Hafumbukū: A Text Recognition Model for the Manchu Script

Citation

Permanent link
https://nrs.harvard.edu/URN-3:HUL.INSTREPOS:37376405

Terms of Use
This article was downloaded from Harvard University’s DASH repository, and is made available under the terms and conditions applicable to Other Posted Material, as set forth at http://nrs.harvard.edu/urn-3:HUL.InstRepos:dash.current.terms-of-use#LAA

Share Your Story
The Harvard community has made this article openly available. Please share how this access benefits you. Submit a story.

Accessibility
Hafumbukū: A Text Recognition Model for the Manchu Script

Chao Cheng

Advised by George A. Alvarez & Mark C. Elliott

Presented to the Department of Computer Science and
the Department of East Asian Languages and Civilizations
in partial fulfillment of the requirements
for the degree with honors of Bachelor of Arts

Harvard College
Cambridge, Massachusetts
March 23, 2023
To all the friends who stayed up with me
until the sun’s first light
# Table of Contents

Acknowledgements 1

1 Introduction 2
   1.1 Who Were the Manchus? 3
   1.2 Origins of the Manchu Script 5
   1.3 A Dying Language 6

2 Text Recognition for Manchu 7
   2.1 An Overview of Manchu 8
   2.2 Background and Related Work 10
   2.3 Data Collection 14
   2.4 Training and Model Architecture 19
   2.5 Finetuning 22
   2.6 Results 24

3 Into The Archives 26
   3.1 A Survey of Manchu Documents 26
   3.2 A Sample Reading 27

4 Conclusion 31

Appendix 32

Bibliography 49
I would like to thank my two wonderful advisors, Professor George Alvarez and Professor Mark Elliott, for their valuable time and steadfast encouragement. It is one thing to set aside time each week to answer my questions and discuss research problems; it is another to constantly provide a moral pillar of advice and support, and I am endlessly grateful to you both.

I would also like to thank my graduate tutor, Jack Rabinovitch, as well as the entire Department of East Asian Languages and Civilizations, for making 2 Divinity Ave into a home-away-from-home during these last two years. Of all the paths I imagined for myself as a starry-eyed freshman, somehow this was the least likely — and yet most fortunate — one I could have chosen.

Finally, were it not for the array of amazing mentors that I was lucky enough to learn from during my time here at Harvard, I think my outlook on life today would be very different. To Professor Jasanoff, thank you for showing me that history is not just what is found in the pages of textbooks. To Professor Mitzenmacher, thank you for demonstrating that there is no such thing in life as an unbiased coin flip. To Professor Bol, thank you for teaching me what it means to be a real junzi. And to Professor McKenna, thank you for pushing me to study the Manchus, all those years ago.

Cindahan
Chapter 1

Introduction

“No nations that have abandoned their own language and taken up another nation’s language have prospered.”

Hong Taiji 皇太极, Emperor Taizong of Qing

One hundred and twelve years ago — before the end of Imperial China and the dawn of modern Chinese history — the sounds of the Manchu language could still be heard throughout the great realm of the Qing dynasty, from the snow-covered Changbai mountains in the northeast and the rugged steppes of the Tibetan plateau to the west, on the streets leading up to the Imperial Palace in Beijing and within the walls of the garrison cities dotting the urban countryside, and, perhaps most importantly, in the lives and everyday conversations of countless people of Manchu heritage, for whom the language represented something akin to culture, identity, history, and much more.

The first of five official languages of the Qing dynasty, Manchu (along with Chinese, Mongolian, Tibetan, and Chagatai) played an important role as the *lingua franca* of a vast, polyethnic empire, particularly in the military administration of the imperial periphery. Wherever they could, the Manchu Qing emperors endeavored to integrate existing models of government into their universal framework of empire. Thus, in Mongolia, bannermen were required to speak both the local language as well as Manchu; in Tibet, imperial *ambans* acted as official liaisons to the Dalai Lama, governing in the name of the emperor and communicating with the capital in Beijing via secret messages written on scrolls in Manchu; in

---

1Chagatai was the literary predecessor of the modern Uzbek and Uyghur languages.
Xinjiang, the use of Manchu as an intermediary language to bridge the various local Turkic dialects was instrumental to governing the region; and, of course, in the capital at Beijing, all court matters were meticulously documented by an army of scribes, each furiously recording every document into two versions: one in Chinese, and one in Manchu.

During the nearly three hundred year-long period that the Qing dynasty held power from the capture of Beijing in 1644 to the last months before the Xinhai Revolution in 1911, the capacity of Manchu as both the mark of the ruling elite and the language of the frontier was vital to the success and growth of the dynasty; and through understanding it, one gains an unparalleled and unique view into the bureaucracy and society of Late Imperial China. Even today, the full legacy of the language is still not fully appreciated, for hidden within the pages of millions of Manchu documents across countless libraries and archives lies an entire chapter waiting to be explored, as written by those who ruled it. But before diving into the world of qingwen 清文 (“the Qing language,” as it was known during the time), we must first understand the people behind it: the Manchu people.

1.1 Who Were the Manchus?

Let us return to the middle of the 16th century, during the height of the Ming dynasty. Living along the tributaries and basins of the great Heilongjiang 黑龙江 (sahaliyan ula in Manchu, meaning “Black River”) were a group of loosely associated tribes known as the Jurchens, who engaged in agriculture, animal husbandry, and hunting and fishing as their way of life, and were especially renowned for their prowess in archery and horseback riding. In the 12th and 13th centuries, the Jurchens had united under and conquered northern China and established their own Jin dynasty (lasting from 1115–1234), but were subsequently overthrown and made to pay tribute by the invading Mongol Yuan dynasty, which was in turn supplanted by the Ming dynasty in 1368. After the fall of the Jin, the lands inhabited by the weakened Jurchen tribes were easily brought back into the imperial fold by the Yuan and their status as tributary subjects was maintained when the Ming rose to power, leading to the situation in the mid-1500s in which the Jurchen people were split into three roughly dis-
unified groups, according to the records of Ming officials: the Jianzhou 建州 group, situated near the Chinese-populated Liaodong peninsula; the Haixi 海西 group, who lived further to the north and "west of the sea"; and the Yeren 野人, inhabiting the areas near the Pacific coast north of Korea. Inhabiting the northeastern frontier, the three groups were themselves split into multiple independent tribes, each with their own official tributary relationships with the Ming. It was from this relatively peaceful and quiet backdrop that the family that would go on to establish the Qing dynasty emerged.

The Jurchen reunification began in the year 1583, when a young man named Nurhaci of the house of Aisin Gioro ascended to the leadership of the Jianzhou Left Branch after his father, the previous chieftain, had been slain by a rival Jurchen clan clandestinely supported by the Ming. At a mere twenty-four years of age, Nurhaci purportedly could count on just thirteen men under his command; yet over the course of the next three decades, he succeeded in uniting the disorganized Jurchen tribes in a whirlwind of conquest, culminating in the creation the Later Jin dynasty in 1616 (reviving in name the former Jin dynasty from four centuries earlier). Two years later, after consolidating control over the Jurchen homeland, Nurhaci officially declared his intention to challenge Ming hegemony by issuing the "Seven Grievances" and launching the invasion of Liaodong, an ethnically Han Chinese commandery of the Ming. Upon his death in 1626, his son and successor, Hong Taiji, took up the mantle, setting his sights upon all lands north of the Great Wall and proclaiming the formation of the Qing dynasty in 1636 (he thought his father’s choice of name – that of a former subject – to be too reminiscent of past humiliation).

In the midst of this decades-long war with the Ming, the new Khan made a momentous decision: from that day forward, all those under his banners were no longer to be known as Jurchens; they were now Manchus. \(^2\) Interestingly, this definition was not restricted to those who had been Nurhaci’s original followers, nor was it limited to the members of the three Jurchen groups prior to unification. Rather, it included (in addition to the previous

\(^2\)Where the name Manchu comes from and why Hong Taiji chose it is unclear. According to the official record of Researches on Manchu Origins from the Qianlong era, the name comes from the bodhisatva Manjushri, but it could also derive from the name the Jianzhou chieftain during the early Ming, who was known as Li Manzhu 里满住.
categories) Mongols who had been brought into the fold via diplomacy and subjugation; Han Chinese soldiers and villagers who had defected from the Ming during the early years of the campaign; and still other tribes from even further north, who spoke not the Jurchen tongue but many closely related languages within the Tungusic linguistic continuum. It was these groups of people, the first Manchus, who formed the core of the early Eight Banners 八旗, an initially military-related organization that would go on to become, as some would argue, the defining part of Manchu culture and identity during the mid to late Qing era — so much so that, eventually, Manchu and qiren 旗人 (bannerman/bannerwoman) became synonymous.

Thus, it can be seen that from the very beginning, the term “Manchu” was designed to be fluid in nature — a characteristic it would retain throughout the centuries of imperial rule. When Hong Taiji first conceived it, he sought to do away with inter-tribal and inter-ethnic divisions in his newborn dynasty, still fresh with the blood of conquest; moreover, by uniting his loyal followers under the Manchu banner, he laid the foundations for the stratified society of Qing China, in which a Manchu elite ruled over a non-Manchu majority. Later on, the Qianlong Emperor would define being Manchu as continuing “the old traditions of shooting, riding, and speaking Manchu, along with being able to handle a lance and a sword” — in other words, be a good warrior and speak the language. As Jonathan Spence wrote, “The Manchus tried to maintain their martial superiority through such practices as hunting and mounted archery; and they emphasized their natural cultural distinctness by using the Manchu spoken and written language.” It was for this reason that the Manchu language came to signify something greater than itself; and consequently, why it is crucial to understanding Qing history.

1.2 Origins of the Manchu Script

According to the Old Manchu Chronicles, the Manchu script was invented in 1599 by order of Nurhaci. Prior to this, Manchu had no written script (the Jurchens of the 13th century had used a script derived from the Khitan script, itself derived from Chinese characters, but
after the fall of the Jin dynasty in 1234, usage of the script declined until it fell out of use); instead, they relied on Mongol scribes within their ranks to write down important notices and records when necessary. Thus, Nurhaci gave two of his translators, Erdeni and Dahai, the task of creating an alphabet for Manchu by adapting the Mongolian script. In 1632, under Nurhaci’s successor Hong Taiji, the script was revised to include diacritics (“dots and circles”), leading to the form of Standard Manchu that survives to the present day.

1.3 A Dying Language

In the world of the present, only the remnants of the Manchu language still remain, spoken in no more than a handful of villages in China’s northeast and by a few thousand secondary speakers. While there are some ten million people identify as being descended from Manchus, the vast majority of them have no knowledge of the language. Nevertheless, there is still yet hope for Manchu: in the last few decades, studying and learning how to read Manchu has become more and more popular among historians, linguists, and anthropologists who study the Qing dynasty, following the publication of groundbreaking works such as *A Translucent Mirror* by Pamela Crossley in 1999 and *The Manchu Way* in 2001 by Mark Elliott which utilize Manchu-language sources to examine historical narratives in a new light. In fact, it is this very phenomenon that leads us to the task that presents itself today: can a computer read Manchu?

---

3 According to *The Economist*, by 1979, there were only 50 fluent Manchu speakers left.
Chapter 2

Text Recognition for Manchu

The first step to building an end-to-end text recognition model is recognizing Manchu words. In this chapter, we will begin with a brief overview of the Manchu script, followed by a survey of the existing work that has been done on the Manchu word recognition task. We will then proceed to train and evaluate a convolutional neural network to recognize Manchu words.
2.1 An Overview of Manchu

The Manchu script, similar to its predecessor — traditional Mongolian — is written in vertical fashion from top to bottom, with lines read from left to right. For instance, the following romanized excerpt:

\[
\text{emu inenggi singgeri siyanšeng emu fempin jasigan bargiyame bahaha. cohome terei goro bai niyamangga usin singgeri i buhe jasigan inu.}^4
\]

is written in the Manchu script as:

![Manchu script]

A Brief Remark on Terminology

For the purposes of this thesis, we refer to standalone units of language (for instance, the endonym \textit{manju}) as \textit{words}, which, much like in English, can be combined to form meaningful sentences. We refer to the elements that serve as the “building blocks” of words, such as \textit{man} and \textit{ju}, as \textit{syllables} or \textit{characters}, which can also carry linguistic meaning but cannot

\footnote{This example comes from one of the lessons of \textit{Manchu: A Textbook for Reading Documents} by Gertraude Roth Li. It means: “One day Mr. Mouse received a letter; it was a letter sent by his relative Country Mouse who lived far away.”}
stand alone like words. Lastly, we refer to the individual sounds and their corresponding visual representations as letters, which may change in appearance depending on where in the word they are located. Note that these definitions are slightly different than how they are defined in Chinese, which consists of monosyllabic characters that combine to form words, and in English, for which letters and characters are synonymous. Thus, while the process of digitally recognizing text using software is traditionally referred to as “Optical Character Recognition” (OCR), we prefer to use the term “text recognition” — which can be used more ambiguously to refer to recognition of individual letter strokes, recognition of character forms, or recognition of entire words — though, occasionally, we will still use “OCR” in relation to Manchu, in light of its the historical usage within the field of computer vision.

2.1.1 Alphabet and Romanization

There are multiple romanization systems for transliterating Manchu, but the one used by most modern academics (and in this thesis) is the Möllendorf system, which consists of 6 vowels — $a$, $e$, $i$, $o$, $u$, and $ā$ — and the following consonants: $b$, $c$, $d$, $f$, $g$, $h$, $j$, $k$, $l$, $m$, $n$, $ng$, $p$, $r$, $s$, $š$, $t$, $w$, $y$, and $ž$. Additionally, there are 10 letters used to represent Chinese sounds.

2.1.2 Morphology

Manchu is an agglutinative language, meaning that each word consists of a sequence of morphemes (syllables) concatenated together. Thus, while words may share the same root, their meaning will change depending on their suffixes. For instance, the following list of words are all related to the verb stem $afa$- (“to attack, to fight”) yet they each have subtle differences:

- $afambo$: to attack, to fight
- $afanumbi$: to attack together, to fight together
- $afandumbi$: to attack each other, to fight each other
- $afanambi$: to go to attack, to go to fight

---

• **afanjimbi**: to come to attack, to come to fight

As we shall see, the agglutinative nature of Manchu leads to certain advantages and disadvantages in terms of character- and word-level recognition.

### 2.2 Background and Related Work

Optical Character Recognition refers to the process of digitizing printed or handwritten text by computers. The development of OCR technology began in the 1970s and has continued to progress with advances in machine learning and computer vision, becoming more reliable and accurate when used in everyday tasks such as document scanning, machine translation, and data entry.

Historically, learning-based approaches to the OCR task have been divided into three types: character-level recognition, word-level recognition, and sequence recognition. The character-level approach starts by recognizing individual letters and then grouping them into words, whereas the word-level approach attempts to recognize whole words at a time, treating it as a classification task. In contrast, the third approach, which was introduced more recently in 2015 by Shi et al. [6], aims to use a sequence-to-sequence (seq2seq) model to approach OCR, using a novel architecture that combines convolutional and recurrent neural layers to label sequences of arbitrary lengths.

Due to its status as a low-resource language, and because Manchu today is used virtually entirely in an academic setting, the commercial desire to develop OCR for Manchu is limited. Nevertheless, in recent years work has been done to establish baseline results for recognizing Manchu text in both the historical and non-historical settings, which is where this survey begins.

---

6 A term used within the field of Natural Language Processing to refer to languages with very few data sets available to use for training. In contrast, English and Chinese are the two most high-resource languages.
### 2.2.1 Character-level Recognition

Unlike alphabetic scripts such as Latin or Cyrillic (which lend themselves towards the character-level approach), Manchu — a semi-syllabary — changes the appearance of its characters depending on where within a word they are placed (the head, middle, or tail). Nevertheless, some of the first works in the field attempted to follow this approach; in 2006, Zhao et. al published “Design and implementation of off-line handwritten document recognition system of Manchu manuscript” [7], in which they used digital image processing techniques to extract handwritten Manchu words from documents, decomposed each word into individual stroke units (note: these do not correspond to letters, nor do they represent characters when considered individually), and lastly converted the stroke sequences into romanized words, using a Hidden Markov Model (HMM) in post-processing to improve the accuracy of their model to 92.3%.

![Pre-processing and stroke partitioning](image)

**Figure 2.1:** Pre-processing and stroke partitioning, from Zhao et al. [7].

In the same year, another group of researchers, Zhang et. al [8], published a different method, in which they instead partitioned each word into Manchu characters (such as gi, su, and re) and aimed to recognize each character before assembling the predictions into words. However, this method was able to achieve an accuracy of at most 88.4%, partially due to the difficulty in determining where to segmentation boundaries of characters, and also because each character may have multiple representations (for instance, in the example below, the
character \( re \) changes appearance based on whether it is placed in the middle or the tail of the word).

![Character-level segmentation, from Zhang et. al [8].](image)

**Figure 2.2:** Character-level segmentation, from Zhang et. al [8].

Efforts to improve upon the character-level approach were continued in [9] and [10], but errors in segmentation accuracy prevented character-level recognition from progressing to a truly meaningful level.

### 2.2.2 Word-level Recognition

In 2017, following the success of leveraging deep convolutional neural networks (CNNs) to recognize Chinese characters via image classification across thousands of classes, a group of researchers at Dalian Minzu University\(^7\) 大连民族大学 proposed a segmentation-free approach to recognizing Manchu text at the word level [11]. In their paper, they used a Directed Acyclic Graph Support Vector Machine (DAG-SVM) classifier to classify images of text into 100 distinct Manchu words with 78\% accuracy. Since labelled training data for Manchu is scarce, especially word-level data, they used a variety of data augmentation methods — in-

\(^7\)Dalian is located in Liaoning province, one of the three provinces of northeast China making up the former Manchu homeland.
cluding dilation, erosion, elastic deformation, and affine transformation — to increase the size of their initial data set, which consisted of a set of 100 printed Manchu words.

![Figure 2.3: Four basic transformations.](image)

In [12], Zheng et al. proposed another segmentation-free recognition method, which achieved 91% accuracy for printed text and 64% accuracy for handwritten text using a 9-layer CNN trained on a synthetic data set consisting of 12 fonts. In [13], Li et al. further improved the accuracy of the CNN-based model to 96.31% by replacing the final maximum pooling layer of the convolutional network with a spatial pyramid pooling layer, thereby removing the need for the input images to be the same size as required by a traditional neural network. However, in practice, the authors noted that the further the distance between the size of the training and test images, the greater a decrease in accuracy was observed.

Most recently, in 2021, Zhang et al. [14] proposed a hybridized approach in which the deep CNN model is used to recognize words within an input image using a sliding window technique to continuously partition words into segments, achieving an accuracy of 98.84% across a dictionary of 671 word classes of a single font.
2.2.3 Seq2seq Approach

Lastly, while a sequence-based approach has not yet been attempted for Manchu, Shi et. al [6] showed that the convolutional recurrent neural network (CRNN) architecture has a large potential to improve upon the OCR task for sequences of text, such as CAPTCHA images and Chinese sentences.

2.3 Data Collection

As mentioned previously, due to its status as an endangered language, there are few (if any) data sets for written Manchu, in the sense that no one has compiled a large amount of images of Manchu words along with their transliterations into a database of labelled images such as MNIST, which contains 70,000 images of handwritten digits from 0 to 9, or CIFAR-100, which has 60,000 images split into 100 categories like tiger, tank, telephone, and so on. However, in order to produce meaningful results in any deep-learning based approach, it is important to obtain a data set of sufficient size. Thus, following the precedent of previous papers published, we will use a series of random image augmentation techniques to increase the size of our data set that will be used for training. Additionally, we use three strategies

---

8The MNIST and CIFAR-10/100 data sets are commonly used to train image classification models.
to create our data set: synthetic word generation, data augmentation, and transliteration alignment.

1. **Synthetic word generation**: rather than relying on existing examples of Manchu words, we can instead generate images of words ourselves using an array of digital fonts, and use these images as a base upon which we apply transformations. Additionally, this allows us to customize the size and background of the training images to most accurately represent real data from historical documents.

2. **Data augmentation**: similar to previous work, by applying a series of transformations to the original training set, we can increase the amount of training data we have, as well as make our model more robust (i.e. less prone to over-fitting).

3. **Transliteration alignment**: historians who translate important Manchu documents into other languages (such as Korean or Japanese) oftentimes include a romanized transliteration of the original work as a byproduct of the translation task. By using image processing techniques to split the original Manchu document into a sequence of word images and aligning these with the romanized transliteration, we can “label” the words of the original document with reasonable accuracy.

In addition, one limitation of convolutional neural networks is that the number of classes is fixed, meaning that any new words that are not in the training dictionary will be unrecognizable. To circumvent this, we can use semi-supervised learning techniques to learn from a combination of unlabelled and labelled training images, which is particularly useful since much of the potential data sources for Manchu will come from historical documents.

### 2.3.1 Synthetic Word Generation

The process of synthetically generating images of Manchu words is composed of the following steps: first, for each word, we use a Python script to generate and crop images of the word in eleven different fonts, and second, we take the images from the first step and apply a number of transformations to each image to increase the size of the data set.
The set of Unicode fonts that are used to generate the images were all made available on
the Manchu Studies Group website, thanks to the hard work of Benjamin Yang of Panlex.¹
There are 11 fonts used in the generation process:

1. XM BiaoHei 锡满标黑
2. XM GuFeng 锡满古风
3. XM LiuYe 锡满柳叶
4. XM ShuKai 锡满书楷
5. XM WenJian 锡满文鉴
6. XM WenQin 锡满文勤
7. XM XingShu 锡满行书
8. XM YaBai 锡满雅白
9. XM YingBi 锡满硬笔
10. XM ZhengBai 锡满正白
11. XM ZhengHei 锡满正黑

Note: the prefix XM denotes xī mǎn 锡满, referring to the Xibe and Manchu scripts.

We use the following script to generate images in the first step:

```python
import arabic_reshaper
import pandas as pd
from pathlib import Path
from PIL import Image, ImageDraw, ImageFont

# Read words from file
words_df = pd.read_csv("data/words.csv")

# Generate images
width = 224
height = 224
padding = 10
base_img = Image.new("RGB", (width, height), color="white")
base_draw = ImageDraw.Draw(base_img)

fonts = ["XM_BiaoHei.ttf", "XM_GuFeng.ttf", "XM_LiuYe.ttf", "XM_ShuKai.ttf",
        "XM_WenJian.ttf", "XM_WenQin.ttf", "XM_XingShu.ttf", "XM_YaBai.ttf",
        "XM_YingBi.ttf", "XM_ZhengBai.ttf", "XM_ZhengHei.ttf"]

for i in range(len(words_df)):
```

¹See [https://www.manchustudiesgroup.org/typing-manchu/](https://www.manchustudiesgroup.org/typing-manchu/).
if (i + 1) % 100 == 0:
    print(f"Generated {i+1} words...")

romanized, word = words_df.iloc[i]

for font_num, font_name in enumerate(fonts):
    font = ImageFont.truetype(f"font/{font_name}", size=100)

    reshaped = arabic_reshaper.reshape(word)

    left, top, right, bottom = base_draw.textbbox((0, 0), reshaped,
                                                font=font, anchor="lt", language="ar-SA")
    text_width = right - left
    text_height = bottom - top

    img = Image.new("RGB", (text_width + 2 * padding, height), color="white")
    img_draw = ImageDraw.Draw(img)
    img_draw.text((padding, height / 2), reshaped, fill=(0, 0, 0), font=font, 
                     anchor="lm", language="ar-SA")

    output_path = Path(f"images/{romanized}/romanized_{font_name[:-4]}.png")
    output_path.parent.mkdir(exist_ok=True, parents=True)
    img.save(output_path)

Figure 2.5: image_generation.py

The Python Pillow library (forked from the Python Imaging Library) provides all of
the image manipulation functionality. In addition, we set the language of the textbox to be
Arabic in order to induce the correct displaying of font ligatures, which are used in both the
Arabic and Manchu scripts. A set of generated images for the word yodan (raincoat) are
shown below:

Figure 2.6: The word yodan in eleven fonts.

Many thanks to Jack Rabinovitch for suggesting this idea!
These serve as the base images for our data set, upon which we will apply random transformations to increase the data set size. We also perform the following modifications to the base images in order to make the data set more robust: first, we resize each image to be 224 by 224 pixels, then convert each image to grayscale and normalize the image such that the mean and standard deviation are both 0.5 across all pixels. After this process, we apply a series of random transformations, which slightly alter the base image to produce a random new image that is noticeably different than the original. These transformations include a combination of perspective skewing, randomized resizing and cropping, and grid distortion.

Lastly, in order to generate the base images, we need to first have a list of words to be generated. For Manchu, this poses the problem that the number of words formed through agglutination is extremely large compared to non-agglutinative languages such as Chinese or English; some agglutinative languages, for instance Korean and Finnish, contain upwards of eight hundred thousand words. In the case of Manchu, we can get an approximate sense of the scope of the language based on the following sources:

- Norman’s *Comprehensive Manchu-English Dictionary*, which is the definitive lexicon for modern-day Manchu studies, contains roughly 26,482 entries.\(^{11}\)

- On the English version of Wiktionary, there are 2,155 articles belonging to Manchu.\(^ {12}\) Of these, after filtering out certain categories (i.e. romanizations, punctuation, etc.), there are 1,593 distinct articles corresponding to Manchu words, which may be considered an approximate list of “common” Manchu words.

- Scraping the text of Wp/mnc, the test version of the Manchu Wikipedia, yields 4,577 articles and roughly 30,000 paragraphs of text. There are 20,451 unique words spread across all articles. The top 100 most frequent words account for 44.61% of all words, the top 500 words account for 62.64%, and the top 3,000 words account for 80.97% of all

---

\(^{11}\)This number comes from parsing the PDF of Norman’s *Dictionary* and counting the number of occurrences of bolded terms followed immediately by a colon and space.  
\(^{12}\)As of March 17, 2023, according to data cached by https://kaikki.org using Wikitext [15].
words. The frequencies of all words that occur more than a hundred times are shown below:

![Figure 2.7: Word frequencies of Wp/mnc.](image)

From these estimates, we propose that compiling a list of roughly 3,000 words to use for training our baseline model will be sufficient to read most texts to a reasonable degree of accuracy, with the words coming from Wiktionary as the initial set and the remainder coming from the most common words on Wp/mnc.

### 2.4 Training and Model Architecture

For the actual model, we begin by training a convolutional neural network from scratch to recognize the different words in our data set. In recent years, the field of computer vision and deep learning has seen major developments ever since 2012, when Krizhevsky et al. published “ImageNet Classification with Deep Convolutional Networks,” introducing a CNN architecture known as AlexNet [16] that performed extremely well on the ImageNet Challenge\(^\text{13}\) for its time, achieving a top-5 error rate of 15.4%, which beat the next best performing model by a margin of more than 10 percent.

AlexNet’s architecture consisted of five convolutional layers, three max pooling layers, and three fully-connected layers, the output of which was fed into a 1,000-way softmax function to produce a distribution of probabilities for each of the 1,000 class labels in the original paper. AlexNet also popularized using the ReLU function (the rectified linear unit activation function).

\(^{13}\text{ImageNet is an extremely large database of over 14 million images belonging to 20,000 categories; every year since 2010, the project holds the ImageNet Large Scale Visual Recognition Challenge (a historically very difficult contest) to determine which program can correctly classify the most images.}\)
function, i.e. \( f(x) = \max(0, x) \)) rather than conventional \( \tanh(x) \) activation function, which led to significant speed increases. Though this architecture might seem somewhat simple in the world of deep learning today, in 2012, it was seen as one of the first truly “deep” machine learning models that proved to be effective, and it also demonstrated the advantages of leveraging multiple GPUs (that is, performing calculations in parallel on specialized Graphical Processing Units) to decrease training times by a factor of 3 or 4.

### 2.4.1 Implementation

Nowadays, with resources such as Google Colab allowing one to write code and train models on the cloud while having access to state-of-the-art GPUs, it is relatively straightforward to devise a model based on AlexNet and train it from scratch for the task of recognizing Manchu words. We use the popular Python machine learning library PyTorch to train our model; the figure below contains the implementation of the model (based on the architecture laid out in the original AlexNet paper), with the minor change that because our training images are grayscale text, our first convolutional layer receives only a single color channel as input rather than the standard 3 channels of RGB images.

```python
class CNN(nn.Module):
    def __init__(self, num_classes=3000, dropout=0.5):
        super(CNN, self).__init__()

        self.features = nn.Sequential(
            nn.Conv2d(1, 64, kernel_size=11, stride=4, padding=2),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=3, stride=2),
            nn.Conv2d(64, 192, kernel_size=5, padding=2),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=3, stride=2),
            nn.Conv2d(192, 384, kernel_size=3, padding=1),
            nn.ReLU(inplace=True),
            nn.Conv2d(384, 256, kernel_size=3, padding=1),
            nn.ReLU(inplace=True),
            nn.Conv2d(256, 256, kernel_size=3, padding=1),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=3, stride=2),
        )

        self.avgpool = nn.AdaptiveAvgPool2d((6, 6))
```
self.classifier = nn.Sequential(
    nn.Dropout(p=dropout),
    nn.Linear(256 * 6 * 6, 4096),
    nn.ReLU(inplace=True),
    nn.Dropout(p=dropout),
    nn.Linear(4096, 4096),
    nn.ReLU(inplace=True),
    nn.Linear(4096, num_classes),
)

def forward(self, x):
    x = self.features(x)
    x = self.avgpool(x)
    x = torch.flatten(x, 1)
    x = self.classifier(x)
    return x

Figure 2.8: Model Architecture

2.4.2 Training Details

Initially, because training from scratch is computationally expensive, we chose to train the model on a subset of 300 words to determine if model is able to learn the fine details of the Manchu alphabet without wasting a large amount of time which is required for training a larger model with 3,000 classes. We trained the model using stochastic gradient descent with a batch size of 64, momentum of 0.9, and weight decay of 0.0005 (following the hyperparameters used in the original AlexNet paper). We used a step learning rate function which decreased the learning rate by a factor of 0.1 every 10 epochs, with an initial learning rate of 0.005, and trained the model for 50 epochs before terminating. We ran the training on Google Colab, using their standard GPU class on the backend.

As we shall see in the following section, although it is possible to successfully train a model from scratch with high accuracy, there are other, less computationally-expensive approaches that may be better in the long run, namely finetuning pre-trained models.
2.5 Finetuning

As an alternative to training from scratch, as well as to offer a point of comparison, we also experiment with using pre-trained models and finetuning them on our data set of Manchu words, which can be done efficiently and without much effort. The benefits of finetuning pre-trained models, which have already been trained on a large corpus of images such as ImageNet, over training a model from scratch are considerable.

First, the burden of assembling a large data set is comparatively alleviated by the large-scale data sets that pre-trained models are usually exposed to, regardless of whether the domains of the pre-training data set and finetuning data set align or not; this is because for any image recognition task, be it text recognition or identifying classes of objects, the primitive knowledge of recognizing basic shapes and curves must be learned regardless before higher level details can be refined.

Second, the time and number of iterations — not to mention computing resources — required to finetune a pre-trained model are much, much less than the time it takes for a model to learn to recognize the data from scratch; this is because downloading an already-pretrained model allows us to skip over the step of the initial training on large-scale data sets of millions of images, which can take anywhere from days to even weeks.

Thus, we elect to use the following pre-trained models:

- **VGG-16 [17]**: VGG Net (2014) uses small $3 \times 3$ filters (compared to $11 \times 11$ in the first layer of AlexNet) for each of its convolutions, but increases the depth of the model to 19 convolutional layers.

- **GoogLeNet [18]**: GoogLeNet (2014) won the 2014 ImageNet Challenge with a top-5 error rate of 6.7% and consists of many layers arranged into Inception modules operating side-by-side.

- **Inception v3 [19]**: Inception v3 (2015) is the third iteration of the Inception-based architecture pioneered by GoogLeNet, and improves upon the previous versions considerably.
• **ResNet-18 and ResNet-50** [20]: ResNet (2015) from Microsoft Research Asia is an extremely deep neural network with up to 152 layers. It won the 2015 ImageNet challenge with a top-5 error rate of 3.6% (beating human performance for the first time).

• **DenseNet-121** [21]: In DenseNet (2017), there are “dense” blocks consisting of multiple layers wherein each layer receives inputs not only from the layer directly preceding it, but all layers before it in the block, leading to the concept of a “dense” network where each layer is connected to all others. This also leads fewer parameters being trained, reducing the amount of computational complexity.

• **ViT-B/16** [22]: Unlike all the above models, Vision Transformer (2020) implements a Transformer-based architecture, rather than a CNN-based one. Traditionally used in NLP research for tasks such as language modeling (e.g. BERT [23], GPT-3 [24]), Transformers [25] take as input sequences of data and use the self-attention mechanism (in lieu of convolutional or recurrent networks) during training, leading to state-of-the-art results with much less computational overhead. ViT enables the Transformer architecture to be used for images by splitting each input into $16 \times 16$ patches aligned into a sequence and feeding it to a Transformer encoder, and has shown to match and even beat the accuracy of traditional CNN-based image classification models on a variety of data sets.

### 2.5.1 Finetuning Details

For each of the models mentioned, we downloaded the pre-trained versions using PyTorch’s built-in `torchvision.models` subpackage with the default weights, and finetuned the models using stochastic gradient descent with a batch size of 64 and momentum of 0.9. For each model, the initial learning rate was set to 0.01, and followed a variable schedule decreasing by a factor of 0.1 every 10 epochs. Each model was trained for 30 epochs before terminating, and the iteration with the highest validation accuracy was saved.

All training was done on Google Colab, using their standard GPU runtime (typically
NVIDIA T4 Tensor Core GPUs\textsuperscript{14} and occasionally switching to premium GPUs (typically NVIDIA V100 or A100 Tensor Core GPUs) to decrease training time.

2.6 Results

After training, each model (including the baseline model trained from scratch and each of the pre-trained models) was evaluated on a test set of 6,000 images,\textsuperscript{15} and the accuracies are reported below.

<table>
<thead>
<tr>
<th>Model Name</th>
<th>Accuracy</th>
</tr>
</thead>
<tbody>
<tr>
<td>Baseline (from scratch)</td>
<td>96.93%</td>
</tr>
<tr>
<td>VGG-16</td>
<td>99.01%</td>
</tr>
<tr>
<td>Googlenet</td>
<td>99.14%</td>
</tr>
<tr>
<td>ResNet-18</td>
<td>99.24%</td>
</tr>
<tr>
<td>ResNet-50</td>
<td>98.89%</td>
</tr>
<tr>
<td>Inception v3</td>
<td>99.08%</td>
</tr>
<tr>
<td>DenseNet-121</td>
<td>98.95%</td>
</tr>
<tr>
<td>ViT-B/16</td>
<td>97.72%</td>
</tr>
</tbody>
</table>

Table 2.1: List of Model Accuracies

It is important to note that these results are merely a rough \emph{approximation} of what the model’s performance will be when used to read historical documents, since the test data used here is also created using the same image generation methods as mentioned in the previous section. Nevertheless, for many documents written by court scribes, we expect that the handwritten words will closely resemble that of printed text, so the accuracies reported here do matter insofar as a higher accuracy on the test data set will imply a better performance on real-world data.

As indicated by the table, ResNet-18 slightly outperforms the rest of the models, although with the exception of ViT and the baseline model, the differences in accuracy are not large in comparison to the other models. Thus, we can leverage a model cascade\textsuperscript{16} to combine the

\textsuperscript{14}According to TensorFlow Blog.
\textsuperscript{15}In the case of the baseline model, since it was only trained on 300 word classes, the test size was 600 rather than 6,000.
\textsuperscript{16}As discussed in [26], an \emph{ensemble} uses multiple models in parallel and averages their predictions, whereas a
results of the individual models into a committee-based model that is more robust and may lead to higher accuracy without any additional computational complexity.

Figure 2.9: Some sample predictions of the ResNet-18 model.
Chapter 3

Into The Archives

We begin this chapter with a survey of the various Manchu archival materials located in both the East and West, and then proceed to try our hand at reading one of these documents.

3.1 A Survey of Manchu Documents

The breadth and utility of Manchu language sources have already been illustrated to great effect in works such as The Manchu-Language Archives of the Qing Dynasty by Elliott, Manchu Archival Materials by Chieh-hsien Chen, and A Profile of the Manchu Language in Ch‘ing History, by Crossley and Rawski. Here, we briefly summarize the main points surrounding the use of Manchu language documents.

Geographically, the majority of historical Manchu documents lie in storage in the People’s Republic of China, namely within the First Historical Archives in Beijing but also at the Third Historical Archives at Shenyang and in other collections in the northeast such as Dalian and Harbin. A significant archive of Qing documents also exists at the National Palace Museum of Taiwan and the National Central Archives of Mongolia. In addition, other institutions in the West, including in Germany, England, and the United States, also house sizeable collections; the Harvard-Yenching Library, which is home to the Manchu Rare Books Collection (comprising of 67 fully digitized texts, and many more non-digitized works) is one, as is the Library of Congress, which has recently embarked on a digitization project for
over 400 Manchu texts within its greater Chinese Rare Book Collection.¹⁷

Perhaps the most influential Manchu language work is the *Jiu Manzhou Dang* 旧满洲档 (Early Manchu Archives), which also happens to be the oldest existing work written in Manchu, detailing the history of the period from 1607 to 1636. Split into 40 volumes — 20 for Nurhaci’s reign, and 20 for Hong Taiji — the *Jiu Manzhou Dang* offers a definitive account of Manchu history pre-conquest, especially in comparison to the scant Chinese language records for the same period. Following suit is the *Manzhou Shilu* 满洲实录 (Manchu Veritable Records), which features a trilingual gloss in Manchu, Mongolian, and Chinese, and which later became part of the greater *Qing Veritable Records* 大清历朝实录, the official record of the Qing dynasty.

One example of Manchu documents that cannot go unmentioned are the secret palace memorials that first became widespread after the Kangxi Emperor’s reform of the memorial system and continued to be exceedingly popular well into the Qianlong era. These documents, written in Manchu, contained all sorts of intelligence from officials across the empire, especially regarding local matters which were rarely mentioned in official histories disseminated by the imperial court.

Nevertheless, the purpose of our work, as is the case with most Qing historians, is not to repeatedly analyze those materials which have already been published, translated, and analyzed repeatedly; rather, our focus is on those documents hiding outside the limelight, which have never properly been studied or explored. These documents, by nature, are hard to get ahold of — one must either travel in person to the First Historical Archives, or hope that these materials are released to the public one day. Thus, for the time being, we must content ourselves with what we already have.

### 3.2 A Sample Reading

On the following page, we show an example of what using our model to read a sample image would look like.

¹⁷See *Exploring Rare Manchu Books at the Library of Congress*.
Figure 3.1: A sample image of a Manchu book from Yenching Library.
Figure 3.2: Extracting Manchu words from the image.
Figure 3.3: Predicting words.
Chapter 4

Conclusion

In this thesis, we endeavored to create a text recognition model to recognize Manchu words, and achieved good results on our test data set as well as on certain historical documents.

However, much work remains to be done before Manchu text recognition software can be used in the field to read historical documents. First, there is the issue of a lack of training data that can be used to train the model, leading to the use of synthetic data sets, which can provide a reasonable baseline for training but often presents difficulties when encountering samples that are different from the training set, such as handwritten Manchu words written imprecisely. Second, the CNN-based model has the limitation of only being able to recognize words within its vocabulary, whereas Manchu, an agglutinative language, consists of many thousands of words (not all of which may be present in the training data set). Lastly, there is the work of exploring other architectures — in particular, sequence-to-sequence-based CRNNs — which may be more suited to the Manchu recognition task.
Appendix

Appendix A: Data Collection

```python
# Data Collection
This notebook consists of the following sections:
1. Scraping a list of Manchu words from Wiktionary
2. Scraping text from Manchu Wikipedia (which is an unpublished test wikipedia)
3. Aggregating statistics to determine the most common words
4. Generating images of 11 different fonts for each image

from google.colab import drive
drive.mount("/content/drive")

## Scraping Wiktionary

*Note: this data is cached by Kaikki.org and is credited to Tatu Ylonen:
Wiktextract: Wiktionary as Machine-Readable Structured Data, Proceedings of
1317-1325, Marseille, 20-25 June 2022.*

import json
!wget https://kaikki.org/dictionary/Manchu/words/kaikki.org-dictionary-Manchu-words.json
lst = []
with open("kaikki.org-dictionary-Manchu-words.json", "r", encoding="utf-8") as f:
    for line in f:
        data = json.loads(line)
        lst.append(data)
```
pos_types = set(data["pos"] for data in lst)
print("Grammar types: ", pos_types)

# Filter out words we don't want
pos_types.remove("character")
pos_types.remove("classifier")
pos_types.remove("punct")
pos_types.remove("romanization")
pos_types.remove("suffix")

# Print some examples
print("Examples of conjunctions in Manchu:")
for idx, data in enumerate(lst):
    if data["pos"] == "conj":
        print(data)

# Compile the romanization to Manchu dictionary
words_dict = {}
for data in lst:
    if data["pos"] in pos_types:
        romanization = None
        if "forms" in data:
            if data["forms"][0]["tags"][0] == "romanization":
                romanization = data["forms"][0]["form"]
            if romanization == "-be":
                continue
        else:
            # goro mama, unggu mama, suru, genggiyen, gala huru
            romanization = data["head_templates"][0]["expansion"].split("(")[1][:-1]
            if romanization == "gegiyen":
                romanization = "genggiyen"
        if romanization:
            if " " in romanization:
                # Split multi-word entries into individual words
                for roman, manchu in zip(romanization.split(), data["word"].split()):
                    roman = Romanizer.lower(roman)
                    if roman not in words_dict:
                        words_dict[roman] = manchu
            else:
romanization = romanization.lower()
    if romanization not in words_dict:
        words_dict[romanization] = data["word"]

# Preview the first 10 words
print(f"Total words: {len(words_dict)}")
for romanization, word in sorted(words_dict.items())[:10]:
    print(romanization, word)

import pandas as pd

# Save words_dict to Drive
words_df = pd.DataFrame(sorted(words_dict.items()), columns=["romanization", "manchu"])
words_df.to_csv("/content/drive/MyDrive/Thesis/data/words.csv", index=False)

# Also save a copy of original data
!cp kaikki.org-dictionary-Manchu-words.json drive/MyDrive/Thesis/data/original/kaikki.org-dictionary-Manchu-words.json

## Scraping Wikipedia

In order to get a working corpus of Manchu text, we scrape [Wp/mnc](https://incubator.wikimedia.org/wiki/Wp/mnc) using the [WikiExtractor.py](https://github.com/apertium/WikiExtractor/blob/master/WikiExtractor.py) script. A mirror of incubator wiki dump files, including the Manchu Wikipedia, can be found [here](https://mirror.accum.se/mirror/wikimedia.org/dumps/incubatorwiki/20221201/).

!wget https://mirror.accum.se/mirror/wikimedia.org/dumps/incubatorwiki/20230301/incubatorwiki-20230301-pages-articles-multistream.xml.bz2
!wget https://raw.githubusercontent.com/apertium/WikiExtractor/master/WikiExtractor.py

# Use the script to extract text from the dump
!python3 WikiExtractor.py --incubator mnc --infn incubatorwiki-20230301-pages-articles-multistream.xml.xml.bz2 > wp_mnc_output.log

# Save the output text and log to Drive
!cp wiki.txt drive/MyDrive/Thesis/data/wiki.txt
!cp wp_mnc_output.log drive/MyDrive/Thesis/data/original/wp_mnc_output.log

# Also save a copy of original data
## Word Frequencies

Using the text scraped from above, we can count the number of occurrences of each word in our dictionary to see which words are the most commonly used.

```python
from collections import defaultdict
import string

# Normally, we would initialize the word counts from words_df:
# words_df = pd.read_csv("/content/drive/MyDrive/Thesis/data/words.csv")
# words_list = words_df["romanization"].values.tolist()
# word_counts = dict.fromkeys(words_list, 0)

# For now, just use a defaultdict
word_counts = defaultdict(int)

with open("/content/drive/MyDrive/Thesis/data/wiki.txt") as f:
    lines = f.readlines()
    for line in lines:
        words = line.split()
        for word in words:
            word = word.translate(str.maketrans("", "", string.punctuation)).lower()
            word_counts[word] += 1

word_counts_df = pd.DataFrame(word_counts.items(), columns=["Word", "Count"])
most_frequent_words = word_counts_df.sort_values(by=["Count"], ascending=False)

print(f"Total number of unique words: {len(word_counts_df)}")
print("Most frequent words:")
most_frequent_words.head()

x = sum(most_frequent_words.iloc[:500]["Count"]) / sum(most_frequent_words["Count"])
print(f"The top 500 most frequent words represent {100*x:.1f}% of all words")

import matplotlib.pyplot as plt

plt.rcParams["figure.figsize"] = [20, 5]
plt.rc("ytick", labelsize=20)

x = most_frequent_words[most_frequent_words["Count"] > 100]
plt.bar(x["Word"], x["Count"], width=0.75)
plt.xticks([])
```

35
# Save words_dict to Drive

```python
most_frequent_words.to_csv("/content/drive/MyDrive/Thesis/data/original/most_frequent_words.csv",
index=False)
```

```python
## Assemble List of Words

```python
words_df = pd.read_csv("/content/drive/MyDrive/Thesis/data/images/words.csv",
na_filter=False)
most_frequent_words = pd.read_csv("/content/drive/MyDrive/Thesis/data/original/most_frequent_words.csv",
na_filter=False)

mollendorf_alphabet = set("abcdefghijklmnoprstuwyzš¯už ")
cleaned_words = []
for word in most_frequent_words["Word"].iloc[:3000].values:
    add = True
    for letter in word:
        if letter not in mollendorf_alphabet:
            add = False
            break
    if add:
        cleaned_words.append(word)

a = set(cleaned_words)
b = set(words_df["romanization"])
word_list = a.union(b)
word_list.remove("")

!rm -rf /content/drive/MyDrive/Thesis/data/images/words_large.csv
```

```python
with open("/content/drive/MyDrive/Thesis/data/images/words_large.csv", "w") as f:
    for word in sorted(word_list):
        f.write(f"{word}\n")
```

```python
## Image Generation

In this section, we procedurally generate images for each word using 11 different Manchu fonts (created by [Benjamin Yang](https://www.manchustudiesgroup.org/typing-manchu/) of the Manchu Studies Group).

```
!pip install arabic_reshaper
!pip install --upgrade Pillow
import arabic_reshaper
import pandas as pd
from pathlib import Path
from PIL import Image, ImageDraw, ImageFont
```

36
# Font names
fonts = [
    "XM_BiaoHei.ttf", "XM_GuFeng.ttf", "XM_LiuYe.ttf", "XM_Shukai.ttf",
    "XM_WenJian.ttf", "XM_WenQin.ttf", "XM_XingShu.ttf", "XM_YaBai.ttf",
    "XM_YingBi.ttf", "XM_ZhengBai.ttf", "XM_ZhengHei.ttf"
]

# Read words from file
words_df = pd.read_csv("/content/drive/MyDrive/Thesis/data/images/words_new.csv",
                      na_filter=False)

# Generate images
width = 224
height = 224
padding = 10
base_img = Image.new("RGB", (width, height), color="white")
base_draw = ImageDraw.Draw(base_img)

for i in range(len(words_df)):
  if (i + 1) % 100 == 0:
    print(f"Generated {i+1} words...")

  romanization, word = words_df.iloc[i]

  for font_num, font_name in enumerate(fonts):
    font_path = "/content/drive/MyDrive/Thesis/data/fonts/" + font_name
    font = ImageFont.truetype(font_path, size=100)

    # Thanks to Jack for suggesting to "pretend" to be Arabic in order to make
    # the ligatures render properly
    reshaped = arabic_reshaper.reshape(word)

    left, top, right, bottom = base_draw.textbbox((0, 0), reshaped, font=font,
                                               anchor="lt", language="ar-SA")
    text_width = right - left
    text_height = bottom - top

    img = Image.new("RGB", (text_width + 2 * padding, height), color="white")
    img_draw = ImageDraw.Draw(img)
    img_draw.text(((padding, height / 2), reshaped, fill=(0, 0, 0), font=font,
                   anchor="lm", language="ar-SA")

    output_path = 
                   Path(f"images/images_new/{romanization}/{romanization}_{font_name[:-4]}.png")
Appendix B: Model Training From Scratch

"""model.ipynb

# Manchu Classifier
A CNN classifier for Manchu words.
"""

import torch
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

from google.colab import drive
drive.mount("/content/drive")
!rm -rf sample_data

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

"""## Load the data
First, let's take a look at the list of words that we're working with:
"""
words_df = pd.read_csv("/content/drive/MyDrive/Thesis/data/words.csv")
# words_df.head(10)

# Some helper functions to print out word representations for classes

def get_roman(word_idx):
    return words_df.iloc[word_idx, 0]

def get_word(word_idx):
    return words_df.iloc[word_idx, 1]

"""Next, we have to load in the images from Drive into the local storage for faster reads:"""

!unzip -q /content/drive/MyDrive/Thesis/data/images/images_small.zip

"""### Create ImageFolder Dataset and DataLoader

We can create custom train and test datasets for our image data (11 images per word * 1622 words) using torchvision:

"""

from torch.utils.data import DataLoader, random_split
from torchvision import datasets, transforms

transform = transforms.Compose([  
    transforms.Grayscale(num_output_channels=1),  
    transforms.Resize((224, 224)),  
    transforms.ToTensor(),  
    transforms.Normalize(mean=(0.5,), std=(0.5,)),  
    transforms.RandomPerspective(  
        distortion_scale=0.1,  
        p=0.9,  
        fill=1,  
        interpolation=transforms.InterpolationMode.NEAREST
    )
])

# Create train/val/test dataset
dataset = datasets.ImageFolder(root="/content/images/images_small",  
    transform=transform)  
test_size = len(dataset) // 11  
train_dataset, val_dataset, test_dataset = random_split(dataset, [9 * test_size,  
    test_size, test_size])

39
batch_size = 64
train_loader = DataLoader(train_dataset, batch_size=batch_size, shuffle=True,
                                      num_workers=2, pin_memory=True)
val_loader = DataLoader(val_dataset, batch_size=batch_size, shuffle=True,
                                      num_workers=2, pin_memory=True)
test_loader = DataLoader(test_dataset, batch_size=batch_size, shuffle=True,
                                      num_workers=2, pin_memory=True)

# Add helper function to lookup class romanization and manchu
idx_to_class = {v: k for k, v in dataset.class_to_idx.items()}
def get_word_idx(class_idx):
    romanization = idx_to_class.get(class_idx)
    if romanization:
        return words_df.index[words_df["romanization"] == romanization].item()
    else:
        return "Error"

"""This is what a sample looks like after transformations for one of the words,
*ainu* (*dulimba* for the small model):""

plt.rcParams["figure.figsize"] = [10, 5]
fig = plt.figure()
for i in range(10):
    plt.subplot(2, 5, i+1)
    plt.imshow(dataset[32*i+1][0].squeeze().rot90(-1), cmap="gray")
    plt.title(f"Image {i+1}")
    plt.xticks([])
    plt.yticks([])

"""and here is an example batch:""
train_iter = iter(train_loader)
example_images, example_labels = next(train_iter)

plt.rcParams["figure.figsize"] = [4, 3]
for i in range(3):
    word_idx = get_word_idx(example_labels[i].item())
    print(f"Label: {get_word(word_idx)} ("get_roman(word_idx)")")
    plt.imshow(example_images[i].squeeze(), cmap="gray")
    plt.xticks([])
    plt.yticks([])
    plt.show()
### Model Training and Evaluation

For our CNN, we will use a popular image classification architecture called AlexNet.

```python
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim

class CNN(nn.Module):
    def __init__(self, num_classes=1000, dropout=0.5):
        super(CNN, self).__init__()

        self.features = nn.Sequential(
            nn.Conv2d(1, 64, kernel_size=11, stride=4, padding=2),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=3, stride=2),
            nn.Conv2d(64, 192, kernel_size=5, padding=2),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=3, stride=2),
            nn.Conv2d(192, 384, kernel_size=3, padding=1),
            nn.ReLU(inplace=True),
            nn.Conv2d(384, 256, kernel_size=3, padding=1),
            nn.ReLU(inplace=True),
            nn.Conv2d(256, 256, kernel_size=3, padding=1),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=3, stride=2),
        )

        self.avgpool = nn.AdaptiveAvgPool2d((6, 6))

        self.classifier = nn.Sequential(
            nn.Dropout(p=dropout),
            nn.Linear(256 * 6 * 6, 4096),
            nn.ReLU(inplace=True),
            nn.Dropout(p=dropout),
            nn.Linear(4096, 4096),
            nn.ReLU(inplace=True),
            nn.Linear(4096, num_classes),
        )

    def forward(self, x):
        x = self.features(x)
        x = self.avgpool(x)
        x = torch.flatten(x, 1)
```

---

41
x = self.classifier(x)
return x

num_classes = len(dataset.classes)
num_epochs = 100
learning_rate = 0.01

# Initialize the model with the hyperparameters above
model = CNN(num_classes).to(device)
criterion = nn.CrossEntropyLoss().to(device)
optimizer = optim.SGD(model.parameters(), lr=learning_rate, weight_decay=0.0005, momentum=0.9)
train_losses = []
best_acc = 0.0

"""Then we train the model on the training data:"""

from tqdm.notebook import tqdm

batches_per_epoch = len(train_loader)
curr_epoch = len(train_losses)
for epoch in tqdm(range(curr_epoch, curr_epoch + num_epochs)):
    running_loss = 0.0
    for batch_idx, (images, labels) in enumerate(train_loader):
        images = images.to(device)
        labels = labels.to(device)
        model.train()

        # Forward
        outputs = model(images)
        loss = criterion(outputs, labels)
        running_loss += loss.item()

        # Backward and optimize
        loss.backward()
        optimizer.step()
        optimizer.zero_grad()

    # Validation
    correct, total = 0, 0
    model.eval()
    for batch_idx, (images, labels) in enumerate(val_loader):
images = images.to(device)
labels = labels.to(device)

outputs = model(images)
loss = criterion(outputs, labels)

_, preds = torch.max(outputs.data, 1)
correct += (preds == labels).sum().item()
total += labels.size(0)

# Print train loss and validation accuracy
epoch_loss = running_loss / batches_per_epoch
accuracy = correct / total * 100
if accuracy > best_acc:
    best_acc = accuracy
    torch.save(model, "best_model.pt")
train_losses.append(epoch_loss)
print(f"Epoch {epoch+1}/num_epochs, loss: {epoch_loss:4f},
      validation accuracy: {accuracy:.2f}%")

# Plot the loss over time
plt.plot(train_losses)
plt.title("Loss vs. Epoch")
plt.show()

if True:
    best_model = torch.load("best_model.pt")

if True:
torch.save(model,
            "/content/drive/MyDrive/Thesis/data/models/small_baseline.pt")
torch.save(best_model,
            "/content/drive/MyDrive/Thesis/data/models/small_baseline_best.pt")

def eval_test(model):
correct = 0
total = 0
model.eval()
with torch.no_grad():
    for images, labels in test_loader:
        images = images.to(device)
        labels = labels.to(device)

        outputs = model(images)
        _, preds = torch.max(outputs.data, 1)

        correct += (preds == labels).sum().item()
total += labels.size(0)

        # Print test loss and accuracy
        test_loss = criterion(outputs, labels)
        accuracy = correct / total * 100
        print(f"Test set, loss: {test_loss:4f}, accuracy: {accuracy:.2f}%")

def show_image(image, label):
    plt.imshow(image.permute(1, 2, 0))
    plt.title(f"Class: {label.item()}")
    plt.show()
correct += (preds == labels).sum().item()
total += labels.size(0)

return f"{correct}/total - {correct / total * 100:.2f}"

print(f"Test accuracy: {eval_test(model)}")
print(f"Best accuracy: {eval_test(best_model)}")

Appendix C: Finetuning (ResNet-18)

# -*- coding: utf-8 -*-

"""resnet18.ipynb

# Finetuning

import torch
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from google.colab import drive

drive.mount("/content/drive")
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

"""## Load data""

!rm -rf sample_data
!unzip -q /content/drive/MyDrive/Thesis/data/images/images_new.zip

from torch.utils.data import DataLoader, random_split
from torchvision import datasets, transforms

transform = transforms.Compose([transforms.Grayscale(num_output_channels=3),
                                transforms.Resize((224, 224)),
                                transforms.ToTensor(),
                                transforms.Normalize(mean=(0.5,), std=(0.5,)),
                                transforms.RandomPerspective(distortion_scale=0.1, p=0.9, fill=1, interpolation=transforms.InterpolationMode.NEAREST)])
# Create train/val/test dataset

dataset = datasets.ImageFolder(root="/content/images/images_new",
                           transform=transform)

test_size = len(dataset) // 11
train_dataset, val_dataset, test_dataset = random_split(dataset, [9 * test_size,
                           test_size, test_size])

batch_size = 64

train_loader = DataLoader(train_dataset, batch_size=batch_size, shuffle=True,
                           num_workers=2, pin_memory=True)
val_loader = DataLoader(val_dataset, batch_size=batch_size, shuffle=True,
                        num_workers=2, pin_memory=True)
test_loader = DataLoader(test_dataset, batch_size=batch_size, shuffle=True,
                        num_workers=2, pin_memory=True)

# Helper functions

words_df = pd.read_csv("/content/drive/MyDrive/Thesis/data/images/words_new.csv",
                     na_filter=False)

idx_to_class = {v: k for k, v in dataset.class_to_idx.items()}

assert(set(dataset.classes) == set(words_df["romanization"])) # Sanity check

def get_roman(word_idx):
    return words_df.iloc[word_idx, 0]

def get_word(word_idx):
    return words_df.iloc[word_idx, 1]

def get_word_idx(class_idx):
    romanization = idx_to_class.get(class_idx)
    if romanization:
        return words_df.index[words_df["romanization"] == romanization].item()
    else:
        return "Error"

def imshow(image):
    image = image * 0.5 + 0.5 # unnormalize
    plt.imshow(image.permute(1, 2, 0).rot90(-1), cmap="gray")
    plt.xticks([])
    plt.yticks([])

plt.rcParams["figure.figsize"] = [10, 5]
fig = plt.figure()

for i in range(10):
    plt.subplot(2, 5, i+1)
    imshow(dataset[32*11+i][0])
    plt.title(f"Image {i+1}"")
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
from torchvision import models

num_classes = len(dataset.classes)
um_epochs = 100
learning_rate = 0.01

model_ft = models.resnet18(weights="DEFAULT")
num_ftrs = model_ft.fc.in_features
model_ft.fc = nn.Linear(num_ftrs, num_classes)
model_ft = model_ft.to(device)

criterion = nn.CrossEntropyLoss().to(device)
optimizer = optim.SGD(model_ft.parameters(), lr=learning_rate, momentum=0.9)
scheduler = optim.lr_scheduler.StepLR(optimizer, step_size=10, gamma=0.1)

train_losses = []
best_acc = 0.0

from tqdm.notebook import tqdm
curr_epoch = len(train_losses)
for epoch in tqdm(range(curr_epoch, curr_epoch + num_epochs)):
    running_loss = 0.0
    for batch_idx, (images, labels) in enumerate(train_loader):
        images = images.to(device)
        labels = labels.to(device)

        model_ft.train()

        # Forward
        outputs = model_ft(images)
        loss = criterion(outputs, labels)
        running_loss += loss.item()

        # Backward and optimize
        loss.backward()
        optimizer.step()
        optimizer.zero_grad()

    # Validation
    correct, total = 0, 0
model_ft.eval()

for batch_idx, (images, labels) in enumerate(val_loader):
    images = images.to(device)
    labels = labels.to(device)

    outputs = model_ft(images)
    loss = criterion(outputs, labels)

    _, preds = torch.max(outputs.data, 1)
    correct += (preds == labels).sum().item()
    total += labels.size(0)

    # Update learning rate
    scheduler.step()

    # Print train loss and validation accuracy
    epoch_loss = running_loss / len(train_loader)
    accuracy = correct / total * 100
    if accuracy > best_acc:
        best_acc = accuracy
        torch.save(model_ft, "best_model.pt")
    train_losses.append(epoch_loss)
    print(f"Epoch {epoch+1}/curr_epoch + num_epochs], loss: {epoch_loss:4f}, val acc: {accuracy:.2f}%, lr: {scheduler.get_last_lr()[0]:.3f}"

    # Plot the loss over time
    plt.plot(train_losses)
    plt.title("Loss vs. Epoch")
    plt.show()

    if True:
        best_model = torch.load("best_model.pt")

    if True:
        torch.save(model_ft, "/content/drive/MyDrive/Thesis/data/models/resnet18.pt")
        torch.save(best_model, "/content/drive/MyDrive/Thesis/data/models/resnet18_best.pt")

def eval_test(model):
    correct = 0
    total = 0
    model.eval()
    with torch.no_grad():
        for images, labels in test_loader:
            images = images.to(device)
            labels = labels.to(device)
outputs = model(images)
..., preds = torch.max(outputs.data, 1)

correct += (preds == labels).sum().item()
total += labels.size(0)

return f"{correct/total} - {correct / total * 100:.2f}%"

print(f"Test accuracy: {eval_test(model_ft)}")
print(f"Best accuracy: {eval_test(best_model)}")

test_iter = iter(test_loader)
example_images, example_labels = next(test_iter)

with torch.no_grad():
    outputs = model(example_images.to(device))

for i in range(3):
    pred_idx = get_word_idx(torch.max(outputs.data, 1)[1][i].item())
imshow(example_images[i].squeeze())
plt.xticks([])
plt.yticks([])
plt.show()
print(f"Prediction: {get_word(pred_idx)} {{get_roman(pred_idx)}}")
print()
Bibliography


