style yet. Some browsers, such as Microsoft Edge, interpret the mouse drawing inside the canvas as a dragging event, so we must set the property *preventDefault* to avoid this.

After this, we created the rest of the page form, with the three questions for the user as radio buttons and the submit button. Now we were ready to start building our server application and database structure.

A **Flask** application always starts with the import statements. A route declaration is used to tell Flask what URL should trigger whatever function we may declare inside. In our case, we declare a route for our index page, with a condition to detect if we are handling a GET or POST request. However, we can not implement any logic in the web server application without having before a database. We set up the **database** following the planned structure in the persistence design section, creating the dataset and classes tables. To start the connection with the database from the server app we need to know the database properties, such as the user, the password, the host...etc. We store these properties in a separated YAML file, so they can be easily modified when we move from our debugging environment to the deployment server.

Following the application usage flow, the first function to implement in the server app was **the display of the requested character**. First, we have to know which characters classes will be included in the dataset we are creating. The dataset definition is stored in a separated JSON file. This way, the administrator/researcher can modify the dataset's character classes at any moment without altering the server app code. Each time the server app is restarted, its checks for newly added classes in the dataset definition file, adding them as new records in the classes table. When the server receives a GET request from the client, it checks with a database query which classes have fewer records. If it finds more than one class with the same number of records, a random one is picked. This would be our requested character. This way, the dataset stays always as balanced as possible. To display this character class on the client-side, we render the HTML file using the **Jinja template engine**, which replaces a variable defined in a special syntax with our character.

After drawing the requested character and marking the answers of the questions, the next logical step is **the sending of the canvas data**. This data is stored inside a hidden field, along with the rest of the radio button fields in the form. When the submit button is pressed, before sending the POST request to the server, a Javascript function is used to convert the canvas data into a proper format for the form field. For this, we use the *toDataURL* method, which converts the canvas content into a data:URI format. Data:URIs are URLs that allow embedding small files inline in documents. If the file is textual, the text is simply added to the URL, but in case of having another format, such as our canvas image, the file is embedded as base64-encoded binary data. We can choose also the image format to be encoded as a parameter in the *toDataURL* function. Our canvas content is attached to the hidden field as a PNG image encoded in base64 format. When the POST request arrives at the server application, the data is decoded again using the Python base64 package into regular image data, ready for preprocessing.



data:image/png;base64,iVBORw0KGgoAAAANSUhEUgA
AAMgAAADICAYAAACtWK6eAAAFv0lEQVR4nO3dzW7cN
hAA4HH+0HMfoOj7v0KA9gV6yCU99VD0UvQSFC3QxrHX6
mGtWnFsZb2WxBnq+4CFAxtZcSkOh+RSUgQAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAGzhonU
BSO3mgd+92LwUDQkQHjOc8LddBQtERBxiPjji9u/j63r1Ek
ECN3HX6J/inP8DJVzG59ng8sz3Ody+uvOqdQEW8NBEM
sL4eM51RLy8/fcQz6+rYfJ+JDJ85dVlr/YMv8R6daO+kzlI3D
sNlr0b5xiPZdznOmVyz0aeeiL2fPLOnXyfY691nMq5J2HN3
jOjsVMY4ji02oJhVmPP7aH20MNNM8bW31HsOVM3t0TFj4
2nN0st1y5hilh3DY+/S0s26iEi/lrw/VobgyLL8DFTWXZh6R7
/4wrv2UrGMf+76Kd+0xsi4mqI9205DFIC5kaYMXC7s/Z8lX
MD+5rsZb+K/GUsz7r9w6qUuYudvln3K231xdbFRsdZyhB1
ruG5iTplfVTGABnnHFuV7TpqBlktMw1kDJBXsW1DeL3hsZ
7LxHdjGQMkok1PucZq2dlqBXMXsl4PsnVjzT4XqRC8bKR
lQ80aJBW3yFQs8xeyDbE+Nj5+1kxyEfW2b7xvXYAeZdkN
mqEMoyqrbA8ZQqAsKlNazlSOqqtXNi4uLEujHLUuT+Xs

Figure 4.17: Example of canvas drawing with its corresponding base64 encoding. Image retrieved from m.mkexdev.net (2010)

Once we have reached this point, we can treat the received character as a normal image, and therefore process it in whatever way we need using OpenCV (*RES-SW-01*). One interesting thing about the canvas image is that when encoded, the white space is interpreted as PNG alpha channel. We need first to **fill the transparent space** with plain white. This is done using an OpenCV colour mask. The mask is composed of every transparent pixel of the image. These pixels will be the ones that will get replaced by white pixels in the original image.

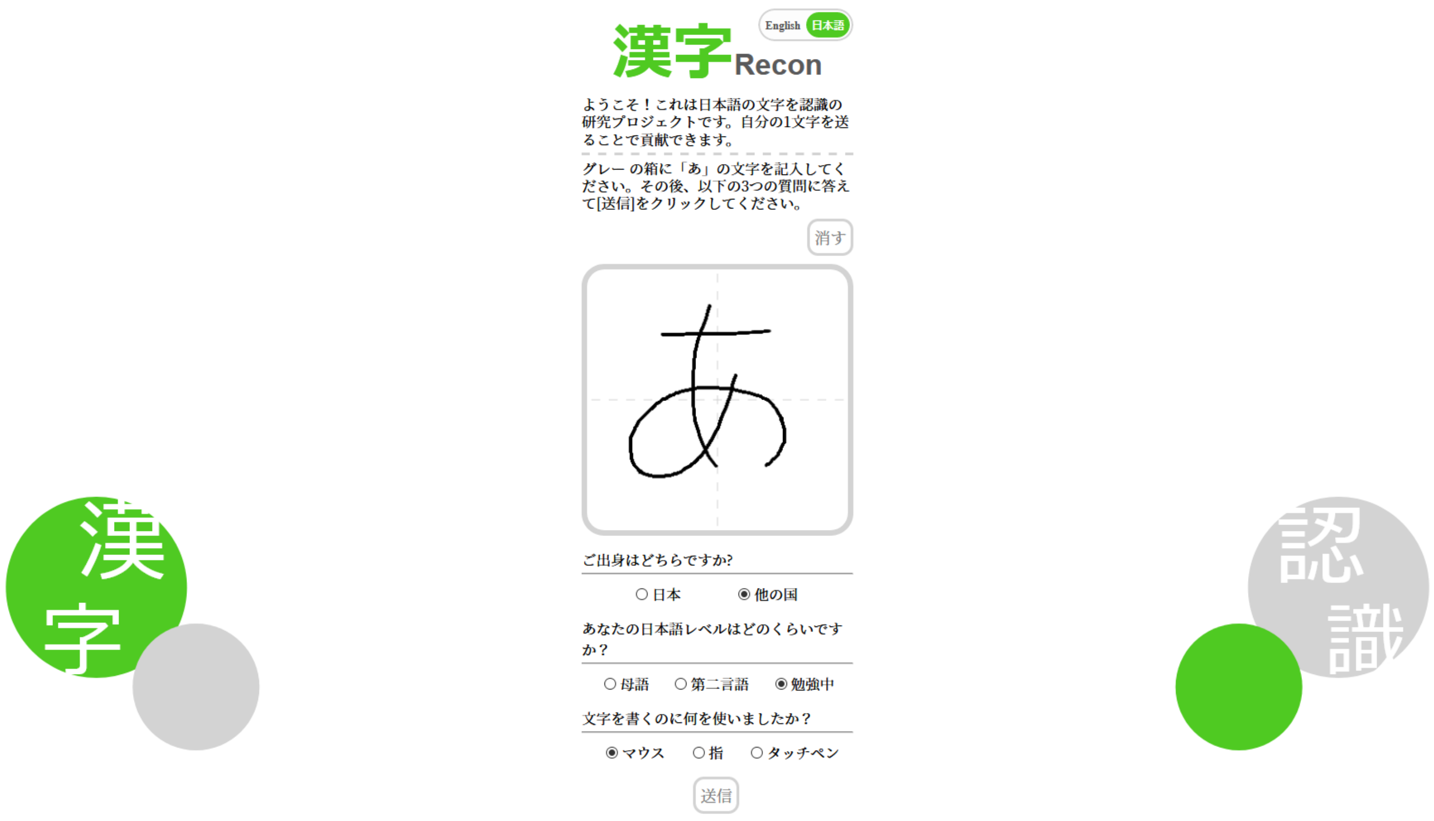
The next steps to follow would be to **erase the grey reference grid lines** from the image and detect if the user has sent an empty canvas or not. At this point, we ran into an unexpected problem. We found out that empty canvas detection and grid removal do not work equally in different browsers. Turns out canvas image data between browsers is not the same for several reasons. The same canvas element can produce exceptional pixels on different web browsers, depending on the system on which it was executed. This happens because web browsers use different image processing engines, compression levels and hashes. Moreover, operating systems use different algorithms and settings for anti-aliasing and sub-pixel rendering. Our solution to overcome this problem is to set colour ranges to include similar colour shades that may vary slightly between browsers and devices. We can now remove the grey grid lines using other OpenCV mask, but this time including a range of similar shades of grey.

For the **detection of an empty canvas** we use an example blank canvas image from the project files. We only have to subtract the current image from this one to know whether the image is empty or not. If it turns out that in fact there is no character on the submitted canvas, the rest of the process is skipped and the HTML is rendered with a variable that triggers a modal message. Otherwise, we can start to process the character image for the dataset.

In our case, the application was configured to **adapt our images to the MNIST format**, but this could be easily modified to store them in any desired format (*NRF-SYS-01*). To adapt each image to the MNIST dataset format, we have to resize them to a 28×28px size. This is done using the OpenCV INTER_AREA algorithm. This method consists on, according to the OpenCV documentation, resampling using pixel area relation, and is the recommended one for shrinking images. We also keep a copy of the original image in full size, just in case we need to use other sizes over the sample images in the future. Once we have both, we create their filename after asking the database how many records of this class are already registered, so ours will be that number plus one. The images are saved in the file system using the build-in Python I/O function *open()*. We use this instead of the traditional OpenCV's *imwrite* because this last one only supports ASCII characters for the filename, while Python's *open()* supports Unicode. We need this because our filenames include the classes, which are Japanese characters, hence only present in Unicode.

The final step in the application logic is the record of the character sample in the dataset table from the database. With this query we store the character image's path, along with the answers for the three questions. Once it's done the server application calls it a successful POST and proceeds to render the HTML page with the "Thank you" message modal, as a completion confirmation. While this modal message is open, all interactions with the page behind are disabled.

At this point, we could say we have the essential part of the application working. Now we can focus on the interface style and other features. The Flask application files structure require us to put the CSS files inside an "styles" folder, within the static files. We created the interface style following the design guide, using the planned colour palette and typography. Also, this is where we define our *media queries* to ensure that the web app is displayed correctly on any screen size. Regarding the language options, we modified a checkbox using pure CSS to look like a toggle switch, so it can be used to instantly change between English and Japanese. This localization is handled using Javascript alone. Finally, we designed the interface decorations to display them on both sides of the page when the application is showing on a large enough screen. They consist of a series of symmetrical circles in the colours of the palette with the words "*kanji* recognition" in Japanese engraved in white. They adapt to any screen size if it is wide enough to accommodate them, otherwise they disappear.



ようこそ！これは日本語の文字を認識の研究プロジェクトです。自分の1文字を送ることで貢献できます。

グレー の箱に「あ」の文字を記入してください。その後、以下の3つの質問に答えて[送信]をクリックしてください。

ご出身はどちらですか?

日本 [✓ 他の国]

あなたの日本語レベルはどのくらいですか？

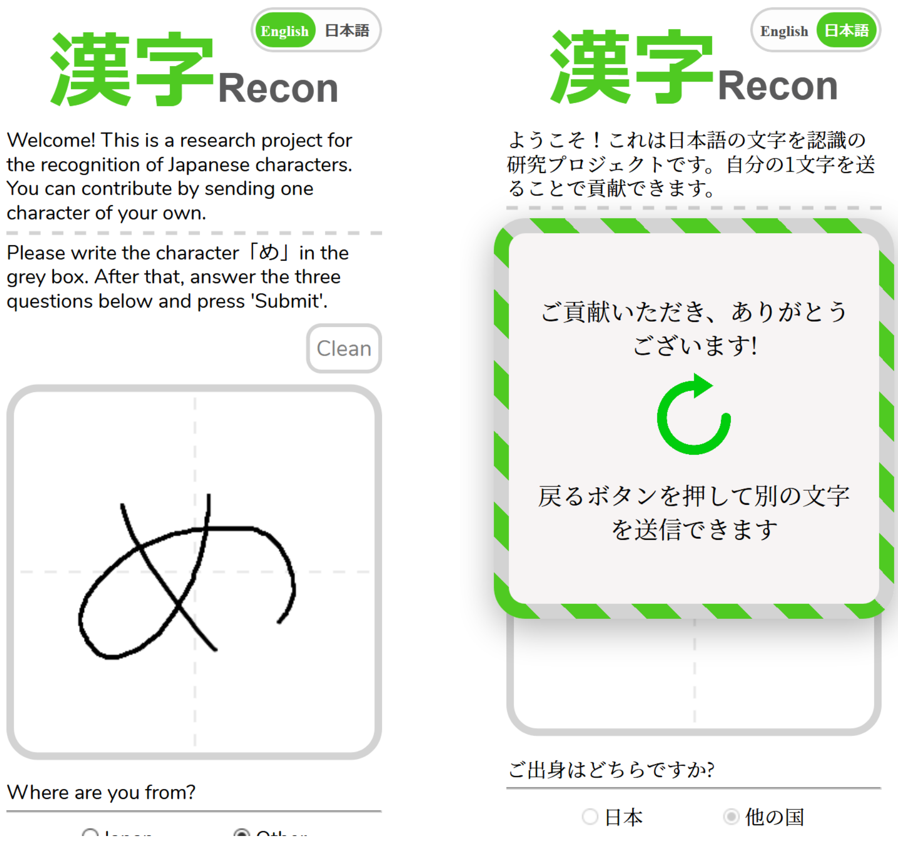
母語 第二言語 [✓ 勉強中]

文字を書くのに何を使いましたか？

[✓ マウス] 指 タッチペン

Figure 4.18: The full-width appearance of the application interface at the time of deployment (Japanese version).

Once our application is ready to collect character samples for us, we can start the process of deployment. As we mentioned previously, the application was published using a shared hosting. We managed our server using *cPanel*, which is a web hosting control panel software. Within *cPanel*, we have available a series of tools and plugins that facilitate the deployment of our application. We used *phpMyAdmin* to create and manage the database. *cPanel* also provides us with a tool to set up Python applications using the WSGI server Phusion Passenger. When we create a new instance using this tool, it also gives us a virtual environment to install any Python package we could possibly need. While installing our Flask application, we bumped into a problem related to the server capabilities (*RES-HW-01*). Using the Phusion Passenger log files, we discovered that our application was trying to use more threads than the number allowed by the hosting. This was solved by limiting the number of threads allowed in the environment to one. Once we fixed this last problem, the application was ready to go.



Welcome! This is a research project for the recognition of Japanese characters. You can contribute by sending one character of your own.

Please write the character「め」in the grey box. After that, answer the three questions below and press 'Submit'.

Where are you from?

ようこそ！これは日本語の文字を認識の研究プロジェクトです。自分の1文字を送ることで貢献できます。

ご貢献いただき、ありがとうございます!

戻るボタンを押して別の文字を送信できます

ご出身はどちらですか?

日本 [✓ 他の国]

Figure 4.19: Left: Interface in mobile size (English version) **Right**: Confirmation modal with its ”Thank you” message (Japanese version).

4.4.4 Testing and validation

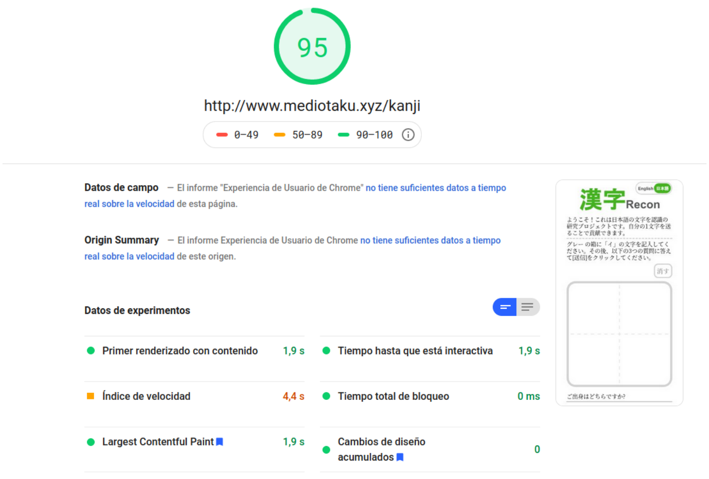
Although we already tested the web application in different desktop and mobile devices using our own local network server, we could not guarantee the proper functioning of it without the feedback of real users. Thanks to them, the application has been tested on a good range of devices with different types of touch interfaces. This includes laptops with touchscreens and iOS devices such as iPhone and iPad Pro using Apple Pencil. After a few tweaks, every touch input is correctly captured by the canvas.

The texts were translated into Japanese by the author of this thesis, but have been revised by a Japanese teacher from this same university, without having to make any correction. Moreover, several Japanese users have used the app and so far no complaints have been received.

Related with the application texts, we received the suggestion to highlight the requested character because some people tend to skip reading most of the text on the websites. We therefore chose to change its colour to our lime green, to make it stand out.

We also got the suggestion to save the form answers between requests for user's convenience. This way it would be faster to send more than one character. We implemented this feature using the browser's *sessionStorage*, saving the question answers and the language selection.

In order to objectively test the application, we used the Google PageSpeed Insights tool. PageSpeed Insights reports on the performance of a page on both mobile and desktop devices, and provides suggestions on how the page may be improved. We obtained a score of 95 points in the mobile test and 97 points in the desktop test.


95

http://www.mediotaku.xyz/kanji

0-49 50-89 90-100

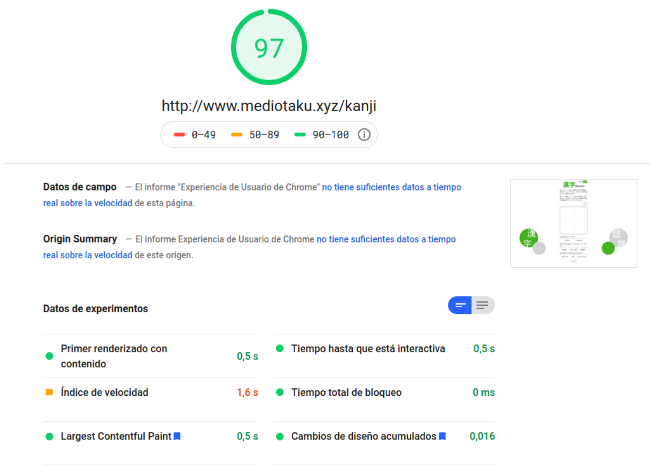
Datos de campo — El informe "Experiencia de Usuario de Chrome" no tiene suficientes datos a tiempo real sobre la velocidad de esta página.

Origin Summary — El informe Experiencia de Usuario de Chrome no tiene suficientes datos a tiempo real sobre la velocidad de este origen.

Datos de experimentos

Primer renderizado con contenido	1,9 s	Tiempo hasta que está interactiva	1,9 s
Índice de velocidad	4,4 s	Tiempo total de bloqueo	0 ms
Largest Contentful Paint	1,9 s	Cambios de diseño acumulados	0

English 日本語
漢字Recon
ようこそ！これは日本語の文字を認識の研究プロジェクトです。自分の1文字を送ることで貢献できます。
グレーの箱に「イ」の文字を記入してください。その後、以下の3つの質問に答えて[送信]をクリックしてください。
消す
ご出身はどちらですか?


97

http://www.mediotaku.xyz/kanji

0-49 50-89 90-100

Datos de campo — El informe "Experiencia de Usuario de Chrome" no tiene suficientes datos a tiempo real sobre la velocidad de esta página.

Origin Summary — El informe Experiencia de Usuario de Chrome no tiene suficientes datos a tiempo real sobre la velocidad de este origen.

Datos de experimentos

Primer renderizado con contenido	0,5 s	Tiempo hasta que está interactiva	0,5 s
Índice de velocidad	1,6 s	Tiempo total de bloqueo	0 ms
Largest Contentful Paint	0,5 s	Cambios de diseño acumulados	0,016

Figure 4.20: Left: Mobile test score **Right**: Desktop test score. Both tests detail other speed statistics.

All the project code is stored in its public *GitHub* repository (under the name Mediotaku/Kanji-Recognition), so anybody can suggest changes or fork the project. All the material generated through the app and the project itself is under an open license CC BY 4.0, whose documentation can be found in the repository (*NRF-SYS-07*). This will give the user total liberty in the use of the application and the dataset material it may generate, even for commercial purposes. Something to note is that we have already received some requests in the repository asking for instructions on how to run the project.

All in all, there are some app styles and behaviours that could be further refined in the future, but the application in this state already constitutes a **minimum viable product** that can be used to collect character samples and fulfil the rest of the objectives of this thesis.

4.5 Creation of MNIST format dataset using obtained data

In Chapter 2.2.2, we talked about the different subsets of *kanji* that are present nowadays in Japanese society. For our new dataset, we did not have the time frame to collect thousands of character classes, not to say gather enough samples of each of them. Therefore, we decided to define a dataset of 100 classes that would correctly represent the mean of Japanese characters. For this purpose, we used the data provided by the Matsushita Laboratory for Language Learning from the University of Tokyo. They have available for download ranking sheets featuring the frequency of Japanese characters along different domains, such as literary works, internet forums, Humanities, Medicine, Engineering, Medicine, Politics...etc (Tatsuhiko Matsushita (JP: 松下達彦), 2014). We took the weighted average of these lists and defined our dataset with, objectively, the 100 most used characters across all the areas. We can see all the classes in Figure 4.21, which is a collage created with our own collected samples.

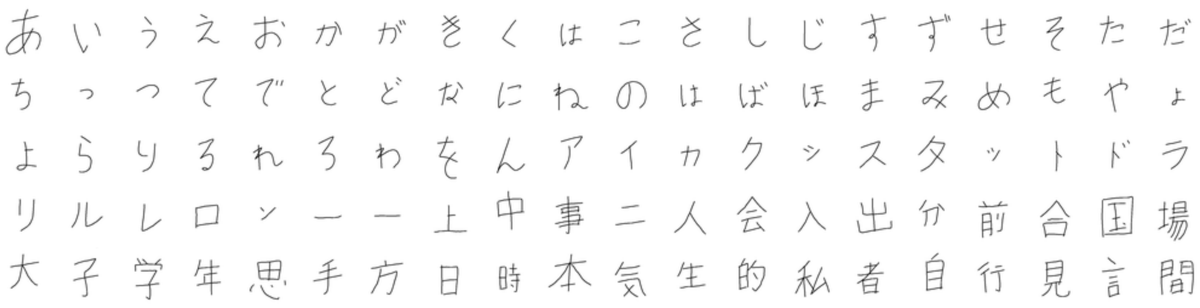

あいうえおかがきくけこさしじすずせそただ
ちっつてでとどなにねのはばほまみめもやょ
よらりるれろわをんアイカクシスタッドラ
リルレロンー一上中事二人会入出分前合国場
大子学年思手方日時本気生的私者自行見言間

Figure 4.21: Every character class present in our dataset represented by one sample. This collage has been created using the Pillow (PIL) package. Be aware this is not the actual size of the images inside the dataset.

Up until now, our web application has collected a total of **2137 samples** since its deployment. This leaves us with a dataset of 100 classes, with 20-21 samples for each class, thanks to the balancing system of our app. But before we could use it for our experiments, we have to make sure that each sample matches the class that was requested. Sometimes the users misclick the submit button before correctly writing the character, other times the written character simply does not match the requested one. Comparatively, these are a very small percentage of the samples, but still need to be reviewed for the sake of a better performance. After replacing these errors with new correct samples, the dataset is ready to be used for experiments. This is not an exclusive problem of our dataset. For example, while using ETL-8B2 we found some samples unusable due to visual artefacts generated by scanning problems.

4.6 Testing with model and comparison

In this section, we will see how well can the previously tested models perform using our dataset. The main differences with the rest are the lower number of samples per class and the presence in a single dataset of *hiragana*, *katakana* and *kanji* characters. Also, our dataset contains characters written with very different input methods: mouse, touchscreen and stylus pen. As we have statistics on the use of each of them, later we will take a look at these data.

For these tests we used the previous CNN models, because they are the ones that could give better results. To make our collect images a suitable input for Keras we used the same technique as with the ETL dataset, shuffling them and slitting 20% of the dataset for validation. At first, we tried to train the Keras CNN with the same images we collected, only resized to 28×28px. The results could have been better, as we got 0.874 of training accuracy and 0.8131 of validation accuracy. In reality, the MNIST dataset images are also inverted to have a black background and white characters. This is done to further highlight the features of the digits. We therefore followed the same format by applying a bitwise NOT operation to each image. Once we had every sample inverted, we repeated the test with Keras CNN, this time obtaining a 17.52% baseline error, 0.9438 of training accuracy and 0.8248 validation accuracy. We can compare the results in Figure 4.22. In the first test with the original images, learning takes a few epochs to take off, perhaps because it is harder to find representative features on them.

If we further test Keras CNN until 50 epochs, we get a decent 13.32% of baseline error, with 0.9807 of training accuracy and 0.8668 validation accuracy at the last epoch. These results are featured in Figure 4.23(a). Then, we tested our own CNN, but this time until 100 epochs. As we can see in Figure 4.23(b), this gave a slightly lower baseline error of 12.38%.

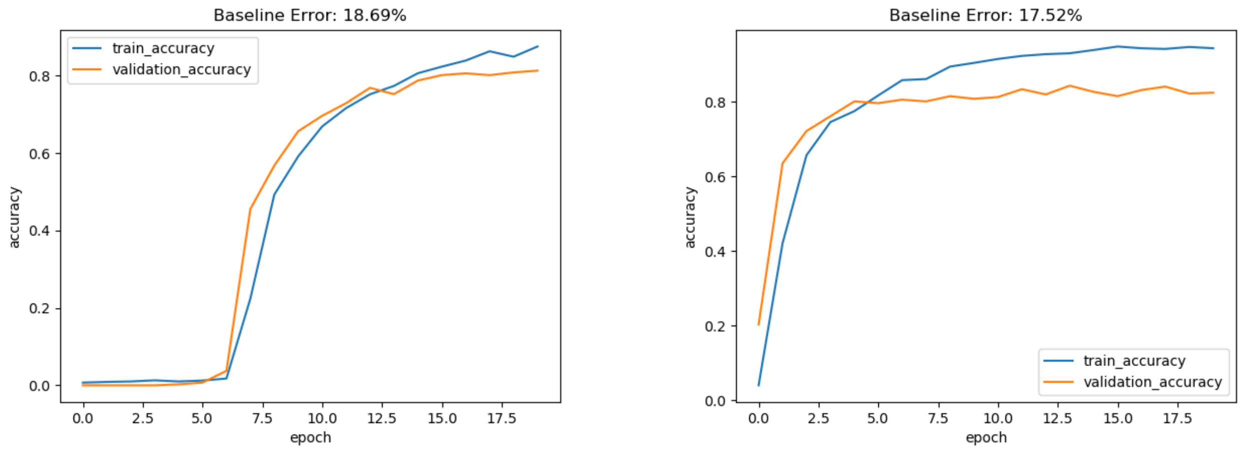


Figure 4.22: Left: New dataset with Keras CNN, batch of 16 and 20 epochs (original images) **Right**: New dataset with Keras CNN, batch of 16 and 20 epochs (inverted images).

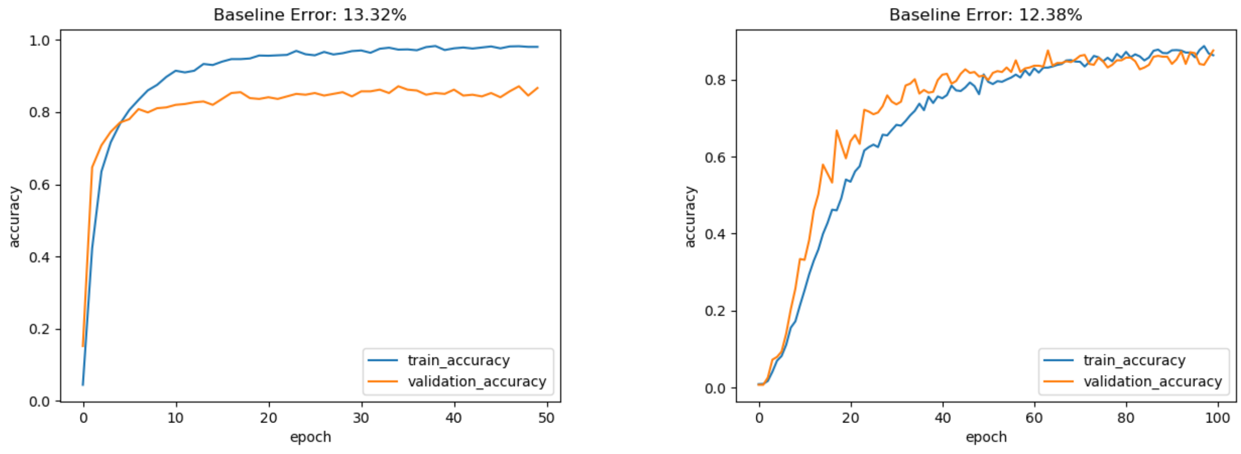


Figure 4.23: Left: New dataset with our CNN, batch of 16 and 20 epochs **Right**: New dataset with our CNN, batch of 16 and 20 epochs.

These results are not bad at all, but not as good as the ones obtained with the ETL dataset, for example. But why? We can only theorize, but it is most likely due to our data. As explained in Chapter 4.2, the data is the most important factor to consider when training an ANN, and ours may not be enough to sustain the problem. Our dataset contains some *hiragana* characters that are particularly similar between them, such as「ツ」and「シ」. Some differ in size, like「ょ」and「よ」, and others are even difficult to differentiate out of context for native speakers, like 「ー」 and 「一」 (one is used to lengthen sounds, the other is the number 1). Not to mention the *dakuten* and *handakuten* diacritics variances. Many Japanese students continue to write these variations wrong even at intermediate levels. Therefore, it is not strange to think that 20 samples may not be enough to correctly generalize these cases.

Regarding **the answers to the questions**, in Figure 4.24 we present their results through pie charts with their correspondent percentages.

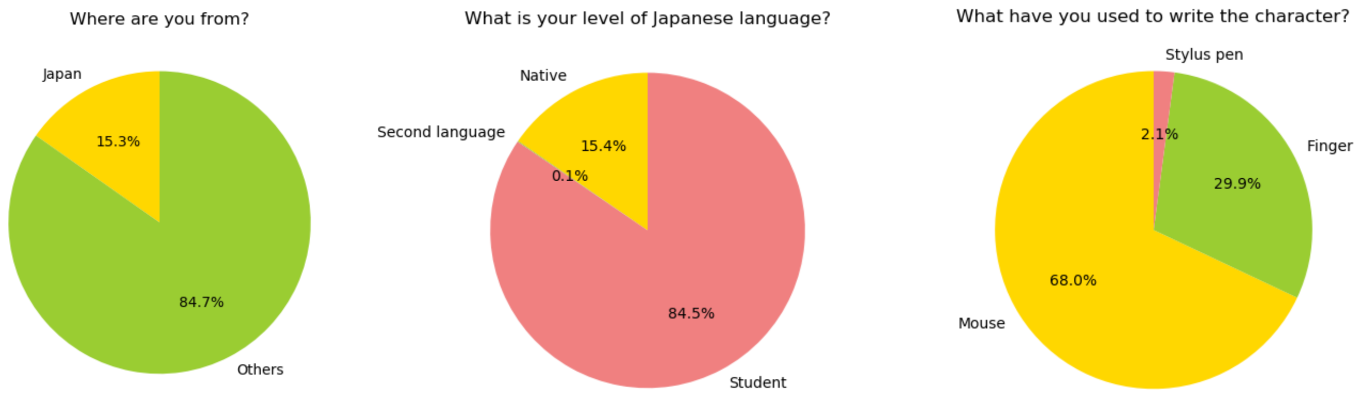


Figure 4.24: Results of the questions answered by the users about their nationality, language and form of input. These pie charts have been created with the Matplotlib package.

These percentages show interesting insights about our users during these months. It can be highlighted that we have managed to get a significant number of Japanese users to use the application (15.3%, 326 samples), which, as expected, agrees with the number of users for whom Japanese is their native language. Of course, some users may have written more than one character and the actual number of individual users may be different. It is also interesting to see the minimal presence of stylus pen users (2.1%), comparatively small but still existing.

Having these subsets of data, we could do some truly interesting experiments. The problem is that currently these subsets have very different numbers of samples between them, and, moreover, they are highly imbalanced. For those reasons, such experiments would be bound to fail, as, again, data is the most important factor in training. Still, out of curiosity, we have conducted a comparative test using our CNN between the two subsets with the most similar number of samples: the "mouse" users and the "finger" users. The results are featured in Figure 4.25.

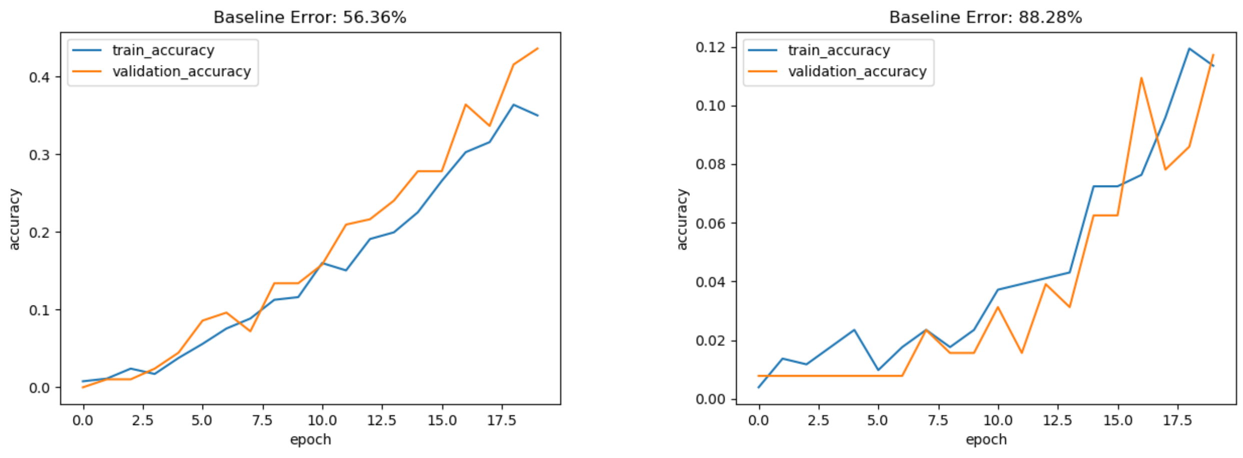


Figure 4.25: Left: “Mouse” subset with our CNN, batch of 16 and 20 epochs **Right**: “Finger” subset with our CNN, batch of 16 and 20 epochs.

As expected, the results have a disproportionate baseline error, yet we can observe that the mouse subset performs considerably better. This could be mostly due to the superior number of samples of this subset, but also because characters drawn with the finger tend to be less defined than those written through other entries.

5 Conclusion

This chapter serves as closure for the whole project. It is made up of 2 different sections: Section 5.1 presents a comparative report between the objectives met and those that can be improved, as well as a brief personal conclusion, and Section 5.2 proposes a number of possible improvements, both in terms of future research and further development of the web application.

5.1 Overview of results

In this thesis, a comprehensive research of the recognition of handwritten Japanese characters using machine learning techniques has been carried out. After all this time, we can say that we have covered all the objectives proposed at the beginning of the project.

During the period of research, every proposed topic has been explored. First, we covered the basics of OCR and its pipeline. Secondly, we presented a comprehensive description of the Japanese language and its peculiarities referred to the world of OCR, giving an extensive introduction for starters and, at the same time, providing unusual information for Japanese-experienced readers. Lastly, the neural networks section spans from basic multilayer models to convolutional networks, providing us with the knowledge needed for the development of the next chapters.

After deeply searching for publicly available Japanese datasets, we used them to carry out our own experiments, first with pre-build ANN models and afterwards with our own CNN model based on VGGNet. Using it, we obtained results that outperform the proposed human recognition rate.

Moving on, we started the development of our crowd-sourcing web application. Following the guidelines of our proposed design guide, we managed to achieve the minimum viable product status on time and therefore, start collecting new characters samples as fast as possible.

Finally, during these months the application has gathered a total of 2136 samples, allowing us to create our own dataset following the MNIST format to carry out our experiments. Using the acquired knowledge, we could diagnose the model performance and identify the most probable origin of its issues. Moreover, we were able to look into the user profiles through the answers statistics, taking note of the potential use of these subsets for further comparative experiments, provided we had enough data for both.

Personally, I consider that with this thesis I have learned a great number of new concepts and technologies that otherwise would not be part of my skills today. On the research side, I really appreciate having been able to learn from the very bottom the basic concepts of neural networks to the point of properly understanding their working. Even in relation to the Japanese language, I learned several facts that I was unaware of until that moment. On the technology side, this project has taught me a new language, Python, from absolute zero. Once I learned how to handle it more easily, we stepped into Flask, which taught me the main concepts of Python web frameworks.

Although some of the concepts were already present in my degree in the form of subjects such as Web Applications Development (DAW), for the web frameworks part, or Computer-generated Image and Video (IVC), for the image recognition part, if it were not for this project, I might not have discovered technologies like Keras or Tensorflow, since these are not part of the curriculum of the degree.

In general, I am very grateful to this project for giving me knowledge that will not stay between these pages, but that I will be able to export to my future works.

Finally, I hope that this project has contributed to the approach of different disciplines, such as machine learning and the Japanese language, so that in the future they can together achieve advances that benefit both in their fields.

5.2 Proposal of possible improvements

During the development of the project, a series of ideas have emerged that were not possible to implement due to our time constraints and because they were outside the objectives of the project.

On the research side, the next logical step would be to take the lessons learned in recognizing individual characters and apply them to text recognition. In this way, the application of contextual techniques such as those explained in Chapter 2.1.2 could be investigated. Since Japanese is a highly contextual language, the use of dictionary techniques could greatly improve recognition performance. Of course, this would not be easy, since before it would be necessary to carry out a segmentation process to separate some words from others, which is a particularly difficult challenge in Japanese since there is typically no separation between words.

Besides, taking advantage of this attempt at word-level recognition, the application of recurrent neural networks and LSTM (Long Short Term Memory) could be investigated, the advantage of which is that they take into account the information from previous predictions to inform the current one. This could greatly improve the performance of the model, as context can be automatically assumed by the neural network.

On the application side, it is our wish that it continue to be an open-source tool used by researchers to easily obtain open training data of Japanese language. Although the dataset is available from its *GitHub* repository, we believe that the inclusion of a simple download system in the application itself could further facilitate the process.

Finally, the application's data balancing system could be improved by including the parameters of the questions. Although we cannot predict user answers, we can detect, for example, from what type of device is accessing the application. In this way, if you were accessing from a mobile device we can deduce that is more likely that you will use its touchscreen as input, thus we can request the character class written by touchscreen with fewer samples. This will help reduce the imbalance of the subset, yet the best solution is to have a similar number of samples between them.

Bibliography

Aboura, K. (2016, 06). A naive bayes classifier for character recognition..

AIST. (2014). *About the etl character database.* Retrieved from http://etlcdb.db.aist.go.jp/ ([Online; accessed July 9, 2020])

Albert Lai. (2018). *Gradient descent for linear regression explained.* Retrieved from https://blog.goodaudience.com/gradient-descent-for-linear-regression-explained-7c60bc414bdd ([Online; accessed June 26, 2020])

Alexander Amini. (2020). *Introduction to deep learning.* Retrieved from http://introtodeeplearning.com/ ([Online; accessed June 21, 2020])

ambs. (2019). *Plotting functions with latex tikz.* Retrieved from http://null.zbr.pt/?p=2259 ([Online; accessed June 25, 2020])

Andreas Refsgaard. (2020). *Looking inside neural nets.* Retrieved from https://ml4a.github.io/ml4a/looking_inside_neural_nets/ ([Online; accessed June 24, 2020])

Andrew Ng. (2018). *Machine learning mooc machine learning course.* Retrieved from https://www.coursera.org/learn/machine-learning ([Online; accessed June 26, 2020])

Baskin, C., Liss, N., Mendelson, A., & Zheltonozhskii, E. (2017, 07). Streaming architecture for large-scale quantized neural networks on an fpga-based dataflow platform.

Chart by u/Danilinky. (2019). *"hiragana+katakana+basic kanji" printable letter-sized chart.* Retrieved from https://www.reddit.com/r/LearnJapanese/comments/b0jlyt/i_made_an_allinone_hiraganakatakanabasic_kanji/ ([Online; accessed June 4, 2020])

Chi-Feng Wang. (2019). *The vanishing gradient problem.* Retrieved from https://towardsdatascience.com/the-vanishing-gradient-problem-69bf08b15484 ([Online; accessed July 1, 2020])

Clanuwat, T., Bober-Irizar, M., Kitamoto, A., Lamb, A., Yamamoto, K., & Ha, D. (2018). Deep learning for classical japanese literature. *CoRR*, *abs/1812.01718*. Retrieved from http://arxiv.org/abs/1812.01718

Clanuwat, T., Lamb, A., & Kitamoto], A. (2019). *Kuronet: Pre-modern japanese kuzushiji character recognition with deep learning.*

François Chollet. (2015). *Simple mnist convnet.* Retrieved from `https://keras.io/examples/vision/mnist_convnet/` ([Online; accessed July 27, 2020])

George V Jose. (2019). *Useful plots to diagnose your neural network.* Retrieved from `https://towardsdatascience.com/useful-plots-to-diagnose-your-neural-network-521907fa2f45` ([Online; accessed July 27, 2020])

Grant Sanderson. (2018). *But what "is" a neural network?* Retrieved from `https://www.youtube.com/playlist?list=PLZHQObOWTQDNU6R1_67000Dx_ZCJB-3pi` ([Online; accessed June 23, 2020])

Grębowiec, M., & Protasiewicz, J. (2018, 09). A neural framework for online recognition of handwritten kanji characters. In (p. 479-483). doi: 10.15439/2018F140

Gus Polly from Wikimedia. (2016). *Correct use of genkō yōshi.* Retrieved from `ByGusPolly-ja: :Genkoyoshi.jpg,CCBY-SA4.0,https://commons.wikimedia.org/w/index.php?curid=51496839` ([Online; accessed June 7, 2020])

How many hidden layers should i use? (n.d.). Retrieved from `http://www.faqs.org/faqs/ai-faq/neural-nets/part3/section-9.html` ([Online; accessed June 24, 2020])

IEEE. (2008). *Especificación de requisitos según el estándar de ieee 830.* Retrieved from `https://www.fdi.ucm.es/profesor/gmendez/docs/is0809/ieee830.pdf` ([Online; accessed July 17, 2020])

Image by 中の人（2 号）. (2017).「明朝体はいらない子？」. Retrieved from `https://www.fontkaruta.com/single-post/2017/06/13/` ([Online; accessed June 10, 2020])

Image by 日本書道協会. (n.d.). 日常に生きる書体. Retrieved from `https://www.u-can.co.jp/shodo/hand_write/study/in_the_life/` ([Online; accessed June 8, 2020])

Image from アップシェア Blog. (2017). *Google chrome* でも *font-size* を *10px* 以下で表示させる *css.* Retrieved from `http://blog.upshare.co.jp/2017/07/3730/` ([Online; accessed June 10, 2020])

Image from 変体仮名 bot. (2020). きそばやぶ. Retrieved from `https://twitter.com/hengana_bot/status/1268082512674185218` ([Online; accessed June 16, 2020])

Information from Ｚ会作文クラブ. (2012). ゼロから学ぶ作文の書き方: 原稿用紙の使い方. Retrieved from https://service.zkai.co.jp/el/course/sakubun_club/sakubun-kakikata/genkouyoushi.html ([Online; accessed June ６, 2020])

Jason Brownlee. (2016). *Overfitting and underfitting with machine learning algorithms.* Retrieved from https://machinelearningmastery.com/overfitting-and-underfitting-with-machine-learning-algorithms/ ([Online; accessed June 28, 2020])

Jason Brownlee. (2018a). *Difference between a batch and an epoch in a neural network.* Retrieved from https://machinelearningmastery.com/difference-between-a-batch-and-an-epoch/ ([Online; accessed June 28, 2020])

Jason Brownlee. (2018b). *A gentle introduction to dropout for regularizing deep neural networks.* Retrieved from https://machinelearningmastery.com/dropout-for-regularizing-deep-neural-networks/ ([Online; accessed June 28, 2020])

Kitamura, K., Nikaido, M., Michio, Y. N., & Yasuda. (2015). 文字認識のための漢字データベースの解析, an analysis of kanji character database for character recognition. Retrieved from https://www.google.com/url?sa=t&rct=j&q=&esrc=s&source=web&cd=&ved=2ahUKEwiJ66z2rcHqAhXcA2MBHUPsCvwQFjAAegQIBhAB&url=https%3A%2F%2Fmeisei.repo.nii.ac.jp%2F%3Faction%3Drepository_action_common_download%26item_id%3D671%26item_no%3D1%26attribute_id%3D22%26file_no%3D1&usg=AOvVaw3Nm_XydRLgZ4GkUo_UeCoZ ([Online; accessed July 9, 2020])

klo uo. (2012). *ndimage/morphology - binary dilation and erosion?* Retrieved from http://scipy-user.10969.n7.nabble.com/ndimage-morphology-binary-dilation-and-erosion-td8567.html ([Online; accessed June 29, 2020])

Lamb, A., Clanuwat, T. Kitamoto, A. (2020). Kuronet: Regularized residual u-nets for end-to-end kuzushiji character recognition. sn comput. sci. 1, 177 (2020). Retrieved from https://doi.org/10.1007/s42979-020-00186-z ([Online; accessed June 15, 2020])

Mathworks, by Auralius Manurung. (2016). *Elliptic fourier for shape analysis.* Retrieved from https://www.mathworks.com/matlabcentral/fileexchange/32800-elliptic-fourier-for-shape-analysis ([Online; accessed June 1, 2020])

Michael Garris. (2019). *The story of the mnist dataset.* Retrieved from https://www.youtube.com/watch?v=oKzNUGz21JM ([Online; accessed July 8, 2020])

Michael Nielsen. (2019). *Neural networks and deep learning.* Retrieved from http://neuralnetworksanddeeplearning.com/chap6.html ([Online; accessed July 1, 2020])

Mikel Bober-Irizar. (2019). *The 10 classes of kuzushiji-mnist, with the first column showing each character's modern hiragana counterpart.* Retrieved from `https://github.com/rois-codh/kmnist` ([Online; accessed July 8, 2020])

m.mkexdev.net. (2010). *[html5 silseub] canvas-e geulin geulim-eul imijilo mandeulgi (original title in korean).* Retrieved from `https://m.mkexdev.net/106` ([Online; accessed July 25, 2020])

Nacho Alonso. (2019). *¿qué es un wsgi?* Retrieved from `https://medium.com/@nachoad/que-es-wsgi-be7359c6e001` ([Online; accessed July 12, 2020])

Nazanin Delam. (2016). *Single-layer artificial neural networks.* Retrieved from `https://medium.com/@nazanindelam/single-layer-artificial-neural-networks-a91cf3752a86` ([Online; accessed June 23, 2020])

Nguyen, V., Brooke, J., & Baldwin, T. (2017, September). Sub-character neural language modelling in Japanese. In *Proceedings of the first workshop on subword and character level models in NLP* (pp. 148–153). Copenhagen, Denmark: Association for Computational Linguistics. Retrieved from `https://www.aclweb.org/anthology/W17-4122` doi: 10.18653/v1/W17-4122

Pranoy Radhakrishnan. (2017). *What are hyperparameters ? and how to tune the hyperparameters in a deep neural network?* Retrieved from `https://towardsdatascience.com/what-are-hyperparameters-and-how-to-tune-the-hyperparameters-in-a-deep-neural-network-d0604917584a` ([Online; accessed June 28, 2020])

Qu J., Lu X., Liu L., Tang Z., Wang Y. (n.d.). A method of density analysis for chinese characters. *Natural Language Processing and Chinese Computing. NLPCC 2014*. Retrieved from `https://link.springer.com/chapter/10.1007/978-3-662-45924-9_6#citeas` ([Online; accessed June 10, 2020])

Sagar Sharma. (2017). *Activation functions in neural networks.* Retrieved from `https://towardsdatascience.com/activation-functions-neural-networks-1cbd9f8d91d6` ([Online; accessed June 25, 2020])

Shi, B., Hou, R., Mazurowski, M., Grimm, L., Ren, Y., Marks, J., … Lo, J. (2018, 02). Learning better deep features for the prediction of occult invasive disease in ductal carcinoma in situ through transfer learning. In (p. 98). doi: 10.1117/12.2293594

S.N. Srihari, G. Srikantan, T. Hong and S.W. Lam. (1996). *Research in japanese ocr, handbook on optical character recognition and document image analysis.* World Scientific Publishing Company. ([Online; accessed June 18, 2020])

S.N. Srihari, T. Hong and Z. Shi. (1997). Cherry blossom: A system for japanese character recognition. Retrieved from https://cedar.buffalo.edu/japanese/files/sdiut97.pdf ([Online; accessed June 18, 2020])

Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., & Salakhutdinov, R. (2014). Dropout: A simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, *15*(56), 1929-1958. Retrieved from http://jmlr.org/papers/v15/srivastava14a.html

Sumit Saha. (2018). *A comprehensive guide to convolutional neural networks —the eli5 way.* Retrieved from https://towardsdatascience.com/a-comprehensive-guide-to-convolutional-neural-networks-the-eli5-way-3bd2b1164a53 ([Online; accessed June 30, 2020])

Tatsuhiko Matsushita (JP: 松下達彦). (2014). *Character database of modern japanese (cdj) version 2.0.* Retrieved from http://www17408ui.sakura.ne.jp/tatsum/english/databaseE.html#cdj ([Online; accessed July 27, 2020])

Tsai, C. (2016). Recognizing handwritten japanese characters using deep convolutional neural networks.. Retrieved from http://cs231n.stanford.edu/reports/2016/pdfs/262_Report.pdf

Ujjwal Karn. (2016). *An intuitive explanation of convolutional neural networks.* Retrieved from https://ujjwalkarn.me/2016/08/11/intuitive-explanation-convnets/ ([Online; accessed June 30, 2020])

Usenet newsgroup sci.lang.japan, maintained by Ben Bullock. (n.d.). *sci.lang.japan frequently asked questions.* Retrieved from https://www.sljfaq.org/afaq/afaq.html ([Online; accessed June 8, 2020])

Velek, O., & Nakagawa, M. (2002, 08). The impact of large training sets on the recognition rate of off-line japanese kanji character classifiers. In (p. 106-110). doi: 10.1007/3-540-45869-7_13

W3C Standard for Japanese. (2012). *Requirements of japanese text layout.* Retrieved from https://www.w3.org/TR/2012/NOTE-jlreq-20120403/ ([Online; accessed June 7, 2020])

webdiz.com.ua. (2019). *Razmery i prokrutka elementov na veb-stranitse (original title in russian).* Retrieved from http://webdiz.com.ua/glava6-rabota-s-dom/razmery-i-prokrutka-elementov-na-veb-straniceb-s/ ([Online; accessed July 25, 2020])

Yani, M., Irawan, S., & S.T., M. (2019, 05). Application of transfer learning using convolutional neural network method for early detection of terry’s nail. *Journal of Physics: Conference Series*, *1201*, 012052. doi: 10.1088/1742-6596/1201/1/012052

Yin, F., Wang, Q.-F., Zhang, X.-Y., & Liu, C.-L. (2013). Icdar 2013 chinese handwriting recognition competition. In (p. 1464-1470).

List of Acronyms y Abbreviations

AD	Anno Domini.
AIST	National Institute of Advanced Industrial Science and Technology.
ANN	Artificial Neural Network.
ASCII	American Standard Code for Information Interchange.
BC	Before Christ.
CEDAR	Center of Excellence for Document Analysis and Recognition.
CNN	Convolutional Neural Network.
CODH	Center for Open Data in the Humanities.
ETL	ElectroTechnical Laboratory.
GPU	Graphical Processing Unit.
GUI	Graphical User Interface.
IDE	Integrated Development Environment.
IEEE	Institute of Electrical and Electronics Engineers.
JEITA	Japan Electronics and Information Technology Industries Association.
JIS	Japanese Industrial Standard.
JP	in Japanese.
JSON	JavaScript Object Notation.
LSTM	Long Term Short Memory.
MICR	Magnetic Ink Character Recognition.
ML	Machine Learning.
MNIST	Modified National Institute of Standards and Technology.
NIJL	National Institute of Japanese Literature.
NTT	Nippon Telegraph and Telephone.
OCR	Optical Character Recognition.
OpenCV	Open Source Computer Vision Library.
PBL	Project-Based Learning.

RELU	Rectified Linear Unit.
ROIS	Research Organization of Information and Systems.
TFG	Trabajo Final de Grado.
TFM	Trabajo Final de Máster.
WSGI	Web Server Gateway Interface.
YAML	Originally Yet Another Markup Language, now YAML Ain't Markup Language.