



UNIVERSITY OF BUCHAREST

FACULTY
OF
MATHEMATICS AND COMPUTER SCIENCE



**Recognizing and Translating
Ancient Egyptian Hieroglyphs**

*Dissertation Thesis Submitted
for the Degree of*

MASTER of DATA SCIENCE

by

Mihai Matei

under the guidance of

Prof. Marius Popescu

Bucharest, September 2021

Acknowledgements

I would like to thank my parents for providing their unconditional support throughout my master studies and thus being able to complete my computer science education by diving into this important and equally interesting field. Without them I would not have been able to finish top of the Class of 2021 in this Master of Data Science.

I would also want to give a big thank you to the professors that supported and defined this Master, to the ones teaching the classes and to any laboratory assistant I interacted. They managed to make this Master a modern one, having the same artificial intelligence and data science curriculum as any of the top universities in the world.

I would like to commend my dissertation advisor, Professor Marius Popescu, for his overall guidance on the structure of the project.

In the end I would like to thanks all the people who dedicated their time to move Ancient Hieroglyphs in our digital era, like the persons who created fonts, corpus of translations, websites and other resources containing the language.

Abstract

Ancient languages are part of the human heritage. Creating tools that help preserve, process and better understand this part of our written history is a task that should be considered by any computer or data science engineer.

This paper will present a short history of Ancient Hieroglyphs language research starting with the first discovery and translation and what has since been done.

Then I will dive into the current machine learning approaches used to recognize and translate the language. A LSTM based OCR engine was employed for the recognition while clustering, language direction heuristics and word identification was used to build a list of words from Unicode hieroglyph symbols. Lastly transformer encoder-decoder models and cnn classification ones were used to build a full sentence English translation.

An important part of the project was building the data sources for the OCR engine and for the translation models. About 3500 images were generated for training the OCR, a dictionary of 38000 Hieroglyph symbols to English words were used and 8000 small fully translated sentences were collected.

The work builds on my current project, GlyphViewer [1], a tool to develop translations using hieroglyphs and other languages.

Contents

1	Hieroglyph Language	7
1.1	Introduction	7
1.2	History of Egyptology	10
1.3	Hieroglyph Symbols	12
2	Recognition	14
2.1	Symbols Dataset Creation	14
2.2	Tesseract LSTM model	16
2.3	OCR results	18
3	Translation	21
3.1	Translation Dataset Creation	21
3.2	Translation to Words	24
3.3	Symbols to Words	25
3.4	Translation Models	25
3.4.1	Input Dataset	26
3.4.2	Transformer Model	27
3.4.3	CNN Model	29
4	Building a Product	33
4.1	GlyphViewer Integration	33
4.2	Conclusion and Future Work	35
	References	37
	Appendices	41
A	Tesseract Training	41
B	Translation Modelling	43
C	GlyphViewer	46

Chapter 1

Hieroglyph Language

This chapter will introduce the reader into the world of hieroglyphs and give a brief, perhaps inaccurate, and definitely incomplete description of the language, its history and the current status of the research that is being done. It is worth mentioning that the language had been quite a subject of popular interest in the 19th century, especially due to Napoleon's campaign in Egypt, the discovery of the Rosetta Stone and the subsequent translation of the hieroglyphs by Champollion. It was used by Napoleon to keep the "educated" people in France busy while he changed the laws of the republic to become the its new Emperor.

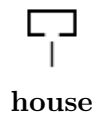
Today it is still a matter of interest, beside the current profession of Egyptology and Archaeology with ties in the Egypt's discovery and preservation of monuments, with faculty classes and research departments in the academia and many published books dedicated to learning the language which sell in around a few hundred thousand copies.

1.1 Introduction

In Greek *hieros* means sacred and *glyphos*, as you may already assume, stands for a symbol or a word. The first inscriptions of the writing system, which evolved in the Nile region, have been first dated to 3400-3200 B.C. with the last usage in 394 A.D. found in the temple of Philae. This writing system developed from a Bronze Age proto-literate language, a language with signs that depicts events, into a multi-thousand sign inscription that stabilized to about 900-1000 signs by the time of the Middle Kingdom (2040-1072 B.C.), the period when most of the monuments were built. This paper is dedicated to this era of the language.

An interesting fact about the language is that the symbols can be logograms, phonograms or determinatives. Logograms, which can also be found in the Mesopotamian writing system, can be dated back to 8000 B.C. and represent signs that depict a whole word. In Ancient Hieroglyphs this is depicted by a vertical line near or below the symbol in the script as you can see in Figure 1.1:













Fig. 1.1: Logograms



God - the flag represents a temple

The most important part of the language are the phonograms. As in modern writing systems a symbol denotes one or more sounds. In the case of Ancient Egyptian Hieroglyphs a symbol can denote one, two or three consonant sounds as you can see in Figure 1.2. Vowels are not depicted as in many modern languages in that geographic area. Nevertheless most egyptologists use this sound representation, known as transliteration, to read and work with the hieroglyph symbols. Though the pronunciation has been lost, there is a convention in the reading of the language by inserting vowels between the consonants with some rules such as **a** is read as a *short a*, **i** is read as *ee* and many more.

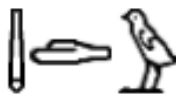
Fig. 1.2: Phonograms

glyph	translit.	biconsonantal			triconsonantal	
	<i>3</i>	 <i>ir</i>	 <i>mr</i>	 <i>sw</i>	 <i>nh</i>	 <i>rw</i>
	<i>i</i>	 <i>wp</i>	 <i>ms</i>	 <i>k3</i>	 <i>w3h</i>	 <i>htp</i>

As in phonetic language the sequence of consonant and vowels means a word or a phrase. This is the same in Ancient Hieroglyphs with the exception that the actual meaning of the pronunciation has been lost until the discovery of the Rosetta Stone.

An interesting fact of the language is the existence of phonetic complements, the doubling certain consonants. This helps the reader disambiguate the pronunciation of the symbols especially when dealing with two or tri-consonant symbols as you can see in Figure 1.3.

Fig. 1.3: Phonetic Complements



tongue - **mdr** written as *md+d+w*



Khepri - **hprj** written as *h+p+hpr+r+j*

Determinatives are also used as a disambiguation method by placing specific signs at the end of a word to be able to give a special meaning. They have no phonetic values and egyptologists do not transliterate it. In Figure 1.4 you see how a determinative symbol is used to denote plural or how multiplying a symbol can be also used for plural as well as the sign for a house to change the meaning of the same word (the same one used in a logogram in Figure 1.1).

Fig. 1.4: Determinatives
multiple meanings of nfr+w(plural) transliteration



beautiful young people - nfrw
the last 3 stokes/lines for plural



foundations of a house - nfrw
denoting plural by multiplying nfr tri-consonant
symbol and the house symbol as determinative

A special kind of determinatives are the cartouches, the oval outline of some of the symbols. They are used to mainly represent a name, either a king or a god.

Fig. 1.5: Cleopatra - last Pharaoh/Ptolemaic dynasty



The first thing that you are being taught in an Egyptology course is how to read the hieroglyphs. They are usually written either horizontally in lines or vertically in columns. Hieroglyphs can be read horizontally either left-to-right or right-to-left. The order is given by where the symbols are looking at (especially the animal symbols). Vertically one should always read them top to down. One should start reading in the horizontal order and go vertical whenever possible. Figure 1.6 tries to highlight the read direction by where the bird is looking at.

Fig. 1.6: Hieroglyphs read order


























Left to right

Right to left

If you want to learn more check the blog page of GlyphViewer site [1] or specifically at <https://www.2glyph.com/blog/word-of-ancient-egyptian-hieroglyphs/>

As with any language because of day to day usage needs in evolved in much simpler and hand-writable scripts. By this they started to loose their elegance and their pictograph beauty that can be found on temple walls. The first simplified form was the Hieratic version used on some Papyrus for business and religious texts. Then there was the Demotic form used in daily activities. They were finally replaced by a form of Greek writing named Coptic.

Fig. 1.7: Evolution of Hieroglyphs

Hieroglyphic		Hieroglyphic Book Hand	Hieratic		Demotic
					
					
					
					
2700-2600 B.C.	ca. 1500 B.C.	ca. 1500 B.C.	ca. 1900 B.C.	ca. 200 B.C.	400-100 B.C.

Source: <https://www.historymuseum.ca/cmc/exhibitions/civil/egypt/egcw02e.html>

This paper only focuses on their Hieroglyphic representation implementing OCR technologies and translation machine learning algorithms.

1.2 History of Egyptology

The existence of Hieroglyphic writing has been known for centuries, due to archeologic excavations, Greek documents, but at the start of the 19th century there was a race for their translation due to the discovery of the precious Rosetta Stone stone slab. For a long time the hieroglyphic script was thought to be purely ideographic but due to the work of Thomas Young [2] and most famous Jean-François Champollion [3] the realization that the script contained mostly phonograms was made.

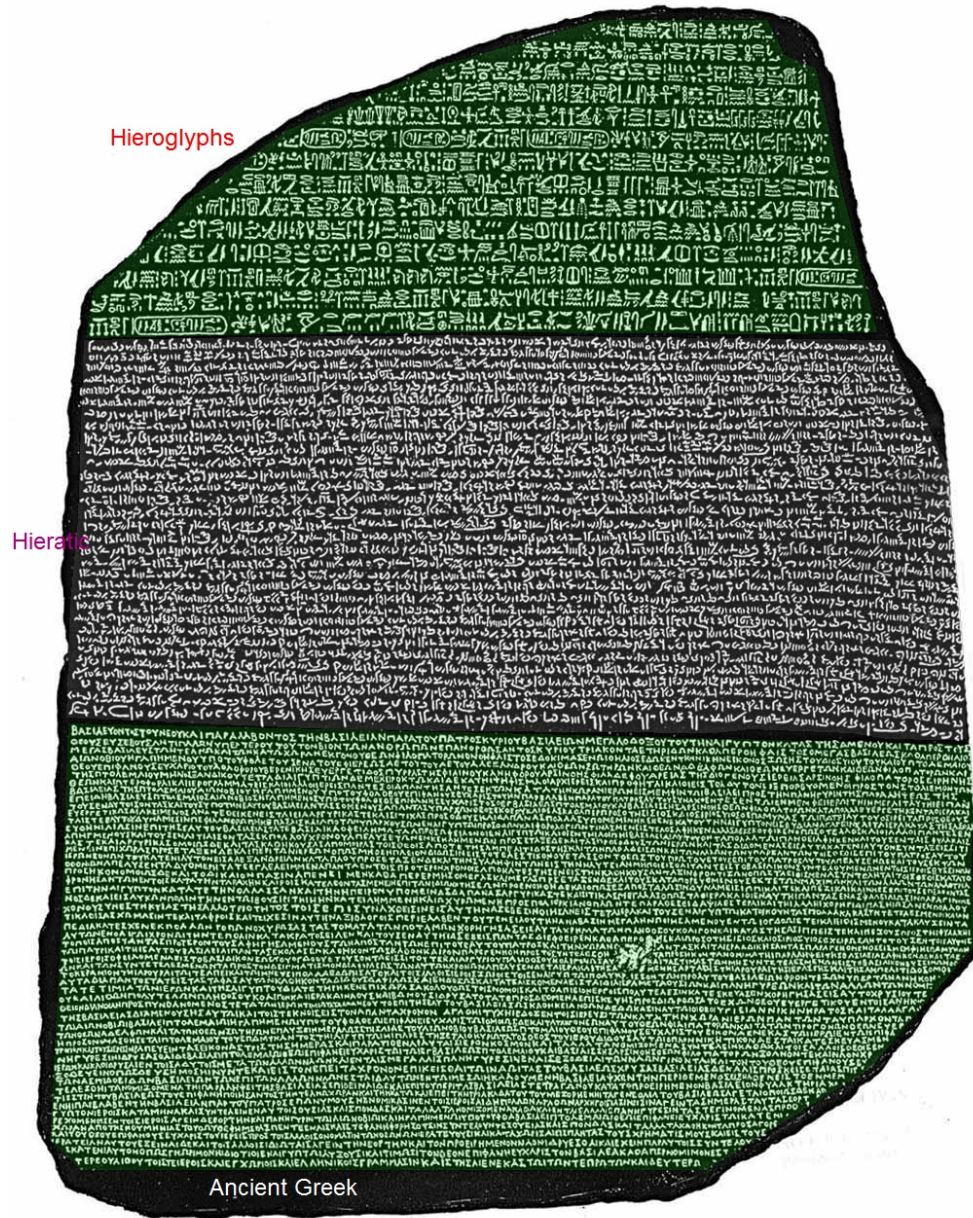
The history of the discovery of the Rosetta Stone is debatable but it is argued that was found by French soldiers during Napoleon's invasion of Egypt while fortifying the town of Rashid [4] and later given to the British as part of a peace treaty so that the defeated French army could be allowed to return to France. It is worth mentioning that Joseph Fourier, the famous mathematician, accompanied Napoleon expedition and was later put in charge of publishing the results and archeological finding and later became one of Champollion's allies. The slab contains a partial decree from king Ptolomy (204-181 B.C.) that was then published around temples as per era customs and contains 14 lines of Hieroglyphic text, 32 in Demotic/Hieratic and 53 in Ancient Greek as you can see in Figure 1.8. Since Ancient Greek was still known for the first time there was a translation of the meaning of the hieroglyphs.

There was a race between the two British and French scholars to decipher the text with Thomas Young identifying the name of Ptolomy transliteration and other meanings by applying symbol-to-word frequency techniques. But it was Champollion's insight that the Hieroglyphic and Demotic scripts contained a mix of ideographic and phonetic symbols that paved the way for a proper translation. Thus and the fact that Champollion was

fluent in Coptic, the last evolution of Hieroglyphic script, spoken in Egypt at that time, that used the Greek alphabet and some Demotic inherited signs for its writing.

It is worth mentioning that Champollion's work was published in 1822, more than a century before the first architecture of a compute machine. With today code breaking software matching common sound patterns to symbols would be trivial. Nevertheless without the Rosetta Stone, given the limited number of actual Hieroglyphic and Hieratic texts still present it is hard to know if a translation could be deduced.

Fig. 1.8: Rosetta Stone



top: Hieroglyphs, middle: Hieratic, bottom: Ancient Greek
Source: <https://www.2glyph.com/blog/rosetta-stone-and-glyphviewer/>

A full translation of the above text can be found at [5].

Since then there have been a number of scholars [6] that dedicated part of their lives to the new field of Egyptology. This led to numerous discoveries in Egypt, text translations, word dictionaries and a number of books.

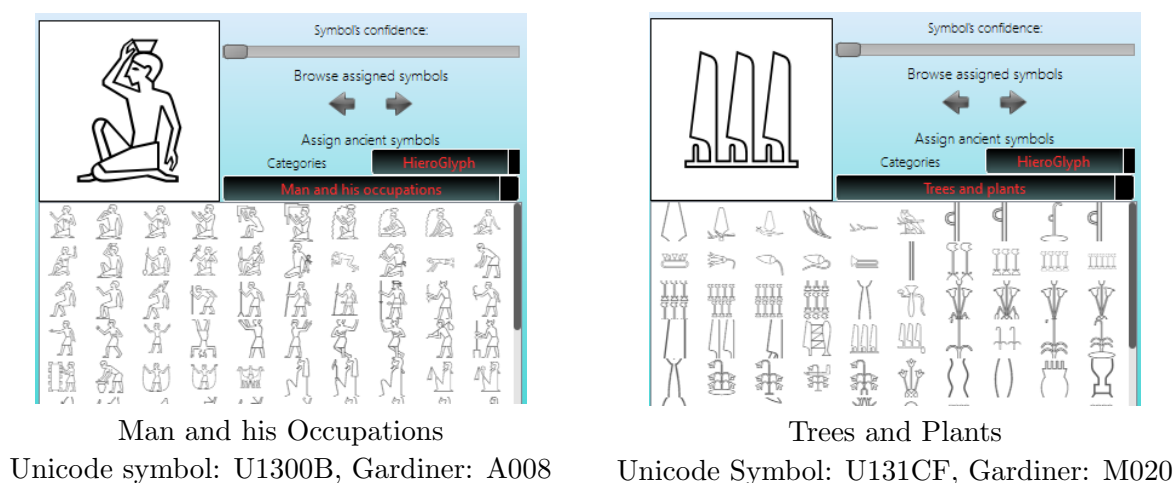
In the last 30 years new computer assisted tools and software, though limited, emerged including fonts and Unicode standardization of the hieroglyphs so that they can be used in any text editor.

1.3 Hieroglyph Symbols

There are more than one thousand identified Hieroglyph symbols used in the Middle Kingdom period. To be able to do anything constructive with it a largely accepted encoding is required, especially since written fonts were not available. A breakthrough for learning and doing research in Hieroglyphic writing was done by the British Egyptologist Sir Alan Gardiner. It is well known for one of the first thorough books on Egyptology, *Egyptian Grammar: Being an Introduction to the Study of Hieroglyphs* [7] that produced one of the first dictionaries of translated words and one of the first hieroglyph fonts used for printing. But the most famous achievement that became the cornerstone of Hieroglyph encoding, is its Gardiner Sign List[8], a list of 1071 glyphs split up into 28 categories based on a common looking theme.

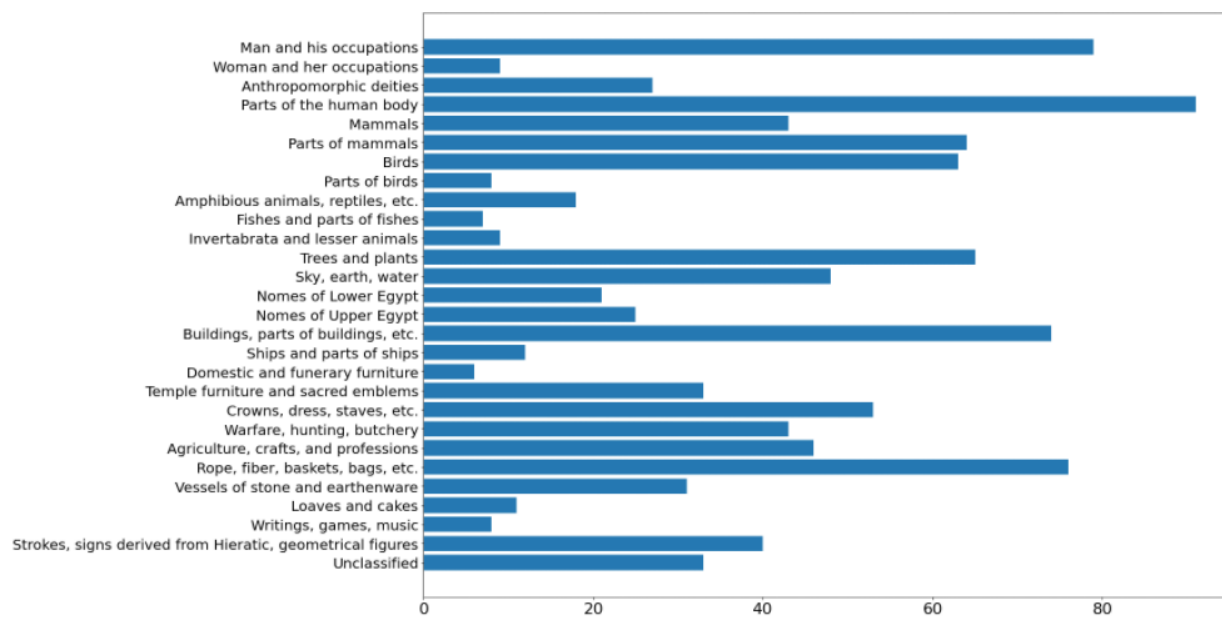
A more modern encoding came with the standardization of Hieroglyphs into Unicode values [9] ranged from U13000-U1342F. It is based on the same Gardiner category classification, with a few exceptions. GlyphViewer supports both standards and can convert between the two encodings but the OCR and translation engines were trained to use the Unicode standard. In Figure 2.1 you can see 2 of these types of categories as they can be seen in the application.

Fig. 1.9: Hieroglyphs categories



The 28 category symbol histogram can be found in Figure 1.10

Fig. 1.10: Number of symbols in each category



For many Egyptologists just encoding the signs on a temple was is not enough, the position and relative shape of the symbols have also value, especially an aesthetic one. For this reason a new encoding scheme was proposed in 1984 called Manuel de Codage [10], or Mdc for short. It uses special codes, Gardiner's notation and transliteration to properly position the symbols relative to one another.

GlyphViewer does not offer support for Mdc but relies on drag and drop and ability to move glyphs around to properly align the symbols.

For my own self-instruction on how to read hieroglyphs I used 2 books, namely *How to read Egyptian hieroglyphs: A step-by-step guide to teach yourself* [11] and *Egyptian hieroglyphs for complete beginners* [12] as well as many online resources.

Chapter 2

Recognition

In this chapter I will talk about symbol recognition from images. The Unicode Middle Kindom hieroglyph specification [9] contains 1071 different symbols grouped by their Gardiner category. The purpose of any recognition algorithm would be to identify sequence of symbols in an image along with their location.

Though a Yolo based symbol detection algorithm along with a CNN would work I focused to offload this compute intensive part to client devices. Hence, as I previously did in GlyphViewer, I used Tesseract [13] OCR engine maintained by Google. Tesseract is probably the best known open source engine out there, originally developed by HP in 1980, and now still being used at the heart of some of the OCR needs of some of the biggest companies around. The usage of an OCR engine for this purpose was made because Hieroglyph texts were written such that they can be read by a user, with proper horizontal and vertical alignment. This is the domain space that any OCR engine has been highly tuned for image segmentation and classification.

2.1 Symbols Dataset Creation

Any OCR engine needs a dataset to be used for training. In the case of Tesseract this is made up of a collection of images constructed from different fonts and sizes along with the corresponding bounding boxes - a text file containing each symbols identifier with their position and size in the image.

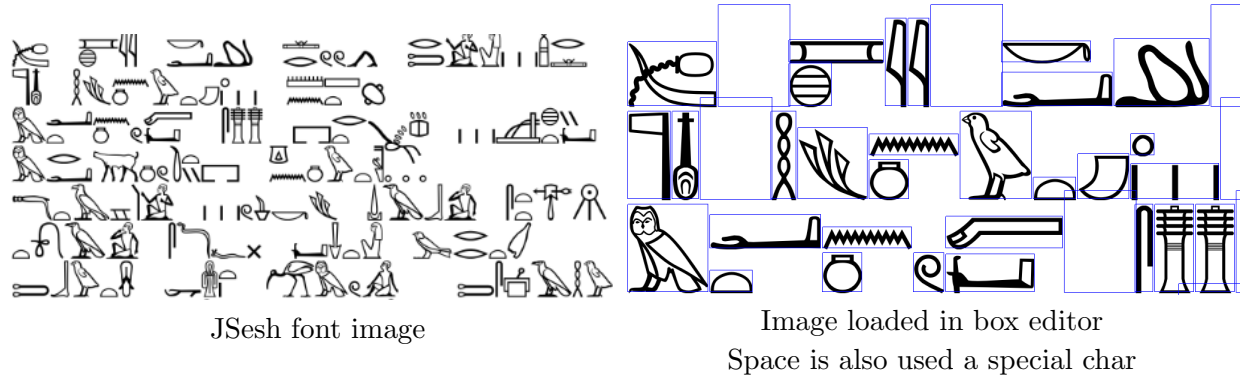
To date there is no symbol dataset of hieroglyphs pictures from actual temples. Luckily there have been some initiative to create actual fonts for Hieroglyphs Unicode. Bob Richmond maintains a list of Hieroglyphs fonts [14]. I used *Aaron font* [15] created by him with both left-to-right and right-to-left flavors, Serge Rosmorduc's *JSesh font* [16], Mark-Jan Nederhof's *NewGardiner font* [17] and *Noto font* [18] from Google. The OCR will be trained on the Unicode value of the symbols.

A new Qt C++ application was created to output text images given a list of fonts and list of hieroglyphic text or dictionary. In my case the *Mark Vygus Middle Egyptian dictionary* [19] was used to get a list of words that was used to randomly generate text. A special characteristic of the text generator is that it tries to group 2 symbols together by putting them one under another to mimic a temple wall writing. A VB .net application

was also developed that could load the generated images with their corresponding bounding boxes to test the quality of the generation.

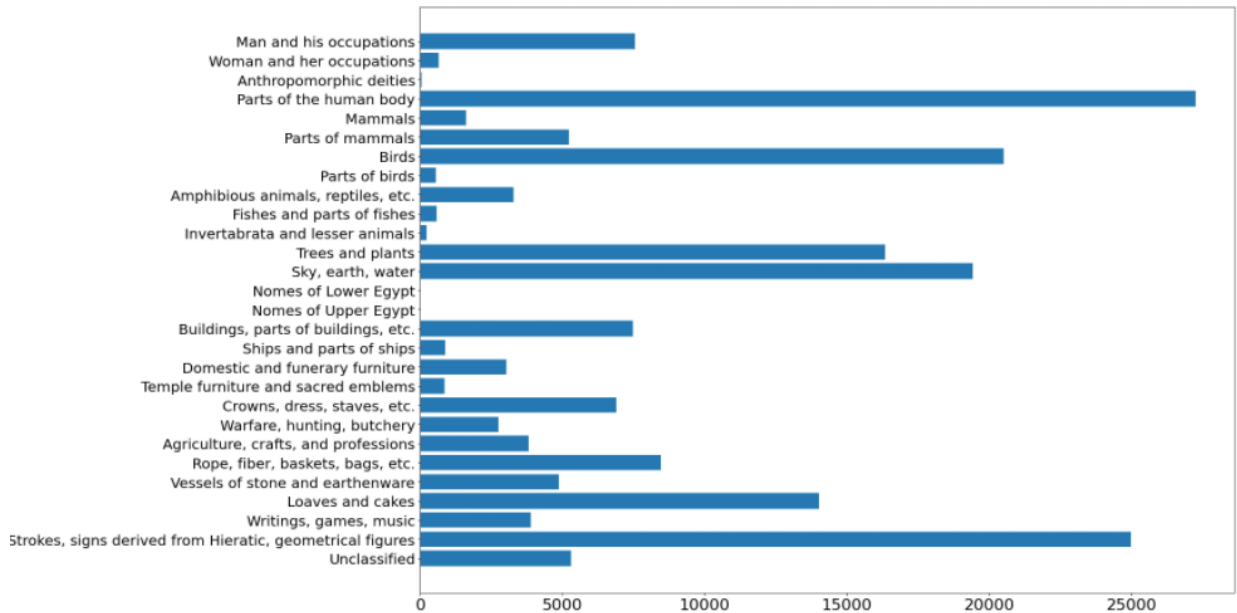
For each font all words in a dictionary were randomly generated with different sizes and resulted in about 2500 images for training and about 200 images for evaluation totaling about 13GB.

Fig. 2.1: Symbol image dataset



The symbol usage histogram, per category, found in the words in the dictionary can be found in Figure 2.2

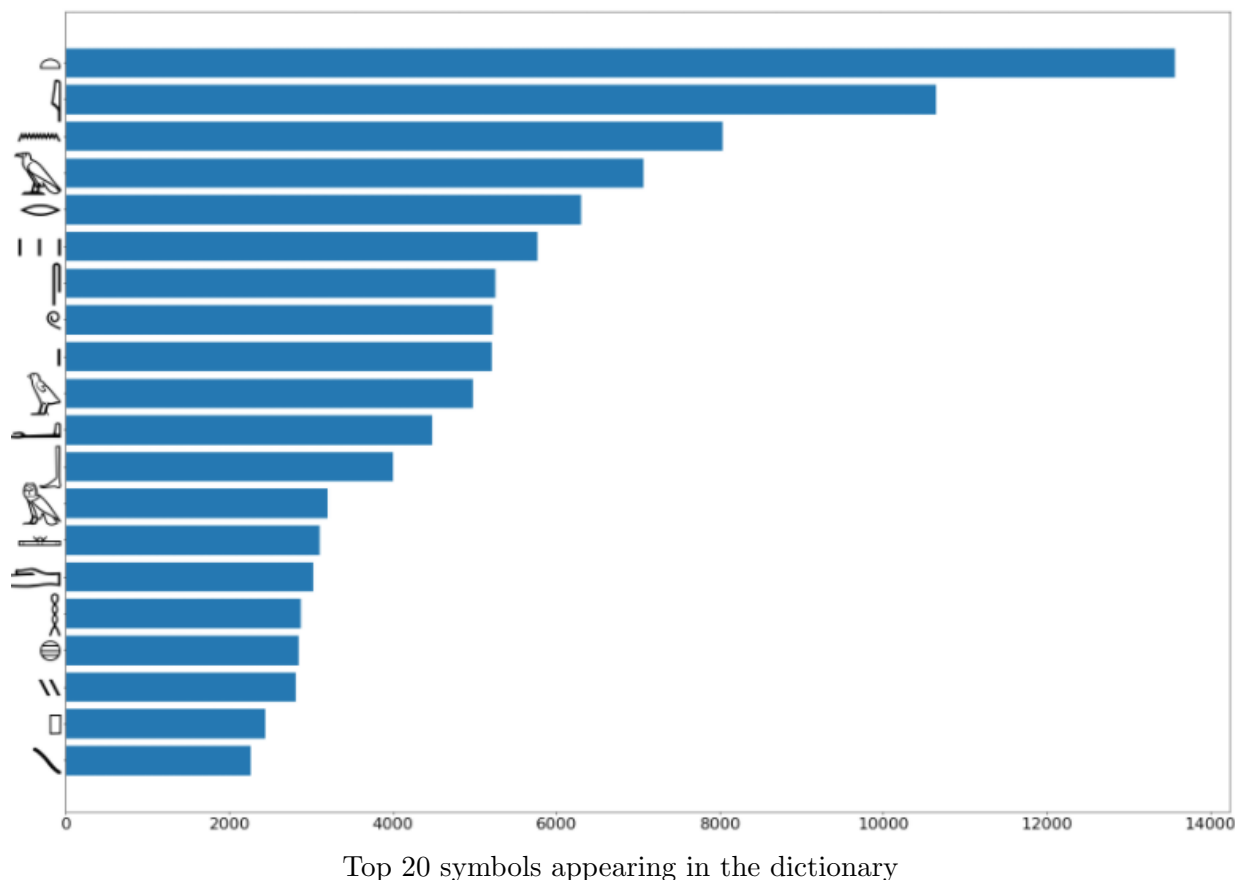
Fig. 2.2: Dictionary Category Symbol Histogram



How many times a symbol in a specific category was used in the dictionary

There are 38645 words in the dictionary and 190604 hieroglyph symbols making up the words. The top 20 symbols used in the dictionary, and hence in the OCR training, are shown in Figure 2.3

Fig. 2.3: Dictionary Top Symbol Histogram



The symbols that were not used in any of the dictionary words were also synthetically generated to be able to recognize them. The special *space* character was also added to the language to properly separate words for OCR training.

Apart from my own C++ image generator application, Tesseract's own *text2image* tool was used for the 5 fonts to generate additional images with certain distortion thresholds.

2.2 Tesseract LSTM model

Starting with version 4, from 2016, a new machine learning framework was added as new OCR engine to Tesseract. Although there are a few tutorials on how to develop, train the new models for a new language, little detail is given on the actual implementation.

The current OCR pipeline still does page layout analysis but once a text line has been identified it feeds it into the LSTM language model that outputs the characters. A new language model can be created for any new script and Tesseract provides tools for its creation and training but most importantly it implements an new model specification known as *Variable-size Graph Specification Language* (VGSL) [20]. This allows a very shortened description of the model in a single string. Multiple deep learning layers, such as convolution lstm, bi-lstm, dense have been implemented directly in C++ and included in Tesseract's engine. For performance improvements the implementation requires AVX

instruction set but has some pretty decent results and outperforms old engines used before, though training on multiple fonts for a Latin based language can take a few days.

An important aspect of the implementation is that it uses Connectionist Temporal Classification [21] loss function so that training does not require the actual symbol position, only the line ground truth, and beam-search decoding [22] for better outputs. Though not well documented one can learn more by looking at the code <https://github.com/tesseract-ocr/tesseract/tree/master/src/lstm>

As explained in the new LSTM engine training page [23] by Ray Smith, one of the original engineers working on Tesseract, the process starts with the same data generation as in Section 2.1.

The training process only works on Linux so a Ubuntu VirtualBox machine was used with tesseract 4.1.1 distribution and tools. Prior to this all the 5 fonts mentioned were installed.

First a charset file, `hie.unicharset`, was created that contained all the characters as identified from the box files of the train dataset using the *unicharset_extractor* tool. Mode 3 was used for pure Unicode compression of the glyph codes. In total there are 1073 character, 1071 glyphs, space and the null character. Next the starter `hie.traineddata` file is created using the *combine_lang_model* tool given the unicharset file and the language configuration file taken from simplified Chinese. The config file contains properties regarding tesseract's page segmentation and char prototyping, clustering and more. They are used in our LSTM model case to get the region of text in an image, namely the horizontal row of a text. I chose Chinese config since it is also a language with many similar symbols.

Now I can use the newly created `hie.traineddata` file to generate the `lstmf` files from the images and boxes files. These `lstmf` files are the binary input of the images that will be given to the LSTM model. This whole process takes about 2 hours to complete in parallel on a Ryzen 5600 4 CPU VirtualBox machine with 16GB of RAM. I created a list of `lstmf` files to use for training and one for validation.

Finally there is the model specification step and the actual training. At this stage 2 models were tried:

- A transfer learning model from Chinese simplified where the last 2 layers were replaced [**1,48,0,1 C3,3,16 Mp3,3 Lfys64 Lfx96 Lrx96**] - pretrained + **Lfx256 O1c1073**
Total weights of the last 2 layers: 637233
- A new CNN+LSTM model trained from scratch
[**1,0,0,1 Ct3,3,32 Mp3,3 Lfys64 Lfx128 Lrx128 Lfx512 O1c1073**]
Total weights: 2118769

Notice the elegance of the specification. Let's dive into describing the second one:

1. **1,0,0,1** specifies the input type that that the row region in the image has to be converted to. In my case it is variable on X and Y and in grayscale (last 1). First 1 denotes a batch of 1 - the only one it is supported. Because the input is variable we need a summarizing LSTM in later layers.
2. **Ct3,3,32** a convolution layer of 3x3 with 32 filters and tanh

3. **Mp3,3** max pool layer of 3x3
4. **Lfys64** LSTM on Y with 64 outputs that is also a summarizing LSTM (the **s**)
5. **Lfx128** LSTM on X with 128 outputs forward direction
6. **Lrx128** LSTM on X with 128 outputs this time in reverse
7. **Lfx512** The same forward LSTM on X but with 512 outputs
8. **O1c1073** Fully connected output layer with 1073 nodes and because of the **c** it is trained with CTC

The 2 models were trained with Adam optimizer with an initial learning rate of 0.002 and momentum of 0.5 and where trained for 120000 epochs for about 7 hours each to train on the above virtual machine.

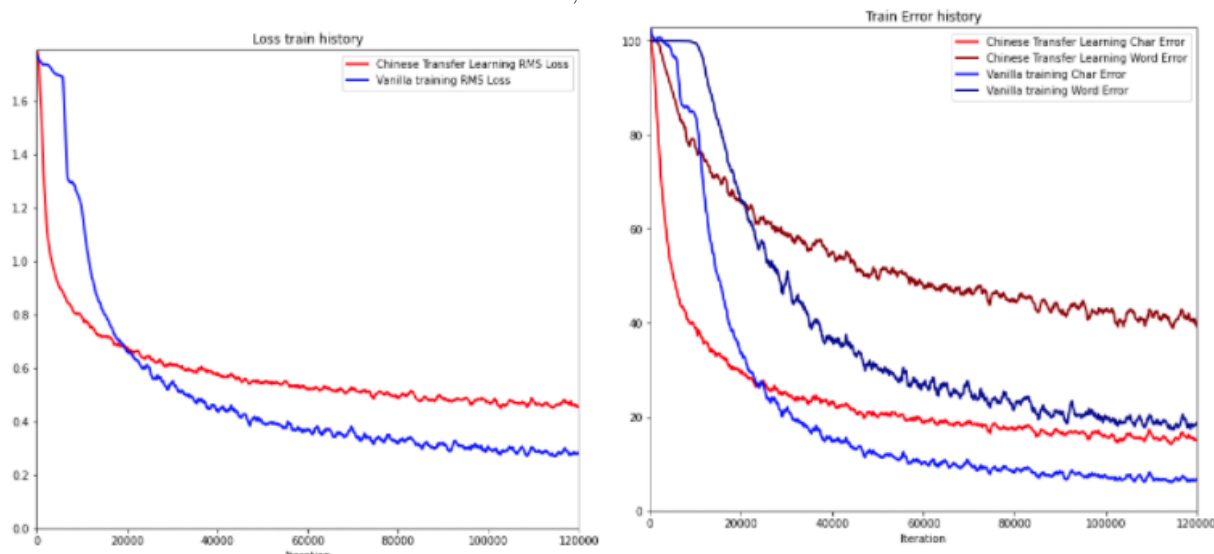
A complete set of commands needed for the above steps can be found in Appendix A.

2.3 OCR results

To be able to show some nice graphs of the training process I had to parse the log outputs of the training and convert them to csv files that could be them loaded in Jupyter Notebook and plotted. The work is available at <https://github.com/glyphier/hieroglyphs>. The results have been plotted for both models in Figure 2.4 and shows that the model trained from scratch has better performance.

Fig. 2.4: Training results - both models

Chinese transfer model in red, model trained from scratch in blue

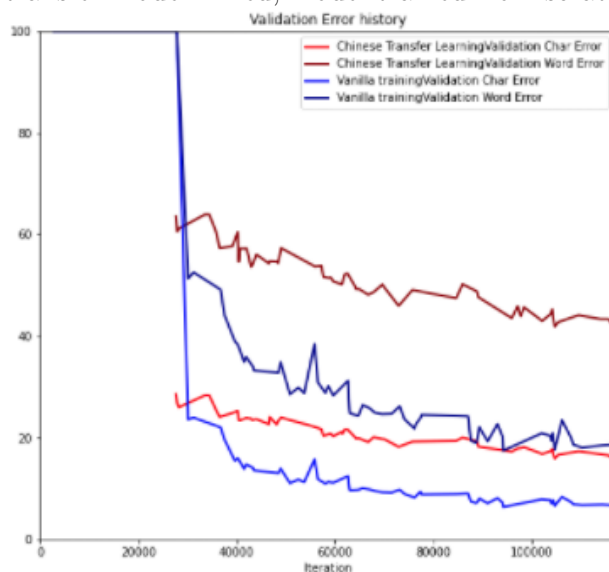


Loss plot - shows decreasing trend

Train error percentage
Both char and word error decreases

Fig. 2.5: Validation results - both models

Chinese transfer model in red, model trained from scratch in blue



Both char and word error decreases but seem to stabilize

The graphs clearly show that the model from scratch works best. Also they might have needed to be trained a bit more since the loss, train and validation error are decreasing. But since the validation error kind of starts to stabilize the benefits might not have been that great. Another interesting thing is that the word error in both models is always greater than the char error. This might be due to the fact that if you miss a symbol you miss the entire word and there are about 6 symbols per word on average.

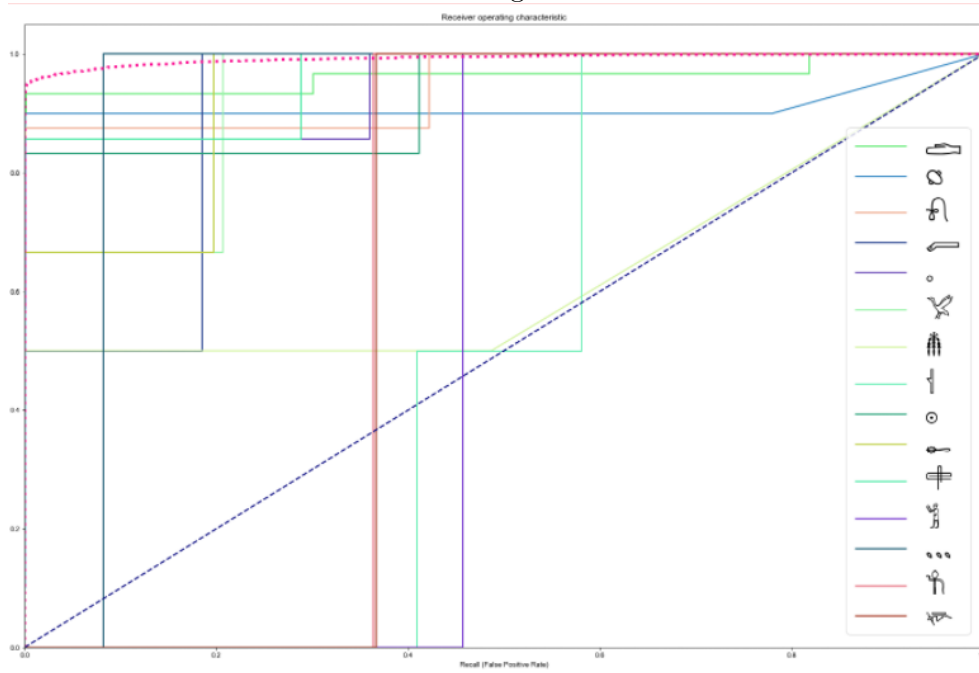
For the Chinese model the best train char error is 14.188% and the word error is 39.4%. For the model trained from scratch the char error is 6.034% and the word error is 17.3%. The validation error for both models are very close to the train error, 16.75% char, 42.9% word error for the Chinese transfer model and 6.77% char, 18.44% word errors for the other model.

The hard part was to actually get the scores of the prediction on the evaluation data set so that I could analyze the model's performance using standard methods from classification modelling such as the ROC curve. I had to modify the *lstmeval* tool's C++ code and force it to spill out the scores. Also because beam search is used each character might had multiple spilled out scores. To overcome this for each of predicted character I averaged the multiple scores of the beam search that had the maximum score the same as the one of the predicted character.

The best model ROC curve for it and the 15 classes with the lowest AUC was plotted in Figure 2.6. In a multi-label classification setup ROC curve for each class denotes how the model can differentiate that class from all the others. Since there are 1072 symbols to be able to make a sense how the model behaves I also created a histogram of all AUC scores in Figure 2.7

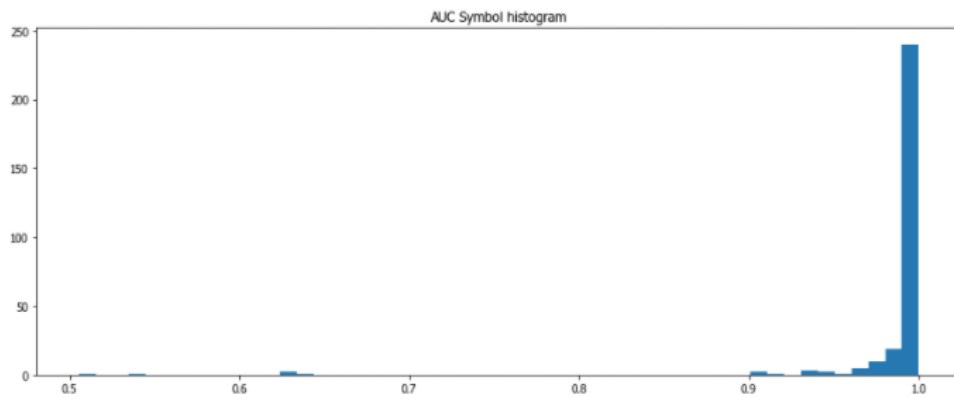
Fig. 2.6: ROC plot - best model

Dotted line is the average ROC for all classes



ROC for 15 symbols with the **lowest** AUC was plotted

Fig. 2.7: AUC symbol histogram



The histogram of all symbols AUC on validation data

Notice that most symbols have a very high AUC

Chapter 3

Translation

This chapter will present the 2 data sets that were created to be able to perform some translation. It will present how some dictionary words are identified in a list of OCR-ed hieroglyph symbols. Finally some transformer based models have been used to link the transliteration of identified words to an English phrase.

This is perhaps the first algorithm that truly tries to translate hieroglyphs from an image. Recent research on this topic has been done at Google in Fabricius project [24] but they do not construct a translation, rather they allow users to generate hieroglyphs from English text. GlyphViewer [1] can do this trans-encoding in both ways and predates this by at least 8 years, being first published in 2013.

3.1 Translation Dataset Creation

There were 2 main datasets constructed for ML purposes in CSV format:

- **dictionary** parsed from Mark Vygyus Middle Egyptian dictionary [19]
- **translations** parsed from Mark-Jan Nederhof's St Andrews Corpus [25]

The dictionary contains **38421** entries as seen in Figure 3.1

Fig. 3.1: Dictionary of Hieroglyph Words

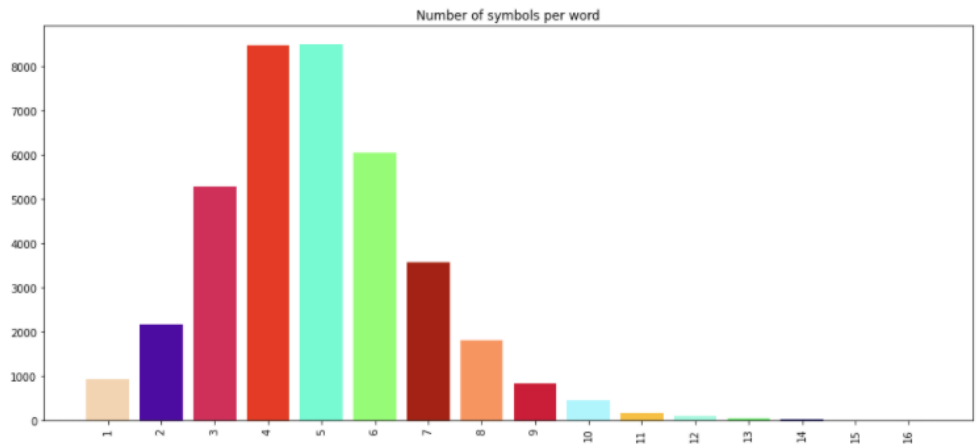
	symbols	text	pos	transliteration
0	𓂀	I, me, my	pronoun	wi
1	𓂁	I stative - past tense		.kwi
2	𓂂	man, someone, anyone, no-one	noun	s
3	𓂃 𓂃𓂃	man, men, mankind, Egyptians, people	collective noun	rmT
4	𓂃𓂃𓂃	people, mankind, humanity	collective noun	rmT

Loaded in a Jupyter Notebook

The most important entries in this dataset are the **symbols**, which contain the Unicode values of the list of Hieroglyph symbols that make up the word, the **transliteration** which, as discussed in Chapter 1.3 is the phonetic reading, without vowels, of the symbols, the part of speech and the English **translation** of the word.

The Hieroglyph words are constructed mostly from 3 to 7 Hieroglyph symbols as can be seen in Figure 3.3

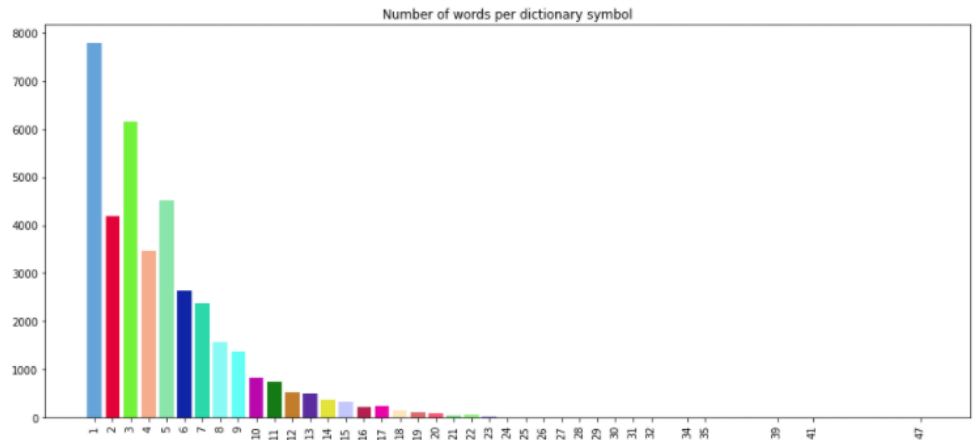
Fig. 3.2: Histogram of number of Hieroglyph symbols in a word



8000 words contain 4 or 5 symbols each
Notice the Gaussian shape of the number of symbol distribution

A Hieroglyph word does not map to a single English word as its meaning may be different based on context as is shown in the histogram in Figure 3.3

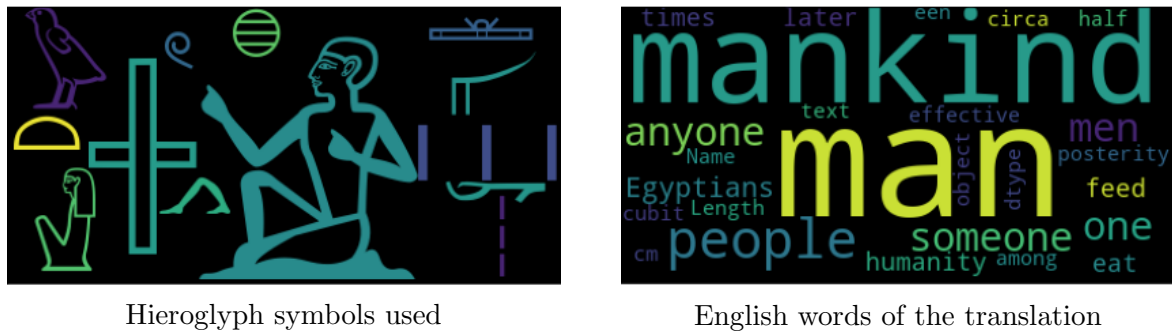
Fig. 3.3: Histogram of number of English words in a Hieroglyph word



Most of the Hieroglyph words have a meaning described in 1 to 7 English words
There are a total of 968 unique English words

To see a nice visualization of the Hieroglyph symbols and English words in a dictionary a WordCloud plot was made for both in Figure 3.4

Fig. 3.4: Dictionary WordCloud



The **translation** dataset contains **8196** entries with the Hieroglyph transliteration and the English translation as can be seen in Figure 3.5

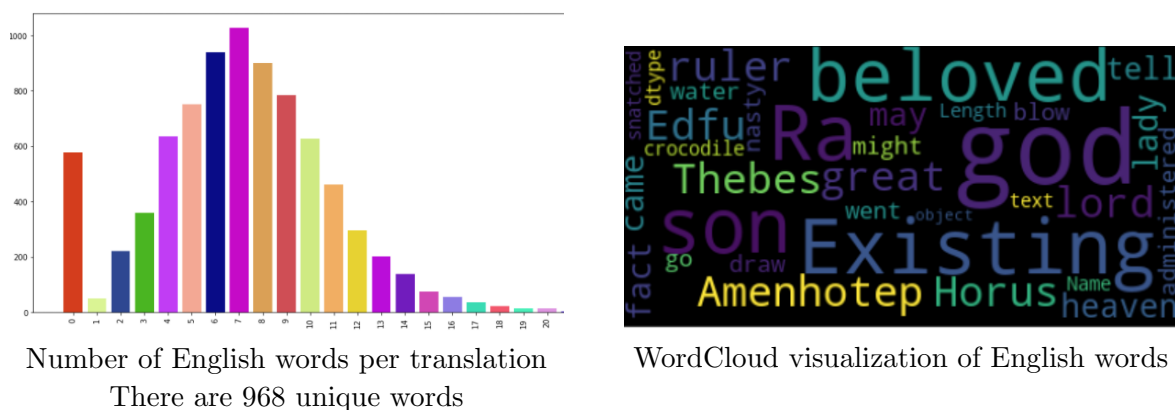
Fig. 3.5: Hieroglyph Translations

	text	transliteration
0	Existing:	wnn
1	the son of Ra,	sA ra
2	his beloved,	mry=f
3	Amenhotep, god and ruler of Thebes...	imn-HTp nTr HqA wAst
4	Horus of Edfu, great god, lord of heaven, may...	^Hr-^bHdtj nTr aA nb pt Dj=f anx

Loaded in a Jupyter Notebook

The English translation are rather small, with less than 110 characters containing on average about 8 words as you can see from Figure 3.6

Fig. 3.6: Translation English text



Next I will talk on how to link the dictionary data with the Hieroglyphs dataset.

3.2 Translation to Words

The translation dataset contains intelligible phrases in English of a desired translation. On contrary the dictionary contains a multitude of English words, for each Hieroglyph words. In order to link the 2 datasets I used the transliteration since it will be the same in either dataset. If a Hieroglyph word was used in making a translation then its transliteration should also be present in the the translation's transliteration. For example in the below phrase transliteration "Dj.n(=j) n=k anx wAs nb" we can identify *wAs* as dominion, *anx* as life and so on:

translation		transliteration
I have given you all life and dominion		Dj.n(=j) n=k anx wAs nb
-----Identified words-----		
transliteration	meaning	
d	help	
.n	we, us, our	
n	to, for, to (persons), in, upon, of, because, belongs to	
anx	life, oath, vow	
wAs	dominion, have dominion, well being, prosperity	
nb	any, every, all, all kinds of	

Since I had no ground truth on which words are part of a translation I used the following heuristic algorithm to solve the problem:

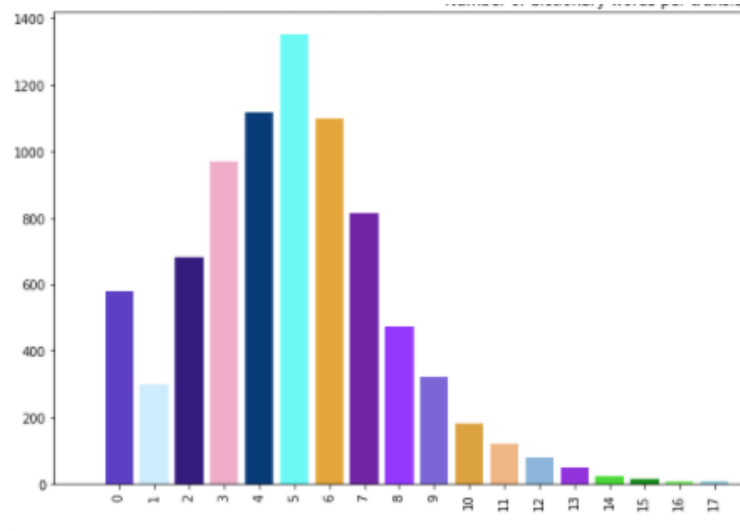
1. Apply Aho-Corasick algorithm from [26] to identify the position of all possible word transliteration in the translation's transliteration in linear time
2. Sort the identified positions and choose the largest transliterations that do not overlap from all possible choices - sort of a grammar parser disambiguation rule
3. For all possible English translations of each remaining word transliteration choose the one with the most common English words with the phrase.

Step 2 tries to find the most important transliterations, and words, from the phrase by assuming they have the largest transliteration. Aho-Corasick algorithm will return all string matches so for example for transliteration *anx* it will return both *an* and *anx* transliterations as they are both found in the dictionary.

Step 3 solves the problem that for each identified word transliteration there are more then one Hieroglyph word and hence English meaning. For example *anx* has 46 different translations, like 'person, inhabitant, citizen, living one' or 'goat, small cattle' and 'life, oath, vow'. From all possible choices I choose the one which has the most English words common with the original English phrase, in our case 'life' is in 'I have given you all life and dominion'.

Figure 3.7 shows the histogram of identified Hieroglyph word transliteration in a translation's transliteration. On average there are 3 to 8 Hieroglyph words used to build a translation.

Fig. 3.7: Identified Hieroglyph words per phrase



The number of identified word transliterations in a phrase transliteration

3.3 Symbols to Words

The first step in the translation process is to map a group of identified Unicode glyph symbols to a list of possible hieroglyph words. This step assumes a horizontal or vertical list of symbols as a continuous vector of Unicode hieroglyphs. The algorithm is the same one as used in Section 3.2 but this time on a `VECTOR<CHAR32_T>` structure:

1. Apply Aho-Corasick algorithm from [26] to identify the position of all possible word from the list of Unicode symbols in linear time
2. Sort the identified positions and choose the word with the largest symbol list that do not overlap from all possible choices.

Step 2 above is chosen because the dictionary contains the same word multiple times with or without determinatives so I could select the word with any possible larger suffix. For example out of the 38421 dictionary entries, 28777 are words that have the same transliteration, leaving 9644 unique word meanings. Remember that in Section 1.1 I discussed that determinatives are not transliterated.

Since there is currently no corpus containing translations as a list of symbols I could not provide any statistics on how this maps in a real word example. Nevertheless the OCR will spill out such a list to be translated.

3.4 Translation Models

This section describes 2 category of models, a transformer text generation and a classifier, that were trained to give an English representation of hieroglyph symbols. As explained in Section 3.1 the main training corpus consists of transliterations to English sentences.

Section 3.3 describes an heuristic that is used to map the list of OCR spilled symbols into dictionary words, for which transliteration is available. Basically the main translation pipeline can be described as:

OCR -> SYMBOLSToWORDS -> TRANSLATION MODEL
 IMAGE -> VECTOR<CHAR32_T> -> TRANSLITERATIONS -> ENGLISH SENTENCES

To be able to compare the results between the 2 models the same input dataset was created as described in Section 3.4.1. All models were implemented in tensorflow 2.6 and trained on a AMD Ryzen 3600/32GB/NVIDIA 1070TI machine.

3.4.1 Input Dataset

To create the tensorflow Dataset for the inputs 2 pandas dataframes were used:

- **original translations** as described in Section 3.1
- **identified word transliterations** by applying the transliteration to words algorithm described in Section 3.2

First the translation corpus was filtered for text duplicates leaving 7130 entries from the original 8196. The then string transliterations that were more than 50 chars were removed for performance reasons, leaving 7119 entries that were used.

Then the second dataframe was created by trying to identify the list of words based on the transliteration and join them together to get a transliteration of the entry that contains only data from the identified words.

	translation	
	I have given you all life and dominion	
original transliteration		word list transliteration
Dj.n(=j) n=k anx wAs nb		d .n n anx wAs nb

Notice the missing transliteration elements in the above word list category as only partial elements were identified in the original transliteration. I computed the BLEU [27] score for the string matching of the 2 transliteration dataframes to check how much of the original transliteration is identified by the dictionary words on a word tokenization or char tokenization of the entries:

- **BLEU word tokenization:** 0.02223
- **BLEU char tokenization:** 0.39723

Since char tokenizer seems to have a larger similarity (1 BLEU score is perfect match) I chose to do this as a pre-processing step for transliteration. For the actual English translations word tokenization is used.

Text preprocessing is done using the *matmih*[28] library. First from the 2 dataframes and the word dictionary is used to build 2 vocabularies as the chars of the transliteration and the English words of the translations. The size of the transliteration vocabulary, of only chars, is 79 and the English words is 14672. A special token for padding is added as 0 and

end of a sentence as 1 in both.

For training both dataframes are concatenated and for validation only a modified word transliteration is used and the following pre-processing is done:

Transliteration:

```
['tokenize_char', 'to_vocabulary_ids', 'add_mask', 'add_end_token', 'padding']
```

English translation:

```
['tokenize', 'to_vocabulary_ids', 'add_mask', 'add_end_token', 'padding']
```

The pre-processing pipeline takes the transliteration or text column, tokenizes the string at a word or char level, build the ids of the words using the pre-computed transliteration or word dictionaries, adds a word mask and an end token and pads the list to 50 tokens because the maximum length for transliteration is 49 chars and 47 words for translation. The resulted transliteration and translation ids and mask is processed as a tensorflow Dataset, described in Annex B, as following:

Transliteration: wHm-a m rDjt st<eos>

Translation: To be repeated by putting it <eos>

```
inputs:          [3 15  9 14  8  2  9  2  7 24 23 19  2  5 19  1  0  0 ...]
input_mask:      [1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  0  0 ...]
targets:         [1320  104 1358  207 1359  246  1  0  0  0  0  0  0  0 ...]
sample_weights: [1      1      1      1      1      1      1  0  0  0  0  0  0 ...]
```

For the classification model targets is replaced by the unique translation identifier in the dataframe.

To be able to generate additional synthetic data and train the models as a Language model the input entries in the dataset will randomly mask/remove 25% of the char ids. Each epoch the original entries as well as the masked entries are used in training the models, hence each epoch will run $13774 * 2$ entries. The masked input data will remove some transliteration ids for the above input with no change to its targets:

```
inputs:          [3 15  9  8  2  9  7 24  2  5 19  1  0  0  0  0  0  0 ...]
input_mask:      [1  1  1  1  1  1  1  1  1  1  1  1  0  0  0  0  0 ...]
```

3.4.2 Transformer Model

The first attempt to model the translation of hieroglyph transliteration was to treat the problem as a text generation, translation problem. Transformer were and still are in 2021 very successful for language translation. In my case I will use it to transform the transliteration input, the pronunciation of hieroglyph symbols, to the English phrase. Usually a transformer is composed of a decoder part, that converts inputs into a latent space and decoder which takes this encoder input and outputs a text representation. The section does not intend to go into the details of a transformer implementation but one advantage of using these heavy models in the availability of pre-trained snapshots and multiple production implementation:

- Encoder only as BERT, ROBERTa, XML

- Decoder only such as GPT-2, Reformer, XLNet
- Sequence to Sequence Encoder + Decoder such as BART, T5, ProphetNet

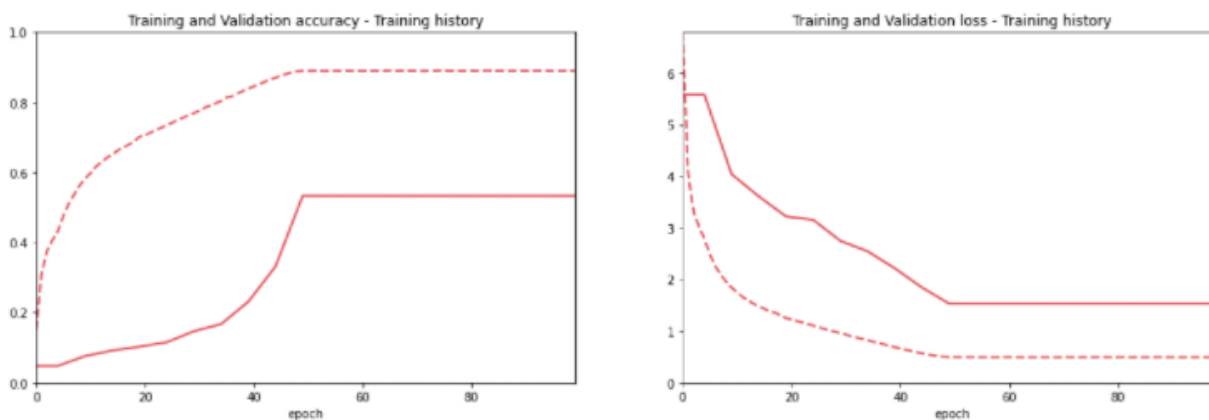
The language translation is modeled as sequence to sequence encoder-decoder transformer using the raw implementation of a transformer from *TensorFlow Model Official* [29] library. The implementation of that transformer is based on *Attention is All You Need* [30] 2017 Google paper. An interesting aspect of the implementation is that there is a single vocabulary for both inputs and outputs, in our case the transliteration vocabulary merged with the word text vocabulary, and the vocabulary embedding weights are shared between the encoder and decoder. A positional encoding layer is also added as input to the Encoder to give it relative position information of the words. Advanced techniques such as beam search are also used in the decoder output when doing prediction.

Input for training consisted as described in Section 3.4.1 of both the original transliteration to translation data and by language masking techniques for better regularization in this case. The loss and gradient is computing as a sparse categorical cross-entropy logit loss between the predicted and actual vocabulary id for each output word in the sentence until the `jeosj` token is found. This is done using the `sample_weights` mask of the input. More can be seen in the actual model implementation in Annex B.

The model was trained for 100 epochs using Adam optimizer with weight decay [31] with an initial learning rate of $3e-2$. The configuration included adding 3 transformer layers for both the Encoder and Decoder, an word embedding size of 32, 8 attention heads and intermediate Dense layers of 512. The model thus has 628096 trainable weights so a batch size of 256 could be used on the 8GB 1070TI GPU. Beam search was set to have 4 beams with an alpha of 0.6, pretty much as in the default settings.

It achieved around 0.8894 training accuracy and 0.5331 validation accuracy after training for about 50 minutes. From Figure 3.8 you can see that both the loss and accuracy stabilizes after epoch 50 which suggest that even a larger capacity model should be tried.

Fig. 3.8: Hieroglyph Transformer accuracy and loss



Dotted lines for training, solid for validation

The BLUE score for the between the original translations and the one generated by the transformer model is 0.5462 and for the output of transliteration generated only by

identified dictionary words is 0.5118.

Some transformer output when a single identified word is discharged and other words are aligned randomly:

```
Translation:          "seeing the favour towards you of the king.'"
Word Transliteration:  mA sn Hswt nt Xr nsw
Transformer:          "seeing the favour towards you of the king . '"
Changed Transliteration: Hswt nsw mA nt sn
Transformer:          "the ruler equal of Re ,"

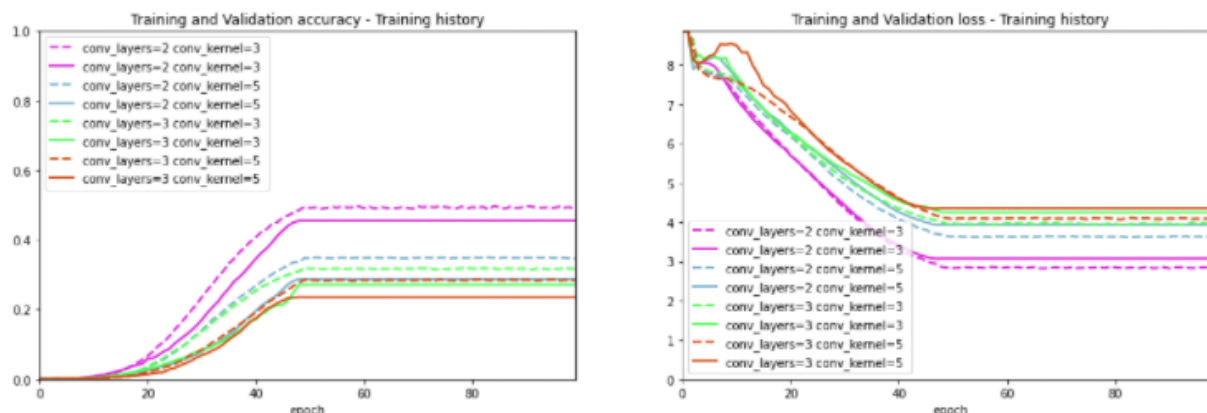
Translation:          "Then Ruddjedet said:"
Word Transliteration:  aHa .n d d .n rD Ddt
Transformer:          "Then Ruddjedet said :"
Changed Transliteration: rD aHa .n d .n d
Transformer:          "Then he said to her :"
```

3.4.3 CNN Model

A safer way to treat this is as a classification problem. The same input data processing was done as in Section 3.4.1 with the exception that now the output has been replaced by the translation id class, the index of each English translation in the corpus. The training set is chosen to be the original translation and identified words dataframe with original data and random masking at each epoch and the validation set the one with the words and only a random cached masking done. No ϵ_0 token was used. There are 7119 classes, sufficient enough to also treat this as a regression problem but I only tried to do classification. The model is quite easy and consists of a word embedding layer using the transliteration vocabulary ids of size 64 and several 128 1D CNN layers, followed by a single max pool layer of filter size 3 and a big classification head of a 512 Dense+ReLU layer followed by a 0.2 Dropout and the final Dense and 7119 softmax activation. The number of convolutional 1D layers and the kernel/filter size of the convolution were the searched hyper-parameters. The search parameters were 2 or 3 CNN layers with filters of sizes 3 or 5 since a single transliteration is 1-3 in size. Annex B presents the implementation of the model.

Each model was trained for 100 epochs using the same Weighted Adam [31] and a starting learning rate of $3e-4$. The training time took 30 minutes on the same machine as above. Figure 3.9 shows the training plots for all tried models. As you can see the training accuracy is around 0.5 which is due to the random masking at each epochs of the inputs. This has a regularization role, not doing this would result in 0.9 or above training accuracy but with obvious over-fitting.

Fig. 3.9: CNN Classifier accuracy and loss



Dotted lines for training, solid for validation
Best model is with 2 layers and conv kernel 3

The model also stabilizes at epoch 50 which might suggest that the discrimination values in the data has been learned and the model might use from additional data type or perhaps be more complex.

Layer (type)	Output Shape	Param #
inputs (InputLayer)	[(None, 50)]	0
embedding (Embedding)	(None, 50, 64)	4160
conv1d_1 (Conv1D)	(None, 48, 128)	24704
conv1d_1 (Conv1D)	(None, 46, 128)	49280
max_pooling1d (MaxPooling1	(None, 15, 128)	0
flatten (Flatten)	(None, 1920)	0
dense (Dense)	(None, 512)	983552
dropout (Dropout)	(None, 512)	0
output (Dense)	(None, 7119)	3652047
Total params: 4,713,751		
Trainable params: 4,713,743		
Non-trainable params: 8		

As you can see the model has much more weights than the transformer due to the last big classification heads. The same 2 altered transliterations were also tried:

Translation: "seeing the favour towards you of the king."
Word Transliteration: mA sn Hswt nt Xr nsw
Transformer: "seeing the favour towards you of the king."
Changed Transliteration: Hswt nsw mA nt sn
Transformer: "a balance that is crooked, the pointer of a balance that errs,"
Translation: "Then Ruddjedet said:"

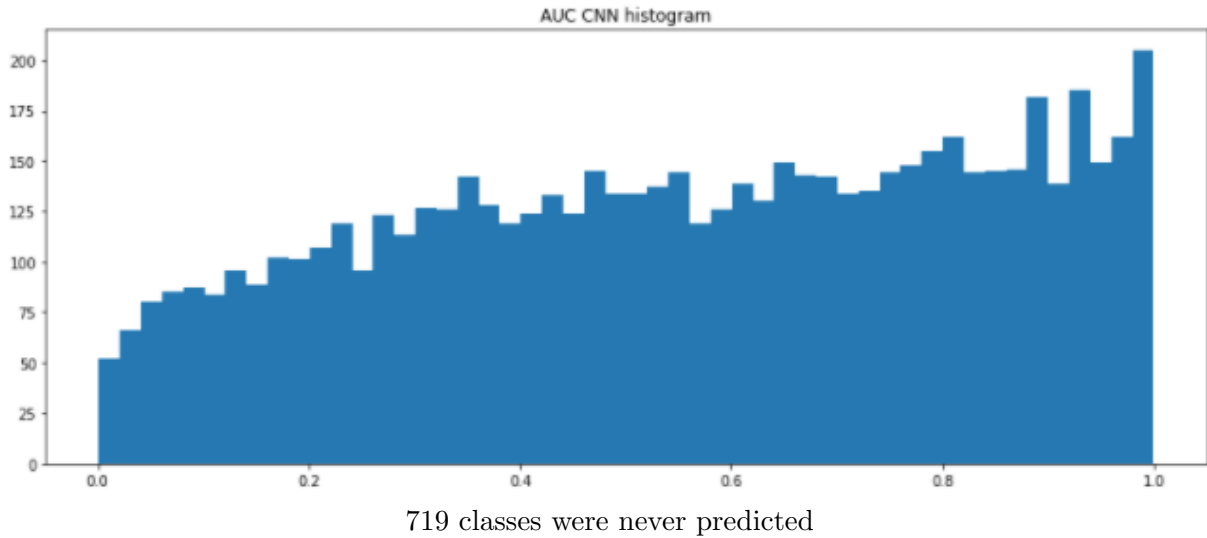
Word Transliteration: aHa .n d d .n rD Ddt
Transformer: "Then Ruddjedet said :"
Changed Transliteration: rD aHa .n d .n d
Transformer: "' and said:'"

As you can see the original word transliteration translation is preserved, since is from the training distribution, but the changed transliteration are quite different since we spill out a translation id. But in the second case the "said" word is preserved due to perhaps the importance of some grams.

The BLUE scores for the original transliteration and the word identified transliteration are 0.8501 and 0.769 respectively.

To check the model from a classification perspective an AUC class predicted histogram was plotted in Figure 3.10 for the 7119 classes. The 7119 entries of the transliteration input was masked and run 10 times to obtain the scores for the prediction.

Fig. 3.10: CNN Classifier AUC class histogram



One caveat of the prediction output is that we take **all** translations with the same transliteration as the translation identified.

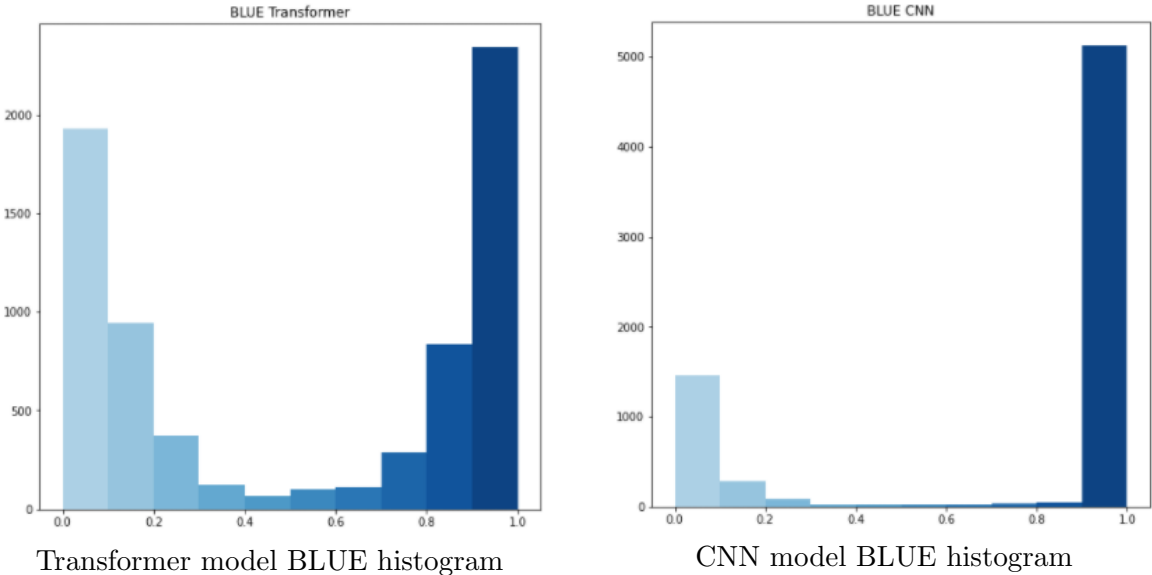
Table 3.1 presents the results of the models. The BLUE scores are the difference between the original translations and the model's output based on the original transliteration or the transliteration constructed from identified words.

Model	No weights	Train acc	Val acc	BLEU original	BLEU words
Transformer	628096	0.8894	0.5331	0.5462	0.5118
CNN	4713743	0.4927	0.4546	0.8501	0.769

Table 3.1: Translation model results

Figure 3.11 shows the histogram of the BLUE scores for each predicted translation in the validation set of the transliteration constructed from identified words. For the CNN model BLUE is either 1 (correctly predicted) or 0 (miss-classified).

Fig. 3.11: Validation BLUE histogram



Chapter 4

Building a Product

Chapter 1 presented a short introduction to the Hieroglyph language while Chapter 2 focuses on training an OCR engine to identify and recognize Unicode symbols in images. Chapter 3 describes models that given a glyph position and Unicode values tries to build an English translation of the data. This Chapter will focus on putting everything together and building an actual product.

4.1 GlyphViewer Integration

GlyphViewer is a large C++ based application, written in Qt 6, to do language translation from images with a codebase of around 3000 files. It features a plug-in based system as well as an event based architectures. To add hieroglyph functionality to the application 3 plug-in were developed:

- **Hieroglyph** plugin that adds the Hieroglyph language entity including Hieroglyph symbols with Gardiner and Unicode values and several Hieroglyph fonts for visualization.
- **Tesseract** plugin for OCR functionality containing the patches and language files required for symbol identification.
- **RemoteTranslator** plugin to implement a Translator entity that will send glyphs to a remote REST server for building translations.

The language schema implemented in the application is based on a Glyph Element which can be a glyph symbol, a word or a translation. A glyph symbol is a polygon part of an image that has assigned one or more Unicode symbols with different confidence values as in Figure C.2. A word, Figure C.1, is made up of several glyph symbols in an image and represent the translation of those symbols in one or more languages (e.g. English, French, German...). Finally, a translation represents the translation of several elements in an image, in any languages, and is composed on one or more words with their character range position in the translation as can be seen in Figure C.3.

Additionally a new REST server was implemented at <https://www.2glyph.com/translate/2> as a Python/Flask application containing 2 important POST endpoints:

- **/lookup/words** that accepts a list of English words and returns a list of Hieroglyph identified dictionary words with their Hieroglyph Unicode symbols that an user can use to build their own images.
- **/translate/glyphs** that, given a list of Hieroglyph positions and Unicode symbol values, builds one or several English translations as well as a list of identified words.

The Flask [32] based application is deployed in a uWSGI [33] container with a NGINX proxy deployed on an AWS T3.medium (2 cores, 4GB) Linux machine. Models are being run on CPU only but the inference time is quite good: 2-3 nanoseccods per inference/ per translation and about 1 second round time for a translate command with a loaded memory footprint of around 3GB. Both Transformer and CNN models are used to return a translation.

There are 2 main flows that the application implements.

The first one is the Hieroglyph dictionary lookup that allows the users to remotely search Hieroglyph words that have certain English words in their translation. On the server side the dictionary Hieroglyphs translations are tokenized using nltk library and loaded in a dictionary of `{word: [hieroglyph unicode symbols]}` so that the given words are efficiently searched. They can be drag'n'dropped in the image for building translations.



The main flow is the actual translation from an empty image. The actual steps are presented below with their corresponding GlyphViewer component:

1. Load hieroglyph image [GlyphViewer]
2. Identify text regions [OCR plugin]
3. Recognize each text region [OCR plugin]
4. Send identified glyph Unicode values to server using POST JSON [Remote Translator plugin]
5. Cluster glyphs using DBScan [Server /translate/glyphs]
6. Identify dictionary words from a cluster of symbols [Server /translate/glyphs]
7. Build transliteration from above words [Server /translate/glyphs]
8. Call models, CNN+Transformer, on the above transliteration [Server /translate/glyphs]
9. Identify English word position in a the resulted translation and return JSON [Server /translate/glyphs]
10. Build translation, word elements from JSON reply [Remote Translation plugin]

11. Show translations, words in the image [GlyphViewer]

Figure C.4 shows the translation process as can be seen on a client side. Notice the result of some text regions, while other still await to be translated.

4.2 Conclusion and Future Work

The paper presented what are the main difficulties translating hieroglyphs and how the problem was approached with the current available resources. By far the main problem is the lack of data, both in the recognition department as well as in hieroglyph translations.

The Recognition OCR engine is now only trained on font data. To be able to use it on actual temple images the bounding boxes and Glyph identification must be made for at least 5 images per symbol. Another approach would be to make custom images starting from the fonts and add them on the background that matches one in a temple. Also the pen strokes must be similar to the stone carving strokes.

The translation models achieved a BLUE score or around 0.5. This is not very good and could use more data. There might be additional resources available that were not identified for language translation. Another feature that could be added to the models might be the actual location. The main idea is that Hieroglyphs location is kind of static, especially for tourist locations. A pre-computed translations could be made for each temple wall with the application just identifying the actual location.

Finally a mobile based application should be developed. Since internet connection might not always be available all translations should be moved to the client application using the Tesseract's machine learning layers. This also means that the model size have to be small to fit mobile devices.

All in all the task of Hieroglyph translation using machine learning is a tractable problem that can be successfully implemented given time, effort and specific know how. Once this is done other specific natural language processing techniques can be run to analyze the change of the writings over time or how the language impacted other modern writings schemes. The specific beauty of the pictographic nature of the Hieroglyph writings has been lost by modern languages and display formats.

References

- [1] Mihai Matei. 2021. GlyphViewer application. (Aug. 2021). <https://www.2glyph.com>
- [2] Wikipedia. 2021a. Thomas Young. (Aug. 2021). [https://en.wikipedia.org/wiki/Thomas_Young_\(scientist\)](https://en.wikipedia.org/wiki/Thomas_Young_(scientist))
- [3] Wikipedia. 2021b. Jean-François Champollion. (Aug. 2021). https://en.wikipedia.org/wiki/Jean-Francois_Champollion
- [4] British Museum. 2021. British Museum blog. (Aug. 2021). <https://blog.britishmuseum.org/everything-you-ever-wanted-to-know-about-the-rosetta-stone/>
- [5] Jean-François Champollion. 2021. Rosetta Stone Translation. (Aug. 2021). <https://www.sacred-texts.com/egy/trs/trs07.htm>
- [6] Wikipedia. 2021. List of Egyptologists. (Aug. 2021). https://en.wikipedia.org/wiki/List_of_Egyptologists
- [7] Sir Alan Gardiner. 1927. *Egyptian Grammar: Being an Introduction to the Study of Hieroglyphs*. Clarendon Press.
- [8] Sir Alan Gardiner. 2021. Gardiner Sign List. (Aug. 2021). https://en.wikipedia.org/wiki/Gardiner's_sign_list
- [9] Unicode Standard. 2021. Unicode Egyptian Hieroglyph. (Aug. 2021). <https://www.unicode.org/charts/PDF/U13000.pdf>
- [10] Hans van den Berg. 2021. Manuel de Codage. (Aug. 2021). <http://www.catchpenny.org/codage/>
- [11] Mark Collier and Bill Manley. 2003. *How to read Egyptian hieroglyphs: A step-by-step guide to teach yourself*. University of California Press.
- [12] Bill Manley. 2012. *Egyptian hieroglyphs for complete beginners*. Thames & Hudson, London, England.
- [13] Google. 2021. Tesseract documentation. <https://tesseract-ocr.github.io/>. (Aug. 2021).

- [14] Bob Richmond. 2021a. List of Hieroglyph Fonts. <https://github.com/HieroglyphsEverywhere/Fonts/blob/master/HieroglyphicFontList.md>. (Aug. 2021).
- [15] Bob Richmond. 2021b. Aaron Font. <https://github.com/HieroglyphsEverywhere/Fonts/tree/master/Experimental>. (Aug. 2021).
- [16] Serge Rosmorduc. 2021. JSesh Font. <http://jsesh.qenherkhopeshef.org/fr/varia/unicodehieroglyphicfont>. (Aug. 2021).
- [17] Mark-Jan Nederhof. 2021. New Gardiner. <https://mjn.host.cs.st-andrews.ac.uk/egyptian/fonts/newgardiner.html>. (Aug. 2021).
- [18] Google. 2021. Noto Font. <https://github.com/googlefonts/noto-fonts>. (Aug. 2021).
- [19] Mark Vygus. 2021. Middle Egyptian Dictionary 2018. https://rhbarnhart.net/VYGUS_Dictionary_2018.pdf. (Aug. 2021).
- [20] Ray Smith. 2021. Variable-size Graph Specification Language. <https://tesseract-ocr.github.io/tessdoc/tess4/VGSLSpecs>. (Aug. 2021).
- [21] Alex Graves, Santiago Fernández, Faustino Gomez, and Jürgen Schmidhuber. 2006. Connectionist temporal classification: Labelling unsegmented sequence data with recurrent neural networks. In *Proceedings of the 23rd international conference on Machine learning - ICML '06*. ACM Press, New York, New York, USA.
- [22] Harald Scheidl, Stefan Fiel, and Robert Sablatnig. 2018. Word beam search: A connectionist temporal classification decoding algorithm. In *2018 16th International Conference on Frontiers in Handwriting Recognition (ICFHR)*. IEEE.
- [23] Ray Smith. 2021. Training tesseraact LSTM models. <https://tesseract-ocr.github.io/tessdoc/tess4/TrainingTesseract-4.00.html>. (Aug. 2021).
- [24] Google. 2021. Google ArtsCulture Fabricius. <https://artsexperiments.withgoogle.com/fabricius/en>. (Aug. 2021).
- [25] Mark-Jan Nederhof. 2021. St Andrews Corpus. <https://mjn.host.cs.st-andrews.ac.uk/egyptian/texts/corpus/pdf/>. (Aug. 2021).
- [26] Frederik Petersen. 2021. ahocorapy - Fast Many-Keyword Search in Pure Python. <https://github.com/abusix/ahocorapy>. (Aug. 2021).
- [27] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. 2002. BLEU: A Method for Automatic Evaluation of Machine Translation. In *Proceedings of the 40th Annual Meeting on Association for Computational Linguistics (ACL '02)*. Association for Computational Linguistics, USA, 311–318. DOI:<http://dx.doi.org/10.3115/1073083.1073135>

- [28] Mihai Matei. 2021. matmih - Machine learning library. <https://github.com/glypher/matmih.git>. (Aug. 2021).
- [29] Google. 2021. TensorFlow Official Models. <https://github.com/tensorflow/models/tree/master/official>. (Aug. 2021).
- [30] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. 2017. Attention Is All You Need. (2017).
- [31] Ilya Loshchilov and Frank Hutter. 2019. Decoupled Weight Decay Regularization. (2019).
- [32] Pallets organization. 2021. Flask web server. <https://github.com/pallets/flask>. (Aug. 2021).
- [33] Open Source. 2021. Micro Web Server Gateway Interface. <https://uwsgi-docs.readthedocs.io/en/latest/>. (Aug. 2021).

Appendices

Appendix A

Tesseract Training

Listing A.1: "Train Tesseract for Hieroglyphs scripts"

```
$ cd ~; mkdir train; cd train; mkdir data
# copy tiff/box files to this folder from HieGenerator/data generated image directory

# Current language data can be found at
#   https://github.com/tesseract-ocr/langdata_lstm
#   https://github.com/tesseract-ocr/tessdata_best
$ mkdir -p langdata/hie
$ wget https://github.com/tesseract-ocr/langdata_lstm/raw/master/chi_sim/chi_sim.config
# Set the hieroglyph language OCR configuration from Chinese settings
$ mv chi_sim.config hie/hie.config
# The following are needed to get tesseract basic segmentation working
$ cp -R ../tesseract-4.1.1/tessdata/ .
$ wget https://github.com/tesseract-ocr/langdata/raw/master/radical-stroke.txt \
    -O langdata/radical-stroke.txt
$ wget https://github.com/tesseract-ocr/langdata/raw/master/Latin.unicharse \
    -O langdata/Latin.unicharset

# Generate additional font images
# Add langdata/hie/hie.text file to contain all the Unicode word symbol values
# from the dictionary, each word separated by space
$ cd data
$ text2image --fonts_dir=~/.local/share/fonts --strip_unrenderable_words --char_spacing=1 \
    --exposure=0 --font "Noto Sans Egyptian Hieroglyphs" --ptsize=16 \
    --outputbase=hie_noto_t2i --text=../langdata/hie/hie.text
$ text2image --fonts_dir=~/.local/share/fonts --strip_unrenderable_words --char_spacing=1 \
    --exposure=0 --font "NewGardiner Medium" --ptsize=16 \
    --outputbase=hie_gard_t2i --text=../langdata/hie/hie.text
$ text2image --fonts_dir=~/.local/share/fonts --strip_unrenderable_words --char_spacing=1 \
    --exposure=0 --font "Aaron UMdC Alpha" --ptsize=16 \
    --outputbase=hie_aaron_t2i --text=../langdata/hie/hie.text
$ text2image --fonts_dir=~/.local/share/fonts --strip_unrenderable_words --char_spacing=1 \
    --exposure=0 --font "Aaron Basic RTL Alpha" --ptsize=16 \
    --outputbase=hie_aaron_rtl_t2i --text=../langdata/hie/hie.text
$ text2image --fonts_dir=~/.local/share/fonts --strip_unrenderable_words --char_spacing=1 \
    --exposure=0 --font "JSesh font" --ptsize=16 \
    --outputbase=hie_jsesh_t2i --text=../langdata/hie/hie.text
$ cd ..

# Generate unicharset
$ box_files=$(ls data/*.box)
$ unicharset_extractor --output_unicharset langdata/hie/hie.unicharset --norm_mode 3 ${box_files}

# Generate lstm-recorder
$ rm -rf out; mkdir out
$ combine_lang_model --lang hie --input_unicharset langdata/hie/hie.unicharset
    --script_dir ./langdata --output_dir ./out
```

```

$ mv out/hie/hie.traineddata tessdata/hie.traineddata
# The following should output the config, lstm-unicharset and lstm-recorder entries
$ combine.tessdata -d tessdata/hie.traineddata

# Tesseract LSTM model training needs tif+box files converted to lstm files
# This will take a long time this is why I am doing it in parallel
$ export TESSDATA_PREFIX='pwd'/tessdata
$ mkdir -r logs/data ; rm logs/data/*.log ; rm data/*.lstmf
$ img_files=$(ls data/*.tif*)
$ for img_file in ${img_files}; do
$     tesseract -l hie ${img_file} ${img_file%.*} --psm 6
$     lstm.train langdata/hie/hie.config > logs/${img_file%.*}.lstmf.log 2>&1 &
$     sleep 80
$ done
# Check for errors and create training and evaluation file list
$ grep -r -i error logs/data
$ ls data/*.lstmf | grep -v "^data/test" > data/lstmf_file_list.txt
$ ls data/test*.lstmf > data/lstmf_file_list_eval.txt

# Start the actual training process...
$ time lstmtraining \
  --traineddata tessdata/hie.traineddata \
  --net.spec '[1,0,0,1 Ct3,3,32 Mp3,3 Lfys64 Lfx128 Lrx128 Lfx512 Olc1073]' \
  --model-output train-checkpoint/hie --learning-rate 20e-4 \
  --train-listfile data/lstmf_file_list.txt \
  --eval-listfile data/lstmf_file_list_eval.txt \
  --max-iterations 120000 > train.log 2>&1

# RETRIEVE the results once it is done. Check network output...
$ tail -f train.log
# Optional: Continue training to epoch 200000
$ time lstmtraining \
  --traineddata hie.traineddata \
  --continue-from hie.lstm \
  --model-output train-checkpoint/hie --learning-rate 5e-4 \
  --train-listfile data/lstmf_file_list.txt \
  --eval-listfile data/lstmf_file_list_eval.txt \
  --max-iterations 200000 >> train.log 2>&1

# Save result as hie/traineddata when done
$ lstmtraining --stop-training \
  --continue-from train-checkpoint/hie-checkpoint \
  --traineddata tessdata/hie.traineddata \
  --model-output hie.traineddata
# This should output config, lstm, lstm-unicharset, lstm-recorder
$ combine.tessdata -d hie.traineddata

# To do transfer learning from Chinese
# Get LSTM network from chinese
$ wget https://github.com/tesseract-ocr/tessdata_best/raw/master/chi_sim.traineddata \
  -O tessdata/chi_sim.traineddata
$ combine.tessdata -e tessdata/chi_sim.traineddata chi_sim.lstm
$ mv chi_sim.lstm langdata/hie/hie.lstm
# Create hie traineddata - this will contain the lstm network for Chinese
$ combine.tessdata -o tessdata/hie.traineddata langdata/hie/hie.lstm
# Replace the last fully connected layer and softmax classification
$ time lstmtraining \
  --traineddata tessdata/hie.traineddata \
  --continue-from langdata/hie/hie.lstm \
  --append-index 5 --net.spec '[Lfx256 Olc1073]' \
  --model-output train-checkpoint/hie --learning-rate 20e-4 \
  --train-listfile data/lstmf_file_list.txt \
  --eval-listfile data/lstmf_file_list_eval.txt \
  --max-iterations 30000 > train.log 2>&1

# Use the same method to stop training and save to hie.traineddata
# Use this language data OCR file to add to GlyphViewer hieroglyph plug-in

```

Appendix B

Translation Modelling

Listing B.1: "Tensorflow input dataset for masked transliterations"

```
class MaskedDataset(tf.data.Dataset):

    def __new__(cls, ds, batch_size=BATCH_SIZE, pad_token_id=PAD_TOKEN, mask=True):

        ds = ds.filter(lambda t: tf.reduce_sum(t['input_mask']) > 0)
        ds = ds.filter(lambda t: tf.reduce_sum(t['output_mask']) > 0)

        @tf.function
        def _mask_data(data):
            inputs = data['input_ids']
            input_mask = data['input_mask']

            encoded = tf.identity(inputs)
            encoded_mask = tf.identity(input_mask)

            text_length = tf.reduce_sum(input_mask) - 1

            if text_length > 0:
                # 25% masking
                mask_length = tf.cast((tf.cast(text_length, tf.float32) * 0.25), tf.int32)

                mask_length = mask_length if mask_length > 1 else 1
                mask_index = tf.random.uniform(shape=[mask_length], minval=0, maxval=text_length, dtype=tf.int32)
                mask_index = tf.reshape(mask_index, [mask_length, 1])

                # mask the words
                mask = tf.ones_like(inputs)
                zeros = tf.zeros([mask_length], dtype=tf.int32)
                mask = tf.tensor_scatter_nd_update(mask, mask_index, zeros)

                encoded = tf.boolean_mask(encoded, mask)

                pad = tf.ones([mask_length], dtype=tf.int32) * pad_token_id
                encoded = tf.concat([encoded, pad], 0)
                encoded = encoded[0 : inputs.shape[0]]

                encoded_mask = tf.boolean_mask(encoded_mask, mask)

                pad = tf.zeros([mask_length], dtype=tf.int32)
                encoded_mask = tf.concat([encoded_mask, pad], 0)
                encoded_mask = encoded_mask[0 : input_mask.shape[0]]

            return {'inputs': encoded, 'input_mask': encoded_mask,
                    'targets': data['output_ids'], 'sample_weights': data['output_mask']}

        @tf.function
```

```

def _identity(data):
    return {'inputs': data['input_ids'], 'input_mask': data['input_mask'],
            'targets': data['output_ids'], 'sample_weights': data['output_mask']}

# since cache() is not used this will be remasked each time on a new epoch
ds_i = ds.map(_identity)
if mask:
    ds_r = ds.map(_mask_data)
    return ds_i.concatenate(ds_r).shuffle(1000).batch(batch_size)

return ds_i.shuffle(1000).batch(batch_size)

```

Listing B.2: "Hieroglyph transformer model"

```

from official.nlp.modeling.models import seq2seq_transformer

class LanguageModel(seq2seq_transformer.Seq2SeqTransformer):
    def __init__(self, **hyper_params):
        encoder_params = {
            'num_layers': hyper_params['num_layers'],
            'num_attention_heads': hyper_params['num_attention_heads'],
            'intermediate_size': hyper_params['intermediate_size']
        }

        super(LanguageModel, self).__init__(
            encoder_layer = seq2seq_transformer.TransformerEncoder(**encoder_params),
            decoder_layer = seq2seq_transformer.TransformerDecoder(**encoder_params),

            vocab_size = hyper_params['vocab_size'] + 1,
            embedding_width = hyper_params['embedding_width'],
            decode_max_length = hyper_params['decode_max_length'],
            beam_size = 4,
            alpha = 0.6,

            padded_decode = True,
            eos_id = hyper_params['eos_id']
        )

        self._loss = tf.keras.losses.SparseCategoricalCrossentropy(from_logits = True,
                                                                    reduction=tf.keras.losses.Reduction.NONE)
        self._metrics = [tf.keras.metrics.Mean(name="loss"),
                         tf.keras.metrics.Accuracy(),
                         tf.keras.metrics.Mean(name="val_loss"),
                         tf.keras.metrics.Accuracy(name="val_accuracy")]

    def train_step(self, inputs):
        targets = inputs['targets']
        sample_weights = inputs['sample_weights']

        with tf.GradientTape() as tape:
            predictions = self(inputs)
            loss = self._loss(targets, predictions, sample_weight=sample_weights)

        trainable_vars = self.trainable_variables
        gradients = tape.gradient(loss, trainable_vars)
        self.optimizer.apply_gradients(zip(gradients, trainable_vars))

        self._metrics[0].update_state(loss, sample_weight=sample_weights)
        self._metrics[1].update_state(targets, tf.math.argmax(predictions, axis=2),
                                      sample_weight=sample_weights)

        return {"loss": self._metrics[0].result(), "accuracy": self._metrics[1].result()}

    def test_step(self, inputs):
        targets = inputs['targets']
        sample_weights = inputs['sample_weights']

```

```

predictions = self({ 'inputs': inputs[ 'inputs' ]})

self._metrics[2].update_state(-predictions[ 'scores' ])
self._metrics[3].update_state(targets, predictions[ 'outputs' ],
                              sample_weight=sample_weights)

return { "loss": self._metrics[2].result(), "accuracy": self._metrics[3].result() }

@property
def metrics(self):
    return self._metrics

```

Listing B.3: "Hieroglyph transformer model"

```

class HieCnnModel(TrainModel):
    def __init__(self, **hyper_params):
        self._hyper_params = hyper_params.copy()

        inputs = tf.keras.Input(name='inputs', shape=(hyper_params[ 'length' ]), dtype=tf.int32)

        layer = tf.keras.layers.Embedding(hyper_params[ 'vocab_size' ] + 1, hyper_params[ 'embedding_size' ],
                                           input_length=hyper_params[ 'length' ])(inputs)

        for _ in range(hyper_params[ 'conv_layers' ]):
            layer = tf.keras.layers.Conv1D(hyper_params[ 'conv_size' ], hyper_params[ 'conv_kernel' ], act

        layer = tf.keras.layers.MaxPooling1D(hyper_params[ 'maxpool_size' ])(layer)

        layer = tf.keras.layers.Flatten()(layer)
        layer = tf.keras.layers.Dense(512, activation='relu')(layer)
        layer = tf.keras.layers.Dropout(0.2)(layer)
        outputs = tf.keras.layers.Dense(hyper_params[ 'no_classes' ], activation='softmax', name='output

super(HieCnnModel, self).__init__({ 'inputs': inputs }, outputs)

```


Appendix C

GlyphViewer

Fig. C.1: GlyphViewer Word View

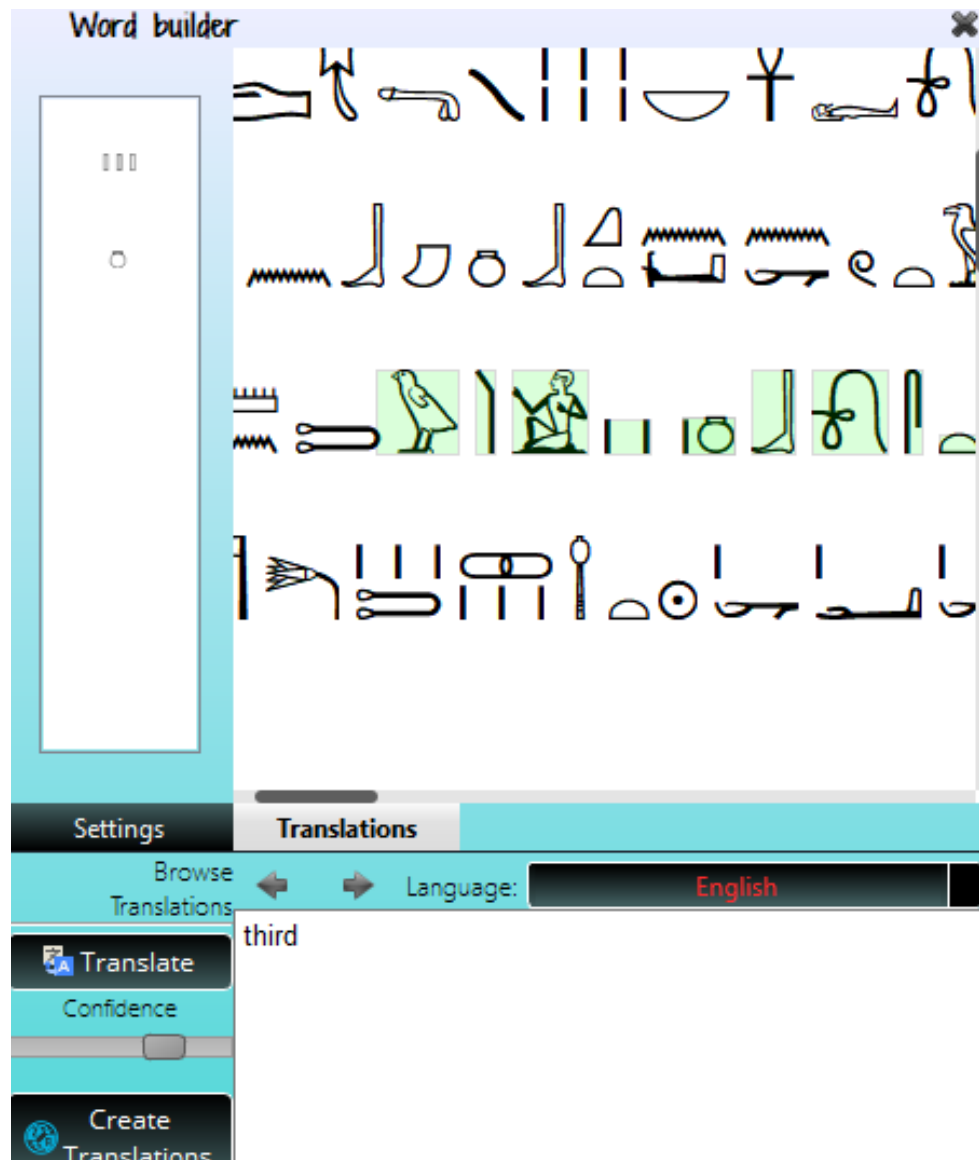


Fig. C.2: GlyphViewer Symbol View

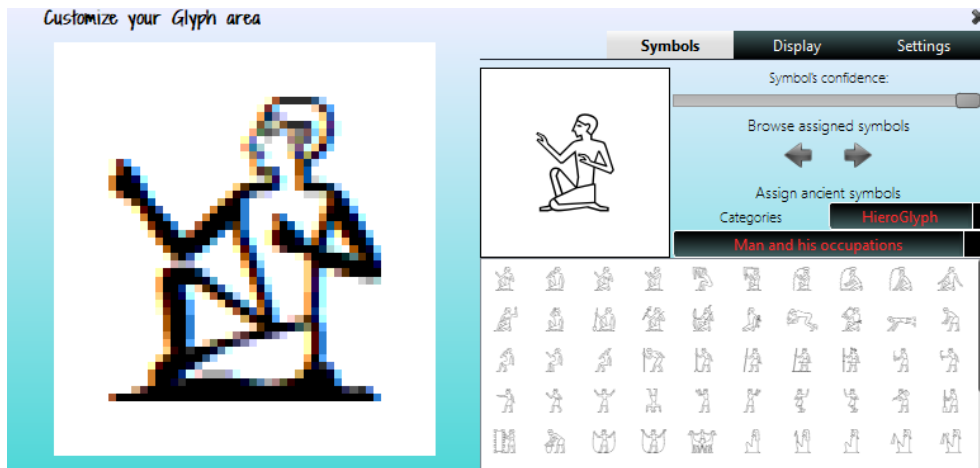


Fig. C.3: GlyphViewer Translation View

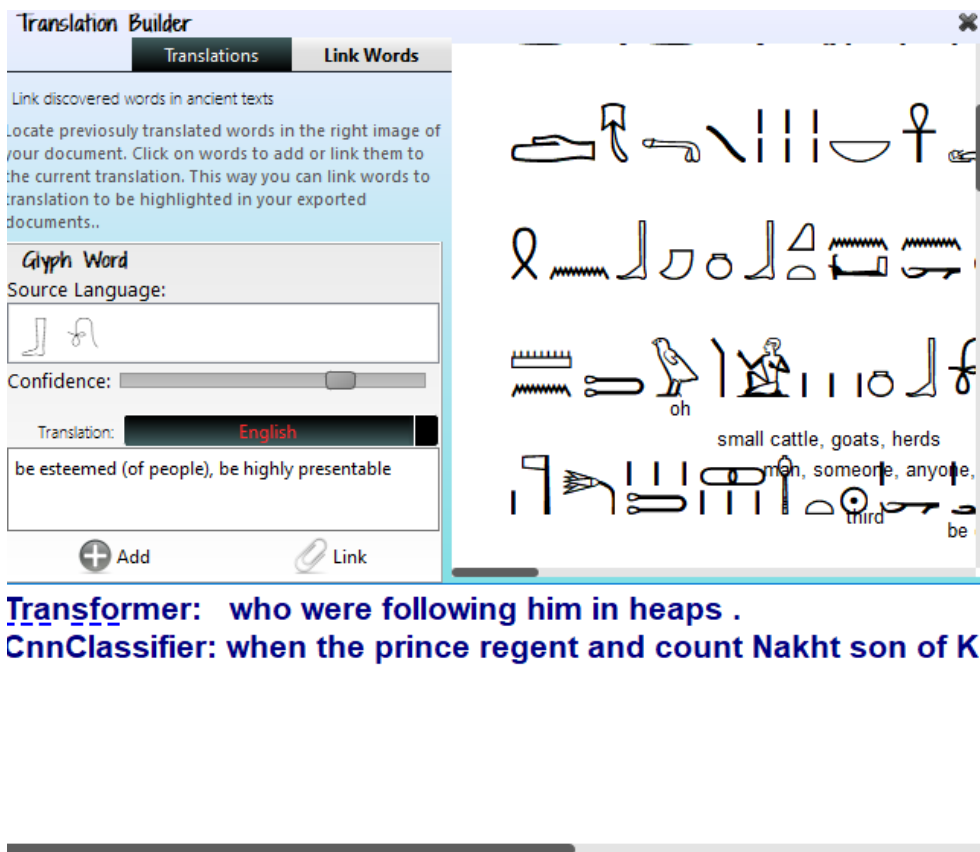


Fig. C.4: GlyphViewer Translation Processing

